

# Trabajo práctico 2: Profiling y Optimización

Alejandro García Marra, *Padrón Nro. 91.516*  
alemarra@gmail.com

Sebastián Javier Bogado, *Padrón Nro. 91.707*  
sebastian.j.bogado@gmail.com

Grupo Nro. 0 - 2do. Cuatrimestre de 2012

66.20 Organización de Computadoras

Facultad de Ingeniería, Universidad de Buenos Aires

**Resumen**

## 1. Introducción

Muchas veces tanto para programas recién terminados, como para aquellos que llevan un tiempo en funcionamiento, se desconoce realmente qué partes del programa insumen la mayor cantidad de recursos, sean estos de tiempo, carga de cpu, etc. Poseer esta información se torna en algo crítico cuando se busca realizar una mejora de performance en dicho programa. Sería poco útil intentar optimizar a ciegas, por no decir inútil.

Haremos uso entonces de dos métodos distintos en el estudio del programa, el profiling (por medio de *gprof* y *cachegrind*) y la medición de los tiempos de ejecución (por medio de *time*).

### 1.1. Profiling

Se denomina así al análisis dinámico de un programa, con el fin de estudiar su comportamiento.

Al recolectar información en tiempo de ejecución, puede utilizarse en aquellos programas demasiado grandes o complejos, donde un análisis por lectura de fuentes sería impracticable.

Como consecuencia del análisis durante la ejecución, los datos con los que se corra el programa afectaran el resultado del profiler. Es decir, distintos datos de entrada pueden provocar distintas ramas de ejecución, dando por resultado que, por ejemplo, no se llamen algunas funciones.

- **gprof:** Permite aprender donde el programa pasa la mayor parte de su tiempo, y cuales funciones llaman a otras mientras se ejecuta.  
Esta informacion puede mostrar qué piezas del programa son mas lentas de lo esperado, convirtiéndolas en candidatas para su reescritura en la etapa de optimización.  
También puede ayudarnos a descubrir cuales funciones son llamadas más o menos veces de lo esperado, pudiendo encontrar nuevos bugs (aunque el descubrimiento de bugs no es el fin principal de esta etapa)
- **cachegrind:** Simula el comportamiento del programa sobre una determinada jerarquía de cache, la cual puede ser establecida por medio de distintas opciones. Como resultado, se obtiene una visión muy precisa de la cantidad de referencias a elementos del cache de instrucciones y al cache de datos, la cantidad de misses para ambos y el miss rate correspondiente. En particular nos interesan los resultados para la cache de datos D1, y el miss rate de la misma.

### 1.2. Medición de Tiempos

Permite conocer con precisión los tiempos de ejecución de un programa, discriminados entre tiempos de sistema, de usuario, tiempos totales, etc., así como también conocer los porcentajes para cada parte del programa, cantidad de entradas, y muchas otras opciones.

La combinación con una herramienta de profiling permite exactitud a la hora de conocer la forma en que se ejecuta el programa bajo estudio, permitiendo optimizar únicamente las partes críticas del ciclo de ejecución.

## 2. Flujo del programa

Se trata de una versión en lenguaje C de la simulación del planeta WATOR. El programa recibe un nombre de archivo en el que se van dejando las cantidades de peces y tiburones en cada turno, y simula 1000 turnos en un planeta de 32x32 celdas.

Comienza por la inicialización de la matriz, recorriendo la misma en su totalidad y ubicando de forma aleatoria espacios vacíos, peces o tiburones. Luego, se muestra completa por pantalla, acción que se realiza en cada uno de los ciclos.

Una vez completada la inicialización, comienza el ciclo de simulación. Por cada ciclo se busca mover todos los elementos no vacíos de la matriz. El comportamiento para peces o tiburones es distinto, por lo que en cada caso se evalúa el curso de acción, dependiendo también de los elementos que rodean la posición actual.

Cada movimiento depende del cálculo de la nueva posición, reemplazando el elemento previo si es necesario y teniendo en cuenta la tasa de natalidad y mortalidad de cada uno de los factores.

## 3. Hipótesis y Aclaraciones

- El objetivo principal es lograr un programa más eficiente, por lo que es inevitable realizar sacrificios respecto de la legibilidad del código y la flexibilidad del mismo. Muchas de las modificaciones realizadas irían en contra de las buenas prácticas de programación utilizadas en un caso normal.
- Todas las mediciones se realizan redireccionando la salida por pantalla a `devnull`. Esto nos permite ahorrar tiempo en las pruebas sin afectar las mediciones, ya que el tiempo de impresión se puede considerar constante para todas las mediciones.
- Consideramos las dimensiones de la matriz (como indica el enunciado, 32x32) como un elemento invariante. Esto nos permite, como veremos más adelante, realizar optimizaciones interesantes que de otra forma no serían posibles.
- Todas las mediciones que resulten de un promedio de corridas serán acompañadas del número de corridas correspondiente, así como el máximo y mínimo de la serie.
- La compilación de los códigos fuente se realiza con el comando:

```
gcc -DNDEBUG fuente.c -o fuente
```

En ningún momento (a menos que aclare lo contrario) se utilizan las opciones de optimización del gcc. El único caso donde se modifica este comando es para la compilación previa a la corrida con `gprof`.

- Los scripts con los que se realizaron las mediciones promediadas se encuentran disponibles en el cd bajo el nombre XXXXCollect.sh

## 4. Comparación de Versiones

Todas las modificaciones se expresan en relación con la versión anterior. Los cambios enunciados son acumulativos.

### 0. Versión original dada por la cátedra.

#### 1. Cambio en la definición del struct *animal*, reemplazando los tipo *int* por tipo *char*.

Reduce considerablemente el tamaño de cada elemento de la matriz, y por ende, el tamaño total de la misma. Esto permite que dentro de un mismo bloque de cache se almacenen mas posiciones, reduciendo el missrate.

#### 2. Modificar la función *moveall*, quitando instrucciones innecesarias y redundantes como las asignaciones a *todo*.

Se pasa de tener dos ciclos distintos a uno único, así como también se reducen las lecturas y escrituras de memoria. En ningún momento del programa se utilizaban los campos *todo* del struct

#### 3. Desenrollar el for de la función *choose*

Optimización recomendada en la bibliografía consultada [6][7]

**Removidas las variables auxiliares *npi* y *npj***

**Sacada afuera del loop la declaración de la variable *t***

Ahorran accesos a memoria

**Cambiados los *var++* por *++var***

Reduce la cantidad de instrucciones necesarias para la misma operación

**Las comparaciones del estilo (*var1 op var2* (ej  $i > j$ )) fueron reemplazadas por (*var1 - var2 op 0* ( $i - j > 0$ ))**

Es más eficiente comparar contra 0, según la bibliografía consultada

**Funciones *move\_to\_empty* y *move\_to\_fish* agregado un struct *animal\** para evitar las varias llamadas a *wator[npi][npj]***

Al guardar la dirección con el puntero, se evita recalcularla cada vez a partir de los índices de la matriz

#### 4. Reemplazar los accesos de tipo *wator[i][j]* en los ciclos por aritmética de punteros. Se guarda un puntero a la primer posición, accediendo por la suma ( $v + row + j$ ), donde *row* es un acumulador equivalente a $i * 32$ (*show\_wator*, *init\_wator*, *move\_all*)

Las direcciones que antes se computaban con una doble desreferencia, ahora se obtienen por una suma simple que se realiza sobre la propia variable.

**Aprovechando el desenrollado del for realizado en la versión anterior, reemplazamos el llamado a *ni()*, *nj()* por la expresión resultante si se evaluase el switch**

Originalmente no se conocía el valor de `dir` en el momento de invocar `ni()` o `nj()`, por lo que era necesario pasar por el `switch`. Ahora, como sabemos de antemano el valor que va a tomar esa expresión, nos podemos ahorrar el llamado a las funciones, así como también los `branch` del `switch`.

5. **Reemplazo en *choose\_fish()*, *choose\_empty()* el cómputo del módulo % MAXI y % MAXJ por la operación bitwise equivalente & 0x1F**

Como por hipótesis,  $\text{MAXI} = \text{MAXJ} = 32$ , la operación módulo puede realizarse de una manera mucho menos costosa a través de la operación `and` bitwise con  $(\text{MAXI}-1)$ . Esto solo sirve cuando  $\text{MAXI}$  y  $\text{MAXJ}$  son potencias de 2, y en particular, como también nos ahorramos la resta al forzar un 31 (0x1F), sólo sirve para  $\text{MAXI}=\text{MAXJ}=32$

6. **Declaración de variables como `register`**

En caso de que el compilador haga caso de la sugerencia, se ahorrarán todos los accesos a memoria relacionados con la lectura y escritura de la variable en cuestión.

**Reemplazo de % MAXI en lugares faltantes**

7. **Pasaje de puntero en vez de recalcular `w[i][j]` en distintas funciones**

**Incremento de puntero directo en vez de sumar  $row + j$**

Evaluamos que incrementar el puntero en cada ciclo del ciclo interno era más eficiente que realizar la suma en cada acceso, siendo el resultado el mismo.

**Cambio de la condición `if (x) return a; return b` por la versión equivalente `return x ? a : b ;`**

La idea surge de leer el manual del gcc y sus optimizaciones [6], donde propone el cambio para lograr una escritura incondicional. En este caso, se espera ver el efecto de un `branch` incondicional, o al menos explicitarle al compilador que la siguiente instrucción será un `branch` siempre.

8. **Cambio llamadas *myrand()* por la forma `inline rand() % max`**

9. **Reemplazados los llamados a `ni` y `nj` por un vector de resultados calculados**

La intención es reducir el overhead provocado por el constante llamado a las funciones en lo relacionado a la memoria y el stack, al utilizar valores pre calculados

**En *choose\_fish()*, *choose\_empty()*, cambiados  $\text{MAXI}|\text{J}-1$  por 31**

## 5. Corridas de prueba y Mediciones

## 6. Análisis de los Datos

A continuación, una tabla que muestra el speedup obtenido entre las distintas versiones, parcial y acumulativo

Versión	Time Promedio*	SU Parcial	SU Acumulado
wator	0.1151		
1wator	0.1149		
2wator	0.1104		
3wator	0.1100		
4wator	0.0834		
5wator	0.0792		
6wator	0.0729		
7wator	0.0685		
8wator	0.0694		
9wator	0.0654		

\* Sobre 1000 ejecuciones

Se puede ver que hubo varias mejoras que no afectaron mucho el tiempo de ejecución del programa, como las versiones x, y z. A su vez, la(s) versión X (Y) aportaron un x % al speed up. En esas versiones, los cambios fueron -blah-, y explicar con más detalle por qué aportó tanto.

Ale, asumo que acá no irán datos de compilación con optimizaciones.

## 7. Conclusiones

Del análisis de los datos se desprende que obtuvimos un speed up de hasta X, luego de muchas ideas aplicadas. Sin embargo, al compilar el programa original con optimizaciones, se obtienen tiempos aún mejores, que implicarían muchas más horas-hombre sobre el trabajo.

El compilador, al optimizar, reordena las instrucciones Assembly que no dependen entre sí, aprovechando el pipeline y los ciclos de *stall*. Retomando un poco las conclusiones del trabajo anterior, se ve de nuevo que no tendría sentido ponerse a trabajar a tan bajo nivel por el esfuerzo que demanda, y porque ya existen herramientas cuya función principal es producir un código objeto muy eficiente, y ni siquiera es necesario mencionar todos los otros cambios que aplica para mejorar el resultado de la compilación.

Uno de los problemas al intentar mejorar el MR fue que el núcleo de lógica del programa era la aleatoriedad misma. Mmm... podría poner el dibujito de la caché y la matriz? mencionando que si siempre da NORTH, va a estar cargado en caché, pero si salen intercalados NORTH y SOUTH, el MR es inmejorable porque en cada lectura, se pide un bloque que reemplaza otro que también era útil

## Referencias

- [1] J. L. Hennessy and D. A. Patterson, "Computer Architecture. A Quantitative Approach," 3ra Edición, Morgan Kaufmann Publishers, 2000.
- [2] "Sharks and fish wage an ecological war on toroidal planet WATOR", A.K.Dewdney, Scientific American, [http://home.cc.gatech.edu/biocs1/uploads/2/wator\\_dewdney.pdf](http://home.cc.gatech.edu/biocs1/uploads/2/wator_dewdney.pdf).



- [3] WA-TOR, Wikipedia, <http://en.wikipedia.org/wiki/Wa-Tor>.
- [4] GNU profiler, <http://sourceware.org/binutils/>.
- [5] Cachegrind, <http://valgrind.org/docs/>.
- [6] GCC Options that Control Optimization -  
<http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>
- [7] Steve McConnell, "Code Complete" 2nd Edition 2004 - Chapter 26 "Code-Tuning Techniques"