Práctica de optimización de código teniendo en cuenta la arquitectura para primer ciclo

M. Anguita, F. J. Fernández, A. F. Díaz, A. Cañas, A. Prieto
Dpto. de Arquitectura y Tecnología de Computadores
Universidad de Granada
18071 Granada
e-mail: manguita@atc.ugr.es

Resumen

Con el objetivo de motivar a los estudiantes en el estudio de materias de Arquitectura y Tecnología de Computadores, estamos proponiendo prácticas en las que los alumnos comprueban que pueden mejorar prestaciones en sus programas aplicando conocimientos adquiridos en estas materias. Aquí presentamos una práctica propuesta para asignaturas de estructura del primer ciclo, en las que se estudia la arquitectura *abstracta* del procesador o arquitectura del repertorio de instrucciones (ISA, *Instruction Set Architecture*). En la práctica propuesta los estudiantes, usando ensamblador dentro de código de alto nivel, mejoran las prestaciones de varios ejemplos: limitar los componentes de una lista a un rango, copia, inicialización y búsqueda. Comprueban que, utilizando conocimientos sobre arquitectura abstracta, pueden reducir el tiempo de ejecución de estos ejemplos en un 25%, 50%, o incluso en más de un 75%. La reducción depende del ejemplo, del procesador concreto, y del tamaño y ubicación en memoria de las entradas.

1. Introducción

La *motivación* es uno de los aspectos que más influye en el proceso de aprendizaje, de hecho el rendimiento y aprendizaje de un alumno motivado se aproxima al máximo de sus posibilidades. Para favorecer la motivación es esencial proporcionar un *sentido de utilidad* a los contenidos que se imparten. Los estudiantes que prevén dedicar su vida laboral a la programación de aplicaciones, a menudo, no se enfrentan al estudio de las asignaturas de arquitectura y de estructura de computadores suficientemente motivados, lo que les puede llevar a fracasar en estas materias. Con objeto de incrementar la motivación, se les puede demostrar que pueden mejorar las prestaciones de sus programas aplicando los conocimientos que adquieren en estas asignaturas. Para ser más convincentes, deberían obtener las mejoras ellos mismos, o se les debería mostrar resultados que estimen que pueden obtener ellos mismos, como por ejemplo, resultados logrados por otros estudiantes de la propia titulación. Otro objetivo que perseguimos es que los estudiantes tomen conciencia de que pueden usar estos conocimientos como argumentos a su favor a la hora de *competir por un trabajo* de programador con estudiantes de otras titulaciones superiores o estudios medios, en las que no se abarca el hardware de los computadores, o no se profundiza en su estudio, en la misma medida en la que se hace en las titulaciones de informática. Hay que tener en cuenta que las empresas de software que aprovechen las características de las arquitecturas disponibles, se verán beneficiadas dentro de un mercado competitivo.

Para alcanzar estos objetivos, por una parte, hemos propuesto trabajos fin de carrera, en los que los estudiantes comprueban que pueden obtener una buena relación entre prestaciones y coste de desarrollo, aplicando los conocimientos sobre arquitectura y estructura que han adquirido en la titulación ([6],[7]). Los resultados obtenidos en estos proyectos se muestran a los estudiantes de asignaturas de arquitectura y de estructura.

Por otra parte, se han introducido prácticas en asignaturas de arquitectura y de estructura en las que los estudiantes comprueban por sí mismos las mejoras en prestaciones que pueden conseguir aplicando contenidos que están estudiando (un trabajo de motivación similar ya se presentó en

Actas del SImposio Nacional de Docencia en la Informática, SINDI2005 (AENUI), pp.145-152
ISBN: 84-9732-443-9

[2]). En asignaturas de la materia troncal Tecnología y Estructura de Computadores de primer ciclo (Real Decreto 1459/1990 de 26 de octubre sobre *directrices generales propias* comunes de los planes de estudio de Ingeniero en Informática, BOE 20/11/1990), introdujimos una práctica sobre mejora de prestaciones en el curso 2002-2003, concretamente en la asignatura Estructura de los Computadores de 2º de Ingeniero Técnico en Informática de Sistemas. Aquí se presenta la práctica desarrollada en el curso 2004-2005.

Para mejorar prestaciones en una aplicación, se pueden utilizar tanto los conocimientos adquiridos sobre la *arquitectura abstracta* del procesador o ISA (repertorio de instrucciones, conjunto de registros, tipos de datos, modos de direccionamiento, lenguaje máquina y ensamblador), como los adquiridos sobre la *arquitectura concreta* o *microarquitectura* del procesador (implementación segmentada, superescalar, VLIW, SIMD, multihebra simultánea, etc.) y la arquitectura concreta de los sistemas de cómputo (multiprocesadores, multicomputadores, cluster). La arquitectura abstracta del procesador es objeto de estudio de asignaturas de la materia troncal de primer ciclo Tecnología y Estructura de Computadores, donde también se introduce la arquitectura concreta. Por otra parte, en las asignaturas de la materia troncal de segundo ciclo Arquitectura e Ingeniería de Computadores, se estudia en detalle la arquitectura concreta de los procesadores y de los computadores actuales.

Dado este reparto de contenidos entre estructura y arquitectura, las prácticas de mejora de prestaciones en asignaturas de arquitectura muestran al estudiante cómo incrementar prestaciones teniendo en cuenta la arquitectura concreta actual de los procesadores y computadores. Mientras que con la práctica para estructura que aquí se presenta se pretende que los estudiantes comprueben la utilidad de usar ensamblador en sus programas, aplicando así los conocimientos sobre arquitectura abstracta que adquieren en estructura: repertorios, tipos de registros, tipos de datos o modos de direccionamiento. Los estudiantes comprueban, con ejemplos muy sencillos, que resulta útil utilizar ensamblador para mejorar el *tiempo de ejecución* o también para obtener una *funcionalidad* que de otra forma no obtendrían. Resultados para aplicaciones completas se muestran al comienzo del curso a través de trabajos fin de carrera, como comentamos más arriba. Por otra parte, esta práctica familiariza al estudiante con otros conceptos abordados en la asignatura como el sistema de memoria y la segmentación del procesador.

En la Sección 2 se describe la práctica propuesta. La Sección 3 describe el trabajo realizado por los estudiantes e ilustra al profesor interesado en utilizar esta práctica sobre cómo usar los resultados como complemento a otros conceptos abordados en clases de teoría (sistema de memoria, segmentación). La Sección 4 muestra la repercusión de la práctica en la motivación de los estudiantes a través de dos encuestas. Por último, la Sección 5 presenta algunas conclusiones.

2. Descripción de la práctica.

En la Tabla 1 se puede ver el índice de la práctica. Se trata de la última práctica de un total de tres que realizan los estudiantes en la primera de las dos asignaturas de estructura impartidas en primer ciclo. Esta práctica tiene como objetivo adicional que los estudiantes aprendan el ensamblador usado tradicionalmente en Linux y la interfaz entre ensamblador y gcc/g++. En prácticas anteriores los estudiantes han utilizado la notación del ensamblador de Intel (usada tradicionalmente bajo DOS y Windows). Por este motivo, el apartado 3 del guión muestra las peculiaridades del ensamblador de Linux.

El apartado 2 del guión presenta al estudiante las herramientas que va a utilizar para desarrollar la práctica. Al buscar y elegir para la práctica un compilador para la sintaxis gcc/g++ y un entorno de desarrollo, se han tenido en cuenta varios criterios: (1) Gratuidad de las herramientas. (2) Compilador reciente. (3) Entorno de desarrollo que permita ejecutar paso a paso las instrucciones en ensamblador incluidas dentro del código en alto nivel. (4) Entorno de trabajo conocido por el estudiante, con el fin de evitar la pérdida de tiempo que supondría el aprendizaje de un nuevo entorno. Este año se ha optado por realizar las prácticas en Linux, ya que actualmente los estudiantes utilizan gcc/g++ en Linux en asignaturas de programación (en lugar de

Windows como en años anteriores). Para depurar se usa DDD (http://www.gnu.org/software/ddd/), que permite ejecutar paso a paso, además de instrucciones de alto nivel, las instrucciones en ensamblador.

Tabla 1. Índice de la práctica.

1. Resumen de objetivos.
2. Herramientas
3. Ensamblador de GNU.
3.1 Diferencias entre el ensamblador GNU y el de Intel.
3.2 Ensamblador en línea en gcc/g++.
4. Ejemplos de programas con ensamblador en línea.
5. Utilidad del ensamblador en línea.
6. Trabajo a desarrollar
6.1 Probar los programas de ejemplo del guión
6.2 Instrucciones de movimiento condicional
6.3 Instrucciones de manejo de cadenas
7. Referencias.

El apartado 4 del guión presenta ejemplos de programas gcc con ensamblador en línea que muestran errores habituales cuando se usa por primera vez la interfaz con gcc.

En asignaturas de estructura, como apoyo a sus contenidos, se suele estudiar en más profundidad el repertorio de instrucciones de algún procesador particular realizándose prácticas en las que se programa en ensamblador dicho procesador. En nuestras titulaciones se estudian como ejemplo los procesadores de la línea x86 tanto en asignaturas de estructura como de arquitectura. Hay varias razones que nos motivan a utilizar la línea x86: (1) Los alumnos, en el primer curso de la titulación, ya se han familiarizado con los conceptos de arquitectura abstracta estudiando un procesador sencillo [5]. (2) Son las arquitecturas disponibles en las aulas de prácticas, por lo que no se familiarizan con la arquitectura a través de simuladores. (3) La arquitectura abstracta de la línea x86 domina el mercado del PC (Athlon, Pentium 4, Pentium M), y se está extendiendo a otros mercados, como el de productos embebidos, de estaciones, de servidores (Xeon, Opteron) e incluso en supercomputadores (hay clusters en la lista de supercomputadores de www.top500.org basados en x86). (4) Destacamos también que Intel ofrece actualmente la arquitectura abstracta del 8086 para aplicaciones embebidas (para escáner, impresora, etc.) a través del 80186 (http://www.intel.com/design/intarch/intel186).

En el apartado 5 del guión se dan al estudiante argumentos a favor de usar los conocimientos que adquieren en estructura en sus programas de alto nivel. Deben tener en cuenta que hay instrucciones de los procesadores que los compiladores no generan o no lo hacen en todos los casos donde resultan rentables, o las generan en casos o de forma que no son rentables. Generalmente, los compiladores advierten al programador que debe comprobar si el código generado usando alguna opción de optimización realmente mejora prestaciones. Usar estas instrucciones puede ser rentable para disminuir tiempo de ejecución o para aprovechar una determinada funcionalidad. Dentro de estas instrucciones que mencionamos están las que se han incorporado recientemente a las arquitecturas, ya que los compiladores que aprovechan una arquitectura aparecen generalmente después de ésta, pero también hay instrucciones más antiguas.

Para poder beneficiarse de estas instrucciones el programador puede usar ensamblador o, si el ensamblador disponible no incluye la instrucción, código máquina. Otra alternativa para aprovecharlas es buscar bibliotecas de funciones que las utilicen, lo que puede encarecer el coste por tiempo (búsqueda, entrenamiento) y dinero (para adquirirlas inicialmente y mantenerlas) de desarrollo del software. Hay que tener en cuenta que las bibliotecas que aprovechan una arquitectura aparecen generalmente en el mercado después de la arquitectura y probablemente con un alto precio. Además, las funciones de biblioteca son generales, no proporcionan las mismas prestaciones que una implementación que se adapte específicamente a las necesidades de la aplicación. Por otro lado, algún informático debe programarlas.

Instrucciones de los procesadores de la línea x86 de Intel cuyo uso explícito en ensamblador (por el programador o a través de bibliotecas) nos pueden permitir reducir tiempo, teniendo en cuenta los compiladores actuales, son por ejemplo: las de manejo de cadenas (*movs, scas, cmps, stos*), de precaptación de memoria, instrucciones que saltan caches, instrucciones con procesamiento SIMD (MMX, SSE, SSE2, SSE3, 3DNow!), instrucciones de movimiento condicional (*cmov*) o instrucciones *setcc* ([3], [1]).

Otras instrucciones nos ofrecen alguna funcionalidad que podemos necesitar, como por ejemplo *xchg* o *rdtsc* en los procesadores de la línea x86. La instrucción *xchg* es útil para implementar primitivas de sincronización al permitir un acceso a memoria de lectura-modificación-escritura atómico. La instrucción *rdtsc* devuelve en 64 bits el número de ciclos de reloj desde que se inició el sistema, por lo que es útil para obtener el tiempo de ejecución con precisión de ciclos de reloj permitiéndonos así comparar la arquitectura concreta de los procesadores.

En el apartado 6 del guión ("Trabajo a desarrollar") se describe al estudiante el trabajo que ha de realizar. El trabajo se ha dividido en tres subapartados. En el primero el estudiante se familiariza con el ensamblador en línea dentro de gcc/g++ probando los ejemplos que hay en el guión de prácticas.

En el segundo, comprueba la utilidad de las instrucciones de movimiento condicional para mejorar prestaciones con dos ejemplos sencillos: (1) cálculo del mínimo de una lista, (2) limitar los elementos de una lista entre dos valores (usado, por ejemplo, en vídeo MPEG). El estudiante compara los ciclos de reloj que consumen estos ejemplos usando instrucciones de movimiento condicional (*cmov*), incluidas en la línea x86 a partir del PentiumPro (1995), y usando instrucciones de salto condicional. En Visual C++ de Microsoft se ha incorporado en la versión .NET 2003 la opción arch:SSE, que permite al compilador buscar puntos en los que resulta rentable usar la instrucción *cmov* además de instrucciones incluidas en el repertorio multimedia SSE [4]. En gcc esto se consigue usando opciones del compilador como -msse. No obstante, los compiladores no son capaces de usar estas instrucciones (incluida *cmov*) en todos los puntos donde resultan rentables. Por ejemplo, los compiladores mencionados son incapaces de generar cmov en el ejemplo que limita los componentes de una lista entre dos valores.

Con el tercer subapartado dentro del apartado 6, se ilustra la utilidad de las instrucciones de manejo de cadenas para mejorar prestaciones con tres ejemplos: (1) copiar de datos de 32 bits de una zona a otra de memoria, (2) inicializar una lista de componentes de 32 bits a un valor, y (3) buscar un dato de 32 bits en una lista. Las instrucciones de tratamiento de cadenas estaban ya incluidas en el repertorio del 8086. Hay funciones de biblioteca de los compiladores que se suelen implementar en ensamblador para utilizar estas instrucciones con el fin de mejorar prestaciones (por ejemplo, las funciones memchr, memcmp, memset o memcpy de la biblioteca string.h).

En todos estos ejemplos se mide el tiempo de ejecución en ciclos de reloj usando una macro basada en *rdtsc* que se da en el apartado 5 del guión.

3. Trabajo del estudiante

La práctica tiene asignada tres sesiones de dos horas, una sesión para cada subapartado de "Trabajo a desarrollar". La primera sesión es una sesión guiada en el que el profesor "obliga" a probar los ejemplos del guión dando puntos a los primeros alumnos en contestar a preguntas relacionadas con la ejecución de estos ejemplos. El objetivo principal de esta sesión es que los estudiantes se familiaricen con la sintaxis del ensamblador de Linux y con la interfaz entre gcc/g++ y ensamblador.

En las dos siguientes sesiones los estudiantes deben implementar, para cada una de las cuatro funciones de la Tabla 2, un código con ensamblador en línea dentro de código C que mejore las prestaciones de una versión que usa exclusivamente instrucciones en alto nivel. Además, realizan comparativas de tiempos de ejecución entre el ejecutable que genera el compilador a partir del fuente basado exclusivamente en instrucciones de alto nivel y el que genera a partir de la versión mejorada implementada por ellos mismos. En ningún momento los estudiantes han manifestado falta de tiempo para desarrollar la práctica, aún cuando la entregan trascurrida una semana desde la última sesión de la práctica.

Aquí presentaremos tiempos de ejecución que se pueden obtener en la práctica para dos de las funciones. Estos resultados permiten ilustrar: (1) en qué medida los alumnos observan que se pueden mejorar prestaciones usando, en ciertos puntos, ensamblador dentro de código de alto nivel; y (2) que el profesor puede usar la práctica para familiarizar al alumno sobre otros contenidos abordados en estructura. Estos resultados se han

obtenido compilado los códigos fuente con gcc 3.3.2 usando las opciones para optimización -O3 y -msse. Se dan resultados para dos computadores con diferentes procesadores: el Pentium 4 (P4) y el Pentium M (PM). Se recomienda a los estudiantes tomar datos en diferentes procesadores para poder así realizar comparativas entre arquitecturas concretas. En la Tabla 3 se pueden ver algunas características de los procesadores utilizados, entre ellos la familia (obtenida con la instrucción *cpuid*); un cambio de familia, frente a un cambio de modelo o actualización, supone un cambio sustancial en la arquitectura concreta del procesador.

Tabla 2. Funciones que realizan los códigos que mejoran los estudiantes.

6. Trabajo a desarrollar
6.2 Instrucciones de movimiento condicional
1) Limitar los componentes de una lista entre -256 y 255
6.3 Instrucciones de manejo de cadenas
2) Copia de datos entre listas
3) Inicializar a un valor una lista
4) Buscar un valor en una lista

Tabla 3. Procesadores utilizados.

Procesador	Familia	cache L1 datos	cache L1 inst.	cache L2
Pentium 4	15	8 KB	12Kμoper.	512KB
Pentium M	6	32KB	32KB	1MB

En todos los ejercicios, se trabaja con listas con componentes de 32 bits. El estudiante realiza ejecuciones para diferentes tamaños de lista: 16, 32, 64, 128, 256, 512, 1024, 2048 y 4096 componentes. Se llega a un tamaño en bytes de 16 KB. Así observan el comportamiento para tamaños pequeños y para tamaños grandes que incluso incluyen varias páginas de memoria (tamaño de páginas de 4 KB). Obtienen, para cada código a evaluar, el número de ciclos que consume su ejecución, y los ciclos que como media supone el procesamiento de una componente. Para cada tamaño obtienen la media de cuatro ejecuciones. Con estos valores rellenan una tabla, en la que también deben indicar la ganancia en tiempo de ejecución conseguida, para cada tamaño, con el código que usa ensamblador en línea frente al que no lo usa. Además, deben representar estos datos en gráficas, para poder así examinar/interpretar cómodamente los resultados.

En el subapartado 6.2 del guión, el estudiante primero comprueba la diferencia en prestaciones que supone para calcular el mínimo utilizar una instrucción de salto condicional frente a la utilización de una instrucción de movimiento condicional. En este ejemplo no debe implementar ningún código, se incluye para ilustrarle sobre cómo ha de implementar los códigos y las comparativas de prestaciones.

Por otra parte, en este subapartado del guión se les da código en alto nivel que limita los componentes de una lista entre -256 y 255, y se les pide que implementen código que mejore las prestaciones para esa función utilizando explícitamente instrucciones de movimiento condicional dentro del código gcc. (Tabla 2). En las Figuras 1 y 2 se pueden ver los resultados de ejecución en los dos procesadores de la Tabla 3 para: (1) el código para limitar que se da a los estudiantes (C), y (2) un programa gcc con ensamblador en línea (ASM) que limita cada componente con dos instrucciones de movimiento condicional en lugar de los saltos condicionales que genera el compilador. En estas gráficas se muestran, para cada tamaño, los ciclos que como media requiere el procesamiento de una componente (eje Y izquierda) y la ganancia en velocidad que se consigue (eje Y derecha) para los diferentes tamaños. En los dos programas se genera aleatoriamente la lista antes de pasar a limitar sus valores. Esto hace que la lista esté completa en la cache de nivel 2 antes de comenzar los cálculos; por tanto, no se observa la influencia ni de los fallos de cache en L2 ni del hardware de precaptación.

En el P4, el programa con ensamblador supone, para tamaños grandes, aproximadamente un 22,5% del tiempo del código generado por gcc (ganancia 4,4). Mientras que en el PM es aproximadamente un 28,5% (ganancia 3,5). Dado que cada etapa del cauce (*pipeline*) consume 1 ciclo, los estudiantes pueden percibir en las gráficas cómo se pone de manifiesto en los resultados el mayor número de etapas del cauce del P4 (tiene el doble de etapas que PM), especialmente para tamaños pequeños (para 16 componentes P4 consume 59,9 ciclos/componente frente a los 28 de PM).

En el tercer subapartado del apartado 6 del guión de prácticas (Tabla 2), los estudiantes escriben código ensamblador dentro de gcc en el que utilizan instrucciones de manejo de cadenas combinadas con prefijos de repetición (*rep*, *repnz*). Se pretende ilustrar a los estudiantes sobre el uso de estas instrucciones y sobre su utilidad para mejorar prestaciones. Utilizan en concreto: *rep movs* para copiar una lista de componentes de 32 bits en otra*, rep stos* para inicializar una lista de componentes de 32 bits a un valor, y *repnz scas* para encontrar un valor en una lista de 32 bits.

Como ejemplo, mostraremos aquí resultados para *rep stos.* El código de alto nivel que ejecutan los estudiantes se basa en:

```
for (i=0;i<num;i++) vector[i]=valor;
```

Para mejorar el código máquina generado por gcc utilizan *rep stos* a través de la siguiente sentencia *asm* (incluye la interfaz con gcc):

```
asm ( "movl %0,%%edi          \n\t"
      "movl %1,%%ecx          \n\t"
      "movl %2,%%eax          \n\t"
      "cld                    \n\t"
      "rep stosl              \n\t"
      ::"m"(vector),"m"(num),"m"(valor)
      :"%eax","%ecx","%edi","memory");
```

Es conveniente en este caso, que los estudiantes realicen tanto ejecuciones en las que los datos estén en cache L2 antes de pasar a ejecutar el código a evaluar, como ejecuciones en las que los datos no estén en cache; de esta forma pueden ver la influencia de los fallos de cache en las prestaciones, ilustrándose así lo comentado en clase de teoría sobre la necesidad de incluir caches en la jerarquía de memoria. En las Figuras 3 y 4 se pueden ver resultados obtenidos para este ejemplo en los procesadores de la Tabla 3. En cada gráfica se presentan los ciclos por componente para cuatro ejecutables. Hay dos ejecutables con código máquina generado por gcc (C y C-Ca), y dos ejecutables con parte del código fuente en ensamblador (ASM y ASM-Ca). En las versiones C-Ca y ASM-Ca los componentes de la lista se generan aleatoriamente antes de pasar al cálculo, la lista estará pues en la cache L2. En estas gráficas se representan también la ganancia para el código con *rep stos* sin la lista en L2 (C/ASM) y la ganancia con la lista en L2 (C/ASM-Ca).

Se debería indicar a los estudiantes que el hecho de que los datos estén o no en cache al realizar el cálculo, dependerá de la aplicación concreta en la que se utilice el código, del hardware de precaptación que incluya el procesador, y también de la utilización o no de instrucciones ensamblador de precaptación.

Con estos ejecutables se observa la efectividad del hardware de precaptación a L2 en los procesadores de la Tabla 3. Este hardware precapta datos a L2 en caso de que se detecte un acceso a memoria secuencial con salto constante. El mecanismo se dispara concretamente cuando se detecta en un acceso regular 2-4 fallos consecutivos. No obstante, la precaptación no se realiza a través de páginas; esto hace que en el momento en que se accede a datos de la lista que están en una nueva página de memoria virtual, el número de ciclos que supone la ejecución de los ejecutables sin la lista en L2 sufra un aumento debido a los fallos de cache. Para observar claramente esta subida, la lista se ha almacenado en los ejemplos cerca del comienzo de una página.

La ganancia en prestaciones de las versiones con la lista en L2 (Ca) respecto a las versiones en la que no está en L2 es poco pronunciada de 16 a 512 componentes, debido a la actuación del hardware de precaptación. Para 1024, 2048 y 4096, el incremento en ciclos de las versiones sin componentes en cache se debe a que el hardware no precapta al cruzar de página: en los tres casos se ha cambiado de página poco antes de terminar la ejecución. Los fallos de L2 ocultan la mejora de las instrucciones de manejo de cadenas.

Los resultados permiten que el estudiante perciba la necesidad de la jerarquía de memoria y la utilidad del hardware de precaptación incorporado en los procesadores para evitar la penalización en el acceso a memoria principal.

Las gráficas además reflejan la arquitectura segmentada de los procesadores. Conforme se aumenta el tamaño de las listas, se decrementa el número de ciclos que requiere el procesamiento de una componente (siempre que no se acceda a un bloque de memoria que no está en cache o que se encuentre en una nueva página). Efectivamente, conforme se aumenta el tamaño se va tendiendo a la productividad que proporciona el cauce segmentado del procesador para el código que se ejecuta cíclicamente (obsérvese que conforme aumenta el tamaño quedan más diluidas en el tiempo de ejecución la penalización por predicción errónea en la última ejecución de la instrucción condicional del bucle, el coste de las instrucciones de fuera y de inicio del bucle y el de

los primeros acceso a memoria hasta que se activa la precaptación automática).

Así pues, como reflejan las gráficas, los resultados que se obtienen en la práctica permiten que el estudiante perciba las ventajas en tiempo de ejecución que puede conseguir utilizando ensamblador en ciertos puntos de su programa en alto nivel. Además, estos resultados familiarizan al estudiante con otros conceptos que han abordado en la asignatura y que se amplían en asignaturas posteriores, como la jerarquía de memoria (paginación, cache) o la segmentación del procesador. Así, por ejemplo, observan los beneficios de la segmentación del procesador y la degradación de prestaciones si hay *paradas* en el cauce debidas a instrucciones de salto (en mínimo, limitar) o a esperas de datos (fallos de cache).

4. Motivación del estudiante

Se han pasado dos encuestas (Tabla 4) para comprobar si la práctica motiva al estudiante a estudiar contenidos de asignaturas de estructura por su aplicabilidad en la mejora de prestaciones del software. La primera encuesta se pasó (a un total de 65 estudiantes) una vez realizada y evaluada en clase la primera práctica de ensamblador. En el guión y en la presentación en clase de esta primera práctica se comentaba la utilidad de programar en ensamblador dentro de código de alto nivel para: (1) aprovechar aspectos de la arquitectura no explotados por los compiladores y (2) conseguir las prestaciones necesarias en algunas aplicaciones.

Para apoyar estos argumentos se presentó a los estudiantes un trabajo fin de carrera que mejora, teniendo en cuenta la arquitectura, un reproductor MP3 [7]. Los estudiantes observan que la primera versión del reproductor, implementada siguiendo exactamente el estándar, no permite una reproducción en tiempo real, y que aplicando sucesivas mejoras, incluido el uso puntual de instrucciones ensamblador, el reproductor se equipara en prestaciones a los disponibles comercialmente sin utilizar la tarjeta de sonido. Esto se muestra en clase ejecutando las diversas versiones del reproductor implementadas y viendo en directo su consumo de CPU.

La segunda encuesta se pasó (a un total de 60 estudiantes) una vez realizadas y evaluadas todas las prácticas (la última de las cuales es la aquí presentada). El tanto por ciento de estudiantes que contestaron "sí" a la primera cuestión (Tabla 4) y dieron una respuesta correcta a la segunda fue de un 32,3% en la primera encuesta, mientras que en la segunda subió a un 71,7%. Tal vez, habría que tener en cuenta además, que un 31,7% de estos 60 alumnos no aprobaron las prácticas. Así pues, el número de respuestas favorables subió una vez que los alumnos comprobaron *por sí mismos* a través de la práctica aquí presentada que pueden aplicar contenidos estudiados en estructura para mejorar su software.

Tabla 4. Cuestiones contestadas por el estudiante.

1) ¿Cree que podrá aprovechar parte de los contenidos de Estructura de los Computadores (por ejemplo, la programación en lenguaje ensamblador) para mejorar, de alguna forma, los programas que desarrolle en su vida laboral? SI/NO.
2) ¿Para qué cree que un informático utilizaría código ensamblador en algún punto de su programa en alto nivel (C/C++)?

5. Conclusiones

Para incrementar la motivación en el estudio de las materias de arquitectura y tecnología de computadores, proponemos a los estudiantes de informática prácticas en las que verifican *por sí mismos* que pueden mejorar las prestaciones de sus programas aplicando contenidos de estas materias. En la práctica para asignaturas de estructura que aquí presentamos, los estudiantes reducen el tiempo de ejecución en varios ejemplos sencillos (limitar los componentes de una lista, copiar, inicializar y buscar) usando conocimientos sobre arquitectura *abstracta* adquiridos en estructura.

Creemos conveniente que los estudiantes perciban ya en el primer ciclo de las titulaciones de informática (cursos de grado en las próximas titulaciones) la importancia de que un ingeniero que va a desarrollar software conozca el hardware actual para poder y saber aprovecharlo. Las encuestas reflejan que esta práctica ha incrementado de un 32,3% a un 71,7% el número de estudiantes que opinan que pueden aplicar para mejorar prestaciones los conocimientos que sobre arquitectura abstracta han adquirido en estructura.

Referencias

[1] "AMD Athlon™ Processor x86 Code Optimization Guide", http://www.amd.com.

[2] J. Flich, J. Real, J.C. Cano y J. Sahuquillo. "Prácticas de ensamblador". JENUI 2000, Alcalá de Henares. Madrid.

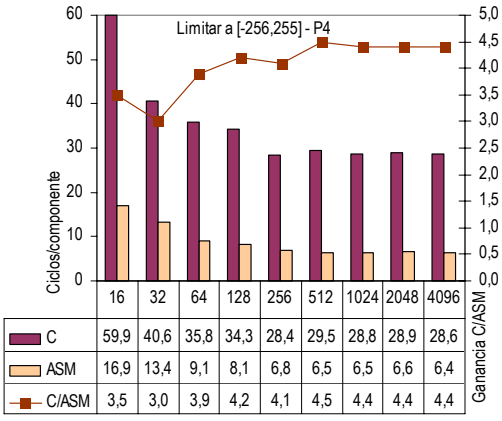
[3] "Intel® Pentium® 4 Processor Optimization Reference Manual", http://www.intel.com

[4] M. Lacey, "Optimizing Your Code with Visual C++" , Microsoft Corporation, 2003

[5] A. Prieto, F.J. Pelayo, F. Gómez-Mula, J. Ortega, A. Cañas, A. Martínez, F.J. Fernández. "Un computador didáctico elemental (CODE-2)", JENUI 2002, Cáceres.

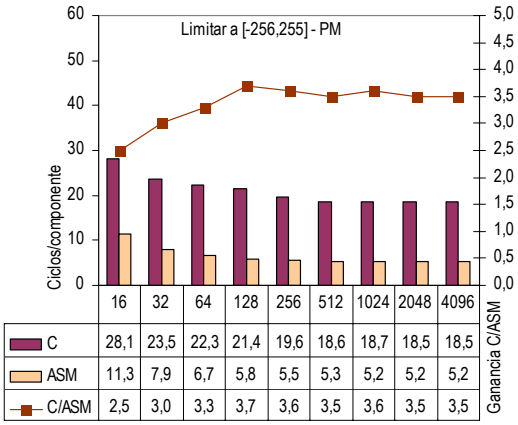
[6] Padilla, J.A; Anguita, M., Fernández, F.J.; Díaz, A.F.; Cañas, A.; Prieto, A. "Optimización de una implementación JPEG teniendo en cuenta la arquitectura actual de los procesadores", JENUI 2003, Cádiz.

[7] J. M. Martínez Lechado, M. Anguita. "Comparativa de implementaciones MP3 con diferente aprovechamiento de la arquitectura y complejidad algorítmica". *XIV Jornadas de Paralelismo*. Leganés, 2003.



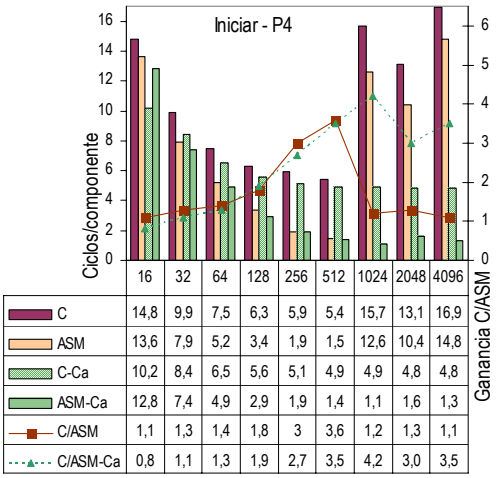
	16	32	64	128	256	512	1024	2048	4096
C	59,9	40,6	35,8	34,3	28,4	29,5	28,8	28,9	28,6
ASM	16,9	13,4	9,1	8,1	6,8	6,5	6,5	6,6	6,4
C/ASM	3,5	3,0	3,9	4,2	4,1	4,5	4,4	4,4	4,4

Figura 1. Limitar en Pentium 4



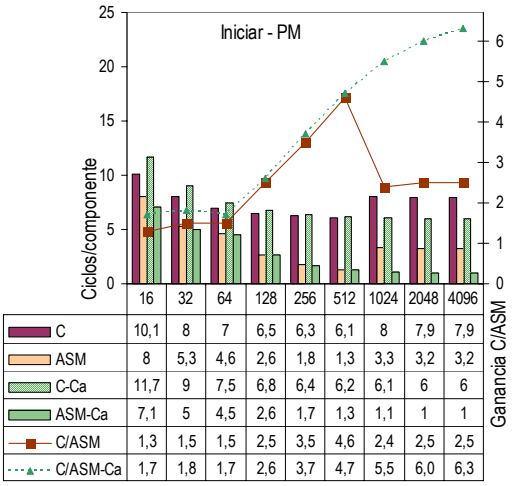
	16	32	64	128	256	512	1024	2048	4096
C	28,1	23,5	22,3	21,4	19,6	18,6	18,7	18,5	18,5
ASM	11,3	7,9	6,7	5,8	5,5	5,3	5,2	5,2	5,2
C/ASM	2,5	3,0	3,3	3,7	3,6	3,5	3,6	3,5	3,5

Figura 2. Limitar en Pentium M



	16	32	64	128	256	512	1024	2048	4096
C	14,8	9,9	7,5	6,3	5,9	5,4	15,7	13,1	16,9
ASM	13,6	7,9	5,2	3,4	1,9	1,5	12,6	10,4	14,8
C-Ca	10,2	8,4	6,5	5,6	5,1	4,9	4,9	4,8	4,8
ASM-Ca	12,8	7,4	4,9	2,9	1,9	1,4	1,1	1,6	1,3
C/ASM	1,1	1,3	1,4	1,8	3	3,6	1,2	1,3	1,1
C/ASM-Ca	0,8	1,1	1,3	1,9	2,7	3,5	4,2	3,0	3,5

Figura 3. Inicializar en Pentium 4



	16	32	64	128	256	512	1024	2048	4096
C	10,1	8	7	6,5	6,3	6,1	8	7,9	7,9
ASM	8	5,3	4,6	2,6	1,8	1,3	3,3	3,2	3,2
C-Ca	11,7	9	7,5	6,8	6,4	6,2	6,1	6	6
ASM-Ca	7,1	5	4,5	2,6	1,7	1,3	1,1	1	1
C/ASM	1,3	1,5	1,5	2,5	3,5	4,6	2,4	2,5	2,5
C/ASM-Ca	1,7	1,8	1,7	2,6	3,7	4,7	5,5	6,0	6,3

Figura 4. Inicializar en Pentium M