Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Proyecto Fin de Carrera
Ingeniería en Informática

Medición precisa del consumo energético en procesadores x86

Víctor Manuel Malo Oiz

Director: Darío Suárez Gracia

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Curso 2016/2017
Febrero 2017

Medición precisa del consumo energético en procesadores x86
Resumen

Históricamente, el criterio más importante en la compilación de código fuente ha sido la eficiencia del ejecutable obtenido, buscando una optimización temporal (reducir el tiempo de ejecución). Otro criterio importante ha sido la optimización espacial (ya sea reduciendo la cantidad de memoria que ocupa el programa al ejecutarse o el tamaño del propio programa en sí mismo).

Con el auge de los dispositivos portátiles, la optimización energética también puede ser muy importante: el consumo energético de un programa es crucial de cara a la autonomía de la batería.

Este proyecto parte de una plataforma desarrollada en anteriores proyectos finales de carrera que obtiene muestras del consumo energético de un programa. Esto permite saber los datos de consumo energético del programa ejecutado a través de la plataforma, pero tiene una limitación: es imposible saber con qué parte del programa está relacionada cada muestra de consumo obtenido.

Para superar esta limitación, se ha desarrollado un proceso por el cual se obtiene la caracterización temporal y energética de dicho programa a nivel de función. Al finalizar el proceso se obtendrá el consumo energético y temporal de las principales funciones del programa, pudiéndose asociar las variaciones de consumo energético a las funciones responsables de dichas variaciones.

Para conseguir estos resultados, se ha desarrollado un mecanismo de instrumentación con el que poder obtener los datos temporales del programa minimizando la sobrecarga. Dichos datos son los instantes en los comienza y termina cada función instrumentada. Así, se puede saber en cada momento qué función está siendo ejecutada (y es la responsable del consumo energético).

En este documento se describirá detalladamente el proceso diseñado, que puede ser aplicado en el futuro a otros programas de una manera sencilla y automatizada, obteniendo resultados que podrán profundizar tanto como la plataforma utilizada sea capaz.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo se describen los motivos por los que un proyecto de estas características puede ser necesario, así como sus antecedentes y objetivos. Finalmente, se desglosan el resto de puntos que aparecerán en la memoria.

1.1. Motivación

En los últimos años, la preocupación por el consumo de energía en los microprocesadores ha pasado a ser un tema muy relevante. Además de que el número de dispositivos portátiles ha aumentado exponencialmente, dichos dispositivos son cada vez más potentes, provocando un mayor consumo energético, lo que conlleva un menor tiempo de descarga de las baterías.

Pero dicho consumo energético no sólo es importante para los dispositivos portátiles, el aumento en el respeto al medio ambiente lleva a intentar no consumir energía innecesaria. Por ello, saber que secuencias de instrucciones son más eficientes energéticamente puede ser un conocimiento muy útil a la hora de la la compilación e incluso de la implementación del código.

1.2. Desarrollo

Este Proyecto de Fin de Carrera como tal se comenzó en el año 2009, pero por motivos laborales fue interrumpido en el año 2012 a pesar de estar finalizado prácticamente (a falta de la memoria). Ha sido retomado este curso 2016-2017 para su completa finalización. Los scripts en Python han sido actualizados (de Python 2.7 a Python 3.5) y mejorados para un análisis más exhaustivo de los datos obtenidos, pero la práctica totalidad de los experimentos se realizó hasta el año 2012.

1.3. Trabajos previos

La plataforma de medición del consumo energético que será descrita en el siguiente capítulo fue puesta en marcha para el Proyecto de Fin de Carrera de Alicia Asín Pérez [16] en el año 2006 y posteriormente mejorada para el Proyecto de Fin de Carrera de Octavio Benedí [9]. Posteriormente, el proyecto de Fin de Carrera de Sergio Gutiérrez Verde [18] añadió la posibilidad de medir también la temperatura, pero ese apartado no ha sido considerado para el presente Proyecto de Fin de Carrera.

1.4. Objetivos

En los trabajos previos se había obtenido la caracterización energética de la ejecución de un programa en su totalidad, es decir, se había obtenido la potencia instantánea y la energía de la ejecución de dicho programa.

Al ejecutar un programa a través de la plataforma, se toman sucesivas muestras del consumo energético instantáneo cada 50 microsegundos, volcándose al final esa información en un fichero de texto para poder ser analizada a posteriori.

La mayor limitación de la plataforma era la imposibilidad de determinar el origen de las muestras de potencia. En otras palabras, no se podía asociar las instrucciones ejecutadas con la potencia consumida, correspondiendo las métricas obtenidas a la granularidad de "programa". El objetivo de este proyecto es poder asociar dicha información y determinar cuál es la secuencia de instrucciones que causa un cambio en la potencia, todo ello sin afectar a la medición del consumo.

Al reducir la granularidad de la medida, se obtiene información a más detalle, pudiendo extraer conclusiones más definitivas acerca del consumo energético y las causas de su variación en cada momento.

Un programa puede estar compuesto por múltiples funciones (subrutinas) que a su vez pueden estar compuestas por múltiples instrucciones. Así pues, el objetivo ideal de un proyecto de estas características sería obtener la caracterización energética de cada instrucción ejecutada en el programa. Pero para eso sería necesaria una plataforma de medición de consumo mucho más rápida de la que se dispone, con lo que este el objetivo de este Proyecto de Fin de Carrera ha sido reducir la granularidad desde "programa" hasta "función".

De acuerdo a las restricciones que se detallarán en apartados posteriores, se verá el proceso implementado para conseguir una caracterización energética a nivel de función.

1.5. Trabajo realizado

La caracterización energética a nivel de función se ha realizado identificando qué función está siendo ejecutada en cada momento y, posteriormente, asignando el consumo energético correspondiente al tiempo que dicha función ha estado en ejecución.

En esta serie de Proyectos de Fin de Carrera en los que se ha medido el consumo energético, se ha trabajado sobre una plataforma modificada al efecto en la que los factores que pueden producir una sobrecarga en el sistema están reducidos al máximo. A diferencia de los proyectos anteriores, en este proyecto ha sido necesario "contaminar" el archivo binario: para determinar el marco temporal de una función es necesario conocer los momentos en los que dicha función comienza y termina su ejecución, y esto sólo se puede llevar a cabo añadiendo instrucciones al código fuente original (en este caso, se ha añadido el menor número posible de instrucciones para minimizar la sobrecarga).

1.6. Estructura de la memoria

El presente documento se divide en los siguientes capítulos. El capítulo 2 analiza el estado del arte de la plataforma utilizada, la instrumentación energética y la sincronización temporal. El capítulo 3 presenta la metodología y el capítulo 4 la solución propuesta. El capítulo 5 analiza los resultados obtenidos aplicando dicha metodología a un benchmark elegido. Finalmente, el capítulo 6 concluye esta memoria.

Respecto a los anexos. El anexo A contiene el código fuente del programa utilizado para las pruebas unitarias del sistema de instrumentación. El anexo B contiene el código fuente desarrollado para llevar a cabo el sistema de instrumentación. El anexo C contiene el script Python desarrollado para obtener las funciones de un código fuente determinado. El anexo D contiene el script Python desarrollado para validar estadísticamente las muestras de consumo obtenidas. El anexo E contiene los resultados obtenidos al ejecutar Intel PTU para el benchmark instrumentado en este proyecto (BZIP2). Para finalizar, el anexo F muestra los resultados intermedios del proceso total de instrumentación aplicado a BZIP2.

Capítulo 2

Estado del arte

En este capítulo se describe la plataforma sobre la que se han realizado los experimentos. También se muestran las distintas opciones existentes para instrumentar, así como para establecer un registro temporal de los eventos de un programa. Finalmente, se señalan las opciones elegidas, cuyo uso será detallado en capítulos posteriores.

2.1. Descripción de la plataforma utilizada

2.1.1. Origen

La plataforma utilizada en el desarrollo de este proyecto partió de la desarrollada en los proyectos de Alicia Asín Pérez [16], Octavio Benedí Sánchez [9] y Sergio Gutierrez Verde [18]. Esta plataforma consta de dos equipos, uno ejecutando el programa de prueba y y otro que almacena las muestras obtenidas. El uso de 2 máquinas permitía originalmente que el proceso de medida no afectara al programa analizado. La plataforma puede medir tres magnitudes, el voltaje, la intensidad y la temperatura, en función del tiempo. Para este Proyecto, la monitorización de temperatura no se ha utilizado.

2.1.2. Funcionamiento

Las medidas se toman con un amplificador Tektronix TCP-300 y una sonda de corriente Tektronix TCP-312 para medir la corriente. El equipo cuenta con una tarjeta de adquisición de datos para monitorizar la tensión y la corriente. Ambos equipos tienen Linux como sistema operativo.

La plataforma es capaz de tomar aproximadamente 20.000 muestras por segundo. Al terminar la ejecución de un programa a través de la plataforma,

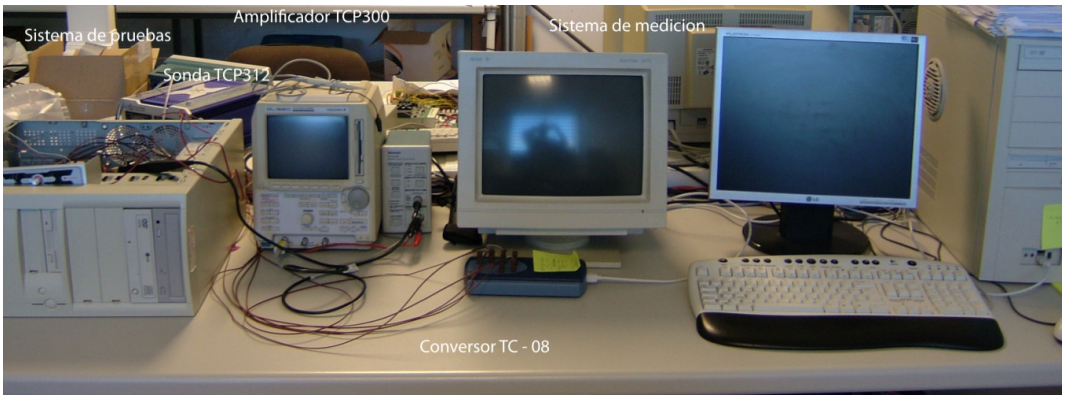

Figura 2.1: Plataforma utilizada

se obtiene un fichero de texto con las muestras de consumo obtenidas, que podrán utilizarse para calcular el consumo mediante técnicas de integración numérica. Cada muestra es una línea del fichero de texto, compuesta de 2 columnas: la primera columna se relaciona con el voltaje y la segunda con la intensidad. Así pues, el valor del consumo para ese muestra será el resultado de multiplicar ambas columnas por el factor de ajuste utilizado en el amplificador (si se ha utilizado algún valor) y por un factor constante igual a 2. Este valor constante debe ser usado porque el voltaje a la entrada del regulador de voltaje del procesador es de 12V, pero el voltaje máximo de la tarjeta de adquisición de datos es de 10V. Por lo tanto, hay que dividir el valor entre 2 para que se ajuste al rango posible de entrada, lo que provoca que haya que realizar la operación inversa al tratar los datos.

2.2. Instrumentación del código

2.2.1. Definición

La instrumentación en el ámbito de la informática es un caso particular de la instrumentación industrial. Se conoce como instrumentación industrial al conjunto de herramientas que permiten realizar la medición, conversión, control o transmisión de las variables de un cierto proceso, permitiendo lograr la optimización de los recursos empleados. Así pues, en informática la instrumentación es una técnica utilizada para el análisis del código fuente.

La instrumentación utilizada para este proyecto podría definirse como la "aplicación de una herramienta a la ejecución de un programa para obtener datos con los que poder realizar un posterior análisis del código fuente de dicho programa".

La instrumentación puede resultar en un aumento desmesurado del tiempo de ejecución de un programa, por lo que está limitada a determinados contextos. Además, en el caso particular de este proyecto y puesto que se quiere analizar el consumo energético, se ha intentado conseguir que la diferencia entre una ejecución instrumentada y otra no instrumentada sea reducida al máximo para no afectar a los resultados, como podrá verse en el capítulo 5.

Hay 2 grandes bloques de técnicas de instrumentación (cada uno de ellos contiene a su vez diversos subtipos) según si la instrumentación ocurre en tiempo de ejecución (instrumentación dinámica) o en tiempo de compilación (instrumentación estática).

2.2.2. Instrumentación dinámica

Fue la primera opción analizada para el desarrollo de este proyecto. Tras analizar y estudiar las distintas opciones en el mercado [13], el sistema probado fue PinTool (actualmente llamado Intel Pin [6])

Con la instrumentación dinámica no se conoce de antemano qué rutinas serán analizadas ya que eso se establece en tiempo de ejecución. Por lo tanto, no permite saber a ciencia cierta la sobrecarga inducida por el propio sistema de instrumentación. Debido a este motivo, se descartó este método de instrumentación. Además, la sobrecarga de la instrumentación dinámica suele ser mayor que la de la instrumentación estática.

2.2.3. Instrumentación estática

Hay diversos tipos de instrumentación estática (re-escritura de bibliotecas, instrumentación "in place", instrumentación del lenguaje intermedio, instrumentación en tiempo de compilación, instrumentación del código fuente, instrumentación en tiempo de enlazado).

La instrumentación estática utilizada en la realización de este proyecto entraría en el tipo de instrumentación del código fuente. Mediante esta técnica, se añaden las instrucciones extra de código deseadas para realizar la función necesaria (en el caso que ocupa, establecer el marco temporal de llamadas a las funciones). Estas sentencias extra se añaden condicionalmente, por lo que la instrumentación se puede activar o desactivar en tiempo de compilación.

En este caso se decidió utilizar la opción `-finstrument-functions` del compilador GCC [2] para añadir código tanto en el prólogo como en el epílogo de las funciones a instrumentar.

2.3. Registro temporal de eventos

Para establecer el marco de referencia en el que situar las llamadas a las distintas funciones, es necesario guardar un registro temporal de las invocaciones/retornos de cada una de ellas. Se barajaron varias opciones con registros existentes en el sistema, siendo el elegido el registro Time Stamp Counter [14].

2.3.1. Algunos temporizadores

Se barajó el uso de algunos temporizadores para poder fijar el marco de referencia sobre el que realizar los cálculos, pero se descartaron al comprobarse que su uso incidía en una mayor sobrecarga del sistema. Algunos ejemplos de ello son HPET (High Precision Event Timer [4] [17]), PIT (Programmable Interval Timer [8] [15]) o RTC (Real-Time Clock Timer [10]).

2.3.2. Funciones de librería estándar

También se evaluó el uso de la función `clock_gettime` de la biblioteca estándar de funciones de C pero se desechó porque su sobrecarga también era mayor.

2.3.3. Time Stamp Counter

Es un registro de 64 bits presente en todos los procesadores x86 desde Pentium (año 1993). Cada ciclo de reloj, se incrementa en una unidad su valor, reseteándose ese valor al reiniciar el procesador. Intel provee la instrucción RDTSC para leer el valor de este registro. Este registro fue el utilizado porque utilizando dicha instrucción (en código ensamblado embebido en el código C utilizado), el acceso al registro es muy rápido (menos de 100 ciclos de reloj) lo que provoca una sobrecarga temporal muy baja.

2.4. Resumen

Así pues, para este proyecto se ha utilizado instrumentación estática en tiempo de compilación sobre código fuente, utilizando la instrucción de compilación `-finstrument-functions` tras elegir qué funciones se quieren medir. Para este proyecto en particular se ha utilizado el compilador GCC, pero la opción `-finstrument-functions` está disponible para otros compiladores como ICC [5] o Clang/LLVM [1].

Así mismo, se ha utilizado el registro "Time Stamp Counter" para obtener el marco temporal de las llamadas.

El nivel de sobrecarga inducido por estas elecciones se podrá ver en el capítulo 5.

Capítulo 3

Metodología

En este capítulo se describen las herramientas utilizadas en la realización del proyecto, ya sean externas o realizadas *ad hoc*. Además, se describen los programas de prueba utilizados, bien sea para analizar y mejorar la sobrecarga inducida por el sistema de instrumentación o para realizar las pruebas completas del proceso.

3.1. Herramientas

3.1.1. Externas

Por motivos que se podrán ver más adelante en el capítulo 4, hay que realizar un primer análisis de las funciones del programa elegido. Para realizar este análisis, se ha utilizado Intel PTU (anteriormente llamado Intel VTune y en la actualidad llamado Intel VTune Amplifier).

Como se ha visto en el capítulo 2, tanto el sistema de pruebas como el sistema de medición tienen Linux como sistema operativo. El sistema de pruebas tiene un Linux modificado *ex professo* por Octavio Benedí en su Proyecto de Fin de Carrera [9] para que el número de interrupciones de sistema sea mínimo y la ejecución de un programa sea lo más "pura" posible. En el sistema de medición había un Linux openSUSE que se actualizó a la versión 11.1.

Para generar los binarios instrumentados, se ha utilizado el compilador GCC.

Para el desensamblado del código se ha utilizado el comando `objdump`, perteneciente a GNU Binutils [3].

3.1.2. Desarrolladas

Utilizando el registro Time Stamp Counter se ha desarrollado un fichero de instrumentación (en C, anexo B) que se compila junto con el programa a instrumentar, y que causará que al terminar la ejecución se disponga de un fichero con la información temporal de las funciones de dicho programa.

Para obtener el listado de funciones del programa junto con sus direcciones de memoria se ha desarrollado un script en Python (anexo C) que utiliza como entrada el archivo de texto que contiene el código desensamblado.

Una vez que se eligen las funciones a instrumentar, se compila el programa utilizando `-finstrument-functions`. El binario obtenido se ejecuta utilizando la plataforma, dando como resultado un fichero de texto con los consumos y otro fichero de texto con los tiempos de las funciones.

Para validar las muestras de consumo se ha desarrollado un script en Python (anexo D). Para obtener los resultados finales (utilizando ambos ficheros) se ha desarrollado un último script en Python (anexo E).

3.2. Programas de prueba

Primero se realizaron unas pruebas para validar el sistema de instrumentación, utilizando un programa pequeño (Gauss-Jordan) con funciones de corta duración. Una vez validado el sistema y diseñado el proceso completo por el que se puede instrumentar un programa, se utilizaron diversos benchmarks del SPEC CPU2006, haciendo una prueba de concepto con el benchmark BZIP2 para realizar la validación de la metodología.

3.2.1. Gauss-Jordan

Para realizar las pruebas de validación del sistema de instrumentación se utilizó un código de inversión de matrices mediante el método de Gauss-Jordan, que puede verse en el anexo A. Este código se modificó partiendo de una versión de Jesús Alastruey y Pablo Ibáñez.

Este código fuente está modularizado en funciones, para que la granularidad sea lo más parecida posible a las pruebas finales realizadas posteriormente a través de la plataforma. Además, se utilizó un tamaño de matriz lo suficientemente grande (1024x1024) para que la duración de cada función no fuera demasiado pequeña.

3.2.2. SPEC CPU2006

SPEC es un estándar de facto, utilizado ampliamente tanto en la industria como en ámbitos académicos. Para la elección del código a instrumentar, se ha utilizado el benchmark SPEC CPU2006 [7], que proporciona multitud de códigos fuente en C y C++.

Antes de realizar la prueba final con BZIP2, y analizando los resultados que pueden verse en [19], se realizaron pruebas con otros benchmarks incluidos en SPEC CPU2006 como Gromacs, Povray o GCC. Algunos de estos benchmarks fueron descartados por que su distribución temporal de funciones no era apropiada para ser instrumentada (tenían una función que acaparaba la mayor parte del tiempo de ejecución del programa). Otros benchmarks no fueron considerados por los motivos que podrán verse en el apartado 4.1.3.

Capítulo 4

Solución propuesta

En este capítulo se detallarán las acciones llevadas a cabo para la realización de este proyecto. Por un lado, las distintas limitaciones que hubo que superar durante el proceso de análisis e incluso ejecución y por el otro, se detalla el proceso integral propuesto para la instrumentación de un programa dado.

4.1. Limitaciones superadas

Una vez elegidos los sistemas de instrumentación (`-finstrument-functions`) y establecimiento del marco temporal (Time Stamp Counter), hubo que superar las limitaciones de la plataforma utilizada y ambos sistemas para poder obtener resultados válidos.

4.1.1. Temporales

La plataforma en la que se ha realizado este proyecto tiene una frecuencia de muestreo de 20.000Hz, es decir, toma 20.000 muestras de consumo por segundo. Esta cifra, que parece muy elevada, palidece en comparación con los 2.8GHz del procesador del equipo utilizado; transcurren unos 140.000 ciclos de reloj entre cada muestra de consumo obtenida. Asumiendo un valor de Instrucciones por Ciclo igual a 1, supone que una muestra de energía se corresponde a 140.000 instrucciones. Esto provocó el cambio en la granularidad que se explicará en el apartado 4.4.1.

4.1.2. De almacenamiento

Para minimizar la sobrecarga, los datos resultantes de la instrumentación (causados al contaminar el binario) que se van obteniendo durante la ejecución

del programa se guardan en memoria RAM en el equipo de pruebas. Los datos generados con el consumo energético se guardan en el equipo de medición. Debido a la gran cantidad de información que debe ser guardada (hay muchas llamadas a funciones, a pesar de la instrumentación selectiva), al lanzar un programa instrumentado se hace un uso extensivo de dicha memoria RAM. Esto provocó que hubiera que ampliar la memoria RAM del equipo de pruebas, pasando de 512 MB a 2GB. Aún así, estos 2 GB de memoria RAM pueden ser insuficientes para programas con tiempos de ejecuciones muy largos. Como muestra, la ejecución de BZIP2 instrumentada tardó algo más de 11 minutos.

4.1.3. De la instrumentación elegida

La opción de compilación `-finstrument-functions` sólo funciona de manera nativa con códigos fuente escritos en lenguaje C. Los códigos escritos en C++ se pueden adaptar para que dicha opción pueda ser usada con ellos, pero al disponer de numerosos benchmarks de lenguaje C en SPEC CPU2006, se optó por elegir finalmente BZIP2 (escrito en C) para realizar la prueba de concepto de la plataforma.

4.1.4. Software

En la parte software, hubo que actualizar el compilador GCC (a la versión 4.3.3) para que soportara la función de instrumentación descrita anteriormente. Además, tras producirse una avería de la placa base del equipo utilizado para monitorizar, se aprovechó para instalarle una versión más moderna del Sistema Operativo openSUSE, a la que hubo que instalarle las cargas de trabajo de pruebas SPEC CPU2006 [12].

4.2. -finstrument-functions

El compilador GCC [2] provee esta opción de compilación para la generación de código. Utilizando esta opción se generan diversas llamadas a funciones de instrumentación.

`-finstrument-functions` permite 3 maneras posibles de instrumentación:

1. Total: Se instrumentan todas las funciones del programa.

2. Excluyendo funciones (`-finstrument-functions-exclude-function-list=sym1,sym2...`), excluyendo de la instrumentación las funciones indicadas en la lista.

3. Excluyendo ficheros (`-finstrument-functions-exclude-file-list=file1,file2...`), excluyendo de la instrumentación todas las funciones pertenecientes a los ficheros de la lista.

Las opciones 2 y 3 deben ser utilizas en adición a la opción 1. Pueden utilizarse las 3 conjuntamente. Para este proyecto se han utilizado las opciones 1 (para las pruebas con Gauss-Jordan) y 1/3 (para las pruebas con los programas de SPEC CPU2006 como BZIP2).

Para instrumentar un binario, `-finstrument-functions` utiliza 4 funciones que son invocadas en distintos momentos de la ejecución. Su implementación puede verse al detalle en el anexo B, a continuación se describen brevemente.

4.2.1. __profile_begin

Esta función se llama una única vez al principio de la ejecución del programa. Se reserva la memoria necesaria y se inicializa el vector que contendrá todas las muestras temporales (130 millones de elementos) que se tomarán durante la instrumentación.

4.2.2. __cyg_profile_func_enter

Es llamada cada vez que una función instrumentada comienza su ejecución. Se lee el registro temporal utilizado (para saber en qué momento ha comenzado la ejecución la función) y se almacena junto a otra información relevante en el vector de muestras. Esa información relevante es el identificador de la función y un indicador para señalar que la función está siendo invocada.

4.2.3. __cyg_profile_func_exit

Es llamada cada vez que una función instrumentada termina su ejecución. Se lee el registro temporal utilizado (para saber en qué momento ha terminado la ejecución la función) y se almacena junto a otra información relevante en el vector de muestras. Esa información relevante es el identificador de la función y un indicador para señalar que la función está finalizando.

4.2.4. __profile_end

Esta función se llama únicamente una vez al finalizar el programa. En este caso, recorre el vector que contiene las muestras temporales y escribe dichas muestras en un fichero de texto, para ser tratadas posteriormente.

4.3. Time Stamp Counter

En el apartado anterior se ha visto cómo en las funciones de perfilado se establecía el marco temporal de llamadas leyendo un registro temporal que no es otro que dicho Time Stamp Counter. Esa información se guarda junto con la dirección de la función ejecutada (32 bits) y un bit que marca si la función comienza o finaliza en ese momento.

Para reducir el tamaño de los datos guardados en memoria, se ha utilizado el bit de mayor peso de los 64 bits que devuelve la lectura del TSC, modificándolo para indicar Call (con un 0) o Return (con un 1).

Puede verse una representación gráfica de cada muestra de instrumentación tomada en la figura 4.1.

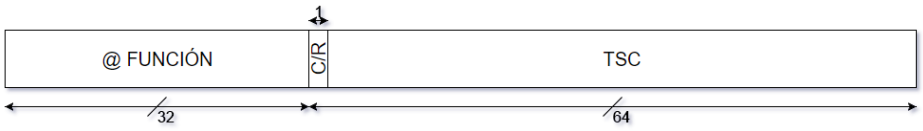

Figura 4.1: Estructura de las muestras de instrumentación

En la lectura del registro y asignación al vector de llamadas en memoria de la información relevante se han utilizado máscaras para minimizar el tiempo consumido por la operación (para más información, consultar anexo B con el código C resultante).

El hecho de que se haya aprovechado el primer bit de los 64 devueltos por la lectura del TSC podría provocar un error si, en mitad de la ejecución del programa, los valores devueltos al leer el registro TSC para ese bit cambiaran de valor, ya que los datos serían inconsistentes.

Como se ha visto en el apartado 2.3.3, el valor del registro TSC se reinicia a 0 al arrancar el ordenador, por lo que con un procesador a 2,8GHz, el valor del primer bit de dicho registro no cambiará de 0 a 1 hasta pasados más de 104 años, por lo que los datos de cualquier ejecución en condiciones normales serán correctos. Aún así, se ha hecho una comprobación de los datos obtenidos (simplemente comprobando que cada valor leído del TSC es mayor que el anterior). En máquinas con instrucciones de 64 bits, las direcciones virtuales son múltiplo de 8, por lo que los 3 bits de menor peso se quedan sin utilizar. Para esas máquinas, se podría evitar este "problema" utilizando uno de los 3 bits de menor peso de la dirección para guardar el indicador de Call/Return, en lugar del bit de mayor peso del TSC.

4.4. Descripción del proceso en dos pasadas

El proceso desarrollado en este proyecto puede verse en la figura 4.2 y puede ser utilizado posteriormente para la instrumentación de cualquier programa.

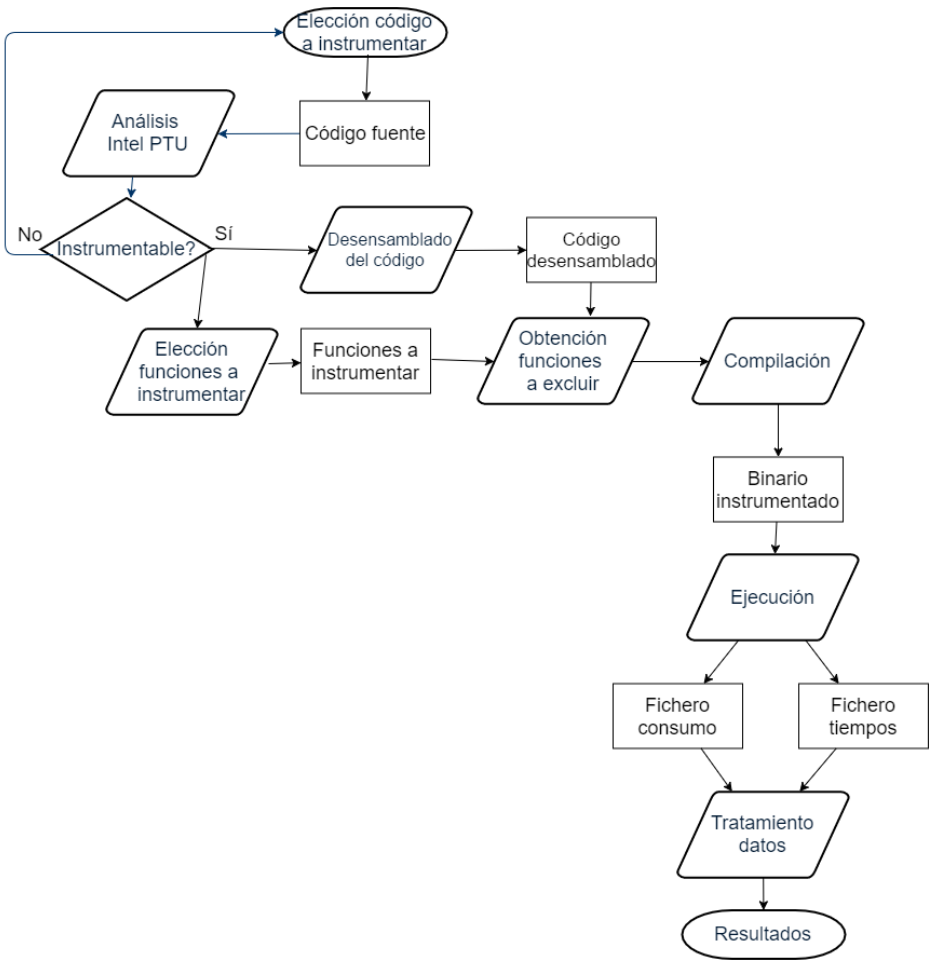

Figura 4.2: Descripción del proceso en dos pasadas.
Primera pasada: *Análisis Intel PTU*. Segunda pasada: *Ejecución*

La idea inicial era instrumentar todas las funciones del binario a estudiar. Pronto se vio que la memoria RAM necesaria para tal efecto sería desmesurada. Además, eso implicaría instrumentar muchísimas funciones (más de lo necesario), provocando una sobrecarga excesiva que invalidaría los resultados obtenidos.

4.4.1. Granularidad

Así pues, el aumento de la granularidad de "instrucción" a "función" no fue suficiente. Según los cálculos descritos en el apartado 4.1.1, transcurren aproximadamente 140.000 ciclos de reloj entre 2 muestras de consumo obtenidas. Esto implica que, todas las funciones que hayan comenzado y terminado su ejecución dentro de ese intervalo de tiempo entre muestras serán "perdidas", pues la plataforma no es lo suficientemente rápida como para poder analizarlas.

Para subsanar este problema se propone un método en 2 pasadas. La primera pasada sirve para realizar un análisis cuantitativo (utilizando Intel PTU [11]) de las funciones del programa y determinar cuáles son las que permanecen un mayor tiempo en ejecución. Estas funciones serán las elegidas para ser instrumentadas utilizando el sistema de instrumentación en la segunda pasada.

Esta decisión se tomó asumiendo que hay correlación entre el tiempo de ejecución y el consumo energético.

En las figuras 4.3 y 4.4 se muestran ejemplos de secuencias de llamadas con respecto a las muestras de consumo, dependiendo de la instrumentación realizada. En la parte superior de las figuras, cada línea vertical representa una muestra de consumo tomada por la plataforma de medición a lo largo del tiempo. En la parte inferior se muestra la secuencia de llamadas de funciones.

En la figura 4.3 se puede ver un ejemplo de lo que podría ser una instrumentación sin una primera pasada o con una primera pasada tras la que se eligieran funciones demasiado cortas: entre 2 muestras de consumo obtenidas hay múltiples comienzos y finales de funciones. Por lo tanto, no es posible saber a qué función asignar el consumo energético relativo a dicho periodo. Este es por tanto un ejemplo de instrumentación incorrecta.

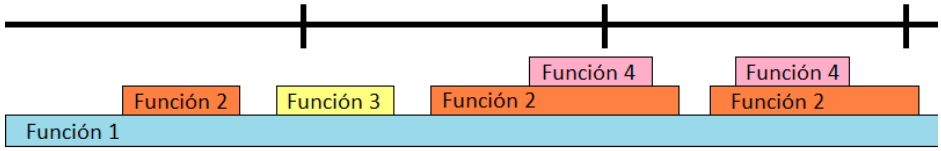

Figura 4.3: Ejemplo de instrumentación incorrecta

En la figura 4.4 se muestra un ejemplo de una instrumentación en la que se ha realizado una primera pasada correcta: se han seleccionado funciones cuyas ejecuciones son suficientemente prolongadas en el tiempo, por lo que durante el tiempo que cada función está activa, hay varias muestras de consumo.

Cuantas más muestras de consumo haya para cada función, más preciso será el calculo del consumo energético.

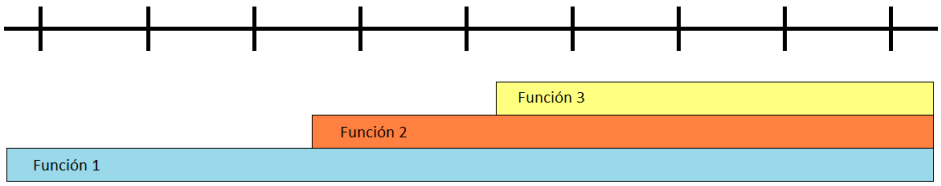

Figura 4.4: Ejemplo de instrumentación correcta

4.5. Instrumentación del binario

4.5.1. Primera pasada: Intel PTU

Según la figura 4.2, corresponde con los nodos "Análisis Intel PTU" y "Elección funciones a instrumentar". Tras la ejecución de Intel PTU para el código fuente elegido, se obtendrá un cuadrante con el número de veces que se llama cada función y con los ciclos de reloj consumidos.

Tras un análisis de dichos datos, habrá que decidir qué funciones merece la pena instrumentar (por supuesto, cuya duración media sea mayor que 140.000 ciclos para esta plataforma, y es recomendable que sea una cifra significativamente superior para que los cálculos de integración numérica sobre las muestras sean válidos).

4.5.2. Obtención de funciones del benchmark

Una vez elegidas las funciones a instrumentar, y debido al funcionamiento de exclusión de funciones de `-finstrument-functions` (no permite decir qué funciones instrumentar si no qué funciones excluir de la instrumentación), hay que preparar una lista con todas las funciones del benchmark excepto las que se quieren instrumentar.

Para ello, y utilizando como entrada el código desensamblado del ejecutable, se ha desarrollado un script en Python (anexo C) que recorre dicho código desensamblado y obtiene los nombres y dirección en memoria de todas las funciones que no deberían ser instrumentadas.

4.5.3. Segunda pasada: Instrumentación propia

Una vez preparada la lista de funciones, ya se puede compilar el código fuente utilizando `-finstrument-functions` con la lista de exclusión, generando un ejecutable instrumentado que dependerá de la elección efectuada. Este ejecutable puede desensamblarse también para ver cómo las únicas funciones

que llaman a las funciones de perfilado descritas en el apartado 4.2 son las funciones a instrumentar.

Realizar una ejecución del binario instrumentado bajo la plataforma de medición de consumo provocará 2 ficheros: Uno que contiene las muestras de consumo (una línea por muestra, aproximadamente 20.000 muestras por segundo) y que se generará en el equipo de medición, y otro que contiene las muestras de tiempo (una linea por muestra, una muestra cada vez que una función comience o termine) y que se generará en el equipo de pruebas.

4.6. Análisis del coeficiente de variación

Con las muestras de consumo obtenidas, para poder realizar un cálculo del consumo energético es necesario realizar una integración numérica.

Para poder elegir entre los distintos métodos de integración valorados (regla de Simpson compuesta o regla del trapecio), se ha desarrollado otro script en Python (anexo D) para saber cuál es el más adecuado o si no hay diferencia entre ambos, calculando el coeficiente de variación de las muestras de consumo obtenidas.

Si dicho coeficiente es cercano a 0, se puede afirmar que la variación de la distribución es pequeña, por lo que ambos métodos darán resultados parecidos. Si el coeficiente es mayor, habría que pensar en aplicar métodos de integración más precisos.

4.7. Obtención de resultados finales

Utilizando como entrada los 2 ficheros obtenidos, se ha desarrollado un último script en Python (anexo E) para combinar los datos de dichos ficheros y mostrar resultados significativos. Para la integración númerica del consumo, el script devuelve los valores según la regla de Simpson compuesta y la regla del trapecio, para poder utilizar una u otra según la variación de las muestras obtenidas.

4.7.1. Resultados totales

La primera versión de este script realizaba el siguiente proceso: recorriendo el fichero de tiempos iba analizando la función en ejecución en cada momento y, para esa ejecución, tomaba la lista de muestras de consumo correspondientes del fichero de consumos. Con esas muestras de consumo, se realizaba una integración (por Simpson y por el trapecio), y se iba acumulando al consumo

total de cada función, obteniendo la energía total consumida por cada función durante la ejecución del programa.

Para una mayor precisión de los datos, hubo que realizar además un proceso de interpolación entre muestras: si sólo se integraran numéricamente todas las muestras obtenidas "dentro" de una función, habría 2 periodos de tiempo no considerados en cada cambio de función (el que va desde el comienzo de la función hasta la primera muestra de consumo de dicha función y el que va desde la última muestra de consumo de la función hasta el fin de dicha función). Para solventar esto, se realizó un sistema de interpolación por el cual se añade una muestra ficticia en el punto de inicio o fin de función, con el valor interpolado linealmente de los muestreos inmediatamente anterior y posterior.

Dicho proceso de interpolación está explicado mediante un ejemplo en las figuras 4.5 y 4.6. En ambas figuras, la línea roja marca el cambio de una función a otra, establecido temporalmente de acuerdo a la lectura del Time Stamp Counter. Las muestras de consumo a su izquierda corresponden a una función, mientras que las muestras de consumo a su derecha corresponden a otra función diferente.

En la figura 4.5 se muestra cómo se calcula el consumo sin aplicar el proceso de interpolación: se toman las muestras que corresponden a cada función y se integran. Pero, como se puede ver en la zona en blanco inmediatamente anterior y posterior a la línea roja, hay una porción del consumo que no se asigna a su función correspondiente.

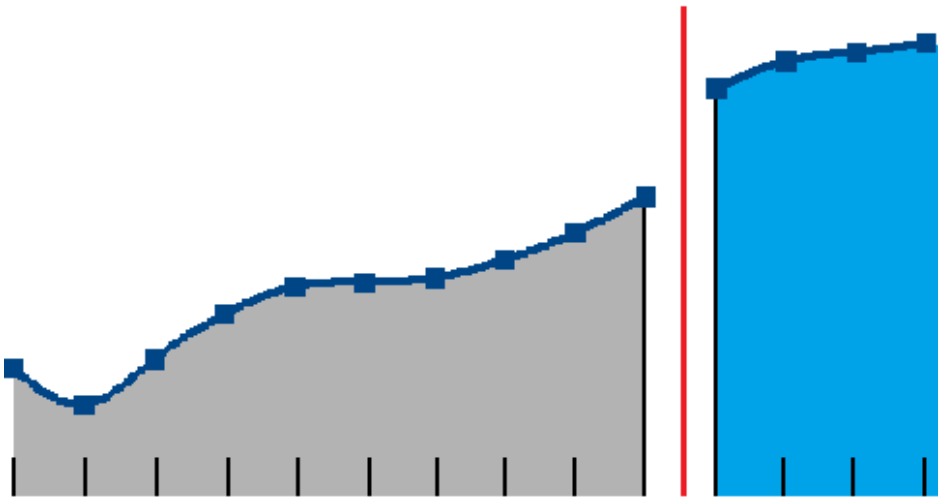

Figura 4.5: Cálculo del consumo antes de la interpolación

Para arreglar este problema, se realiza una interpolación lineal y se crea una muestra extra (punto rojo en la figura) en el momento temporal de la lectura del Time Stamp Counter, trazando una línea recta entre las muestras anterior y posterior a dicha lectura. Dicha muestra extra es añadida como muestra final de la primera función y como muestra inicial de la segunda función. De esta manera,todo el consumo energético es asignado a su función correspondiente. Este proceso de interpolación queda explicado gráficamente en la figura 4.6.

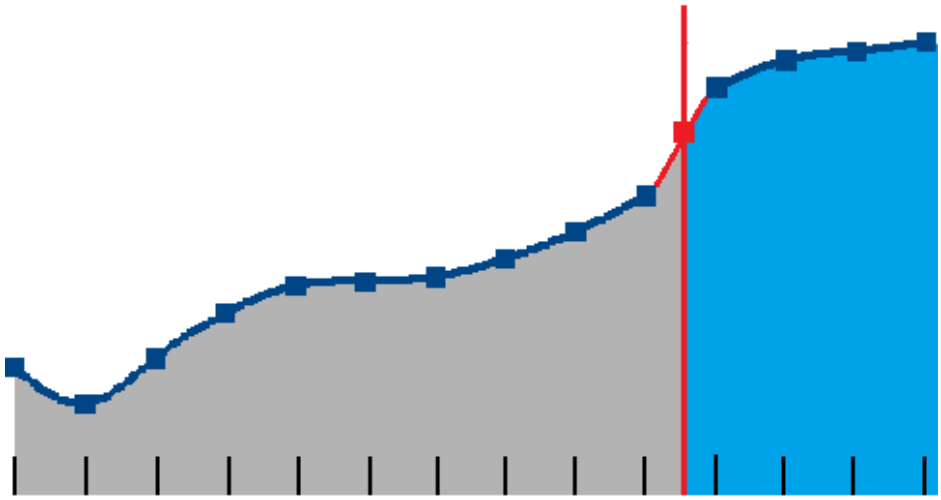

Figura 4.6: Cálculo del consumo tras la interpolación

Aunque totalmente válidos, los resultados totales obtenidos mediante este sistema eran mejorables, pues no daban una visión "temporal" del consumo. Por este motivo, se decidió mejorar el script.

4.7.2. Resultados por intervalos

Para mejorar los resultados obtenidos y darles una representatividad mayor, se optó por dividir el análisis en intervalos, en vez de considerar un único intervalo. Así, para cada intervalo el funcionamiento sería como el descrito en el apartado anterior, obteniendo el consumo total de cada función para el tiempo abarcado en dicho intervalo. De esta manera, haciendo los intervalos lo suficientemente pequeños, se puede comprobar qué funciones se ejecutan en cada momento y cuáles pueden estar incidiendo en variaciones en el consumo energético.

Para este método de análisis por intervalos, hubo que realizar otra interpolación extra similar a la explicada en el apartado anterior (aunque mucho menos significativa en términos de tiempo): por definición, hay una función ejecutándose en el momento en el que se cambia de intervalo. El consumo de dicha función se reparte entre los 2 intervalos de acuerdo a otra interpolación lineal.

Tras este cambio en el script, se ha descartado el primer script (no anexado a este Proyecto de Fin de Carrera), ya que el segundo script devuelve los mismos resultados si se utiliza con número de intervalos = 1 y además permite aumentar dicho número de intervalos para obtener resultados más significativos.

Capítulo 5

Resultados

Antes de mostrar los resultados obtenidos en el proceso completo de instrumentación para BZIP2, hubo que demostrar empíricamente que el sistema de instrumentación elegido no suponía un sobrecoste (de tiempo y, por lo tanto, de consumo) excesivo, pues esto supondría que los resultados obtenidos estarían excesivamente influidos por la modificación del código fuente.

5.1. Sobrecarga mínima (Gauss-Jordan)

Para probar la sobrecarga del sistema de instrumentación utilizado se ha utilizado un código fuente que implementa el algoritmo Gauss-Jordan, para el cálculo de la inversa de una matriz.

Según puede verse en dicho código fuente (anexo A), se ha utilizado una matriz de 1024x1024 floats, para intentar que el tiempo de ejecución de cada rutina fuera razonablemente largo.

Se han realizado dos tipos de ejecución del código: completamente instrumentado (utilizando `-finstrument-functions` sin excluir ninguna subrutina) y mínimamente instrumentado (utilizando `-finstrument-functions` pero excluyendo todas las subrutinas presentes en el código C). La ejecución mínimamente instrumentada es equivalente a una ejecución sin instrumentar, por lo que de esta manera se puede calcular la sobrecarga inducida por el sistema de instrumentación.

Para aumentar la confianza en los resultados obtenidos, se realizaron 10 ejecuciones de cada tipo de ejecución, obteniendo los resultados que se pueden ver en las tablas 5.1 y 5.2.

Instr. mínima	Total ciclos	Tiempo (s)
Prueba 1	16.034.682.202	7,29
Prueba 2	15.844.850.600	7,20
Prueba 3	15.982.268.101	7,26
Prueba 4	15.634.094.741	7,11
Prueba 5	15.847.178.204	7,20
Prueba 6	15.741.618.033	7,16
Prueba 7	15.626.291.316	7,10
Prueba 8	15.681.309.262	7,13
Prueba 9	15.568.891.818	7,08
Prueba 10	15.639.735.816	7,11
Media	**15.760.092.009,3**	**7,16**
Desviación típica	**160.173.200,107**	**0,07**

Tabla 5.1: Gauss-Jordan - Instrumentación mínima.

Instr. total	Total ciclos	Tiempo (s)
Prueba 1	16.635.110.109	7,56
Prueba 2	16.434.287.516	7,47
Prueba 3	16.659.374.593	7,57
Prueba 4	16.792.361.168	7,63
Prueba 5	16.575.228.220	7,53
Prueba 6	16.748.083.012	7,61
Prueba 7	16.792.361.168	7,63
Prueba 8	16.726.103.947	7,60
Prueba 9	16.849.090.400	7,66
Prueba 10	16.615.967.631	7,55
Media	**16.682.796.776,4**	**7,58**
Desviación típica	**124.014.642,0907**	**0,06**

Tabla 5.2: Gauss-Jordan - Instrumentación total.

Las pruebas fueron hechas en un ordenador con procesador a 2,2GHz, y siguiendo esa velocidad se ha realizado la conversión de ciclos a segundos. Como se puede comprobar, los valores en los distintos experimentos son consistentes y la desviación típica es adecuada con respecto a la media. Por tanto, se puede concluir que tal y como se pretendía, el método propuesto no añade variabilidad en los resultados obtenidos.

Comparando la media del número de ciclos en ambas ejecuciones (sin instrumentar y con instrumentación total), hay un incremento del número de ciclos del 5,85 %.

Es una sobrecarga importante, pero razonable para un programa en el que la duración de las funciones es muy pequeña. Analizando el fichero de salida obtenido, se han obtenido los datos que se pueden ver en la tabla 5.3.

Función	**Número de ciclos**	**Número de llamadas**	**Media ciclos/llamada**
generar_matriz_aleatoria	41.695.702	1	41.695.702
permutar_filas	1.047.725.455	64.667	16.201,86
escalonar_matriz	56.411	1	56.411
multip_fila	11.540.387	1.024	11.269,91
sumar_fila_multip	15.516.653.806	1.047.552	14.812,30
ceros_arriba	105.291	1.024	102,82
ceros_abajo	12.110.989	1.024	11.827,14
generar_matriz_identidad	5.221.176	1	5.221.176
hallar_inversa	813	1	813
main	79	1	79
TOTAL	**16.635.110.109**	**1.115.296**	**14.915,42**

Tabla 5.3: Gauss-Jordan - Análisis de la instrumentación total.

La rutina más larga es la utilizada para generar aleatoriamente los valores de la matriz 1024x1024 de entrada. Para que la matriz de entrada sea la misma para las distintas ejecuciones, se ha utilizado la misma semilla.

Según se ve en la tabla 5.3, para una ejecución de poco más de 7 segundos hay más de un millón de llamadas a subrutinas (lo que provoca que haya más de 2 millones de líneas en el fichero de salida). Esto implica que, de media, se hace una llamada al código de instrumentación desarrollado cada menos de 15.000 ciclos.

Es decir, de ejecutarse este programa en la plataforma de medición de consumo, se tomaría una muestra de consumo cada 9 o 10 funciones ejecutadas. Este caso de instrumentación sería incluso peor que el descrito en el ejemplo de la figura 4.3.

Según se ha visto en el capítulo 2, la plataforma de medición de consumo toma una muestra cada 140.000 ciclos aproximadamente, por lo que la ejecución "definitiva" de BZIP2 tendrá rutinas mucho más largas que estas de Gauss-Jordan (no tiene sentido instrumentar rutinas que sean tan cortas que en cuya ejecución no se haya tomado ninguna muestra de consumo), resultando en una sobrecarga muchísimo menor.

5.2. Proceso completo (BZIP2)

Tras una prueba de BZIP2 con instrumentación completa en la que la memoria RAM se agotó debido a la gran cantidad de datos almacenados, se optó por la instrumentación selectiva.

Tras el análisis de Intel PTU para BZIP2 (correspondiente a la primera pasada), cuyos resultados se pueden ver en el anexo F, se optó por instrumentar 6 funciones suficientemente largas, que se pueden ver en la tabla 5.4 y que abarcan la práctica totalidad del tiempo de ejecución del programa.

Función
main
BZ2_compressBlock
BZ2_blockSort
BZ2_bzDecompress
fallbackSort
BZ2_decompress

Tabla 5.4: Funciones instrumentadas de BZIP2

Una vez elegidas las 6 funciones, se llamó al script de obtención de funciones del benchmark (anexo C), para obtener una lista con todas las funciones a excluir (todas excepto las 6 elegidas).

Utilizando `-finstrument-functions-exclude-function-list` con dicha lista de funciones se compiló BZIP2 para obtener el ejecutable según la instrumentación elegida.

Lanzando dicho ejecutable a través de la plataforma se obtuvieron los 2 ficheros "importantes" de salida que contienen toda la información relevante: el fichero de consumo, con 13.610.062 líneas, y el fichero de salida de la instrumentación, con 83.368 líneas, no anexados a este documento por razones obvias.

Utilizando sólo el fichero de consumo obtenido en la ejecución de BZIP2 (de más de 13 millones de líneas y más de 300 megas), se lanzó el script se análisis estadístico (anexo D), obteniéndose los resultados que se pueden ver en la tabla 5.5.

Como el fichero de consumo tiene 2 columnas (voltaje e intensidad) que se multiplican para obtener el consumo total (descrito en el apartado 2.1.2), se ha realizado el análisis estadístico de cada una de las columnas por separado, así como del consumo derivado de los valores de ambas columnas.

Además, el script se mejoró para calcular el rango intercuartílico, cuyos valores también pueden verse en la tabla.

	Media	**DT**	**Coef. var.**	**Q1**	**Q3**	**Rango interc.**
Voltaje	0,3172	0,0637	0,2009	0,261	0,354	0,093
Intensidad	5,778	0,0197	0,0034	5,766	5,795	0,029
Total	36,6318	7,2488	0,1979	30,1977	40,9932	10,7955

Tabla 5.5: Análisis estadístico de las muestras de consumo para la ejecución de BZIP2

Un coeficiente de variación cercano a 0 significa que la variación de la distribución es pequeña, causando que tanto el método de Simpson como el del trapecio den resultados similares.

Utilizando el script de tratamiento de datos (anexo E) en su primera versión con un único intervalo, se obtuvieron los datos globales de la ejecución que se pueden ver en la tablas 5.6, 5.7 y 5.8.

Como se puede ver en la tabla 5.6, los datos de consumo calculados con el método de Simpson son prácticamente iguales a los calculados con el método del trapecio debido al reducido coeficiente de variación de las muestras de consumo, es decir, a que la variación entre muestras de consumo consecutivas es pequeña.

Función	Consumo Simpson (J)	Consumo Trapecio (J)
main	8,96	8,96
BZ2_compressBlock	80,07	80,07
BZ2_blockSort	2.011,24	2.011,24
BZ2_bzDecompress	489,34	489,34
fallbackSort	22.074,54	22.074,53
BZ2_decompress	73,97	73,97
TOTAL	**24.738,12**	**24.738,11**

Tabla 5.6: Datos totales de consumo para BZIP2

Respecto a la distribución temporal de las subrutinas instrumentadas, en la tabla 5.7 se puede ver como la media de cada subrutina son más de 45 millones de ciclos. Comparando con la prueba de Gauss-Jordan vista en el apartado 5.1, es una instrumentación más de 3.000 veces menor, lo que resultaría en una sobrecarga del 0,0019 %, prácticamente inexistente.

Función	Número de ciclos	Número de llamadas	Media ciclos/llamada
main	913.962.312	1	913.962.312
BZ2_compressBlock	6.147.446.128	306	20.089.693,23
BZ2_blockSort	156.064.233.924	306	510.013.836,35
BZ2_bzDecompress	32.493.502.600	40.361	805.071,79
fallbackSort	1.690.651.214.724	306	5.525.003.969,69
BZ2_decompress	4.858.034.660	404	12.024.838,27
TOTAL	**1.891.128.394.348**	**41.684**	**45.368.208,29**

Tabla 5.7: Datos totales de tiempo para BZIP2

Haber conseguido una sobrecarga tan pequeña es el primer objetivo de este Proyecto de Fin de Carrera, pues se puede afirmar que el sistema de instrumentación es correcto y no influye de una manera reseñable en el consumo del programa instrumentado.

Si se observan los porcentajes totales de consumo y tiempo (tabla 5.8), se puede ver una relación prácticamente lineal.

Para una mejor representación visual de los datos de estas 3 tablas, se pueden consultar las figuras 5.1, 5.2 y 5.3.

Función	Simpson	Trapecio	Tiempo
main	0,0362 %	0,03622 %	0,0483 %
BZ2_compressBlock	0,32365 %	0,3237 %	0,325 %
BZ2_blockSort	8,1301 %	8,1301 %	8,2524 %
BZ2_bzDecompress	1,9781 %	1,9781 %	1,718 %
fallbackSort	89,2329 %	89,2329 %	89,399 %
BZ2_decompress	0,299 %	0,299 %	0,2569 %

Tabla 5.8: Porcentajes totales para BZIP2

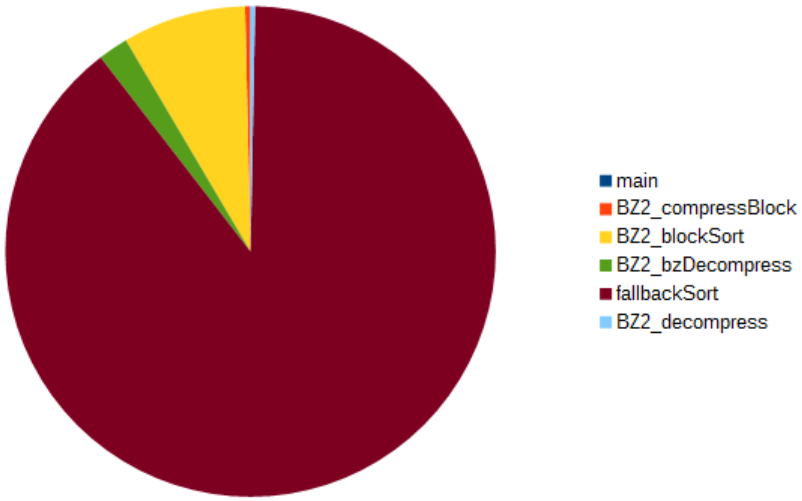

Figura 5.1: BZIP2: Consumo obtenido mediante el método de Simpson

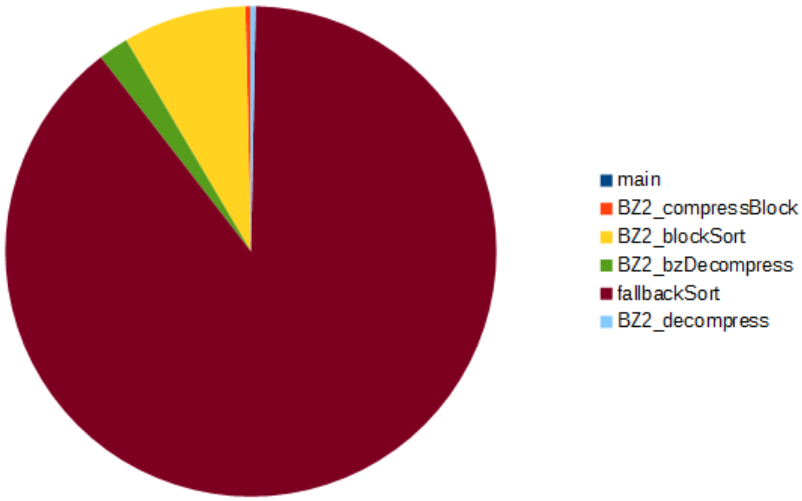

Figura 5.2: BZIP2: Consumo obtenido mediante el método del trapecio

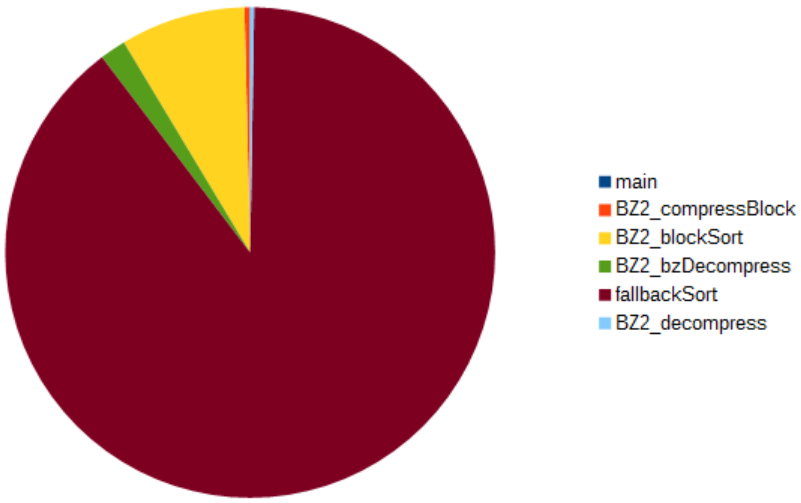

Figura 5.3: BZIP2: Reparto de tiempo

Aunque esta información da una visión global de la relación entre consumo y tiempo durante la ejecución del programa (de más de 11 minutos), no permite saber cuándo ha podido haber incrementos en el consumo o qué subrutinas han podido provocarlo.

Para ello, y como se ha indicado en el apartado 4.7.2, se realizó una modificación del script, realizando un tratamiento por intervalos . Se hicieron sucesivas pruebas aumentando el número de intervalos (con 20, 50 o 100) llegando hasta 200 intervalos. Los resultados de dichas pruebas intermedias pueden consultarse en el anexo G. El resultado del tratamiento final de datos creando 200 intervalos (cada intervalo correspondería a algo menos de 3,5 segundos) se puede ver en las figuras 5.4, 5.5 y 5.6.

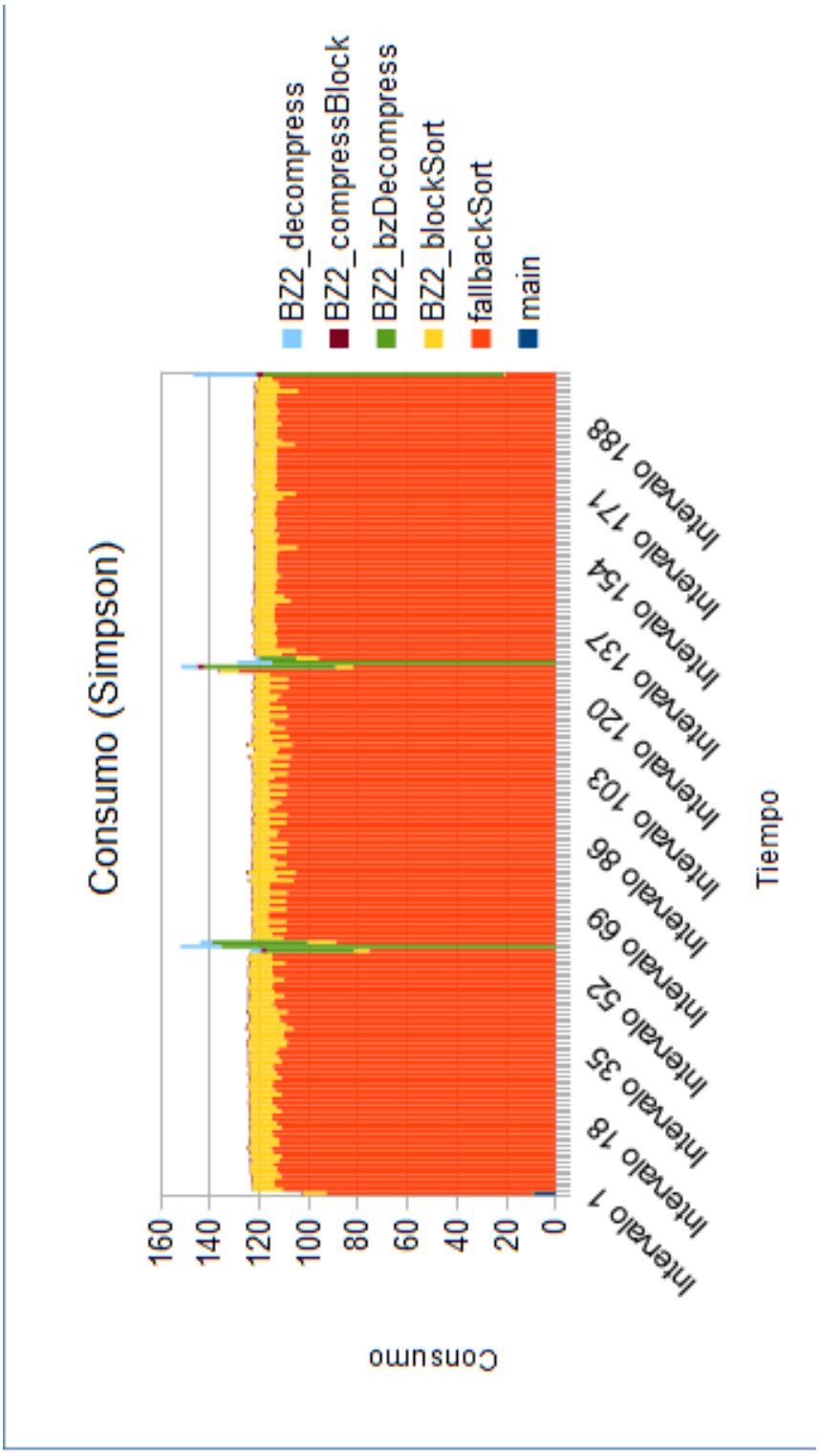

Figura 5.4: BZIP2 200 intervalos: Consumo obtenido mediante el método de Simpson

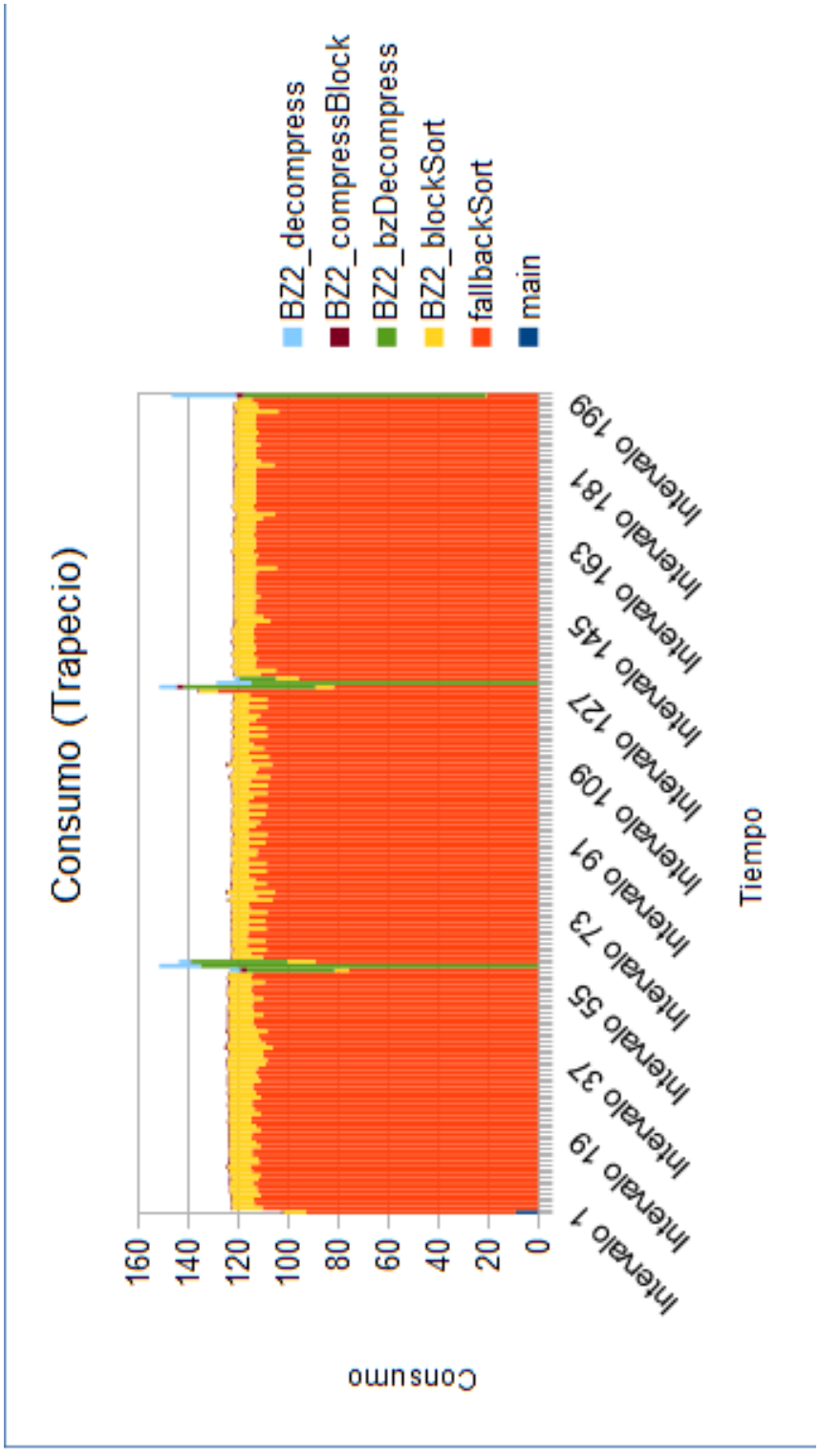

Figura 5.5: BZIP2 200 intervalos: Consumo obtenido mediante el método del trapecio

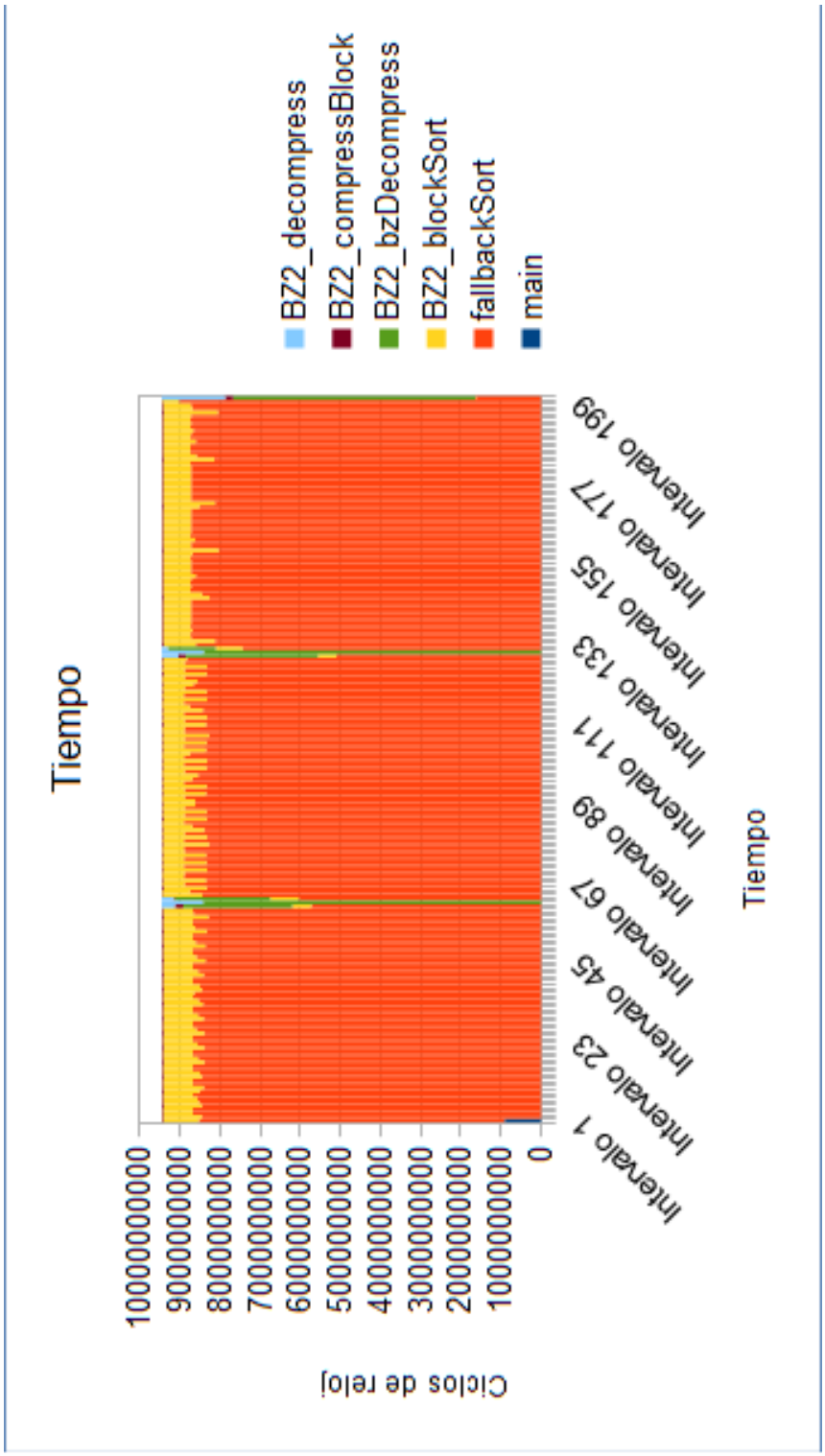

Figura 5.6: BZIP2 200 intervalos: Reparto de tiempo

Antes de analizar la salida de este tratamiento con 200 intervalos, es recomendable saber qué hace BZIP2 realmente: Como entrada tiene una serie de archivos: 2 JPEGs, un binario ejecutable, código fuente en un .tar, un html y un fichero "combinado" (que contiene archivos altamente comprimibles y no muy comprimibles) Para esta entrada, realiza 3 series de compresión-descompresión, utilizando distintos niveles de compresión.

Analizando las imágenes, se puede ver como las funciones de descompresión (color azul claro y verde) se ejecutan durante un corto periodo de tiempo después de haberse ejecutado las funciones de compresión.

También se puede ver cómo dichas funciones de descompresión son más costosas energéticamente: en los intervalos en los que están activas, el consumo total aumenta significativamente.

Esto supondría que se ha conseguido el objetivo final que se tenía para este Proyecto de Fin de Carrera: ser capaz de identificar qué funciones provocan aumentos de consumo sin influir significativamente en dicho consumo (debido a la baja sobrecarga conseguida con el sistema de instrumentación desarrollado).

Capítulo 6

Conclusiones

Mediante el proceso descrito en este Proyecto de Fin de Carrera, se ha construido un mecanismo de instrumentación muy poco intrusivo (y que por lo tanto no influye en los resultados).

El proceso construido se puede considerar válido por los siguientes motivos:

- Está automatizado. Siguiendo el flujo descrito, se pueden obtener los resultados siempre que se cumplan ciertos prerrequisitos.
- Es sencillo de usar. Las herramientas ya están desarrolladas, así que sólo hay que lanzar los programas y los scripts para obtener los resultados.
- Los resultados son buenos. La sobrecarga inducida es mínima.

El procesador Intel Pentium IV era un procesador muy agresivo que utilizaba muchos recursos en control, por lo que todas las instrucciones consumían prácticamente lo mismo. Por lo tanto, y como se ha podido comprobar en el capítulo anterior, el consumo energético es prácticamente proporcional al tiempo de ejecución.

Sin embargo, siguiendo el proceso descrito, se ha podido analizar el consumo energético de BZIP2, programa perteneciente a SPEC CPU2006, y se han podido identificar subrutinas que provocan pequeños picos de consumo.

De cara a una continuación de este Proyecto, con una plataforma de medición de consumo mejor (que tomara más muestras por segundo), se podría reducir la granularidad (instrumentando rutinas más pequeñas), pudiéndose obtener resultados todavía más precisos.

Bibliografía

[1] [online] clang: a c language family frontend for llvm. https://clang.llvm.org/.

[2] [online] gcc, the gnu compiler collection. https://gcc.gnu.org/.

[3] [online] gnu binutils. https://www.gnu.org/software/binutils.

[4] [online] ia-pc hpet (high precision event timers) specification. http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf.

[5] [online] intel c++ compilers. https://software.intel.com/en-us/c-compilers.

[6] [online] pin - a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[7] [online] spec cpu 2006. https://www.spec.org/cpu2006/.

[8] [online] the pit: A system clock. http://www.osdever.net/bkerndev/Docs/pit.htm.

[9] Octavio Benedí. Determinación del consumo en procesadores de altas prestaciones y caracterización energética de programas compilados. proyecto fin de carrera. zaragoza, septiembre 2008.

[10] P. Chapin, J.R. Costello, T. Desai, B. Farrell, D.J. King, and T. Schaffer. Real time clock, September 8 2009. US Patent 7,586,377.

[11] Intel Corporation. Intel performance tuning utility.

[12] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. 40(6):190–200, 2005.

[14] Gabriele Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures, September 2010.

[15] Michael J. Paul and John E. Gochenouer. A high resolution event timer ada package for dos environments. *Ada Lett.*, XIV(1):61–67, January 1994.

[16] Alicia Asín Pérez. Evaluación del consumo en procesadores de altas prestaciones. proyecto fin de carrera, November 2006.

[17] Julien Ridoux, Darryl Veitch, and Timothy Broomhead. The case for feed-forward clock synchronization. *IEEE/ACM Trans. Netw.*, 20(1):231–242, February 2012.

[18] Sergio Gutiérrez Verde. Aspectos térmicos de la ejecución de programas: estudio experimental sobre un pentium iv. proyecto fin de carrera. zaragoza, febrero 2009.

[19] Reinhold P. Weicker and John L. Henning. Subroutine profiling results for the cpu2006 benchmarks. *SIGARCH Comput. Archit. News*, 35(1):102–111, March 2007.

Anexos

Anexo A

Pruebas de validacion sobre Gauss-Jordan

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include<time.h>
5
6  /*********** VARIABLES GLOBALES **********************/
7  #define N 1024                /*dimension de la matriz*/
8  float m[N][N];
9  float id[N][N];
10
11 /*-------------------------------------------------------------------------*/
12 void generar_matriz_aleatoria(void)
13 {
14     srand(0);
15     for(int i = 0; i < N; i++)
16     {
17         for(int j = 0; j < N; j++)
18         {
19             m[i][j] = rand() % 6;
20         }
21     }
22 }
23
24 /*-------------------------------------------------------------------*/
25 /* SE DEFINEN LAS 3 OPERACIONES ELEMENTALES DE FILA */
26 /* */
27 /* Las operaciones que se le realizen a la matriz para reducirla */
28 /* tambien deberan realizarsele a la matriz identidad para obtener */
29 /* la matriz inversa */
30 /*-------------------------------------------------------------------*/
31 void permutar_filas(int fila1, int fila2)
32 {
33     float auxval;
34     int cont;
35     for(cont = 0; cont < N; cont++)
36     {
37         auxval = m[fila1][cont];
38         m[fila1][cont] = m[fila2][cont];
39         m[fila2][cont] = auxval;
```

```
40
41          auxval = id[fila1][cont];
42          id[fila1][cont] = id[fila2][cont];
43          id[fila2][cont] = auxval;
44      }
45  }
46
47  /*-------------------------------------------------------------------------*/
48  /* */
49  /* Ordena la matriz de forma que quede en su forma escalonada por */
50  /* renglones */
51  /* */
52  /*-------------------------------------------------------------------------*/
53  void escalonar_matriz(void)
54  {
55      int cont, col, ceros, vec[N];
56      int flag, aux;
57
58      for(cont=0;cont<N;cont++)
59      {
60          col = 0;
61          ceros = 0;
62
63          if(m[cont][col] == 0)
64          {
65              do
66              {
67                  ceros++;
68                  col++;
69              } while(m[cont][col] == 0);
70          }
71          vec[cont] = ceros;
72      }
73
74      do
75      {
76          flag = 0;
77          for(cont = 0; cont < N - 1; cont++)
78          {
79              if(vec[cont] > vec[cont+1])
80              {
81                  aux = vec[cont];
82                  vec[cont] = vec[cont + 1];
83                  vec[cont + 1] = aux;
84                  permutar_filas(cont, cont + 1);
85                  flag = 1;
86              }
87          }
88      } while(flag == 1);
89  }
90
91  /*-------------------------------------------------------------------------*/
92  void multip_fila(int fila, double factor)
93  {
94      int cont;
95      for(cont=0; cont < N; cont++)
96      {
97          m[fila][cont] = (m[fila][cont]) * factor;
98          id[fila][cont] = (id[fila][cont]) * factor;
99      }
100 }
101
```

```
102 /*-------------------------------------------------------------------------*/
103 void sumar_fila_multip(int fila1, int fila2, double factor)
104 {
105     int cont;
106     for(cont = 0; cont<N; cont++)
107     {
108         m[fila1][cont] = (m[fila1][cont]) + ((m[fila2][cont]) * factor);
109         id[fila1][cont] = (id[fila1][cont]) + ((id[fila2][cont]) * factor);
110     }
111 }
112
113 /*-------------------------------------------------------------------------*/
114 void ceros_arriba(int fila_pivote, int columna_pivote)
115 {
116     for(int cont = 0; cont < fila_pivote; cont++)
117     {
118         sumar_fila_multip(cont, fila_pivote,((m[cont][columna_pivote]) *
                (-1)));
119     }
120 }
121
122 /*-------------------------------------------------------------------------*/
123 void ceros_abajo(int fila_pivote, int columna_pivote)
124 {
125     for(int cont = columna_pivote + 1; cont < N; cont++)
126     {
127         sumar_fila_multip(cont, fila_pivote, ((m[cont][columna_pivote]) *
                (-1)));
128     }
129 }
130
131 /*-------------------------------------------------------------------------*/
132 void generar_matriz_identidad(void)
133 {
134     for(int i = 0; i < N; i++)
135     {
136         for(int j = 0; j < N; j++)
137         {
138             if(i == j) id[i][j] = 1;
139             else id[i][j] = 0;
140         }
141     }
142 }
143
144 /*-------------------------------------------------------------------------*/
145 void hallar_inversa(void)
146 {
147     int cont, cont2, flag=0;
148
149     escalonar_matriz();
150     generar_matriz_identidad(); //rellena la matriz identidad
151
152     for(cont = 0; cont < N; cont++) //recorre filas
153     {
154         for(cont2 = 0; cont2 < N; cont2++) //recorre columnas
155         {
156             if(m[cont][cont2] != 0) //busca pivote (elemento ditinto de 0)
157             {
158                 if(m[cont][cont2] != 1) //si pivote no es 1, se lo
                        multiplica
159                 {
160                     multip_fila(cont, pow(m[cont][cont2], -1));
```

```
161                 }
162                 ceros_arriba(cont,cont2); // se hacen 0's por arriba
163                 ceros_abajo(cont,cont2); // y por debajo del pivote
164                 break;
165             }
166         }
167     }
168 
169 /*-----------------------------------------------------------------*/
170 /* Una vez terminada esta operacion, la matriz identidad estara */
171 /* transformada en la inversa */
172 /* */
173 /* Ahora se comprueba que la matriz original este transformada */
174 /* en la matriz identidad, de no ser asi la inversa obtenida */
175 /* no es valida y la matriz no tiena inversa */
176 /*-----------------------------------------------------------------*/
177     for(cont = 0; cont < N; cont++)
178     {
179         for(cont2 = 0; cont2 < N; cont2++)
180         {
181             if(cont == cont2)
182             {
183                 if(m[cont][cont2] != 1)
184                 {
185                     flag = 1;
186                 }
187             }
188             else
189             {
190                 if(m[cont][cont2]!=0)
191                 {
192                     flag = 1;
193                 }
194             }
195         }
196     }
197 
198     if(flag == 1)
199     {
200         printf("\nLa matriz no tiene inversa\n");
201     }
202 }
203 
204 /*-----------------------------------------------------------------------*/
205 int main()
206 {
207     generar_matriz_aleatoria();
208     hallar_inversa();
209     return 0;
210 }
```

codigos/gaussJordan1024.c

Anexo B

Fichero de instrumentación

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 /* Lectura del Time Stamp Counter */
6 #define rdtscll(val) asm volatile ("rdtsc" : "=A" (val))
7
8 #define NUM_ELEMENTOS 130000000 /*max tamano del vector de instrumentacion
      -> 130 millones*/
9
10 typedef struct nodo {
11   long long unsigned tsc; /* valor del time stamp counter (contador de
        ciclos) */
12   unsigned crfunc; /* call or return y direccion de  function, el bit
13           de mayor peso si vale 0 es un call y si es 1 un return */
14 } nodo;
15
16 #define MASK_SET_RETURN (0x80000000) /*mascara con un 1 y todo 0s*/
17 #define MASK_SET_CALL  (0x7FFFFFFF) /*mascara con un 0 y todo 1s*/
18
19 #define LEER_FUNC_DIR(nd) ((nd).crfunc & MASK_SET_CALL)
20 #define LEER_CALL_RET(nd) ((nd).crfunc >> 31)
21
22
23 /*********** VARIABLES GLOBALES **********************/
24 static long long unsigned tInicio;
25 static nodo * vectNodos = NULL;
26 static long indice; /* iterador del vector de nodos */
27
28
29 /************ FUNCIONES DE INSTRUMENTACION ************/
30 #ifdef __cplusplus
31 extern "C" {
32 #endif
33
34 void __profile_begin(void) __attribute__ ((constructor,
      no_instrument_function)); /* Inicio del programa */
35 void __profile_end(void) __attribute__ ((destructor,
      no_instrument_function)); /* Fin del programa */
36 void __cyg_profile_func_enter (void *, void *) __attribute__
      ((no_instrument_function)); /* Inicio de cada funcion */
37 void __cyg_profile_func_exit (void *, void *) __attribute__
```

```
     ((no_instrument_function)); /* Fin de cada funcion */
38
39 void __profile_begin (void)
40 {
41   if (NULL == (vectNodos = (nodo *) calloc(NUM_ELEMENTOS, sizeof(nodo))))
42   {
43     fprintf(stderr, "Error adquiriendo memoria\n");
44     exit(1);
45   }
46   indice = 0;
47
48   rdtscll(tInicio);
49 #ifdef DEBUG
50   printf("Inicio instrumentacion, num ciclos: %llu\n", tInicio);
51 #endif
52 }
53
54 void __profile_end (void)
55 {
56   FILE * f_salida = NULL;
57   int i;
58   const char *etiquetas[2] = {"0", "1"};
59
60 #ifdef DEBUG
61   long long unsigned tFin;
62
63   rdtscll(tFin);
64   printf("Fin instrumentacion, indice %li, num ciclos: %llu\n", indice,
       tFin - tInicio);
65 #endif
66
67   if (NULL == (f_salida = fopen("salida_instrumentacion_propia.txt","w")))
68   {
69     printf("No se puede abrir el fichero de salida.\n");
70     exit(1);
71   }
72
73   for (i = 0; i < indice; i++)
74   {
75     fprintf(f_salida, "%X\t%s\t%llu\n", LEER_FUNC_DIR(vectNodos[i]),
76         etiquetas[LEER_CALL_RET(vectNodos[i])], vectNodos[i].tsc - tInicio);
77   }
78
79   fclose(f_salida);
80 }
81
82 void __cyg_profile_func_enter(void *func, void *caller)
83 {
84   unsigned long long t;
85
86   rdtscll(t);
87   vectNodos[indice].tsc = t;
88   vectNodos[indice++].crfunc = ((unsigned) func & MASK_SET_CALL);
89 #ifdef DEBUG
90   printf("Entro en la funcion %X, indice=%li\n", (unsigned) func &
       MASK_SET_CALL, indice);
91 #endif
92
93 }
94
95 void __cyg_profile_func_exit(void *func, void *caller)
96 {
```

```
97    unsigned long long t;
98
99    rdtscll(t);
100   vectNodos[indice].tsc = t;
101   vectNodos[indice++].crfunc = ((unsigned) func | MASK_SET_RETURN);
102 #ifdef DEBUG
103   printf("Salgo de la funcion %X, indice=%li\n", (unsigned) func |
          MASK_SET_RETURN, indice);
104 #endif
105
106 }
107 #ifdef __cplusplus
108 }
109 #endif
```

codigos/ficheroInstrumentacion.c

Anexo C

Obtención de funciones del benchmark

```
1  import array
2
3  # ------------------------------------------------
4  # Constantes
5  # ------------------------------------------------
6  FICHERO_ENTRADA = "tags"
7  FICHERO_SALIDA = "funciones"
8  EXTENSION_TEXTO = ".txt"
9  RUTA_FICHEROS = "/home/victor/"
10 #RUTA_FICHEROS = "/home/victor/Desktop/PFC/python/"       # ->  En Susy
11 #RUTA_FICHEROS = ""               # ->  Si el fichero python y el de tags
       estan en el mismo directorio
12 INSTRUMENTACION = "-finstrument-functions-exclude-function-list="
13 lon_nombre_funcion = 13
14
15 funciones_instrumentar =
       ['regmove_optimize','compute_transp','bitmap_operation']
16
17
18 # ------------------------------------------------
19 # Variables
20 # ------------------------------------------------
21 lista_salida = [] # Contendra las distintas funciones almacenadas en el
       fichero
22
23 # ------------------------------------------------
24 # Programa
25 # ------------------------------------------------
26
27 # Se abre el fichero de tags
28 nombre_fichero = FICHERO_ENTRADA #+ EXTENSION_TEXTO
29 try:
30   #fichero_entrada = open(RUTA_FICHEROS + nombre_fichero, "r")
31   fichero_entrada = open(nombre_fichero, "r")
32 except:
33   print "El archivo " + nombre_fichero + " no existe."
34
35 # Se guardan las lineas del fichero en una lista
36 lista_lineas = fichero_entrada.readlines()
```

```
37
38 # Se recorre la lista de lineas
39 for linea in lista_lineas:
40
41   # Se divide la linea en los distintas partes
42   tupla_valores = linea.split("\t")
43
44   # Obtenemos el numero de elementos de la linea
45   len_tupla = len(tupla_valores)
46
47   indicador_funcion = tupla_valores[len_tupla - 2]  # Penultimo elemento ->
         Indicador de si es funcion o no
48   # Si es una funcion la guardamos en la lista de salida
49   if indicador_funcion == "f":
50     nombre_funcion = tupla_valores[0]
51     print nombre_funcion
52     lista_salida.append(nombre_funcion)
53
54   indicador_funcion = tupla_valores[len_tupla - 1]  # Ultimo elemento ->
         Tambien puede ser indicador de si es funcion o no
55   # Si es una funcion la guardamos en la lista de salida
56   if indicador_funcion == "f\n": #Si es el ultimo lleva "pegado" el \n
57     nombre_funcion = tupla_valores[0]
58     print nombre_funcion
59     lista_salida.append(nombre_funcion)
60
61 # Se eliminan nombres de funciones duplicados:
62 # primero convirtiendola en set (eliminandose los duplicados) y luego
       pasandola a lista otra vez
63 print "Numero de elementos en la lista antes de eliminar duplicados: ",
       len(lista_salida)
64 lista_salida = list(set(lista_salida))
65 print "Numero de elementos en la lista despues de eliminar duplicados: ",
       len(lista_salida)
66
67 # Se abre el fichero de salida en el que se mostrara
68 try:
69 #  fichero_salida = open(RUTA_FICHEROS + FICHERO_SALIDA + EXTENSION_TEXTO,
       "w")
70   fichero_salida = open(FICHERO_SALIDA + EXTENSION_TEXTO, "w")
71 except:
72   print "Error al crear el fichero de salida " + FICHERO_SALIDA +
         EXTENSION_TEXTO
73
74 # Se eliminan las funciones que esten en el vector o sean subcadenas de
       algun elemento del vector
75
76 print "Las funciones a eliminar de la lista son: ", funciones_instrumentar
77
78 fichero_salida.write(INSTRUMENTACION)
79 for funcion in lista_salida:
80   subcadena = 0
81   for elemento in funciones_instrumentar:
82     if funcion in elemento:
83       print "La funcion " + funcion + " es una subcadena de " + elemento
84       subcadena = 1
85   if not subcadena:
86     fichero_salida.write(funcion)
87     fichero_salida.write(",")
```

codigos/obtiene_funciones.py

Anexo D

Validación estadística de las muestras

```
1 import numpy as np
2
3 # -------------------------------------------------------
4 # Constantes
5 # -------------------------------------------------------
6 FICHERO_CONSUMO = "bzip2_base.i386-m32-gcc42-nn_0"
7 FICHERO_LOG = "log"
8 EXTENSION_TEXTO = ".txt"
9 AJUSTE_CONSUMO = 20 # 2 (siempre) * 10 (amplificador a 10 A/V)
10
11
12 def main():
13     c1 = []
14     c2 = []
15     ct = []
16
17     try:
18         nombre_fichero = (FICHERO_CONSUMO + EXTENSION_TEXTO)
19         consumo = open(nombre_fichero, "r")
20         nombre_fichero = (FICHERO_LOG + EXTENSION_TEXTO)
21         log = open(nombre_fichero, "w")
22     except:
23         s = "El archivo " + nombre_fichero + " no existe\n"
24         log.write(s)
25
26     # Vuelco el fichero en lista
27     lista_consumo = consumo.readlines()
28     long_consumo = len(lista_consumo)
29     s = "\nNumero de muestreos de consumo: " + str(long_consumo)
30     log.write(s)
31
32     for linea in lista_consumo:
33         tupla = linea.strip().split(",",2)
34         c1.append(float(tupla[0]))
35         c2.append(float(tupla[1]))
36         ct.append(AJUSTE_CONSUMO * float(tupla[0]) * float(tupla[1]))
37
38     log.write("\n====================")
39     log.write("\nANALISIS ESTADISTICO")
```

```
40      log.write("\n====================")
41      mc1 = np.mean(c1)
42      mc2 = np.mean(c2)
43      mct = np.mean(ct)
44      dc1 = np.std(c1)
45      dc2 = np.std(c2)
46      dct = np.std(ct)
47      p25c1 = np.percentile(c1,25)
48      p25c2 = np.percentile(c2,25)
49      p25ct = np.percentile(ct,25)
50      p75c1 = np.percentile(c1,75)
51      p75c2 = np.percentile(c2,75)
52      p75ct = np.percentile(ct,75)
53      s = "\nMedia muestras (columna 1): " + str(mc1)
54      log.write(s)
55      s = "\nMedia muestras (columna 2): " + str(mc2)
56      log.write(s)
57      s = "\nMedia muestras (ambas columnas): " + str(mct)
58      log.write(s)
59      log.write("\n============")
60      s = "\nDesviacion Tipica muestras (columna 1): " + str(dc1)
61      log.write(s)
62      s = "\nDesviacion Tipica muestras (columna 2): " + str(dc2)
63      log.write(s)
64      s = "\nDesviacion Tipica muestras (ambas columnas): " + str(dct)
65      log.write(s)
66      log.write("\n============")
67      s = "\nCoeficiente variacion muestras (columna 1): " + str(float(dc1 /
            mc1))
68      log.write(s)
69      s = "\nCoeficiente variacion muestras (columna 2): " + str(float(dc2 /
            mc2))
70      log.write(s)
71      s = "\nCoeficiente variacion muestras (ambas columnas): " +
            str(float(dct / mct))
72      log.write(s)
73      log.write("\n============")
74      s = "\nPercentil 25 (columna 1): " + str(p25c1)
75      log.write(s)
76      s = "\nPercentil 25 (columna 2): " + str(p25c2)
77      log.write(s)
78      s = "\nPercentil 25 (ambas columnas): " + str(p25ct)
79      log.write(s)
80      log.write("\n============")
81      s = "\nPercentil 75 (columna 1): " + str(p75c1)
82      log.write(s)
83      s = "\nPercentil 75 (columna 2): " + str(p75c2)
84      log.write(s)
85      s = "\nPercentil 75 (ambas columnas): " + str(p75ct)
86      log.write(s)
87      log.close()
88
89  if __name__ == "__main__":
90      main()
```

codigos/analisis_estadistico.py

Anexo E

Tratamiento (por intervalos) de las muestras

```
1 from scipy.integrate import simps, trapz
2 import sys
3
4 # ------------------------------------------------------
5 # Constantes
6 # ------------------------------------------------------
7 FICHERO_CONSUMO = "bzip2_base.i386-m32-gcc42-nn_0"
8 FICHERO_SALIDA = "salida_instrumentacion_propia"
9 FICHERO_FUNCIONES = "direcciones_funciones_instrumentadas"
10 FICHERO_LOG = "log"
11 EXTENSION_TEXTO = ".txt"
12 AJUSTE_CONSUMO = 20 # 2 (siempre) * 10 (el amplificador esta a 10 A/V)
13 VELOCIDAD_PROCESADOR = 2800000000 # 2.8 GHz
14 NUM_INTERVALOS = 200 # Numero de "agrupamientos" de consumo
15
16 # ------------------------------------------------------
17 # Variables
18 # ------------------------------------------------------
19 # Clave de la tabla: dir de la funcion. Valor: Lista con nombre, num de
       llamadas y num de ciclos
20 pila = []       # Lista en la que voy a simular la pila de llamadas a
       funciones
21 # Esta pila esta compuesta por tuplas formadas a su vez por por [dir, TSC y
       num_ciclos consumidos por las funciones "hijas"]
22
23 dict_funciones = {}
24 lista_total_puntos = [] # Contendra todos los puntos de la funcion
25 muestras_funcion_actual = [] # Ira conteniendo las muestras de cada funcion
       "activa"
26
27 dict_list_simps = [dict() for x in range(NUM_INTERVALOS)]
28 dict_list_trapz = [dict() for x in range(NUM_INTERVALOS)]
29 dict_list_tiempo = [dict() for x in range(NUM_INTERVALOS)]
30
31 # ------------------------------------------------------
32 # Programa
33 # ------------------------------------------------------
34 def main():
35
```

```
36      try:
37          nombre_fichero = (FICHERO_SALIDA + EXTENSION_TEXTO)
38          salida = open(nombre_fichero, "r")
39          nombre_fichero = (FICHERO_CONSUMO + EXTENSION_TEXTO)
40          consumo = open(nombre_fichero, "r")
41          nombre_fichero = (FICHERO_FUNCIONES + EXTENSION_TEXTO)
42          funciones = open(nombre_fichero, "r")
43          log = open(FICHERO_LOG + EXTENSION_TEXTO, "w")
44      except:
45          print("El archivo", nombre_fichero, "no existe")
46
47      s = "Consumo"
48      lista_funciones = funciones.readlines()
49      lista_funciones.sort()
50      for linea_funcion in lista_funciones:
51          tupla_funcion = linea_funcion.strip().split(":",2)
52          if tupla_funcion[0]:
53              dict_funciones[tupla_funcion[0]] = tupla_funcion[1]
54              s = s + "\t" + tupla_funcion[1]
55      print("Direcciones de las funciones a tratar y nombres de estas:",
            dict_funciones)
56      log.write(s)
57
58      # Vuelco los ficheros en 2 listas.
59      lista_salida = salida.readlines()    # Mete todas las lineas del
            fichero de salida en una lista para recorrerla despues
60      print("Fichero de instrumentacion volcado a memoria.")
61      lista_consumo = consumo.readlines()
62      print("Fichero de consumo volcado a memoria.")
63      long_salida = len(lista_salida)
64      long_consumo = len(lista_consumo)
65      print("Numero de calls/returns:", long_salida)
66      print("Numero de muestreos de consumo:", long_consumo)
67
68      num_TSCs = int(lista_salida[long_salida - 1].strip().split("\t",3)[2])
            - int(lista_salida[0].strip().split("\t",3)[2])
69      ciclos_intervalo = float(num_TSCs / NUM_INTERVALOS)
70      consumos_intervalo = float(long_consumo / NUM_INTERVALOS)
71      ciclos_por_linea = float(num_TSCs) / (long_consumo - 1) #138.950'757745
72      dx_modif = float(ciclos_por_linea) / float(VELOCIDAD_PROCESADOR)
            #Teoricamente era 0.00005
73
74      print("TSC final segun la instrumentacion:", num_TSCs, "ciclos.")
75      print("Numero de intervalos:", NUM_INTERVALOS)
76      print("Numero de ciclos en cada intervalo:", num_TSCs, "/",
            NUM_INTERVALOS, "=", ciclos_intervalo)
77      print("Tiempo teorico segun las muestras de consumo si fuera una
            muestra cada 140.000 ciclos exactos):", long_consumo * 140000,
            "ciclos.")
78      print("Valor empirico obtenido para el numero de ciclos transcurridos
            entre cada muestra de consumo:", num_TSCs, "/", long_consumo - 1,
            "=", ciclos_por_linea, "ciclos.")
79      print("Diferencia entre muestras de consumo (en segundos): ", dx_modif)
80
81      indice_consumo = 0 #Para que la primera muestra sea a los 140.000
            ciclos aprox.
82      indice_salida = 0
83      x = 0
84      valor_interpolado = 0
85      x_interpolada = 0
86      intervalo = 0
87
```

```
88      for linea_salida in lista_salida:
89          if not indice_salida: #Primera iteracion
90              tupla_vieja = linea_salida.strip().split("\t",3)
91              primer_tsc = int(tupla_vieja[2])
92              pila.append(tupla_vieja)
93          else:
94              tupla_nueva = linea_salida.strip().split("\t",3)
95              num_lineas_consumo = (int(tupla_nueva[2]) -
                    int(tupla_vieja[2])) / ciclos_por_linea
96              parte_entera = int(num_lineas_consumo)
97              parte_decimal = num_lineas_consumo - parte_entera
98
99              if num_lineas_consumo:  #Solo hacemos el proceso si hay lineas
                    de consumo para la linea de salida
100                 indice = 0
101
102                 #Si existen, anyado como iniciales los valores interpolados
                        calculados en la iteracion anterior
103                 if (valor_interpolado):
104                     valor_consumo = valor_interpolado
105                     puntos_funcion_activa = [valor_interpolado]
106                     valor_interpolado = 0
107                 else:
108                     puntos_funcion_activa = []
109                 if (x_interpolada):
110                     x = x_interpolada
111                     xs_funcion_activa = [x_interpolada]
112                     x_interpolada = 0
113                 else:
114                     xs_funcion_activa = []
115
116                 while indice < parte_entera: #Leer las lineas de consumo
                        correspondiente a la funcion actual
117                     if indice_consumo == long_consumo:    #Finalizado
                            fichero de consumo?
118                         print("Fin anomalo fichero consumo, indice =",
                                indice_consumo, ", num_lineas_consumo =",
                                num_lineas_consumo)
119                         print("Linea salida:", indice_salida + 1)
120                         sys.exit()
121                     else:
122                         tupla_consumo =
                                lista_consumo[indice_consumo].strip().split(",",
                                2)
123                         valor_consumo = float(tupla_consumo[0]) *
                                float(tupla_consumo[1]) * AJUSTE_CONSUMO
124                         puntos_funcion_activa.append(valor_consumo)
125                         x += dx_modif
126                         xs_funcion_activa.append(x)
127                         lista_total_puntos.append(valor_consumo)
128                         indice_consumo  += 1
129                     indice += 1
130                 if parte_decimal:   #Tratamos la parte "sobrante"
131                     # Calculamos el punto "extra" mediante interpolacion y
                            lo anyadimos a la lista
132                     tupla_siguiente =
                            lista_consumo[indice_consumo].strip().split(",", 2)
133                     valor_siguiente = float(tupla_siguiente[0]) *
                            float(tupla_siguiente[1]) * AJUSTE_CONSUMO
134                     valor_interpolado = valor_consumo + parte_decimal *
                            (valor_siguiente - valor_consumo)
135                     x_interpolada = x + dx_modif * parte_decimal
```

```
136                     puntos_funcion_activa.append(valor_interpolado)
137                     xs_funcion_activa.append(x_interpolada)
138             # Se han guardado las muestras correspondientes a la funcion
                    actual, se llama al metodo de integracion y se asigna a la
                    funcion del tope de la pila
139                 simps_funcion_activa = simps(puntos_funcion_activa,
                        x=xs_funcion_activa, even="avg")
140                 trapz_funcion_activa = trapz(puntos_funcion_activa,
                        x=xs_funcion_activa)
141                 tiempo_funcion_activa = int(tupla_nueva[2]) -
                        int(tupla_vieja[2])
142                 # Cambio de intervalo?
143                 if (int(tupla_nueva[2]) - primer_tsc) > ((intervalo + 1) *
                        ciclos_intervalo) and (intervalo < (NUM_INTERVALOS -
                        1)):
144                     ratio = (ciclos_intervalo * (intervalo + 1) -
                            int(tupla_vieja[2])) / (int(tupla_nueva[2]) -
                            int(tupla_vieja[2]))
145                     dict_list_simps[intervalo][tupla_vieja[0]] =
                            dict_list_simps[intervalo].get(tupla_vieja[0], 0) +
                            (simps_funcion_activa * ratio)
146                     dict_list_trapz[intervalo][tupla_vieja[0]] =
                            dict_list_trapz[intervalo].get(tupla_vieja[0], 0) +
                            (trapz_funcion_activa * ratio)
147                     dict_list_tiempo[intervalo][tupla_vieja[0]] =
                            dict_list_tiempo[intervalo].get(tupla_vieja[0], 0)
                            + (tiempo_funcion_activa * ratio)
148                     intervalo += 1
149                     dict_list_simps[intervalo][tupla_vieja[0]] =
                            dict_list_simps[intervalo].get(tupla_vieja[0], 0) +
                            (simps_funcion_activa * (1 - ratio))
150                     dict_list_trapz[intervalo][tupla_vieja[0]] =
                            dict_list_trapz[intervalo].get(tupla_vieja[0], 0) +
                            (trapz_funcion_activa * (1 - ratio))
151                     dict_list_tiempo[intervalo][tupla_vieja[0]] =
                            dict_list_tiempo[intervalo].get(tupla_vieja[0], 0)
                            + (tiempo_funcion_activa * (1 - ratio))
152                 else:
153                     dict_list_simps[intervalo][tupla_vieja[0]] =
                            dict_list_simps[intervalo].get(tupla_vieja[0], 0) +
                            simps_funcion_activa
154                     dict_list_trapz[intervalo][tupla_vieja[0]] =
                            dict_list_trapz[intervalo].get(tupla_vieja[0], 0) +
                            trapz_funcion_activa
155                     dict_list_tiempo[intervalo][tupla_vieja[0]] =
                            dict_list_tiempo[intervalo].get(tupla_vieja[0], 0)
                            + tiempo_funcion_activa
156
157             if tupla_nueva[1] == "1": #Es un 1 = RET -> Desapilo y guardo
                    como tupla_vieja (=activa) la que estuviera en el tope de
                    la pila (Si es que habia alguna)
158                 long_pila = len(pila)
159                 if not long_pila:
160                     print("Error en la secuencia de Calls/Rets")
161                 pila.pop()
162             else:       #Es un 0 = CALL -> Apilo y actualizo el valor de la
                    funcion a tratar
163                 pila.append(tupla_nueva)
164             tupla_vieja = tupla_nueva
165         indice_salida += 1
166
167     if lista_total_puntos:
```

```
168         consumo_total_simps = simps(lista_total_puntos, dx=dx_modif,
                even="avg")
169         print("Consumo total obtenido por el metodo de Simpson es",
                consumo_total_simps)
170         consumo_total_trapz = trapz(lista_total_puntos, dx=dx_modif)
171         print("Consumo total obtenido por el metodo del trapecio es",
                consumo_total_trapz)
172
173     else:
174         print("No hay puntos en la lista total de puntos.")
175
176     for i in range(NUM_INTERVALOS):
177         print("======================")
178         print("DATOS DEL INTERVALO", i+1)
179         print("======================")
180         suma_consumos_simps = sum(dict_list_simps[i].values())
181         suma_consumos_trapz = sum(dict_list_trapz[i].values())
182         suma_tiempos = sum(dict_list_tiempo[i].values())
183
184         print("Consumo listulado Simpson:", dict_list_simps[i])
185         print("Suma de consumos Simpson:", suma_consumos_simps)
186         print("Consumo acumulado trapezio:", dict_list_trapz[i])
187         print("Suma de consumos Trapezio:", suma_consumos_trapz)
188
189         print("Tiempo acumulado:", dict_list_tiempo[i])
190         print("Suma de tiempos:", suma_tiempos)
191
192     log.write("\nSimpson")
193     for i in range(NUM_INTERVALOS):
194         s = "\nIntervalo " + str(i + 1)
195         for linea_funcion in lista_funciones:
196             tupla_funcion = linea_funcion.strip().split(":", 2)
197             if tupla_funcion[0]:
198                 s += "\t" + str(dict_list_simps[i].get(tupla_funcion[0],0))
199         log.write(s)
200
201     log.write("\nTrapecio")
202     for i in range(NUM_INTERVALOS):
203         s = "\nIntervalo " + str(i + 1)
204         for linea_funcion in lista_funciones:
205             tupla_funcion = linea_funcion.strip().split(":", 2)
206             if tupla_funcion[0]:
207                 s += "\t" + str(dict_list_trapz[i].get(tupla_funcion[0],0))
208         log.write(s)
209
210     log.write("\nTiempo")
211     for i in range(NUM_INTERVALOS):
212         s = "\nIntervalo " + str(i + 1)
213         for linea_funcion in lista_funciones:
214             tupla_funcion = linea_funcion.strip().split(":", 2)
215             if tupla_funcion[0]:
216                 s += "\t" + str(dict_list_tiempo[i].get(tupla_funcion[0],0))
217         log.write(s)
218
219 if __name__ == "__main__":
220     main()
```

codigos/trata_datos_intervalos.py

Anexo F

Intel PTU: Análisis de BZIP2

Function	Clockticks	Instructions Retired	Process
BZ2_blockSort	35.040	12.177	bzip2_base.i386
fallbackSort	12.332	2.287	bzip2_base.i386
BZ2_bzDecompress	11.679	2.164	bzip2_base.i386
mainGtU	10.922	8.513	bzip2_base.i386
BZ2_compressBlock	10.203	7.523	bzip2_base.i386
BZ2_decompress	9.001	5.641	bzip2_base.i386
handle_compress	3.301	3.442	bzip2_base.i386
¡unknown(s)¿	527	43	bzip2_base.i386
memcpy	252	3	bzip2_base.i386
memset	232	0	bzip2_base.i386
BZ2_hbMakeCodeLengths	180	105	bzip2_base.i386
main	9	0	bzip2_base.i386
spec_getc	8	1	bzip2_base.i386
spec_init	7	3	bzip2_base.i386
BZ2_hbCreateDecodeTables	6	13	bzip2_base.i386
BZ2_bzWrite	3	0	bzip2_base.i386
BZ2_hbAssignCodes	3	6	bzip2_base.i386
BZ2_bzRead	2	0	bzip2_base.i386
skge_intr	2	0	bzip2_base.i386
compressStream	2	1	bzip2_base.i386
ext3_get_blocks_handle	1	0	bzip2_base.i386
_IO_printf	1	0	bzip2_base.i386
spec_fwrite	1	0	bzip2_base.i386
uncompressStream	1	1	bzip2_base.i386

Tabla F.1: Resultados de IntelPTU para BZI2

Anexo G

Análisis intermedios sobre BZIP2

Tras mejorar el script de tratamiento de datos para que manejara intervalos en vez de la función en su totalidad, se fue aumentando progresivamente el número de intervalos, desde 1 (tratamiento sin intervalos) hasta 200.

A continuación se muestran las diversas pruebas realizadas antes de la prueba final (cuyos resultados ya se han mostrado en las figuras 5.4, 5.5 y 5.6).

En cada una de las siguientes secciones se muestran los resultados obtenidos cambiando el número de intervalos en los que dividir el tiempo de ejecución del programa.

G.1. Para 20 intervalos

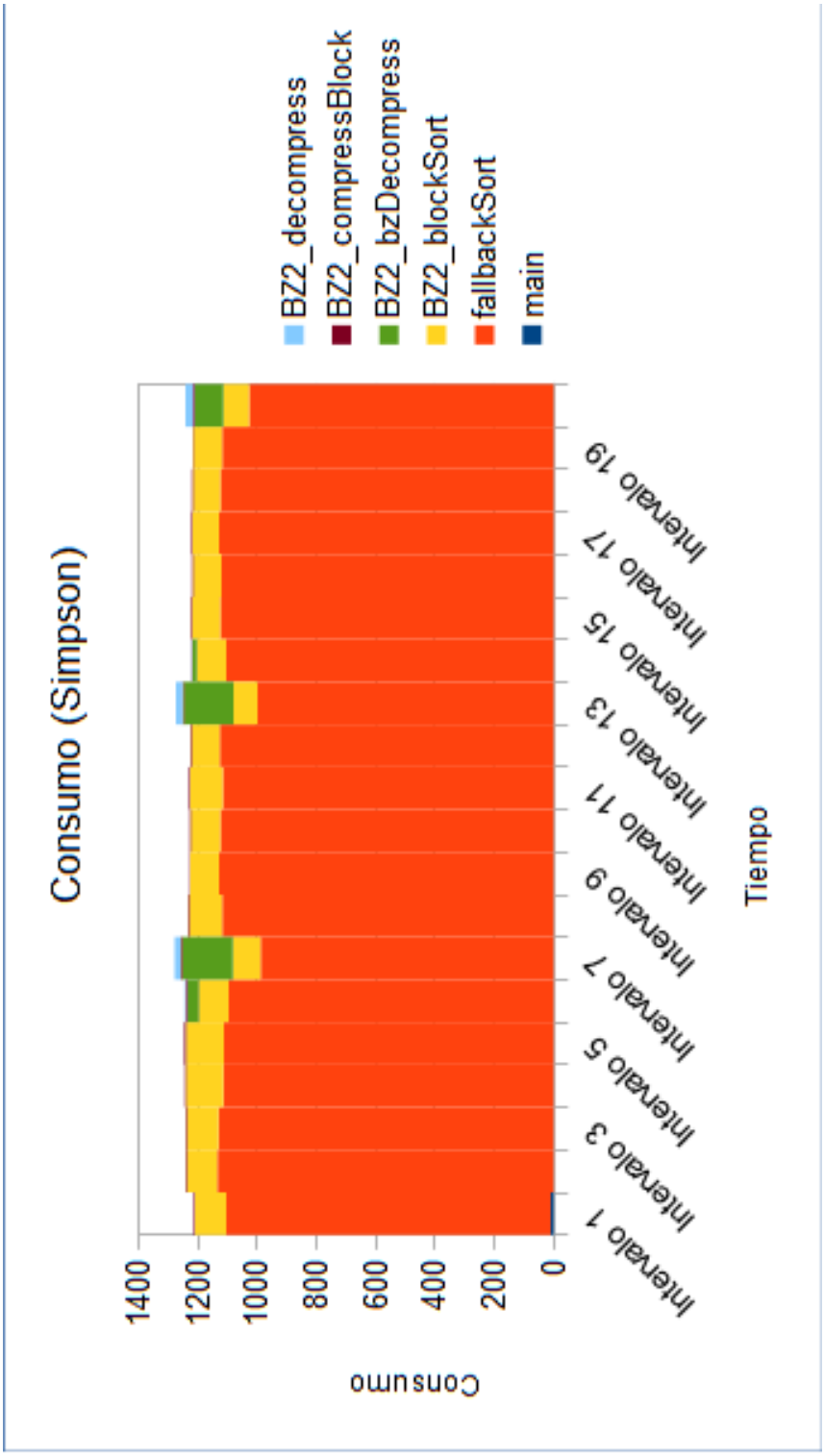

Figura G.1: BZIP2 20 intervalos: Consumo obtenido mediante el método de Simpson

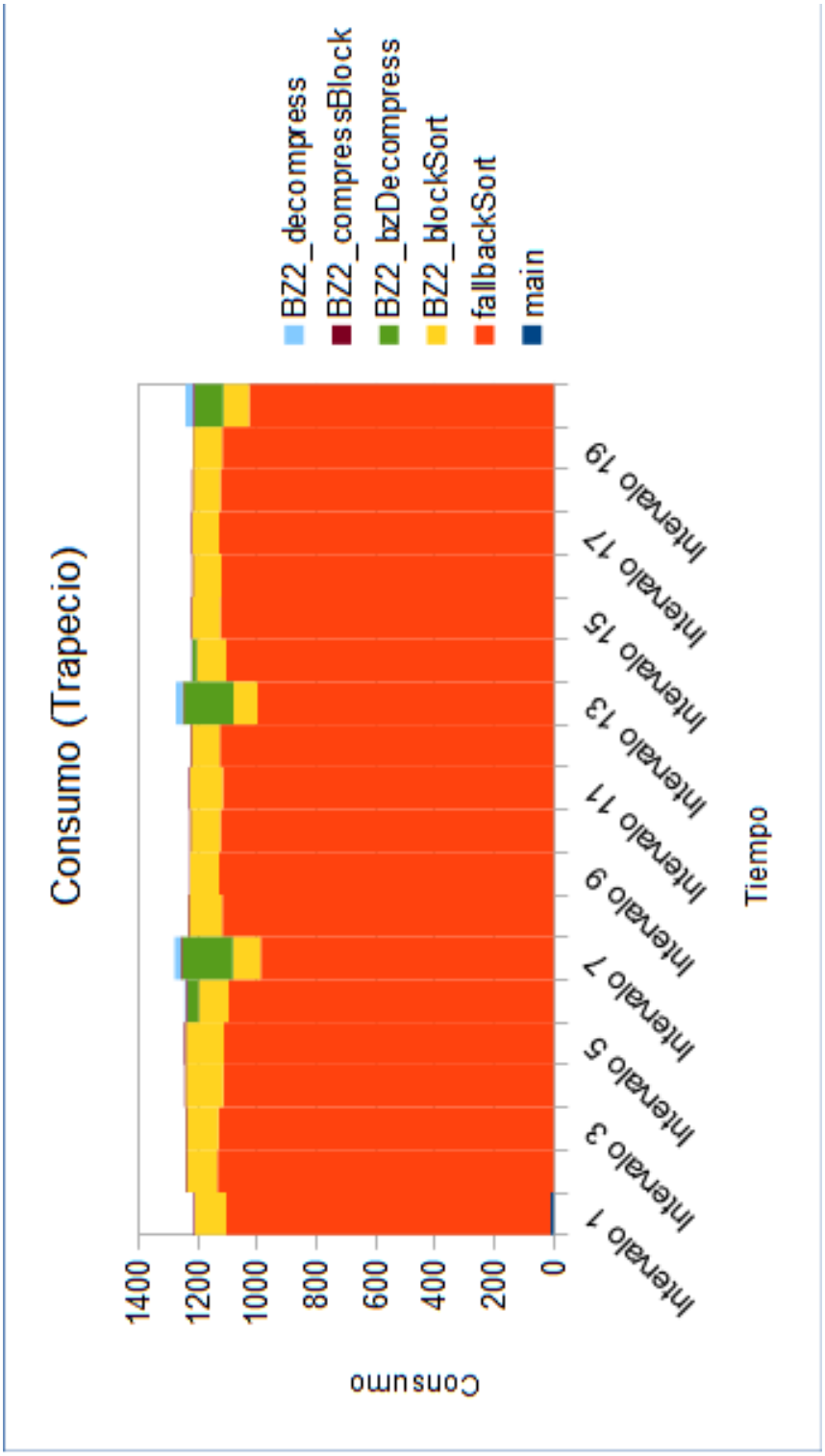

Figura G.2: BZIP2 20 intervalos: Consumo obtenido mediante el método del trapecio

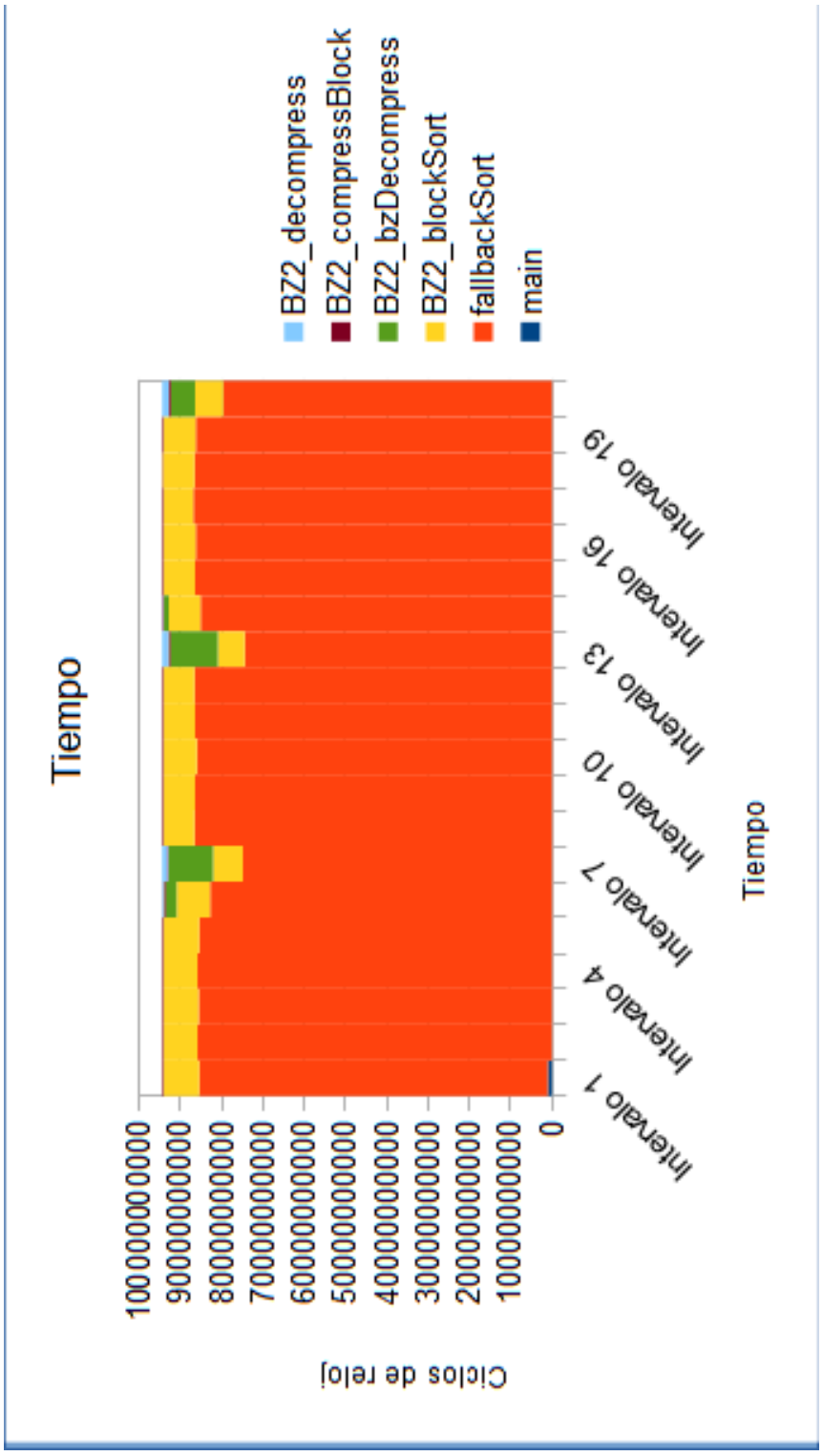

Figura G.3: BZIP2 20 intervalos: Reparto de tiempo

G.2. Para 50 intervalos

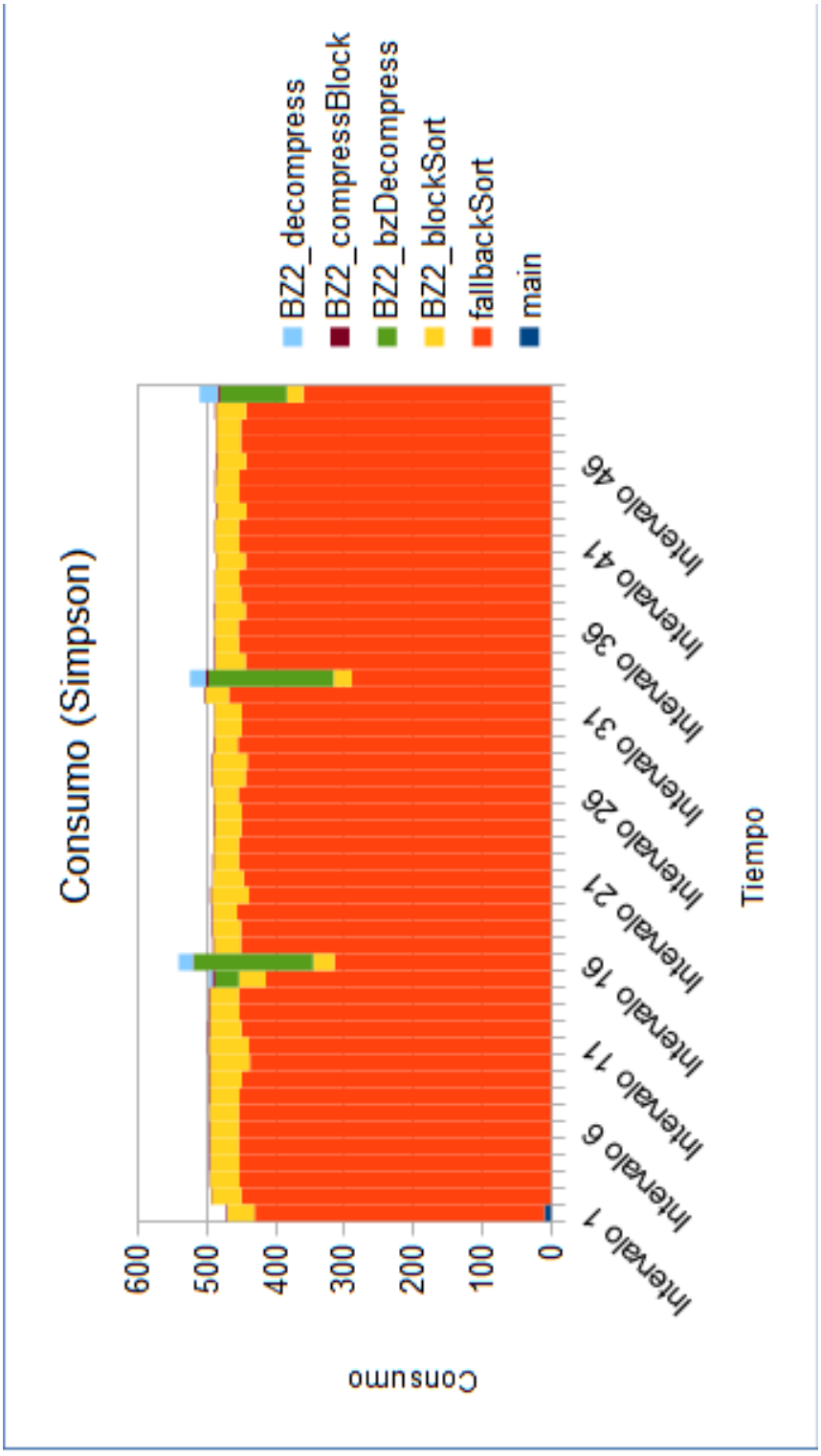

Figura G.4: BZIP2 50 intervalos: Consumo obtenido mediante el método de Simpson

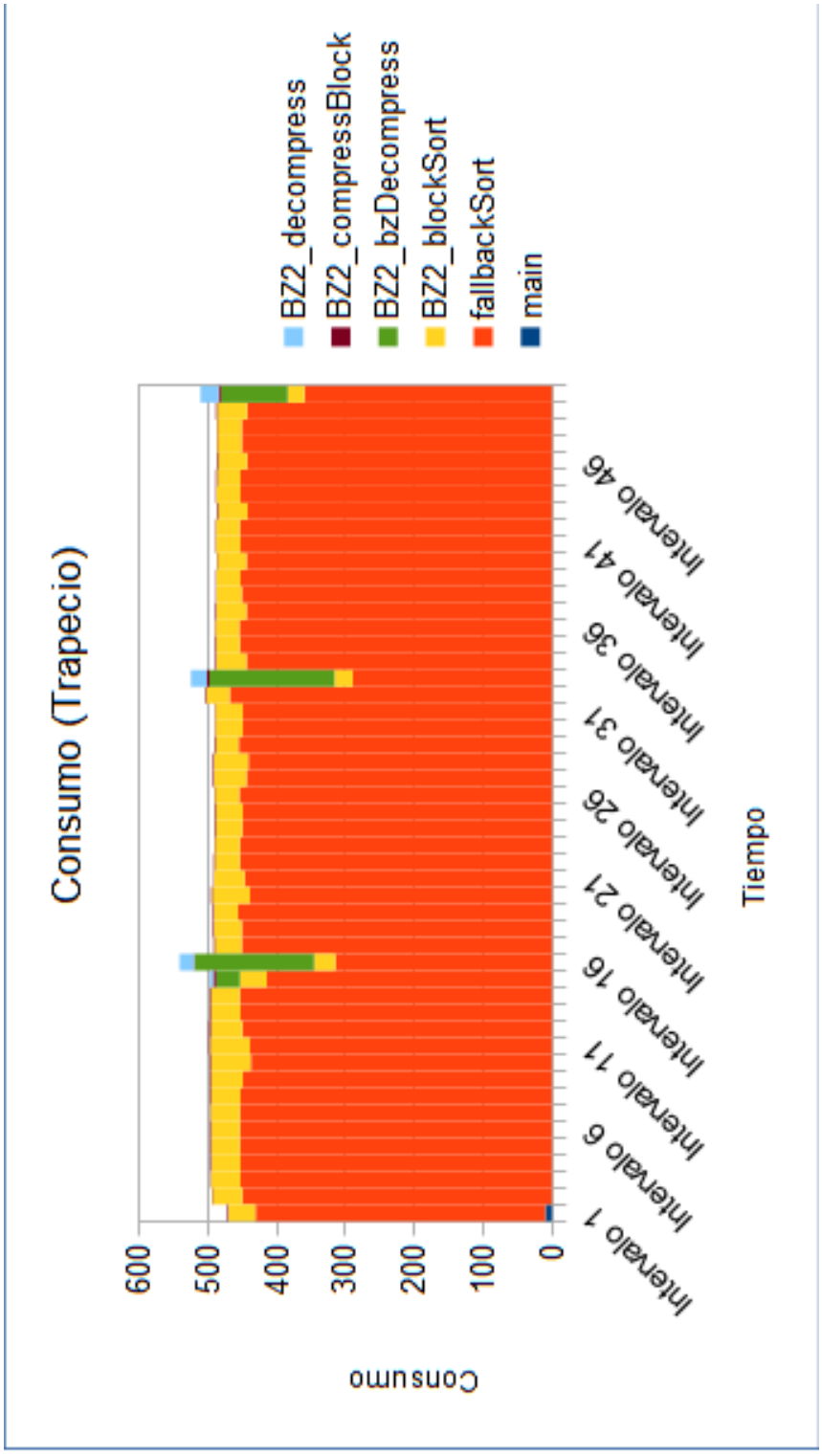

Figura G.5: BZIP2 50 intervalos: Consumo obtenido mediante el método del trapecio

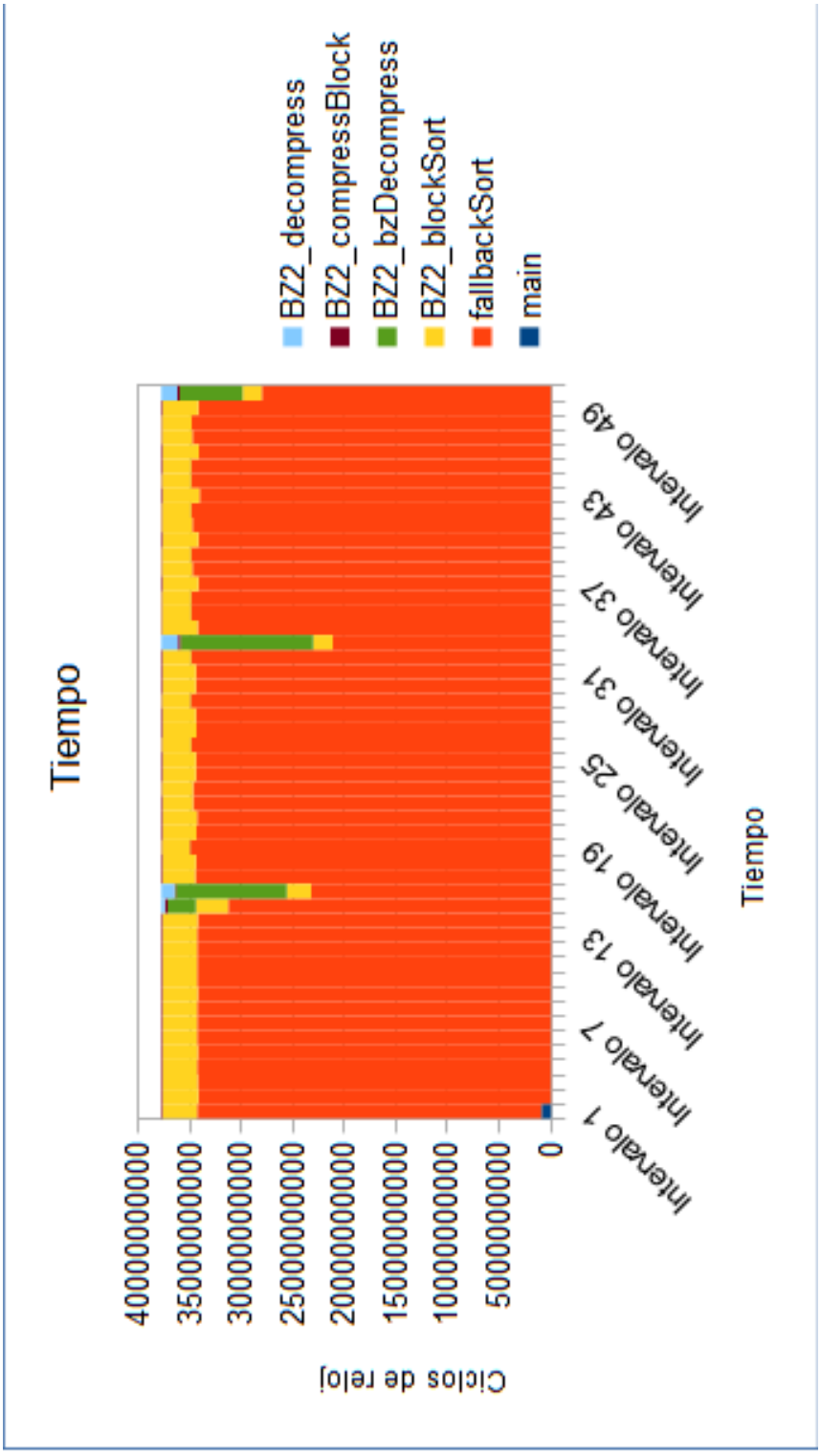

Figura G.6: BZIP2 50 intervalos: Reparto de tiempo

G.3. Para 100 intervalos

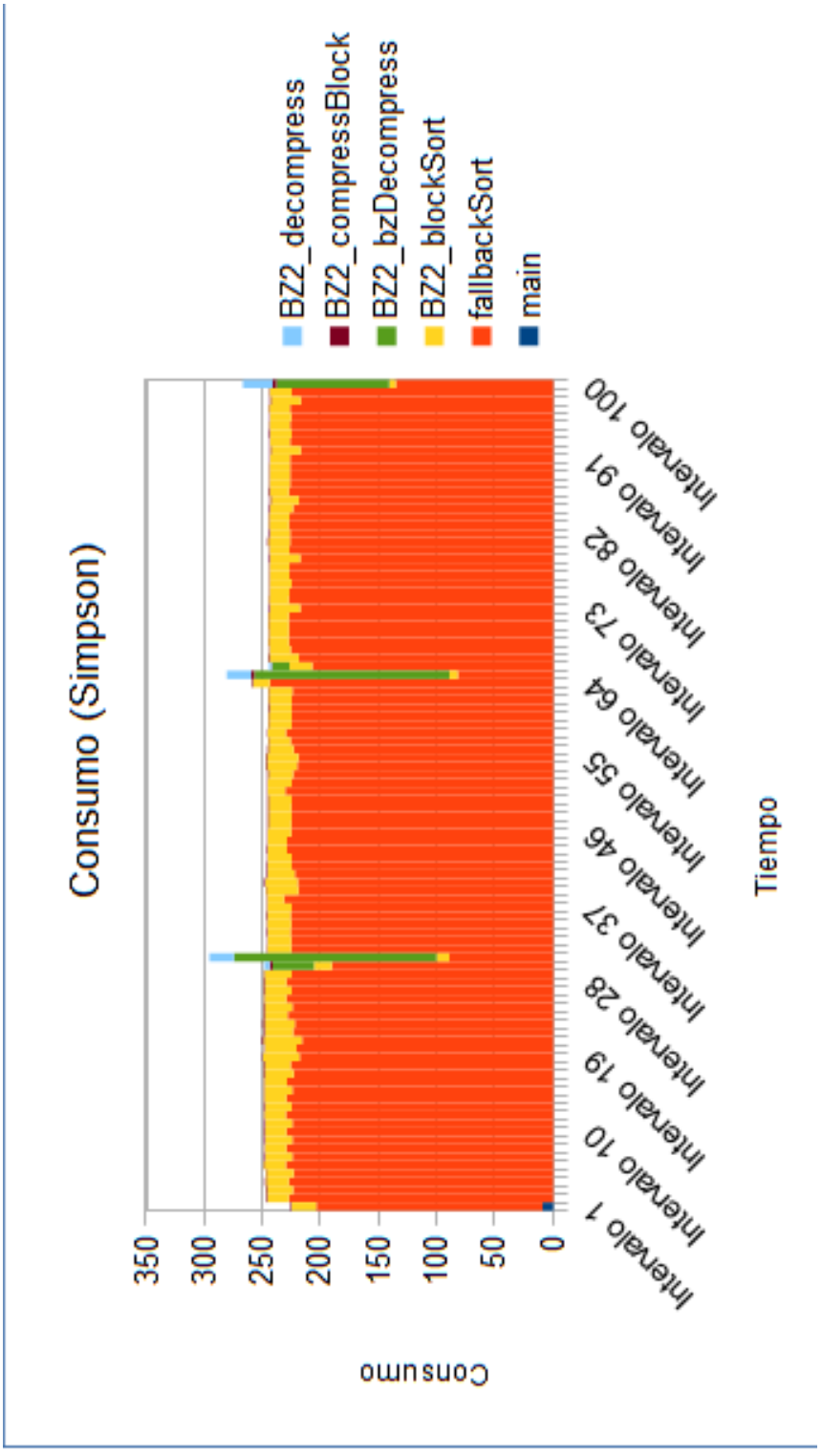

Figura G.7: BZIP2 100 intervalos: Consumo obtenido mediante el método de Simpson

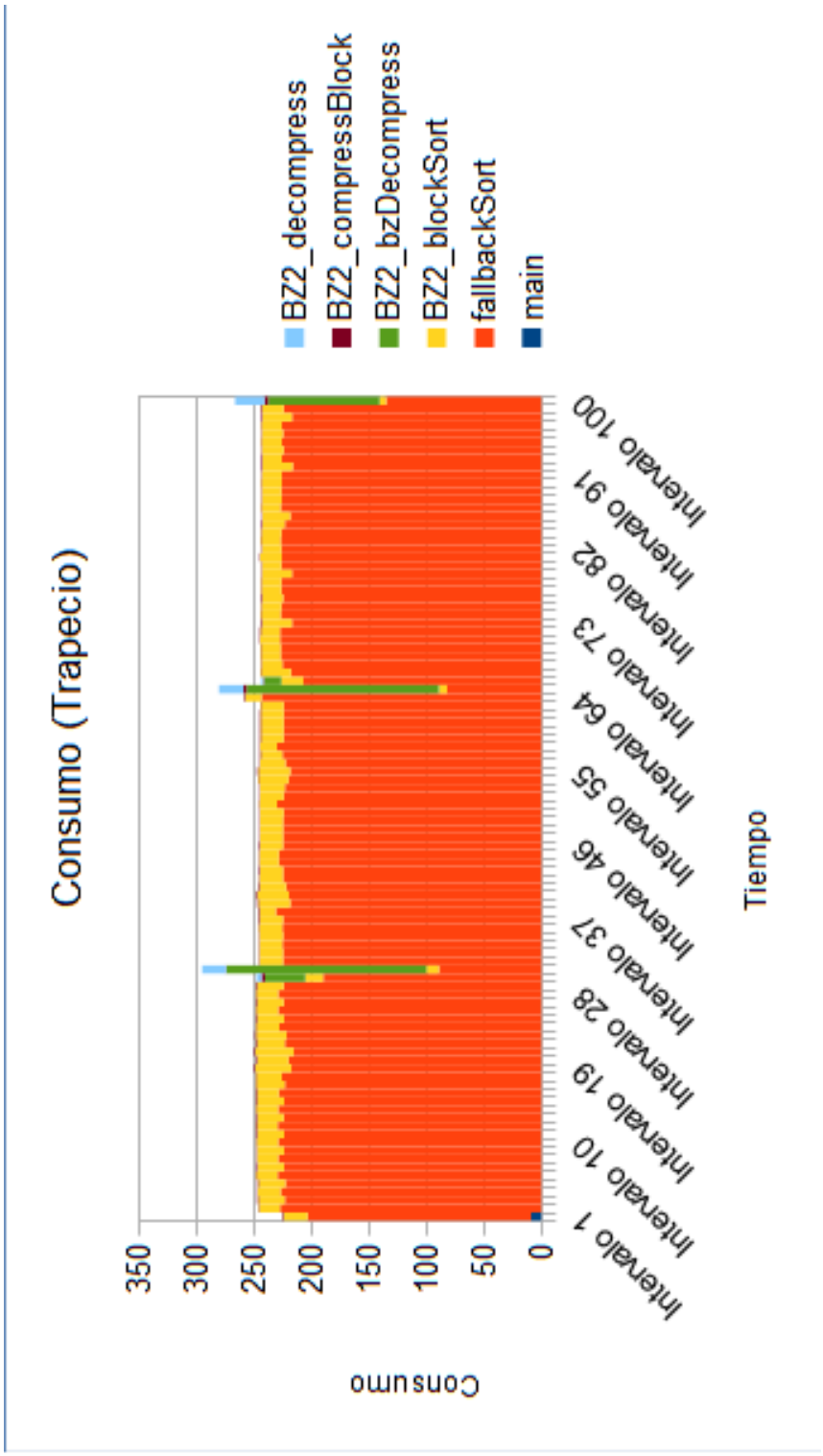

Figura G.8: BZIP2 100 intervalos: Consumo obtenido mediante el método del trapecio

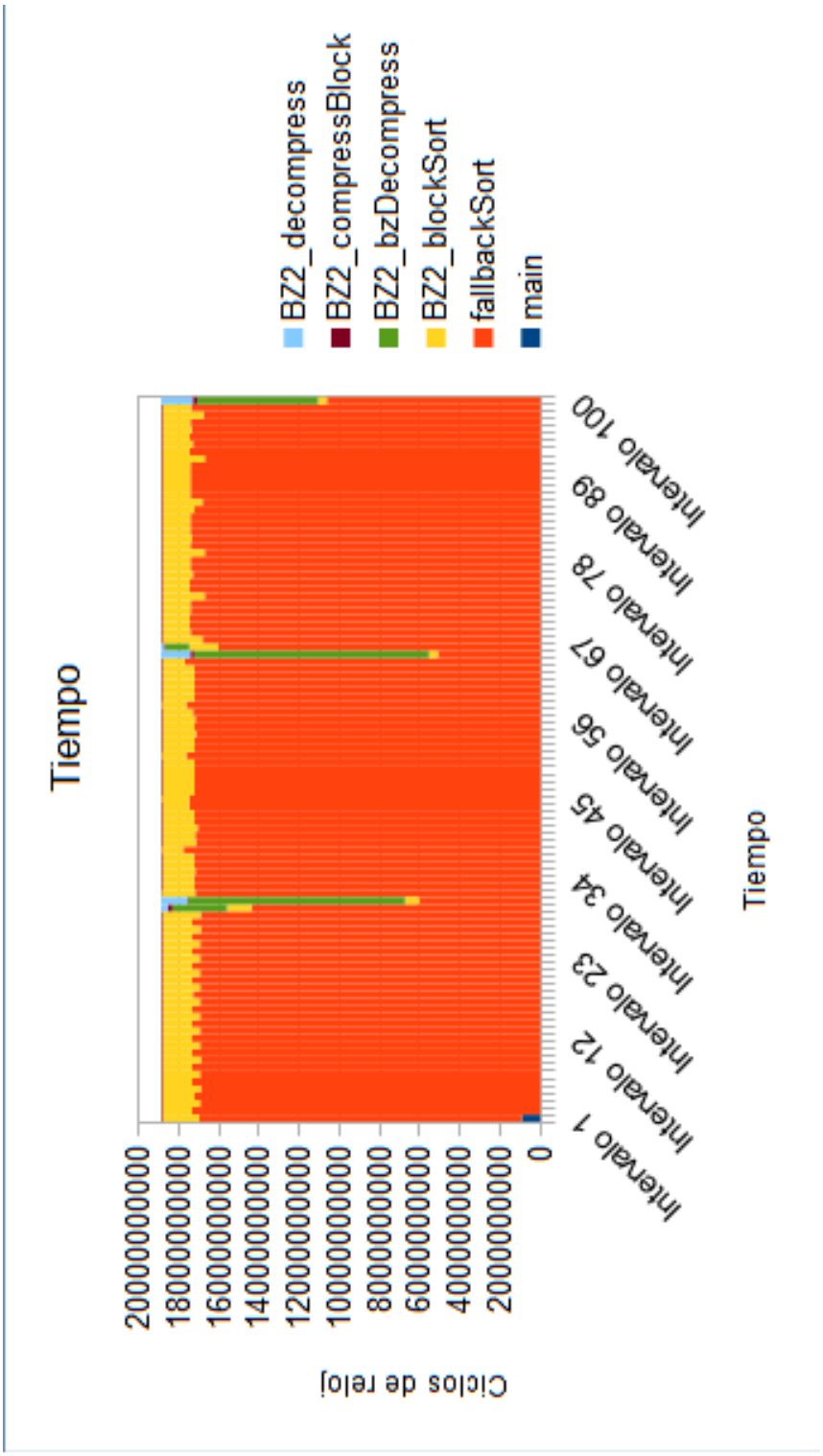

Figura G.9: BZIP2 100 intervalos: Reparto de tiempo