Análisis del rendimiento de programas

Área de Arquitectura y Tecnología de Computadores
Versión 1.0,
12/07/2016

Índice

Objetivos de la sesión

En esta práctica se introduce al alumno en los conceptos de análisis del rendimiento de programas utilizando *profilers*. Se aplicará la ley de Amdahl para guiar el proceso de optimización y para determinar el grado de mejora máximo que se puede conseguir mediante este proceso. Además, se verá cómo influyen las características de la arquitectura en el rendimiento de los programas.

Conocimientos y materiales necesarios

Para poder realizar esta sesión, el alumno debe:

- Disponer de una máquina, física o virtual, con un sistema operativo Ubuntu.

- Disponer en Windows de la hoja de cálculo que comenzaste a rellenar en la sesión anterior. Debería estar en tu repositorio.

- Repasar la ley de Amdahl.

- Durante la sesión se plantearán una serie de preguntas que puedes responder en el correspondiente cuestionario en el campus virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `Ctrl`.

1. Introducción al análisis de rendimiento de programas

- Arranca Linux.

- Cambia al directorio de tu repositorio y sincronízalo con el repositorio de Bitbucket para actualizar los cambios que se hubiesen realizado con anterioridad.

```
$> git pull
```

- Descarga los ficheros necesarios para la práctica y descomprímelos con estas órdenes:

```
$> wget http://rigel.atc.uniovi.es/grado/2ac/files/sesion1-3.tar.gz
```

- `$> tar xvfz sesion1-3.tar.gz`

- Añade los ficheros al índice git, confirma los cambios y súbelos al servidor con estas órdenes:

```
$> git add sesion1-3
```

- `$> git commit`
- `$> git push`

- Cámbiate al directorio `sesion1-3`.

- Copia el fichero de la hoja de cálculo a Windows para seguir modificándolo.

El rendimiento de los programas es un aspecto muy importante de su funcionamiento. El proceso de optimización intenta mejorar este rendimiento, pero

debe ser llevado a cabo de manera racional. Sin embargo, muchas veces se hacen de manera intuitiva «optimizaciones» (modificaciones del código que el programador cree que harán al programa más eficiente) que en realidad no mejoran el rendimiento y, además, pueden ir en contra de otras características muy importantes del código, como son la claridad, la legibilidad y la mantenibilidad. Una famosa cita de Donald Knuth, uno de los más reconocidos expertos en ciencias de la computación, recalca este problema: «La optimización prematura es la raíz de todos los males».

La optimización prematura es aquella que intenta optimizar antes de conocer qué partes del código tienen importancia real en el tiempo de ejecución del programa. El análisis de la complejidad de los algoritmos es una herramienta muy útil para comparar el rendimiento teórico de distintos algoritmos. Sin embargo, los programas reales tienen una gran cantidad de líneas de código, se ejecutan sobre arquitecturas de computadores muy complejas y pasan por una fase de compilación que puede llevar a cabo optimizaciones que no son obvias en el lenguaje de alto nivel; todo esto hace que el análisis de la complejidad no sea suficiente. Se requiere, además, medir la ejecución del programa para conocer en realidad qué partes interesa optimizar.

La medición se puede llevar a cabo de varias formas. Como se ha hecho en la práctica anterior, se puede instrumentar el código manualmente, es decir, introducir en el código sondas que tomen tiempos y, mediante la diferencia entre el tiempo en dos puntos del código, se puede saber cuánto se ha tardado en ejecutar esa parte del código. Si la marca de tiempo que toman esas sondas está basada en la hora del sistema, se obtiene lo que se conoce como *wall-clock time*. En un sistema multitarea, este tiempo no dependerá sólo de la propia ejecución del código del programa, sino que puede verse afectado por la ejecución de otros programas. Lógicamente, a la hora de optimizar, lo que importa es el tiempo realmente consumido por el programa, por lo que se intentará realizar la medición cuando haya menos elementos que puedan perturbarlo y se repetirá la prueba varias veces para que los errores aleatorios tengan menos importancia.

Este método de medición es muy útil. Una buena práctica que se sigue en muchos programas, en especial en entornos servidores, es incluir de manera habitual sondas en el código para llevar un registro (*log*) que sirve tanto para depurar el programa como para conocer qué partes del código están consumiendo la mayor parte del tiempo. Este registro suele estar activo «en producción», es decir, cuando el programa está siendo utilizado normalmente, lo que permite obtener datos muy valiosos sobre el funcionamiento del sistema en condiciones reales. Sin embargo, estas mediciones suelen ser de grandes fragmentos de código, ya que si se introducen demasiadas sondas, el código de medición ralentiza la aplicación.

Para conocer con precisión cuánto tiempo consume cada parte del código, se han desarrollado herramientas específicas que se utilizan en la fase final del desarrollo y no en producción. Las más importantes son los denominados *profilers*, que obtienen un perfil (*profile*) de la ejecución del programa. Este perfil está compuesto por una serie de datos sobre el programa como, por ejemplo, cuántas veces se ejecuta un fragmento de código (típicamente una función, aunque hay *profilers* que pueden llegar al nivel de línea de código), cuánto tarda de media en cada ejecución, etc. Con estos datos ya se pueden tomar decisiones de optimización adecuadas.

Los *profilers* habitualmente pueden obtener dos tipos de perfiles:

- Perfil plano: indica cuánto tiempo se consume de media en cada función y cuántas veces se llama en total a cada función.

- Grafo de llamadas: indica cuántas veces se llamó cada función desde otra función, lo que permite identificar relaciones entre funciones.

El objetivo de los *profilers* es encontrar «puntos calientes» (*hotspots*), es decir, zonas del programa que consumen mucho tiempo. Siguiendo la Ley de Amdahl, estos son los puntos donde más interesa realizar optimizaciones porque serán los que proporcionen una mayor aceleración.

En esta práctica se va a mostrar el proceso de análisis de rendimiento de programas

a través de pequeños programas. Lógicamente, cuanto más complejo es el programa más difícil es este proceso y más útiles resultan las optimizaciones.

2. Análisis del rendimiento de programas en C

2.1. Estudio del programa a analizar

Antes de utilizar un *profiler* se va a realizar una toma de contacto con el código a analizar.

- Abre con un editor el programa `prog1.c`, que forma parte de los ficheros que descargaste al principio de la prática.

- Observa la función `main` situada al final del código. Como verás, esta función declara dos vectores y a continuación llama a una función que los rellena con números aleatorios. Después llama a la función `contarPrimosComunes`, que cuenta los números primos que están en ambos vectores. Finalmente, `main` llama a una función que ordena el `vector`.

- Estudia el código de la función `contarPrimosComunes`. Como verás, realiza un bucle que recorre el primer vector y, para cada uno de sus elementos, comprueba si está en el otro vector y si es un número primo. La función `estaEnVector` simplemente recorre un vector hasta que encuentre el número por el que se le pregunta y, si no lo encuentra, devuelve `FALSE`. La función `esPrimo` recorre todos los números entre 2 y el número que se está analizando comprobando si hay

el número que se está analizando comprobando si hay alguna división con resto cero, en cuyo caso el número no será primo. Si todas las divisiones tienen resto distinto de cero, el número será primo.

- Estudia el código de la función `ordenaVector`. Esta función realiza una ordenación siguiendo el método de *selection sort*.

- Compila el programa con esta orden:

```
$> gcc prog1.c -o prog1
```

Esta orden utiliza GCC, el compilador de C desarrollado por GNU, para compilar y enlazar en un solo paso el archivo `prog1.c`. El parámetro `-o prog1` indica que el ejecutable de salida (*output*) se llame `prog1`.

- Ejecuta el programa midiendo cuánto tarda con esta orden:

```
$> time ./prog1
```

- Haz la medición cinco veces y apunta los valores `real` obtenidos en la fila `Prog1 sin optimizar` de la hoja `1-3` de la hoja de cálculo, calcula la media en la celda correspondiente, así como la desviación típica y el intevalo de confianza para la media con un nivel de confianza del 95%.
  ¿Qué debes introducir en la celda H3 de la hoja 1-3 para calcular la desviación estándar? Responde en el cuestionario: pregunta 1.

  ¿Qué debes introducir en la celda I3 de la hoja 1-3 para calcular el límite inferior del intervalo con un 95% de

> confianza? Responde en el cuestionario: pregunta 2.

A continuación, para comprobar lo difícil que es optimizar sin realizar mediciones, vamos a realizar algunas suposiciones y algunos cálculos y luego comprobaremos en qué grado acertamos.

¿Cuál crees que es la función que más tiempo consume del código? Apunta el valor en la celda `B14`. ¿Qué porcentaje del tiempo de ejecución crees que se corresponde con esa función? Apunta el valor en la celda `B15`.

Imagina que consigues optimizar el código de esa función haciendo que tarde la mitad de lo que tarda ahora. ¿Cuál sería la aceleración lograda en esa función? Escribe el valor en la celda `B16`. Responde en el cuestionario: pregunta 3. ¿Cuál sería la aceleración lograda en el programa completo? Escribe la fórmula correspondiente en la celda `B17`, utilizando referencias a otras celdas, no sus valores. Responde en el cuestionario: pregunta 4. ¿Cuál sería la fórmula para obtener el tiempo de ejecución del programa? Escríbela en la celda `B18`. Responde en el cuestionario: pregunta 5.

Para comprobar cuál es la realidad vamos a utilizar diversos *profilers*.

2.2. *Profiling* con GProf

En primer lugar se va a mostrar el proceso de *profiling* utilizando `gprof`, que es una herramienta libre con licencia GNU. A modo de resumen, el proceso es el siguiente:

- Recompilar el programa a analizar con `-pg`, opción que le indica al compilador que incluya el código del *profiler*.

- Ejecutar el programa. La ejecución generará el perfil, que es un archivo binario de nombre `gmon.out`. Si ya existiese un archivo con ese nombre, será sobreescrito.

- Utilizar el programa `gprof` para mostrar de manera textual la información de `gmon.out`.

En el caso concreto de tu programa, debes hacer lo siguiente:

- Recompila el programa añadiendo el *profiler* con esta orden:

```
$> gcc prog1.c -pg -o prog1
```

Fíjate que se ha añadido `-pg`.

- Ejecuta el programa con esta orden:

  ```
  $> time ./prog1
  ```

  No deberías observar un cambio significativo en el tiempo de ejecución con respecto a las ejecuciones anteriores. Si fuese así, eso significaría que el código de instrumentación está cambiando significativamente la ejecución del programa.

- Muestra los ficheros del directorio para comprobar que se ha generado el fichero `gmon.out`:

  ```
  $> ls
  ```

- Genera el informe de *profiling* con esta orden:

```
$> gprof ./prog1 > informe.txt
```

Fíjate que a `gprof` no hay que pasarle el fichero `gmon.out` sino el ejecutable con el que se generó el `gmon.out`. Como `gprof` genera la salida por pantalla, se utiliza la última parte de la orden, `> informe.txt`, para redireccionar la salida de `gprof` al fichero `informe.txt`.

- Edita el fichero `informe.txt`.

La salida de `gprof` tiene dos partes: el perfil plano y el grafo de llamadas. En el primero verás una línea por cada función. En cada línea, las columnas significan lo siguiente:

- `% time`: tanto por ciento del tiempo que el programa pasa en esa función.
- `cumulative seconds`: número de segundos en los que el programa está en esa función y en todas las que están sobre ella dentro del archivo generado.
- `self seconds`: número de segundos que el programa pasa en esa función. Es la columna que se utiliza para ordenar las funciones.
- `calls`: número de llamadas a la función.
- `self ms/call`: media del número de milisegundos que se emplean en esa función en cada llamada.
- `total ms/call`: media del número de milisegundos que se emplean en

cada llamada en esta función y las funciones a las que llama.

- `name`: nombre de la función.

Tras la explicación de cada columna, se encuentra el grafo de llamadas. Este grafo está divido por líneas de guiones en secciones. En cada sección se muestra información de la función que tiene un número entre corchetes en la primera columna. Las funciones de la misma sección que aparecen antes de la función son las que la llaman y las que aparecen después son a las que esa función llama. Por ejemplo:

```
index % time    self  children    called     name

[...]

                0.00    0.32       1/1           main [1]
[3]      2.1    0.00    0.32       1
contarPrimosComunes[...]
                0.27    0.00  100000/100000      estaEnVector[...]
                0.05    0.00  100000/100000      esPrimo(int) [5]
-----------------------------------------------
```

da información de la función `contarPrimosComunes`, que es llamada por `main` y llama a `estaEnVector` y `esPrimo`.

Para la función principal de la sección, las columnas significan lo siguiente:

- `% time` indica el tanto por ciento que consume la función y las funciones a las que llama en el total del programa.

- `self` indica el tiempo total en segundos consumido en la función (sin contar el tiempo consumido en las funciones a las que llama).

- `children` indica el tiempo que consumen las funciones a las que llama la función.

- `called` indica el número total de veces que ha sido llamada la función.

Ahora que has medido con el *profiler* la aplicación, puedes comprobar si tus suposiciones fueron adecuadas:

- ¿Cuál es la función que consume más tiempo? Escribe el valor en la celda `C14`. Responde en el cuestionario: pregunta 6.

- ¿Qué porcentaje del tiempo consume? Escribe el valor en la celda `C15`. Responde en el cuestionario: pregunta 7.

El *profiler* nos puede indicar dónde consume tiempo el programa, pero no la forma de solucionarlo. Hay que analizar por qué ocurre eso: ¿se está utilizando un algoritmo muy complejo en esa función que se podría solucionar utilizando un algoritmo más eficiente? ¿Se está llamando demasiadas veces a esa función y hay que cambiar el algoritmo en la función que provoca esas llamadas? ¿Las instrucciones que ejecuta la función no son las más eficientes? ¿Se podrían utilizar tipos de datos más eficientes?

En este sencillo caso, sabiendo que *selection sort* es uno de los peores algoritmos de ordenación, la primera solución podría ser cambiarlo por *quicksort*, lo que se hace en `prog2.c`. Vamos a comprobar que funciona:

- Compila y mide cinco veces el tiempo de ejecución de esta nueva versión del programa. Escribe estos valores en la fila `Prog2` optimizado de la hoja de cálculo. Calcula la media en la celda `G4`. ¿Cuál ha sido la aceleración del programa? Escribe la fórmula en la celda `K4`. Responde en el cuestionario: pregunta 8.

- Para hacer más rápido el programa ahora habría que hacer otro análisis de rendimiento. Compila con `-pg` el programa `prog2`, ejecútalo y obtén con `gprof` el perfil. ¿Cuál es ahora la función que consume la mayor parte del tiempo? Responde en el cuestionario: pregunta 9. ¿Qué porcentaje de tiempo consume? Responde en el cuestionario: pregunta 10.

GProf es una herramienta con una interfaz rudimentaria. Una alternativa en Linux es utilizar KCachegrind, un visualizador que utiliza los datos obtenidos por Callgrind, que es una herramienta que utiliza el sistema de instrumentación de Valgrind. Esta última herramienta tiene como función principal analizar el uso de memoria de los programas pero con Callgrind se convierte en un *profiler*. Para utilizar KCachegrind habría que usar Linux con interfaz gráfica, así que no vamos a verlo aquí.

2.3. Influencia de la optimización del compilador en el rendimiento

Una instrucción de un lenguaje de alto nivel se puede traducir a instrucciones máquina de muchas formas, algunas con un rendimiento mejor que otras. Los compiladores modernos incluyen una fase de optimización que se puede controlar mediante opciones. Activar las optimizaciones tiene algunas desventajas:

- Al tener que hacer más trabajo, el compilador tarda más. En proyectos con una gran cantidad de código esto puede suponer una pérdida de

productividad de los programadores si deben esperar demasiado tiempo en cada compilación.

- El compilador, al igual que los seres humanos, puede equivocarse creyendo que ciertas modificaciones harán el código más eficiente.

- Las optimizaciones pueden llegar a realizar cambios tan profundos en el código generado que sea difícil relacionar el código máquina con el código fuente. Esto dificulta la depuración línea a línea del código fuente y hace menos inteligible el resultado de los *profilers*.

Estos motivos pueden aconsejar la no utilización de las optimizaciones en algunos casos, pero en general lo conveniente es activarlas, por lo menos las más básicas para el código final si se encuentran problemas de rendimiento.

Desde el punto de vista del proceso de optimización mediante *profilers* que se está trabajando en esta práctica, hay que tener en cuenta que no merece la pena optimizar cambiando el código fuente cuando el compilador puede lograr la misma optimización de manera transparente para el programador por dos razones: 1) se está perdiendo tiempo del programador en realizar una tarea que pueden hacer automáticamente los compiladores y 2) como ya se ha comentado, las optimizaciones en muchas ocasiones hacen el código fuente menos inteligible. En cualquier caso, hay optimizaciones, sobre todo de alto nivel (por ejemplo, cambiar el algoritmo usado como se hizo en el ejemplo anterior) que no puede hacer el compilador. La forma correcta de trabajar es medir el código con las optimizaciones del compilador y sólo entonces, si no se logra el rendimiento requerido, se procede al proceso de optimización del código fuente por parte del programador.

Vamos a comprobar a continuación cómo afectan las optimizaciones del compilador. En los ejemplos que se han hecho hasta ahora no se había activado ninguna optimización. Vamos a volver a repetir algunos experimentos activando

opciones de optimización para ver las diferencias.

- Para activar el nivel más básico de optimización, se debe utilizar el parámetro `-O1` (la `O` viene de «Optimización») [1]. Compila el programa `prog1.c` con esta orden:

```
$> gcc -O1 prog1.c -o prog1
```

- Ejecuta el programa cinco veces tomando tiempos con esta orden:

```
$> time ./prog1
```

- Apunta los valores obtenidos en la fila `Prog1 -O1` de la hoja de cálculo y calcula el resto de celdas de esa fila extendiendo la fórmula de la fila inmediatamente superior, pero teniendo cuidado de que la aceleración se calcule sobre el programa 1 sin optimizar. ¿Mejora el tiempo de respuesta con respecto a la versión sin optimizar? Responde en el cuestionario: pregunta 11.

- Compila ahora con el segundo nivel de optimización mediante esta orden:

```
$> gcc -O2 prog1.c -o prog1
```

- Ejecuta el programa cinco veces tomando tiempos, apúntalos en la hoja de datos en la fila `Prog1 -O2` y calcula el resto de celdas de esa fila. ¿Mejora el tiempo de respuesta con respecto a `-O1`? Responde en el cuestionario: pregunta 12.

- Finalmente, repite el proceso con el máximo nivel de optimización:

> ```
> $> gcc -O3 prog1.c -o prog1
> ```
>
> - Ejecuta el programa cinco veces tomando tiempos, apúntalos en la hoja de datos en la fila `Prog1 -O3` y calcula el resto de celdas de esa fila. ¿Mejora el tiempo de respuesta respecto a `-O2`? Responde en el cuestionario: pregunta 13.

Como ves, el proceso de optimización no es nada sencillo. Hasta el momento hemos probado estas opciones:

1. Compilar el programa original con `-O1`.
2. Compilar el programa original con `-O2`.
3. Compilar el programa original con `-O3`.
4. Utilizar `prog2`, el programa optimizado a mano.

A la vista de todos los resultados que has obtenido, ¿cuál es la opción más rápida? Responde en el cuestionario: pregunta 14. Pero todavía queda saber qué ocurre si se aplican las optimizaciones a `prog2`. Pruébalo con los tres niveles de optimización. ¿Cuál es el más rápido? Responde en el cuestionario: pregunta 15.

Se podría dar el caso de que el nivel de optimización `-O3` no fuese más rápido que el `-O1`. Estudiar con detalle las razones por las que podría ocurrir esa situación requeriría un análisis detallado del código generado por GCC que escapa del alcance de esta práctica. Sin embargo, y solo para ver los complejos modos en los que influye la arquitectura en el rendimiento, se va exponer una posible explicación: el nivel `-O3` aplica optimizaciones que, en principio, suponen una mejora en el tiempo de ejecución pero incrementan el tamaño del ejecutable. Una consecuencia de esto

es que es posible que código (por ejemplo, un bucle) que antes cabía dentro de la memoria caché del procesador ahora no quepa y haya que ir a buscarlo a un nivel superior, lo que, como se estudiará en el tema de memoria, supone una penalización muy alta. En los ejercicios adicionales de esta práctica se estudia más este problema.

Ten en cuenta que los resultados que has obtenido en estas prácticas no reflejan necesariamente lo que vas a observar en los programas reales que, en general, serán mucho más complejos. Lo más importante que debes haber aprendido es que el proceso de optimización no es sencillo porque, debido a las diversas capas de abstracción que hay entre el alto nivel y la arquitectura que ejecuta los programas, se producen a veces resultados no intuitivos y, por lo tanto, es fundamental realizar mediciones y emplear herramientas como los *profilers* y las optimizaciones del compilador, pero no confiando ciegamente en ellas.

2.4. Influencia de la optimización del compilador en el *profiler*

Como se ha señalado anteriormente, las optimizaciones del compilador pueden afectar al proceso de *profiling*. Vamos a comprobarlo:

- Compila `prog1` con el máximo nivel de optimizaciones y el código del *profiler*:

```
$> gcc prog1.c -O3 -pg -o prog1
```

- Ejecuta el programa para que se genere `gmon.out`:

```
$> ./prog1
```

- Genera el perfil:

```
$> gprof ./prog1 > informe.txt
```

- Edita el archivo `informe.txt`. ¿Cuántas funciones aparecen? Responde en el cuestionario: pregunta 16. Apunta el valor en la celda `B30`. Esto es porque una de las optimizaciones que ha realizado el compilador consiste en transformar el código de una función en código que se pone directamente donde se llama la función, evitando así realizar un `call`.

- Repite el mismo proceso utilizando el nivel `-O1`. ¿Cuántas funciones aparecen? Responde en el cuestionario: pregunta 17. Apunta el valor en la celda `B31`. Analiza si puedes ver en el perfil todas las funciones importantes.

Archivos de la práctica

- Copia la hoja de cálculo a Linux, añádela al repositorio con `git add Tema1.xls`, confirma los cambios con y `git commit` y súbelos al servidor con `git push`.

Ejercicios

- GCC ofrece un nivel de optimización indicado por el parámetro `-Os` que prioriza generar código optimizado sin incrementar el tamaño del código objeto generado. Para ello, aplica las mismas optimizaciones que en `-O2` excepto aquellas que incrementen el tamaño del código objeto. Prueba `prog1` con esta opción. ¿Obtienes mejor rendimiento que con `-O1`? ¿Y que con `-O2`? Compara el tamaño de los ejecutables obtenidos con las distintas opciones de optimización. ¿Cuál es el mayor? ¿Y el más pequeño?

- Como se ha comentado, el nivel de optimizaciones elegido hace variar, además del tiempo de ejecución del programa compilado, el tiempo de compilación, es decir, el tiempo que necesita el compilador para hacer su tarea. Para comprobarlo, mide el tiempo de compilación de `prog1` sin optimizaciones y con los tres niveles de optimización. ¿Cuál es la ganancia de compilar sin optimizaciones con respecto a cada nivel de optimización?

- El operador `&&` de C realiza una evaluación perezosa, lo que significa que evalúa las sub-expresiones que forman parte de la expresión en orden secuencial y, en cuanto encuentra una que es falsa, ya no evalúa el resto porque el resultado será falso independientemente del valor de las sub-expresiones que queden por evaluar. Esto es, en principio, una optimización, aunque teniendo en cuenta las complejidades de las arquitecturas actuales, que intentan ejecutar instrucciones por adelantado y si tienen que cancelar su ejecución sufren una penalización, puede no resultar en un rendimiento mejor.
  En el programa `prog2`, el tiempo de ejecución está dominado por la función `contarPrimosComunes`, que tiene una sentencia condicional que llama a dos funciones: `estaEnVector` y `esPrimo`. Usando un *profiler* determina si cambiar el orden de las llamadas dentro de la expresión cambia el número de llamadas a cada función y si cambia el rendimiento del programa.

- A veces los *profilers* dan información incorrecta. Lee el artículo *Evaluating the accuracy of Java profilers*, donde explican cómo utilizaron cuatro *profilers* distintos sobre un mismo código y sólo dos de ellos señalaron la misma función como la que más tiempo consumía, lo que demuestra que al menos dos estaban dando datos incorrectos (y es posible que los cuatro estuvieran equivocados).

1. Puedes conocer con detalle las opciones de optimización que ofrece GCC ejecutando la orden `man gcc`.