Bisoños Usuarios de GNU/Linux de Mallorca y Alrededores | Bergantells Usuaris de GNU/Linux de Mallorca i Afegitons

Introducción al OProfile (15158 lectures)

Per **Domingo Fiesta Segura**, ***C2H5OH*** (http://www.krenel.net)

Creado el 05/10/2005 06:40 modificado el 05/10/2005 06:40

El OProfile(1) es una herramienta para hacer profiling con una serie de características que lo diferencian del GNU gprof(2). Si tienes curiosidad o un momento de aburrimiento, sigue leyendo.

Índice

1. Mmm, profiling... ¿mande?

La traducción más próxima a este tecnicismo sería *perfilar*, dado que consiste en obtener un perfil de ejecución de nuestros programas que nos indique en qué partes se está utilizando la mayoría del tiempo. Una actividad muy recomendable a realizar **antes** de ir ciegamente a optimizar un programa pues este *perfilamiento*[1] nos puede dar pistas de, en el caso de que sea necesario optimizar código, **dónde** serán más notorias las mejoras.

Para obtener esta información de *profiling* lo que se suele hacer es interrumpir la ejecución del programa de forma periódica y, en cada *sample*, recolectar información diversa para analizar *a posteriori*. Se pueden tomar, a mi conocer, dos alternativas:

1. Instrumentalizar el programa para que él mismo genere esta información a la vez que se ejecuta. Por ejemplo, la clásica combinación GCC(12) con parámetro `-pg` y gprof(2).
2. Interrumpir la ejecución periódicamente mediante mecanismos externos (normalmente hardware y kernel) para ir tomando muestras del estado de la ejecución. OProfile(1) lo hace así.

1.1. Ventajas y desventajas del método clásico

Ninguna de las dos opciones es mejor que la otra en todos los casos. Utilizar gprof(2) es muy cómodo cuando sólo queremos buscar las funciones que mayor porcentaje de ejecución se llevan y disponemos del código fuente, tiempo y ganas para compilar el programa[2].

Pero gprof(2) no da una información de tiempo y recursos (user, system, elapsed) demasiado fiable. Esto es culpa de la escala de tiempo que utiliza; la comprobación fácil es medir los tiempos de un programa instrumentalizado con el comando time(1)(13) y compararlo con los resultados que muestra gprof(2).

1.2. Cuándo usar OProfile

OProfile(1) no es tan cómodo de usar como gprof(2), pero es mucho más potente en algunos aspectos. Por ejemplo, OProfile(1) es capaz de perfilar binarios **NO** instrumentalizados y librerías dinámicas. Es más, incluso puede utilizarse para analizar la ejecución del propio kernel Linux.

Además, no sólo es capaz de obtener información acerca de consumo de CPU sino también puede sacar otras estadísticas como fallos en algunos niveles de cache[3], predicciones de salto acertadas, instrucciones SIMD ejecutadas, fallos del TLB, etc. Pero cuidado, las posibilidades varían de un procesador a otro como veremos más adelante.

2. Cómo funciona OProfile

Algunos procesadores modernos incluyen contadores hardware del rendimiento que se van incrementando a medida que ocurren eventos específicos en la CPU. OProfile(1) aprovecha estos contadores para indicar a la CPU qué eventos quiere monitorizar y a qué intervalos. Cuando un contador alcanza un cierto valor la CPU interrumple la ejecución actual para que OProfile(1) pueda guardar la información de ese mismo *sample*.

Pongamos un ejemplo. Digamos que configuramos OProfile(1) para que vaya controlando los ciclos de reloj que transcurren en la ejecución y le decimos que tome muestras cada 500000 ciclos. Aquí cada evento es un ciclo de reloj, con lo que el contador se incrementa a cada ciclo de ejecución de la CPU. Cuando el contador alcance el valor 500000 se interrumpla la ejecución actual para dar paso al profiler; este recolecta toda la información que necesita, resetea el contador de ciclos y reanuda la ejecución en el punto en que se interrumpió. Más adelante veremos cómo configurar el tamaño del *sample* para cada tipo de evento, nótese ahora que cuando aumenta el número que indicamos al contador perdemos precisión; pero, para este caso en concreto, no deberíamos establecer valores muy bajos[4] porque se estaría interrumpiendo la ejecución cada pocos ciclos con lo que el sistema se vendría abajo.

En realidad, el tema de los contadores no funciona exactamente así pero es la mejor forma de entenderlo para poder usar la información OProfile(1) más provechosamente.

Dado que OProfile(1) **depende** totalmente del soporte hardware que haya debajo, las posibilidades de monitorización varían mucho de un procesador a otro. Esto también limita la portabilidad. En este enlace(14) se puede encontrar un listado de los procesadores soportados.

2.1. Eventos

Hay un concepto importante que no se ha aclarado del todo: los **eventos**. Un evento está asociado a un recurso hardware del que se puede obtener información referente a la ejecución y su rendimiento. Cada evento puede englobar distintos contadores que se activan o desactivan mediante una máscara. OProfile(1) es capaz de rastrear varios contadores de un mismo evento e incluso varios eventos distintos a la vez hasta un cierto límite que varía en función del procesador sobre el que se esté trabajando.

3. Manos a la obra

Ahora paso a explicar rápidamente los pasos a realizar para configurar OProfile(1) y cómo hacer el profiling. Para simplificar boy a limitar nuestro entorno de trabajo a Debian GNU/Linux(15) con un kernel de la rama 2.6. Para otras distribuciones lo único que cambia es la instalación del OProfile(1).

Utilizar OProfile(1) no es difícil pero sí **tedioso** para algunas tareas debido al modelo de trabajo que tiene.

3.1. Configuración del kernel

OProfile(1) se compone de varias partes. Una de ellas es un módulo del kernel que permiten a las otras componentes acceder a ciertos privilegios de la CPU. Con la llegada del kernel 2.6 el soporte para OProfile(1) ya viene de serie en los kernels *vanilla*. Con esta nueva rama del kernel también han sido soportadas nuevas arquitecturas. Aunque hay otras (IA64) que únicament están soportadas en la rama 2.4. Si tu kernel es 2.4 es necesario compilar un módulo aparte. Yo sólo explicaré cómo utilizarlo con un kernel 2.6 porque es la única rama en la que lo he probado.

Pues con cualquiera de los sabores de configuración, asegúrate de tener activada la opción de soporte para profiling del kernel:

```
#
# Profiling support
#
CONFIG_PROFILING=y
CONFIG_OPROFILE=m
```

Si utilizáis `make menuconfig`, basta con ir al submenú *Profiling support* en el propio menú general y seleccionar:

```
[*] Profiling support (EXPERIMENTAL)
<M>   Oprofile system profiling (EXPERIMENTAL)
```

Compilarlo como módulo o integrado es cosa tuya.

3.2. Instalación de las herramientas

La vida es fácil con Debian[15]:

```
etanol@om:~$ sudo apt-get install oprofile
```

3.3. Profiling, mayormente

Como ya he mencionado, OProfile[1] son varias componentes. Aparte del módulo del kernel hay un demonio y unas cuantas herramientas (binarios) de control y visualización. La carga/descarga del módulo así como el control del demonio se realizan mediante el comando `opcontrol`. Este comando necesia privilegios de `root` por lo que puedes ejecutarlo como superusuario, asignarle el bit SUID o bien utilizar sudo[16].

Para simplificar voy a suponer que se está ejecutando como usuario `root` en esta demostración, aunque el prompt indique lo contrario ;-)

Primero de todo hay que inicializar todo el tinglado, esto se hace con el comando:

```
etanol@om:~$ opcontrol --init
etanol@om:~$ opcontrol --reset
```

Esto carga el módulo del kernel y limpia las estadísticas de ejecuciones anteriores. En este estado ya podemos obtener información sobre los eventos soportados en la CPU que estamos trabajando gracias al comando `ophelp` sin argumentos. A continuación tenemos que configurar los eventos del procesador a capturar. Por ejemplo, tenemos un *Pentium IV* y queremos seguir la pista de los fallos en memoria cache de nivel 2:

```
etanol@om:~$ opcontrol --setup --no-vmlinux --event=BSQ_CACHE_REFERENCE:7500:0x100:0:1
```

El parámetro `--no-vmlinux` indica que **NO** queremos que tenga en cuenta el código del kernel. El parámetro `--event` activa el seguimiento de un evento en particular, en nuestro caso el nombre del evento es `BSQ_CACHE_REFERENCE`; esta información está disponible en `ophelp`. El número `7500` es el número de eventos que provocan una interrupción en la ejecución de la CPU, es decir, el tamaño del *sample*. El número `0x100` es la máscara que habilita los contadores a utilizar, si queremos activar más de un contador para este evento basta con calcular el **OR** de las máscaras que activan cada uno de los contadores; esto también está disponible en `ophelp` para cada evento. Los dos últimos números indican si queremos contar eventos en modo kernel (nivel de privilegio en la CPU) y en modo usuario respectivamente. En resumen, con el argumento anterior hemos indicado que OProfile[1] debe interrumpir la CPU cada 7500 fallos de cache de segundo nivel que se hayan producido en modo usuario. Tras esto vamos a empezar a perfilar :-P

```
etanol@om:~$ opcontrol --start
etanol@om:~$ ./mi_programa_a_investigar
etanol@om:~$ opcontrol --stop
```

En este punto ya tenemos información de profiling almacenada en algún lugar de nuestro disco duro. Ahora el abanico de posibilidades para visualizar la información se abre. Aquí hay dos comandos que nos interesan: `opreport` y `opannotate`. El primero da la información similar a la que da gprof[2][5]. Al segundo comando hay que indicarle qué fichero en particular debe anotar. Anotar consiste en mostrar información de profiling por línea de código en lugar de por función. Para que el `opannotate` funcione es necesario que el/los binario/s que queramos anotar haya/n sido compilado/s con los símbolos de *debug* (opción `-g` del GCC[12]).

No quiero enrrollarme mucho en cómo se usan los comandos de visualización de datos porque podréis encontrar ejemplos más directos aquí[17].

Una vez terminada la sesión de profiling, podemos recoger el chiringuito:

```
etanol@om:~$ opcontrol --shutdown
etanol@om:~$ opcontrol --deinit
```

Y esto ha sido todo por hoy, amiguitos del progresif. Si queréis saber más o tenéis dudas, ya sabéis: RTFM.

---

[1] En el resto del artículo usaré el término en inglés

[2] Hay programas que son poco afines ser recompilados: xorg, kde, gnome...

[3] Normalmente cualquiera **salvo** el nivel 1

[4] Muy bajo sería inferior a 75000

[5] Incluso puede mostrar los árboles de llamadas a funciones

---

Lista de enlaces de este artículo:

1. http://oprofile.sourceforge.net
2. http://sourceware.org/binutils/docs-2.16/gprof/index.html
3. http://bulma.net/body.phtml?nIdNoticia=2238&nIdPage=1#sec_1
4. http://bulma.net/body.phtml?nIdNoticia=2238&nIdPage=1#sec_1_1
5. http://bulma.net/body.phtml?nIdNoticia=2238&nIdPage=1#sec_1_2
6. http://bulma.net/body.phtml?nIdNoticia=2238&nIdPage=1#sec_2
7. http://bulma.net/body.phtml?nIdNoticia=2238&nIdPage=1#sec_2_1
8. http://bulma.net/body.phtml?nIdNoticia=2238&nIdPage=1#sec_3
9. http://bulma.net/body.phtml?nIdNoticia=2238&nIdPage=1#sec_3_1
10. http://bulma.net/body.phtml?nIdNoticia=2238&nIdPage=1#sec_3_2
11. http://bulma.net/body.phtml?nIdNoticia=2238&nIdPage=1#sec_3_3
12. http://gcc.gnu.org
13. http://unixhelp.ed.ac.uk/CGI/man-cgi?time
14. http://oprofile.sourceforge.net/docs/
15. http://www.debian.org
16. http://bulma.net/body.phtml?nIdNoticia=1779
17. http://oprofile.sourceforge.net/examples/

E-mail del autor: etanol _ARROBA_ krenel.net
Podrás encontrar este artículo e información adicional en: http://bulma.net/body.phtml?nIdNoticia=2238