Universidad Zaragoza
1542

Trabajo Fin de Máster

Técnicas de control avanzadas para gestionar la temperatura en multiprocesadores

Advanced control techniques for managing temperature in multiprocessors

Autor/es

Pablo Hernández Almudi

Director/es

Eduardo Montijano Muñoz
Darío Suárez Gracia

Escuela de Ingeniería y Arquitectura

2019

Muchas gracias Eduardo
Muchas gracias Darío

Porque pocas palabras se pueden usar para expresar la gratitud que os debo y porque por muchas veces que os dé las gracias me seguirán pareciendo pocas, especialmente con la pieza que os ha tocado

Muchas gracias a profesores y compañeros que tantas veces han ayudado y han convertido el paso por la universidad en una experiencia inolvidable

Gracias Ana Cris, Jose María y Jesús

Gracias en especial a las abejas del gaZ que han hecho este viaje mucho más ameno

Y gracias a mi familia y amigos porque sé que lidiar con un informático no es sencillo

TÉCNICAS DE CONTROL AVANZADAS PARA GESTIONAR LA TEMPERATURA EN MULTIPROCESADORES - RESUMEN

Los sistemas ciberfísicos y el internet de las cosas están tomando una relevancia mayor en nuestro día a día, y su capacidad de cálculo se incrementa a la par que su utilidad. Esto causa problemas cuando se trata de disipar toda esa energía generada en entornos donde no se pueden aplicar técnicas de disipación activas y las exigencias de diseño impiden una correcta disipación pasiva. Es en este tipo de entornos donde nuestro trabajo está centrado.

Este proyecto aúna conocimientos de ingeniería informática y de ingeniería de control, y presenta una metodología para la comprensión de la disipación de un procesador y una propuesta de control para el mantenimiento de una temperatura constante mediante el uso de escalado dinámico de la frecuencia del procesador.

El trabajo comienza haciendo un repaso del estado del arte actual en materia de gestión de temperaturas y energía. Tras lo que se presenta el esquema de control que se realizará y como se ha ampliado mediante el uso de reconocimiento de cargas de trabajo, lo que permite realizar un ajuste más preciso del control. Se da la metodología a usar para poder ajustar el control y realizar las distintas pruebas adaptadas para cada caso de uso. Se explica como se realiza la implementación en el sistema operativo del control y se realizan pruebas de comportamiento y evaluación.

Con esto, las aportaciones principales de este proyecto son, el diseño de una metodología que permite entender como es el comportamiento de la temperatura de un procesador en base a la carga de trabajo y frecuencia. El diseño de una arquitectura de control con un supervisor que ajusta el control en base al tipo de carga del trabajo. Una serie de pruebas y programas que permiten realizar la caracterización y obtener datos relevantes de la ejecución. Y la implementación de dos módulos de kernel que permiten clasificar la carga de trabajo realizada y realizar el control de la temperatura.

Todo esto se ha evaluado usando una plataforma moderna con software ampliamente utilizado en sistemas *ciberfísicos* e internet de las cosas.

ÍNDICE GENERAL

1 INTRODUCCIÓN

1.1 MOTIVACIÓN

Los sistemas *ciberfísicos* y el internet de las cosas (IoT) están tomando cada vez una mayor relevancia en nuestro día a día. Estos sistemas, tales como coches autónomos, robots, monitores médicos o sistemas de vigilancia han visto incrementada su capacidad de cálculo lo que ha permitido que ejecuten cargas complejas, tales como aplicaciones de inteligencia artificial, que han incrementando notablemente su utilidad.

Obviando el debate sobre si es el software el que tiende a ocupar todo el hardware disponible, o bien, si es el hardware el que se intenta adaptar al software disponible [1]. Lo cierto es, que el incremento de la demanda de cálculo, implica un incremento en la cantidad de calor disipado, a la vez que una mayor demanda de energía. En entornos de cálculo clásicos tales como centros de cálculo, el calor no es siempre el problema principal, ya que mediante el uso de grandes disipadores y ventiladores expulsamos el calor, y con mayores fuentes de alimentación mantenemos la suficiente demanda energética. Por supuesto estas soluciones de disipación activas también implican que el costo económico y medioambiental es mucho mayor debido al requerimiento de un alto consumo. Por otro lado, en dispositivos más pequeños o en entornos con grandes restricciones térmicas el calor se convierte en un problema, sino el mayor de ellos. Por ejemplo, no querríamos que nuestro dron se cayera porque el sobrecalentamiento del procesador ha deteriorado la unidad o no le ha permitido detectar un obstáculo. También es famoso un caso reciente en el que la empresa *Apple* saco al mercado un ordenador que se ralentizaba por sobrecalentamiento [2].

Para solucionar estos problemas se encuentran soluciones de refrigeración pasivas que no requieren efectuar un trabajo para extraer la energía del sistema sino que lo que se hace es controlar la generación del calor. Y es que, las soluciones actuales de refrigeración pasiva se basan en actuar cuando el procesador empieza a sufrir lo que se conoce como *thermal throttling* o ahogamiento térmico. Es decir, cuando el procesador sufre temperaturas excesivamente altas se activan mecanismos de reducción de temperatura, que reducen el rendimiento. El más empleado dentro de los métodos de refrigeración pasiva es la reducción de frecuencia. La mayoría de las implementaciones se basan en heurísticas simples y puntos de activación, es decir, umbrales a partir de los cuales se empieza a reducir las frecuencias e incluso, de llegar a ser necesario, apagar por completo el procesador para evitar dañarlo.

Para tratar de mejorar las soluciones actuales nuestro trabajo presenta un nuevo control de frecuencia basado en teoría de control formal, cuyo objetivo es el mantenimiento de una temperatura constante a lo largo del tiempo. De este modo podremos asegurar una temperatura segura de funcionamiento que además causará un comportamiento más suave del rendimiento del procesador

dado que los cambios de frecuencia serán mucho más predecibles. Este control además trata de paliar una de las limitaciones con las que cuentan la soluciones basadas en control de temperatura, que no se adaptan bien a los tipos de carga de trabajo, para lo que se ha desarrollado una propuesta de mecanismo de supervisión que adapta el control a los diferentes tipos de cargas.

La solución propuesta se basa en tres observaciones. La primera, que las cargas de trabajo de los sistemas *ciberfísicos* son relativamente homogéneas a lo largo de su ejecución y además se pueden clasificar sus fases de ejecución de manera aproximada en tres tipos, cada uno con su propio comportamiento como se verá posteriormente. Segunda, que la relación entre la temperatura y la frecuencia puede ser descrita mediante una función de transferencia de primer orden. Y, por último, que el modelo puede ser utilizado para la creación de un *framework* de control que unifique los controles de temperatura y frecuencia en un supervisor unificado. Esto es opuesto a como funcionan las soluciones actuales en las que las decisiones se toman individualmente sin el conocimiento del otro debido a que son soluciones legadas dentro de los sistemas operativos.

Teniendo esto en cuenta, las principales contribuciones de este trabajo incluyen una metodología para realizar la caracterización térmica de la carga ejecutada en la CPU. En base a esta se ha desarrollado un esquema de control en bucle cerrado *Single-Input Single-Output* (SISO) para controlar la dinámica y valor de la temperatura del procesador. De este modo se previenen temperaturas excesivamente altas y la restricción de la frecuencia debido a alarmas térmicas. Esta metodología de caracterización y posterior aplicación en un control puede ser fácilmente aplicada a diferentes configuraciones de procesador y carga de trabajo reduciendo el trabajo de optimización manual que se hace para cada procesador en general. Esta estrategia de control se ha implementado tanto en espacio de usuario como driver dentro del *kernel* de Linux.

1.2 OBJETIVOS

El objetivo principal de este trabajo consiste en controlar la temperatura a la que opera un procesador haciendo uso del control de frecuencias. Para alcanzar este objetivo se plantean varios subobjetivos más pequeños.

Primero, debemos entender como se genera el calor y realizar un modelado del comportamiento teniendo en cuenta distintas cargas de trabajo y modelos de procesador. De este modo podremos ver como afecta la cantidad de núcleos al procesador, la frecuencia y el tipo de trabajo que tenga que realizar.

Segundo, se realizara un análisis de las cargas de trabajo que se ejecutan para tratar de entender que partes del procesador son las que más influyen en la generación de calor. Así también podremos realizar una clasificación en tiempo de ejecución del tipo de programa para realizar un control más ajustado sobre lo que se está ejecutando.

Tercero, proponer, diseñar y estudiar controles automáticos que permitan controlar la temperatura mediante la frecuencia usando para ello los modelos de comportamiento previamente estudiados. Este control se deberá implementar tanto en simulación para realizar experimentos sobre su diseño, como en espacio

de usuario para probarlo en un sistema real, además de como un módulo de *kernel* que permita llevar el control a un entorno de desarrollo.

Por último, se evaluará el control realizado para comprobar empíricamente su validez. Además, se evaluará también en cuestión de rendimiento mediante la característica más representativa de cada carga de trabajo. Todo esto además se comparará con técnicas actuales para ver como varían entre ellas.

1.3 ALCANCE Y CONTRIBUCIONES

Este trabajo parte de un TFG [9] realizado por el mismo autor en el que también se estudia el control de la temperatura por medio de la frecuencia. En este trabajo el control presentado era un esquema sencillo en realimentación con un controlador PID (Proporcional-integral-derivativo). Se realizaba un modelado simple de la respuesta de la temperatura a los cambios de frecuencia, por medio de escalones de frecuencia únicos, sin tener en cuenta no linealidades. Y las pruebas y evaluaciones realizadas se realizaban con un único tipo de carga de trabajo. El trabajo previo y parte de este han sido presentados como póster en la *ARM Research Summit*, en la que además fue seleccionado para hacer una pequeña presentación oral.

Los aportes principales de este trabajo son una metodología más completa y correcta para realizar el modelado de la respuesta temperatura-frecuencia. El estudio y aporte de distintos esquemas de control más complejos y robustos. Se han identificado y usado 3 tipos de cargas de trabajo que tienen un comportamiento térmico diferenciado. Para mejorar el comportamiento del controlador en respuesta al tipo de carga se aporta también un supervisor del control que lo adapta en función de la ejecución.

La plataforma usada también ha cambiado, de una con un procesador con cuatro núcleos se ha pasado a una más moderna con ocho. Esto no solo se nota en el incremento del rendimiento sino también en una mayor producción del calor, lo que ha causado que el comportamiento térmico sea más errático y el control deba ser más rápido y agresivo.

En el modelado y diseño del control se ha vuelto a usar las herramientas *Matlab* y *Simulink*. Y en este caso además se ha realizado una implementación propia para simular el esquema de control en Python con el fin de simplificar la implementación de varias características especiales que hemos probado. Se ha preferido esta solución al uso de *Simulink* dado que para la simulación de la planta se usa una función de transferencia sencilla y por tanto simple en su implementación.

Para las cargas de trabajo se ha reimplementado el benchmark de multiplicación de matrices para simular aplicaciones con gran cálculo numérico como las convoluciones empleadas en *Machine Learning*, y se han seleccionado hasta 6 programas distintos para realizar las pruebas y evaluación.

Uno de los aportes nuevos y más importantes es el de modificar el control en base al tipo de programa que se esté ejecutando. Para hacer esto se ha realizado un estudio de las cargas de trabajo y una propuesta de clasificación de bajo coste computacional mediante el uso de los contadores hardware que se encuentran en el procesador. Este estudio usa el lenguaje de programación estadístico *R*

y los conocimientos aprendidos en la asignatura de estadística tanto de grado como de máster.

Como resultado de este trabajo se han realizado dos módulos de *kernel* que tienen implementados, la clasificación de programas y el control de la frecuencia. El módulo de clasificación de programas permite a los usuarios y al resto de módulos el acceso a estos contadores y a conocer el tipo de programa.

Como resultado de este trabajo se ha logrado mantener una temperatura constante a lo largo de diversas cargas de trabajo. En cargas de trabajo con una alta variabilidad, es decir aquellas muy adversas para el controlador, se ha conseguido una reducción de un 8 % de la temperatura con una perdida de rendimiento entre el 7 y el 13 % para cargas uni y multinúcleo, respectivamente. Por otro lado, en cargas de trabajo homogéneas se consigue mejorar el rendimiento en hasta un 6 % con la temperatura máxima.

En paralelo a este trabajo se ha asistido como oyente a la asignatura del máster de ingeniería industrial de *Diseño electrónico y control avanzado*. Gracias a la cual se han podido comprender los mecanismos más complejos de control que han sido aplicados en este trabajo. Además se ha contribuido en el repositorio público de una herramienta ampliamente usada solucionando un error encontrado.

En la figura 1.1 se muestra el diagrama de Gantt con la distribución de esfuerzos realizados para las distintas partes.

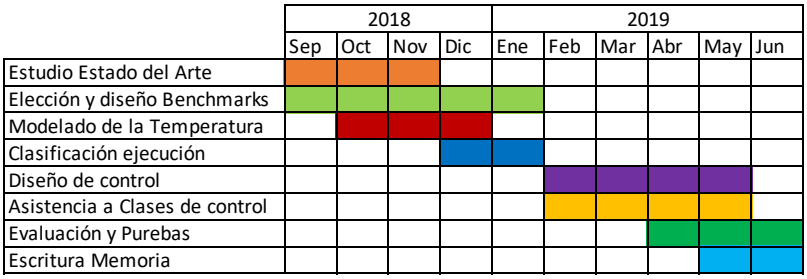


Figura 1.1: Diagrama Gantt

1.4 ORGANIZACIÓN

El documento se encuentra dividido en 8 capítulos. Tras la introducción en el capítulo 2 se habla del estado del arte actual sobre la gestión de la temperatura, tanto en la literatura como en los sistemas actuales. Además se da una explicación sobre como se genera y disipa la temperatura. En el capítulo 3 se explica cual es la estrategia de control seguida, identificación del sistema, arquitectura e implementación algorítmica del control. El capítulo 4 amplía el control propuesto introduciendo nuestra propuesta de clasificación de programas. El capítulo 5 introduce la metodología que se ha seguido a la hora de realizar las pruebas del sistema, tanto hardware como software. A continuación en el capítulo 6 se discute como se ha implementado la propuesta bajo un sistema operativo actual y los desafíos que supone. Capítulo 7, son los resultados obtenidos, desde modelado del sistemas a evaluación con comparaciones con otras soluciones para comprobar su comportamiento. El último capítulo 8 concluye y describe las posibles lineas de trabajo futuro.

2 ESTADO DEL ARTE

En este capítulo se trata el estado del arte y soluciones actuales en cuestión de gestión de temperatura y energía en multiprocesadores y se hace un estudio preliminar sobre la generación y disipación del calor en multiprocesadores.

2.1 ESTADO DEL ARTE

El control del temperatura ha suscitado mucho interés en la academia generando un gran número de propuestas. Por un lado, tenemos soluciones basadas en hardware tanto pasivas como activas, como disipadores dedicados, o el uso de otros elementos del dispositivo como la batería para disipar el exceso de calor, ventiladores para forzar la entrada de aire o refrigeración líquida. Yueh [26] propone un sistema de refrigeración por agua que prueba consumir menos que otras técnicas mientras logra mantener un alto desempeño. Aunque estas técnicas mitigan el problema de la disipación tienen un coste extra que no siempre es posible.

Por otro lado, tenemos las soluciones software pasivas, las cuales, están mejor enfocadas en los dispositivos *ciberfísicos*. Entre ellas podemos distinguir entre soluciones basadas en heurísticas y soluciones que consideran herramientas de ingeniería de control. Entre el primer tipo encontramos técnicas que se enfocan en el consumo de energía. Park [18] propone priorizar las tareas que se ejecutan en primer plano, parando incluso las tareas de fondo cuando el consumo es excesivamente alto. Esto por supuesto supone que el sistema deja de hacer cosas que pueden ser importantes por lo que estaríamos perdiendo la utilidad que tiene el dispositivo. CoScale [4] mejora el consumo en aplicaciones con muchos accesos a memoria mediante la coordinación de las frecuencias tanto de la CPU como de la memoria. Pese a ser una buena solución, solo se centra en ese tipo de aplicaciones mientras que la solución que proponemos en este trabajo busca poder adaptarse a todo tipo de tareas.

Algunas soluciones basadas en control son las propuestas por Pothukuchi [19], usando un controlador MIMO (multiple-input multiple-output), e Isci [12], proponiendo distintas propuestas que buscan el mejor compromiso entre rendimiento y consumo. Maggio [16] propone un controlador PID enfocado en mantener una calidad de servicio (QoS) constante, mediante el uso de bibliotecas que insertan señales en los programas que se ejecutan. Las propuestas anteriores solo han sido probadas mediante simuladores, por lo que no han sido evaluadas en una plataforma real, mientras que la última propuesta no considera la temperatura, solo calidad de servicio.

Otras propuestas están más orientadas al mantenimiento de un buen QoS. La metodología principal consiste en ejecutar varios experimentos para comprender como cambia el QoS bajo ciertos parámetros para poder proponer distintos modelos de consumo y algoritmos de control. SPECTR [21] usa un controlador

MIMO basado en máquina de estados para mantener la QoS medida en imágenes por segundo a la par que se busca la mejor relación con el consumo. Rahmani [20] también propone un control retro-alimentado para proteger sistemas multi-core contra picos de energía usando un PID para realizar una estimación de la energía disponible. De manera similar, IPA de Arm [24] implementa un governor térmico basado en PID para restringir las frecuencias disponibles en base a un modelo de energía. Todos estos trabajos tienen en común el uso de la estimación de energía disponible para calcular un valor de frecuencia aceptable. En nuestra propuesta trabajamos directamente con la temperatura simplificando el problema de control evitando la necesidad de modelos de energía complejos.

Finalmente, el uso de teoría de control para la gestión de la temperatura ya ha sido explorado por Leva [15] mediante el uso de un esquema más sencillo con un controlador PI. Ellos prueban su propuesta usando ejecuciones de *LINPACK* en un Intel core I5-6600K con un TDP de 91W con una implementación realizada en espacio de usuario. Nosotros vamos a enfocarnos en procesadores móviles de bajo consumo que no tienen demandas tan grandes de energía ni de disipación. Además vamos a realizar el control dentro del kernel del sistema operativo simplificando su uso por los usuarios.

2.2 GENERACIÓN DE CALOR

Dada la naturaleza de la computación toda la energía que se introduce en un sistema se transforma en calor. Esto es debido a que un procesador no es más que un circuito de conmutación y por tanto, simplificando un poco, el consumo se puede dividir en dos tipos, estático y dinámico,

$$P_{total} = P_{est} + P_{dyn}. \tag{2.1}$$

El consumo estático es el causado por las fugas producidas en los transistores cuando no están conmutando. Estas fugas son producidas por la llamada corriente subumbral. Esta se produce cuando la tensión de entrada a un transistor no alcanza el valor de activación. Pese a que el transistor no está activado permite un pequeño paso entre la tensión de alimentación y tierra. En la figura 2.1 se puede observar el esquema de una puerta NOT en CMOS, con un transistor PMOS arriba y un NMOS en la parte de abajo. El condensador aparece porque los cables se suelen modelar como un circuito RC. Cuando la tensión de entrada V_{in} no activa los transistores, estos permiten una pequeña corriente entre V_{dd} y tierra. Por tanto el consumo estático puede verse como,

$$P_{est} = I_{sub} V_{dd}, \tag{2.2}$$

donde I_{sub} es la corriente subumbral y V_{dd} la tensión de alimentación. Por otro lado, la mayor fuente de calor de un procesador es la producida por las conmutaciones, también conocida como potencia dinámica. En la figura 2.1 podemos observar que cuando la entrada conmuta, el cable que actúa como condensador, se carga o descarga provocando la corriente dinámica. En un circuito que conmuta constantemente podemos expresar este consumo como,

$$P_{dyn} = C V_{dd}^2 f, \tag{2.3}$$

donde C es el valor de capacitancia, V_{dd} la tensión de alimentación y f la frecuencia de conmutación. Dado que la capacitancia varía dependiendo de que partes del procesador se estén usando lo que se hace es usar un valor medio para poder usar la variable como constante o se emplea la capacidad total con un factor de activación, que suele ser 0,2.

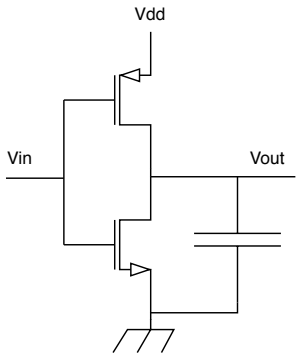


Figura 2.1: Esquema puerta NOT en CMOS

Teniendo en cuenta la manera en la que se consume la potencia y sabiendo que este consumo en un procesador se trasforma en calor, podemos obtener la siguiente fórmula donde P es ahora representado por la cantidad de energía transformada Q,

$$Q = Q_{dinámica} + Q_{estática} = CV_{dd}^2 f + I_{sub} V_{dd}. \tag{2.4}$$

Podemos observar como dentro de los términos de la ecuación el valor de la tensión es el que toma una mayor presencia. Esto es importante dado que la frecuencia es dependiente de la tensión, como explica Mudge [17] en donde se ve que la frecuencia máxima que se puede alcanzar es la dada por,

$$f_{max} \propto (V_{dd} - V_{th})/V_{dd}, \tag{2.5}$$

la cual es además aproximadamente lineal con respecto a la tensión de alimentación V_{dd} y la tensión de cambio de los transistores V_{th}. En el procesador lo único que podemos cambiar es la frecuencia, la tensión es modificada por el procesador mediante una tabla de relación frecuencia-temperatura a la que solo tiene acceso el procesador. Por tanto, si queremos incrementar la frecuencia y por tanto incrementar el rendimiento se deberá incrementar también la tensión. Dado que para que las señales puedan llegar a tiempo de un lugar a otro en un entorno en el que todos los elementos internos actúan como condensadores deberemos incrementar la tensión que requieren para su carga. Por tanto, incrementar la frecuencia del procesador implica incrementar al cuadrado la tensión dando como resultado un mayor incremento no lineal en el calor generado.

2.3 DISIPACIÓN DE CALOR

A la hora de disipar el calor generado existen dos variables importantes a tener en cuenta. Por un lado la cantidad de superficie que tengamos disponibles para expulsar el calor, sobre lo cual el diseño del dispositivo es un factor limitante, y

por otro lado, algo sobre lo que no tenemos influencia, sobretodo en dispositivos ciberfísicos como es la temperatura exterior. Cuanta mayor sea la diferencia de temperaturas entre el exterior y el procesador, mejor se podrá disipar el calor.

La disipación del calor se rige por la ecuación:

$$Q = mC_p \frac{dT_{\mu P}}{dt} + \frac{T_{\mu P} - T_{air}}{R_t} \tag{2.6}$$

donde:

- Q es la potencia que se disipa en forma de calor (W)
- m es la masa total del procesador (Kg)
- C_p es el calor especifico, medida como la cantidad de energía que hay que suministrar a una cantidad de masa para incrementar un grado su temperatura ($\frac{J}{^\circ CKg}$)
- T_{air} es la temperatura del aire ($^\circ C$)
- $T_{\mu P}$ es la temperatura del procesador ($^\circ C$)
- $\frac{dT_{\mu P}}{dt}$ es la derivada de la temperatura del procesador ($^\circ C$)
- R_t es la resistencia térmica entre dos medios, en este caso entre el procesador y el aire, medida como la diferencia de temperatura entre dos medios cuando una unidad de potencia los atraviesa ($\frac{^\circ Cs}{J}$)

La primera parte de la fórmula lo que se describe es el procesador como tal, junto con el disipador. Y en la segunda parte, se describe la expulsión al exterior de la temperatura. Para incrementar la disipación se pueden usar técnicas hardware y software. Mediante hardware podemos usar disipadores más grandes para incrementar la masa, así como una mayor superficie para disminuir la resistencia térmica, también se pueden usar mejores materiales para reducir el calor específico. Con el uso de ventiladores lo que haríamos sería incrementar el flujo de aire para poder sustituir el aire caliente cerca del disipador por aire más frío para incrementar la diferencia de temperatura. Mediante software podemos actuar sobre la parte de la fórmula que genera el calor, en concreto, el término $\frac{dT_{\mu P}}{dt}$. Es aquí donde nosotros actuaremos para conseguir un mejor control de la temperatura.

2.4 GESTIÓN DE TEMPERATURA Y FRECUENCIA EN LINUX

Esta sección describe la gestión de la temperatura en Linux dado que es el sistema operativo que vamos a utilizar, es ampliamente usado en la práctica totalidad de los dispositivos *ciberfísicos* y es código abierto, por lo que podemos realizar modificaciones y recompilarlo sin problemas.

2.4.1 *Gestión de temperatura*

El sistema que monitoriza la temperatura se conoce como *thermal governor*. La tarea de este sistema es muy simple, comprueba la temperatura del dispositivo,

y en base a unos puntos de disparo activa distintos mecanismos de enfriamiento activo. En base a estos eventos, este sistema controla por ejemplo, la velocidad de ventiladores o limitar la frecuencia máxima.

En la figura 2.2 podemos ver un resumen de los elementos del subsistema de temperatura. Por un lado, en el sistema operativo en espacio de usuario tenemos una interfaz para modificar el comportamiento del sistema térmico. Este se encuentra en espacio de *kernel* donde también están los dispositivos de enfriamiento y el *governor*. Por último en el hardware tenemos los sensores y dispositivos mecánicos de enfriamiento. El *thermal governor* se organiza en

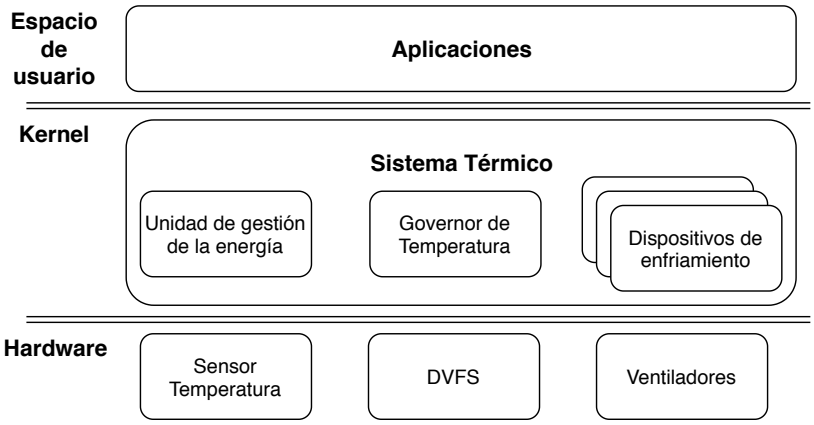


Figura 2.2: Esquema componentes del subsistema térmico en Linux

zonas térmicas. La figura 2.3 muestra los componentes de una de ellas. El sensor obtiene la temperatura y el *governor* regula y actúa sobre los dispositivos de enfriamiento que, a su vez, puede que controlen elementos mecánicos como ventiladores. De este modo se posibilita asignar dispositivos a un *governor* en la configuración del *kernel* personalizándolos para cada procesador.

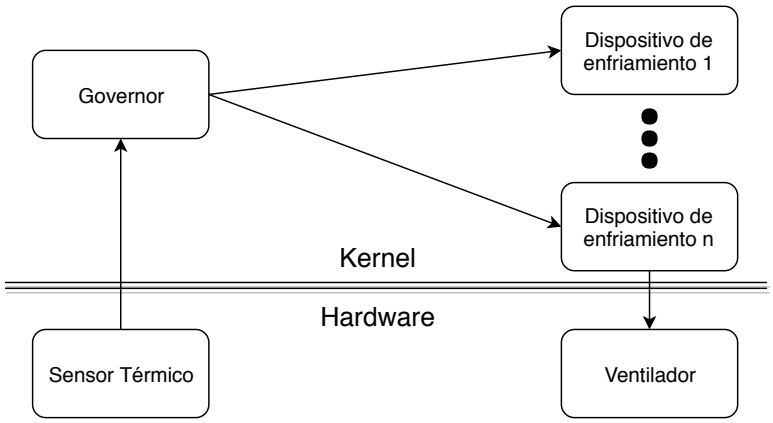


Figura 2.3: Esquema de una zona térmica

2.4.2 *Gestión de frecuencia*

El *governor* de frecuencias es un sistema que se encarga de elegir las frecuencias de trabajo del procesador. La elección se realiza de acuerdo a distintos

parámetros, en general se emplea la carga de trabajo del procesador calculado mediante el tamaño de las colas de procesos, y siempre eligiendo entre las frecuencias que se encuentran disponibles. Las frecuencias disponible pueden ser modificadas por el *governor* de temperatura cuando se da el caso de tener que decrementar la temperatura debido al ahogamiento térmico.

Los principales *governor*s de frequencias son:

- **Ondemand** es el que viene por defecto en la mayoría de sistemas, elige la frecuencia en base a la carga de trabajo del procesador. La carga del procesador se calcula periódicamente revisando en las colas de trabajo del *kernel* la cantidad de tareas por realizar. A esta carga de trabajo le aplica una función que la mapea con las frecuencias disponibles para seleccionar. Las frecuencias que tiene disponible para seleccionar son limitadas por el *governor* de temperatura.
- **Performance** elige siempre la frecuencia más alta disponible para mejorar el rendimiento.
- **Userspace** no es un governor como tal, sino un sistema que permite al usuario elegir la frecuencia del procesador.
- **Powersave** mantiene la mínima frecuencia disponible.
- **Intelligent Power Allocator** este es el governor más parecido a nuestro trabajo y se describe en detalle en la siguiente sección.

2.4.3 *Intelligent Power Allocator*

Actualmente ya existe una aproximación al control de la temperatura utilizando un controlador PID, llamado *Intelligent Power Allocator* [24]. Este fue diseñado por la empresa *ARM* para su uso en dispositivos móviles. Funciona como un governor de temperatura, el cual, mediante la lectura de la misma hace un modelo de la energía que puede seguir disipando sin saltarse los límites de temperatura. En base a esta estimación modifica las frecuencias máximas que los *governors* de frecuencia pueden elegir del procesador. Además, para el mercado móvil añade que durante los primeros segundos se puede dar una pequeña ráfaga en la que se sobrepase la temperatura límite. Esto está pensado para cargas de móvil donde se desea que se comporte de manera más responsiva, como cargar una web. Por tanto, es un control centrado en aplicaciones cortas y cuyo PID sirve para hacer una estimación de energía disponible, en vez de un control directo sobre las acciones que modifican el comportamiento del sistema.

3 ESTRATEGIA DE CONTROL

En este capítulo se desarrolla la teoría detrás del control. Desde la identificación del sistema, el diseño de control hasta la formulación para ser implementado sobre un sistema real.

3.1 IDENTIFICACIÓN DEL SISTEMA

El primer paso en el diseño de control es tratar de entender el comportamiento del sistema a controlar. Por tanto, la primera tarea es crear un modelo matemático de la relación entre la temperatura y la frecuencia. Porque, a pesar que el comportamiento térmico puede ser teóricamente modelado en base a fórmulas de disipación [25], tal y como hemos visto en la sección 2.2, es difícil obtener los valores de todas las constantes envueltas en el proceso con suficiente precisión. Por tanto, para este trabajo hemos preferido utilizar una aproximación lineal que sigue dinámicas simples, bajo la asunción de que este modelo es suficientemente representativo del comportamiento del sistema para permitir un control de temperatura preciso.

Si tomamos como base la ecuación 2.4 podemos simplificarla sabiendo que V_{dd} y f son variables dependientes, dado que para incrementar f también se debe incrementar V_{dd}. Dado que la frecuencia es lo que podemos modificar en el sistema podemos simplificar la ecuación como,

$$Q = \alpha f^3, \tag{3.1}$$

donde α es un parámetro que agrupa las variables restantes de la ecuación. Esta fórmula la podemos simplificar mediante el teorema de Taylor linearizandola usando incrementos de la frecuencia con respecto a un punto de trabajo f_0. Además debido a características del sistema que se verán más tarde añadimos un término de retraso, para contabilizar el tiempo que pasa desde que se cambia la entrada hasta que se nota en la salida. La ecuación quedaría de la forma,

$$Q = \beta f(t - T_d), \tag{3.2}$$

donde β es ahora el parámetro que agrupa al resto de las variables del sistema.

Esta ecuación la podemos sustituir por su equivalente en la ecuación 2.6 de manera que el calor que se genera en el procesador es el que se disipa quedando que,

$$\beta F(t - T_d) = mC_p\dot{T_{\mu P}} + \frac{T_{air} - T_{\mu P}}{R_t}. \tag{3.3}$$

Puesto que lo que nos interesa es la relación que existe entre la temperatura del procesador y el calor generado, de momento vamos a omitir la temperatura del aire, T_{air}, que será modelada más adelante como una perturbación que entra al sistema. Por tanto si seguimos simplificando terminamos con,

$$R_t mC_p\dot{T_{\mu P}}(t) + T_{\mu P}(t) = \beta R_t F(t - T_d), \tag{3.4}$$

en la que los términos $R_t m C_p$ y βR_t dado que son muy difíciles de obtener de manera empírica, se pueden agrupar como T_p y K_p, que serán las variables que se deberán obtener en base a los experimentos. Para el estudio del comportamiento de los sistemas se transforma la ecuación mediante Laplace de variable real t a variable compleja s quedando tras la transformación como,

$$(T_p s + 1) T_{\mu P}(s) = K_p F(s) e^{-T_d s}. \tag{3.5}$$

Por último, usamos esta ecuación a modo de función de transferencia para modelar nuestro sistema relacionando la temperatura del procesador $T(s)$ con la frecuencia de entrada $F(s)$ de la forma,

$$G(s) = \frac{T_{\mu P}(s)}{F(s)} = \frac{k_p}{1 + T_p s} e^{-T_d s}. \tag{3.6}$$

Esta ecuación es típicamente usada para modelado de sistemas de intercambio de calor [22], y en la que debemos ajustar los parámetros k_p proporcional, T_p tiempo de respuesta y T_d retraso ante cambio en la entrada. Podríamos haber usado un modelo más complejo, por ejemplo, añadiendo polos extra, pero nuestros experimentos en la sección 7.1 muestran que este modelo es suficientemente bueno. Además este modelo como ya hemos visto en el desarrollo no lo vamos a utilizar en términos absolutos como así hicimos en el trabajo de fin de grado, sino que lo vamos a linealizar. Esto lo hacemos mediante la elección de un punto de trabajo deseado de temperatura-frecuencia, (f_0, T_0). Este punto lo podemos extraer a partir de las pruebas viendo en qué pares de valores el comportamiento del sistema es más estable. A partir de este punto de trabajo se caracteriza mediante la ecuación con los cambios de frecuencia con respecto a f_0 y los cambios en temperatura asociados con respecto a T_0.

3.2 ARQUITECTURA DE CONTROL REALIMENTADO

La función de transferencia previa no modela por completo el comportamiento del sistema, debido a que existen otras variables que afectan a la temperatura del procesador, como por ejemplo, la temperatura ambiente. Esto se puede ver en rojo en la figura 3.1, donde la temperatura del procesador puede ser obtenida teniendo en cuenta la frecuencia aplicada y la perturbación externa de temperatura con una función de transferencia desconocida G_p. Incluso sin los medios para medir esta perturbación el control, la estrategia de control debería ser capaz de tenerla en cuenta para anularla.

El control que se utilizó en el trabajo de fin de grado se puede ver en azul en la figura 3.1. Consiste en un control PID que toma el error entre la temperatura deseada T^* y la temperatura real T medida por el sensor integrado en el procesador. El controlador selecciona la frecuencia de funcionamiento y se aplica directamente al sistema. Este tipo de control, aunque funcional, tiene una serie de problemas cuando se trata de controlar un sistema que no es linear debido al tan amplio abanico de valores de temperatura que debe controlar. Para paliar estos problemas se ha aplicado un diseño de control mucho más robusto basado en puntos de linearización.

El esquema escogido, puede verse en azul en la figura 3.2. Nuestro método coge como entrada el valor deseado de temperatura para el procesador, T^* y la

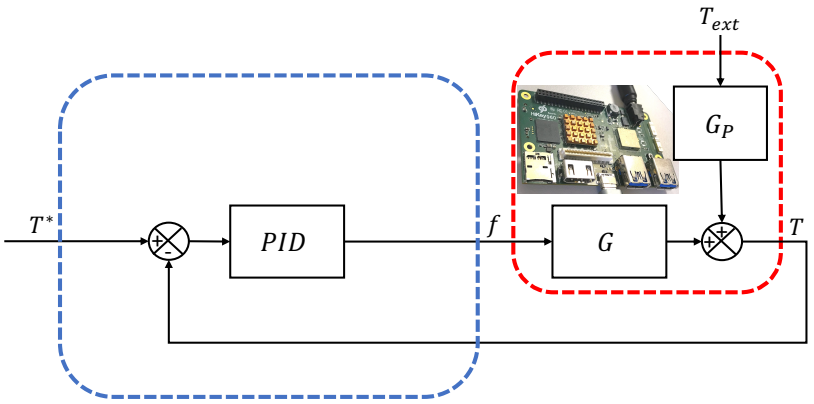


Figura 3.1: Diagrama control PID simple

temperatura real, T, medida en el procesador. Dado que el modelo descrito en la sección 3.1 solo tiene en cuenta variaciones en la salida con respecto al punto de trabajo, el algoritmo de control esta expresado en esos cambios. Esto es realizado restando T_0 de ambas entradas del sistema. Entonces, la frecuencia seleccionada es computada por la suma de dos términos, un término en prealimentación, FF (feedforward), junto a un control en retroalimentación PID.

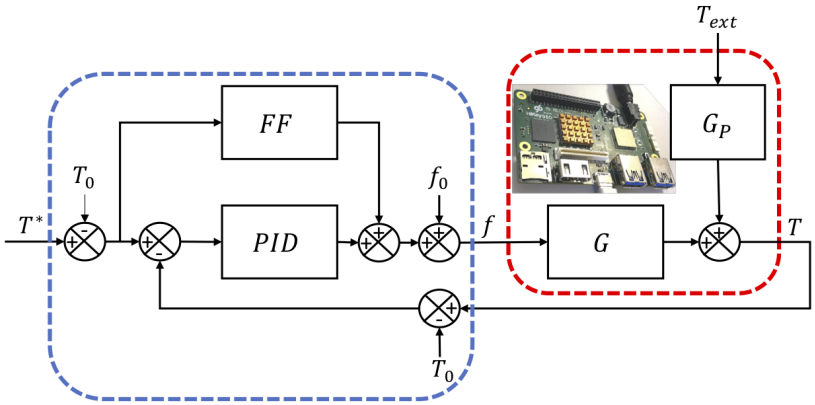


Figura 3.2: Diagrama control PID avanzado

El término en prealimentación es usado para compensar las variaciones de la temperatura deseada con respecto a las condiciones de trabajo, seleccionándolo a $1/k_p$. De este modo, el controlador tiene una idea inicial del estado estacionario requerido para la frecuencia. Sin embargo, este término por sí mismo no es capaz de compensar el efecto de las perturbaciones, ni por errores de modelado o incertidumbre en la ecuación (3.6), por lo que el PID toma un rol importante. La función de transferencia de un controlador PID tiene la forma

$$PID(s) = K\frac{(T_i s + 1)}{T_i s}\frac{(T_d s + 1)}{(\alpha T_d s + 1)} \tag{3.7}$$

donde K es la acción proporcional, T_i es la constante de tiempo asociada a la acción integral, T_d la constante de tiempo de la acción derivativa y $0 < \alpha < 1$ es un parámetro necesario debido a limitaciones físicas de la parte derivativa [6]. En pocas palabras, las acciones proporcional y derivativa se

encargarán de las dinámicas de temperatura, mientras que la acción integral se encargará de eliminar la influencia de las perturbaciones, así como las posibles imprecisiones en el modelado del sistema, asegurando que T converge a T^* en estado estacionario. Finalmente, a la variación de la acción se le añade el punto de trabajo f_0 y se aplica al procesador la frecuencia más cercana.

Uno de los problemas que nos encontramos con el control propuesto es, que la acción escogida es continua pero la que de verdad se aplica es cuantizada. Esto implica como se verá en la siguiente sección 3.3 que, para realizar el cálculo de la acción en el siguiente instante temporal hace falta la acción previa. En un control clásico deberíamos usar la acción incremental realmente escogida y no la acción discreta escogida por nuestro control. Al realizar las pruebas sin embargo descubrimos que la distancia entre acciones es tan grande que el control es incapaz de dar una acción suficiente como para cambiar la frecuencia aplicada al procesador. Para tratar de subsanar este problema y hacer que el control sea lo más correcto posible diseñamos el control mostrado en la figura 3.3.

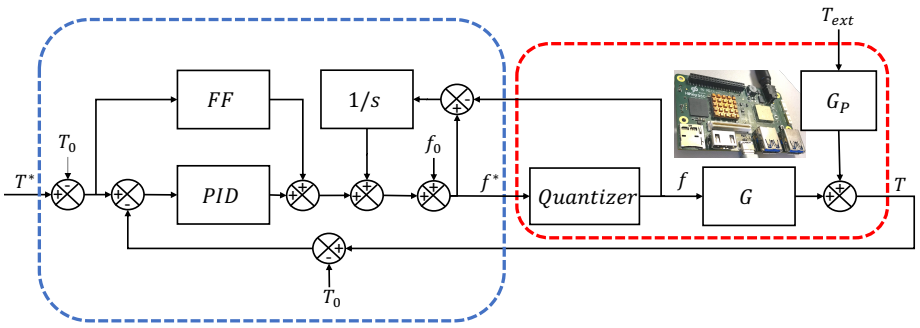


Figura 3.3: Diagrama control PID avanzado con cuantización

En este esquema se ha añadido el bloque que representa al cuantizador de la acción. Para poder subsanar la falta de cambio de frecuencia se ha añadido un término integrador que acumula el error entre la acción elegida y la finalmente usada. De este modo el error se irá acumulando hasta que se cambie la frecuencia. Esto plantea un debate sobre la complejidad que se añade al esquema de control y sobre el comportamiento que al final se obtiene.

Por un lado, el esquema es completamente correcto con lo que nos plantea la teoría de control. Pero, por otro lado, cuando el incremento en la acción no es lo suficientemente grande para cambiar la frecuencia causa en la implementación discreta del control que este no modifique sus valores y siempre realice la misma acción. De este modo el término integral es el único que produce cambios de frecuencia y por tanto que esta solución no se haya usado y se haya optado por el esquema de la figura 3.2.

3.3 IMPLEMENTACIÓN ALGORÍTMICA DISCRETA DEL CONTROL

Dado que el controlador necesita ser implementado como un algoritmo en el sistema, debe ser discretizado. Para un buen comportamiento, el tiempo de muestreo debe ser igual al tiempo que pasa entre iteraciones consecutivas del *governor* donde se encuentra implementado. Para el desarrollo de la discretiza-

ción vamos a tomar la fórmula de un controlador PI, dado que en el trabajo de fin de grado comprobamos que el término derivativo es muy sensible a las entradas muy variables como el sensor de temperatura aquí usados. Por tanto la ecuación que vamos a usar es la de un PI,

$$PI(s) = K\frac{T_i s + 1}{T_i s}. \tag{3.8}$$

En el anexo A se encuentra el desarrollo matemático que nos lleva a la ecuación,

$$f(k) = FF(T^*(k) - T_0) + f(k-1) + \delta_1 e(k) + \delta_2 e(k-1), \tag{3.9}$$

que es la que terminaremos usando en el controlador y que podemos ver es muy sencillo. Solo se requieren 3 multiplicaciones y 4 sumas para poder hacer el cálculo del control minimizando la sobrecarga de nuestro control en el sistema. Este control se ejecutará con un periodo de 100ms aunque se puede modificar en tiempo de ejecución, debiendo recalcular el valor de los parámetros.

4

CLASIFICACIÓN DE PROGRAMAS EN EJECUCIÓN

Mientras se trabajaba en el modelado y se escogían programas de prueba, se observó que el comportamiento del sistema cambiaba dependiendo del programa que se ejecutaba, e incluso de las fases de cada programa. Más en concreto, se observó que dependía del tipo de cálculo que se estaba realizando. En en este capítulo al hablar de tipos programas se hará en sentido amplio y también se referirá a sus fases de ejecución. Por ejemplo, en el inicio de un programa de cálculo intenso se puede observar una primera fase en la que se prepara el programa leyendo de memoria datos antes de pasar a calcular. Estos comportamientos se clasificaron en tres grupos mayoritarios, ejecución flotante, entera y operaciones de memoria. Con esta información hemos ampliado el control para que pueda reconocer las fases que va atravesando el procesador y modifique su comportamiento.

4.1 CONTROL SUPERVISADO

A la hora de clasificar las fases de ejecución ya existen múltiples soluciones entre las que se incluyen la inserción de marcas en programas, o llevar un registro de los programas ejecutados y tener una base de datos con nombres y tipos [5, 8, 18]. Nuestra propuesta pretende hacer una identificación dinámica y sin información previa, dado que lo que pretende es tener una visión general del comportamiento que tiene el procesador para intentar una mejor adaptación del control.

El nuevo control propuesto utiliza un supervisor que se encarga de tomar lecturas de contadores hardware con los que se realiza una clasificación para analizar la fase ejecutada. En base a la fase se modifican los parámetros de control de los distintos elementos tal y como se puede observar en la figura 4.1 donde se presenta el esquema de control finalmente utilizado.

Podemos ver que el supervisor puede modificar tanto los valores del control PID, como los puntos de trabajo. Además, también puede modificar la temperatura objetivo, permitiendo que dependiendo de la carga de trabajo el sistema pueda alcanzar temperaturas más altas con el fin de poder obtener un mejor rendimiento.

Los contadores hardware permiten contar distintos eventos del procesador como, accesos a memoria, tipos de instrucciones, saltos realizados, etc. En el apéndice B se detalla información técnica sobre como se configuran y se leen; y en el capítulo 6 la implementación en módulo de *kernel* que se ha realizado para poder leer valores de estos contadores. Gracias a este módulo podremos leer los valores de los contadores por segundo desde el sistema virtual */sys/*, así como desde dentro del propio *kernel* para que el supervisor pueda leerlos directamente.

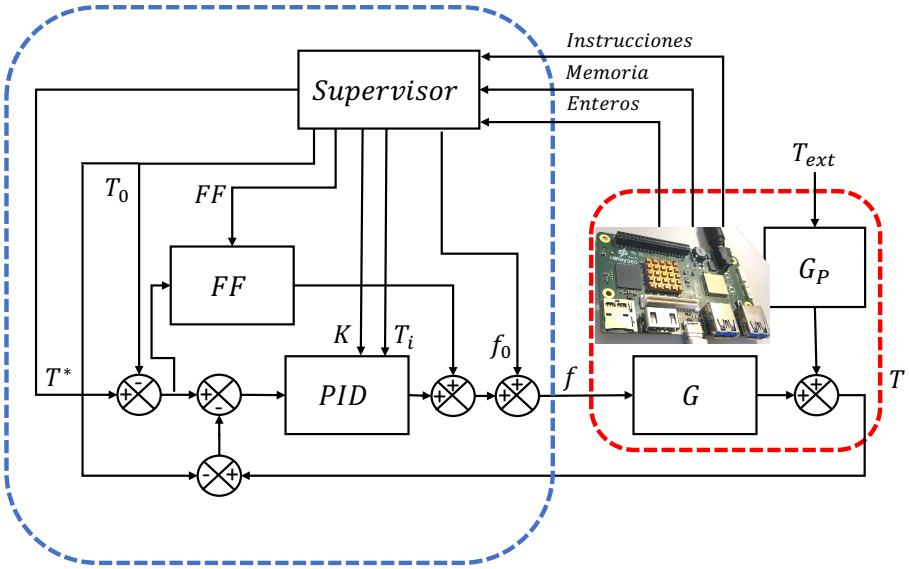


Figura 4.1: Diagrama control PID con supervisor

Dado que el supervisor lee los contadores cada segundo la clasificación también se realizará cada segundo. Como trabajo futuro quedaría probar que pasa si se usan periodos más cortos para realizar la clasificación y la lectura.

4.2 METODOLOGÍA PROPUESTA

La figura 4.2 muestra el funcionamiento general de la metodología propuesta para la clasificación de los programas en ejecución. La clasificación esta basada en una regresión logística multinomial sobre diversos contadores hardware. Tal y como muestra la figura 4.2, el cálculo de los parámetros de la regresión se realiza en la fase de entrenamiento donde se ejecutan programas característicos de los tipos a clasificar. Posteriormente durante la ejecución, de manera periódica, se emplean los parámetros calculados junto a los contadores hardware para determinar el tipo del programa y el supervisor ajustará el control de acuerdo al tipo observado. Esta metodología no requiere ningún tipo de memoria sobre el estado del sistema y tiene una sobrecarga muy pequeña.

La metodología seguida ha sido la selección de varios eventos que se cree que pueden influir en cada uno de los tipos de programas. Estos han sido, las instrucciones de enteros, de acceso a memoria principal, de tipo SIMD (single-instruction multiple-data) e instrucciones totales ejecutadas. Se han elegido porque a priori cuentan los eventos relacionados con cada una de las fases que queremos discernir, por ejemplo, el contador de instrucciones a memoria nos puede dar una buena idea de si el programa está haciendo varios accesos a memoria. Las instrucciones SIMD son importantes porque hacen uso de varias unidades del procesador a la vez, lo que ocasiona que se incremente la C de la ecuación 2.3 y por tanto generen más calor. Estos eventos se cuentan por segundo mediante el módulo creado para ello y se usa un programa automático que va recolectando sus valores. Se ejecuta cada uno de los programas y se almacenan los valores.

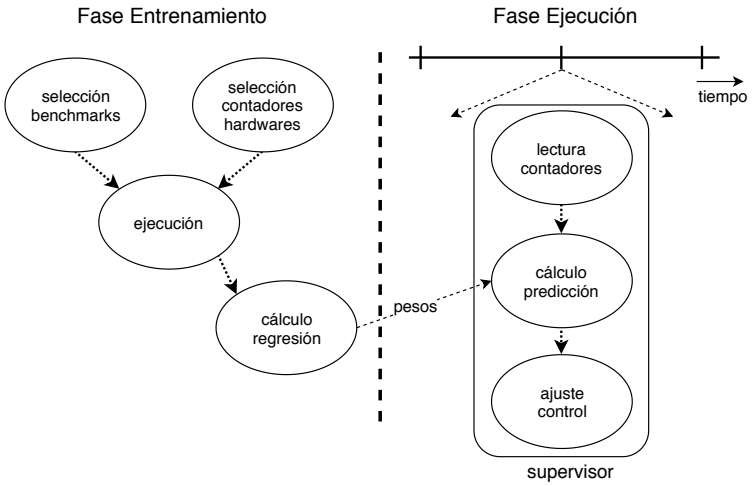


Figura 4.2: Esquema general de la metodología propuesta

A los valores extraídos de los contadores, aplicaremos una *regresión logística multinomial*. Esta nos permite clasificar Y variables respuestas categóricas con p variables explicativas. La fórmula de la regresión tiene la forma,

$$\begin{aligned} P(Y = i|X) &= \frac{e^{\sum \beta_j^i X_j}}{1 + e^{\sum \beta_j^1 X_j} + \cdots + e^{\sum \beta_j^k X_j}} \\ P(Y = 0|X) &= \frac{1}{1 + e^{\sum \beta_j^1 X_j} + \cdots + e^{\sum \beta_j^k X_j}}, \end{aligned} \tag{4.1}$$

en la que se tiene el caso base $Y = 0$ y cada uno de los restantes $Y = i$. El denominador es común para todos los factores por lo que para nuestro clasificador significa que solo debemos calcular el caso base y otro de las casos para poder obtener todos los valores de probabilidad que se encuentran normalizados a 1. X es la matriz de las p variables explicativas y β son parámetros que se deben encontrar realizando el ajuste del modelo, por tanto debemos encontrar un parámetro para cada uno de las variables explicativas.

En base a esta fórmula realizaremos el ajuste del modelo probando distintas variables explicativas para encontrar cual es la mejor configuración para maximizar el acierto del clasificador.

5 METODOLOGÍA

Es este capítulo explicamos la metodología seguida para realizar las pruebas y evaluaciones. Se describe la plataforma hardware, el software, modelado de la temperatura y como se realiza la clasificación.

5.1 HARDWARE

Para la evaluación y pruebas del control hemos seleccionado por recomendación de uno de los ingenieros responsables del *governor* térmico IPA de *ARM*, la plataforma HiKey 960 mostrada en la figura 5.1. La HiKey es una plataforma de desarrollo con un system on chip (SoC). Este SoC es un *HiSilicon Kirin 960* de 8 núcleos con arquitectura *big.LITTLE* de *ARM*, lo que significa que tiene 4 procesadores de alto rendimiento (Cortex-A73) y 4 de bajo consumo (Cortex-A53), todo ello fabricado con un proceso de 16nm.

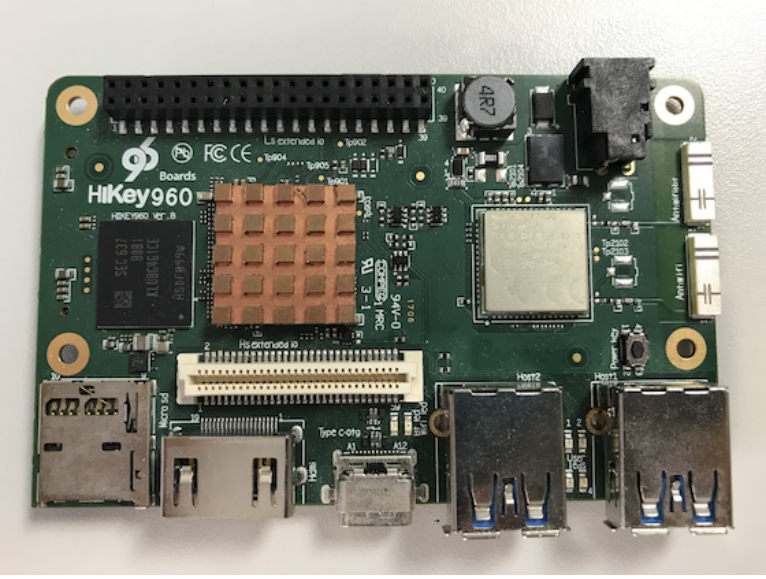


Figura 5.1: Plataforma de pruebas

Para los experimentos y la evaluación usaremos los núcleos de alto rendimiento, porque son los que más calor generan y además, permiten realizar operaciones de coma flotante al contrario que los procesadores de bajo consumo. La plataforma cuenta además con un pequeño disipador de cobre, como se puede ver en el centro de la figura 5.1. Este disipador no es suficiente para disipar el calor generado a por los 4 núcleos de alto rendimiento por lo que lo hace conveniente para la realización de nuestro experimentos y posterior comparación con las soluciones actuales.

5.2 SOFTWARE

5.2.1 *Plataforma*

Sobre la plataforma se ejecuta Android 8.0 del proyecto AOSP. Esto significa que todo el software ejecutado es software libre y por tanto puede ser modificado y recompilado por cualquiera. La posibilidad de ejecutar Android hace que nuestra solución sea mucho más rica dado que implica que puede ser usada directamente para dispositivos móviles. En cuanto al Kernel, usamos Linux en la versión 4.9 modificada por nosotros para poder ejecutar las aplicaciones sobre un sistema operativo con nuestros módulos de detección de fases y el *governor* de frecuencia.

5.2.2 *Benchmarks*

Para realizar las distintas pruebas y experimentos se han elegido varios programas que buscan simular cargas de trabajo comunes en procesadores. También se ha usado un software comercial ampliamente utilizado, que realiza un conjunto variado de test para probar todas las características de los sistemas. Los test escogidos se han agrupado en tres grupos principales que representan la naturaleza de las aplicaciones.

Por un lado, tenemos los programas que realizan una gran carga de trabajo en la jerarquía de memoria. Este tipo de programas incluirían servidores de ficheros y bases de datos. Para probar este tipo de ejecuciones usamos los benchmarks memcpy y sysbench [13]. Memcpy es un programa sintético que realiza copias de datos entre dos zonas de la memoria principal. Sysbench por el contrario es un conjunto de herramientas para la evaluación de sistemas operativos, entre las cuales se encuentra la posibilidad de probar la memoria, opción que hemos escogido.

Por otro lado, tenemos programas orientados al cálculo de operaciones de coma flotante. En este tipo de aplicaciones se incluye, *machine learning*, visión por computador y análisis de datos. Los programas seleccionados han sido, stress-ng [3], un conjunto de herramientas parecido a sysbench en el cual se ha elegido la opción de estresar la coma flotante. También se ha querido simular la ejecución de aplicaciones de *machine learning*, para lo cual, se ha realizado un programa sintético que realiza multiplicaciones de matrices, las cuales son la base de cualquier convolución en una red neuronal. Este programa hace uso de la biblioteca matemática de código abierto eigen [7] para ejecutar multiplicaciones de matrices haciendo uso de instrucciones vectoriales SIMD (single-instruction multiple-data) las cuales pueden realizar varias operaciones de coma flotante a la vez. El programa a su vez es altamente configurable permitiendo la selección de varios hilos de ejecución, su asignación a distintos núcleos, tamaño de matrices y duración de experimentos.

Por último, tenemos los programas más generalistas que representan una carga más relajada y en la que podemos incluir la mayoría de aplicaciones actuales. Para realizar los experimentos hemos seleccionado también dos programas distintos. Stress_ng, como en el caso de coma flotante pero con la opción de

solo enteros, y dhrystone. Dhrystone es un benchmark sintético veterano creado para servir de referencia como carga de trabajo entera para los procesadores.

Además de estas cargas de trabajo sintéticas se ha añadido una conocida aplicación comercial de benchmarking largamente usada en una gran variedad de dispositivos y sistemas. Geekbench [14] es una de las mayores referencias a la hora de comparar dispositivos, una de las razones de su elección es, que la puntuación que se obtiene en Geekbench implica una mayor venta de dispositivos. Otra de las razones para seleccionarlo es, que los distintas pruebas que realizan son aplicaciones reales, como abrir una página web, aplicar filtros a fotografías o medir la latencia de la memoria. Geekbench al igual que nuestro trabajo también clasifica los programas dependiendo de si son enteros, de coma flotante o de memoria. Geekbench no proporciona información sobre en que momentos se esta ejecutando cada uno de los tests para poder clasificarlos. Para poder usar Geekbench como datos de prueba para la clasificación de tipos de programa se investigó hasta descubrir que la ejecución deja una traza en los logs del sistema. Esto es debido a que la aplicación muestra un mensaje por pantalla en el que indica en qué momento se va ejecutando cada aplicación, y Android registra esos cambios. Se ha debido desarrollar una pequeña herramienta Perl que parsea todos los logs y registros y los sincroniza unos a los otros.

Para ayudar a automatizar todas las pruebas se ha hecho uso del *framework* de ejecución de pruebas y recolección de datos de *ARM* llamado Workload Automation (WA) [27]. Este *framework* permite ejecutar lo que llama agendas, ficheros en formato YAML, donde se describen los experimentos y se modifican los ajustes de configuración de la plataforma. Mediante esto se puede seleccionar la cantidad de núcleos que se usan, la frecuencia de funcionamiento, instrumentación a usar, tiempos de ejecución,... Uno de los instrumentos usados es un programa llamado *poller* que permite monitorizar ficheros de sistema y extraer sus valores periódicamente. Destacar que se detecto un fallo a la hora de leer ficheros del sistema virtual que fue solucionado y aceptado como pull-request en el repositorio oficial del framework en github [10].

5.3 MODELADO

Para la realización y ajuste del modelado vamos a hacer uso de Matlab. Primero, ejecutaremos las pruebas necesarias en el sistema, registrando los valores de temperatura y frecuencia, haciendo uso de la herramienta previamente descrita *poller*. Esta es configurada para que lea los ficheros del sistema virtual */sys/*, */sys/class/thermal/thermal_zoneo/temp* para la temperatura y */sys/devices/system/cpu/cpu{numero cpu}/cpufreq/scaling_cur_freq* para la frecuencia. Tras recoger la información se hace que Matlab ajuste los parámetros de la fórmula dada por la ecuación 3.6. Por último simulamos los parámetros dados para comprobar que efectivamente el modelo se parece al sistema real.

6 IMPLEMENTACIÓN EN KERNEL

Ya hemos visto cual es el estado de arte actual en materia de control de temperatura, cual es nuestra estrategia de control y la metodología a seguir para la realización de los experimentos y pruebas. En este capítulo vamos a ver como se ha implementado la estrategia de control dentro del *kernel* de Linux, el cual es usado por el sistema operativo Android. Linux es un *kernel* monolítico al contrario que el resto de *kernels* mayoritariamente usados como Windows NT y XNU (MacOS) que son híbridos. Esto significa que todos los servicios que se quieran añadir al *kernel* que interactúen directamente con el hardware deben ser programados dentro de este. Los *kernels* híbridos por el contrario permiten crear servicios en espacio de usuario que no requieren tantos privilegios. Las ventajas principales de un sistema monolítico es el mayor rendimiento y libertad a la hora de añadir funcionalidad. En desventaja tienen que un fallo en uno de estos servicios puede causar un fallo general del sistema. Es por esto que es necesario seguir una serie de pasos y normas [23] para garantizar un buen funcionamiento del sistema. La inclusión del control dentro del *kernel* también facilita la transparencia de uso y la facilidad de inserción en sistemas reales sin requerir interacción por parte de los usuarios. Empezaremos viendo las características en común que comparten y luego nos centraremos en los detalles individuales.

6.1 INICIALIZACIÓN

Lo primero que vamos a ver es la declaración de un módulo de *kernel*. Para poder funcionar se debe tener tanto una función e inicialización como de destrucción. Estas además se deben declarar tal y como se puede observar en el siguiente fragmento de código, extraído del módulo del controlador.

```
MODULE_AUTHOR("Pablo Hernandez <pabloheralm@gmail.com>");
MODULE_DESCRIPTION("PID_governor - A PID governor to keep"
    "a constant temperature");
MODULE_LICENSE("GPL");

#ifdef CONFIG_CPU_FREQ_DEFAULT_GOV_PIDGOV
fs_initcall(cpufreq_gov_dbs_init);
#else
module_init(cpufreq_gov_dbs_init);
#endif
module_exit(cpufreq_gov_dbs_exit);
```

Aquí se puede ver por un lado la declaración de autoría y descripción, así como la licencia del módulo. Esta es importante debido a que dependiendo de la misma pueden existir restricciones a la hora de interactuar con otros módulos. Se ha elegido usar GPL debido al carácter académico de este trabajo permitiendo que cualquier persona pueda usar, modificar y usar el software en otros trabajos.

Tras esto tenemos tenemos las declaraciones de inicialización y salida del módulo. La declaración condicional del arranque se debe a que el momento de carga del módulo varia dependiendo de si se va a usar en el momento o más tarde. En el caso del *governor* si se elige por defecto, este debe arrancar cuando se carga el sistema de ficheros `fs_initcall`. En el caso del módulo clasificador, debido a la dependencia que tiene con el módulo perf se debe cargar al final del arranque, es por ello que se usa `late_initcall_sync`. Para casos generales basta con arrancar el módulo de manera normal con `module_init`.

6.2 SYSFS

No solo queremos que los módulos funcionen sino que queremos ser capaces de extraer información sobre ellos así como poder modificarlos, por ejemplo, extraer información sobre los contadores hardware o poder modificar la temperatura deseada de funcionamiento. Para ello, Linux ofrece desde su versión 2.6 el sistema virtual de archivos. Este sistema usa la fórmula tomada por los sistemas UNIX más antiguos donde todos los dispositivos del sistema se toman como abstracciones de ficheros. De este modo podemos habilitar la comunicación entre espacio de usuario y *kernel* mediante la escritura y lectura de ficheros.

Cada módulo que quiera mostrar estos ficheros debe crea un directorio donde mostrarlos para, de esta manera organizarlos en una jerarquía de acceso. Dado que el controlador se diseña como un *governor*, los ficheros se crean dentro del directorio de CPUs, mientras que el clasificador se crea dentro del directorio del *kernel*. Para el controlador el código incluye una serie de atributos que se registran junto al *governor* y la llamada `sysfs_create_group` tal y como se puede ver en el siguiente fragmento de código.

```
static struct attribute *attributes[] = {
    &sampling_rate.attr,
    &K_int.attr,
&Ti_int.attr,
NULL
};

static struct attribute_group dbs_attr_group = {
    .attrs = attributes,
    .name = "PID_governor",
};

sysfs_create_group(cpufreq_global_kobject,
                                    &dbs_attr_group);
```

Para el clasificador se usa una llamada para crear el directorio dentro del directorio *kernel*, y otra para añadir cada uno de los ficheros, tal y como se puede ver en el código.

```
// Creacion del directorio
files_kobject = kobject_create_and_add("hwcounters",kernel_kobj);

// Adicion de los ficheros
error = sysfs_create_file(files_kobject, &simd_attribute.attr);
```

A esto se debe añadir métodos para motrar y leer la información. Cada vez que el usuario realice una lectura o escritura lo que hará el *kernel* será llamar a uno de estos métodos. En el siguiente fragmento de código podemos ver un ejemplo de lectura por parte del usuario, escritura por parte del módulo. Se usan llamadas conocidas como *sprintf* y *sscanf* para realizar estas comunicaciones.

```
static ssize_t simd_show(struct kobject *kobj,
                  struct kobj_attribute *attr,
                                  char *buf)
{
    int i=0;
    u64 aux = 0;
    for (i=0; i<num_possible_cpus(); i++) {
            if(cpu_online(i)) {
                    aux += simd_count[i];
            }
    }
    return sprintf(buf, "%lu\n", aux);
}
```

6.3 PERIODICIDAD

Tanto el control como el clasificador requieren realizarse de manera periódica y lo más exacta posible. Esto en el *kernel* lo podemos conseguir mediante el uso de colas de trabajo, estas, permiten asignar un trabajo a un núcleo en un tiempo dado. Dado que el control de la frecuencia se realiza global para los cuatro núcleos solo se requiere de un trabajo en cada uno de los módulos.

La función a usar se trata de `schedule_delayed_work_on`, a la que se le indica el núcleo que realizará el cálculo, el trabajo a realizar y un retraso en *jiffies*, la medida de tiempo interna de los sistemas Linux. En el siguiente fragmento se muestra el método usado para la inicialización de las tareas.

```
static void do_timer(struct work_struct *work)
{
    int delay;
    // Obtencion de la informacion
    struct cpu_info_s *dbs_info =
            container_of(work, struct cpu_info_s, work.work);
    // Calculo de los jiffies en base al tiempo asignado por usuario
    mutex_lock(&dbs_mutex);
    delay = usecs_to_jiffies(dbs_tuners_ins.sampling_rate*1000);
    mutex_unlock(&dbs_mutex);
    delay -= jiffies % delay;
    // Calculo del control
    dbs_check_cpu(dbs_info);
    // Asignacion del trabajo a la cpu 4
    schedule_delayed_work_on(4, &dbs_info->work, delay);
}
```

Dado que el tiempo de cálculo puede ser variable y se quiere que las tareas se ejecuten siempre en el mismo tiempo se realiza una operación de módulo de los *jiffies*.

6.4 CLASIFICACIÓN

El módulo de clasificación que hemos implementado tiene dos funciones distintas. Por un lado, va a servir como interfaz para mostrar al usuario las estadísticas de cada un de los contadores seleccionados, en número por segundo, y por otro lado, va a realizar la función de clasificador publicando un método dentro del *kernel* que permite la comprobar la al resto de módulos que tipo de aplicaciones se están ejecutando. De este modo usaremos el módulo tanto para el análisis, leyendo los valores ofrecidos, como para realizar las pruebas y permitir al control que se auto-ajuste.

Ya hemos visto como hemos realizado el resto de partes del módulo, ahora vamos a ver la más importante, el acceso a los contadores hardware. En un primer momento se pensó en realizar de manera directa mediante ensamblador siguiendo las instrucciones seguidas en el anexo B, pero se encontró que existía conflicto con uno de los *watchdogs* del sistema, en concreto *watchdog_nmi* que también usa contadores para controlar la ejecución. De este modo, se optó por una segunda opción que además, permite una mayor interoperabilidad con otros núcleos usando el módulo perf del *kernel*. Este módulo no está pensado para usarse dentro del *kernel*, sino por programas en espacio de usuario por lo que su uso dentro del *kernel* no está recomendado. Pese a ello se ha conseguido el acceso a los contadores satisfactoriamente. En el siguiente fragmento de código mostramos las principales funciones encargadas de interactuar con perf.

```
// Lectura tipo de core
asm volatile(''MRS %0, PMCR_EL0'' : ''=r'' (val));
// Creacion de\emph{kernel} contador
pe[cpu][0] = perf_event_create_kernel_counter(&pea_INST_RETIRED,cpu,
    NULL,NULL,NULL);
// Lectura de valor
total = (u32) perf_event_read_value(pe[aux][0], &enabled, &running);
// Invalidacion y liberacion del contador
perf_event_disable(pe[cpu][0]);
perf_event_release_kernel(pe[cpu][0]);
```

La primera función que tenemos es la declaración de una instrucción en ensamblador. Esta se usa para poder identificar el modelo de núcleo debido a que los núcleos *LITTLE* del procesador no disponen de los contadores que necesitamos para la identificación del programa en ejecución. La segunda función se usa para declarar y crear el evento perf que se va a proceder a contar, el cual se lee mediante la tercera función. Por último tenemos las funciones que deshabilitan el contador y liberan los recursos tomados en la creación.

Como ya vimos en las ecuaciones 4.1, necesitamos realizar unos cuantos cálculos con la dificultad de tener que hacerlos en el menor tiempo posible y sin la posibilidad de usar operaciones de coma flotante las cuales están prohibidas en el *kernel*. Empezamos calculando uno de los numeradores para lo cual usamos el porcentaje de instrucciones de memoria y enteros por instrucciones totales con sus coeficientes correspondientes. Para el cálculo de la exponencial hacemos una aproximación del número e a 2 para poder aprovechar el desplazamiento a la izquierda de números binarios para calcular el resultado más rápido.

```
// e^(beta + x1*Y1 + x2*Y2)
```

```
        aux1 = inter_int_ + (mem * mem_int_)/100 + (integ * integ_int_)
            /100;
        // Con esta comparacion realizamos una implementacion rapida del
            numero e
        if (aux1 >= 0)
                if (aux1 < 62) int_num = e_c << aux1;
                else int_num = (s64)(1) << 63;
        else
        int_num = 0;
```

EL resto de cálculos son más directos, se calcula en denominador como la suma de los dos numeradores y se calcula la probabilidad para cada una de las clases. Finalmente, se selecciona la probabilidad más alta.

6.5 MÓDULO DE CONTROL

El módulo de control va a ser un *governor*, un módulo especial, esto quiere decir que no solo lo vamos a registrar como módulo en el *kernel* sino que además se va a registrar como driver de frecuencia. Esto permitirá al *kernel* saber que el módulo puede modificar la frecuencia del procesador así como ajustes en el sistema virtual de ficheros. Además, tendrá que leer la temperatura del procesador, algo para lo que Linux no está preparado, y por lo que este ha sido un punto en el que ha habido problemas. Para poder leer la temperatura del procesador no se dispone de una interfaz sencilla mediante la cual poder preguntar para poder obtener las temperaturas de los distintos sensores. Es por ello que este *governor* no se puede aplicar directamente a cualquier procesador *ARM* dado que el descubrimiento del sensor térmico esta embebido en el código mediante la llamada `tz0 = thermal_zone_get_zone_by_name("cls0");`. Para la lectura de la temperatura se usa `thermal_zone_get_temp(tz0, &temp_ac);` que devuelve la temperatura en miligrados.

El control va a poderse modificar en tiempo de ejecución así como cambiar su control dependiendo de lo que se este ejecutando en el procesador. Por tanto, tenemos dos tareas básicas que se deben realizar, el cálculo de la discretización de las variables de control y el cálculo del control como tal. Al igual que el caso del clasificador todas las cuentas deben ser cuantizadas en enteros sin poder usarse operaciones flotantes.

7 RESULTADOS

Una vez tenemos nuestro nuestro esquema de control propuesto, modelado e implementaciones, vamos a pasar a ver los resultados. Tanto de modelado del sistema, contadores y clasificación como pruebas de control y evaluación.

7.1 IDENTIFICACIÓN DEL SISTEMA

Empezamos viendo como ha sido realizada el modelado del sistema, mediante el uso de las herramientas explicadas en la sección 5. Para ello ejecutaremos las distintos pruebas con escalones unitarios de frecuencia mantenidos en el tiempo para poder comprobar como es el comportamiento tanto dinámico como en régimen permanente.

Empezamos viendo un ejemplo de experimento ejecutando el benchmark de multiplicación de matrices. Vamos cambiando la frecuencia a lo largo del tiempo permitiendo que la temperatura se estabilice para cada uno de los escalones de frecuencia. En la figura 7.1 podemos ver el resultado de esta ejecución, donde se aprecia el comportamiento dinámico durante los primeros instantes del cambio de frecuencia así como que para cada una de las frecuencias existe un estado estacionario constante.

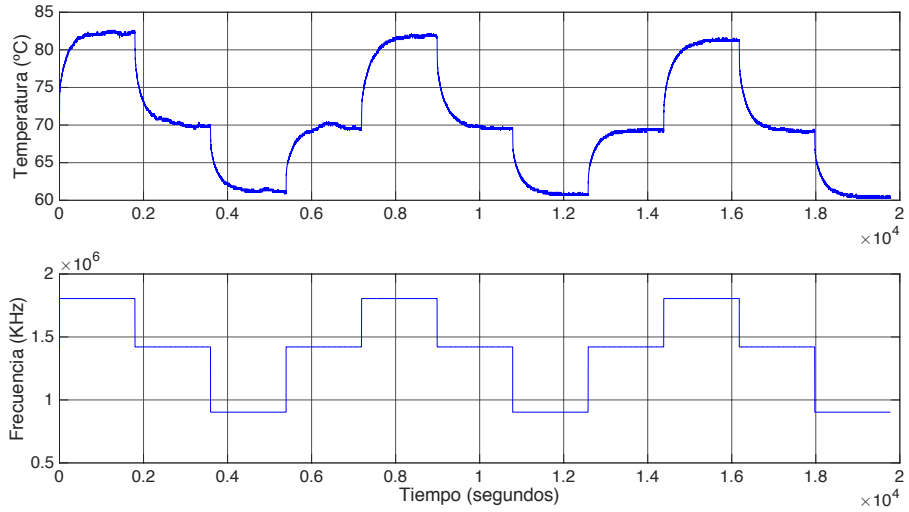


Figura 7.1: Experimento modelado de sistema con benchmark de matrices

Con estos resultados podemos empezar a ajustar nuestro modelo, eligiendo para ello el punto de trabajo $(f_0, T_0) = (1421MHz, 70°C)$, escogido tras varias ejecuciones de prueba donde se observaban la relación entre ambos valores y considerando que $70°C)$ es una buena temperatura de trabajo. De este modo podemos mejorar el control de un sistema que es claramente no lineal como se puede observar en la figura 7.1 en el que los cambios de frecuencia inferiores son más grandes que los superiores y sin embargo el salto de temperatura es

menor. Entre las frecuencias 903 y 1421MHz existe un salto menor de $10°C$), mientras que entre 1421 y 1805MHz es superior a $12°C$).

Con estos datos ajustamos el modelo y lo comparamos con el comportamiento real obtenido para comprobar lo bien ajustado que está. En la figura 7.2 podemos ver el resultado superponiendo ambos comportamientos, en rojo el real y en azul el simulado. Como ya sabíamos no se podía ajustar a la perfección debido a la falta de linealidad existente entre frecuencias y temperaturas pero, como el control se va a centrar en controlar las temperaturas altas que son las más peligrosas y donde se va a trabajar, el ajuste es suficientemente bueno como para darlo como válido con solo un 2 % de error.

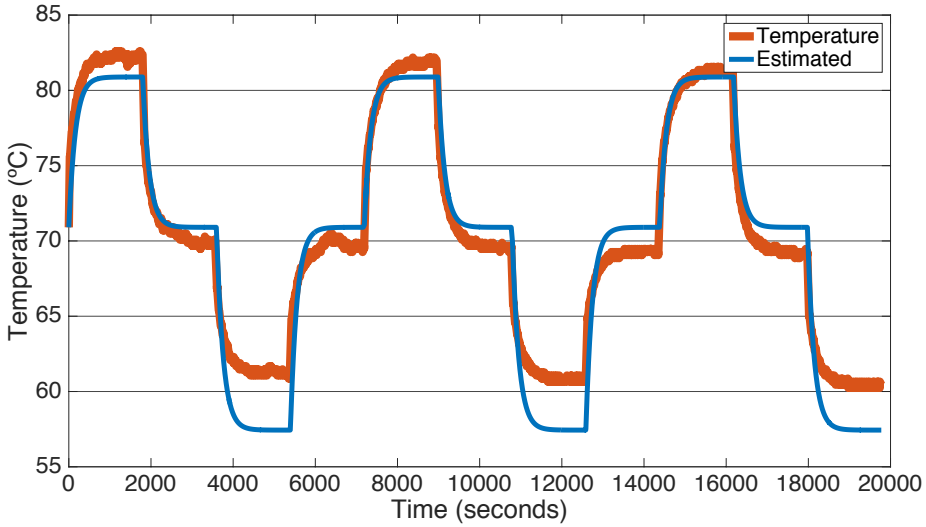


Figura 7.2: Caracterización benchmark de matrices

Tras ejecutar los experimentos en los otros tipos de benchmarks obtenemos distintos parámetros para cada uno como se pueden ver resumidos en la tabla 7.1. Lo primero que podemos observar es la falta de retraso del modelo, lo que nos indica la rapidez del procesador en cambiar de temperatura. Vemos también como el ajuste es bastante bueno para los tres casos, disminuyendo en el caso de la memoria al ser menos computacionalmente intensa. Por último podemos observar como clasificar los programas en ejecución para aplicar un control distinto es deseable al ver los distintos comportamientos de cada uno de los modelos. No podemos aplicar el mismo control a un programa de memoria que tiene una dinámica más lenta que a un programa de coma flotante con una dinámica mucho más alta.

También se ha probado a modelar el comportamiento con modelos algo más complejos usando hasta 3 polos. Pero, tal y como se puede observar en la tabla 7.2, estos no son influyentes y tienen unos valores demasiado pequeños como para poder tomarlos en consideración, teniendo en cuenta que no mejoran el modelo.

Tabla 7.1: Parámetros obtenidos tras el modelado

Benchmark	K_p	T_{p1}	T_d	Fit %
Matrix	$2{,}21 \cdot 10^{-5}$	133	0.0	78.02
Dhrystone	$1{,}74 \cdot 10^{-5}$	164	0.0	78.96
Memcpy	$1{,}92 \cdot 10^{-5}$	234	0.0	71.9

Tabla 7.2: Parámetros de funciones de transferencia para benchmark de matrices

Polos	K_p	T_{p1}	T_{p2}	T_{p3}	T_d	Fit %
1	$2{,}21 \cdot 10^{-5}$	133	0.0	0.0	0.0	78.02
2	$2{,}21 \cdot 10^{-5}$	133	0.049	0.0	0.0	78.02
3	$2{,}21 \cdot 10^{-5}$	133	0.0036	0.007	0.0	78.02

7.2 ANÁLISIS DE CONTADORES

Para el análisis de contadores hemos usado también las herramientas explicadas en la sección de metodología 5. El programa de lectura del pseudo-sistema de ficheros *sysfs* y el módulo de contadores que nos irá mostrando las estadísticas de uso de cada uno de los contadores escogidos. En este caso vamos a usar 4 contadores que a priori nos parecían que iban a servir para clasificar los programas; el número de instrucciones ejecutadas en total, instrucciones de acceso a memoria, operaciones de tipo entero y operaciones especiales SIMD (single-instruction multiple-data) normalmente usadas por programas de cálculo. No hemos usado otros contadores como por ejemplo de saltos, debido a que para las categorías seleccionadas no tendrían a priori mayor influencia y se buscaba simplificar el proceso de clasificación de la fase.

Los experimentos realizados han sido la ejecución de los tres tipos de benchmarks con distintas configuraciones del procesador, tanto de frecuencia como de número de núcleos usados. En la tabla 7.3 se puede observar una tabla resumen con el porcentaje de uso de cada una de las instrucciones especializadas por instrucciones totales por segundo. Los valores están obtenidos con la media de mediciones totales de varias combinaciones de los benchmarks ejecutados uno detrás de otro debido a que queremos obtener valores de los contadores que nos sirvan para entrenar. Hemos ejecutado los benchmarks de manera individual con la intención de obtener valores realistas de la ejecución de un programa. Es por ello que para luego realizar la regresión se han agrupado los datos de todos los benchmarks utilizados.

Vamos a ver en profundidad el caso de los programas de operaciones de coma flotante los cuales han dado unos resultados más variables y han contradicho los resultados esperados, dado que se esperaba que los programas que realizan operaciones flotantes usaran las operaciones de alto rendimiento SIMD pero, al ver los resultados vemos como estas son menores. Viendo por ejemplo el desarrollo temporal de los contadores durante la ejecución del benchmark de matrices en la figura 7.3 podemos ver el comportamiento esperado con un alto uso de instrucciones SIMD.

Tabla 7.3: Resumen contadores por tipo de programa, número de cores y frecuencia

Tipo Fase	Núcleos	Frec. MHz	%Mem	%Int	%SIMD
Flotante	1	903	12.22	69.49	0.55
Flotante	1	1421	9.63	70.80	0.41
Flotante	2	903	9.57	68.94	0.56
Flotante	2	1421	9.64	71.34	0.42
Entero	1	903	14.68	51.07	0.58
Entero	1	1421	12.64	54.80	0.43
Entero	2	903	20.20	52.50	0.59
Entero	2	1421	10.10	52.94	0.43
Memoria	1	903	79.70	10.24	0.00
Memoria	1	1421	79.80	10.17	0.00
Memoria	2	903	79.71	10.24	0.00
Memoria	2	1421	79.80	10.18	0.00

Por el contrario si mostramos el comportamiento del benchmark stress-ng en la figura 7.4 podemos ver como prácticamente no usa instrucciones SIMD. Esto contradice nuestra hipótesis de que se podían usar las instrucciones SIMD para clasificar programas de alta carga de cálculo flotante. Por que aunque las instrucciones flotantes y SIMD compartan unidades, los contadores las diferencian las unas de las otras. Es por ello que es necesario el uso de varios programas que sirvan para obtener datos de entrenamiento suficiente para realizar un buen análisis, como nuestro caso en el que hemos usado dos programas para cada tipo de programa.

En el caso de los programas más generalistas de enteros la diferencia entre los dos benchmarks ejecutados es mucho menor, aunque la carga de trabajo en dhrystone es mucho más homogénea que la producida por el programa stress-ng. En stress podemos ver ver en la figura 7.5 lo que parecen ciclos de ejecución repetitivos en los que la carga entre memoria y cálculo va variando.

Por último, en el caso de los programas de memoria ambos son también muy parecidos con una alta carga de instrucciones de memoria.

Ahora que tenemos todos los datos necesarios vamos a pasar a realizar la clasificación de los programas en base a los contadores hardware. Vamos a aprovechar el software de análisis estadístico *R* para realizar ajustes de la regresión logística multinomial vista en la sección 4, y probaremos la clasificación con los datos de test obtenidos con Geekbench. Los ajustes se han probado usando distintas variables de entrada, usando todos los contadores, usando el porcentaje de uso por instrucciones, usando memoria y SIMD, y memoria y enteros. Los resultados pueden observarse en la tabla 7.4, donde se muestra la precisión obtenida con cada uno de los dos conjuntos de datos. Podemos observar como usar una medida independiente de la velocidad a la que funciona el procesador como es el uso del porcentaje usado por instrucción hace que mejore la precisión en los datos de test. No así con los contadores en bruto dado que, los datos de test han sido obtenidos en entornos muy controlados. También

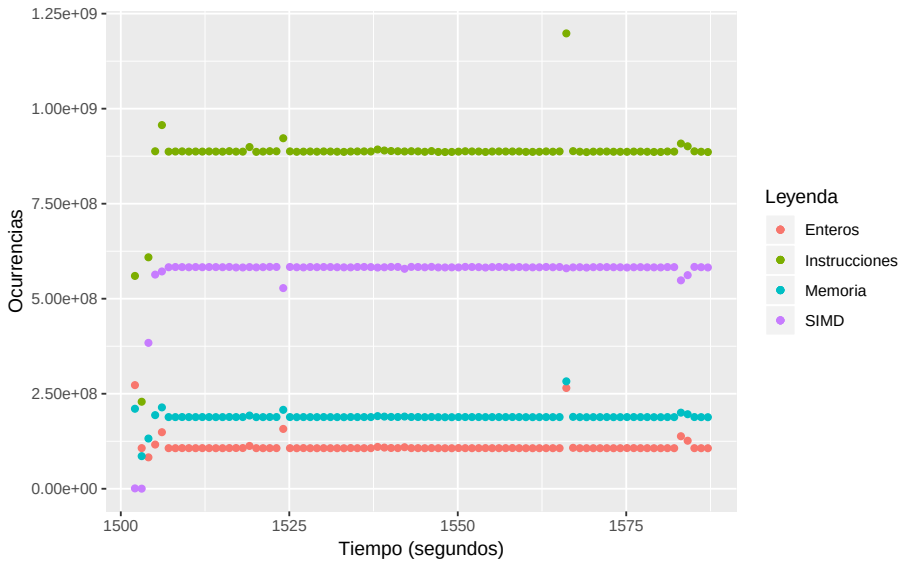


Figura 7.3: Análisis de contadores benchmark de matrices 903MHz 1 core

podemos observar a raíz de ver los resultados de SIMD previos como, su uso hace que empeore la clasificación debido a que en uno de los casos vistos se hace un gran uso mientras que en otro no.

La opción escogida ha sido la de usar la cantidad de instrucciones de memoria y enteros por instrucción ejecutada, no solo porque es la que mejor precisión consigue sino porque también simplifica el cálculo de la regresión al usar solo dos variables.

Tabla 7.4: Resumen precisión regresión

	Precisión (%)	
Variables	Datos ajuste	Test
% todas	79.5	33.88
% por instrucciones	79.3	51.97
% memoria y SIMD	79.3	43.03
% memoria y enteros	79.6	58.62

7.3 PRUEBAS DE CONTROL

Ya tenemos identificado nuestro procesador, tenemos implementado nuestro control y además sabemos con cierta probabilidad el tipo de aplicaciones se pueden estar ejecutando en cada momento. Ahora, vamos a realizar pruebas sobre el control para revisar su correcto funcionamiento y ajustar los parámetros necesarios. Empezaremos ajustando los tipos por separado y luego procederemos a comprobar el comportamiento cuando esté todo funcionando a la vez.

Empezamos con los programas de coma flotante, los que usamos el benchmark de multiplicación de matrices. Los parámetros escogidos han sido, un punto de trabajo $(f_0, T_0) = (1421MHz, 70°C)$, temperatura deseada $T = 72°C$ para

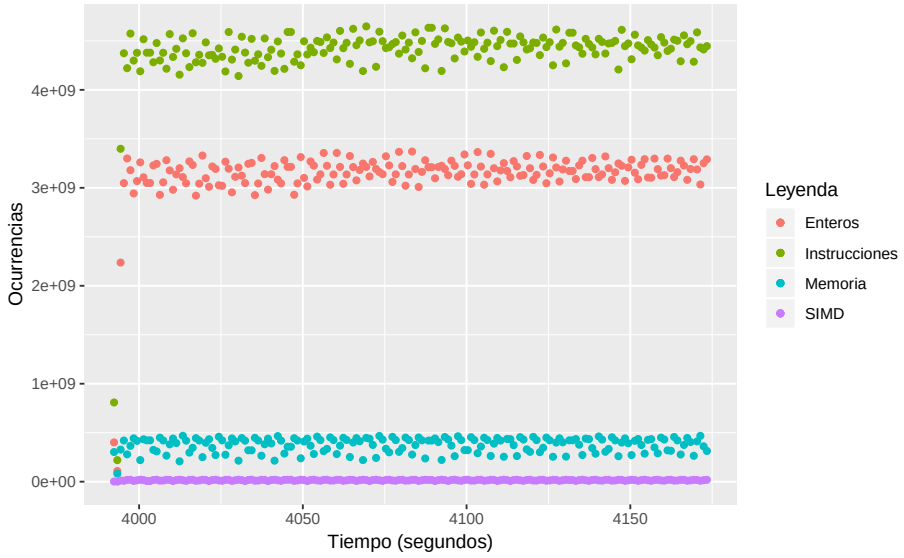


Figura 7.4: Análisis de contadores benchmark de stress-ng 1421MHz 2 cores

poder probar el termino en pre-alimentación con un valor de $FF = 32$, una constante proporcional $K = 50k$ y dos valores distintos de integrador $Ti = 250$ y $Ti = 80$ para comprobar los efectos que tiene modificar este valor. En la figura 7.6 podemos observar el resultado de dos pruebas de aproximadamente 10 minutos de duración. Podemos observar como ambas pruebas logran estabilizarse en los 72 grados que hemos requerido, aunque con un tiempo integral mucho más pequeño vemos que tenemos cierta sobreoscilación de la temperatura debido a que el control mantiene durante más tiempo la consigna para tratar de alcanzar la temperatura deseada en mucho menos tiempo.

Si pasamos a programas de memoria, estos son mucho más constantes y es más sencillo controlarlos. Con un proporcional de $K = 80k$ y un integrador $Ti = 80$, junto con una temperatura objetivo de $T = 70°C$, conseguimos un control suficientemente bueno como podemos ver en la figura 7.7. No se dan sobreoscilaciones y el control logra mantener la temperatura con una baja variabilidad. Pese a ello, vemos que hasta que se alcanza la temperatura estacionaria el control tiene una alta variabilidad hasta que logra estabilizarse al final de la prueba.

Para la última prueba individual vamos a ver una de las pruebas realizadas con operaciones de enteros usando dhrystone. En la figura 7.8 podemos observar como se comporta el controlador con un proporcional $K = 120k$ e integrador $Ti = 50$. Vemos que se presenta una pequeña sobre oscilación que termina siendo bien controlada y ajustada a la temperatura deseada. También podemos ver como al final del programa se terminan haciendo otras operaciones no tan intensas como durante la ejecución por lo que el control incrementa la frecuencia para poder mantener la temperatura en el rango deseado.

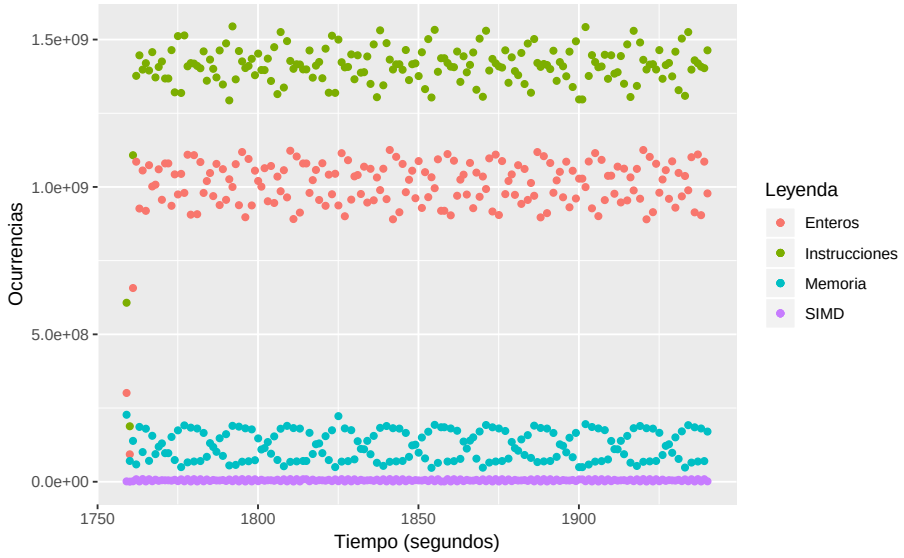
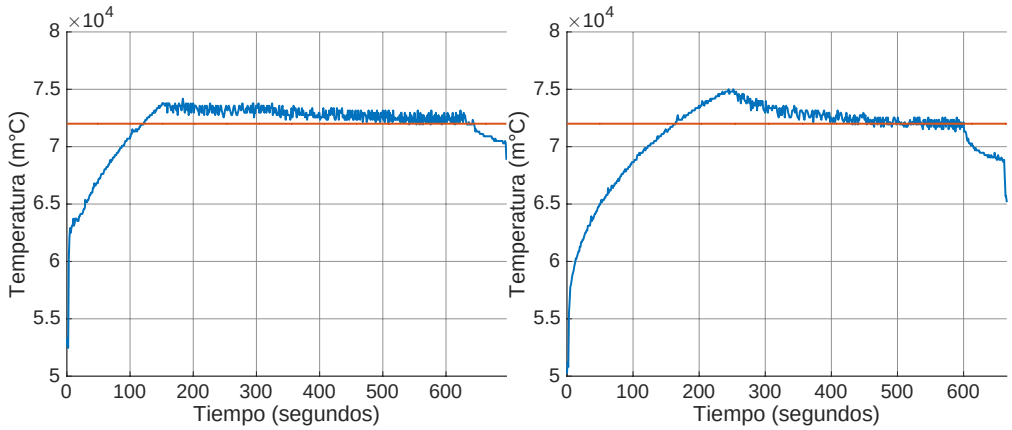


Figura 7.5: Análisis de contadores Stress enteros 903MHz 1 core



(a) Temperatura $Ti = 250$

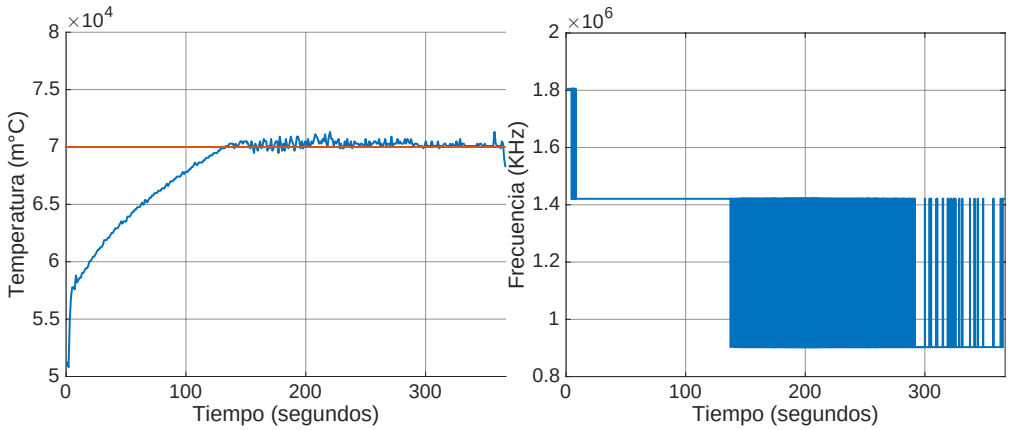
(b) Temperatura $Ti = 80$

Figura 7.6: Prueba de control para flotante: $T = 72°C$ $T_0 = 70°C$ $f_0 = 1421k$ $K = 50k$ con $Ti = 250$ Izquierda y $Ti = 80$ Derecha

7.4 EVALUACIÓN

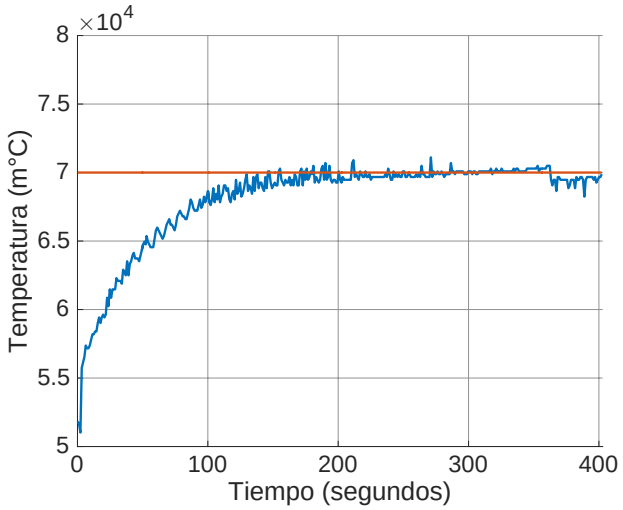
Para la evaluación vamos a estudiar por un lado si se obtiene beneficio usando un control que se adapta a las distintas fases de un programa, y por otro lado estudiaremos el comportamiento térmico cuando se compara con el controlador IPA que viene por defecto en la placa.

Para las comparaciones entre usar supervisor o no, configuramos el controlador único con unos valores medios para tratar de adaptarlo lo mejor posible a todos los casos, mientras que para el caso del supervisor cada fase ha sido adaptada en exclusiva. En el anexo C se encuentran los resultados con las temperaturas de todas las ejecuciones, aquí vamos a ver unos ejemplos representativos. Nos vamos a fijar principalmente en que el control alcance la temperatura de



(a) Temperatura (b) Frecuencia

Figura 7.7: Prueba de control para memoria: $T = 70°C$ $T_0 = 70°C$ $f_0 = 1421k$ $K = 80k$ $Ti = 80$



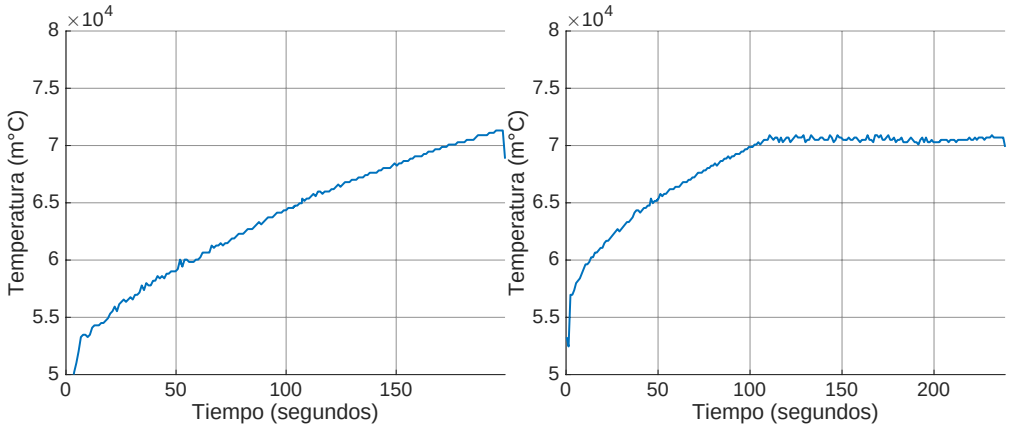
(a) Temperatura

Figura 7.8: Prueba de control para enteros: $T = 70°C$ $T_0 = 70°C$ $f_0 = 1421k$ $K = 120k$ $Ti = 50$

consigna, el tiempo en el que la alcanza y en caso de que sean comparables revisaremos las frecuencias.

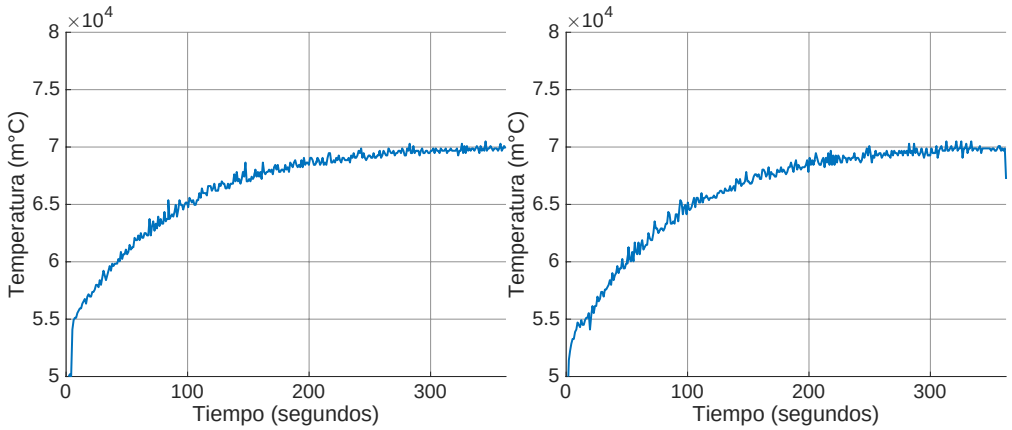
Debido a que el control único está adaptado a todo tipo de comportamientos este no debe ser excesivamente agresivo para evitar problemas de sobrecalentamiento en los programas más exigentes. Esto causa que en programas con un comportamiento más lento como memcpy en la figura 7.9, el controlador más conservador alcance el estado estacionario en un tiempo de 200 segundos mientras que el control adaptativo lo pueda hacer en solo 100.

Para los programas de enteros no existen tantas diferencia con el control debido a que los programas de enteros tienen un comportamiento intermedio entre flotante y memoria, por lo que el control funciona de manera similar en ambos. En la figura 7.10 se puede ver la temperatura alcanzada por stress-ng de enteros en ambos casos.



(a) Sin Supervisor (b) Con Supervisor

Figura 7.9: Temperaturas Memcpy



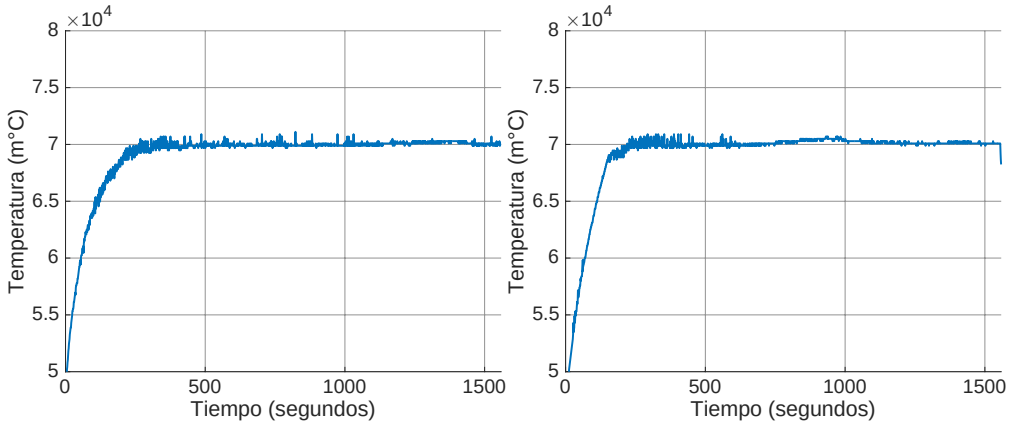
(a) Sin Supervisor (b) Con Supervisor

Figura 7.10: Temperaturas Stress Entero

En el caso de los programas de coma flotante como el benchmark de Matrices el tiempo hasta el estado estacionario es también más rápido en el caso del supervisor como se puede ver en la figura 7.11 en la que uno tarda 250 segundos frente al otro que solo le cuesta 100.

Para terminar vamos a ver una comparación entre IPA y nuestro trabajo en la ejecución de Geekbench. Este, como ya hemos visto, realiza una carga de trabajo mucho más variada y muy intensa. Nuestro control ha sido configurado con los valores previamente probados para cada una de las fases, con flotantes y enteros fijados a 72°C y memoria a 70°C. En la figura 7.12 tenemos la evolución de la temperatura a lo largo de la ejecución usando ambos controles.

Como se puede ver al comienzo de la ejecución nuestro control permite una temperatura más alta para tratar de alcanzar la temperatura de consigna. Mas tarde, pese a que el control tiene errores debidos a la alta variabilidad de las pruebas, consigue mantener una temperatura más baja con un poco de impacto en los resultados del benchmark. En la tabla 7.5 se muestra un resumen de los datos en el que se ve el menor número de cambios y la menor temperatura



(a) Sin Supervisor (b) Con Supervisor

Figura 7.11: Temperaturas Benchmark Matrices

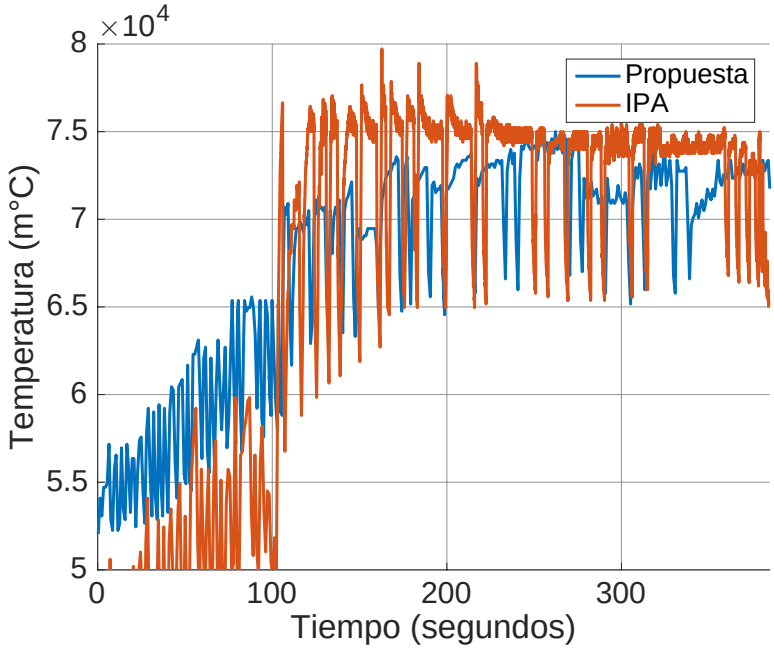


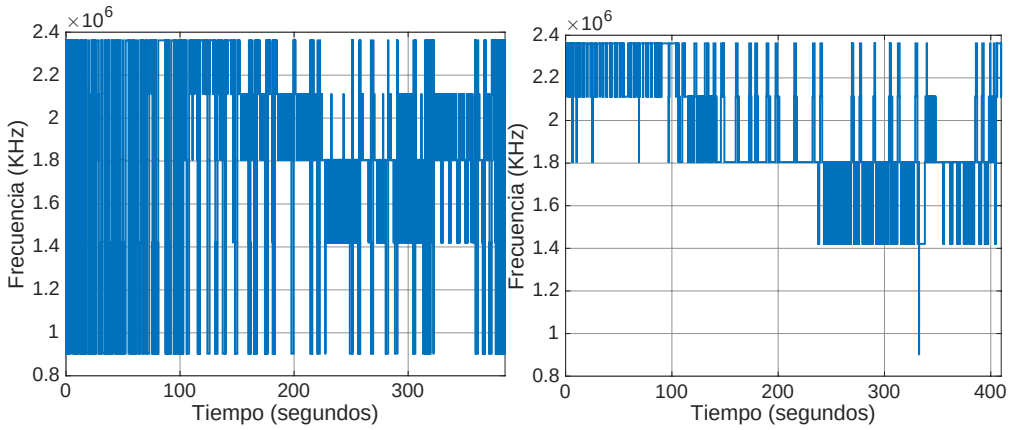
Figura 7.12: Comparación Temperaturas Geeckbench

alcanzada. También se puede ver como el rendimiento pese a ser menor no se ve tan perjudicado como podría parecer por la bajada de la temperatura.

Tabla 7.5: Comparación IPA y nuestra propuesta Geekbench

Medidas	IPA	Propuesta	Diferencia (%)
Puntuación unicore	1894	1766	93.2
Puntuación multicore	2690	2365	87.9
Temperatura media (°C)	73.17	67.77	92.6
Cambios de frecuencia	2600	1000	38.4
Desviación Típica (MHz)	392.8	312.3	79.5

En la figura 7.13 se ve la comparación de las frecuencias seleccionadas por ambos controladores. La gran cantidad de cambios de frecuencia se debe a la separación entre el *governor* térmico y el de frecuencia, y la sobrecarga que esto supone. El térmico selecciona una frecuencia máxima y activa al *governor* de frecuencia que selecciona la frecuencia en base a la carga de trabajo. Esto puede generar que en ocasiones no se llegue a tiempo a acelerar la ejecución, o que la reducción de frecuencia debido a alarmas térmicas sea demasiado alta.



(a) IPA (b) Nuestra propuesta

Figura 7.13: Comparación Frecuencias Benchmark Matrices

Geekbench al tratarse de un benchmark con una carga muy variable no permite ver las diferencias de usar un control más formal. En la tabla 7.6 se muestra la media de instrucciones totales y SIMD ejecutadas por segundo usando IPA y nuestra propuesta de control. Dado que IPA está configurado para limitar la ejecución a 75°C hemos usado nuestro control con dos temperaturas de consigna, 72 y 75°C. Se puede observar que a igualdad de temperatura el control ofrece un rendimiento superior. Esto es debido a los mecanismos que entran en funcionamiento en cada propuesta. IPA al ser un *governor* térmico realiza cálculos de estimación de energía cada vez que la temperatura cambia, luego tiene que actuar el governor de frecuencia y modificar la frecuencia a la deseada o máxima permitida. Nuestra propuesta por el contrario actúa cada periodo de 100ms, donde realiza el cálculo y modifica la frecuencia.

Tabla 7.6: Comparación IPA y nuestra propuesta Geekbench

	Instrucciones ($x10^9$)	Inst. SIMD ($x10^9$)
IPA	2.12	1.39
Propuesta $T^* = 75°C$	2.26	1.48
Propuesta $T^* = 72°C$	2.03	1.33

8 CONCLUSIONES

A lo largo de este proyecto hemos realizado un estudio completo de un control supervisado para la gestión de la temperatura mediante el escalado dinámico de la frecuencia. Hemos realizado el modelado del comportamiento de la plataforma, así como descritos los pasos y herramientas necesarios para replicarlo. Tras entender como se genera este calor se han diseñado y probado una serie de controles para gestionar esta temperatura. El controlador elegido ha sido implementado como módulo de un *kernel* ampliamente usado hoy en día y con mucha extensión.

Durante la fase de modelado se descubrió una gran diferencia de comportamiento entre las diferentes cargas de trabajo. Esto hizo que se completará el controlador con un supervisor que ajustara el controlar en base a la carga de trabajo que se ejecuta. Para poder reconocer la carga del procesador se hizo un análisis de los contadores hardware que terminó consiguiéndose con suficiente precisión.

Este TFM une conocimientos de 2 áreas distintas, arquitectura de computadores y control. De la primera se han consolidado los conocimientos aprendidos durante el grado y máster además de ampliar los conocimientos sobre el kernel de Linux y su programación. De la segunda área, no solo se ha aprendido cosas nuevas asistiendo a clases, sino que lo aprendido se ha consolidado mediante un trabajo completo sobre control exitoso aplicado a procesadores. Por otra parte y no en tan gran medida, ha sido importante el aporte que ha tenido la estadística a la hora de poder aplicar técnicas de clasificación y análisis con un coste computacional bajo. Este trabajo pretende demostrar que el intercambio de conocimientos entre distintas áreas es muy importante a la hora de buscar nuevas soluciones a problemas existentes.

Todos los objetivos han resultado completados, tanto modelado, análisis de contadores, como resultados con el esquema de control propuesto. Estos resultados han demostrado la utilidad de adaptar el control en base a las fases de ejecución, permitiendo un mejor comportamiento y posibilidad de ajuste. En pruebas con una alta variabilidad de carga es capaz de mantener la temperatura en un rango más bajo con una pérdida de rendimiento de 7 %. Con cargas de trabajo más homogéneas se han conseguido speedups de hasta el 6 %. Además se ha abierto puertas a otros tipo de controles que puedan aprovechar también la clasificación de fases de programas.

Para trabajos futuros como hemos dicho se podría ampliar el campo de estudio de control por tipo de programa. Además se podría trabajar en modificar Linux para permitir el descubrimiento automático de zonas de temperatura, así como de los contadores hardware de cada plataforma para realizar la clasificación.

Todo el código utilizado se encuentra disponible en un repositorio de Github [11].

BIBLIOGRAFÍA

[1] J. Alastruey, J. L. Briz, P. Ibanez y V. Vinals. «Software Demand, Hardware Supply». En: *IEEE Micro* 26.4 (2006), págs. 72-82. ISSN: 0272-1732. DOI: 10.1109/MM.2006.80.

[2] Dieter Bohn. *Apple confirms MacBook Pro thermal throttling, software fix coming today*. https://web.archive.org/web/20190406051030/https://www.theverge.com/2018/7/24/17605652/macbook-pro-thermal-throttling-apple-software-fix. [Online; publicado 24-Julio-2018; accedido 7-Junio-2019].

[3] Canonical. *Stress-ng*. URL: https://wiki.ubuntu.com/Kernel/Reference/stress-ng.

[4] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch y Ricardo Bianchini. «CoScale: Coordinating CPU and Memory System DVFS in Server Systems». En: *MICRO*. 2012, págs. 143-154.

[5] Ashutosh S. Dhodapkar y James E. Smith. «Comparing Program Phase Detection Techniques». En: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, págs. 217-. ISBN: 0-7695-2043-X. URL: http://dl.acm.org/citation.cfm?id=956417.956539.

[6] Gene F. Franklin, David J. Powell y Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. 4th. Prentice Hall PTR, 2001.

[7] Gaël Guennebaud, Benoît Jacob y col. *Eigen v3*. URL: http://eigen.tuxfamily.org.

[8] Greg Hamerly, Erez Perelman, Jeremy Lau y Brad Calder. «Simpoint 3.0: Faster and more flexible program phase analysis». En: *Journal of Instruction Level Parallelism* 7.4 (2005), págs. 1-28.

[9] Pablo Hernández Almudi, Eduardo Montijano Muñoz y Darío Suárez Gracia. «Diseño de un Governor basado en control inteligente de temperatura». En: (2017).

[10] Pablo Hernández. *Merged Pull Request*. URL: https://github.com/ARM-software/workload-automation/pull/953.

[11] Pablo Hernández. *sPIDer*. URL: https://github.com/Pablololo12/sPIDer.

[12] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose y Margaret Martonosi. «An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget». En: *MICRO*. 2006, págs. 347-358.

[13] Alexey Kopytov. *Sysbench*. URL: https://github.com/akopytov/sysbench.

[14] PRIMATE LABS. *Geekbench*. URL: https://www.geekbench.com/.

[15] A. Leva, F. Terraneo, I. Giacomello y W. Fornaciari. «Event-Based Power/Performance-Aware Thermal Management for High-Density Microprocessors». En: *IEEE Transactions on Control Systems Technology* 26.2 (2018), págs. 535-550.

[16] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal y A. Leva. «Controlling software applications via resource allocation within the heartbeats framework». En: *CDC*. 2010, págs. 3736-3741.

[17] T. Mudge. «Power: a first-class architectural design constraint». En: *Computer* 34.4 (2001), págs. 52-58. ISSN: 0018-9162. DOI: 10.1109/2.917539.

[18] Jihoon Park, Seokjun Lee y Hojung Cha. «App-Oriented Thermal Management of Mobile Devices». En: *ISLPED*. 2018, 36:1-36:6.

[19] R. P. Pothukuchi, A. Ansari, P. Voulgaris y J. Torrellas. «Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures». En: *ISCA*. 2016, págs. 658-670.

[20] A. Rahmani, M. Haghbayan, A. Kanduri, A. Y. Weldezion, P. Liljeberg, J. Plosila, A. Jantsch y H. Tenhunen. «Dynamic power management for many-core platforms in the dark silicon era: A multi-objective control approach». En: *ISLPED*. 2015, págs. 219-224.

[21] Amir M. Rahmani, Bryan Donyanavard, Tiago Mück, Kasra Moazzemi, Axel Jantsch, Onur Mutlu y Nikil Dutt. «SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management». En: *ASPLOS*. 2018, págs. 169-183.

[22] George Stephanopoulos. *Chemical process control: an introduction to theory and practice*. 1984.

[23] *The Linux Kernel Module Programming Guide*. http://www.tldp.org/LDP/lkmpg/2.6/html/.

[24] Xin Wang. *Intelligent Power Allocator*. Inf. téc. ARM, mar. de 2017.

[25] Neil Weste y David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. 4th. Addison-Wesley Publishing Company, 2010.

[26] W. Yueh, Z. Wan, Y. Joshi y S. Mukhopadhyay. «Experimental characterization of in-package microfluidic cooling on a System-on-Chip». En: *ISLPED*. 2015, págs. 43-48.

[27] arm. *Workload Automation*. URL: https://github.com/ARM-software/workload-automation.

Parte I

APPENDIX

A

DISCRETIZACIÓN CONTROL PID

El método que vamos a usar para discretizar es la transformación en Z por el método de polos y ceros. Esto es, buscamos que la nueva ecuación tenga los mismos polos y ceros que la primera, haciendo la equivalencia,

$$z = e^{sTs}. \tag{A.1}$$

Donde Ts es el tiempo de muestra, s variable compleja de la transformada de Laplace y z la variable compleja de la transformada en z. Por tanto buscamos los polos y ceros y encontramos sus equivalentes,

$$\begin{aligned} \text{Polo: } s = 0 &\rightarrow z = e^{0Ts} = 1 \\ \text{Cero: } s = \frac{-1}{Ti} &\rightarrow z = e^{\frac{-Ts}{Ti}} = Q, \end{aligned} \tag{A.2}$$

y formamos la ecuación en espacio z,

$$PI(z) = K_z \frac{z - Q}{z - 1}. \tag{A.3}$$

Ahora necesitamos obetener el valor de K_z, para ello, y como ambas ecuaciones tienen que llegar al mismo estado estacionario buscamos que sus límites cuando tiendan a infinito sean iguales,

$$\begin{aligned} \lim_{s \to 0} sPI(s) &= \frac{K}{Ti} \\ \lim_{z \to 1} \frac{z-1}{Ts} PI(z) &= \frac{1 - Q}{Ts}, \end{aligned} \tag{A.4}$$

por tanto,

$$K_z = \frac{\frac{K}{Ti}}{\frac{1-Q}{Ts}} = \frac{KTs}{Ti - TiQ}. \tag{A.5}$$

El siguiente paso es, dado que la fórmula nos representa la relación entre las acciones y las entradas, igualar a acciones $U(z)$ y entradas $E(z)$. Además dividimos por el término z de mayor exponente para llevar todos los valores hacia el pasado, de este modo podremos hacer la relación $z^{-1} = k - 1$. El desarrollo quedaría,

$$\begin{aligned} K_z \frac{z - Q}{z - 1} = \frac{U(z)}{E(z)} \implies zU(z) - U(z) &= K_z zE(z) - K_z QE(z) \implies \\ \frac{zU(z) - U(z)}{z} &= \frac{K_z zE(z) - K_z QE(z)}{z} \implies \\ U(z) - z^{-1}U(z) &= K_z E(z) - K_z Qz^{-1}E(z) \implies \\ U(k) &= U(k-1) + K_z E(k) - K_z QE(k-1) \implies \\ U(k) = U(k-1) + \frac{KTs}{Ti - Tie^{\frac{-Ts}{Ti}}} E(k) &- \frac{KTse^{\frac{-Ts}{Ti}}}{Ti - Tie^{\frac{-Ts}{Ti}}} E(k-1). \end{aligned} \tag{A.6}$$

Simplificando la fórmula tenemos que el control del incremento de frecuencia con respecto a f_0, a un tiempo discreto k, se reduce a

$$f(k) = FF(T^*(k) - T_0) + f(k-1) + \delta_1 e(k) + \delta_2 e(k-1), \tag{A.7}$$

donde $e(k) = T^*(k) - T(k)$ es el error entre la temperatura deseada y la obtenida en el instante k, y δ_i parámetros que solo requieren ser calculados al arranque del control mediante la fórmula antes calculada dado un tiempo de muestreo Ts tal y como podemos ver en las fórmulas siguientes

$$\begin{aligned} \delta_1 &= \frac{KTs}{Ti - Tie^{\frac{-Ts}{Ti}}} \\ \delta_2 &= -\frac{KTse^{\frac{-Ts}{Ti}}}{Ti - Tie^{\frac{-Ts}{Ti}}}. \end{aligned} \tag{A.8}$$

B

ARM PMU

B.1 INTRODUCCIÓN

Los procesadores modernos incluyen hardware especial para contar los diferentes tipos de eventos que ocurren en ellos. Eventos como el número de ciclos, saltos, accesos a memoria,... Estos eventos permiten a los usuarios realizar análisis y ajuste del rendimiento de las aplicaciones ejecutadas. En los procesadores actuales estos contadores están gestionados desde una unidad especial, Intel y AMD la llaman PCM (Performance Counter Monitor) Y ARM PMU (Performance Management Unit). En este documento presentamos la tercera versión de la unidad de contadores de ARM, la PMUv3, presente en las nuevas arquitecturas ARMv8-A.

B.2 ARQUITECTURA

En la figura B.1 se muestran los principales componentes de la unidad. Cuenta con 7 contadores de 32 bits conectados a los diferentes eventos que se pueden contar en la cpu. 6 de estos contadores son configurables en cuanto a que evento se quiera registrar, menos uno de ellos que solo puede contar ciclos de reloj. En la configuración se puede seleccionar no solo el evento sino también si queremos que se genere una interrupción cada vez que se alcanza un valor deseado del contador. Esto es especialmente útil a la hora de tener en cuenta overflows o para controlar la ejecución mediante watchdogs.

B.3 CONFIGURACIÓN

Para la configuración el PMU actúa como un coprocesador que muestra una serie de registros que pueden ser accedidos mediante instrucciones especiales del procesador. El registro principal es el PMCR, este da información sobre el fabricante bits[31:24], el modelo de procesador bits[23:16] y el numero de contadores de eventos disponibles bits[15:11]. Se usa también para realizar ajustes generales de la unidad siendo los bits más importantes,

- *bit[2] Clock Counter Reset* cuando se escribe 1 resetea el contador de reloj (PMCCNTR)
- *bit[1] Event Counter Reset* cuando se escribe 1 resetea todos los contadores de eventos
- *bit[0] Enable* con 1 abilita todos los contadores, con 0 los desabilita

Otros registros importantes son los mostrados en la tabla B.1.
Para realizar la cuenta de un evento los pasos a seguir serían:

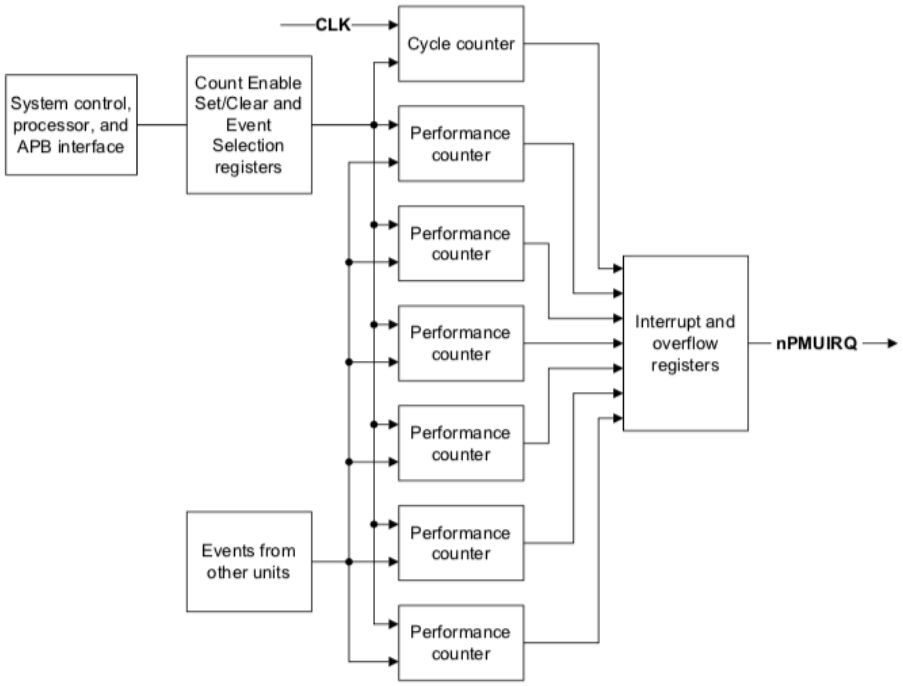


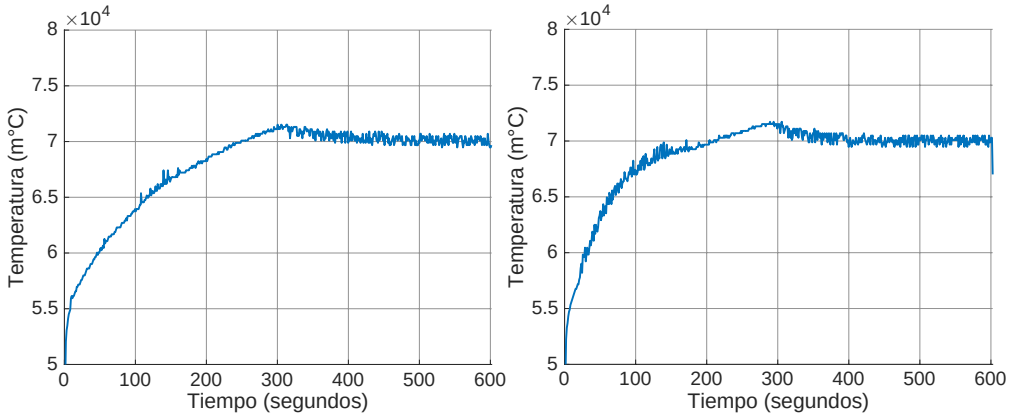
Figura B.1: Bloques principales PMUv3. Imagen extraida de ARM Cortex-A73 MPCore Processor. Technical Reference http://infocenter.arm.com/help/topic/com.arm.doc.100048_0100_06_en/cortex_a73_trm_100048_0100_06_en.pdf

Tabla B.1: Listado de los registros importantes de PMU

PMCR	PMU registro principal
PMCNTENSET	cada uno de los bits de este registro activa un contador de eventos, por ejemplo el bit[1] activa el contador 1
PMCNTENCLR	como PMCNTENSET, pero cada bit desactiva un contador
PMSELR	selecciona un contador de eventos
PMXEVTYPER	sirve para seleccionar que evento debe contar el contador seleccionado por PMSELR
PMXEVCNTRo	almacena el valor del contador seleccionado por PMSELR

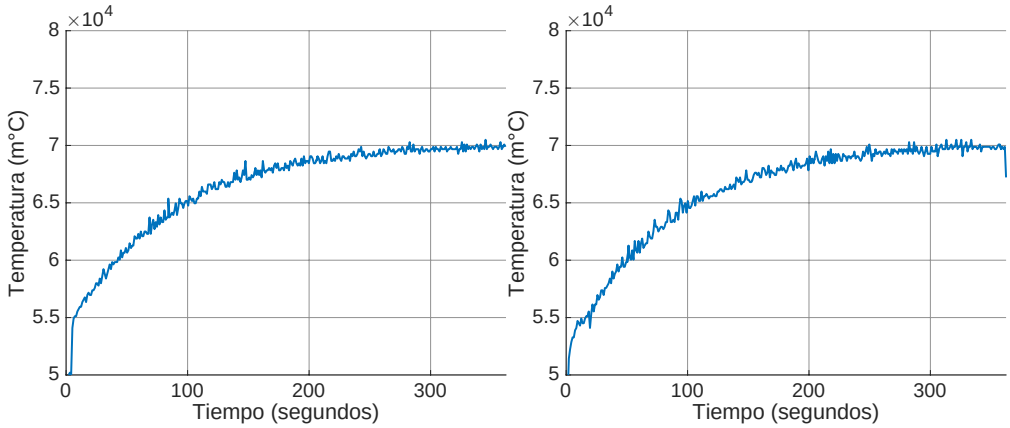
```
Escribir 0b0110 en PMCR para resetear todos los contadores
Escribir en PMSELR el numero del contador a usar
Escribir en PMXEVTYPER el ID del evento que se desea contar
Escribir en PMCNTENSET un uno en el bit con indice igual al del
    contador seleccionado

// Codigo a contar

Escribir en PMCNTENCLR un 1 en el mismo bit que el usado en PMCNTENSET
    para detener el contador
```

```
Escribir en PMSELR el numero del contador a leer
Leer de PMXEVCNTR0 el valor del contador
```



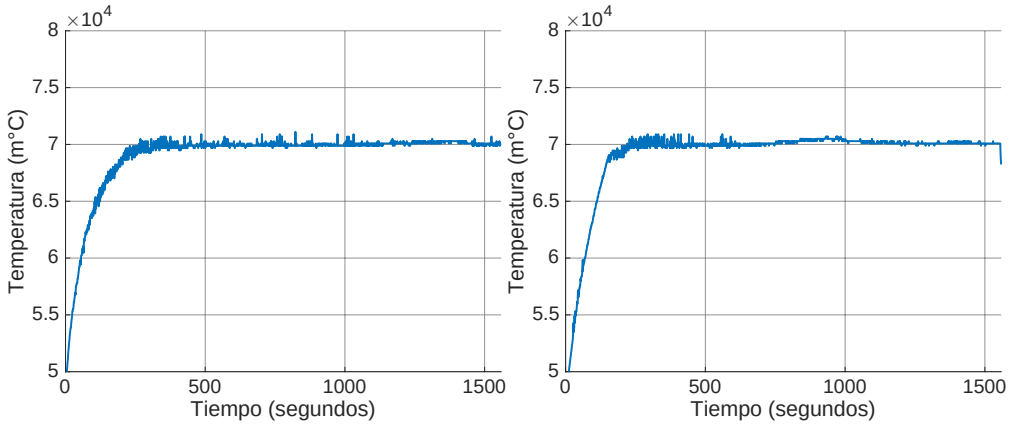
(a) Sin Supervisor (b) Con Supervisor

Figura C.1: Temperaturas Dhrystone



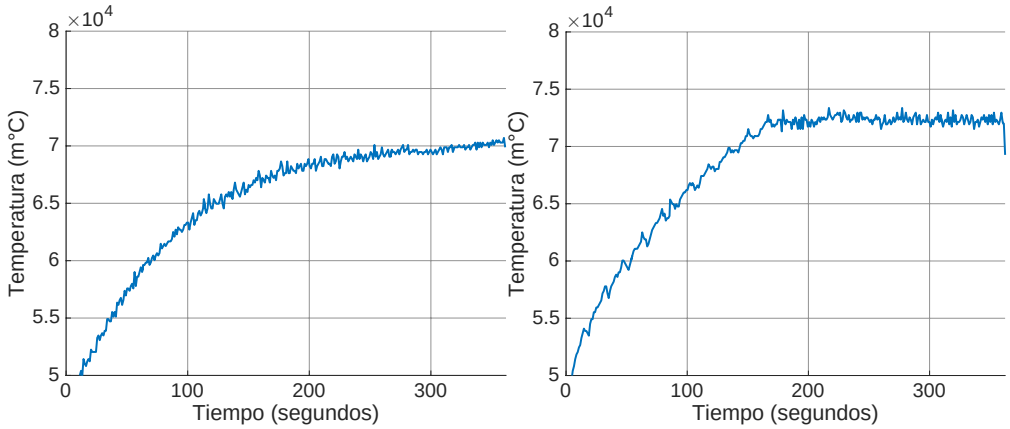
(a) Sin Supervisor (b) Con Supervisor

Figura C.2: Temperaturas Stress Entero



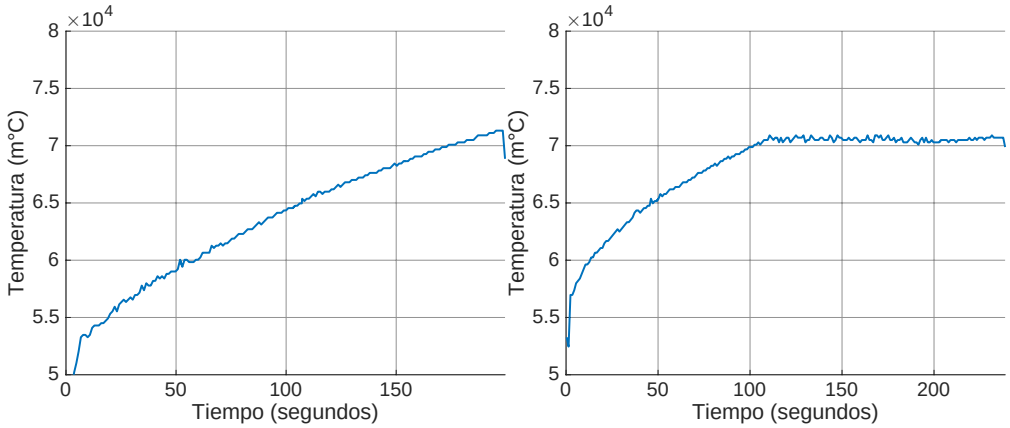
(a) Sin Supervisor (b) Con Supervisor

Figura C.3: Temperaturas Benchmark Matrices



(a) Sin Supervisor (b) Con Supervisor

Figura C.4: Temperaturas Stress Flotante



(a) Sin Supervisor (b) Con Supervisor

Figura C.5: Temperaturas Memcpy