Universidad Zaragoza

Proyecto Fin de Carrera
Ingeniería en Informática

Impacto de las optimizaciones de compilación en la energía, potencia y temperatura: el caso Intel Pentium 4

Raúl Ceresuela Palomera

Codirectores: Rubén Gran Tejero y Darío Suárez Gracia

Departamento de Informática e Ingeniería de Sistemas
Centro Politécnico Superior
Universidad de Zaragoza

Curso 2010/2011
Septiembre 2011

Resumen

Uno de los principales objetivos al compilar un programa es reducir su tiempo de ejecución: las opciones de optimización están orientadas mayoritariamente a reducir dicho tiempo sin considerar ningún otro objetivo. Pero por otro lado, factores como el consumo energético y la disipación de potencia están cobrando una importancia cada vez mayor. El consumo energético, por ejemplo, supone una parte muy importante del coste de las empresas informáticas y de los grandes centros de datos. Y no sólo el coste económico es importante, también el impacto ecológico. Uno de los términos más en boga en el mundo de la informática desde hace unos años es el de *green computing*. Es decir, el uso eficiente de los recursos computacionales, minimizando el impacto ambiental y maximizando su viabilidad económica.

Por otra parte, la disipación de potencia se ha convertido en un factor limitante en el diseño de los procesadores. Dicha limitación llega a condicionar el rendimiento que puede ofrecer el procesador, tanto en entornos de altas prestaciones (debido al coste del equipo disipador) como en entornos móviles (duración de la batería). Determinar cómo afectan las distintas opciones de compilación al consumo de energía del procesador o a la disipación de potencia no puede hacerse sólo mediante simulaciones. Hay efectos secundarios como: el consumo energético de la señal del reloj, el consumo debido a las corrientes de fuga (cada vez más significativo debido al descenso de la tensión de alimentación y de la tensión umbral) o la relación existente entre el aumento de la temperatura y la disipación de potencia que deben medirse *in situ* para determinarse.

Por ello en este proyecto hemos estudiado en el laboratorio el impacto de distintas opciones de compilación, analizando cómo afectan al consumo energético, la disipación de potencia y la temperatura en un procesador Intel Pentium 4. Para ello, hemos hecho uso de la plataforma de medición de energía y temperatura de la que dispone el grupo de Arquitectura de Computadores de la Universidad de Zaragoza (gAZ). Dicha plataforma permite monitorizar las mediciones de manera no invasiva. De esta manera, no se alteran los resultados de las mediciones.

Con los resultados de este proyecto, tenemos una lista de las optimizaciones que mejoran el rendimiento desde el punto de vista del consumo energético, el aumento de la temperatura o la disipación de potencia. Este análisis podría servir de base para elaborar unos posibles *flags* de compilación como -Oenergy, -Opower y -Otemp; que incluyeran las mejores optimizaciones desde el punto de vista de la energía, la potencia y la temperatura, respectivamente.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1
Introducción

La industria de los microprocesadores ha participado durante más de 30 años en una carrera por aumentar el rendimiento en los procesadores. Este esfuerzo ha permitido multiplicar la potencia computacional en varios órdenes de magnitud. Por ejemplo, el escalado CMOS ha permitido aumentar la frecuencia y la capacidad de integración de los circuitos (según la Ley de Moore, cada 2 años se ha duplicado el número de transistores en un circuito integrado). Pero esta impresionante mejora ha tenido un gran impacto en el consumo energético y en la disipación de potencia. Tanto es así, que hace tiempo que se alcanzó el punto en el que la disipación de potencia es uno de los factores limitantes en el diseño de un microprocesador [16, 6].

Pero el consumo energético y la disipación de potencia no sólo afectan al diseño del procesador, también al funcionamiento y rendimiento de los dispositivos que utilizan microprocesadores. Por un lado, en los sistemas empotrados (por ejemplo, la telefonía móvil) reducir el consumo energético del procesador aumenta la duración de la batería, permitiendo que el dispositivo funcione más tiempo sin la necesidad de una recarga. Por otro lado, tanto en los computadores comerciales como en los entornos de altas prestaciones (supercomputadores), el rendimiento puede degradarse si el sistema disipador no puede refrigerar correctamente y disipar todo el calor producido por las elevadas potencias. El procesador se ve forzado, entonces, a reducir su frecuencia y voltaje.

A pesar de la importancia técnica que tiene la reducción del consumo energético en la fabricación de microprocesadores, su importancia va más allá. Se ha convertido en un objetivo colectivo no sólo para las compañías sino también para la sociedad. Por ejemplo, *Green computing* aboga por una computación y unas comunicaciones sostenibles, con el menor impacto posible para el medio ambiente (huella energética). Junto con conceptos como el coste total de propiedad (que incluye el coste de eliminación de residuos y reciclaje), *Green computing* incluye como punto clave la eficiencia energética.

La disipación de potencia en los microprocesadores de propósito general ha alcanzado un punto en el que el problema tiene que abordarse desde varios niveles del diseño del sistema. Existen muchas técnicas para reducir la potencia a nivel tecnológico, arquitectural, a nivel de sistema operativo y a nivel algorítmico. En este proyecto fin de carrera se estudia cómo el código de los programas generado por el compilador afecta al consumo de energía, la disipación de potencia y la temperatura. En concreto, se examina qué impacto tienen las distintas optimizaciones del compilador sobre la energía y la potencia en un procesador moderno de propósito general como el Intel Pentium 4.

Un compilador es un programa informático que traduce un programa escrito en un lenguaje

de programación (origen) a otro lenguaje (destino), generando un programa equivalente que el procesador será capaz de ejecutar. Usualmente el lenguaje destino es lenguaje máquina, pero también puede traducirse a otro lenguaje de programación. Esta traducción, o compilación, se realiza en varias fases. Primero se realizan las fases de análisis léxico, sintáctico y semántico. A continuación, se transforma en un lenguaje intermedio, que por norma general es independiente de la arquitectura sobre la que se ejecutará el programa final. Sobre ese lenguaje intermedio se aplican optimizaciones de compilación que mejoren dicho código de tal forma que se ejecute lo más rápido posible. Después de la fase de optimización, el compilador traduce el lenguaje intermedio al código máquina que se ejecutará finalmente en el procesador.

El objetivo de las optimizaciones de compilación es reducir el tiempo de ejecución de los programas, normalmente sin considerar ningún otro objetivo. En compiladores como *gcc*, *icc* y *llvm* pueden aplicarse varias optimizaciones en bloque al activar los *flags* -O1, -O2 o -O3. A mayor nivel de optimización aplicado, se aplican mayor número de optimizaciones y también más agresivas, haciendo que el tiempo de compilación aumente pero generando un programa que se ejecuta más rápido. Con las medidas realizadas en este proyecto fin de carrera analizamos las optimizaciones de compilación desde la óptica del consumo energético, la disipación de potencia y la temperatura, en vez de hacerlo desde el punto de vista del tiempo de ejecución. Este análisis podría servir de base para elaborar unos posibles *flags* de compilación como -Oenergy, -Opower y -Otemp; que incluyeran las mejores optimizaciones desde el punto de vista de la energía, la potencia y la temperatura, respectivamente.

1.1 Contexto del proyecto

Para la realización de este proyecto fin de carrera se ha hecho uso de la plataforma de medición de potencia y temperatura de la que dispone el Grupo de Arquitectura de Computadores de la Universidad de Zaragoza (gaZ) en su laboratorio de investigación en el Centro Politécnico Superior. Dicha plataforma realiza una monitorización de manera no invasiva para no alterar los resultados obtenidos.

Anteriormente, en esta plataforma, se han realizado otros proyectos fin de carrera que estudiaban el consumo energético y la disipación de potencia [3, 5, 10]. Estos proyectos han estudiado qué efecto tienen los distintos niveles de optimización (-O1, -O2 y -O3) sobre el consumo energético y el aumento de temperatura del código optimizado. Con este proyecto fin de carrera se pretende estudiar con una granularidad más fina el efecto que tienen las optimizaciones de compilación. Para ello, se estudia cómo afectan al consumo energético, la disipación de potencia y la temperatura diversas optimizaciones de compilación aplicadas individualmente, en vez de hacerse en bloques o niveles de optimización.

1.2 Objetivos

El objetivo de este proyecto es estudiar el impacto de las optimizaciones de compilación desde el punto de vista de la energía, potencia y temperatura. Las tareas principales en las que se puede dividir este proyecto son:

1. Comprender los fundamentos del consumo energético y su relación con el rendimiento del procesador.

2. Estudiar los fundamentos teóricos de las principales optimizaciones del rendimiento existentes, así como cuáles de ellas han sido implementadas en compiladores de uso extendido.

3. Buscar o elaborar programas de prueba que permitan medir las optimizaciones analizadas.

4. Medir, con ayuda de la plataforma de medición, el tiempo de ejecución, la energía consumida, la potencia disipada y la temperatura al aplicar las optimizaciones.

5. Análisis de resultados.

1.3 Tecnología utilizada

Para la realización de este proyecto se han utilizado las siguientes tecnologías:

- El compilador *gcc* (GNU Compiler Collection) en su versión 4.5.2 [8] para compilar el código y aplicar las optimizaciones.
- El lenguaje de programación *C* para elaborar los programas de prueba.
- El editor *vi*.
- Un sistema operativo *Linux* (*Ubuntu* personalizado) para ejecutar los programas que se querían medir.
- La plataforma de medida del laboratorio del gaZ [11].
- LaTeX [18] para la redacción de este documento.
- El programa *gnuplot* [9] para realizar las gráficas de esta memoria.
- El programa *Dia* [17] para editar los diagramas e imágenes de la memoria.

1.4 Organización de la memoria

Este documento se organiza de la siguiente manera:

Capítulo 2 Metodología. En este capítulo se presenta la metodología seguida para hacer las mediciones.

Capítulo 3 Relación entre energía, potencia, tiempo de ejecución y temperatura. En este capítulo se presentan brevemente las relaciones existentes entre energía, potencia, tiempo de ejecución y temperatura.

Capítulo 4 Impacto de las optimizaciones de compilación en la energía, potencia y temperatura. En este capítulo se presentan y se comentan los resultados más significativos obtenidos en las mediciones.

Capítulo 5 Conclusiones. En este capítulo se presentan las conclusiones obtenidas al realizar el proyecto.

Se incluyen como apéndices:

A. Plataforma de medida.

B. Metodología (ampliación del capítulo 2).

C. Impacto de las optimizaciones de compilación en la energía, potencia y temperatura (ampliación del capítulo 4).

D. Condiciones en las que se han realizado los experimentos.

E. Código fuente utilizado en las pruebas de laboratorio.

F. Gestión del proyecto: control de esfuerzos.

1.5 Agradecimientos

Quiero agradecer a mis directores del proyecto, Darío y Rubén, la ayuda prestada. Tanto para resolver las dudas que me surgían durante la realización del proyecto, como por las numerosas revisiones detalladas de este documento.

También a mis amigos y compañeros de fatigas, en especial a Adrián, José, Edu, Javi y César. Por los buenos momentos y los malos cafés.

Por último, y no menos importante, a mis padres. Por decidir, en su momento, que ahorrar algo de dinero para que sus hijos fueran a la Universidad era una buena idea. Y lo ha sido.

Capítulo 2
Metodología

En este capítulo se comenta la metodología seguida al realizar este proyecto. Abarca las fases de estudio previo, elección del compilador, prueba de la plataforma, elección de los programas de prueba y la realización de las mediciones. En el capítulo se presentan de manera resumida parte de las fases. Para una explicación más detallada hay que consultar el apéndice B.

2.1 Estudio previo

La primera tarea que se abordó fue la del estudio de distintas optimizaciones de compilación [1, 4, 2] y también del consumo de energía, la disipación de potencia y la temperatura en los procesadores [22, 19].

2.2 Elección del compilador

A continuación, se estudió qué *flags* de los compiladores activaban las distintas optimizaciones. Para ello, se analizaron tres compiladores de uso extendido: *gcc* (versión 4.5.2) [8], *icc* (versión 10.1) [7] y *llvm* (versión 2.7) [20].

Tras estudiar sus manuales de uso, se decidió descartar al compilador *icc* ya que no presentaba un control individual sobre cada optimización de compilación. La única manera de activar las optimizaciones es haciéndolo por bloques, es decir, activando alguno de los niveles generales de optimización como -O1, -O2 o -O3. Al no permitir un control más fino sobre las optimizaciones, no se puede medir el efecto que cada una de ellas tiene en el consumo energético, la disipación de potencia y la temperatura.

En cuanto a *gcc* y *llvm*, ambos permiten controlar individualmente las optimizaciones. Pero nos quedamos con *gcc* porque es más completo, ya que con sus *flags* permite aplicar más optimizaciones.

Una característica de *gcc* que resultó problemática a la hora de realizar este proyecto es el hecho de que para activar una optimización de compilación, tiene que estar activo uno de los niveles de optimización (-O1, -O2, -O3, -Os). Es decir, para aplicar una optimización cualquiera tenemos que activar más optimizaciones de las deseadas, incluso aunque utilicemos el nivel mínimo de optimización -O1. Este problema se solventó a la hora de realizar las mediciones con el uso de programas muy específicos (en los que sólo cabía aplicar la optimización deseada) y también

con el uso de los *flags* de desactivación de las optimizaciones cuando fue necesario (*flags* tipo -fno-*optimización*).

2.3 Elección de los programas de prueba

Para aplicar una optimización hace falta que el código sobre el que quiere aplicarse cumpla unas determinadas condiciones. Por ello, en vez de utilizar *benchmarks* genéricos como *SPEC*, se ha decidido utilizar *benchmarks* sintéticos. Es decir, pequeños programas de prueba que contengan fragmentos de código fácilmente optimizables. Además, al ser muy específicos, limitan la posibilidad de que se apliquen otras optimizaciones no deseadas al activar algún nivel de optimización. Para que la medición de los programas fuera más fiable, el código optimizable se ha colocado en el interior de un bucle. De esa manera se repite varias veces la ejecución del código que interesa medir.

Estos pequeños programas de prueba se han realizado utilizando, en la mayoría de las ocasiones, el código de ejemplo de "Compiler transformations for high-performance computing" [4]. Cuando no ha sido posible aplicar la optimización a ese código, se ha acudido al *testsuite* de *gcc* y se ha utilizado directamente el código C que utiliza *gcc* para comprobar las optimizaciones en el momento de instalarlo.

Para medir el efecto que tiene la optimización hacen falta dos versiones de los programas. Una versión básica u optimizada con el mínimo nivel posible y otra en la que se aplica la optimización deseada. Tras generar las dos versiones con *scripts* que automatizan el proceso, hay que estudiar el código ensamblador generado para aseguarse de que la optimización se está aplicando; y entonces, puede realizarse la medición del consumo energético, la disipación de potencia y la temperatura.

Capítulo 3
Relación entre energía, potencia, tiempo de ejecución y temperatura

En este capítulo se hace una breve introducción a la relación existente entre tiempo de ejecución, energía, potencia y temperatura. El objetivo de esta introducción es repasar conceptos que se manejarán posteriormente a la hora de analizar los resultados obtenidos en las mediciones.

3.1 Tiempo de ejecución

Un procesador es el componente del ordenador que ejecuta las instrucciones de los programas y procesa los datos. Los procesadores son el componente básico de los ordenadores, junto con la memoria y los dispositivos de entrada/salida. El procesador se construye mediante un circuito integrado. En el caso del Intel Pentium 4 (procesador sobre el que se realizan las mediciones de los experimentos), es un circuito integrado de tecnología CMOS. Este circuito puede ser modelado con una capacitancia equivalente C_{tot} y una resistencia equivalente R_{tot} de todo el procesador, resultando un modelo sencillo del mismo (ver figura 3.1).

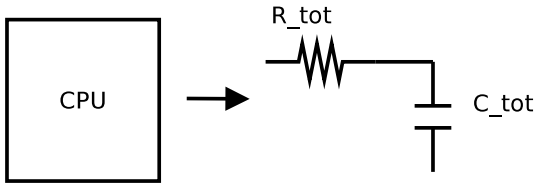

Figura 3.1: Simplificación de la CPU en un circuito electrónico equivalente.

En un procesador, el tiempo de ejecución T_{ex} de un programa se puede expresar como

$$T_{ex} = N_{inst} * CPI * T_{ciclo} \tag{3.1}$$

donde N_{inst} es el número total de instrucciones ejecutadas, CPI representa el número medio de ciclos por instrucción y T_{ciclo} es el tiempo de ciclo del procesador. Para reducir T_{ex} los compiladores tienen que reducir el número total de ciclos, es decir, el producto $N_{inst} * CPI$.

3.2 Potencia

La potencia disipada es proporcional a la corriente de alimentación $i_{DD}(t)$ y la tensión de alimentación V_{DD}:

$$P(t) = i_{DD}(t)V_{DD} \tag{3.2}$$

La disipación de potencia en los circuitos CMOS tiene dos componentes [22, 16]: la disipación estática y la dinámica. La disipación estática se produce por el mero hecho de tener conectado el circuito. Idealmente, los transistores conducen la corriente cuando están en modo de saturación y no la conducen cuando están en modo de corte. Pero, en realidad, siempre hay una pequeña corriente de fuga que lo atraviesa y causa el consumo estático. La principal corriente de fuga es la que se produce entre fuente y drenador. Por ejemplo, en la figura 3.2 tenemos un inversor construido con tecnología CMOS. Si la entrada es "0", el transistor pMOS está en modo de saturación. Eso hace que la salida quede conectada a la alimentación V_{DD} y tengamos un "1" en la salida. El transistor nMOS está en modo de corte e, idealmente, no debería conducir ninguna corriente; pero una pequeña corriente I_{fuga} cruza del fuente al drenador, haciendo que se produzca una disipación de potencia estática.

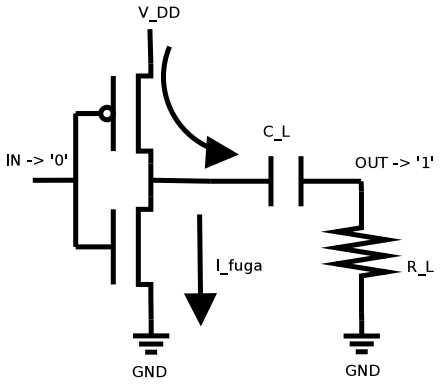

Figura 3.2: Inversor CMOS.

Asumiendo que las corrientes de fuga son constantes, la disipación de potencia estática es el producto del total de las corrientes de fuga I_{fuga} y la tensión de alimentación V_{DD}:

$$P_{est} = I_{fuga}V_{DD} \tag{3.3}$$

En cuanto a la disipación dinámica, la componente más importante es la carga y descarga de la capacitancia del circuito. Suponiendo que una capacitancia C_L se carga y descarga entre V_{DD} y tierra (GND) a una frecuencia f. Dado un intervalo de tiempo T, se cargará y descargará un número $T * f$ de veces. La corriente fluye de V_{DD} a las puertas CMOS durante la carga, y de las puertas a GND durante la descarga. En un ciclo de carga/descarga, una carga total de $Q = C_L V_{DD}$ se transfiere de V_{DD} a GND.

La potencia dinámica puede formularse como:

$$P_{din} = C_L V_{DD}^2 f \tag{3.4}$$

La suma de la potencia estática (ecuación 3.3) y la potencia dinámica (ecuación 3.4) da como resultado la potencia total disipada:

$$P_{tot} = P_{din} + P_{est} = C_L V_{DD}^2 f + V_{DD} I_{fuga} \tag{3.5}$$

3.3 Temperatura

En las ecuaciones 3.1 y 3.5 podemos observar que las optimizaciones del compilador sólo afectan a la potencia indirectamente. Por un lado, en la potencia dinámica P_{din} es difícil establecer una relación entre N_{inst} y C_L porque el hecho de ejecutar más, menos, o diferentes instrucciones puede (o no) modificar la actividad realizada por ciclo. Por otro lado, CPI tiene mayor impacto en la potencia dinámica (C_L) porque las optimizaciones que aumentan o descienden el paralelismo a nivel de instrucción (ILP), pueden incrementar o decrementar la actividad por ciclo C_L.

La potencia estática P_{est} se ve menos afectada por las optimizaciones del compilador ya que depende sobretodo de parámetros tecnológicos, en este caso los de la familia lógia CMOS, a la que pertenece el procesador Intel Pentium 4. Sin embargo, P_{est} puede verse afectada cuando las optimizaciones incrementan o decrementan la actividad del procesador, y esto hace que la temperatura del procesador varíe. Si aumenta la temperatura del procesador y no se puede disipar correctamente, provocará un aumento de las corrientes de fuga. Esa I_{fuga} afecta directamente a la potencia estática, como se ha visto en la ecuación 3.5. Al aumentar la potencia estática P_{est}, aumenta también la potencia total P_{tot} y realimenta el problema. En la fórmula 3.6 puede verse el círculo vicioso que se produce entre la potencia y la temperatura [14].

$$P_{tot} \uparrow \longrightarrow Temperatura \uparrow \longrightarrow I_{fuga} \uparrow \longrightarrow P_{est} \uparrow \longrightarrow P_{tot} \uparrow \tag{3.6}$$

3.4 Energía

La energía consumida en un intervalo de tiempo T es la integral de la potencia disipada:

$$E = \int_0^T i_{DD}(t) V_{DD} \, \mathrm{d}t \tag{3.7}$$

Multiplicando en la ecuación 3.5 la potencia total media disipada P_{tot} por T_{ex} tenemos la energía consumida durante la ejecución de un programa:

$$E_{tot} = P_{tot} * T_{ex} = E_{din} + E_{est} = C_{tot} V_{DD}^2 + V_{DD} I_{fuga} T_{ex} \tag{3.8}$$

donde C_{tot} es la capacitancia total medida a lo largo de todos los ciclos de ejecución (ver figura 3.1), que es equivalente a:

$$C_{tot} = C_L * N_{inst} * CPI \tag{3.9}$$

Capítulo 4
Impacto de las optimizaciones de compilación en la energía, potencia y temperatura

En este capítulo se presentan algunas de las distintas optimizaciones de compilación estudiadas junto con los resultados obtenidos en las pruebas de laboratorio que se consideran más significativos. El resto de optimizaciones, junto con sus resultados se incluyen en el apéndice C.

4.1 Introducción

Las optimizaciones estudiadas son las que aparecen en el artículo "Compiler Transformations for High-Performance Computing" [4]. El primer bloque estudiado es el de *Data-Flow-Based Loop Transformations* que contiene las optimizaciones *loop-based strength reduction*, *induction variable elimination*, *loop-invariant code motion* y *loop unswitching*. Después se estudia el bloque de *Loop Reordering* con la optimización *loop tiling*. En tercer lugar, se estudia el bloque de *Loop Restructuring* con la optimización *loop unrolling*. A continuación, el bloque de *Memory Access Transformations* con la optimización de *scalar replacement*. Después de ese bloque se presenta el de *Partial Evaluation* con las optimizaciones de *constant propagation* y *copy propagation*. A continuación el bloque de *Redundancy Elimination* presenta las optimizaciones de *unreachable-code elimination* y *useless-code elimination*. Por último, el bloque de *Procedure Call Transformations* presenta las optimizaciones *procedure inlining*, *procedure cloning* y *tail recursion elimination*. En este capítulo, por motivos de espacio, aparecen *loop-invariant code motion*, *loop unswitching*, *loop tiling* y *tail recursion elimination*; que son optimizaciones de bucles, ya que por su naturaleza repetitiva los bucles representan buena parte del tiempo de ejecución de los programas. El resto de las optimizaciones estudiadas se muestran en el apéndice C.

Para cada optimización, en primer lugar aparece su nombre (en inglés, como en el artículo original, para evitar posibles confusiones con la traducción). A continuación, se incluyen los *flags* del compilador que se han utilizado para compilar tanto la versión no optimizada, que sirve de referencia en las mediciones, como la versión optimizada que aplica la optimización estudiada. Después, se incluye una breve explicación de cómo funciona la optimización, ilustrada con un código de ejemplo sobre el que puede aplicarse. Dicho código de ejemplo está escrito en un lenguaje de pseudocódigo de alto nivel similar a Fortran 90, ya que es un lenguaje muy utilizado en las aplicaciones de cálculo científico (el código real medido en los experimentos se incluye en el apéndice E). En último lugar, se incluyen las gráficas de las mediciones realizadas junto con una explicación de los resultados obtenidos.

4.2 Loop invariant code motion

Flags aplicados:

- Versión no optimizada: -O0
- Versión optimizada: -O1

El *flag* que activa la optimización es -ftree-loop-im, activo en -O0. Sin embargo, al aplicar -O1 se aplica realmente la optimización.

4.2.1 Descripción

Está considerada como un caso particular de *code hoisting* (optimización que mueve algunas operaciones a un punto anterior del programa, reordena el código para mayor eficiencia). *Loop invariant code motion* mueve fuera de un bucle las operaciones invariantes. Puede aplicarse en alto nivel a las expresiones del código fuente que permanezcan constantes. También en bajo nivel para, el cálculo de direcciones que no varíen. En el código de ejemplo se muestra cómo el cálculo de la raíz cuadrada, que tiene una alta latencia de ejecución, puede moverse fuera del bucle para no repetir el cálculo en cada iteración. Generalmente, el valor precalculado se almacena en un registro, por lo que hay un compromiso entre coste de la operación precalculada y uso de registros. Si la ocupación del banco de registros es elevada y la operación precalculada tiene poca latencia de ejecución, la optimización puede llegar a afectar negativamente al rendimiento del código.

4.2.2 Código ejemplo

En este código de ejemplo se muestra un bucle que en todas las iteraciones calcula la raíz cuadrada de x. Como es un cálculo invariante, puede calcularse antes del bucle.

```
1 do i = 1, n
2    a[i] = i + sqrt(x)
3 end do
```

Al aplicar *loop invariant code motion* el cálculo de la raíz cuadrada de la línea 2 se mueve fuera del bucle:

```
1 if (n > 0) C = sqrt(x)
2 do i = 1, n
3    a[i] = i + C
4 end do
```

4.2.3 Resultados

Loop invariant code motion actúa disminuyendo agresivamente el número de instrucciones ejecutadas N_{inst}, por lo que según la ecuación 3.1 el tiempo de ejecución T_{ex} disminuirá. En cuanto al CPI, presumiblemente también disminuirá ya que se eliminan instrucciones costosas en ciclos

de CPU e invocaciones a funciones, pero no podemos determinar cómo afectará a C_L. Por tanto, se puede esperar una disminución del consumo de energía (ecuación 3.8) pero no podemos determinar qué ocurrirá con la disipación de potencia (ecuacion 3.5).

Al analizar el código ensamblador de la versión no optimizada, vemos como calcula la raíz cuadrada en cada iteración del bucle (realiza siempre la llamada a *fsqrt*), tal y como puede observarse en el siguiente extracto de código ensamblador del bucle del programa; que corresponde a las líneas 26-37 de la versión no optimizada del apéndice E:

```
.L6:
        fildl   4800044(%esp)   # i
        fstpl   16(%esp)        # %sfp
        fldl    4800032(%esp)   # x
        fld     %st(0)  #
        fsqrt
        fucomi  %st(0), %st     #,
        jp      .L8     #,
        fucomi  %st(0), %st     #,
        je      .L9     #,
        fstp    %st(0)  #
        jmp     .L5     #
```

Sin embargo, en la versión optimizada, calcula la raíz cuadrada antes de que comience el bucle (*fsqrt* y *jmp* a .L2). El extracto corresponde a las líneas 7-27 de la versión optimizada del apéndice E:

```
main:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        andl    $-16, %esp      #,
        pushl   %esi    #
        pushl   %ebx    #
        subl    $4800056, %esp  #,
        movl    $0, 12(%esp)    #,
        movl    $10, 8(%esp)    #,
        movl    $0, 4(%esp)     #,
        movl    12(%ebp), %eax  # argv, argv
        movl    4(%eax), %eax   #, tmp81
        movl    %eax, (%esp)    # tmp81,
        call    __strtol_internal       #
        movl    %eax, 44(%esp)  # D.4421,
        fildl   44(%esp)        #
        fstl    24(%esp)        # %sfp
        movl    $0, %esi        #, rep
        fsqrt
        fstpl   32(%esp)        # %sfp
        jmp     .L2     #
```

Una vez realizadas las mediciones puede verse en la figura 4.1 cómo la optimización disminuye drásticamente el tiempo de ejecución y la potencia disipada P_{tot}. Ese descenso drástico en el tiempo de ejecución T_{ex} se debe a que la raíz cuadrada es una operación con una latencia de ejecución muy alta. En la versión no optimizada se está calculando esa operación en cada iteración del bucle, mientras que en la versión optimizada se calcula una sola vez al comienzo del

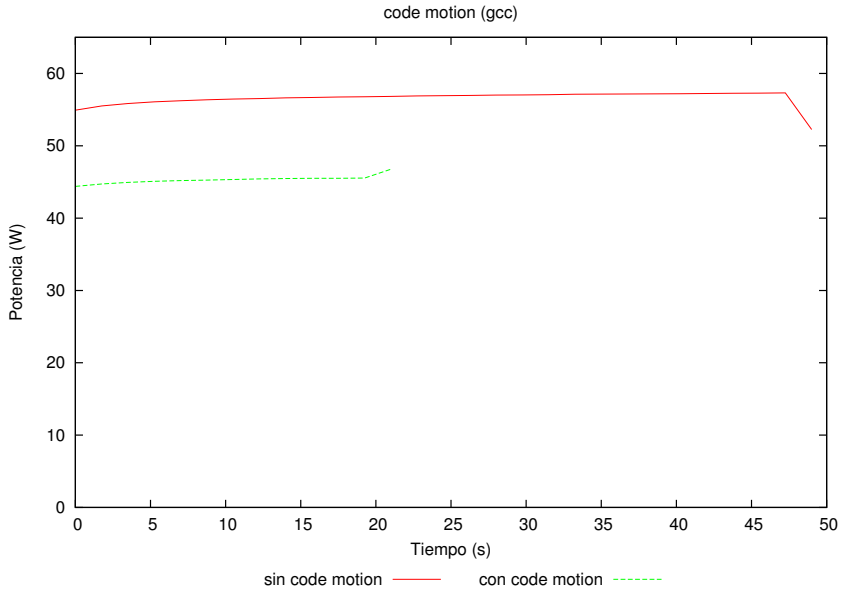

Figura 4.1: Potencia usando *loop invariant code motion*.

bucle. Además, el procesador Pentium 4 sólo tiene una unidad funcional para el cálculo de la raíz cuadrada [12], por lo que se produce una dependencia estructural entre iteraciones en la versión no optimizada. En la versión optimizada, en cambio, dicha dependencia no se produce y eso permite mayor paralelismo a nivel de instrucción (ILP), lo que hace que el tiempo de ejecución se reduzca.

Viendo la gráfica, y de acuerdo a la ecuación 3.7, podemos anticipar que el consumo energético también disminuirá drásticamente ya que tanto la potencia disipada como el tiempo de ejecución han disminuido. En la tabla 4.1 se muestra el consumo energético total y normalizado, y puede apreciarse que el consumo energético se reduce en un 64 %.

Versión	Energía (Julios)	Energía normalizada
No optimiz.	2871.40	1
Optimizada	1031.08	0.36

Tabla 4.1: Energía usando *loop invariant code motion*.

Analizando el comportamiento térmico (figura 4.2) podemos observar que la temperatura va aumentando conforme avanza la ejecución del programa. Podemos observar que la temperatura alcanzada en el centro del procesador (T1) en la versión no optimizada es mayor que en la optimizada. Este comportamiento se debe a que la versión del programa que no aplica *loop invariant code motion* tiene que disipar más potencia durante la ejecución (figura 4.1). Esa mayor potencia disipada causa un incremento de la temperatura en el procesador.

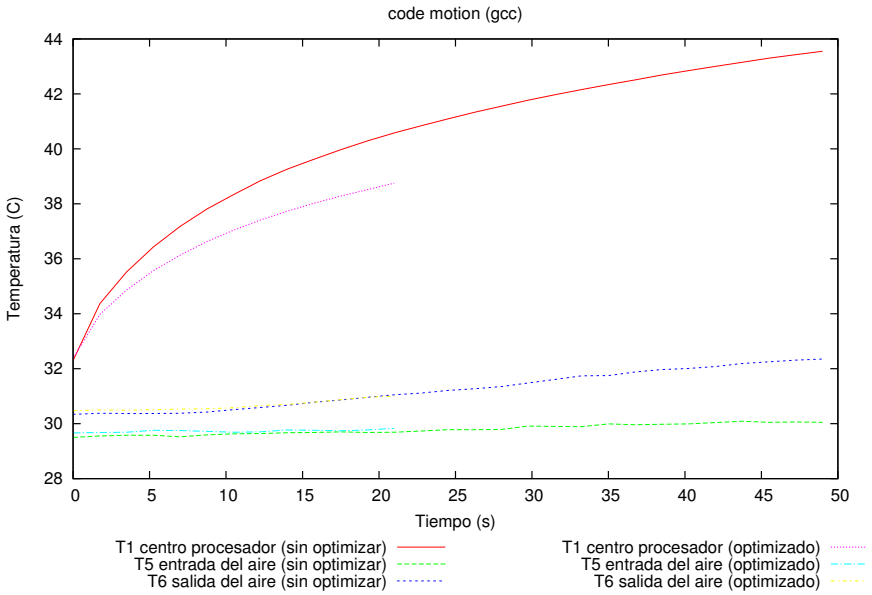

Figura 4.2: Temperaturas usando *loop invariant code motion*.

4.3 Loop unswitching

Flags aplicados:

- Versión no optimizada: -O1
- Versión optimizada: -O1 -funswitch-loops

4.3.1 Descripción

Esta optimización puede aplicarse cuando un bucle contiene en su interior un condicional cuya condición es invariante. El bucle, entonces, puede replicarse dentro de cada rama del condicional. De esta manera, la evaluación de la condición sólo se realiza la primera vez, en lugar de hacerse en cada una de las n iteraciones. Se reduce, por tanto, la sobrecarga asociada a la evaluación de la condición y también el tamaño del código del bucle (aunque el tamaño de código total aumenta, ya que hemos replicado en bucle en cada rama del condicional). Además, esta optimización puede permitir la paralelización del código.

4.3.2 Código ejemplo

En este apartado vemos un código de ejemplo sobre el que puede aplicarse *loop unswitching* para replicar el bucle en las dos ramas del condicional.

```
1  do i = 2, n
2     a[i] = a[i] + x
3     if (x < 7) then
4        b[i] = a[i] + c[i]
5     else
6        b[i] = a[i-1] + b[i-1]
7     end if
8  end do
```

La instrucción de la línea 2 se ejecuta a lo largo de todas las iteraciones, x no cambia de valor en el espacio de iteraciones por lo que la condición de la línea 3 es invariante. Por lo tanto, esa condición puede evaluarse al principio y replicar el bucle en cada rama del condicional junto con la instrucción de la línea 2. El código queda de la siguiente manera:

```
1  if (n > 1) then
2     if (x < 7) then
3        do all i = 2, n
4           a[i] = a[i] + x
5           b[i] = a[i] + c[i]
6        end do all
7     else
8        do i = 2, n
9           a[i] = a[i] + x
10          b[i] = a[i-1] + b[i-1]
11       end do
12    end if
13 end if
```

Se evalúa si $x < 7$ y, dependiendo del resultado, se ejecuta el bucle de la rama *if* o el de la rama *else*.

4.3.3 Resultados

Loop unswitching actúa reduciendo el número de instrucciones N_{inst}, ya que elimina la sobrecarga de evaluar en cada iteración la condición del condicional en el interior del bucle. Esto afecta al consumo energético, reduciéndolo según la ecuación 3.8. En cuanto a la potencia, no sabemos *a priori* cómo *loop unswitching* afectará al CPI y a C_L por lo que no podemos asegurar qué ocurrirá con la ecuación 3.5.

Si analizamos el código ensamblador generado por el programa de prueba (apéndice E) vemos que en la versión no optimizada, se realiza la comparación en todas las iteraciones del bucle (*cmpl $6*) tal y como puede observarse en este extracto de las líneas 84-100:

```
.L7:
        movl    %esi, %edx      # D.3291, D.3255
        addl    480032(%esp,%eax,4), %edx       # a, D.3255
        movl    %edx, 480032(%esp,%eax,4)       # D.3255, a
        cmpl    $6, %esi        #, D.3291
        jg      .L5     #,
        addl    32(%esp,%eax,4), %edx   # c, tmp135
        movl    %edx, 240032(%esp,%eax,4)       # tmp135, b
        jmp     .L6     #
.L5:
        movl    240028(%esp,%eax,4), %edx       # b, tmp141
        addl    480028(%esp,%eax,4), %edx       # a, tmp140
        movl    %edx, 240032(%esp,%eax,4)       # tmp140, b
.L6:
        addl    $1, %eax        #, i
        cmpl    $60000, %eax    #, i
        jne     .L7     #,
```

Si $x < 7$ ejecuta 10 instrucciones en cada iteración; si no, ejecuta 11 instrucciones.

En cambio, en la versión optimizada la comparación se realiza fuera del bucle y después se ejecuta el bucle en una rama u otra. La optimización puede observarse en este extracto de las líneas 84-104 de la versión optimizada:

```
        cmpl    $6, %edi        #, D.3291
        jle     .L10    #,
.L6:
        addl    %edi, 480032(%esp,%eax,4)       # D.3291, a
        movl    240028(%esp,%eax,4), %edx       # b, tmp143
        addl    480028(%esp,%eax,4), %edx       # a, tmp142
        movl    %edx, 240032(%esp,%eax,4)       # tmp142, b
        addl    $1, %eax        #, i
        cmpl    $60000, %eax    #, i
        jne     .L6     #,
        jmp     .L7     #
.L10:
        movl    %edi, %edx      # D.3291, D.3255
        addl    480032(%esp,%eax,4), %edx       # a, D.3255
        movl    %edx, 480032(%esp,%eax,4)       # D.3255, a
        addl    32(%esp,%eax,4), %edx   # c, tmp148
        movl    %edx, 240032(%esp,%eax,4)       # tmp148, b
        addl    $1, %eax        #, i
```

```
19          cmpl    $60000, %eax    #, i
20          jne     .L10    #,
21  .L7:
```

Si $x < 7$ ejecuta 8 instrucciones en cada iteración; si no, ejecuta 7 instrucciones.

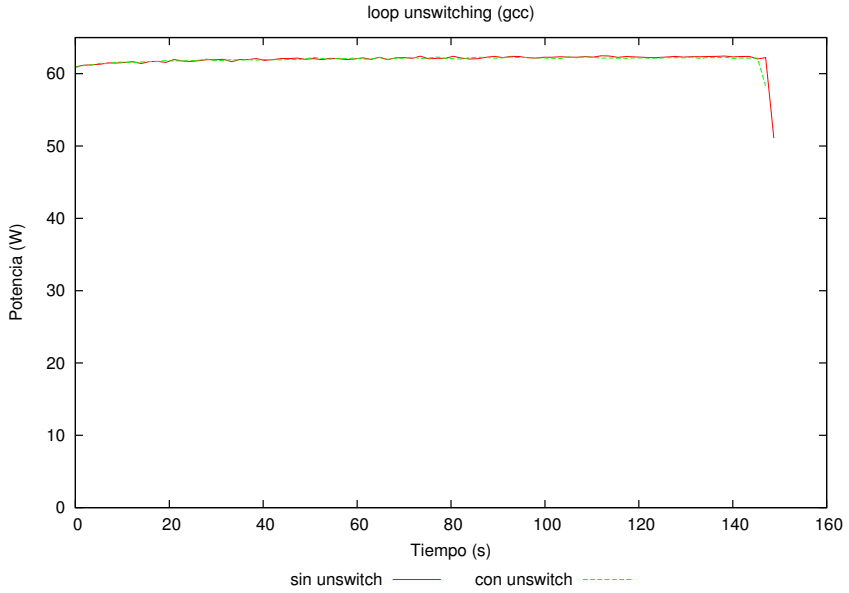

Figura 4.3: Potencia usando *loop unswitching*.

En la figura 4.3 se puede observar que la potencia disipada, tanto en la versión optimizada como en la que no lo está, sigue el mismo comportamiento. Sin embargo, debido a la reducción del número de instrucciones ejecutadas (menos evaluaciones de la condición), el tiempo de ejecución disminuye un poco en la versión optimizada. No hay apenas variación entre una versión y otra porque el código utilizado es intensivo en cálculo y hay dependencias entre las instrucciones, pero aún así la reducción de la sobrecarga del condicional ayuda a reducir un poco el tiempo de ejecución. Utilizando la ecuación 3.7, podemos calcular el consumo de energía de ambas versiones (tabla 4.2).

Versión	Energía (Julios)	Energía normalizada
No optimiz.	9321.94	1
Optimizada	9135.51	0.98

Tabla 4.2: Energía usando *loop unswitching*.

Analizando la gráfica de las temperaturas (figura 4.4) podemos ver que sigue el mismo comportamiento y alcanza los mismos valores en ambas versiones ya que la disipación de potencia es igual en ambas versiones.

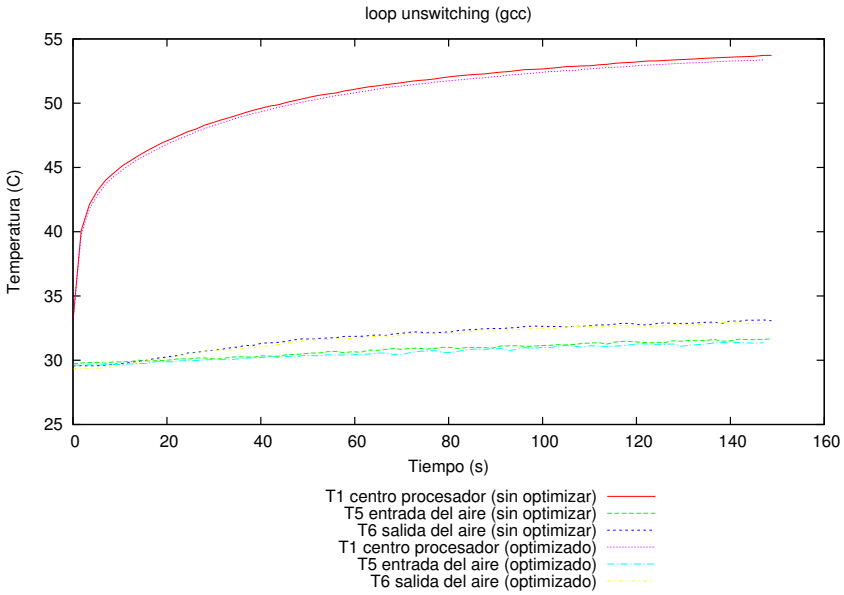

Figura 4.4: Temperaturas usando *loop unswitching*.

4.4 Loop tiling

Flags aplicados:

- Versión no optimizada: -O1
- Versión optimizada: -O1 -floop-block

4.4.1 Descripción

También conocida como *blocking*. El objetivo de esta optimización es mejorar la tasa de aciertos en memoria *cache*. Para ello, actúa dividiendo el espacio de iteración en bloques y transformando el bucle para iterar sobre ellos. De esa manera, se trabaja el máximo posible con bloques de datos que ya están en *cache*. Esa reutilización de los datos de *cache* hace que mejore la tasa de aciertos en memoria, y con ello el tiempo de ejecución del programa.

4.4.2 Código ejemplo

A continuación un código de ejemplo sobre el que puede aplicarse *loop tiling*. Nótese que los índices de acceso a las matrices están cambiados, por lo que una accederá a elementos contiguos en cada iteración y la otra tendrá que acceder con desplazamiento n, ocasionando un fallo en *cache* en cada acceso.

```
do i = 1, n
   do j = 1, n
      a[i, j] = b[j, i]
   end do
end do
```

Suponiendo un almacenamiento de las matrices en memoria por columnas, el acceso a los elementos de b se hace con desplazamiento 1, mientras que a los elementos de a se accede con desplazamiento n. Al iterar sobre subbloques de las matrices, el bucle accede a elementos que ya están en memoria *cache*, logrando un mejor aprovechamiento de la misma. El código, transformado para iterar sobre bloques de tamaño $64 * 64$, queda de la siguiente manera:

```
do TI = 1, n, 64
   do TJ = 1, n, 64
      do i = TI, min(TI+63, n)
         do j = TJ, min(TJ+63, n)
            a[i, j] = b[j, i]
         end do
      end do
   end do
end do
```

4.4.3 Resultados

A pesar de que el número de instrucciones N_{inst} aumenta debido a la sobrecarga de los bucles, el tiempo de ejecución T_{ex} disminuye. Esto se debe a que las instrucciones que acceden a memoria lo hacen con una tasa de aciertos en *cache* superior, es decir, su CPI es menor. En la versión optimizada se maximiza la tasa de aciertos en la *cache* de primer nivel L1 cuya latencia son 2

ciclos. En cambio, en la versión no optimizada se producen muchos fallos en L1 y hay que acceder a la *cache* de segundo nivel L2, que tiene una latencia de acceso de 7 ciclos [12]. Esa diferencia provoca que el tiempo de ejecución de la versión optimizada se reduzca considerablemente. En cuanto a la potencia, tal y como se aprecia en la figura 4.5 la potencia media disipada es idéntica en ambas versiones. Al ser el tiempo de ejecución mucho menor en la versión optimizada, el consumo energético es también menor según la ecuación 3.7.

En el código ensamblador del apéndice E puede verse que en la versión no optimizada, la función *foo()* itera a lo largo de toda la matriz (extracto de las líneas 5-37):

```
1  foo:
2          pushl   %ebp
3          movl    %esp, %ebp
4          pushl   %edi
5          pushl   %esi
6          pushl   %ebx
7          movl    $0, %esi
8          movl    $0, %edi
9          jmp     .L2
10 .L3:
11         movl    (%ecx), %ebx
12         addl    $1, %ebx
13         movl    %ebx, (%edx)
14         addl    $1, %eax
15         addl    $16000, %ecx
16         addl    $4, %edx
17         cmpl    $4000, %eax
18         jne     .L3
19         addl    $1, %esi
20         cmpl    $4000, %esi
21         je      .L6
22 .L2:
23         leal    b(,%esi,4), %ecx
24         imull   $16000, %esi, %edx
25         addl    $a, %edx
26         movl    %edi, %eax
27         jmp     .L3
28 .L6:
29         popl    %ebx
30         popl    %esi
31         popl    %edi
32         popl    %ebp
33         ret
```

En cambio, en la versión optimizada la función *foo()* itera en bloques de 50 datos (extracto de las líneas 5-75):

```
1  foo:
2          pushl   %ebp
3          movl    %esp, %ebp
4          pushl   %edi
5          pushl   %esi
6          pushl   %ebx
7          subl    $32, %esp
```

```
8           movl    $50, %edx
9           movl    $3999, -40(%ebp)
10  .L9:
11          movl    $0, %eax
12          leal    -50(%edx), %ecx
13          movl    %ecx, -32(%ebp)
14          movl    %edx, -36(%ebp)
15  .L8:
16          movl    -32(%ebp), %ecx
17          imull   $4000, %eax, %ebx
18          movl    %ebx, -24(%ebp)
19          leal    50(%eax), %ebx
20          movl    %ebx, -20(%ebp)
21          movl    %ebx, -28(%ebp)
22  .L7:
23          movl    -36(%ebp), %ebx
24          cmpl    $3999, %edx
25          jbe     .L3
26          movl    $3999, %ebx
27  .L3:
28          cmpl    %ecx, %ebx
29          jb      .L2
30          movl    -24(%ebp), %ebx
31          addl    %ecx, %ebx
32          leal    b(,%ebx,4), %edi
33          imull   $4000, %ecx, %ebx
34          addl    %eax, %ebx
35          leal    a(,%ebx,4), %esi
36          movl    %eax, %ebx
37          movl    %eax, -44(%ebp)
38  .L6:
39          movl    -28(%ebp), %eax
40          movl    %eax, -16(%ebp)
41          cmpl    $3999, -20(%ebp)
42          jbe     .L5
43          movl    -40(%ebp), %eax
44          movl    %eax, -16(%ebp)
45  .L5:
46          cmpl    -16(%ebp), %ebx
47          ja      .L4
48          movl    (%edi), %eax
49          addl    $1, %eax
50          movl    %eax, (%esi)
51          addl    $1, %ebx
52          addl    $16000, %edi
53          addl    $4, %esi
54          jmp     .L6
55  .L4:
56          movl    -44(%ebp), %eax
57          addl    $1, %ecx
58          jmp     .L7
59  .L2:
60          addl    $51, %eax
61          cmpl    $4029, %eax
62          jne     .L8
63          addl    $51, %edx
```

```
64          cmpl    $4079, %edx
65          jne     .L9
66          addl    $32, %esp
67          popl    %ebx
68          popl    %esi
69          popl    %edi
70          popl    %ebp
71          ret
```

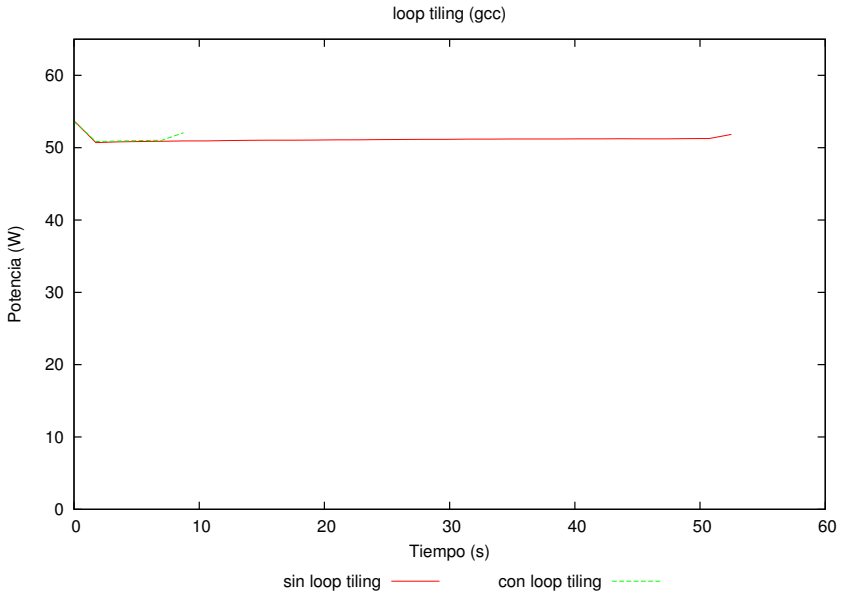

Figura 4.5: Potencia usando *loop tiling*.

Tal y como se aprecia en la tabla 4.3 el consumo energético desciende en la versión optimizada en un 81 %.

Versión	Energía (Julios)	Energía normalizada
No optimiz.	2777.60	1
Optimizada	541.59	0.19

Tabla 4.3: Energía usando *loop tiling*.

En cuanto al comportamiento térmico (figura 4.6), la temperatura sigue el mismo comportamiento en las dos versiones ya que la potencia disipada también es la misma en ambas versiones.

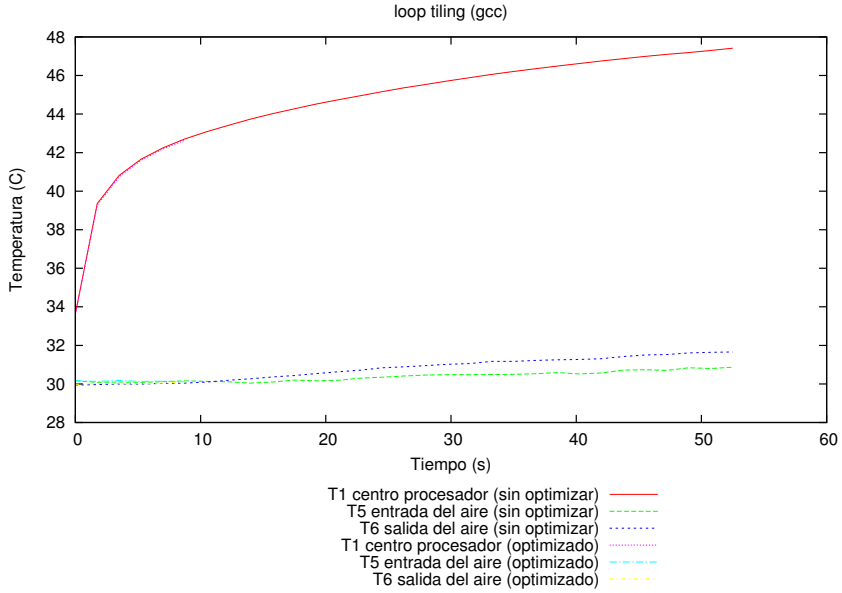

Figura 4.6: Temperaturas usando *loop tiling*.

4.5 Tail recursion elimination

Flags aplicados:

- Versión no optimizada: -O1
- Versión optimizada: -O1 -foptimize-sibling-calls

4.5.1 Descripción

Esta optimización transforma las funciones con recursión final en iterativas. Una función es recursiva si se invoca a sí misma, y tiene recursividad final si la última acción de esa función es invocarse a sí misma y devolver el valor de la llamada recursiva. La razón por la que se transforma una función recursiva en iterativa es para no pagar el coste adicional en tiempo del mecanismo de llamada a procedimientos (cambio de contexto) y del paso de parámetros. También para no pagar el coste adicional de memoria que lleva implícita la implementación de la recursividad. Esto se debe a que la implementación más frecuente utiliza una pila, cada uno de cuyos elementos reserva espacio para una activación de la función recursiva (es decir, para las direcciones de los parámetros, y para las variables locales).

4.5.2 Código ejemplo

A continuación un código de ejemplo donde la función recursiva se transforma en una iterativa.

```
1  recursive logical function inarray(a, x, i, n)
2     real x, a[n]
3     integer i, n
4
5     if (i > n) then
6        inarray = FALSE
7     else if (a[i] = x) then
8        inarray = TRUE
9     else
10       inarray = inarray(a, x, i+1, n)
11    end if
12    return
```

Esta función devuelve *TRUE* si el elemento x está en el vector a. De lo contrario devuelve *FALSE*. Al transformarla, se elimina la llamada recursiva de la línea 10.

```
1  logical function inarray(a, x, i, n)
2     real x, a[n]
3     integer i, n
4
5  etiq:   if (i < n) then
6             inarray = FALSE
7          else if (a[i] = x) then
8             inarray = TRUE
9          else
10            i = i+1
11            goto etiq
12         end if
13         return
```

4.5.3 Resultados

Tail recursion elimination actúa eliminando el número de instrucciones ejecutadas N_inst y reduciendo el tiempo de ejecución T_{ex} según la ecuación 3.1. En cuanto al CPI, lo más probable es que disminuya dado que se eliminan instrucciones costosas en ciclos de procesador como las invocaciones a procedimientos y las relacionadas con cambios de contexto. Por lo tanto, cabe esperar una disminución del consumo energético E_{tot} (ver ecuación 3.8). Si el CPI disminuye mucho, podría afectar a C_L haciendo que éste aumentara, lo que podría provocar que la potencia disipada P_{tot} aumentara (ver ecuación 3.5).

En el código ensamblador del apéndice E puede observarse que en la versión no optimizada, la función *inarray()* tiene una invocación recursiva (extracto de código de las líneas 4-28):

```
1  inarray:
2          pushl   %ebp    #
3          movl    %esp, %ebp      #,
4          subl    $24, %esp       #,
5          movl    8(%ebp), %edx   # vector, vector
6          movl    12(%ebp), %ecx  # elemento, elemento
7          movl    16(%ebp), %eax  # indice, indice
8          cmpl    $230000, %eax   #, indice
9          jg      .L3     #,
10         cmpl    %ecx, (%edx,%eax,4)     # elemento,* vector
11         je      .L4     #,
12         addl    $1, %eax        #, tmp68
13         movl    %eax, 8(%esp)   # tmp68,
14         movl    %ecx, 4(%esp)   # elemento,
15         movl    %edx, (%esp)    # vector,
16         call    inarray #
17         jmp     .L1     #
18 .L3:
19         movl    $0, %eax        #, D.3254
20         jmp     .L1     #
21 .L4:
22         movl    $1, %eax        #, D.3254
23 .L1:
24         leave
25         ret
```

En la versión optimizada, la invocación recursiva se elimina (extracto de las líneas 4-33):

```
1  inarray:
2          pushl   %ebp    #
3          movl    %esp, %ebp      #,
4          pushl   %ebx    #
5          movl    8(%ebp), %ebx   # vector, vector
6          movl    12(%ebp), %ecx  # elemento, elemento
7          movl    16(%ebp), %edx  # indice, indice
8          movl    $0, %eax        #, D.3254
9          cmpl    $230000, %edx   #, indice
10         jg      .L2     #,
11         movb    $1, %al #,
12         cmpl    %ecx, (%ebx,%edx,4)     # elemento,* vector
13         jne     .L9     #,
```

```
14          jmp     .L2     #
15  .L5:
16          cmpl    %ecx, (%ebx,%edx,4)     # elemento,* vector
17          .p2align 4,,3
18          je      .L8     #,
19  .L9:
20          addl    $1, %edx        #, indice
21          cmpl    $230000, %edx   #, indice
22          jle     .L5     #,
23          movl    $0, %eax        #, D.3254
24          jmp     .L2     #
25  .L8:
26          movl    $1, %eax        #, D.3254
27  .L2:
28          popl    %ebx    #
29          popl    %ebp    #
30          ret
```

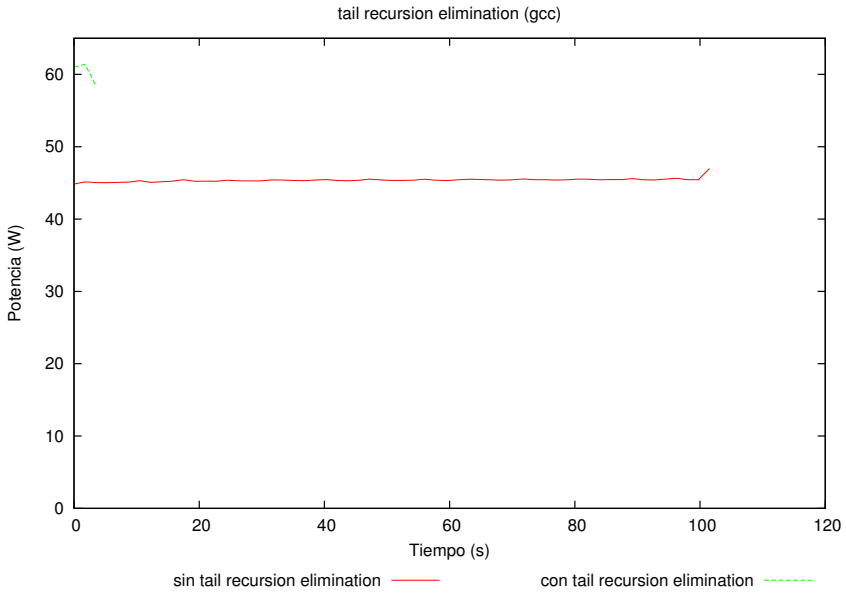

Figura 4.7: Potencia usando *tail recursion elimination*.

Una vez realizadas las mediciones y tal y como se aprecia en la figura 4.7, el tiempo de ejecución T_{ex} desciende drásticamente en la versión optimizada. Este descenso se debe a la reducción de todas las operaciones de pila tipo *push* y *pop* que lleva asociadas la invocación recursiva. Al eliminar esas instrucciones de acceso a memoria, el tiempo de ejecución disminuye espectacularmente. Por contra, la potencia media disipada también es bastante mayor. Al eliminar instrucciones con un alto CPI, hace que la actividad del procesador aumente y C_L aumente también. Con ayuda de la ecuación 3.7 podemos calcular cómo la energía consumida desciende en un 93 % debido al importante descenso en el tiempo de ejecución (ver tabla 4.4).

En cuanto al comportamiento de las temperaturas, podemos observar en la figura 4.8 cómo la curva de la temperatura disipada en el centro de la CPU por la versión optimizada crece más deprisa que la de la versión no optimizada. Esto se debe a que la disipación de potencia de la versión optimizada es también considerablemente mayor (figura 4.7).

Versión	Energía (Julios)	Energía normalizada
No optimiz.	4685.94	1
Optimizada	316.44	0.07

Tabla 4.4: Energía usando *tail recursion elimination*.

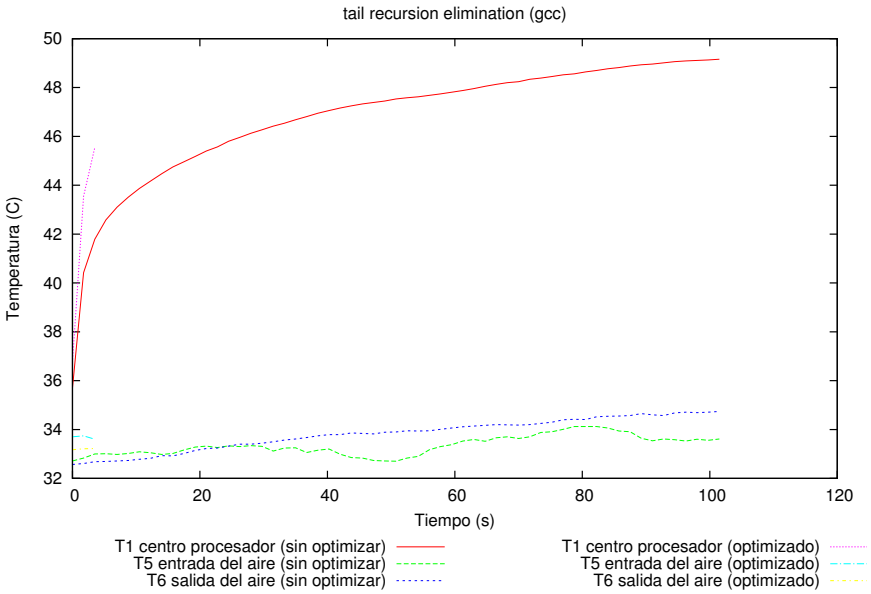

Figura 4.8: Temperaturas usando *tail recursion elimination*.

Capítulo 5
Conclusiones

En este capítulo se presentan las conclusiones obtenidas al estudiar las optimizaciones de compilación de este proyecto fin de carrera.

Una conclusión importante que podemos extraer de este proyecto es que una reducción en el tiempo de ejecución tiene como consecuencia una disminución del consumo energético. Incluso aunque la potencia media disipada aumente, proporcionalmente disminuye más el tiempo de ejecución; haciendo que el consumo energético sea menor en la versión optimizada (ecuación 3.8). Además, dicha reducción suele ser proporcional a la reducción en el tiempo de ejecución. Esto indica que, en la arquitectura Pentium 4, el consumo energético es prácticamente independiente del tipo de instrucciones ejecutadas. Exceptuando las operaciones de memoria, que debido a los posibles fallos en *cache* no se sabe cuánto pueden tardar *a priori*; el coste energético del resto de instrucciones es muy similar. Esto se debe al diseño del Intel Pentium 4: al contar con un *pipeline* segmentado en 20 etapas [12], la mayoría de ellas comunes a todas las instrucciones, cada instrucción pasa por todas las fases y sólo difieren en la etapa de ejecución.

Otra conclusión importante es que el acceso a memoria es costoso en términos de energía. Los accesos a memoria, las operaciones en pila, los cambios de contexto; todas esas operaciones causan un incremento importante del consumo energético. El ejemplo más claro es el de *tail recursion elimination*, donde con la transformación de un algoritmo recursivo en uno iterativo se logra una reducción del 93 % en el consumo energético. Otro buen ejemplo es el de *loop tiling*, que al cambiar la forma de iterar un bucle, hace que itere sobre bloques de datos que caben en *cache* y entonces maximiza su tasa de aciertos logrando una reducción del 81 % en el consumo energético.

Por último, la potencia disipada: como se puede observar en las gráficas del capítulo 4, a una mayor disipación de potencia corresponde una mayor temperatura alcanzada por el procesador. Si se busca, por el motivo que sea, minimizar la temperatura alcanzada; optimizaciones como *tail recursion elimination* o *scalar replacement* no son adecuadas. También puede interesar minimizar la potencia disipada por otros motivos distintos al de la temperatura alcanzada. Por ejemplo, en dispositivos móviles, la potencia entregada por la batería es limitante. La batería tiene una cota de potencia máxima que puede entretar, con lo cual, no podremos ejecutar código que requiere más potencia que esa cota máxima. Incluso teniendo en cuenta que la ejecución de ese código optimizado consumiría menos energía que si no estuviera optimizado y la batería tendría más vida útil; al no poder entregar tanta potencia, no puede ejecutarse.

En cuanto a los posibles *flags* de compilación -Oenergy, -Opower y -Otemp: teniendo en cuenta los programas que se han medido, para -Oenergy servirían casi todas las optimizaciones

estudiadas, puesto que con las optimizaciones se intenta reducir el tiempo de ejecución y una reducción en el tiempo de ejecución causa una reducción en el consumo energético. Cuanto mayor sea la reducción en tiempo de ejecución, mejor. Por ello sería conveniente utilizar *tail recursion elimination* en algorimos recursivos y *loop tiling* en las matrices de los programas de cálculo científico. En cuanto a -Opower y -Otemp, las mismas optimizaciones que sirven para uno lo hacen para el otro, dado que una disminución de la potencia disipada causa una disminución de la temperatura. Por ello, optimizaciones como *scalar replacement* o *tail recursion elimination* no serían apropiadas para estos *flags*.

Bibliografía

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, Boston, MA, USA, second edition, 2007.

[2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[3] Alicia Asín. Evaluación del consumo en procesadores de altas prestaciones. 2006.

[4] Bacon, Graham, and Sharp. Compiler transformations for high-performance computing. *CSURV: Computing Surveys*, 26, 1994.

[5] Octavio Benedí. Determinación del consumo en procesadores de altas prestaciones y caracterización energética de programas compilados. 2008.

[6] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July/August 1999.

[7] Intel Corporation. *Intel C++ Compiler User's Guide*, 2002.

[8] Free Software Foundation. *Using the GNU Compiler Collection (GCC)*. `http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/`.

[9] gnuplot project team. *gnuplot documentation*. `http://www.gnuplot.info/documentation.html`.

[10] Sergio Gutierrez. Aspectos térmicos de la ejecución de programas: estudio experimental sobre un pentium 4. 2009.

[11] Sergio Gutiérrez and Octavio Benedí. Processor energy and temperature in computer architecture courses: a hands-on approach. gaZ. Dpto. de Informática e Ingeniería de Sistemas.

[12] Glenn Hinton, Dave Sager, Mike Upton, Darrel Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[13] ADLINK Technology Inc. *PCIS-DASK Data Acquisition Software Development Kit for NuDAQ PCI Bus Cards: User's Manual*. `http://www.adlinktech.com/`.

[14] Nam Sung Kim, Todd M. Austin, David Blaauw, Trevor N. Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut T. Kandemir, and Narayanan Vijaykrishnan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12):68–75, 2003.

[15] Michal Mienik. *CPU Burn-in*. `http://www.cpuburnin.com/`.

[16] Trevor N. Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, 2001.

[17] Dia project team. *Dia documentation.* `http://projects.gnome.org/dia/`.

[18] LaTeX project team. *LaTeX documentation.* `http://www.latex-project.org/guides/`.

[19] John S. Seng and Dean M. Tullsen. The effect of compiler optimizations on pentium 4 power consumption. In *Interaction between Compilers and Computer Architectures*, pages 51–56. IEEE Computer Society, 2003.

[20] LLVM team. *LLVM user guides.* `http://llvm.org/docs/#userguide`.

[21] Tektronix. *TCPA300/400 Amplifiers & TCP300/400 Series AC/DC Current Probes Instruction Manual.* `http://www.tek.com/`.

[22] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective (3rd Edition).* Addison Wesley, 3 edition, May 2004.

Apéndice A
Plataforma de medida

En este apéndice se describe la plataforma del laboratorio 2.08 del gaZ, en la cual se han llevado a cabo las mediciones.

La plataforma de medida [11] está compuesta por dos ordenadores. Uno, el ordenador que ejecuta (OE), está monitorizado y su misión es ejecutar el código que queremos medir. El otro, el ordenador que mide (OM), toma muestras de la potencia y la temperatura, y las almacena para permitir su análisis posterior.

A.1 Ordenador que ejecuta

El OE tiene un sistema operativo GNU/Linux de la distribución *Ubuntu* con un kernel 2.6.25 modificado en el que se han desactivado los módulos y servicios que no son estrictamente necesarios (X-Windows, impresión, USB, ...) para minimizar la energía consumida por el sistema operativo y alterar lo mínimo posible las mediciones. El procesador es un Intel Pentium 4 2.8 GHz *Norwood* y la placa base es una ASUS P4 P8000.

A.2 Ordenador que mide

El OM también tiene un sistema operativo GNU/Linux y además cuenta con *software* para capturar datos y distintos *scripts* para el tratamiento de los datos. Recoge la información de la potencia y la temperatura con una tarjeta capturadora Adlink PCI-9112 [13] muestreada a una frecuencia de 2000 muestras/segundo por canal. En la imagen A.1 puede observarse una vista general de la plataforma con los distintos equipos. En la imagen A.2 puede observarse con más detalle el OE con la sonda amperimétrica conectada.

A.3 Medición de la potencia

Para calcular la potencia disipada hace falta conocer el valor de dos magnitudes físicas: el voltaje y la intensidad de la corriente, el producto de ambas da el valor de la potencia. La intensidad de la corriente se mide con una sonda amperimétrica conectada a un amplificador Tektronix TPC-312 [21] (ver figura A.3) y a los cables de alimentación del procesador. El voltaje se mide con un divisor de tensión (posteriormente hay que corregir el valor) a la salida del VRM (voltage

Figura A.1: De izquierda a derecha: OE, osciloscopio, amplificador y OM.

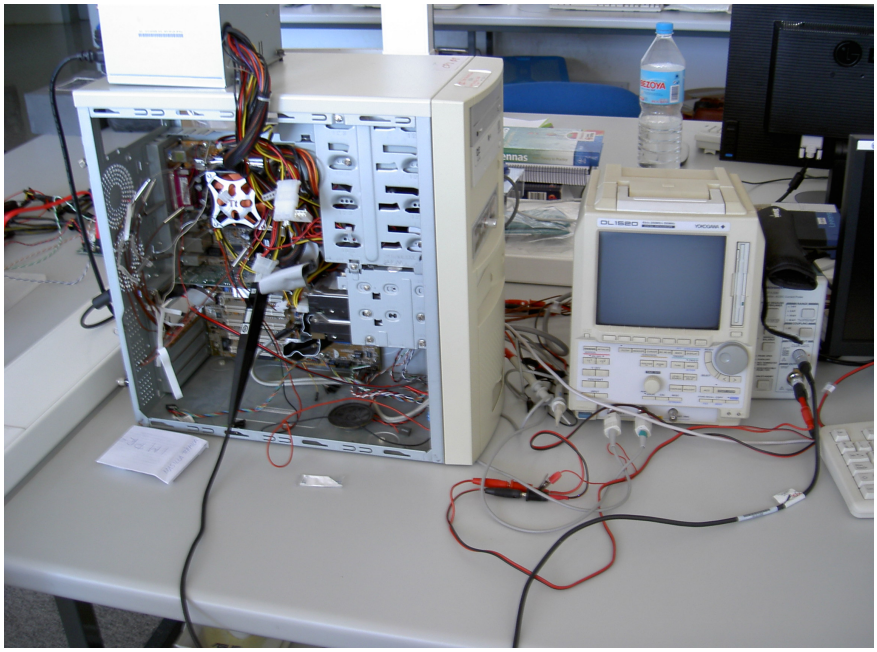

Figura A.2: Vista del amplificador Tektronix con la sonda conectada a OE.

regulator module) de la placa base. Tanto la intensidad como el voltaje se muestrean con la tarjeta capturadora del OM.

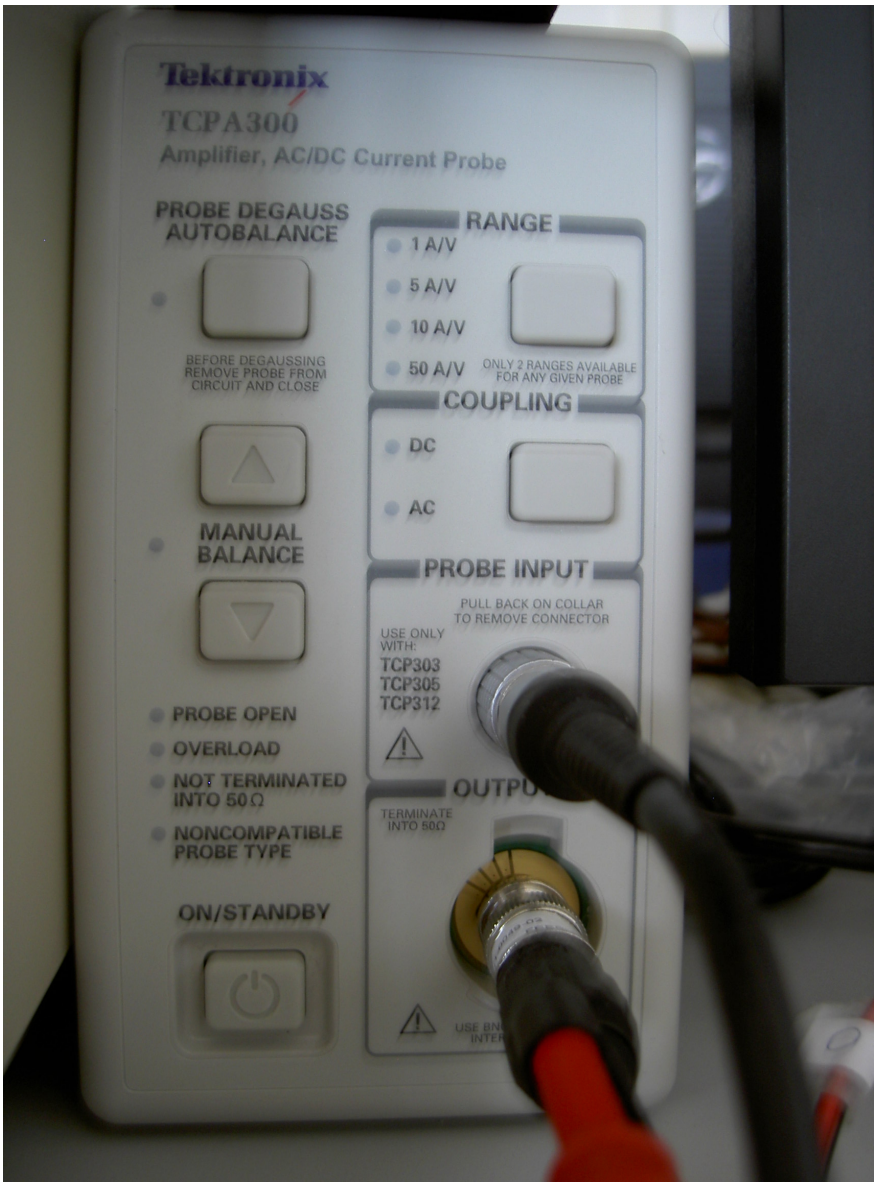

Figura A.3: Vista frontal del amplificador Tektronix.

A.4 Medición de la temperatura

La temperatura se mide con 6 termopares con un rango de temperatura que va de 0 °C a 100 °C:

- Termopar 1 (T1) está colocado en el centro del disipador, en contacto con el procesador (ver figura A.4).
- Termopar 2 (T2) en el borde del disipador (ver figura A.4).
- Termopar 3 (T3) en una aleta, en el centro del disipador (ver figura A.5).

- Termopar 4 (T4) en una aleta, en el borde del disipador (ver figura A.5).
- Termopar 5 (T5) en el ventilador, en la entrada de aire (ver figura A.6).
- Termopar 6 (T6) en la salida del aire (ver figura A.6).

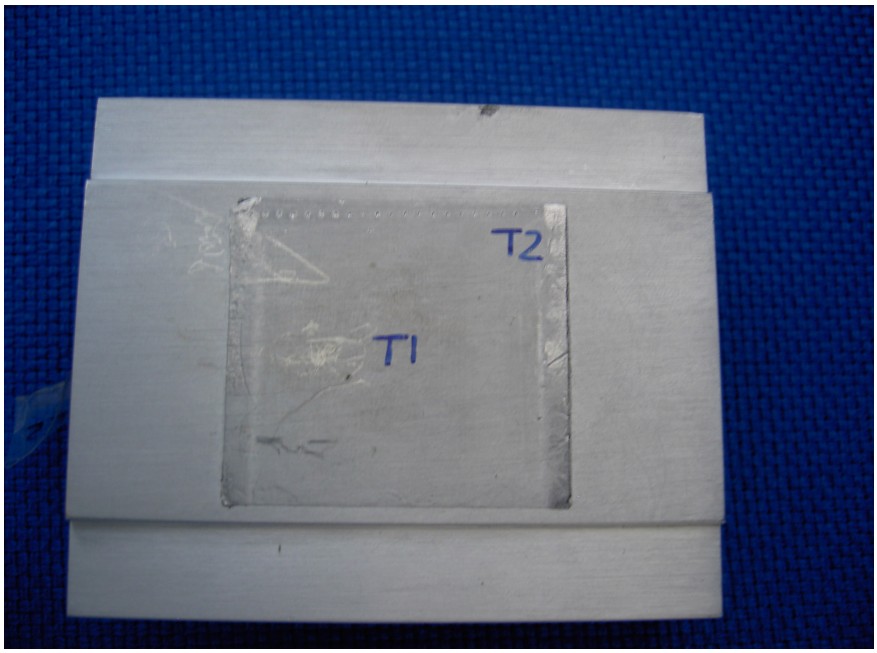

Figura A.4: Vista inferior del disipador, con las posiciones de T1 y T2.

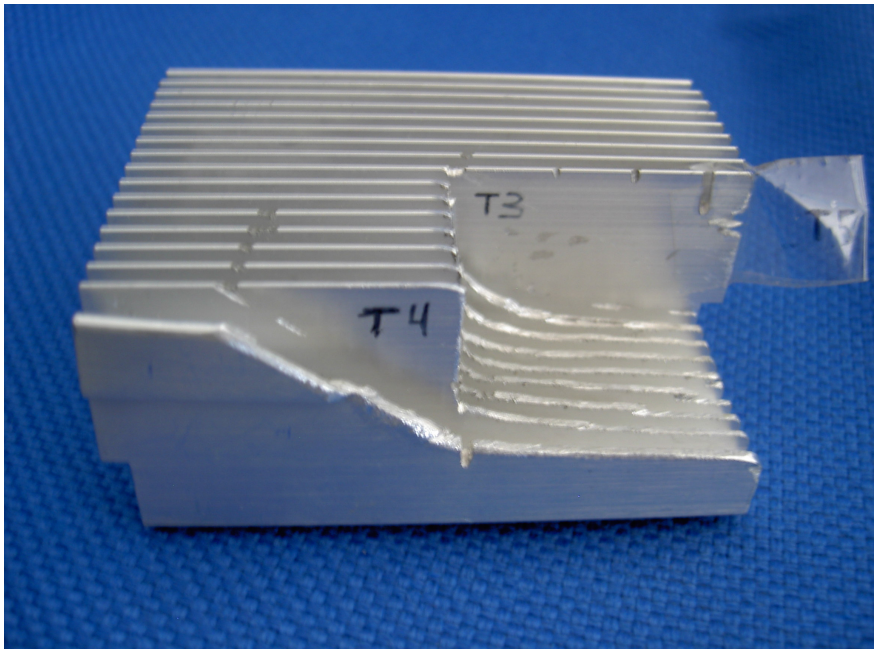

Figura A.5: Vista superior del disipador, con las posiciones de T3 y T4.

Los termopares también están conectados a la tarjeta capturadora de OM. La frecuencia de muestreo de los termopares es de 0.73 muestras/segundo. Esta frecuencia de muestreo menor

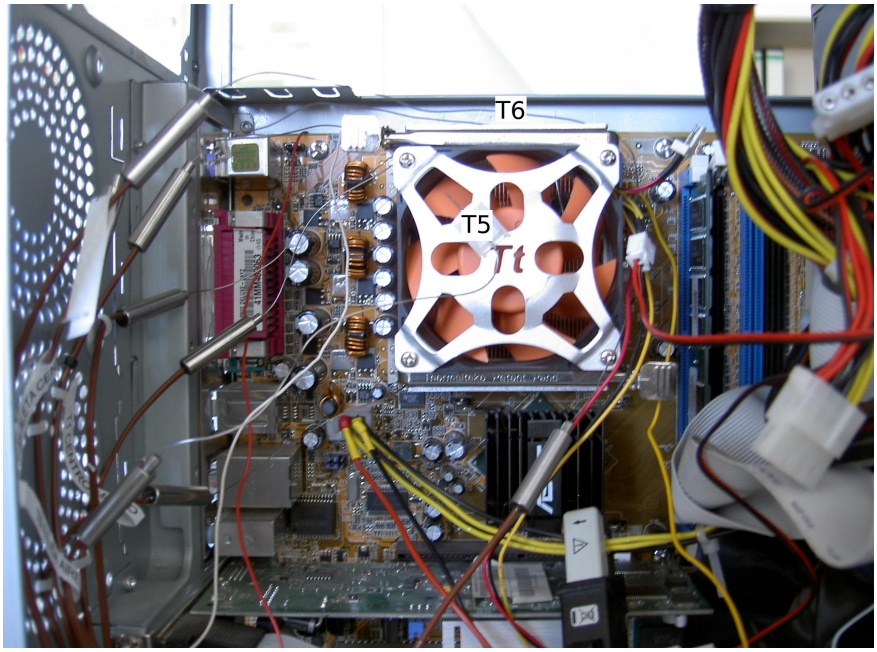

Figura A.6: Vista del ventilador con los termopares T5 y T6.

que la de la potencia está justificada porque los cambios en la temperatura se producen más despacio que los producidos en la potencia. Los termopares más importantes son T1, T5 y T6. El termopar T1 porque mide la temperatura que alcanza el procesador. Los termopares T5 y T6 porque con ellos se puede calcular el gradiente de temperatura y saber cuánto calor se está disipando:

$$q = h * A * \Delta T \tag{A.1}$$

donde h es una constante experimental, A es el área del disipador y ΔT es el gradiente de temperatura.

Apéndice B
Metodología

En este apéndice se completa y extiende lo presentado en el capítulo 2 acerca de la metodología seguida en las mediciones.

B.1 Prueba de la plataforma

Para comprobar el funcionamiento de la plataforma y familiarizarse con el entorno, se llevó a cabo una primera ronda de mediciones. El programa de *benchmark* elegido para estas mediciones fue *gauss*, programado por Jesús Alastruey y Pablo Ibáñez. Dicho programa se compiló con *gcc*, aplicándole los niveles de optimización -O1, -O2, -O3, -O1 con -funroll-loops (desenrollado de bucles) y -O3 con -funroll-loops.

Después de compilar las distintas versiones de *gauss*, cada una de ellas se ejecutó y se midió 3 veces. Después de recogier los resultados, se realizaron gráficos para estudiar el comportamiento de las distintas versiones respecto al consumo energético, la disipación de potencia y la temperatura. Pudimos observar, que las gráficas de potencia y temperatura eran prácticamente iguales, ya que las diferencias eran menores de un 1 %; y lo mismo ocurría con los valores de la energía consumida. Como no había apenas diferencias entre las distintas ejecuciones, se tomó la decisión de realizar tan sólo una medición de cada optimización; ya que sería suficientemente representativa.

Además, en esta fase, se midió también la potencia disipada por el procesador en reposo (9.2 W) y la temperatura máxima alcanzada por el procesador (termopar T1, 72 °C). Para la medida de la temperatura máxima, se redujo la velocidad del procesador con un potenciómetro que tiene la plataforma y se ejecutó el programa *cpuburn* [15] hasta que la temperatura dejó de crecer. Estas mediciones tenían como objetivo determinar la temperatura máxima (en el termopar T1) a la que podía funcionar el procesador, para controlar que no se sobrepasara en nuestros experimentos. Conocer la temperatura máxima que alcanza el procesador es importante para las mediciones porque, si se llega a esa cota, el procesador reduce su frecuencia para poder disipar todo el calor. Esa reducción de la frecuencia introduciría distorsiones en las mediciones y daría lugar a resultados incorrectos. En cuanto a la potencia disipada en reposo por el procesador, es útil conocerla para controlar que en nuestros experimentos no bajara de esa cota.

Esta fase del proyecto, aparte de servir para familiarizarse con el entorno y aprender a usar la plataforma, fue muy útil cuando, un tiempo después, se produjeron errores y distorsiones en las mediciones debido a una elevada temperatura en la plataforma de medida. Como teníamos

almacenados los resultados de las pruebas de *gauss* y además este programa realiza un control muy fino de los tiempos de ejecución de los distintos tests (detecta varianzas mayores de 0.0025 en los tiempos de ejecución), pudimos comparar las gráficas de las dos mediciones y determinar que las distorsiones que se producían en los tiempos de ejecución se debían al aumento de temperatura del procesador. Este fallo se corrigió ajustando la velocidad de giro del ventilador al máximo posible ($\simeq$ 6750 rpm).

B.2 Realización de las mediciones

El protocolo para realizar las mediciones de cada optimización ha consistido en compilar las dos versiones del programa que queremos medir con los *flags* necesarios. Después, se ha analizado el código ensamblador de la versión no optimizada y de la optimizada, para asegurarse de que la optimización realmente se está aplicando. A continuación, se ha ejecutado 5 veces cada versión y se han tomado los tiempos de ejecución (comando `time` de *UNIX*) para comprobar que no hubiera diferencias entre distintas ejecuciones del mismo programa. Después, se realizaba la medición del programa. Para realizar una medición hay que seguir los siguientes pasos:

B.2.1 Preparación de la plataforma

Antes de empezar a medir, hay que preparar la plataforma. Estos son los pasos para ello:

1. Encender el amplificador y dejarlo encendido durante 20 minutos, que es su tiempo de calentamiento [21].

2. Tras ese período de tiempo, en el amplificador: poner *coupling* en modo DC, seleccionar el rango deseado (puede ser 1 A/V ó 10 A/V), cerrar la sonda amperimétrica (apoyada en la mesa, sin cables en su interior) y presionar el botón de *degauss*.

3. Conectar la sonda amperimétrica al par de cables amarillos que alimentan el procesador y cerrar la sonda.

4. En el ordenador que mide (OM), ejecutar como usuario *root*:
   `/pci-dask_426/drivers/dask_inst.pl`
   para cargar los módulos `p9112` y `adl_mem_mng`

5. En OM, ejecutar `lsmod` y comprobar que se han cargado los módulos `p9112` y `adl_mem_mng`

6. En OM, ejecutar `/libusbtc08-1.7.2/examples/*test*` para comprobar el funcionamiento de los termopares. En caso de que se produzca algún error, conviene desenchufar y volver a enchufar el conector *usb* de los termopares.

B.2.2 Realización de las mediciones

Ahora que la plataforma está lista, podemos realizar las mediciones. Para ello:

1. En OM, ejecutar `/pruebas/driver/capturador_datos` y pedirle que muestree 6 canales.

2. En el ordenador que ejecuta el programa (OE), ejecutar:
   `cliente direccion-IP-OM programa-que-medimos`
   la aplicación cliente realizar la sincronización entre OE y OM, por ello hay que pasarle como parámetros la dirección IP de la máquina OM y el programa que queremos medir.

3. Si se produce algún error durante el proceso de medición, volver al punto 1 de este apartado. Si en el amplificador se ilumina el led rojo de *overload*: desconectar la sonda de los cables, cambiar el rango a 10 A/V, hacer *degauss*, conectar de nuevo la sonda y volver al punto 1 de este apartado.

4. Si no ha habido ningún problema, en OM ejecutar `/pruebas/driver/conv_dat_to_txt.sh` para convertir los ficheros de datos de las mediciones en texto plano: `potencias.txt` y `temperaturas.txt`

5. Después, en OM también, ejecutar:
   `/pruebas/driver/hacer_graf potencias.txt temperaturas.txt salida.txt`
   para que deje los datos listos en `salida.txt` con un formato tabulado para dibujar gráficas.

Una vez hecha la primera medición, para realizar las siguientes basta con seguir de nuevo los pasos de este apartado.

Se ha comentado anteriormente que la aplicación `cliente` realiza una sincronización entre OE y OM. Para ello ejecuta el siguiente código:

```
1 send(socket, "start");
2 system(programa-que-medimos);
3 send(socket, "close");
```

Realiza una llamada `send()` para iniciar la ejecución. Cuando OM la recibe desbloquea `capturador_datos`, y OE ejecuta el programa que queremos medir con la orden del sistema `system`. Al finalizar la ejecución, realiza otra llamada a `send()` para avisar a OM de que ha terminado. Esta última llamada a `send()` también se mide; y esa medición puede provocar una pequeña distorsión al final, como se observa en algunas de las gráficas que aparecen en el capítulo 4.

Apéndice C
Impacto de las optimizaciones de compilación en la energía, potencia y temperatura

Este apéndice completa el capítulo 4 añadiendo los resultados del resto de optimizaciones estudiadas.

C.1 Loop-based strength reduction & induction variable elimination

Flags aplicados:

- Versión no optimizada: -O0
- Versión optimizada: -O1

Se aplica un nivel entero de optimización (-O1) porque el *script* de compilación del *testsuite* de *gcc* así lo indica. El *flag* -fivopts que controla el par de optimizaciones en teoría está activo desde el nivel -O0, pero en la práctica hay que activar al menos -O1 para que aplique las optimizaciones.

C.1.1 Descripción

Strength reduction reemplaza una expresión en un bucle por otra equivalente pero que utiliza un operador de menor latencia de ejecución. Dicho reemplazo se lleva a cabo aprovechando el recorrido en el espacio de iteraciones que hace el bucle. En la tabla C.1 se muestra cómo *strength reduction* puede aplicarse a algunas operaciones. La operación de la primera columna aparece en un bucle que itera sobre la variable i en un rango que va de 1 a n. Cuando el bucle es optimizado, el compilador inicializa una variable temporal T con la expresión de la segunda columna. La operación del bucle es sustituida por la expresión de la tercera columna, y el valor de T es actualizado en cada iteración con el valor de la cuarta columna.

Una vez aplicada *strength reduction* en las expresiones que contienen la variable de inducción, el compilador puede, a menudo, aplicar *induction variable elimination* para eliminar la variable de inducción original por completo. La condición de salida del bucle queda en función de la variable de inducción reducida, por ejemplo, la dirección del último elemento de un vector. Esta optimización no sólo elimina operaciones del bucle como el incremento de la variable de inducción en cada iteración, también libera los registros utilizados por dicha variable de inducción.

Expresión	Inicialización	Uso	Actualización
$c * i$	$T = c$	T	$T = T + c$
c^i	$T = c$	T	$T = T * c$
$(-1)^i$	$T = -1$	T	$T = -T$
x/c	$T = 1/c$	$x * T$	

Tabla C.1: *Strength reduction* (la variable c es invariante, x puede variar entre diferentes iteraciones).

C.1.2 Código ejemplo

En este apartado veremos como se aplica primero *strength reduction* a un código de ejemplo. Después se traduce a un pseudocódigo ensamblador y sobre ese ensamblador se aplica la optimización *induction variable elimination*. En primer lugar el código original, cuya operación $c*i$ (c constante, i variable de inducción) quiere reemplazarse por otra de menor latencia de ejecución:

```
1  do i = 1, n
2     a[i] = a[i] + c*i
3  end do
```

Al aplicar *strength reduction*, de acuerdo con la tabla C.1, el código se transforma en:

```
1  T = c
2  do i = 1, n
3     a[i] = a[i] + T
4     T = T + c
5  end do
```

Al traducir este código a ensamblador quedaría de la siguiente manera:

```
1           LOAD   r0, c             ;T = c
2           LOAD   r1, n             ;r1 = n
3           LOAD   r2, 1             ;i = 1
4           LOAD   r3, a             ;r3 = @(a[1])
5  loop:    LOAD   r4, (r3)          ;r4 = a[i]
6           ADD    r4, r4, r0        ;r4 = a[i] + c
7           STORE  (r3), r4          ;store de a[i]
8           ADD    r0, r0, r0        ;T = T + c
9           ADD    r2, r2, 1         ;i = i + 1
10          ADD    r3, r3, desp      ;r3 = @ siguiente elemento del vector
11          CMP    r5, r2, r1        ;i < n
12          JMPNZ  r5, loop
```

Sobre este código puede aplicarse *induction variable elimination* para eliminar la operación de incremento de la variable de inducción (línea 9) y liberar su registro ($r2$ no se utiliza dentro del bucle).

```
1           LOAD   r0, c             ;T = c
2           LOAD   r1, n             ;r1 = n
3           LOAD   r2, desp          ;r2 = desplazamiento
4           LOAD   r3, a             ;r3 = @(a[1])
5           MUL    r1, r1, r2        ;r1 = n * desplazamiento
6           ADD    r1, r1, r3        ;r1 = @(a[n+1])
```

```
7  loop:   LOAD   r4, (r3)          ;r4 = a[i]
8          ADD    r4, r4, r0        ;r4 = a[i] + c
9          STORE  (r3), r4          ;store de a[i]
10         ADD    r0, r0, r0        ;T = T + c
11         ADD    r3, r3, desp      ;r3 = @ siguiente elemento del vector
12         CMP    r5, r3, r1        ;@(a[i]) < @(a[n+1])
13         JMPNZ  loop
```

C.1.3 Resultados

La optimización *strength reduction* no reduce N_{inst} pero, al sustituir algunas instrucciones por otras, puede afectar al CPI haciendo que disminuya ya que las nuevas operaciones tardan menos ciclos de CPU.

Por otro lado, *induction variable elimination* elimina la variable de inducción del bucle, reduciendo el número de instrucciones ejecutadas N_{inst}, pero no podemos saber qué ocurre con el CPI. Al eliminar instrucciones como el incremento de la variable de inducción (menos costosa en términos de ciclos de CPU que la media de instrucciones), podría suceder que el CPI aumentara.

Al actuar conjuntamente estas dos optimizaciones, podemos decir que si logran reducir el tiempo de ejecución (ecuación 3.1) sin penalizar C_L en la ecuación 3.9, la energía consumida disminuirá también (ecuación 3.8). En cambio, no podemos determinar *a priori* qué ocurre con la P_{tot} ya que una variación en el CPI puede aumentar o disminuir la actividad por ciclo C_L de la ecuación 3.5.

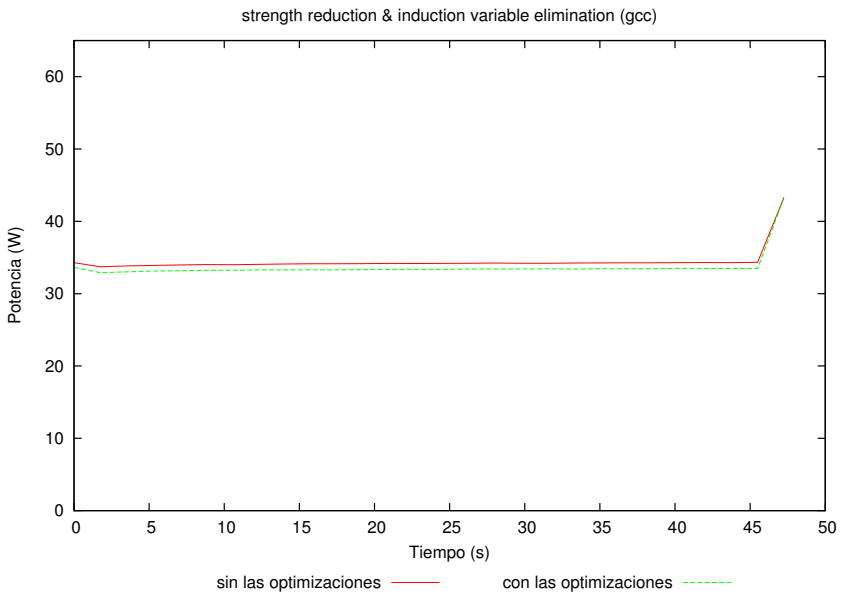

Figura C.1: Potencia usando *strength reduction & induction variable elimination*.

Tal y como se aprecia en la figura C.1, el par de optimizaciones actúa reduciendo ligeramente la potencia y el tiempo de ejecución en la versión optimizada. Esta diferencia en cuanto a potencia y tiempo de ejecución es tan pequeña debido a que en la versión no optimizada la multiplicación no se hace con una operación tipo MUL en bajo nivel. El compilador traduce dicha multiplicación

en una serie de desplazamientos aritméticos y sumas, con lo que la multiplicación se realiza ya de manera eficiente en la versión no optimizada. En cuanto a la energía, teniendo en cuenta la ecuación 3.7 podemos calcular el consumo energético, viendo cómo disminuye ligeramente en la versión optimizada ya que tanto la potencia como el tiempo de ejecución disminuyen. El resultado se muestra en la tabla C.2.

Este fragmento de código ensamblador muestra las líneas 14-28 de la versión no optimizada. En él puede verse cómo la multiplicación se hace con desplazamientos aritméticos (instrucción *sall*):

```
1  .L4:
2          movl    -4(%ebp), %edx  # iter, tmp60
3          movl    %edx, %eax      # tmp60, tmp61
4          sall    $4, %eax        #, tmp62
5          leal    (%eax,%edx), %ecx       #, D.3162
6          movl    -4(%ebp), %eax  # iter, tmp63
7          sall    $6, %eax        #, tmp65
8          leal    0(,%eax,8), %edx        #, tmp66
9          subl    %eax, %edx      # tmp64, tmp66
10         leal    arr_base+176(%edx), %eax        #, tmp67
11         movl    %ecx, 12(%eax)  # D.3162, arr_base[iter_1].y
12         addl    $1, -4(%ebp)    #, iter
13 .L3:
14         cmpl    $599999, -4(%ebp)       #, iter
15         jle     .L4     #,
```

Este extracto muestra las líneas 9-14 de la versión optimizada. En él puede observarse cómo se sustituye la multiplicación por la suma (instrucción *addl $17, %eax*) y se elimina la variable de inducción porque se compara con el valor de la multiplicación de la última iteración (instrucción *cmpl $10200000, %eax*):

```
1  .L3:
2          movl    %eax, (%edx)    # ivtmp.8, arr_base[iter].y
3          addl    $17, %eax       #, ivtmp.8
4          addl    $448, %edx      #, ivtmp.9
5          cmpl    $10200000, %eax #, ivtmp.8
6          jne     .L3     #,
```

El resto del código se encuentra en el apéndice E.

Versión	Energía (Julios)	Energía normalizada
No optimiz.	1689.20	1
Optimizada	1651.32	0.98

Tabla C.2: Energía usando *strength reduction & induction variable elimination.*

En cuanto a la temperatura, podemos observar en la figura C.2 que sigue el mismo comportamiento y alcanza prácticamente los mismos valores en ambas versiones (no optimizada y optimizada) debido a que las dos versiones están disipando prácticamente la misma potencia. Tal y como se comenta en el apéndice A, los termopares T1, T5 y T6 miden la temperatura en el centro del procesador, a la entrada del aire, y a la salida del aire, respectivamente.

El comportamiento térmico es el esperado. La temperatura aumenta progresivamente a lo largo de la ejecución, pero sin sobrepasar la cota de 72 °C, que es la asíntota hacia la que convergen las temperaturas máximas obtenidas en las pruebas de laboratorio.

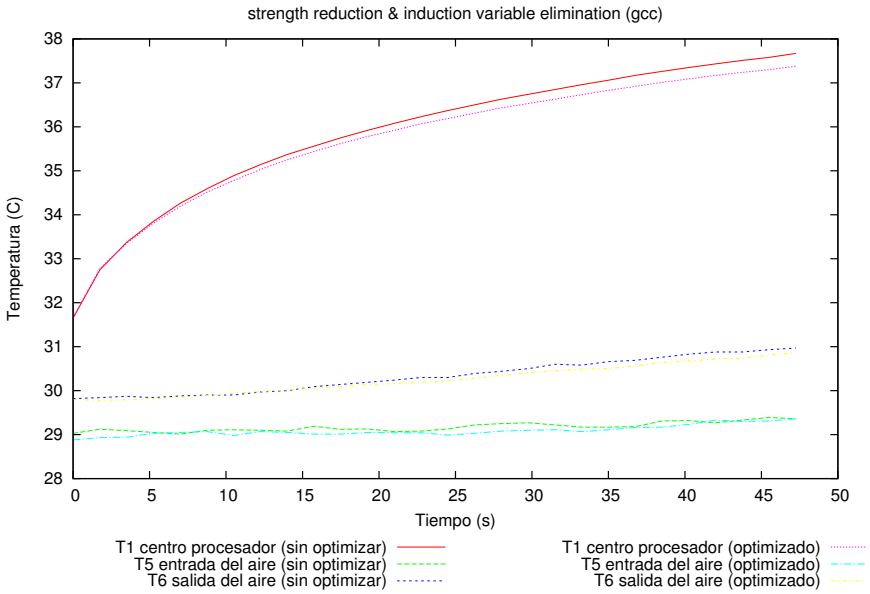

Figura C.2: Temperaturas usando *strength reduction & induction variable elimination.*

C.2 Loop unrolling

Flags aplicados:

- Versión no optimizada: -O1
- Versión optimizada: -O1 -funroll-loops

C.2.1 Descripción

La optimización de *loop unrolling* consiste en replicar el contenido del bucle un número determinado de veces, llamado el factor de desenrollado *d*. Así, la iteración en vez de avanzar de 1 en 1, lo hace de *d* en *d*. *Loop unrolling* puede mejorar el rendimiento al reducir la sobrecarga del bucle (reduce el número de instrucciones ejecutadas), aumentar el paralelismo a nivel de instrucciones (ILP) y mejorar la localidad de la *cache* de datos. Por contra, si el factor de desenrollado *d* es muy grande, el código generado puede ser demasiado grande para la *cache* de instrucciones y producir tantos fallos en ella que empeoren el resultado que obtendríamos si no aplicáramos la optimización. Por ello, compiladores como *gcc* y *llvm* no aplican *loop unrolling* por defecto en ningún nivel de optimización. Dejan al usuario la responsabilidad de activarla o no.

C.2.2 Código ejemplo

En este apartado vamos a ver un ejemplo sencillo de desenrollado de bucles. El cuerpo del bucle (línea 3) se desenrollará tantas veces como indique el factor de desenrrollado. A continuación el código sobre el que aplicaremos desenrollado:

```
1 do i = 2, n-1
2    a[i] = a[i] + a[i-1] * a[i+1]
3 end do
```

Si aplicamos *loop unrolling*, con un factor de desenrollado $d = 2$, el bucle queda de la siguiente manera:

```
1 do i = 2, n-2, 2
2    a[i] = a[i] + a[i-1] * a[i+1]
3    a[i+1] = a[i+1] + a[i] * a[i+2]
4 end do
5
6 /* epilogo */
7 if (mod(n-2, 2) = 1) then
8    a[n-1] = a[n-1] + a[n-2] * a[n]
9 end if
```

Sea N el número de iteraciones y d el factor de desenrollado. Si:

$$N \text{ mód } d \neq 0 \tag{C.1}$$

quedará un número de iteraciones residuales por hacerse. Por ello, se añade el epílogo al final del bucle desenrollado (línea 6 del código de ejemplo). Para $d > 2$, el epílogo consiste en un bucle con las iteraciones que faltan.

C.2.3 Resultados

Loop unrolling reduce el número de instrucciones ejecutadas N_{inst}, por lo que si no se producen muchos más fallos en cache que afecten de manera importante al CPI, tanto el tiempo como la energía consumida por la versión optimizada serán menores (ver ecuaciones 3.8 y 3.1).

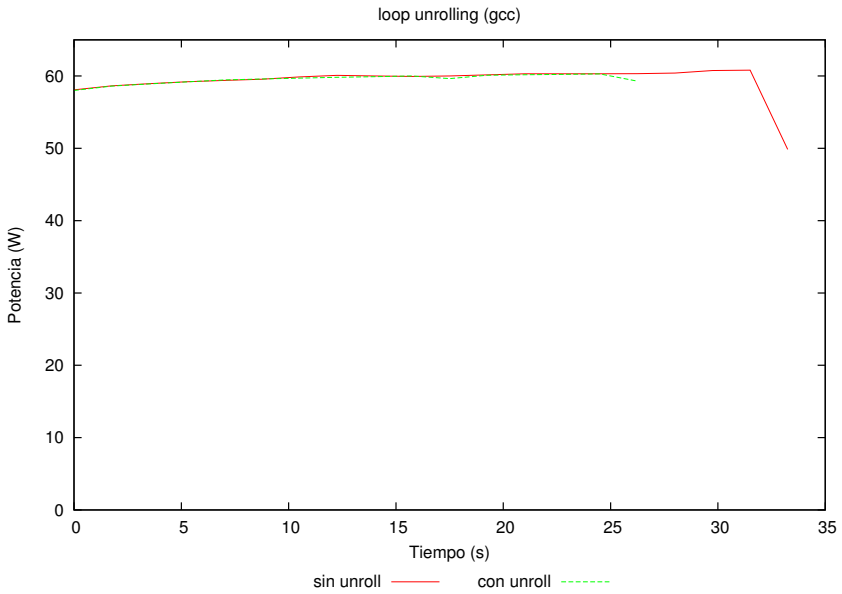

Figura C.3: Potencia usando *loop unrolling*.

Tal y como se muestra en la figura C.3 la potencia disipada en ambas versiones sigue el mismo comportamiento, y la potencia media disipada es prácticamente igual. Pero el tiempo de ejecución T_{ex} es menor en la versión optimizada ya que *loop unrolling* elimina instrucciones reduciendo la sobrecarga del bucle. Tal y como puede observarse en el apéndice E, en la versión optimizada hace un 8-desenrollado del bucle haciendo que el número de instrucciones ejecutadas por el bucle principal del programa descienda de 7000000 a 5125000. Operando con la ecuación 3.7 podemos observar como la energía consumida por la versión optimizada es un 20 % menor (ver tabla C.3).

Versión	Energía (Julios)	Energía normalizada
No optimiz.	2077.10	1
Optimizada	1667.52	0.80

Tabla C.3: Energía usando *loop unrolling*.

En cuanto a la disipación térmica, sigue el mismo comportamiento en las dos versiones (ver figura C.4) ya que la potencia disipada a lo largo de la ejecución es igual en ambas versiones.

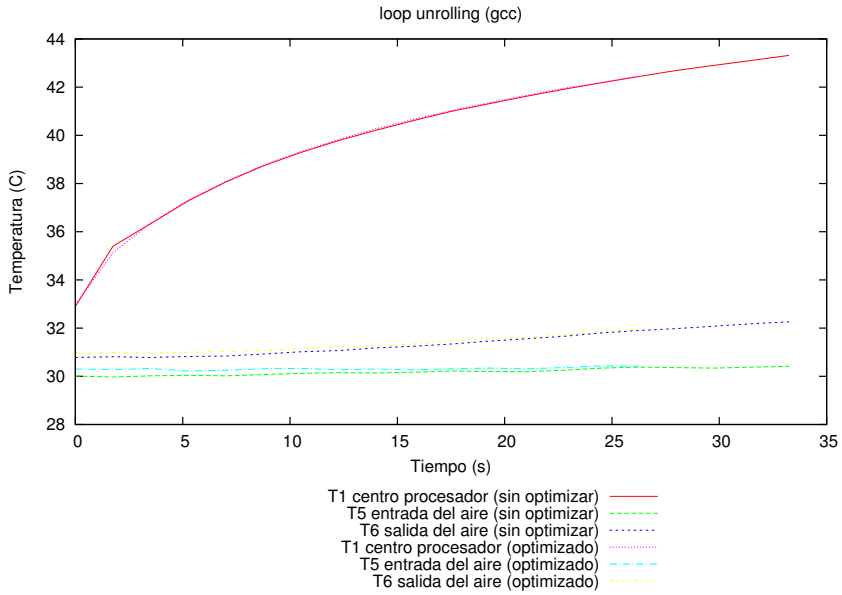

Figura C.4: Temperaturas usando *loop unrolling*.

C.3 Scalar replacement

Flags aplicados:

- Versión no optimizada: -O1
- Versión optimizada: -O1 -fipa-sra

C.3.1 Descripción

Scalar replacement puede aplicarse a un elemento de un vector al que se accede continuamente, sustituyéndolo por un escalar (y permitiendo, presumiblemente, que sea cargado en un registro). Cuando se aplica intraproceduralmente, esta optimización permite también eliminar los parámetros no utilizados por los procedimientos y sustituir los parámetros que se pasan por referencia por parámetros pasados por valor.

C.3.2 Código ejemplo

Aquí un ejemplo de código que ilustra cómo *scalar replacement* sustituye un acceso a un vector por un escalar:

```
1 do i = 1, n
2    do j = 1, n
3       total[i] = total[i] + a[i, j]
4    end do
5 end do
```

El elemento *total[i]* de la línea 3 es invariante respecto al bucle más interno (itera en j), por lo tanto puede sustituirse por un escalar y actualizarse cuando lo haga la variable i de la que depende.

```
1 do i = 1, n
2    T = total[i]
3    do j = 1, n
4       T = T + a[i, j]
5    end do
6 end do
```

En la línea 2 del código optimizado se carga *total[i]* en el escalar T. En la línea 4 se accede a T para hacer la operación.

C.3.3 Resultados

En este experimento cabe señalar que el código medido es radicalmente distinto al que se muestra en el apartado "código ejemplo", donde se muestra un código lo más sencillo posible para explicar la optimización. El código real medido en el experimento está sacado del *testsuite* de *gcc* y es bastante más largo. Se encuentra, como todos los códigos utilizados en las mediciones, en el apéndice E.

La optimización *scalar replacement* actúa sustituyendo variables o valores no escalares (como vectores, listas, objetos, ...) que se acceden con mucha frecuencia por escalares. Esto hace que

las instrucciones tipo *load* y *store* disminuyan. Además, al aplicarlo intraproceduralmente, elimina los parámetros no utilizados y sustituye los parámetros por referencia por parámetros por valor, por lo que el número total de instrucciones N_{inst} disminuye (tal y como puede observarse en el apéndice E, donde en la versión optimizada se reduce el número de instrucciones *mov* ya que se eliminan los parámetros que no se utilizan). En cuanto al CPI, lo más seguro es que al eliminar instrucciones tipo *load* y *store* disminuya algo, ya que eliminamos instrucciones de mayor latencia de ejecución que la media. Esa disminución del CPI puede causar un aumento de C_L que haga aumentar la potencia dinámica y la total (ecuaciones 3.4 y 3.5).

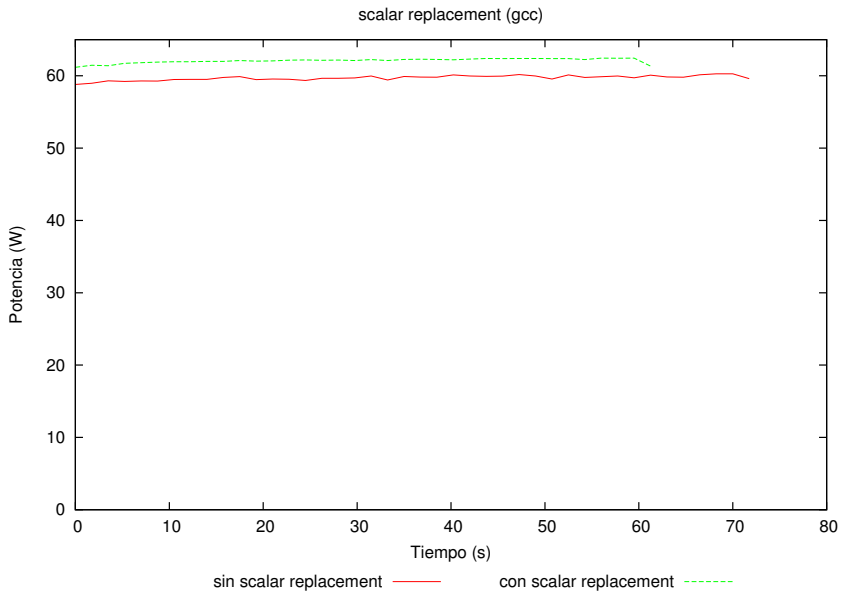

Figura C.5: Potencia usando *scalar replacement*.

Tal y como se muestra en la figura C.5 podemos observar que la optimización mejora el tiempo de ejecución T_{ex} debido que el número de instrucciones ejecutadas N_{inst} y su latencia de ejecución son menores, como se ha comentado anteriormente. En cuanto a la potencia, la versión optimizada disipa más potencia que la no optimizada debido al aumento de C_L. Para calcular la energía consumida, tenemos que integrar con la ecuación 3.7. En la tabla C.4 podemos observar cómo, a pesar de una mayor potencia disipada, su consumo energético es un 11 % menor en la versión optimizada debido a la reducción del tiempo de ejecución.

Versión	Energía (Julios)	Energía normalizada
No optimiz.	4389.17	1
Optimizada	3911.55	0.89

Tabla C.4: Energía usando *scalar replacement*.

En cuanto a la disipación térmica, conviene señalar que la diferencia de potencia disipada entre la versión optimizada y la que no lo está se traduce en una mayor temperatura en el procesador a lo largo de la ejecución de la versión optimizada (ver figura C.6).

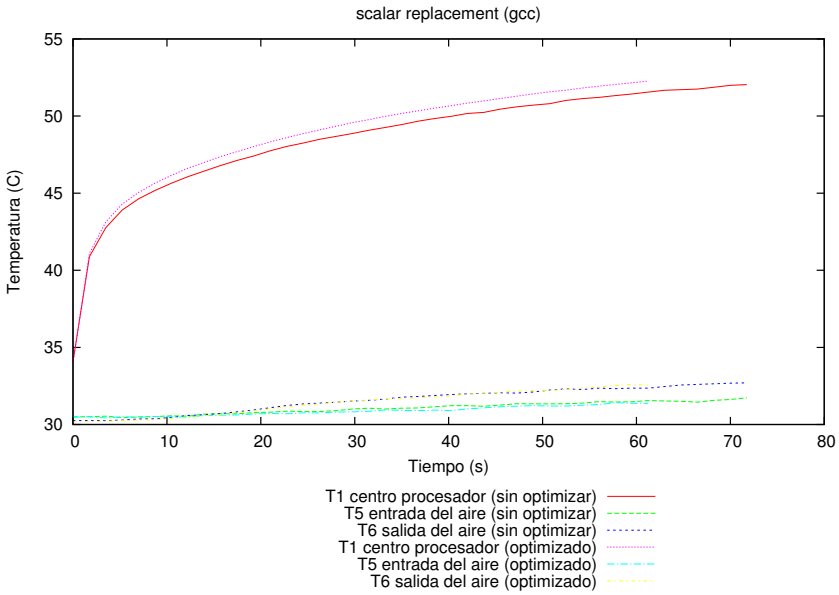

Figura C.6: Temperaturas usando *scalar replacement*.

C.4 Constant propagation

Flags aplicados:

- Versión no optimizada: -O0
- Versión optimizada: -O1

El *flag* que activa la optimziación se activa en -O1 (-ftree-ccp).

C.4.1 Descripción

Constant propagation es una de las optimizaciones más importantes que puede hacer un compilador. Los programas suelen tener muchas constantes, al propagarlas a través de todo el programa, el compilador puede sustituir un buen número de esas constantes por su valor en tiempo de compilación. Esta optimización posiblita, además, optimizaciones posteriores como *dead-code elimination* y optimizaciones de bucles, ya que la propagación de constantes revela, a menudo, el rango de iteración de los bucles.

C.4.2 Código ejemplo

Aquí se muestra un código de ejemplo en el que las variables n y c tienen valor constante.

```
1 n = 64
2 c = 3
3 do i = 1, n
4    a[i] = a[i] + c
5 end do
```

Al aplicar *constant propagation* el valor de las variables n y c se propaga hacia delante.

```
1 do i = 1, 64
2    a[i] = a[i] + 3
3 end do
```

C.4.3 Resultados

Constant propagation actúa, esencialmente, haciendo que disminuya el número de instrucciones N_{inst}. En cuanto al CPI, no sabemos *a priori* cómo puede afectarle la disminución del número de instrucciones. Si se eliminan instrucciones tipo *load* y *store* podría disminuir también (la constante vendría codificada en la propia instrucción, con lo que no es necesario hacer el acceso a memoria). Como podemos observar en la figura C.7 la potencia media disipada es ligeramente inferior en la versión optimizada. En cuanto al tiempo de ejecución T_{ex}, debido a la drástica reducción del número de instrucciones, es mucho menor.

En este extracto de código ensamblador del apéndice E pueden observarse las líneas 41-52 de la versión no optimizada. El valor de las variables se carga en cada iteración:

```
1 .L4:
2         movl    -36(%ebp), %eax # d, tmp86
3         movl    -32(%ebp), %edx # c, tmp87
```

```
4           leal    (%edx,%eax), %ecx       #, D.3174
5           movl    -28(%ebp), %eax # a.5, tmp88
6           movl    -16(%ebp), %edx # j, tmp89
7           movl    %ecx, (%eax,%edx,4)     # D.3174,
8           addl    $1, -16(%ebp)   #, j
9   .L3:
10          movl    -16(%ebp), %eax # j, tmp90
11          cmpl    -20(%ebp), %eax # n, tmp90
12          jl      .L4     #,
```

En cambio en la versión optimizada se conoce ya el valor de la suma de las variables (*movl $7*) por lo que no hay que cargarlas y realizar la operación, como puede verse en este extracto de las líneas 21-25:

```
1   .L3:
2           movl    $7, (%edx,%eax,4)       #,* a.2
3           addl    $1, %eax        #, j
4           cmpl    $64, %eax       #, j
5           jne     .L3     #,
```

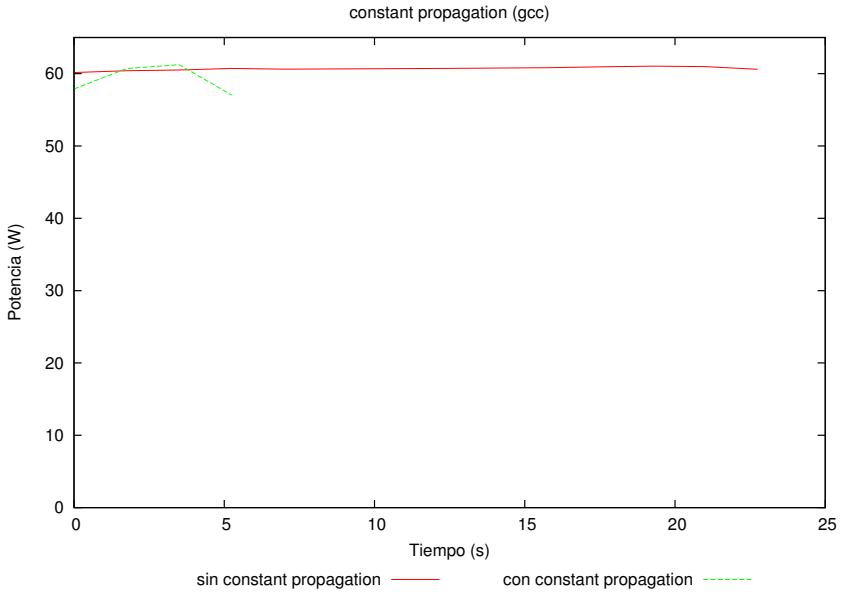

Figura C.7: Potencia usando *constant propagation*.

Integrando con la ecuación 3.7 sobre el resultado obtenido en la figura C.7 obtenemos los resultados de la tabla C.5 donde podemos oberservar que, debido a la reducción de N_{inst} el consumo de energía es un 72 % menor en la versión optimizada.

Versión	Energía (Julios)	Energía normalizada
No optimiz.	1486.90	1
Optimizada	414.51	0.28

Tabla C.5: Energía usando *constant propagation*.

En cuanto a la disipación térmica, sigue el mismo comportamiento en las dos versiones (ver

figura C.6), ya que la potencia disipada también es similar en ambas versiones.

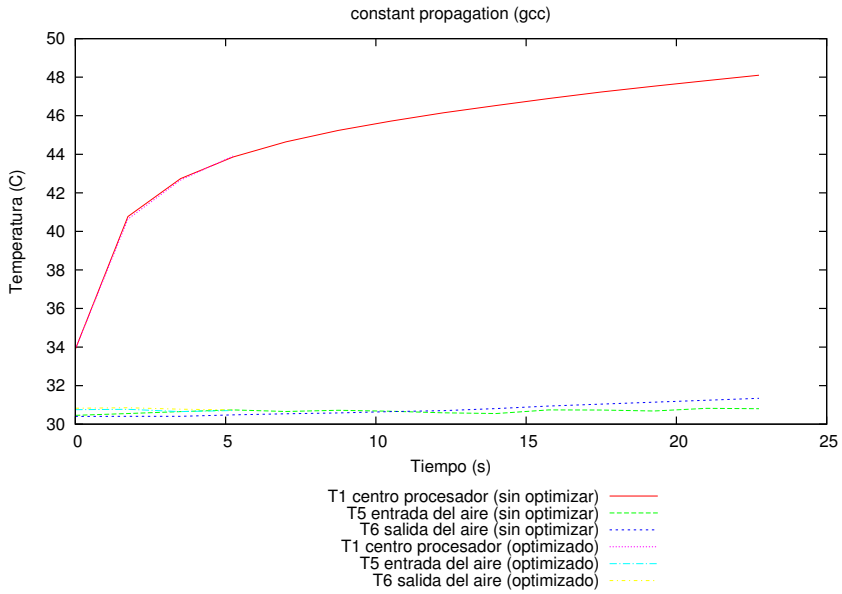

Figura C.8: Temperaturas usando *constant propagation*.

C.5 Copy propagation

Flags aplicados:

- Versión no optimizada: -O0
- Versión optimizada: -O1 -fno-gcse-lm

El *flag* -fno-gcse-lm desactiva la eliminación de subexpresiones comunes que aparece en el nivel -O1, para evitar que aplique más optimizaciones de las que realmente proceden.

C.5.1 Descripción

El compilador propaga la variable original y elimina las copias redundantes. Se aplica generalmente porque optimizaciones como la eliminación de variables de inducción y la eliminación de subexpresiones comunes suelen dar lugar a varias copias de un mismo valor original. Al hacer *copy propagation* se reduce el uso de registros y se eliminan instrucciones tipo *move* entre registros.

C.5.2 Código ejemplo

En este apartado se muestra un código de ejemplo en el que se puede aplicar *copy propagation* para eliminar copias redundantes.

```
1 t = i*4
2 s = t
3 print *, a[s]
4 r = t
5 a[r] = a[r] + c
```

Al aplicar *copy propagation* se propaga t a lo largo del código y se eliminan sus copias redundantes.

```
1 t = i*4
2 print *, a[t]
3 a[t] = a[t] + c
```

C.5.3 Resultados

Copy propagation actúa disminuyendo el número de instrucciones ejecutadas N_{inst}. Dicha disminución afecta al tiempo de ejecución T_{ex} debido a la ecuación 3.1 y también al consumo energético por la ecuación 3.8. En cuanto al CPI, si se eliminan instrucciones tipo *load* y *store* puede bajar, ya que suelen tardar más ciclos de media; pero si, además, se eliminan instrucciones tipo *move* entre registros, no podemos asegurar *a priori* cómo variará CPI.

Tal y como observamos en la figura C.9, tanto la disipación de potencia P_{tot} como el tiempo de ejecución T_{ex} son mucho menores en la versión optimizada. Esto se debe a que hay una reducción muy importante en el número de instrucciones ejecutadas y en los accesos a memoria. Integrando con la fórmula de la ecuación 3.7 podemos calcular la tabla C.6 y ver cómo el consumo energético desciende drásticamente en la versión optimizada: un 87 %.

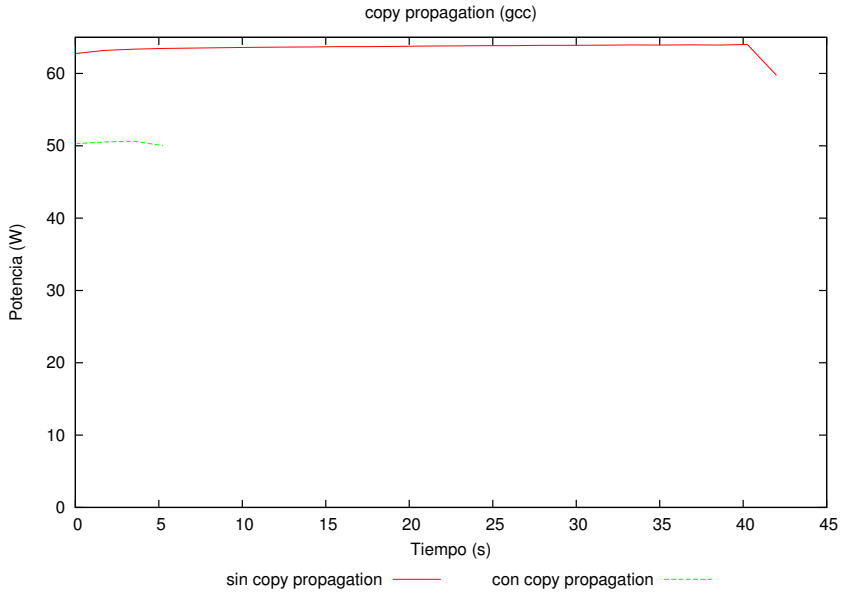

Figura C.9: Potencia usando *copy propagation*.

Versión	Energía (Julios)	Energía normalizada
No optimiz.	2779.81	1
Optimizada	352.59	0.13

Tabla C.6: Energía usando *copy propagation*.

En cuanto a las temperaturas alcanzadas (figura C.10), podemos observar cómo la versión optimizada alcanza temperaturas muy inferiores en la CPU (termopar T1). Esto se debe a que la disipación de potencia en la versión optimizada es mucho menor que en la que no lo está (comparar la figura C.10 con la figura C.9).

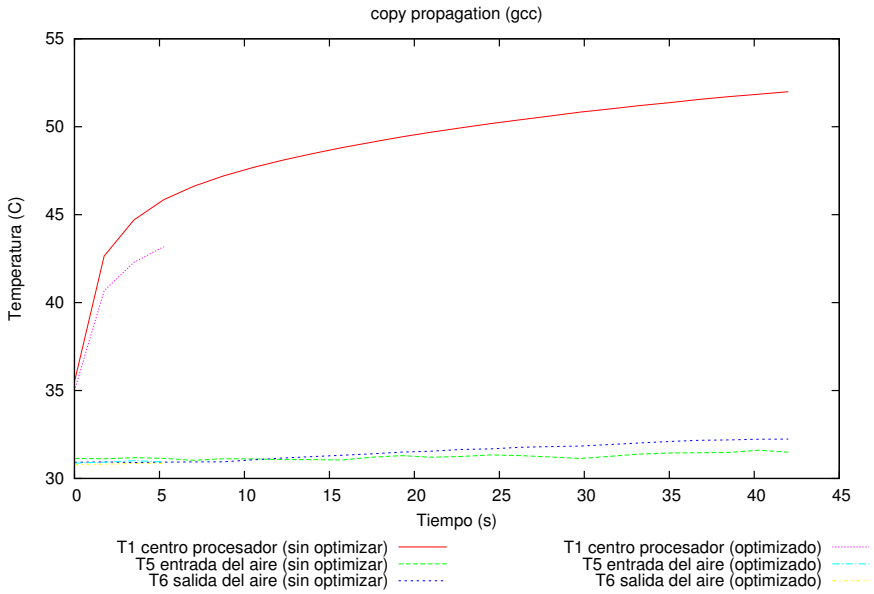

Figura C.10: Temperaturas usando *copy propagation*.

C.6 Unreachable-code elimination & useless-code elimination

Flags aplicados:

- Versión no optimizada: -O0
- Versión optimizada: -O1 -fdce

C.6.1 Descripción

Unreachable-code elimination elimina el código que nunca se ejecuta en un programa, por ejemplo una rama de un condicional a la que nunca se accede porque la condición es invariante, o un bucle que no realiza ninguna iteración. *Useless-code elimination* elimina el código que no afecta al resultado final de la ejecución. Esta optimización se suele aplicar tras *unreachable-code elimination*, ya que ésta deja tras su aplicación código que no tiene ninguna utilidad. A este par de optimizaciones se las conoce también por el nombre de *dead-code elimination*.

C.6.2 Código ejemplo

A continuación se muestra un código de ejemplo en el que pueden aplicarse consecutivamente las dos optimizaciones.

```
1  debug = 0
2  n = 0
3  a = b+7
4  if (debug > 1) then
5     c = a + b + d
6     print *, 'Warning -- total is ', c
7  end if
8  call foo(a)
9  do i = 1, n
10    a[i] = a[i] + c
11 end do
```

Con ayuda de *constant propagation* se puede ver que el condicional de la línea 4 y el bucle de la línea 9 no se ejecutarán. Al aplicar *unreachable-code elimination* el código queda así:

```
1  debug = 0
2  n = 0
3  a = b+7
4  if (0 > 1) then
5  end if
6  call foo(a)
7  do i = 1, 0
8  end do
```

Ahora el condicional de la línea 4 queda vacío, y lo mismo ocurre con el bucle de la línea 7. Aplicando *useless-code elimination* el código queda de la siguiente manera:

```
1  a = b+7
2  call foo(a)
```

C.6.3 Resultados

Este par de optimizaciones actúa conjuntamente reduciendo el número total de instrucciones ejecutadas N_{inst} ya que aparte de eliminar el código al que nunca se accede también se elimina el código cuya ejecución no afecta al resultado final. Por ello, al aplicar la optimización el tiempo de ejecución T_{ex} disminuirá (ver ecuación 3.1) y la energía total consumida también descenderá (ver ecuación 3.8). En cuanto al CPI, es difícil aventurar qué ocurrirá con él, ya que el par de optimizaciones actúa eliminando instrucciones de todo tipo.

Al observar el resultado de las mediciones (figura C.11) vemos que el tiempo de ejecución desciende en la versión optimizada. Esto se debe a que se reduce el número de instrucciones ejecutadas (consultar apéndice E), entre ellas instrucciones aritméticas y de acceso a memoria, que tienen más latencia de ejecución. Integrando con la ecuación 3.7 podemos ver que el consumo de energía, efectivamente, desciende en la versión optimizada un 30 % (ver tabla C.7) debido tanto al descenso en tiempo de ejecución como de potencia disipada.

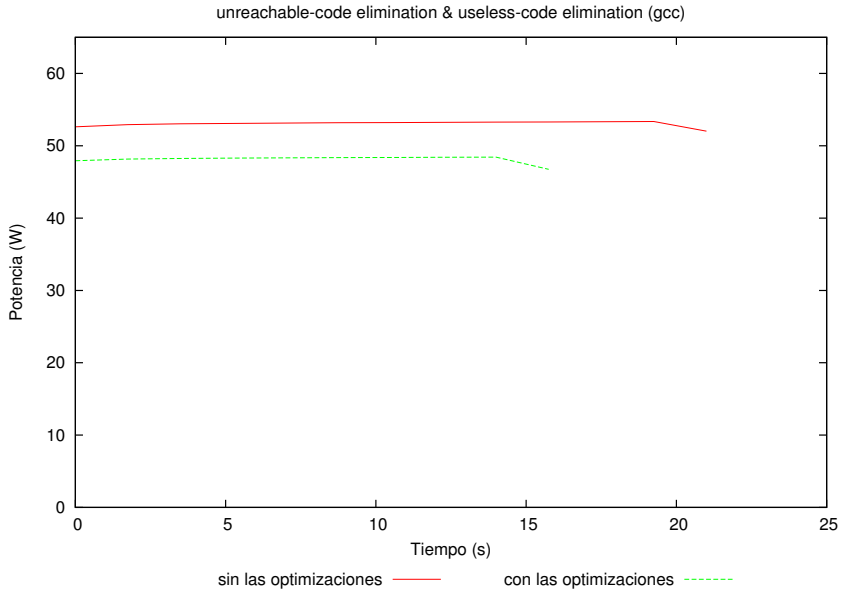

Figura C.11: Potencia usando *unreachable-code elimination* & *useless-code elimination*.

Versión	Energía (Julios)	Energía normalizada
No optimiz.	1207.08	1
Optimizada	842.29	0.70

Tabla C.7: Energía usando *unreachable-code elimination* & *useless-code elimination*.

En cuanto al comportamiento térmico (figura C.11), la temperatura alcanzada en la versión no optimizada es mayor que en la versión optimizada ya que la disipación de potencia es también mayor en la versión no optimizada.

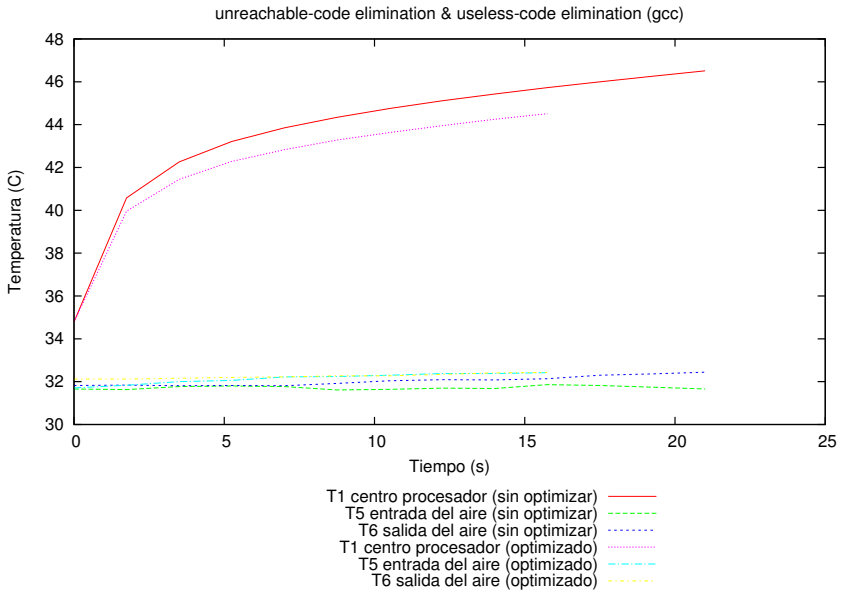

Figura C.12: Temperaturas usando *unreachable-code elimination & useless-code elimination.*

C.7 Procedure inlining

Flags aplicados:

- Versión no optimizada: -O2 -fno-inline-functions-called-once -fno-inline-small-functions -fno-inline-functions
- Versión optimizada: -O2 -finline-functions -finline-small-functions

C.7.1 Descripción

Procedure inlining sustituye una llamada a un procedimiento por el cuerpo (o código) del procedimiento invocado. Esto contribuye a eliminar la sobrecarga de las invocaciones, pero también hace que el código final aumente de tamaño; especialmente si se aplica a procedimientos grandes. Además de ser una optimización por sí sola, *procedure inlining* también ayuda a poder a aplicar otras optimizaciones. Al expandir el código de los procedimientos invocados el compilador tiene más información del código y puede eliminar operaciones redundantes, propagar constantes, eliminar código muerto y mejorar el uso de los registros. Por contra, el aumento de tamaño de código que conlleva puede llevar a que secciones críticas de código empeoren su comportamiento en la *cache* de instrucciones, haciendo que el programa empeore su rendimiento.

C.7.2 Código ejemplo

En este apartado se presenta un código de ejemplo, en el cual se invoca a una subrutina a la que se le puede aplicar *inlining*.

```
1 subroutine f(x, j)
2    dimension x[*]
3    x[j] = x[j] + c
4    return
5 end subroutine
6
7 do i = 1, n
8    call f(a, i)
9 end do
```

Al aplicar *procedure inlining*, el cuerpo de la subrutina f se expande dentro del bucle donde se invoca (línea 8):

```
1 do all i = 1, n
2    a[i] = a[i] + c
3 end do all
```

Tras aplicar la optimización, el compilador tiene más información del bucle y decide que puede paralelizarse (bucle *do all* de la línea 1).

C.7.3 Resultados

Al eliminar la sobrecarga de las invocaciones, desciende el número de instrucciones ejecutadas N_{inst} ya que los cambios de contexto requieren operaciones tipo *pop* y *push* en la pila. Según la ecuación 3.1 la disminución del número de instrucciones reducirá también el tiempo de ejecución T_{ex}, siempre y cuando el CPI no aumente (proporcionalmente) más de lo que disminuye N_{inst}.

En cuanto al CPI, no podemos saber *a priori* cómo se comportará, ya que al eliminar la sobrecarga de las invocaciones lo más probable es que disminuya; pero si el código aumenta mucho de tamaño y se producen muchos más fallos en *cache*, entonces puede que aumente.

En el apéndice E podemos ver cómo en la versión no optimizada, se realiza la invocación al procedimiento *mcd()* como se muestra en este extracto de la línea 99 del ensamblador:

```
1           call    mcd     #
```

En cambio, en la versión optimizada se sustituye la invocación al procedimiento por su código como puede verse en este extracto de las líneas 92-110 de la versión optimizada:

```
1           movl    %edi, %esi      # D.3248, b
2           je      .L9     #,
3           movl    %edi, %ecx      # b, c
4           movl    %ebx, %eax      # a, c
5           jmp     .L10    #
6           .p2align 4,,7
7           .p2align 3
8   .L14:
9           movl    %ecx, %eax      # c, c
10          movl    %edx, %ecx      # tmp78, c
11  .L10:
12          movl    %eax, %edx      # c, tmp78
13          sarl    $31, %edx       #, tmp78
14          idivl   %ecx    # c
15          testl   %edx, %edx      # tmp78
16          jne     .L14    #,
17  .L9:
18          cmpl    $3333, %ecx     #, c
19          je      .L16    #,
```

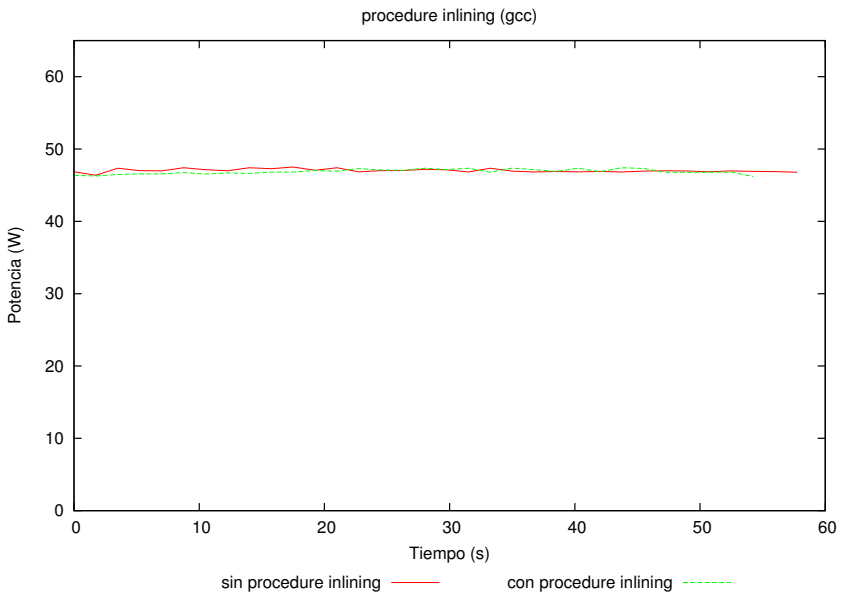

Figura C.13: Potencia usando *procedure inlining*.

En la figura C.13 podemos observar cómo el tiempo de ejecución T_{ex} desciende en la versión optimizada, debido a que se han eliminado las instrucciones correspondientes a la invocación de procedimientos y el cambio de contexto. En cambio, la potencia disipada P_{tot} sigue un comportamiento parecido en las dos versiones, lo que indica que el CPI no ha variado mucho entre las dos versiones. En cuanto a la energía consumida E_{tot}, al integrar con la ecuación 3.7 obtenemos los resultados de la tabla C.8, en la que podemos observar que el consumo energético en la versión optimizada desciende en un 6 %.

Versión	Energía (Julios)	Energía normalizada
No optimiz.	2797.88	1
Optimizada	2625.47	0.94

Tabla C.8: Energía usando *procedure inlining*.

En cuanto al comportamiento térmico, es muy similar en las dos versiones tal y como puede apreciarse en la figura C.14. La temperatura aumenta en ambas versiones a lo largo de la ejecución, siguiendo la misma curva y alcanzando los mismos valores, debido a que la disipación de potencia es muy similar también en ambas versiones (figura C.13).

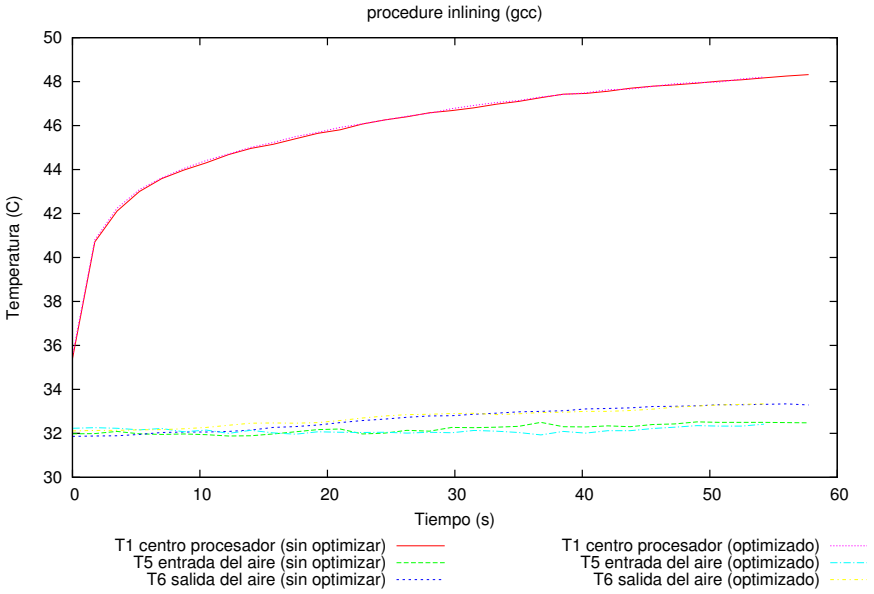

Figura C.14: Temperaturas usando *procedure inlining*.

C.8 Procedure cloning

Flags aplicados:

- Versión no optimizada: -O3 -fno-ipa-cp -fno-ipa-cp-clone -fno-early-inlining
- Versión optimizada: -O3 -fipa-cp -fipa-cp-clone -fno-early-inlining

C.8.1 Descripción

Procedure cloning crea copias de los procedimientos, especializándolos según el valor de sus parámetros.

C.8.2 Código ejemplo

En este apartado se muestra un código de ejemplo sobre el que puede aplicarse *procedure cloning*.

```
1  subroutine f(x, n, p)
2     real x[*]
3     integer n, p
4     do i = 1, n
5        x[i] = x[i]**p
6     end do
7  end subroutine
8
9  call f(a, n, 2)
```

La subrutina f calcula la potencia x_i^p, como $p = 2$ podemos crear una copia especializada de f que calcule x_i^2.

```
1  subroutine f(x, n, p)
2     real x[*]
3     integer n, p
4     do i = 1, n
5        x[i] = x[i]**p
6     end do
7  end subroutine
8
9  subroutine f_2(x, n)
10    real x[*]
11    integer n
12    do i = 1, n
13       x[i] = x[i] * x[i]
14    end do
15 end subroutine
16
17 call f_2(a, n)
```

La subrutina especializada f_2 (línea 9) puede aplicar *strength reduction* para reducir el cálculo de una potencia a una multiplicación, ya que sabe que la potencia calculada va a ser un cuadrado.

C.8.3 Resultados

En esta optimización, el código medido en los experimentos es distinto del que aparece en el ejemplo. Se ha decidido dejar un código más sencillo para poder explicarlo mejor. En el apéndice

E se encuentra el código medido.

Esta optimización actúa reduciendo el número de instrucciones N_{inst}, ya que elimina el paso de parámetros a los procedimientos debido a su especialización. Por contra, puede provocar un aumento del código significativo si replica muchas funciones especializadas. Dicho aumento de código podría suponer más fallos en la *cache* y un aumento del CPI. Por lo tanto, el tiempo de ejecución podrá aumentar o disminuir, según varíen N_{inst} y CPI en la ecuación 3.1.

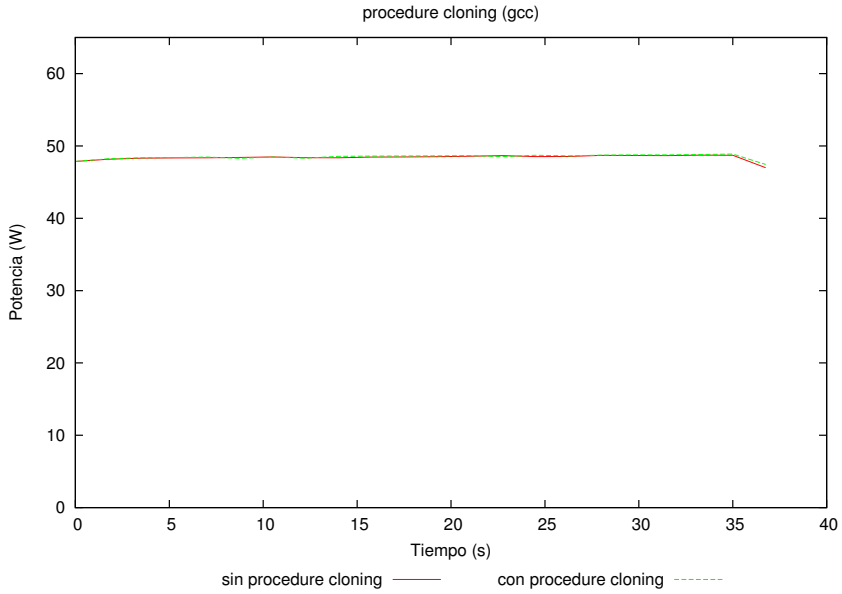

Figura C.15: Potencia usando *procedure cloning*.

En la figura C.15 podemos observar como la potencia disipada en ambas versiones es prácticamente igual. En cuanto al tiempo de ejecución, es un poco mayor en la versión optimizada. Este ligero aumento del tiempo de ejecución puede deberse a que al clonar cada función, se multiplica por dos el número de funciones del programa (ver apéndice E); provocando un aumento de código y más fallos en la *cache* de instrucciones. Integrando con la ecuación 3.7 podemos calcular cómo la energía consumida (ver tabla C.9) es prácticamente igual en las dos versiones, aunque ligeramente mayor en la versión optimizada.

Versión	Energía (Julios)	Energía normalizada
No optimiz.	1863.88	1
Optimizada	1866.88	1

Tabla C.9: Energía usando *procedure cloning*.

En cuanto al comportamiento térmico, es muy similar en las dos versiones tal y como puede apreciarse en la figura C.16. Esto se debe a que no hay apenas diferencia en la disipación de potencia entre versiones.

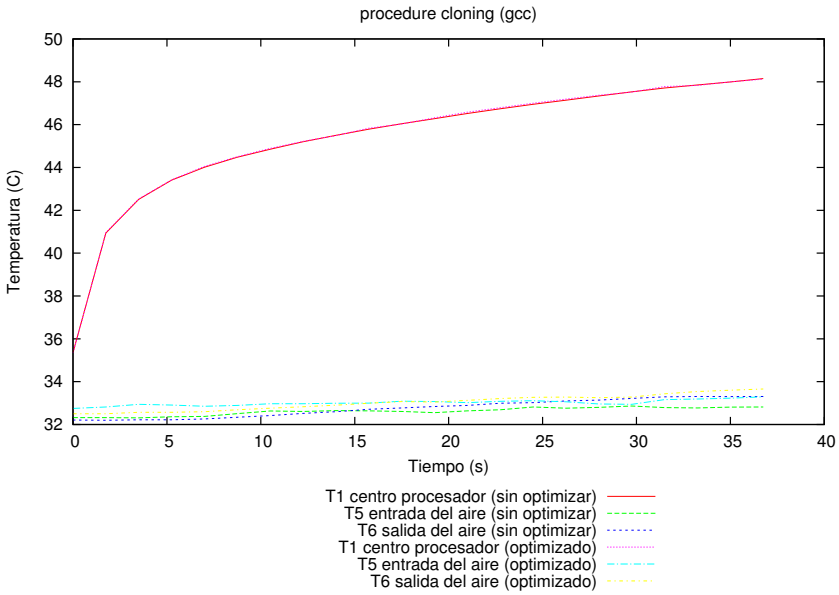

Figura C.16: Temperaturas usando *procedure cloning*.

Apéndice D
Condiciones en las que se han realizado los experimentos

En este apéndice se incluye una tabla con las condiciones del entorno en el que se realizaron los experimentos. En la primera columna aparece el nombre de la optimización, en la segunda el programa medido, en la tercera los parámetros de entrada (si los requiere), en la cuarta la versión del compilador utilizado, en la quinta la temperatura de la CPU en reposo medida en el termopar T1, y en la sexta el rango de la sonda amperimétrica.

La frecuencia de giro del ventilador en los experimentos ha sido la máxima posible permitida por la plataforma: 6750 rpm en todos los casos.

Optimización	Programa	Parámetros	Compilador	Temperatura	Rango
Loop-based strength reduction & induction variable elimination	reduction2.c	-	gcc-4.5.2	32 °C	1 A/V
Loop invariant code motion	motion.c	1048576	gcc-4.5.2	32 °C	1 A/V
Loop unswitching	unswitch.c	2	gcc-4.5.2	32 °C	10 A/V
Loop tiling	block2.c	-	gcc-4.4.3 con librerías PPL y CLooG	33 °C	1 A/V
Loop unrolling	unroll.c	-	gcc-4.5.2	32 °C	1 A/V
Scalar replacement	replacement.c	-	gcc-4.5.2	33 °C	10 A/V
Constant propagation	constant.c	-	gcc-4.5.2	34 °C	1 A/V
Copy propagation	copy.c	-	gcc-4.5.2	32 °C	10 A/V
Unreachable-code elimination & useless-code elimination	elimination.c	-	gcc-4.5.2	35 °C	1 A/V
Procedure inlining	inline.c	-	gcc-4.5.2	35 °C	1 A/V
Procedure cloning	clone.c	-	gcc-4.5.2	35 °C	1 A/V
Tail recursion elimination	recursion.c	-	gcc-4.5.2	35 °C	1 A/V

Tabla D.1: Condiciones de la medición de los experimentos.

Apéndice E
Código utilizado

En este apéndice se incluye el código fuente de los programas medidos. Las condiciones bajo las cuales se han realizado las mediciones se muestran en el apéndice D.

E.1 Loop-based strength reduction & induction variable elimination

Programa: reduction2.c

```
#include <stdio.h>
#include <stdlib.h>

#define N 600000

struct bla
{
        char x[187];
        int y;
        char z[253];
} arr_base[N];

int main(int argc, char *argv[]) {
        int iter, i;

        for (i=0; i<1000; i++) {
                for (iter=0; iter<N; iter++) {
                        arr_base[iter].y = 17 * iter;
                }
        }

}
```

Ensamblador de la versión no optimizada:

```
        .comm   arr_base,268800000,32
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp    #
```

```
        movl    %esp, %ebp      #,
        subl    $16, %esp       #,
        movl    $0, -8(%ebp)    #, i
        jmp     .L2     #
.L5:
        movl    $0, -4(%ebp)    #, iter
        jmp     .L3     #
.L4:
        movl    -4(%ebp), %edx  # iter, tmp60
        movl    %edx, %eax      # tmp60, tmp61
        sall    $4, %eax        #, tmp62
        leal    (%eax,%edx), %ecx       #, D.3162
        movl    -4(%ebp), %eax  # iter, tmp63
        sall    $6, %eax        #, tmp65
        leal    0(,%eax,8), %edx        #, tmp66
        subl    %eax, %edx      # tmp64, tmp66
        leal    arr_base+176(%edx), %eax        #, tmp67
        movl    %ecx, 12(%eax)  # D.3162, arr_base[iter_1].y
        addl    $1, -4(%ebp)    #, iter
.L3:
        cmpl    $599999, -4(%ebp)       #, iter
        jle     .L4     #,
        addl    $1, -8(%ebp)    #, i
.L2:
        cmpl    $999, -8(%ebp)  #, i
        jle     .L5     #,
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.5.2"
        .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        movl    $0, %ecx        #, i
        jmp     .L2     #
.L3:
        movl    %eax, (%edx)    # ivtmp.8, arr_base[iter].y
        addl    $17, %eax       #, ivtmp.8
        addl    $448, %edx      #, ivtmp.9
        cmpl    $10200000, %eax #, ivtmp.8
        jne     .L3     #,
        addl    $1, %ecx        #, i
        cmpl    $1000, %ecx     #, i
        je      .L5     #,
.L2:
        movl    $arr_base+188, %edx     #, ivtmp.9
        movl    $0, %eax        #, ivtmp.8
        jmp     .L3     #
```

```
22  .L5:
23          popl    %ebp    #
24          ret
25          .size   main, .-main
26          .comm   arr_base,268800000,32
27          .ident  "GCC: (GNU) 4.5.2"
28          .section        .note.GNU-stack,"",@progbits
```

E.2 Loop invariant code motion

Programa: motion.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 600000

int main(int argc, char *argv[]) {
        double a[N], x;
        int i, rep;

        x = atoi(argv[1]);

        for (rep=0; rep<5000; rep++) {
                for (i=0; i<N; i++){
                        a[i] = i + sqrt(x);
                }
        }

        srand(1);
        i = rand() % N;

        printf("a[%d] -> %f\n", i, a[i]);
}
```

Ensamblador de la versión no optimizada:

```
        .section        .rodata
.LC0:
        .string "a[%d] -> %f\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        andl    $-16, %esp      #,
        pushl   %ebx    #
        subl    $4800060, %esp  #,
        movl    12(%ebp), %eax  # argv, tmp68
        addl    $4, %eax        #, D.4228
        movl    (%eax), %eax    #* D.4228, D.4229
        movl    %eax, (%esp)    # D.4229,
        call    atoi    #
        movl    %eax, 28(%esp)  # D.4230,
        fildl   28(%esp)        #
        fstpl   4800032(%esp)   # x
        movl    $0, 4800040(%esp)       #, rep
        jmp     .L2     #
.L7:
        movl    $0, 4800044(%esp)       #, i
        jmp     .L3     #
```

```
.L6:
        fildl   4800044(%esp)   # i
        fstpl   16(%esp)        # %sfp
        fldl    4800032(%esp)   # x
        fld     %st(0)  #
        fsqrt
        fucomi  %st(0), %st     #,
        jp      .L8     #,
        fucomi  %st(0), %st     #,
        je      .L9     #,
        fstp    %st(0)  #
        jmp     .L5     #
.L8:
        fstp    %st(0)  #
.L5:
        fstpl   (%esp)  #
        call    sqrt    #
        jmp     .L4     #
.L9:
        fstp    %st(1)  #
.L4:
        faddl   16(%esp)        # %sfp
        movl    4800044(%esp), %eax     # i, tmp73
        fstpl   32(%esp,%eax,8) # a
        addl    $1, 4800044(%esp)       #, i
.L3:
        cmpl    $599999, 4800044(%esp)  #, i
        jle     .L6     #,
        addl    $1, 4800040(%esp)       #, rep
.L2:
        cmpl    $4999, 4800040(%esp)    #, rep
        jle     .L7     #,
        movl    $1, (%esp)      #,
        call    srand   #
        call    rand    #
        movl    %eax, %ecx      #, D.4234
        movl    $1876499845, %edx       #, tmp75
        movl    %ecx, %eax      # D.4234,
        imull   %edx    # tmp75
        sarl    $18, %edx       #, tmp76
        movl    %ecx, %eax      # D.4234, tmp77
        sarl    $31, %eax       #, tmp77
        movl    %edx, %ebx      # tmp76,
        subl    %eax, %ebx      # tmp77,
        movl    %ebx, %eax      #, tmp78
        movl    %eax, 4800044(%esp)     # tmp78, i
        movl    4800044(%esp), %eax     # i, tmp80
        imull   $600000, %eax, %eax     #, tmp80, tmp79
        movl    %ecx, %edx      # D.4234,
        subl    %eax, %edx      # tmp79,
        movl    %edx, %eax      #, tmp81
        movl    %eax, 4800044(%esp)     # tmp81, i
        movl    4800044(%esp), %eax     # i, tmp82
        fldl    32(%esp,%eax,8) # a
        movl    $.LC0, %eax     #, D.4236
        fstpl   8(%esp) #
```

```
82          movl    4800044(%esp), %edx     # i, tmp83
83          movl    %edx, 4(%esp)   # tmp83,
84          movl    %eax, (%esp)    # D.4236,
85          call    printf  #
86          addl    $4800060, %esp  #,
87          popl    %ebx    #
88          movl    %ebp, %esp      #,
89          popl    %ebp    #
90          ret
91          .size   main, .-main
92          .ident  "GCC: (GNU) 4.5.2"
93          .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
1           .section        .rodata.str1.1,"aMS",@progbits,1
2   .LC0:
3           .string "a[%d] -> %f\n"
4           .text
5   .globl main
6           .type   main, @function
7   main:
8           pushl   %ebp    #
9           movl    %esp, %ebp      #,
10          andl    $-16, %esp      #,
11          pushl   %esi    #
12          pushl   %ebx    #
13          subl    $4800056, %esp  #,
14          movl    $0, 12(%esp)    #,
15          movl    $10, 8(%esp)    #,
16          movl    $0, 4(%esp)     #,
17          movl    12(%ebp), %eax  # argv, argv
18          movl    4(%eax), %eax   #, tmp81
19          movl    %eax, (%esp)    # tmp81,
20          call    __strtol_internal       #
21          movl    %eax, 44(%esp)  # D.4421,
22          fildl   44(%esp)        #
23          fstl    24(%esp)        # %sfp
24          movl    $0, %esi        #, rep
25          fsqrt
26          fstpl   32(%esp)        # %sfp
27          jmp     .L2     #
28  .L5:
29          fldl    32(%esp)        # %sfp
30          fucomi  %st(0), %st     #,
31          jnp     .L3     #,
32          fstp    %st(0)  #
33          fldl    24(%esp)        # %sfp
34          fstpl   (%esp)  #
35          call    sqrt    #
36  .L3:
37          movl    %ebx, 44(%esp)  # i,
38          fildl   44(%esp)        #
39          faddp   %st, %st(1)     #,
40          fstpl   48(%esp,%ebx,8) # a
```

```
41          addl    $1, %ebx        #, i
42          cmpl    $600000, %ebx   #, i
43          jne     .L5     #,
44          addl    $1, %esi        #, rep
45          cmpl    $5000, %esi     #, rep
46          je      .L6     #,
47  .L2:
48          movl    $0, %ebx        #, i
49          jmp     .L5     #
50  .L6:
51          movl    $1, (%esp)      #,
52          call    srand   #
53          call    rand    #
54          movl    $600000, %ecx   #, tmp90
55          movl    %eax, %edx      # D.4400, tmp88
56          sarl    $31, %edx       #, tmp88
57          idivl   %ecx    # tmp90
58          fldl    48(%esp,%edx,8) # a
59          fstpl   8(%esp) #
60          movl    %edx, 4(%esp)   # tmp88,
61          movl    $.LC0, (%esp)   #,
62          call    printf  #
63          addl    $4800056, %esp  #,
64          popl    %ebx    #
65          popl    %esi    #
66          movl    %ebp, %esp      #,
67          popl    %ebp    #
68          ret
69          .size   main, .-main
70          .ident  "GCC: (GNU) 4.5.2"
71          .section        .note.GNU-stack,"",@progbits
```

E.3 Loop unswitching

Programa: unswitch.c

```
#include <stdio.h>
#include <stdlib.h>

#define N 60000

int main(int argc, char *argv[]) {
        int a[N], b[N], c[N];
        int i, x, repeticiones;

        if (argc != 2) {
                printf("Uso: ./test valor_x\n");
                exit(0);
        }

        x = atoi(argv[1]);
        printf("x -> %d\n", x);

        srand(x);

        for (repeticiones=0; repeticiones<20000; repeticiones++) {

                for (i=0; i<N; i++) {
                        a[i] = rand() % 10;
                        b[i] = rand() % 10;
                        c[i] = rand() % 10;
                }

                for (i=2; i<N; i++) {
                        a[i] = a[i] + x;
                        if (x < 7) {
                                b[i] = a[i] + c[i];
                        }
                        else {
                                b[i] = a[i-1] + b[i-1];
                        }
                }
        }
        printf("a[%d] -> %d\n", x, a[x]);
        printf("b[%d] -> %d\n", x+1, b[x+1]);
        printf("c[%d] -> %d\n", x+2, c[x+2]);
}
```

Ensamblador de la versión no optimizada:

```
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "Uso: ./test valor_x"
.LC1:
        .string "x -> %d\n"
.LC2:
        .string "a[%d] -> %d\n"
```

```
.LC3:
        .string "b[%d] -> %d\n"
.LC4:
        .string "c[%d] -> %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        andl    $-16, %esp      #,
        pushl   %edi    #
        pushl   %esi    #
        pushl   %ebx    #
        subl    $720036, %esp   #,
        cmpl    $2, 8(%ebp)     #, argc
        je      .L2     #,
        movl    $.LC0, (%esp)   #,
        call    puts    #
        movl    $0, (%esp)      #,
        call    exit    #
.L2:
        movl    $0, 12(%esp)    #,
        movl    $10, 8(%esp)    #,
        movl    $0, 4(%esp)     #,
        movl    12(%ebp), %eax  # argv, argv
        movl    4(%eax), %eax   #, tmp94
        movl    %eax, (%esp)    # tmp94,
        call    __strtol_internal       #
        movl    %eax, %esi      #, D.3291
        movl    %eax, 4(%esp)   # D.3291,
        movl    $.LC1, (%esp)   #,
        call    printf  #
        movl    %esi, (%esp)    # D.3291,
        call    srand   #
        movl    $20000, 28(%esp)        #, %sfp
        movl    $1717986919, %edi       #, tmp146
        jmp     .L3     #
.L4:
        call    rand    #
        movl    %eax, %ecx      #, D.3248
        imull   %edi    # tmp146
        sarl    $2, %edx        #, tmp98
        movl    %ecx, %eax      # D.3248, tmp99
        sarl    $31, %eax       #, tmp99
        subl    %eax, %edx      # tmp99, tmp100
        leal    (%edx,%edx,4), %eax     #, tmp104
        addl    %eax, %eax      # tmp105
        subl    %eax, %ecx      # tmp105, tmp106
        movl    %ecx, 480032(%esp,%ebx,4)       # tmp106, a
        call    rand    #
        movl    %eax, %ecx      #, D.3250
        imull   %edi    # tmp146
        sarl    $2, %edx        #, tmp110
        movl    %ecx, %eax      # D.3250, tmp111
        sarl    $31, %eax       #, tmp111
```

```
64          subl    %eax, %edx      # tmp111, tmp112
65          leal    (%edx,%edx,4), %eax     #, tmp116
66          addl    %eax, %eax      # tmp117
67          subl    %eax, %ecx      # tmp117, tmp118
68          movl    %ecx, 240032(%esp,%ebx,4)       # tmp118, b
69          call    rand    #
70          movl    %eax, %ecx      #, D.3252
71          imull   %edi    # tmp146
72          sarl    $2, %edx        #, tmp122
73          movl    %ecx, %eax      # D.3252, tmp123
74          sarl    $31, %eax       #, tmp123
75          subl    %eax, %edx      # tmp123, tmp124
76          leal    (%edx,%edx,4), %eax     #, tmp128
77          addl    %eax, %eax      # tmp129
78          subl    %eax, %ecx      # tmp129, tmp130
79          movl    %ecx, 32(%esp,%ebx,4)   # tmp130, c
80          addl    $1, %ebx        #, i
81          cmpl    $60000, %ebx    #, i
82          jne     .L4     #,
83          movl    $2, %eax        #, i
84  .L7:
85          movl    %esi, %edx      # D.3291, D.3255
86          addl    480032(%esp,%eax,4), %edx       # a, D.3255
87          movl    %edx, 480032(%esp,%eax,4)       # D.3255, a
88          cmpl    $6, %esi        #, D.3291
89          jg      .L5     #,
90          addl    32(%esp,%eax,4), %edx   # c, tmp135
91          movl    %edx, 240032(%esp,%eax,4)       # tmp135, b
92          jmp     .L6     #
93  .L5:
94          movl    240028(%esp,%eax,4), %edx       # b, tmp141
95          addl    480028(%esp,%eax,4), %edx       # a, tmp140
96          movl    %edx, 240032(%esp,%eax,4)       # tmp140, b
97  .L6:
98          addl    $1, %eax        #, i
99          cmpl    $60000, %eax    #, i
100         jne     .L7     #,
101         subl    $1, 28(%esp)    #, %sfp
102         je      .L8     #,
103 .L3:
104         movl    $0, %ebx        #, i
105         jmp     .L4     #
106 .L8:
107         movl    480032(%esp,%esi,4), %eax       # a, tmp142
108         movl    %eax, 8(%esp)   # tmp142,
109         movl    %esi, 4(%esp)   # D.3291,
110         movl    $.LC2, (%esp)   #,
111         call    printf  #
112         leal    1(%esi), %eax   #, D.3267
113         movl    240032(%esp,%eax,4), %edx       # b, tmp143
114         movl    %edx, 8(%esp)   # tmp143,
115         movl    %eax, 4(%esp)   # D.3267,
116         movl    $.LC3, (%esp)   #,
117         call    printf  #
118         addl    $2, %esi        #, D.3270
119         movl    32(%esp,%esi,4), %eax   # c, tmp144
```

```
120             movl    %eax, 8(%esp)   # tmp144,
121             movl    %esi, 4(%esp)   # D.3270,
122             movl    $.LC4, (%esp)   #,
123             call    printf  #
124             addl    $720036, %esp   #,
125             popl    %ebx    #
126             popl    %esi    #
127             popl    %edi    #
128             movl    %ebp, %esp      #,
129             popl    %ebp    #
130             ret
131             .size   main, .-main
132             .ident  "GCC: (GNU) 4.5.2"
133             .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
1               .section        .rodata.str1.1,"aMS",@progbits,1
2       .LC0:
3               .string "Uso: ./test valor_x"
4       .LC1:
5               .string "x -> %d\n"
6       .LC2:
7               .string "a[%d] -> %d\n"
8       .LC3:
9               .string "b[%d] -> %d\n"
10      .LC4:
11              .string "c[%d] -> %d\n"
12              .text
13      .globl main
14              .type   main, @function
15      main:
16              pushl   %ebp    #
17              movl    %esp, %ebp      #,
18              andl    $-16, %esp      #,
19              pushl   %edi    #
20              pushl   %esi    #
21              pushl   %ebx    #
22              subl    $720036, %esp   #,
23              cmpl    $2, 8(%ebp)     #, argc
24              je      .L2     #,
25              movl    $.LC0, (%esp)   #,
26              call    puts    #
27              movl    $0, (%esp)      #,
28              call    exit    #
29      .L2:
30              movl    $0, 12(%esp)    #,
31              movl    $10, 8(%esp)    #,
32              movl    $0, 4(%esp)     #,
33              movl    12(%ebp), %eax  # argv, argv
34              movl    4(%eax), %eax   #, tmp98
35              movl    %eax, (%esp)    # tmp98,
36              call    __strtol_internal       #
37              movl    %eax, %edi      #, D.3291
38              movl    %eax, 4(%esp)   # D.3291,
```

```
39          movl    $.LC1, (%esp)   #,
40          call    printf  #
41          movl    %edi, (%esp)    # D.3291,
42          call    srand   #
43          movl    $20000, 28(%esp)        #, %sfp
44          movl    $1717986919, %esi       #, tmp153
45          jmp     .L3     #
46  .L4:
47          call    rand    #
48          movl    %eax, %ecx      #, D.3248
49          imull   %esi    # tmp153
50          sarl    $2, %edx        #, tmp102
51          movl    %ecx, %eax      # D.3248, tmp103
52          sarl    $31, %eax       #, tmp103
53          subl    %eax, %edx      # tmp103, tmp104
54          leal    (%edx,%edx,4), %eax     #, tmp108
55          addl    %eax, %eax      # tmp109
56          subl    %eax, %ecx      # tmp109, tmp110
57          movl    %ecx, 480032(%esp,%ebx,4)       # tmp110, a
58          call    rand    #
59          movl    %eax, %ecx      #, D.3250
60          imull   %esi    # tmp153
61          sarl    $2, %edx        #, tmp114
62          movl    %ecx, %eax      # D.3250, tmp115
63          sarl    $31, %eax       #, tmp115
64          subl    %eax, %edx      # tmp115, tmp116
65          leal    (%edx,%edx,4), %eax     #, tmp120
66          addl    %eax, %eax      # tmp121
67          subl    %eax, %ecx      # tmp121, tmp122
68          movl    %ecx, 240032(%esp,%ebx,4)       # tmp122, b
69          call    rand    #
70          movl    %eax, %ecx      #, D.3252
71          imull   %esi    # tmp153
72          sarl    $2, %edx        #, tmp126
73          movl    %ecx, %eax      # D.3252, tmp127
74          sarl    $31, %eax       #, tmp127
75          subl    %eax, %edx      # tmp127, tmp128
76          leal    (%edx,%edx,4), %eax     #, tmp132
77          addl    %eax, %eax      # tmp133
78          subl    %eax, %ecx      # tmp133, tmp134
79          movl    %ecx, 32(%esp,%ebx,4)   # tmp134, c
80          addl    $1, %ebx        #, i
81          cmpl    $60000, %ebx    #, i
82          jne     .L4     #,
83          movl    $2, %eax        #, i
84          cmpl    $6, %edi        #, D.3291
85          jle     .L10    #,
86  .L6:
87          addl    %edi, 480032(%esp,%eax,4)       # D.3291, a
88          movl    240028(%esp,%eax,4), %edx       # b, tmp143
89          addl    480028(%esp,%eax,4), %edx       # a, tmp142
90          movl    %edx, 240032(%esp,%eax,4)       # tmp142, b
91          addl    $1, %eax        #, i
92          cmpl    $60000, %eax    #, i
93          jne     .L6     #,
94          jmp     .L7     #
```

```
.L10:
        movl    %edi, %edx      # D.3291, D.3255
        addl    480032(%esp,%eax,4), %edx       # a, D.3255
        movl    %edx, 480032(%esp,%eax,4)       # D.3255, a
        addl    32(%esp,%eax,4), %edx   # c, tmp148
        movl    %edx, 240032(%esp,%eax,4)       # tmp148, b
        addl    $1, %eax        #, i
        cmpl    $60000, %eax    #, i
        jne     .L10    #,
.L7:
        subl    $1, 28(%esp)    #, %sfp
        je      .L8     #,
.L3:
        movl    $0, %ebx        #, i
        jmp     .L4     #
.L8:
        movl    480032(%esp,%edi,4), %eax       # a, tmp149
        movl    %eax, 8(%esp)   # tmp149,
        movl    %edi, 4(%esp)   # D.3291,
        movl    $.LC2, (%esp)   #,
        call    printf  #
        leal    1(%edi), %eax   #, D.3267
        movl    240032(%esp,%eax,4), %edx       # b, tmp150
        movl    %edx, 8(%esp)   # tmp150,
        movl    %eax, 4(%esp)   # D.3267,
        movl    $.LC3, (%esp)   #,
        call    printf  #
        addl    $2, %edi        #, D.3270
        movl    32(%esp,%edi,4), %eax   # c, tmp151
        movl    %eax, 8(%esp)   # tmp151,
        movl    %edi, 4(%esp)   # D.3270,
        movl    $.LC4, (%esp)   #,
        call    printf  #
        addl    $720036, %esp   #,
        popl    %ebx    #
        popl    %esi    #
        popl    %edi    #
        movl    %ebp, %esp      #,
        popl    %ebp    #
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.5.2"
        .section        .note.GNU-stack,"",@progbits
```

E.4 Loop tiling

Programa: block2.c

```
#include <stdio.h>
#include <stdlib.h>

#define N 4000
#define NUM_TESTS 50

int a[N][N], b[N][N];

int foo()
{

   int i, j;

   for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
         a[i][j] = b[j][i] + 1;
      }
   }
}

int main()
{
   int i, j, test;

   srand(1);

   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
         b[i][j] = rand() % 100;
      }
   }

   for (test = 0; test < NUM_TESTS; test++) {
      foo();
   }

   i = rand() % N;
   j = rand() % N;

   printf("a[%d][%d] -> %d\n", i, j, a[i][j]);
}
```

Ensamblador de la versión no optimizada:

```
        .file   "block2.c"
        .text
.globl foo
        .type   foo, @function
foo:
        pushl   %ebp
        movl    %esp, %ebp
```

```
8           pushl   %edi
9           pushl   %esi
10          pushl   %ebx
11          movl    $0, %esi
12          movl    $0, %edi
13          jmp     .L2
14  .L3:
15          movl    (%ecx), %ebx
16          addl    $1, %ebx
17          movl    %ebx, (%edx)
18          addl    $1, %eax
19          addl    $16000, %ecx
20          addl    $4, %edx
21          cmpl    $4000, %eax
22          jne     .L3
23          addl    $1, %esi
24          cmpl    $4000, %esi
25          je      .L6
26  .L2:
27          leal    b(,%esi,4), %ecx
28          imull   $16000, %esi, %edx
29          addl    $a, %edx
30          movl    %edi, %eax
31          jmp     .L3
32  .L6:
33          popl    %ebx
34          popl    %esi
35          popl    %edi
36          popl    %ebp
37          ret
38          .size   foo, .-foo
39          .section        .rodata.str1.1,"aMS",@progbits,1
40  .LC0:
41          .string "a[%d][%d] -> %d\n"
42          .text
43  .globl main
44          .type   main, @function
45  main:
46          pushl   %ebp
47          movl    %esp, %ebp
48          andl    $-16, %esp
49          pushl   %edi
50          pushl   %esi
51          pushl   %ebx
52          subl    $52, %esp
53          movl    $1, (%esp)
54          call    srand
55          movl    $0, 44(%esp)
56          movl    $1374389535, %edi
57          jmp     .L9
58  .L10:
59          call    rand
60          movl    %eax, %ecx
61          imull   %edi
62          sarl    $5, %edx
63          movl    %ecx, %eax
```

```
64          sarl    $31, %eax
65          subl    %eax, %edx
66          imull   $100, %edx, %edx
67          subl    %edx, %ecx
68          movl    %ecx, (%ebx)
69          addl    $1, %esi
70          addl    $4, %ebx
71          cmpl    $4000, %esi
72          jne     .L10
73          addl    $1, 44(%esp)
74          cmpl    $4000, 44(%esp)
75          je      .L11
76  .L9:
77          imull   $16000, 44(%esp), %ebx
78          addl    $b, %ebx
79          movl    $0, %esi
80          jmp     .L10
81  .L11:
82          movl    $0, %ebx
83  .L12:
84          call    foo
85          addl    $1, %ebx
86          cmpl    $50, %ebx
87          jne     .L12
88          call    rand
89          movw    $4000, %bx
90          movl    %eax, %edx
91          sarl    $31, %edx
92          idivl   %ebx
93          movl    %edx, %esi
94          call    rand
95          movl    %eax, %edx
96          sarl    $31, %edx
97          idivl   %ebx
98          imull   $4000, %esi, %eax
99          addl    %edx, %eax
100         movl    a(,%eax,4), %eax
101         movl    %eax, 16(%esp)
102         movl    %edx, 12(%esp)
103         movl    %esi, 8(%esp)
104         movl    $.LC0, 4(%esp)
105         movl    $1, (%esp)
106         call    __printf_chk
107         addl    $52, %esp
108         popl    %ebx
109         popl    %esi
110         popl    %edi
111         movl    %ebp, %esp
112         popl    %ebp
113         ret
114         .size   main, .-main
115         .comm   a,64000000,32
116         .comm   b,64000000,32
117         .ident  "GCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3"
118         .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
        .file   "block2.c"
        .text
.globl foo
        .type   foo, @function
foo:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        subl    $32, %esp
        movl    $50, %edx
        movl    $3999, -40(%ebp)
.L9:
        movl    $0, %eax
        leal    -50(%edx), %ecx
        movl    %ecx, -32(%ebp)
        movl    %edx, -36(%ebp)
.L8:
        movl    -32(%ebp), %ecx
        imull   $4000, %eax, %ebx
        movl    %ebx, -24(%ebp)
        leal    50(%eax), %ebx
        movl    %ebx, -20(%ebp)
        movl    %ebx, -28(%ebp)
.L7:
        movl    -36(%ebp), %ebx
        cmpl    $3999, %edx
        jbe     .L3
        movl    $3999, %ebx
.L3:
        cmpl    %ecx, %ebx
        jb      .L2
        movl    -24(%ebp), %ebx
        addl    %ecx, %ebx
        leal    b(,%ebx,4), %edi
        imull   $4000, %ecx, %ebx
        addl    %eax, %ebx
        leal    a(,%ebx,4), %esi
        movl    %eax, %ebx
        movl    %eax, -44(%ebp)
.L6:
        movl    -28(%ebp), %eax
        movl    %eax, -16(%ebp)
        cmpl    $3999, -20(%ebp)
        jbe     .L5
        movl    -40(%ebp), %eax
        movl    %eax, -16(%ebp)
.L5:
        cmpl    -16(%ebp), %ebx
        ja      .L4
        movl    (%edi), %eax
        addl    $1, %eax
```

```
54          movl    %eax, (%esi)
55          addl    $1, %ebx
56          addl    $16000, %edi
57          addl    $4, %esi
58          jmp     .L6
59  .L4:
60          movl    -44(%ebp), %eax
61          addl    $1, %ecx
62          jmp     .L7
63  .L2:
64          addl    $51, %eax
65          cmpl    $4029, %eax
66          jne     .L8
67          addl    $51, %edx
68          cmpl    $4079, %edx
69          jne     .L9
70          addl    $32, %esp
71          popl    %ebx
72          popl    %esi
73          popl    %edi
74          popl    %ebp
75          ret
76          .size   foo, .-foo
77          .section        .rodata.str1.1,"aMS",@progbits,1
78  .LC0:
79          .string "a[%d][%d] -> %d\n"
80          .text
81  .globl main
82          .type   main, @function
83  main:
84          pushl   %ebp
85          movl    %esp, %ebp
86          andl    $-16, %esp
87          pushl   %edi
88          pushl   %esi
89          pushl   %ebx
90          subl    $52, %esp
91          movl    $1, (%esp)
92          call    srand
93          movl    $0, 44(%esp)
94          movl    $1374389535, %edi
95          jmp     .L14
96  .L15:
97          call    rand
98          movl    %eax, %ecx
99          imull   %edi
100         sarl    $5, %edx
101         movl    %ecx, %eax
102         sarl    $31, %eax
103         subl    %eax, %edx
104         imull   $100, %edx, %edx
105         subl    %edx, %ecx
106         movl    %ecx, (%ebx)
107         addl    $1, %esi
108         addl    $4, %ebx
109         cmpl    $4000, %esi
```

```
110             jne     .L15
111             addl    $1, 44(%esp)
112             cmpl    $4000, 44(%esp)
113             je      .L16
114     .L14:
115             imull   $16000, 44(%esp), %ebx
116             addl    $b, %ebx
117             movl    $0, %esi
118             jmp     .L15
119     .L16:
120             movl    $0, %ebx
121     .L17:
122             call    foo
123             addl    $1, %ebx
124             cmpl    $50, %ebx
125             jne     .L17
126             call    rand
127             movw    $4000, %bx
128             movl    %eax, %edx
129             sarl    $31, %edx
130             idivl   %ebx
131             movl    %edx, %esi
132             call    rand
133             movl    %eax, %edx
134             sarl    $31, %edx
135             idivl   %ebx
136             imull   $4000, %esi, %eax
137             addl    %edx, %eax
138             movl    a(,%eax,4), %eax
139             movl    %eax, 16(%esp)
140             movl    %edx, 12(%esp)
141             movl    %esi, 8(%esp)
142             movl    $.LC0, 4(%esp)
143             movl    $1, (%esp)
144             call    __printf_chk
145             addl    $52, %esp
146             popl    %ebx
147             popl    %esi
148             popl    %edi
149             movl    %ebp, %esp
150             popl    %ebp
151             ret
152             .size   main, .-main
153             .comm   a,64000000,32
154             .comm   b,64000000,32
155             .ident  "GCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3"
156             .section        .note.GNU-stack,"",@progbits
```

E.5 Loop unrolling

Programa: unroll.c

```
#include <stdio.h>
#include <stdlib.h>

#define N 1000000
#define REPEAT 5000

int main()
{
        int i, iter, a[N];

        srand(7);

        for(i=0; i<N; i++)
                a[i] = rand() % 5;

        for (iter=0; iter<REPEAT; iter++)
                for(i=0; i<N; i++)
                        a[i] = a[i] + a[i-1] - a[i-2];

        i = rand() % N;
        printf("a[%d] -> %d\n", i, a[i]);
}
```

Ensamblador de la versión no optimizada:

```
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "a[%d] -> %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        andl    $-16, %esp      #,
        pushl   %esi    #
        pushl   %ebx    #
        subl    $4000024, %esp  #,
        movl    $7, (%esp)      #,
        call    srand   #
        movl    $0, %ebx        #, i
        movl    $1717986919, %esi       #, tmp102
.L2:
        call    rand    #
        movl    %eax, %ecx      #, D.3237
        imull   %esi    # tmp102
        sarl    %edx    # tmp84
        movl    %ecx, %eax      # D.3237, tmp85
        sarl    $31, %eax       #, tmp85
        subl    %eax, %edx      # tmp85, tmp86
        leal    (%edx,%edx,4), %eax     #, tmp90
```

```
27          subl    %eax, %ecx      # tmp90, tmp91
28          movl    %ecx, 16(%esp,%ebx,4)   # tmp91, a
29          addl    $1, %ebx        #, i
30          cmpl    $1000000, %ebx  #, i
31          jne     .L2     #,
32          movl    $0, %ebx        #, iter
33          leal    4000016(%esp), %ecx     #, D.3270
34          jmp     .L3     #
35  .L4:
36          movl    -4(%eax), %edx  # a, tmp94
37          addl    (%eax), %edx    # a, tmp92
38          subl    -8(%eax), %edx  # a, tmp95
39          movl    %edx, (%eax)    # tmp95, a
40          addl    $4, %eax        #, ivtmp.9
41          cmpl    %ecx, %eax      # D.3270, ivtmp.9
42          jne     .L4     #,
43          addl    $1, %ebx        #, iter
44          cmpl    $5000, %ebx     #, iter
45          je      .L5     #,
46  .L3:
47          leal    16(%esp), %eax  #, ivtmp.9
48          jmp     .L4     #
49  .L5:
50          call    rand    #
51          movl    $1000000, %ecx  #, tmp99
52          movl    %eax, %edx      # D.3246, tmp97
53          sarl    $31, %edx       #, tmp97
54          idivl   %ecx    # tmp99
55          movl    16(%esp,%edx,4), %eax   # a, tmp100
56          movl    %eax, 8(%esp)   # tmp100,
57          movl    %edx, 4(%esp)   # tmp97,
58          movl    $.LC0, (%esp)   #,
59          call    printf  #
60          addl    $4000024, %esp  #,
61          popl    %ebx    #
62          popl    %esi    #
63          movl    %ebp, %esp      #,
64          popl    %ebp    #
65          ret
66          .size   main, .-main
67          .ident  "GCC: (GNU) 4.5.2"
68          .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
1           .section        .rodata.str1.1,"aMS",@progbits,1
2   .LC0:
3           .string "a[%d] -> %d\n"
4           .text
5   .globl main
6           .type   main, @function
7   main:
8           pushl   %ebp    #
9           movl    %esp, %ebp      #,
10          andl    $-16, %esp      #,
```

```
        pushl   %edi    #
        pushl   %esi    #
        pushl   %ebx    #
        subl    $4000020, %esp  #,
        movl    $7, (%esp)      #,
        call    srand   #
        movl    $0, %ebx        #, i
        movl    $1717986919, %esi       #, tmp102
.L2:
        call    rand    #
        movl    %eax, %ecx      #, D.3237
        imull   %esi    # tmp102
        sarl    %edx    # tmp84
        movl    %ecx, %eax      # D.3237, tmp85
        sarl    $31, %eax       #, tmp85
        subl    %eax, %edx      # tmp85, tmp86
        leal    (%edx,%edx,4), %edi     #, tmp90
        subl    %edi, %ecx      # tmp90, tmp91
        movl    %ecx, 16(%esp,%ebx,4)   # tmp91, a
        addl    $1, %ebx        #, tmp108
        call    rand    #
        movl    %eax, %ecx      #, D.3237
        imull   %esi    # tmp102
        sarl    %edx    # tmp114
        movl    %ecx, %eax      # D.3237, tmp115
        sarl    $31, %eax       #, tmp115
        subl    %eax, %edx      # tmp115, tmp116
        leal    (%edx,%edx,4), %edi     #, tmp118
        subl    %edi, %ecx      # tmp118, tmp119
        movl    %ecx, 16(%esp,%ebx,4)   # tmp119, a
        leal    1(%ebx), %edi   #, i
        call    rand    #
        movl    %eax, %ecx      #, D.3237
        imull   %esi    # tmp102
        sarl    %edx    # tmp125
        movl    %ecx, %eax      # D.3237, tmp126
        sarl    $31, %eax       #, tmp126
        subl    %eax, %edx      # tmp126, tmp127
        leal    (%edx,%edx,4), %edx     #, tmp129
        subl    %edx, %ecx      # tmp129, tmp130
        movl    %ecx, 16(%esp,%edi,4)   # tmp130, a
        leal    2(%ebx), %edi   #, i
        call    rand    #
        movl    %eax, %ecx      #, D.3237
        imull   %esi    # tmp102
        sarl    %edx    # tmp136
        movl    %ecx, %eax      # D.3237, tmp137
        sarl    $31, %eax       #, tmp137
        subl    %eax, %edx      # tmp137, tmp138
        leal    (%edx,%edx,4), %edx     #, tmp140
        subl    %edx, %ecx      # tmp140, tmp141
        movl    %ecx, 16(%esp,%edi,4)   # tmp141, a
        leal    3(%ebx), %edi   #, i
        call    rand    #
        movl    %eax, %ecx      #, D.3237
        imull   %esi    # tmp102
```

```
67          sarl    %edx    # tmp147
68          movl    %ecx, %eax      # D.3237, tmp148
69          sarl    $31, %eax       #, tmp148
70          subl    %eax, %edx      # tmp148, tmp149
71          leal    (%edx,%edx,4), %eax     #, tmp151
72          subl    %eax, %ecx      # tmp151, tmp152
73          movl    %ecx, 16(%esp,%edi,4)   # tmp152, a
74          addl    $4, %ebx        #, i
75          cmpl    $1000000, %ebx  #, i
76          jne     .L2     #,
77          movl    $0, %ebx        #, iter
78          leal    4000016(%esp), %ecx     #, D.3270
79          jmp     .L3     #
80  .L4:
81          movl    -4(%edx), %esi  # a, tmp94
82          addl    (%edx), %esi    # a, tmp92
83          subl    -8(%edx), %esi  # a, tmp95
84          movl    %esi, (%edx)    # tmp95, a
85          leal    4(%edx), %edi   #, tmp107
86          addl    4(%edx), %esi   # a, tmp156
87          subl    -8(%edi), %esi  # a, tmp157
88          movl    %esi, 4(%edx)   # tmp157, a
89          leal    4(%edi), %edx   #, ivtmp.9
90          movl    -4(%edx), %eax  # a, tmp160
91          addl    4(%edi), %eax   # a, tmp161
92          subl    -8(%edx), %eax  # a, tmp161
93          movl    %eax, 4(%edi)   # tmp162, a
94          leal    8(%edi), %edx   #, ivtmp.9
95          movl    -4(%edx), %eax  # a, tmp165
96          addl    8(%edi), %eax   # a, tmp166
97          subl    -8(%edx), %eax  # a, tmp166
98          movl    %eax, 8(%edi)   # tmp167, a
99          leal    12(%edi), %edx  #, ivtmp.9
100         movl    -4(%edx), %eax  # a, tmp170
101         addl    12(%edi), %eax  # a, tmp171
102         subl    -8(%edx), %eax  # a, tmp171
103         movl    %eax, 12(%edi)  # tmp172, a
104         leal    16(%edi), %edx  #, ivtmp.9
105         movl    -4(%edx), %eax  # a, tmp175
106         addl    16(%edi), %eax  # a, tmp176
107         subl    -8(%edx), %eax  # a, tmp176
108         movl    %eax, 16(%edi)  # tmp177, a
109         leal    20(%edi), %edx  #, ivtmp.9
110         movl    -4(%edx), %eax  # a, tmp180
111         addl    20(%edi), %eax  # a, tmp181
112         subl    -8(%edx), %eax  # a, tmp181
113         movl    %eax, 20(%edi)  # tmp182, a
114         leal    24(%edi), %edx  #, ivtmp.9
115         movl    -4(%edx), %eax  # a, tmp185
116         addl    24(%edi), %eax  # a, tmp186
117         subl    -8(%edx), %eax  # a, tmp186
118         movl    %eax, 24(%edi)  # tmp187, a
119         leal    28(%edi), %edx  #, ivtmp.9
120         cmpl    %ecx, %edx      # D.3270, ivtmp.9
121         jne     .L4     #,
122 .L41:
```

```
123           addl    $1, %ebx        #, iter
124           cmpl    $5000, %ebx     #, iter
125           je      .L5     #,
126   .L3:
127           leal    16(%esp), %esi  #, ivtmp.9
128           movl    %ecx, %eax      # D.3270, tmp104
129           subl    %esi, %eax      # ivtmp.9, tmp104
130           subl    $4, %eax        #, tmp105
131           shrl    $2, %eax        #, tmp103
132           andl    $7, %eax        #, tmp106
133           movl    12(%esp), %edi  # a, tmp190
134           addl    16(%esp), %edi  # a, tmp191
135           subl    8(%esp), %edi   # a, tmp192
136           movl    %edi, 16(%esp)  # tmp192, a
137           leal    4(%esi), %edx   #, ivtmp.9
138           cmpl    %ecx, %edx      # D.3270, ivtmp.9
139           jne     .L34    #,
140           jmp     .L41    #
141   .L5:
142           call    rand    #
143           movl    $1000000, %ecx  #, tmp99
144           movl    %eax, %edx      # D.3246, tmp97
145           sarl    $31, %edx       #, tmp97
146           idivl   %ecx    # tmp99
147           movl    16(%esp,%edx,4), %ebx   # a, tmp100
148           movl    %ebx, 8(%esp)   # tmp100,
149           movl    %edx, 4(%esp)   # tmp97,
150           movl    $.LC0, (%esp)   #,
151           call    printf  #
152           movl    $0, %eax        #,
153           addl    $4000020, %esp  #,
154           popl    %ebx    #
155           popl    %esi    #
156           popl    %edi    #
157           movl    %ebp, %esp      #,
158           popl    %ebp    #
159           ret
160   .L34:
161           testl   %eax, %eax      # tmp106
162           je      .L4     #,
163           cmpl    $1, %eax        #, tmp106
164           je      .L35    #,
165           cmpl    $2, %eax        #, tmp106
166           .p2align 4,,3
167           je      .L36    #,
168           cmpl    $3, %eax        #, tmp106
169           .p2align 4,,2
170           je      .L37    #,
171           cmpl    $4, %eax        #, tmp106
172           .p2align 4,,2
173           je      .L38    #,
174           cmpl    $5, %eax        #, tmp106
175           .p2align 4,,2
176           je      .L39    #,
177           cmpl    $6, %eax        #, tmp106
178           .p2align 4,,2
```

```
179             je      .L40    #,
180             movl    -4(%edx), %eax  # a, tmp195
181             addl    4(%esi), %eax   # a, tmp196
182             subl    -8(%edx), %eax  # a, tmp197
183             movl    %eax, 4(%esi)   # tmp197, a
184             addl    $4, %edx        #, ivtmp.9
185     .L40:
186             movl    -4(%edx), %esi  # a, tmp199
187             addl    (%edx), %esi    # a, tmp200
188             subl    -8(%edx), %esi  # a, tmp201
189             movl    %esi, (%edx)    # tmp201, a
190             addl    $4, %edx        #, ivtmp.9
191     .L39:
192             movl    -4(%edx), %edi  # a, tmp203
193             addl    (%edx), %edi    # a, tmp204
194             subl    -8(%edx), %edi  # a, tmp205
195             movl    %edi, (%edx)    # tmp205, a
196             addl    $4, %edx        #, ivtmp.9
197     .L38:
198             movl    -4(%edx), %eax  # a, tmp207
199             addl    (%edx), %eax    # a, tmp208
200             subl    -8(%edx), %eax  # a, tmp209
201             movl    %eax, (%edx)    # tmp209, a
202             addl    $4, %edx        #, ivtmp.9
203     .L37:
204             movl    -4(%edx), %esi  # a, tmp211
205             addl    (%edx), %esi    # a, tmp212
206             subl    -8(%edx), %esi  # a, tmp213
207             movl    %esi, (%edx)    # tmp213, a
208             addl    $4, %edx        #, ivtmp.9
209     .L36:
210             movl    -4(%edx), %edi  # a, tmp215
211             addl    (%edx), %edi    # a, tmp216
212             subl    -8(%edx), %edi  # a, tmp217
213             movl    %edi, (%edx)    # tmp217, a
214             addl    $4, %edx        #, ivtmp.9
215     .L35:
216             movl    -4(%edx), %eax  # a, tmp219
217             addl    (%edx), %eax    # a, tmp220
218             subl    -8(%edx), %eax  # a, tmp221
219             movl    %eax, (%edx)    # tmp221, a
220             addl    $4, %edx        #, ivtmp.9
221             cmpl    %ecx, %edx      # D.3270, ivtmp.9
222             jne     .L4     #,
223             jmp     .L41    #
224             .size   main, .-main
225             .ident  "GCC: (GNU) 4.5.2"
226             .section        .note.GNU-stack,"",@progbits
```

E.6 Scalar replacement

Programa: replacement.c

```
#include <stdio.h>
#include <stdlib.h>

struct bovid
{
        float red;
        int green;
        void *blue;
};

static int
__attribute__((noinline))
ox (struct bovid cow, int z, struct bovid calf, long l)
{
        if ((rand() % 10000000) == 0) {
                printf("cow.red, l -> %f, %d\n", cow.red, l);
        }
        return 0;
}

static int
__attribute__((noinline))
ox2 (struct bovid cow, int z, struct bovid calf, long l)
{
        if ((rand() % 50000000) == 0) {
                printf("cow.red, l -> %f, %d\n", cow.red, l);
        }
        return 0;
}

static int
__attribute__((noinline))
ox3 (struct bovid cow, int z, struct bovid calf, long l)
{
        if ((rand() % 100000000) == 0) {
                printf("cow.red, l -> %f, %d\n", cow.red, l);
        }
        return 0;
}


int main()
{

        srand(0);

        struct bovid cow, calf;
        int i;

        cow.red = 7.4;
        cow.green = 6;
```

```
52          cow.blue = &cow;
53
54          calf.red = 8.4;
55          calf.green = 5;
56          calf.blue = &cow;
57
58          for (i=0; i < 500000000; i++) {
59                  ox (cow, 4, calf, 2);
60                  ox2 (calf, 4, cow, 5);
61                  ox3 (cow, 4, calf, 3);
62          }
63  }
```

Ensamblador de la versión no optimizada:

```
1           .section        .rodata.str1.1,"aMS",@progbits,1
2   .LC0:
3           .string "cow.red, 1 -> %f, %d\n"
4           .text
5           .type   ox, @function
6   ox:
7           pushl   %ebp    #
8           movl    %esp, %ebp      #,
9           subl    $40, %esp       #,
10          movl    %eax, -20(%ebp) #, cow
11          movl    %edx, -16(%ebp) #, cow
12          movl    %ecx, -12(%ebp) #, cow
13          call    rand    #
14          movl    %eax, %ecx      #, D.3253
15          movl    $1801439851, %edx       #, tmp67
16          imull   %edx    # tmp67
17          sarl    $22, %edx       #, tmp68
18          movl    %ecx, %eax      # D.3253, tmp69
19          sarl    $31, %eax       #, tmp69
20          subl    %eax, %edx      # tmp69, tmp65
21          imull   $10000000, %edx, %edx   #, tmp65, tmp70
22          cmpl    %edx, %ecx      # tmp70, D.3253
23          jne     .L2     #,
24          movl    24(%ebp), %eax  # l, l
25          movl    %eax, 12(%esp)  # l,
26          flds    -20(%ebp)       # cow.red
27          fstpl   4(%esp) #
28          movl    $.LC0, (%esp)   #,
29          call    printf  #
30  .L2:
31          movl    $0, %eax        #,
32          leave
33          ret
34          .size   ox, .-ox
35          .type   ox2, @function
36  ox2:
37          pushl   %ebp    #
38          movl    %esp, %ebp      #,
39          subl    $40, %esp       #,
40          movl    %eax, -20(%ebp) #, cow
```

```
41          movl    %edx, -16(%ebp) #, cow
42          movl    %ecx, -12(%ebp) #, cow
43          call    rand    #
44          movl    %eax, %ecx      #, D.3262
45          movl    $1441151881, %edx       #, tmp67
46          imull   %edx    # tmp67
47          sarl    $24, %edx       #, tmp68
48          movl    %ecx, %eax      # D.3262, tmp69
49          sarl    $31, %eax       #, tmp69
50          subl    %eax, %edx      # tmp69, tmp65
51          imull   $50000000, %edx, %edx   #, tmp65, tmp70
52          cmpl    %edx, %ecx      # tmp70, D.3262
53          jne     .L4     #,
54          movl    24(%ebp), %eax  # l, l
55          movl    %eax, 12(%esp)  # l,
56          flds    -20(%ebp)       # cow.red
57          fstpl   4(%esp) #
58          movl    $.LC0, (%esp)   #,
59          call    printf  #
60  .L4:
61          movl    $0, %eax        #,
62          leave
63          ret
64          .size   ox2, .-ox2
65          .type   ox3, @function
66  ox3:
67          pushl   %ebp    #
68          movl    %esp, %ebp      #,
69          subl    $40, %esp       #,
70          movl    %eax, -20(%ebp) #, cow
71          movl    %edx, -16(%ebp) #, cow
72          movl    %ecx, -12(%ebp) #, cow
73          call    rand    #
74          movl    %eax, %ecx      #, D.3271
75          movl    $1441151881, %edx       #, tmp67
76          imull   %edx    # tmp67
77          sarl    $25, %edx       #, tmp68
78          movl    %ecx, %eax      # D.3271, tmp69
79          sarl    $31, %eax       #, tmp69
80          subl    %eax, %edx      # tmp69, tmp65
81          imull   $100000000, %edx, %edx  #, tmp65, tmp70
82          cmpl    %edx, %ecx      # tmp70, D.3271
83          jne     .L6     #,
84          movl    24(%ebp), %eax  # l, l
85          movl    %eax, 12(%esp)  # l,
86          flds    -20(%ebp)       # cow.red
87          fstpl   4(%esp) #
88          movl    $.LC0, (%esp)   #,
89          call    printf  #
90  .L6:
91          movl    $0, %eax        #,
92          leave
93          ret
94          .size   ox3, .-ox3
95  .globl  main
96          .type   main, @function
```

```
main:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        andl    $-16, %esp      #,
        pushl   %ebx    #
        subl    $76, %esp       #,
        movl    $0, (%esp)      #,
        call    srand   #
        movl    $0x40eccccd, 52(%esp)   #, cow.red
        movl    $6, 56(%esp)    #, cow.green
        leal    52(%esp), %eax  #, tmp62
        movl    %eax, 60(%esp)  # tmp62, cow.blue
        movl    $0x41066666, 40(%esp)   #, calf.red
        movl    $5, 44(%esp)    #, calf.green
        movl    %eax, 48(%esp)  # tmp62, calf.blue
        movl    $500000000, %ebx        #, ivtmp.4
.L8:
        movl    $2, 16(%esp)    #,
        movl    40(%esp), %eax  # calf, calf
        movl    %eax, 4(%esp)   # calf,
        movl    44(%esp), %eax  # calf, calf
        movl    %eax, 8(%esp)   # calf,
        movl    48(%esp), %eax  # calf, calf
        movl    %eax, 12(%esp)  # calf,
        movl    $4, (%esp)      #,
        movl    52(%esp), %eax  # cow,
        movl    56(%esp), %edx  # cow,
        movl    60(%esp), %ecx  # cow,
        call    ox      #
        movl    $5, 16(%esp)    #,
        movl    52(%esp), %eax  # cow, cow
        movl    %eax, 4(%esp)   # cow,
        movl    56(%esp), %eax  # cow, cow
        movl    %eax, 8(%esp)   # cow,
        movl    60(%esp), %eax  # cow, cow
        movl    %eax, 12(%esp)  # cow,
        movl    $4, (%esp)      #,
        movl    40(%esp), %eax  # calf,
        movl    44(%esp), %edx  # calf,
        movl    48(%esp), %ecx  # calf,
        call    ox2     #
        movl    $3, 16(%esp)    #,
        movl    40(%esp), %eax  # calf, calf
        movl    %eax, 4(%esp)   # calf,
        movl    44(%esp), %eax  # calf, calf
        movl    %eax, 8(%esp)   # calf,
        movl    48(%esp), %eax  # calf, calf
        movl    %eax, 12(%esp)  # calf,
        movl    $4, (%esp)      #,
        movl    52(%esp), %eax  # cow,
        movl    56(%esp), %edx  # cow,
        movl    60(%esp), %ecx  # cow,
        call    ox3     #
        subl    $1, %ebx        #, ivtmp.4
        jne     .L8     #,
        addl    $76, %esp       #,
```

```
153         popl    %ebx    #
154         movl    %ebp, %esp      #,
155         popl    %ebp    #
156         ret
157         .size   main, .-main
158         .ident  "GCC: (GNU) 4.5.2"
159         .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
1           .section        .rodata.str1.1,"aMS",@progbits,1
2   .LC0:
3           .string "cow.red, l -> %f, %d\n"
4           .text
5           .type   ox.clone.0, @function
6   ox.clone.0:
7           pushl   %ebp    #
8           movl    %esp, %ebp      #,
9           pushl   %ebx    #
10          subl    $20, %esp       #,
11          movl    %eax, %ebx      # l, l
12          call    rand    #
13          movl    %eax, %ecx      #, D.3290
14          movl    $1801439851, %edx       #, tmp66
15          imull   %edx    # tmp66
16          sarl    $22, %edx       #, tmp67
17          movl    %ecx, %eax      # D.3290, tmp68
18          sarl    $31, %eax       #, tmp68
19          subl    %eax, %edx      # tmp68, tmp64
20          imull   $10000000, %edx, %edx   #, tmp64, tmp69
21          cmpl    %edx, %ecx      # tmp69, D.3290
22          jne     .L2     #,
23          movl    %ebx, 12(%esp)  # l,
24          flds    8(%ebp) # ISRA.0
25          fstpl   4(%esp) #
26          movl    $.LC0, (%esp)   #,
27          call    printf  #
28  .L2:
29          movl    $0, %eax        #,
30          addl    $20, %esp       #,
31          popl    %ebx    #
32          popl    %ebp    #
33          ret
34          .size   ox.clone.0, .-ox.clone.0
35          .type   ox2.clone.1, @function
36  ox2.clone.1:
37          pushl   %ebp    #
38          movl    %esp, %ebp      #,
39          pushl   %ebx    #
40          subl    $20, %esp       #,
41          movl    %eax, %ebx      # l, l
42          call    rand    #
43          movl    %eax, %ecx      #, D.3304
44          movl    $1441151881, %edx       #, tmp66
45          imull   %edx    # tmp66
```

```
46          sarl    $24, %edx       #, tmp67
47          movl    %ecx, %eax      # D.3304, tmp68
48          sarl    $31, %eax       #, tmp68
49          subl    %eax, %edx      # tmp68, tmp64
50          imull   $50000000, %edx, %edx   #, tmp64, tmp69
51          cmpl    %edx, %ecx      # tmp69, D.3304
52          jne     .L4     #,
53          movl    %ebx, 12(%esp)  # l,
54          flds    8(%ebp) # ISRA.1
55          fstpl   4(%esp) #
56          movl    $.LC0, (%esp)   #,
57          call    printf  #
58  .L4:
59          movl    $0, %eax        #,
60          addl    $20, %esp       #,
61          popl    %ebx    #
62          popl    %ebp    #
63          ret
64          .size   ox2.clone.1, .-ox2.clone.1
65          .type   ox3.clone.2, @function
66  ox3.clone.2:
67          pushl   %ebp    #
68          movl    %esp, %ebp      #,
69          pushl   %ebx    #
70          subl    $20, %esp       #,
71          movl    %eax, %ebx      # l, l
72          call    rand    #
73          movl    %eax, %ecx      #, D.3318
74          movl    $1441151881, %edx       #, tmp66
75          imull   %edx    # tmp66
76          sarl    $25, %edx       #, tmp67
77          movl    %ecx, %eax      # D.3318, tmp68
78          sarl    $31, %eax       #, tmp68
79          subl    %eax, %edx      # tmp68, tmp64
80          imull   $100000000, %edx, %edx  #, tmp64, tmp69
81          cmpl    %edx, %ecx      # tmp69, D.3318
82          jne     .L6     #,
83          movl    %ebx, 12(%esp)  # l,
84          flds    8(%ebp) # ISRA.2
85          fstpl   4(%esp) #
86          movl    $.LC0, (%esp)   #,
87          call    printf  #
88  .L6:
89          movl    $0, %eax        #,
90          addl    $20, %esp       #,
91          popl    %ebx    #
92          popl    %ebp    #
93          ret
94          .size   ox3.clone.2, .-ox3.clone.2
95  .globl main
96          .type   main, @function
97  main:
98          pushl   %ebp    #
99          movl    %esp, %ebp      #,
100         andl    $-16, %esp      #,
101         pushl   %esi    #
```

```
102         pushl   %ebx    #
103         subl    $40, %esp       #,
104         movl    $0, (%esp)      #,
105         call    srand   #
106         movl    $0x40eccccd, 20(%esp)   #, cow.red
107         movl    $6, 24(%esp)    #, cow.green
108         leal    20(%esp), %eax  #, tmp62
109         movl    %eax, 28(%esp)  # tmp62, cow.blue
110         movl    $500000000, %esi        #, ivtmp.8
111         movl    $0x40eccccd, %ebx       #, tmp66
112 .L8:
113         movl    %ebx, (%esp)    # tmp66,
114         movl    $2, %eax        #,
115         call    ox.clone.0      #
116         movl    $0x41066666, (%esp)     #,
117         movl    $5, %eax        #,
118         call    ox2.clone.1     #
119         movl    %ebx, (%esp)    # tmp66,
120         movl    $3, %eax        #,
121         call    ox3.clone.2     #
122         subl    $1, %esi        #, ivtmp.8
123         jne     .L8     #,
124         addl    $40, %esp       #,
125         popl    %ebx    #
126         popl    %esi    #
127         movl    %ebp, %esp      #,
128         popl    %ebp    #
129         ret
130         .size   main, .-main
131         .ident  "GCC: (GNU) 4.5.2"
132         .section        .note.GNU-stack,"",@progbits
```

E.7 Constant propagation

Programa: constant.c

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int i, j, n=64;
        int a[n], c, d;
        int num_iter;

        c = 3;
        d = 4;
        num_iter = 100000000;

        for (i=0; i<num_iter; i++) {
                for (j=0; j<n; j++) {
                        a[j] = c + d;
                }
        }

        srand(c);
        i = rand() % n;

        printf("a[%d] -> %d\n", i, a[i]);
}
```

Ensamblador de la versión no optimizada:

```
        .section        .rodata
.LC0:
        .string "a[%d] -> %d\n"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx   #,
        andl    $-16, %esp      #,
        pushl   -4(%ecx)        #
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        pushl   %ebx    #
        pushl   %ecx    #
        subl    $48, %esp       #,
        movl    %esp, %eax      #, tmp78
        movl    %eax, %ebx      # tmp78, saved_stack.7
        movl    $64, -20(%ebp)  #, n
        movl    -20(%ebp), %eax # n, n.0
        leal    -1(%eax), %edx  #, D.3160
        movl    %edx, -24(%ebp) # D.3160, D.3161
        sall    $2, %eax        #, D.3171
        addl    $15, %eax       #, tmp79
        addl    $15, %eax       #, tmp80
```

```
25          shrl    $4, %eax        #, tmp81
26          sall    $4, %eax        #, tmp82
27          subl    %eax, %esp      # tmp82,
28          leal    12(%esp), %eax  #, D.3173
29          addl    $15, %eax       #, tmp83
30          shrl    $4, %eax        #, tmp84
31          sall    $4, %eax        #, tmp85
32          movl    %eax, -28(%ebp) # D.3173, a.5
33          movl    $3, -32(%ebp)   #, c
34          movl    $4, -36(%ebp)   #, d
35          movl    $100000000, -40(%ebp)   #, num_iter
36          movl    $0, -12(%ebp)   #, i
37          jmp     .L2     #
38  .L5:
39          movl    $0, -16(%ebp)   #, j
40          jmp     .L3     #
41  .L4:
42          movl    -36(%ebp), %eax # d, tmp86
43          movl    -32(%ebp), %edx # c, tmp87
44          leal    (%edx,%eax), %ecx       #, D.3174
45          movl    -28(%ebp), %eax # a.5, tmp88
46          movl    -16(%ebp), %edx # j, tmp89
47          movl    %ecx, (%eax,%edx,4)     # D.3174,
48          addl    $1, -16(%ebp)   #, j
49  .L3:
50          movl    -16(%ebp), %eax # j, tmp90
51          cmpl    -20(%ebp), %eax # n, tmp90
52          jl      .L4     #,
53          addl    $1, -12(%ebp)   #, i
54  .L2:
55          movl    -12(%ebp), %eax # i, tmp91
56          cmpl    -40(%ebp), %eax # num_iter, tmp91
57          jl      .L5     #,
58          movl    -32(%ebp), %eax # c, c.6
59          movl    %eax, (%esp)    # c.6,
60          call    srand   #
61          call    rand    #
62          movl    %eax, %edx      # D.3176, tmp93
63          sarl    $31, %edx       #, tmp93
64          idivl   -20(%ebp)       # n
65          movl    %edx, -12(%ebp) # tmp93, i
66          movl    -28(%ebp), %eax # a.5, tmp95
67          movl    -12(%ebp), %edx # i, tmp96
68          movl    (%eax,%edx,4), %edx     #, D.3177
69          movl    $.LC0, %eax     #, D.3178
70          movl    %edx, 8(%esp)   # D.3177,
71          movl    -12(%ebp), %edx # i, tmp97
72          movl    %edx, 4(%esp)   # tmp97,
73          movl    %eax, (%esp)    # D.3178,
74          call    printf  #
75          movl    %ebx, %esp      # saved_stack.7,
76          leal    -8(%ebp), %esp  #,
77          addl    $0, %esp        #,
78          popl    %ecx    #
79          popl    %ebx    #
80          popl    %ebp    #
```

```
81          leal    -4(%ecx), %esp  #,
82          ret
83          .size   main, .-main
84          .ident  "GCC: (GNU) 4.5.2"
85          .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
1           .section        .rodata.str1.1,"aMS",@progbits,1
2   .LC0:
3           .string "a[%d] -> %d\n"
4           .text
5   .globl main
6           .type   main, @function
7   main:
8           leal    4(%esp), %ecx   #,
9           andl    $-16, %esp      #,
10          pushl   -4(%ecx)        #
11          pushl   %ebp    #
12          movl    %esp, %ebp      #,
13          pushl   %ebx    #
14          pushl   %ecx    #
15          subl    $288, %esp      #,
16          leal    27(%esp), %ebx  #, tmp73
17          andl    $-16, %ebx      #, tmp75
18          movl    %ebx, %edx      # tmp75, a.2
19          movl    $0, %ecx        #, i
20          jmp     .L2     #
21  .L3:
22          movl    $7, (%edx,%eax,4)       #,* a.2
23          addl    $1, %eax        #, j
24          cmpl    $64, %eax       #, j
25          jne     .L3     #,
26          addl    $1, %ecx        #, i
27          cmpl    $100000000, %ecx        #, i
28          je      .L4     #,
29  .L2:
30          movl    $0, %eax        #, j
31          jmp     .L3     #
32  .L4:
33          movl    $3, (%esp)      #,
34          call    srand   #
35          call    rand    #
36          movl    $64, %ecx       #, tmp78
37          movl    %eax, %edx      # D.3249, tmp76
38          sarl    $31, %edx       #, tmp76
39          idivl   %ecx    # tmp78
40          movl    (%ebx,%edx,4), %eax     #, tmp79
41          movl    %eax, 8(%esp)   # tmp79,
42          movl    %edx, 4(%esp)   # tmp76,
43          movl    $.LC0, (%esp)   #,
44          call    printf  #
45          leal    -8(%ebp), %esp  #,
46          popl    %ecx    #
47          popl    %ebx    #
```

```
48          popl    %ebp    #
49          leal    -4(%ecx), %esp  #,
50          ret
51          .size   main, .-main
52          .ident  "GCC: (GNU) 4.5.2"
53          .section        .note.GNU-stack,"",@progbits
```

E.8 Copy propagation

Programa: copy.c

```
#include <stdio.h>
#include <stdlib.h>

#define N 20000

int main()
{
        int r, s, t;
        int a[N], c;
        int i, j, k;

        srand(0);

        i = rand() % 10;
        t = i * 4;
        s = t;
        c = rand() % 10;
        r = t;

        for (j=0; j<N; j++)
                a[j] = c;

        for (j=0; j<N; j++) {
                printf("a[%d] -> %d\n", t, a[t]);
                for (k=0; k<30*N; k++) {
                        t = i * 4;
                        s = t;
                        r = t;
                }
        }

}
```

Ensamblador de la versión no optimizada:

```
        .section        .rodata
.LC0:
        .string "a[%d] -> %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        andl    $-16, %esp      #,
        pushl   %ebx    #
        subl    $80060, %esp    #,
        movl    $0, (%esp)      #,
        call    srand   #
        call    rand    #
        movl    %eax, %ecx      #, D.3163
```

```
17          movl    $1717986919, %edx       #, tmp64
18          movl    %ecx, %eax      # D.3163,
19          imull   %edx    # tmp64
20          sarl    $2, %edx        #, tmp65
21          movl    %ecx, %eax      # D.3163, tmp66
22          sarl    $31, %eax       #, tmp66
23          movl    %edx, %ebx      # tmp65,
24          subl    %eax, %ebx      # tmp66,
25          movl    %ebx, %eax      #, tmp67
26          movl    %eax, 80032(%esp)       # tmp67, i
27          movl    80032(%esp), %edx       # i, tmp68
28          movl    %edx, %eax      # tmp68, tmp69
29          sall    $2, %eax        #, tmp69
30          addl    %edx, %eax      # tmp68, tmp69
31          addl    %eax, %eax      # tmp70
32          movl    %ecx, %edx      # D.3163,
33          subl    %eax, %edx      # tmp69,
34          movl    %edx, %eax      #, tmp71
35          movl    %eax, 80032(%esp)       # tmp71, i
36          movl    80032(%esp), %eax       # i, tmp73
37          sall    $2, %eax        #, tmp72
38          movl    %eax, 80044(%esp)       # tmp72, t
39          movl    80044(%esp), %eax       # t, tmp74
40          movl    %eax, 80028(%esp)       # tmp74, s
41          call    rand    #
42          movl    %eax, %ecx      #, D.3164
43          movl    $1717986919, %edx       #, tmp76
44          movl    %ecx, %eax      # D.3164,
45          imull   %edx    # tmp76
46          sarl    $2, %edx        #, tmp77
47          movl    %ecx, %eax      # D.3164, tmp78
48          sarl    $31, %eax       #, tmp78
49          movl    %edx, %ebx      # tmp77,
50          subl    %eax, %ebx      # tmp78,
51          movl    %ebx, %eax      #, tmp79
52          movl    %eax, 80024(%esp)       # tmp79, c
53          movl    80024(%esp), %edx       # c, tmp80
54          movl    %edx, %eax      # tmp80, tmp81
55          sall    $2, %eax        #, tmp81
56          addl    %edx, %eax      # tmp80, tmp81
57          addl    %eax, %eax      # tmp82
58          movl    %ecx, %edx      # D.3164,
59          subl    %eax, %edx      # tmp81,
60          movl    %edx, %eax      #, tmp83
61          movl    %eax, 80024(%esp)       # tmp83, c
62          movl    80044(%esp), %eax       # t, tmp84
63          movl    %eax, 80020(%esp)       # tmp84, r
64          movl    $0, 80040(%esp) #, j
65          jmp     .L2     #
66  .L3:
67          movl    80040(%esp), %eax       # j, tmp85
68          movl    80024(%esp), %edx       # c, tmp86
69          movl    %edx, 20(%esp,%eax,4)   # tmp86, a
70          addl    $1, 80040(%esp) #, j
71  .L2:
72          cmpl    $19999, 80040(%esp)     #, j
```

```
73          jle     .L3     #,
74          movl    $0, 80040(%esp) #, j
75          jmp     .L4     #
76  .L7:
77          movl    80044(%esp), %eax       # t, tmp87
78          movl    20(%esp,%eax,4), %edx   # a, D.3165
79          movl    $.LC0, %eax     #, D.3166
80          movl    %edx, 8(%esp)   # D.3165,
81          movl    80044(%esp), %edx       # t, tmp88
82          movl    %edx, 4(%esp)   # tmp88,
83          movl    %eax, (%esp)    # D.3166,
84          call    printf  #
85          movl    $0, 80036(%esp) #, k
86          jmp     .L5     #
87  .L6:
88          movl    80032(%esp), %eax       # i, tmp90
89          sall    $2, %eax        #, tmp89
90          movl    %eax, 80044(%esp)       # tmp89, t
91          movl    80044(%esp), %eax       # t, tmp91
92          movl    %eax, 80028(%esp)       # tmp91, s
93          movl    80044(%esp), %eax       # t, tmp92
94          movl    %eax, 80020(%esp)       # tmp92, r
95          addl    $1, 80036(%esp) #, k
96  .L5:
97          cmpl    $599999, 80036(%esp)    #, k
98          jle     .L6     #,
99          addl    $1, 80040(%esp) #, j
100 .L4:
101         cmpl    $19999, 80040(%esp)     #, j
102         jle     .L7     #,
103         addl    $80060, %esp    #,
104         popl    %ebx    #
105         movl    %ebp, %esp      #,
106         popl    %ebp    #
107         ret
108         .size   main, .-main
109         .ident  "GCC: (GNU) 4.5.2"
110         .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
1           .section        .rodata.str1.1,"aMS",@progbits,1
2   .LC0:
3           .string "a[%d] -> %d\n"
4           .text
5   .globl main
6           .type   main, @function
7   main:
8           pushl   %ebp    #
9           movl    %esp, %ebp      #,
10          andl    $-16, %esp      #,
11          pushl   %esi    #
12          pushl   %ebx    #
13          subl    $80024, %esp    #,
14          movl    $0, (%esp)      #,
```

```
15          call    srand   #
16          call    rand    #
17          movl    $10, %esi       #, tmp77
18          movl    %eax, %edx      # D.3242, tmp75
19          sarl    $31, %edx       #, tmp75
20          idivl   %esi    # tmp77
21          leal    0(,%edx,4), %ebx        #, t
22          call    rand    #
23          movl    %eax, %edx      # D.3243, c
24          sarl    $31, %edx       #, c
25          idivl   %esi    # tmp77
26          movl    $0, %eax        #, j
27  .L2:
28          movl    %edx, 16(%esp,%eax,4)   # c, a
29          addl    $1, %eax        #, j
30          cmpl    $20000, %eax    #, j
31          jne     .L2     #,
32          movl    $20000, %esi    #, ivtmp.1
33  .L4:
34          movl    16(%esp,%ebx,4), %eax   # a, tmp82
35          movl    %eax, 8(%esp)   # tmp82,
36          movl    %ebx, 4(%esp)   # t,
37          movl    $.LC0, (%esp)   #,
38          call    printf  #
39          movl    $600000, %eax   #, ivtmp.2
40  .L3:
41          subl    $1, %eax        #, ivtmp.2
42          jne     .L3     #,
43          subl    $1, %esi        #, ivtmp.1
44          jne     .L4     #,
45          addl    $80024, %esp    #,
46          popl    %ebx    #
47          popl    %esi    #
48          movl    %ebp, %esp      #,
49          popl    %ebp    #
50          ret
51          .size   main, .-main
52          .ident  "GCC: (GNU) 4.5.2"
53          .section        .note.GNU-stack,"",@progbits
```

E.9 Unreachable-code eliminaton & useless-code elimination

Programa: elimination.c

```
#include <stdio.h>
#include <stdlib.h>

#define N 10000

int foo (int a)
{
        printf("valor de a: %d\n", a);
}


int main()
{
        int a, b, c, d;
        int i, debug, n;
        int v[N], test;

        srand(0);
        b = rand() % 100;
        c = rand() % 100;
        d = rand() % 100;

        for (i=0; i<N; i++) {
                v[i] = rand() % 100;
        }

        for (test=0; test<200000; test++) {
                debug = 0;
                n = 0;
                a = b + 7;
                if (debug > 1) {
                        c = a + b + d;
                        printf("warning: total is %d\n", c);
                }

                foo(a);

                for (i=0; i<N; i++) {
                        v[i] = v[i] + c;
                }
        }
}
```

Ensamblador de la versión no optimizada:

```
        .text
.globl foo
        .type   foo, @function
foo:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
```

```
        movl    8(%ebp), %eax   # a, tmp60
        addl    $1, %eax        #, D.3185
        popl    %ebp    #
        ret
        .size   foo, .-foo
        .section        .rodata
.LC0:
        .string "warning: total is %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        andl    $-16, %esp      #,
        pushl   %esi    #
        pushl   %ebx    #
        subl    $40072, %esp    #,
        movl    $0, (%esp)      #,
        call    srand   #
        call    rand    #
        movl    %eax, %ecx      #, D.3169
        movl    $1374389535, %edx       #, tmp72
        movl    %ecx, %eax      # D.3169,
        imull   %edx    # tmp72
        sarl    $5, %edx        #, tmp73
        movl    %ecx, %eax      # D.3169, tmp74
        sarl    $31, %eax       #, tmp74
        movl    %edx, %ebx      # tmp73,
        subl    %eax, %ebx      # tmp74,
        movl    %ebx, %eax      #, tmp75
        movl    %eax, 40044(%esp)       # tmp75, b
        movl    40044(%esp), %eax       # b, tmp77
        imull   $100, %eax, %eax        #, tmp77, tmp76
        movl    %ecx, %esi      # D.3169,
        subl    %eax, %esi      # tmp76,
        movl    %esi, %eax      #, tmp78
        movl    %eax, 40044(%esp)       # tmp78, b
        call    rand    #
        movl    %eax, %ecx      #, D.3170
        movl    $1374389535, %edx       #, tmp80
        movl    %ecx, %eax      # D.3170,
        imull   %edx    # tmp80
        sarl    $5, %edx        #, tmp81
        movl    %ecx, %eax      # D.3170, tmp82
        sarl    $31, %eax       #, tmp82
        movl    %edx, %ebx      # tmp81,
        subl    %eax, %ebx      # tmp82,
        movl    %ebx, %eax      #, tmp83
        movl    %eax, 40060(%esp)       # tmp83, c
        movl    40060(%esp), %eax       # c, tmp85
        imull   $100, %eax, %eax        #, tmp85, tmp84
        movl    %ecx, %esi      # D.3170,
        subl    %eax, %esi      # tmp84,
        movl    %esi, %eax      #, tmp86
        movl    %eax, 40060(%esp)       # tmp86, c
```

```
63          call    rand    #
64          movl    %eax, %ecx      #, D.3171
65          movl    $1374389535, %edx       #, tmp88
66          movl    %ecx, %eax      # D.3171,
67          imull   %edx    # tmp88
68          sarl    $5, %edx        #, tmp89
69          movl    %ecx, %eax      # D.3171, tmp90
70          sarl    $31, %eax       #, tmp90
71          movl    %edx, %ebx      # tmp89,
72          subl    %eax, %ebx      # tmp90,
73          movl    %ebx, %eax      #, tmp91
74          movl    %eax, 40040(%esp)       # tmp91, d
75          movl    40040(%esp), %eax       # d, tmp93
76          imull   $100, %eax, %eax        #, tmp93, tmp92
77          movl    %ecx, %esi      # D.3171,
78          subl    %eax, %esi      # tmp92,
79          movl    %esi, %eax      #, tmp94
80          movl    %eax, 40040(%esp)       # tmp94, d
81          movl    $0, 40056(%esp) #, i
82          jmp     .L3     #
83  .L4:
84          call    rand    #
85          movl    %eax, %ecx      #, D.3172
86          movl    $1374389535, %edx       #, tmp96
87          movl    %ecx, %eax      # D.3172,
88          imull   %edx    # tmp96
89          sarl    $5, %edx        #, tmp97
90          movl    %ecx, %eax      # D.3172, tmp98
91          sarl    $31, %eax       #, tmp98
92          movl    %edx, %ebx      # tmp97,
93          subl    %eax, %ebx      # tmp98,
94          movl    %ebx, %eax      #, D.3173
95          imull   $100, %eax, %eax        #, D.3173, tmp99
96          movl    %ecx, %esi      # D.3172,
97          subl    %eax, %esi      # tmp99,
98          movl    %esi, %eax      #, D.3173
99          movl    40056(%esp), %edx       # i, tmp100
100         movl    %eax, 28(%esp,%edx,4)   # D.3173, v
101         addl    $1, 40056(%esp) #, i
102 .L3:
103         cmpl    $9999, 40056(%esp)      #, i
104         jle     .L4     #,
105         movl    $0, 40048(%esp) #, iter
106         jmp     .L5     #
107 .L10:
108         movl    $0, 40052(%esp) #, test
109         jmp     .L6     #
110 .L9:
111         movl    $0, 40036(%esp) #, debug
112         movl    $0, 40032(%esp) #, n
113         movl    40044(%esp), %eax       # b, tmp102
114         addl    $7, %eax        #, tmp101
115         movl    %eax, 40028(%esp)       # tmp101, a
116         cmpl    $1, 40036(%esp) #, debug
117         jle     .L7     #,
118         movl    40044(%esp), %eax       # b, tmp103
```

```
119             movl    40028(%esp), %edx       # a, tmp104
120             leal    (%edx,%eax), %eax       #, D.3176
121             addl    40040(%esp), %eax       # d, tmp105
122             movl    %eax, 40060(%esp)       # tmp105, c
123             movl    $.LC0, %eax     #, D.3177
124             movl    40060(%esp), %edx       # c, tmp106
125             movl    %edx, 4(%esp)   # tmp106,
126             movl    %eax, (%esp)    # D.3177,
127             call    printf  #
128     .L7:
129             movl    40052(%esp), %ecx       # test, tmp107
130             movl    $274877907, %edx        #, tmp109
131             movl    %ecx, %eax      # tmp107,
132             imull   %edx    # tmp109
133             sarl    $6, %edx        #, tmp110
134             movl    %ecx, %eax      # tmp107, tmp111
135             sarl    $31, %eax       #, tmp111
136             movl    %edx, %ebx      # tmp110,
137             subl    %eax, %ebx      # tmp111,
138             movl    %ebx, %eax      #, D.3178
139             imull   $1000, %eax, %eax       #, D.3178, tmp112
140             movl    %ecx, %esi      # tmp107,
141             subl    %eax, %esi      # tmp112,
142             movl    %esi, %eax      #, D.3178
143             testl   %eax, %eax      # D.3178
144             jne     .L8     #,
145             movl    40028(%esp), %eax       # a, tmp113
146             movl    %eax, (%esp)    # tmp113,
147             call    foo     #
148     .L8:
149             movl    40052(%esp), %ebx       # test, tmp114
150             movl    $1759218605, %edx       #, tmp116
151             movl    %ebx, %eax      # tmp114,
152             imull   %edx    # tmp116
153             sarl    $12, %edx       #, tmp117
154             movl    %ebx, %eax      # tmp114, tmp118
155             sarl    $31, %eax       #, tmp118
156             movl    %edx, %ecx      # tmp117, D.3181
157             subl    %eax, %ecx      # tmp118, D.3181
158             imull   $10000, %ecx, %eax      #, D.3181, tmp119
159             movl    %ebx, %ecx      # tmp114, D.3181
160             subl    %eax, %ecx      # tmp119, D.3181
161             movl    40052(%esp), %ebx       # test, tmp120
162             movl    $1759218605, %edx       #, tmp122
163             movl    %ebx, %eax      # tmp120,
164             imull   %edx    # tmp122
165             sarl    $12, %edx       #, tmp123
166             movl    %ebx, %eax      # tmp120, tmp124
167             sarl    $31, %eax       #, tmp124
168             movl    %edx, %esi      # tmp123,
169             subl    %eax, %esi      # tmp124,
170             movl    %esi, %eax      #, D.3182
171             imull   $10000, %eax, %eax      #, D.3182, tmp125
172             movl    %ebx, %edx      # tmp120,
173             subl    %eax, %edx      # tmp125,
174             movl    %edx, %eax      #, D.3182
```

```
175         movl    28(%esp,%eax,4), %eax   # v, D.3183
176         addl    40060(%esp), %eax       # c, D.3184
177         movl    %eax, 28(%esp,%ecx,4)   # D.3184, v
178         addl    $1, 40052(%esp) #, test
179 .L6:
180         cmpl    $1999999, 40052(%esp)   #, test
181         jle     .L9     #,
182         addl    $1, 40048(%esp) #, iter
183 .L5:
184         cmpl    $999, 40048(%esp)       #, iter
185         jle     .L10    #,
186         addl    $40072, %esp    #,
187         popl    %ebx    #
188         popl    %esi    #
189         movl    %ebp, %esp      #,
190         popl    %ebp    #
191         ret
192         .size   main, .-main
193         .ident  "GCC: (GNU) 4.5.2"
194         .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
1           .text
2   .globl foo
3           .type   foo, @function
4   foo:
5           pushl   %ebp    #
6           movl    %esp, %ebp      #,
7           movl    8(%ebp), %eax   # a, a
8           addl    $1, %eax        #, tmp61
9           popl    %ebp    #
10          ret
11          .size   foo, .-foo
12  .globl main
13          .type   main, @function
14  main:
15          pushl   %ebp    #
16          movl    %esp, %ebp      #,
17          andl    $-16, %esp      #,
18          pushl   %ebx    #
19          subl    $28, %esp       #,
20          movl    $0, (%esp)      #,
21          call    srand   #
22          call    rand    #
23          call    rand    #
24          .p2align 4,,5
25          call    rand    #
26          movl    $10000, %ebx    #, ivtmp.4
27  .L3:
28          call    rand    #
29          subl    $1, %ebx        #, ivtmp.4
30          jne     .L3     #,
31          movl    $0, %edx        #, iter
32          jmp     .L4     #
```

```
33  .L5:
34          subl    $1, %eax        #, ivtmp.3
35          jne     .L5     #,
36          addl    $1, %edx        #, iter
37          cmpl    $1000, %edx     #, iter
38          je      .L7     #,
39  .L4:
40          movl    $2000000, %eax  #, ivtmp.3
41          jmp     .L5     #
42  .L7:
43          addl    $28, %esp       #,
44          popl    %ebx    #
45          movl    %ebp, %esp      #,
46          popl    %ebp    #
47          ret
48          .size   main, .-main
49          .ident  "GCC: (GNU) 4.5.2"
50          .section        .note.GNU-stack,"",@progbits
```

E.10 Procedure inlining

Programa: inline.c

```
#include <stdio.h>
#include <stdlib.h>

#define N 200000000

int mcd(int a, int b)
{
        int t;

        while(b != 0) {
                t = b;
                b = a % b;
                a = t;
        }
        return a;
}

int pintar_resultado(int a, int b, int c)
{
        printf("El mcd de %d y %d es %d\n", a, b, c);
}

int main()
{
        int a, b, c, i;

        srand(0);

        for (i=0; i<N; i++) {
                a = rand() % 10000;
                b = rand() % 10000;
                c = mcd(a, b);
                if (c == 3333) pintar_resultado(a, b, c);
        }
}
```

Ensamblador de la versión no optimizada:

```
        .text
        .p2align 4,,15
.globl mcd
        .type   mcd, @function
mcd:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        movl    12(%ebp), %ecx  # b, b
        movl    8(%ebp), %eax   # a, a
        testl   %ecx, %ecx      # b
        jne     .L5     #,
        jmp     .L2     #
        .p2align 4,,7
```

```
14          .p2align 3
15  .L4:
16          movl    %ecx, %eax      # b, a
17          movl    %edx, %ecx      # tmp62, b
18  .L5:
19          movl    %eax, %edx      # a, tmp62
20          sarl    $31, %edx       #, tmp62
21          idivl   %ecx    # b
22          testl   %edx, %edx      # tmp62
23          jne     .L4     #,
24          movl    %ecx, %eax      # b, a
25  .L2:
26          popl    %ebp    #
27          ret
28          .size   mcd, .-mcd
29          .section        .rodata.str1.1,"aMS",@progbits,1
30  .LC0:
31          .string "El mcd de %d y %d es %d\n"
32          .text
33          .p2align 4,,15
34  .globl pintar_resultado
35          .type   pintar_resultado, @function
36  pintar_resultado:
37          pushl   %ebp    #
38          movl    %esp, %ebp      #,
39          subl    $24, %esp       #,
40          movl    16(%ebp), %eax  # c, c
41          movl    $.LC0, (%esp)   #,
42          movl    %eax, 12(%esp)  # c,
43          movl    12(%ebp), %eax  # b, b
44          movl    %eax, 8(%esp)   # b,
45          movl    8(%ebp), %eax   # a, a
46          movl    %eax, 4(%esp)   # a,
47          call    printf  #
48          leave
49          ret
50          .size   pintar_resultado, .-pintar_resultado
51          .p2align 4,,15
52  .globl main
53          .type   main, @function
54  main:
55          pushl   %ebp    #
56          movl    %esp, %ebp      #,
57          andl    $-16, %esp      #,
58          pushl   %edi    #
59          pushl   %esi    #
60          movl    $200000000, %esi        #, ivtmp.7
61          pushl   %ebx    #
62          subl    $20, %esp       #,
63          movl    $0, (%esp)      #,
64          call    srand   #
65          jmp     .L10    #
66          .p2align 4,,7
67          .p2align 3
68  .L9:
69          subl    $1, %esi        #, ivtmp.7
```

```
        je      .L12    #,
.L10:
        .p2align 4,,8
        call    rand    #
        movl    %eax, %ecx      #, D.3247
        movl    $1759218605, %eax       #,
        imull   %ecx    # D.3247
        movl    %ecx, %eax      # D.3247, tmp69
        sarl    $31, %eax       #, tmp69
        movl    %edx, %ebx      #, a
        sarl    $12, %ebx       #, a
        subl    %eax, %ebx      # tmp69, a
        imull   $10000, %ebx, %ebx      #, a, tmp70
        subl    %ebx, %ecx      # tmp70, D.3247
        movl    %ecx, %ebx      # D.3247, a
        call    rand    #
        movl    %ebx, (%esp)    # a,
        movl    %eax, %ecx      #, D.3248
        movl    $1759218605, %eax       #,
        imull   %ecx    # D.3248
        movl    %ecx, %eax      # D.3248, tmp74
        sarl    $31, %eax       #, tmp74
        movl    %edx, %edi      #, b
        sarl    $12, %edi       #, b
        subl    %eax, %edi      # tmp74, b
        imull   $10000, %edi, %edi      #, b, tmp75
        subl    %edi, %ecx      # tmp75, D.3248
        movl    %ecx, 4(%esp)   # b,
        movl    %ecx, %edi      # D.3248, b
        call    mcd     #
        cmpl    $3333, %eax     #, c
        jne     .L9     #,
        movl    $3333, 8(%esp)  #,
        movl    %edi, 4(%esp)   # b,
        movl    %ebx, (%esp)    # a,
        call    pintar_resultado        #
        subl    $1, %esi        #, ivtmp.7
        jne     .L10    #,
        .p2align 4,,7
        .p2align 3
.L12:
        addl    $20, %esp       #,
        popl    %ebx    #
        popl    %esi    #
        popl    %edi    #
        movl    %ebp, %esp      #,
        popl    %ebp    #
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.5.2"
        .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
        .text
```

```
        .p2align 4,,15
.globl mcd
        .type   mcd, @function
mcd:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        movl    12(%ebp), %ecx  # b, b
        movl    8(%ebp), %eax   # a, a
        testl   %ecx, %ecx      # b
        jne     .L5     #,
        jmp     .L2     #
        .p2align 4,,7
        .p2align 3
.L4:
        movl    %ecx, %eax      # b, a
        movl    %edx, %ecx      # tmp62, b
.L5:
        movl    %eax, %edx      # a, tmp62
        sarl    $31, %edx       #, tmp62
        idivl   %ecx    # b
        testl   %edx, %edx      # tmp62
        jne     .L4     #,
        movl    %ecx, %eax      # b, a
.L2:
        popl    %ebp    #
        ret
        .size   mcd, .-mcd
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "El mcd de %d y %d es %d\n"
        .text
        .p2align 4,,15
.globl pintar_resultado
        .type   pintar_resultado, @function
pintar_resultado:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        subl    $24, %esp       #,
        movl    16(%ebp), %eax  # c, c
        movl    $.LC0, (%esp)   #,
        movl    %eax, 12(%esp)  # c,
        movl    12(%ebp), %eax  # b, b
        movl    %eax, 8(%esp)   # b,
        movl    8(%ebp), %eax   # a, a
        movl    %eax, 4(%esp)   # a,
        call    printf  #
        leave
        ret
        .size   pintar_resultado, .-pintar_resultado
        .p2align 4,,15
.globl main
        .type   main, @function
main:
        pushl   %ebp    #
        movl    %esp, %ebp      #,
        andl    $-16, %esp      #,
```

```
58          pushl   %edi    #
59          pushl   %esi    #
60          pushl   %ebx    #
61          subl    $36, %esp       #,
62          movl    $0, (%esp)      #,
63          call    srand   #
64          movl    $200000000, 28(%esp)    #, %sfp
65          .p2align 4,,7
66          .p2align 3
67  .L12:
68          call    rand    #
69          movl    %eax, %ecx      #, D.3247
70          movl    $1759218605, %eax       #,
71          imull   %ecx    # D.3247
72          movl    %ecx, %eax      # D.3247, tmp71
73          sarl    $31, %eax       #, tmp71
74          movl    %edx, %ebx      #, a
75          sarl    $12, %ebx       #, a
76          subl    %eax, %ebx      # tmp71, a
77          imull   $10000, %ebx, %ebx      #, a, tmp72
78          subl    %ebx, %ecx      # tmp72, D.3247
79          movl    %ecx, %ebx      # D.3247, a
80          call    rand    #
81          movl    %ebx, %ecx      # a, c
82          movl    %eax, %edi      #, D.3248
83          movl    $1759218605, %eax       #,
84          imull   %edi    # D.3248
85          movl    %edi, %eax      # D.3248, tmp76
86          sarl    $31, %eax       #, tmp76
87          movl    %edx, %esi      #, b
88          sarl    $12, %esi       #, b
89          subl    %eax, %esi      # tmp76, b
90          imull   $10000, %esi, %esi      #, b, tmp77
91          subl    %esi, %edi      # tmp77, D.3248
92          movl    %edi, %esi      # D.3248, b
93          je      .L9     #,
94          movl    %edi, %ecx      # b, c
95          movl    %ebx, %eax      # a, c
96          jmp     .L10    #
97          .p2align 4,,7
98          .p2align 3
99  .L14:
100         movl    %ecx, %eax      # c, c
101         movl    %edx, %ecx      # tmp78, c
102 .L10:
103         movl    %eax, %edx      # c, tmp78
104         sarl    $31, %edx       #, tmp78
105         idivl   %ecx    # c
106         testl   %edx, %edx      # tmp78
107         jne     .L14    #,
108 .L9:
109         cmpl    $3333, %ecx     #, c
110         je      .L16    #,
111 .L11:
112         subl    $1, 28(%esp)    #, %sfp
113         jne     .L12    #,
```

```
114           addl    $36, %esp       #,
115           popl    %ebx    #
116           popl    %esi    #
117           popl    %edi    #
118           movl    %ebp, %esp      #,
119           popl    %ebp    #
120           ret
121           .p2align 4,,7
122           .p2align 3
123   .L16:
124           movl    $3333, 12(%esp) #,
125           movl    %esi, 8(%esp)   # b,
126           movl    %ebx, 4(%esp)   # a,
127           movl    $.LC0, (%esp)   #,
128           call    printf  #
129           jmp     .L11    #
130           .size   main, .-main
131           .ident  "GCC: (GNU) 4.5.2"
132           .section        .note.GNU-stack,"",@progbits
```

E.11 Procedure cloning

Programa: clone.c

```
#include <stdio.h>
#include <stdlib.h>

#define N 2000000000

__attribute__((noinline))
int g (int b, int c)
{
        printf ("%d %d\n", b, c);
}

__attribute__((noinline))
int f0 (int a)
{
        if (a > 0)
                g(a, 3);
        else
                g(a, 5);
}

__attribute__((noinline))
int f1 (int a)
{
        if (a > 1)
                g(a, 4);
        else
                g(a, 6);
}

__attribute__((noinline))
int f2 (int a)
{
        if (a > 2)
                g(a, 5);
        else
                g(a, 7);
}

__attribute__((noinline))
int f3 (int a)
{
        if (a > 3)
                g(a, 6);
        else
                g(a, 8);
}

int main()
{
        int i, k;

```

```
52          for (k=0; k<3; k++) {
53                  for (i=0; i<N; i++) {
54                          if ((i % 500000000) == 0) {
55                                  f0(7);
56                                  f1(0);
57                                  f2(8);
58                                  f3(1);
59                          }
60                  }
61          }
62          return 0;
63  }
```

Ensamblador de la versión no optimizada:

```
1           .section        .rodata.str1.1,"aMS",@progbits,1
2   .LC0:
3           .string "%d %d\n"
4           .text
5           .p2align 4,,15
6   .globl g
7           .type   g, @function
8   g:
9           pushl   %ebp    #
10          movl    %esp, %ebp      #,
11          subl    $24, %esp       #,
12          movl    12(%ebp), %eax  # c, c
13          movl    $.LC0, (%esp)   #,
14          movl    %eax, 8(%esp)   # c,
15          movl    8(%ebp), %eax   # b, b
16          movl    %eax, 4(%esp)   # b,
17          call    printf  #
18          leave
19          ret
20          .size   g, .-g
21          .p2align 4,,15
22  .globl f0
23          .type   f0, @function
24  f0:
25          pushl   %ebp    #
26          movl    %esp, %ebp      #,
27          subl    $24, %esp       #,
28          movl    8(%ebp), %eax   # a, a
29          testl   %eax, %eax      # a
30          jle     .L3     #,
31          movl    $3, 4(%esp)     #,
32          movl    %eax, (%esp)    # a,
33          call    g       #
34          leave
35          ret
36          .p2align 4,,7
37          .p2align 3
38  .L3:
39          movl    $5, 4(%esp)     #,
40          movl    %eax, (%esp)    # a,
```

```
41          call    g       #
42          leave
43          ret
44          .size   f0, .-f0
45          .p2align 4,,15
46  .globl f1
47          .type   f1, @function
48  f1:
49          pushl   %ebp    #
50          movl    %esp, %ebp      #,
51          subl    $24, %esp       #,
52          movl    8(%ebp), %eax   # a, a
53          cmpl    $1, %eax        #, a
54          jle     .L8     #,
55          movl    $4, 4(%esp)     #,
56          movl    %eax, (%esp)    # a,
57          call    g       #
58          leave
59          ret
60          .p2align 4,,7
61          .p2align 3
62  .L8:
63          movl    $6, 4(%esp)     #,
64          movl    %eax, (%esp)    # a,
65          call    g       #
66          leave
67          ret
68          .size   f1, .-f1
69          .p2align 4,,15
70  .globl f2
71          .type   f2, @function
72  f2:
73          pushl   %ebp    #
74          movl    %esp, %ebp      #,
75          subl    $24, %esp       #,
76          movl    8(%ebp), %eax   # a, a
77          cmpl    $2, %eax        #, a
78          jg      .L15    #,
79          movl    $7, 4(%esp)     #,
80          movl    %eax, (%esp)    # a,
81          call    g       #
82          leave
83          ret
84          .p2align 4,,7
85          .p2align 3
86  .L15:
87          movl    $5, 4(%esp)     #,
88          movl    %eax, (%esp)    # a,
89          call    g       #
90          leave
91          ret
92          .size   f2, .-f2
93          .p2align 4,,15
94  .globl f3
95          .type   f3, @function
96  f3:
```

```
97          pushl   %ebp    #
98          movl    %esp, %ebp      #,
99          subl    $24, %esp       #,
100         movl    8(%ebp), %eax   # a, a
101         cmpl    $3, %eax        #, a
102         jg      .L20    #,
103         movl    $8, 4(%esp)     #,
104         movl    %eax, (%esp)    # a,
105         call    g       #
106         leave
107         ret
108         .p2align 4,,7
109         .p2align 3
110 .L20:
111         movl    $6, 4(%esp)     #,
112         movl    %eax, (%esp)    # a,
113         call    g       #
114         leave
115         ret
116         .size   f3, .-f3
117         .p2align 4,,15
118 .globl main
119         .type   main, @function
120 main:
121         pushl   %ebp    #
122         movl    %esp, %ebp      #,
123         andl    $-16, %esp      #,
124         pushl   %edi    #
125         movl    $3, %edi        #, ivtmp.13
126         pushl   %esi    #
127         movl    $1152921505, %esi       #, tmp73
128         pushl   %ebx    #
129         subl    $20, %esp       #,
130 .L22:
131         xorl    %ebx, %ebx      # i
132         jmp     .L24    #
133         .p2align 4,,7
134         .p2align 3
135 .L23:
136         addl    $1, %ebx        #, i
137         cmpl    $2000000000, %ebx       #, i
138         je      .L27    #,
139 .L24:
140         movl    %ebx, %eax      # i,
141         imull   %esi    # tmp73
142         movl    %ebx, %eax      # i, tmp68
143         sarl    $31, %eax       #, tmp68
144         sarl    $27, %edx       #, tmp64
145         subl    %eax, %edx      # tmp68, tmp64
146         imull   $500000000, %edx, %edx  #, tmp64, tmp69
147         cmpl    %edx, %ebx      # tmp69, i
148         jne     .L23    #,
149         addl    $1, %ebx        #, i
150         movl    $7, (%esp)      #,
151         call    f0      #
152         movl    $0, (%esp)      #,
```

```
153             call    f1      #
154             movl    $8, (%esp)      #,
155             call    f2      #
156             movl    $1, (%esp)      #,
157             call    f3      #
158             cmpl    $2000000000, %ebx       #, i
159             jne     .L24    #,
160     .L27:
161             subl    $1, %edi        #, ivtmp.13
162             jne     .L22    #,
163             addl    $20, %esp       #,
164             xorl    %eax, %eax      #
165             popl    %ebx    #
166             popl    %esi    #
167             popl    %edi    #
168             movl    %ebp, %esp      #,
169             popl    %ebp    #
170             ret
171             .size   main, .-main
172             .ident  "GCC: (GNU) 4.5.2"
173             .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
1               .section        .rodata.str1.1,"aMS",@progbits,1
2       .LC0:
3               .string "%d %d\n"
4               .text
5               .p2align 4,,15
6       .globl g
7               .type   g, @function
8       g:
9               pushl   %ebp    #
10              movl    %esp, %ebp      #,
11              subl    $24, %esp       #,
12              movl    12(%ebp), %eax  # c, c
13              movl    $.LC0, (%esp)   #,
14              movl    %eax, 8(%esp)   # c,
15              movl    8(%ebp), %eax   # b, b
16              movl    %eax, 4(%esp)   # b,
17              call    printf  #
18              leave
19              ret
20              .size   g, .-g
21              .p2align 4,,15
22              .type   f3.clone.0, @function
23      f3.clone.0:
24              pushl   %ebp    #
25              movl    %esp, %ebp      #,
26              subl    $24, %esp       #,
27              movl    $8, 4(%esp)     #,
28              movl    $1, (%esp)      #,
29              call    g       #
30              leave
31              ret
```

```
32          .size   f3.clone.0, .-f3.clone.0
33          .p2align 4,,15
34          .type   f0.clone.1, @function
35  f0.clone.1:
36          pushl   %ebp    #
37          movl    %esp, %ebp      #,
38          subl    $24, %esp       #,
39          movl    $3, 4(%esp)     #,
40          movl    $7, (%esp)      #,
41          call    g       #
42          leave
43          ret
44          .size   f0.clone.1, .-f0.clone.1
45          .p2align 4,,15
46          .type   f1.clone.2, @function
47  f1.clone.2:
48          pushl   %ebp    #
49          movl    %esp, %ebp      #,
50          subl    $24, %esp       #,
51          movl    $6, 4(%esp)     #,
52          movl    $0, (%esp)      #,
53          call    g       #
54          leave
55          ret
56          .size   f1.clone.2, .-f1.clone.2
57          .p2align 4,,15
58          .type   f2.clone.3, @function
59  f2.clone.3:
60          pushl   %ebp    #
61          movl    %esp, %ebp      #,
62          subl    $24, %esp       #,
63          movl    $5, 4(%esp)     #,
64          movl    $8, (%esp)      #,
65          call    g       #
66          leave
67          ret
68          .size   f2.clone.3, .-f2.clone.3
69          .p2align 4,,15
70  .globl f0
71          .type   f0, @function
72  f0:
73          pushl   %ebp    #
74          movl    %esp, %ebp      #,
75          subl    $24, %esp       #,
76          movl    8(%ebp), %eax   # a, a
77          testl   %eax, %eax      # a
78          jle     .L7     #,
79          movl    $3, 4(%esp)     #,
80          movl    %eax, (%esp)    # a,
81          call    g       #
82          leave
83          ret
84          .p2align 4,,7
85          .p2align 3
86  .L7:
87          movl    $5, 4(%esp)     #,
```

```
88          movl    %eax, (%esp)    # a,
89          call    g       #
90          leave
91          ret
92          .size   f0, .-f0
93          .p2align 4,,15
94  .globl f1
95          .type   f1, @function
96  f1:
97          pushl   %ebp    #
98          movl    %esp, %ebp      #,
99          subl    $24, %esp       #,
100         movl    8(%ebp), %eax   # a, a
101         cmpl    $1, %eax        #, a
102         jle     .L12    #,
103         movl    $4, 4(%esp)     #,
104         movl    %eax, (%esp)    # a,
105         call    g       #
106         leave
107         ret
108         .p2align 4,,7
109         .p2align 3
110 .L12:
111         movl    $6, 4(%esp)     #,
112         movl    %eax, (%esp)    # a,
113         call    g       #
114         leave
115         ret
116         .size   f1, .-f1
117         .p2align 4,,15
118 .globl f2
119         .type   f2, @function
120 f2:
121         pushl   %ebp    #
122         movl    %esp, %ebp      #,
123         subl    $24, %esp       #,
124         movl    8(%ebp), %eax   # a, a
125         cmpl    $2, %eax        #, a
126         jg      .L19    #,
127         movl    $7, 4(%esp)     #,
128         movl    %eax, (%esp)    # a,
129         call    g       #
130         leave
131         ret
132         .p2align 4,,7
133         .p2align 3
134 .L19:
135         movl    $5, 4(%esp)     #,
136         movl    %eax, (%esp)    # a,
137         call    g       #
138         leave
139         ret
140         .size   f2, .-f2
141         .p2align 4,,15
142 .globl f3
143         .type   f3, @function
```

```
144 f3:
145         pushl   %ebp    #
146         movl    %esp, %ebp      #,
147         subl    $24, %esp       #,
148         movl    8(%ebp), %eax   # a, a
149         cmpl    $3, %eax        #, a
150         jg      .L24    #,
151         movl    $8, 4(%esp)     #,
152         movl    %eax, (%esp)    # a,
153         call    g       #
154         leave
155         ret
156         .p2align 4,,7
157         .p2align 3
158 .L24:
159         movl    $6, 4(%esp)     #,
160         movl    %eax, (%esp)    # a,
161         call    g       #
162         leave
163         ret
164         .size   f3, .-f3
165         .p2align 4,,15
166 .globl main
167         .type   main, @function
168 main:
169         pushl   %ebp    #
170         movl    %esp, %ebp      #,
171         andl    $-16, %esp      #,
172         pushl   %edi    #
173         movl    $3, %edi        #, ivtmp.21
174         pushl   %esi    #
175         movl    $1152921505, %esi       #, tmp73
176         pushl   %ebx    #
177         subl    $4, %esp        #,
178 .L26:
179         xorl    %ebx, %ebx      # i
180         jmp     .L28    #
181         .p2align 4,,7
182         .p2align 3
183 .L27:
184         addl    $1, %ebx        #, i
185         cmpl    $2000000000, %ebx       #, i
186         je      .L31    #,
187 .L28:
188         movl    %ebx, %eax      # i,
189         imull   %esi    # tmp73
190         movl    %ebx, %eax      # i, tmp68
191         sarl    $31, %eax       #, tmp68
192         sarl    $27, %edx       #, tmp64
193         subl    %eax, %edx      # tmp68, tmp64
194         imull   $500000000, %edx, %edx  #, tmp64, tmp69
195         cmpl    %edx, %ebx      # tmp69, i
196         jne     .L27    #,
197         addl    $1, %ebx        #, i
198         call    f0.clone.1      #
199         call    f1.clone.2      #
```

```
200         call    f2.clone.3      #
201         .p2align 4,,5
202         call    f3.clone.0      #
203         cmpl    $2000000000, %ebx       #, i
204         jne     .L28    #,
205 .L31:
206         subl    $1, %edi        #, ivtmp.21
207         jne     .L26    #,
208         addl    $4, %esp        #,
209         xorl    %eax, %eax      #
210         popl    %ebx    #
211         popl    %esi    #
212         popl    %edi    #
213         movl    %ebp, %esp      #,
214         popl    %ebp    #
215         ret
216         .size   main, .-main
217         .ident  "GCC: (GNU) 4.5.2"
218         .section        .note.GNU-stack,"",@progbits
```

E.12 Tail recursion elimination

Programa: recursion.c

```
#include <stdio.h>
#include <stdlib.h>

#define N   230000
#define REPEAT 10000

int inarray(int vector[], int elemento, int indice)
{
        if (indice > N) {
                return 0;
        }
        else {
                if (vector[indice] == elemento) {
                        return 1;
                }
                else {
                        inarray(vector, elemento, indice+1);
                }
        }
}

int main(int argc, char *argv[])
{
   int a[N];
   int i, r, x, existe;

   /* Inicializacion */

   srand(0);

   for (i = 0; i < N; i++)
   {
      a[i] = rand();
   }

   /* Ejecuta el experimento "repeat" veces */
   for (r = 0; r < REPEAT; r++)
   {
      x = rand();
      existe = inarray(a, x, 0);
      if (existe) printf("El numero %d esta en el vector\n", x);
      if (r % 1000 == 0) printf("iter: %d\n", r);
   }
}
```

Ensamblador de la versión no optimizada:

```
        .text
.globl inarray
        .type   inarray, @function
inarray:
```

```
5           pushl   %ebp     #
6           movl    %esp, %ebp      #,
7           subl    $24, %esp       #,
8           movl    8(%ebp), %edx   # vector, vector
9           movl    12(%ebp), %ecx  # elemento, elemento
10          movl    16(%ebp), %eax  # indice, indice
11          cmpl    $230000, %eax   #, indice
12          jg      .L3     #,
13          cmpl    %ecx, (%edx,%eax,4)     # elemento,* vector
14          je      .L4     #,
15          addl    $1, %eax        #, tmp68
16          movl    %eax, 8(%esp)   # tmp68,
17          movl    %ecx, 4(%esp)   # elemento,
18          movl    %edx, (%esp)    # vector,
19          call    inarray #
20          jmp     .L1     #
21  .L3:
22          movl    $0, %eax        #, D.3254
23          jmp     .L1     #
24  .L4:
25          movl    $1, %eax        #, D.3254
26  .L1:
27          leave
28          ret
29          .size   inarray, .-inarray
30          .section        .rodata.str1.4,"aMS",@progbits,1
31          .align 4
32  .LC0:
33          .string "El n\303\272mero %d est\303\241 en el vector\n"
34          .section        .rodata.str1.1,"aMS",@progbits,1
35  .LC1:
36          .string "iter: %d\n"
37          .text
38  .globl main
39          .type   main, @function
40  main:
41          pushl   %ebp     #
42          movl    %esp, %ebp      #,
43          andl    $-16, %esp      #,
44          pushl   %edi     #
45          pushl   %esi     #
46          pushl   %ebx     #
47          subl    $920020, %esp   #,
48          movl    $0, (%esp)      #,
49          call    srand   #
50          movl    $0, %ebx        #, i
51  .L6:
52          call    rand    #
53          movl    %eax, 16(%esp,%ebx,4)   # D.3244, a
54          addl    $1, %ebx        #, i
55          cmpl    $230000, %ebx   #, i
56          jne     .L6     #,
57          movl    $0, %ebx        #, r
58          leal    16(%esp), %edi  #, tmp83
59  .L9:
60          call    rand    #
```

```
61          movl    %eax, %esi      #, x
62          movl    $0, 8(%esp)     #,
63          movl    %eax, 4(%esp)   # x,
64          movl    %edi, (%esp)    # tmp83,
65          call    inarray #
66          testl   %eax, %eax      # existe
67          je      .L7     #,
68          movl    %esi, 4(%esp)   # x,
69          movl    $.LC0, (%esp)   #,
70          call    printf  #
71  .L7:
72          movl    $274877907, %eax        #,
73          imull   %ebx    # r
74          sarl    $6, %edx        #, tmp79
75          movl    %ebx, %eax      # r, tmp80
76          sarl    $31, %eax       #, tmp80
77          subl    %eax, %edx      # tmp80, tmp76
78          imull   $1000, %edx, %edx       #, tmp76, tmp81
79          cmpl    %edx, %ebx      # tmp81, r
80          jne     .L8     #,
81          movl    %ebx, 4(%esp)   # r,
82          movl    $.LC1, (%esp)   #,
83          call    printf  #
84  .L8:
85          addl    $1, %ebx        #, r
86          cmpl    $10000, %ebx    #, r
87          jne     .L9     #,
88          addl    $920020, %esp   #,
89          popl    %ebx    #
90          popl    %esi    #
91          popl    %edi    #
92          movl    %ebp, %esp      #,
93          popl    %ebp    #
94          ret
95          .size   main, .-main
96          .ident  "GCC: (GNU) 4.5.2"
97          .section        .note.GNU-stack,"",@progbits
```

Ensamblador de la versión optimizada:

```
1           .text
2   .globl inarray
3           .type   inarray, @function
4   inarray:
5           pushl   %ebp    #
6           movl    %esp, %ebp      #,
7           pushl   %ebx    #
8           movl    8(%ebp), %ebx   # vector, vector
9           movl    12(%ebp), %ecx  # elemento, elemento
10          movl    16(%ebp), %edx  # indice, indice
11          movl    $0, %eax        #, D.3254
12          cmpl    $230000, %edx   #, indice
13          jg      .L2     #,
14          movb    $1, %al #,
15          cmpl    %ecx, (%ebx,%edx,4)     # elemento,* vector
```

```
16          jne     .L9     #,
17          jmp     .L2     #
18  .L5:
19          cmpl    %ecx, (%ebx,%edx,4)     # elemento,* vector
20          .p2align 4,,3
21          je      .L8     #,
22  .L9:
23          addl    $1, %edx        #, indice
24          cmpl    $230000, %edx   #, indice
25          jle     .L5     #,
26          movl    $0, %eax        #, D.3254
27          jmp     .L2     #
28  .L8:
29          movl    $1, %eax        #, D.3254
30  .L2:
31          popl    %ebx    #
32          popl    %ebp    #
33          ret
34          .size   inarray, .-inarray
35          .section        .rodata.str1.4,"aMS",@progbits,1
36          .align 4
37  .LC0:
38          .string "El n\303\272mero %d est\303\241 en el vector\n"
39          .section        .rodata.str1.1,"aMS",@progbits,1
40  .LC1:
41          .string "iter: %d\n"
42          .text
43  .globl main
44          .type   main, @function
45  main:
46          pushl   %ebp    #
47          movl    %esp, %ebp      #,
48          andl    $-16, %esp      #,
49          pushl   %edi    #
50          pushl   %esi    #
51          pushl   %ebx    #
52          subl    $920020, %esp   #,
53          movl    $0, (%esp)      #,
54          call    srand   #
55          movl    $0, %ebx        #, i
56  .L11:
57          call    rand    #
58          movl    %eax, 16(%esp,%ebx,4)   # D.3244, a
59          addl    $1, %ebx        #, i
60          cmpl    $230000, %ebx   #, i
61          jne     .L11    #,
62          movl    $0, %ebx        #, r
63          leal    16(%esp), %edi  #, tmp81
64  .L14:
65          call    rand    #
66          movl    %eax, %esi      #, x
67          movl    $0, 8(%esp)     #,
68          movl    %eax, 4(%esp)   # x,
69          movl    %edi, (%esp)    # tmp81,
70          call    inarray #
71          testl   %eax, %eax      # existe
```

```
72          je      .L12    #,
73          movl    %esi, 4(%esp)   # x,
74          movl    $.LC0, (%esp)   #,
75          call    printf  #
76  .L12:
77          movl    $274877907, %eax        #,
78          imull   %ebx    # r
79          sarl    $6, %edx        #, tmp77
80          movl    %ebx, %eax      # r, tmp78
81          sarl    $31, %eax       #, tmp78
82          subl    %eax, %edx      # tmp78, tmp74
83          imull   $1000, %edx, %edx       #, tmp74, tmp79
84          cmpl    %edx, %ebx      # tmp79, r
85          jne     .L13    #,
86          movl    %ebx, 4(%esp)   # r,
87          movl    $.LC1, (%esp)   #,
88          call    printf  #
89  .L13:
90          addl    $1, %ebx        #, r
91          cmpl    $10000, %ebx    #, r
92          jne     .L14    #,
93          addl    $920020, %esp   #,
94          popl    %ebx    #
95          popl    %esi    #
96          popl    %edi    #
97          movl    %ebp, %esp      #,
98          popl    %ebp    #
99          ret
100         .size   main, .-main
101         .ident  "GCC: (GNU) 4.5.2"
102         .section        .note.GNU-stack,"",@progbits
```

Apéndice F
Gestión del proyecto: control de esfuerzos

En este apéndice se incluye un calendario de la realización del proyecto y una gráfica con las distintas tareas y el número de horas que ha costado cada una.

Este proyecto empezó a realizarse a finales de diciembre de 2010. A continuación se muestra una tabla con las distintas tareas y su coste en horas:

Tarea	Número de horas	Porcentaje
Estudio previo / Documentación	70 h	16 %
Preparación de la plataforma / Instalación del software	30 h	7 %
1ª ronda de mediciones	40 h	9 %
Resolución de los problemas de la plataforma	20 h	5 %
2ª ronda de mediciones	160 h	36 %
Memoria	120 h	27 %
Número total de horas	440 h	100 %

Tabla F.1: Número de horas trabajadas.