Universidad De Málaga

E.T.S Ingeniería Infórmatica

Ingeniería de Computadores, 3A

PRÁCTICA 2. MEDICIÓN DEL RENDIMIENTO

Arquitectura de Computadores

Francisco Javier Cano Moreno

23 de Octubre de 2022

1. Ejecuta *perf list* y analiza que tipos de eventos proporciona.

La herramienta *perf* es una herramienta de Linux que nos permite analizar el rendimiento de un programa por medio de la línea de comandos. Para ello, la herramienta admite contadores de rendimiento hardware y software, puntos de rastreo y sondas dinámicas.
Uno de los subcomandos que podemos utilizar es *list*, que nos muestra una lista de todos los tipos de eventos que podemos ver:

- **Hardware**. Son eventos producidos por el procesador como, por ejemplo, el número de ciclos o los fallos de la caché.
- **Software**. Los produce el software y algunos ejemplos son los cambios de contexto o los fallos de página.
- **Caché**. Eventos producidos por el acceso a la caché como fallos en el primer nivel de ésta.

2. Ejecuta *perf stat –e cpu-cycles,instructions ./benchmark* y compara el tiempo proporcionado por perf con la salida del programa. ¿Hay diferencia? ¿por qué?

Otro de los subcomandos que tiene *perf* es *stat*. Este subcomando se encarga de hacer un recuento de eventos para un programa o un sistema durante un periodo de tiempo. En este caso, los eventos que queremos que nos muestre son los ciclos de cpu y el número de instrucciones totales y se lo indicamos en el comando con *-e*.

Una vez ejecutamos el comando, en primer lugar nos muestra su ejecución normal, la cual nos dice el tiempo que ha tardado en ejecutarse y, a continuación, nos muestra el número total de eventos que ha contado.

Por este motivo, el tiempo de ejecución es mayor, ya que tiene que llevar a acabo el recuento de todos los eventos que le hemos especificado, por lo que al final se podría decir que es el resultado del tiempo de ejecución normal más el tiempo de recuento.

3. Si ejecutas varias veces el programa, ¿da siempre el mismo número de eventos? Justifica la respuesta.

Tras varias ejecuciones, se puede observar que el número de eventos varía con cada ejecución. Esto se debe a varios motivos:

- **Organización de procesos del SO**. En función del número de procesos que haya y de la prioridad que tiene cada uno, el tiempo de ejecución variará ya que debe esperar a que se le ceda la CPU para poder seguir con la ejecución. Esto puede generar fallos de conflicto con los cambios de contexto (cambio de proceso en la CPU) y provocar que el número de instrucciones y ciclos aumente.
- **Fallos de caché**. Los ciclos necesarios para acceder a memoria pueden variar con cada acceso, por lo que, en función de los fallos que se produzcan, el número de ciclos y el tiempo de ejecución se verán afectados.

4. Utiliza ahora el comando *record* en lugar de *stat* y examina la salida usando *perf report*. Utiliza la opción annotate para relacionar los eventos con el código. ¿Qué instrucción es la que tiene mayor impacto en los eventos examinados?

La opción *record* permite guardar la información de los eventos ocurridos en la ejecución de un programa en un archivo llamado *perf.data*. Después esa información podemos analizarla con *perf report* y con *perf annotate*.
Ejecutamos *perf record -e cpu-cycles,instructions ./benchmark* y analizamos:

- Con *report* podemos ver que se han necesitado 447 ciclos de cpu y 447 instrucciones ejecutadas.
- Con *annotate*, obetenemos el código con el porcentaje de muestras que se han tomado en una instrucción concreta. En nuestro caso, el 30% de las muestras se han tomado en la instrucción mulss debido a que es la que más impacto tiene en la ejecución.

5. Cambia el valor de optimización del compilador, controlado con –O0 en OPT, y vuelve a obtener el número de eventos para cada opción (-O1, -O2, -O3). Rellena una tabla con los resultados obtenidos.

Tenemos las siguientes opciones de compilación:

- **-O0**. Esta opción es la que tiene por defecto y compila el código sin optimizarlo, por lo que genera muchos más eventos y tarda más en ejecutarse.
- **-O1**. En este caso, se mejora bastante la compilación debido a que optimiza un poco el código, reduciendo así el número de eventos pero aumentando el tiempo de compilación.
- **-O2**. Este es la opción que ofrece la optimización recomendada, ya que se tiene una mejor optimización que el caso anterior pero sin ser excesiva, lo que evita aumentar más el tiempo de compilación y reduce mucho el número de eventos.
- **-O3**. Esta opción de compilación solo se usaría en el caso de que tengamos un disco muy pequeño ya que optimiza muchísimo el código, lo que lleva a un tiempo de compilación y uso de memoria bastante alto.

Al ejecutar el comando para cada opción de compilación, obtenemos la siguiente tabla:

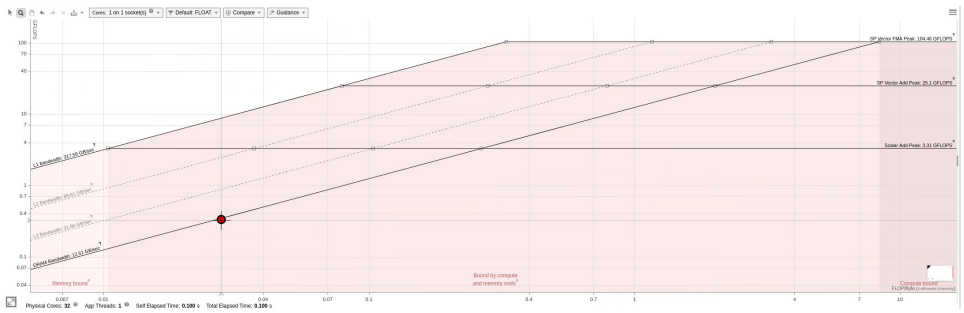
OPT	Instrucciones	Ciclos	Tiempo (s)
O0	487.384.046	231.380.997	0,1
O1	165.772.897	83.522.702	0,037
O2	132.152.222	78.000.824	0,035
O3	132.119.564	80.042.055	0’036

Table 1: Número de eventos para cada opción de compilación.

6. El modelo Roofline visto en clase permite describir el rendimiento de un procesador de forma un poco más precisa. Usa la herramienta Intel Advisor para calcular la gráfica del modelo Roofline y situar nuestra rutina dentro de esa gráfica.

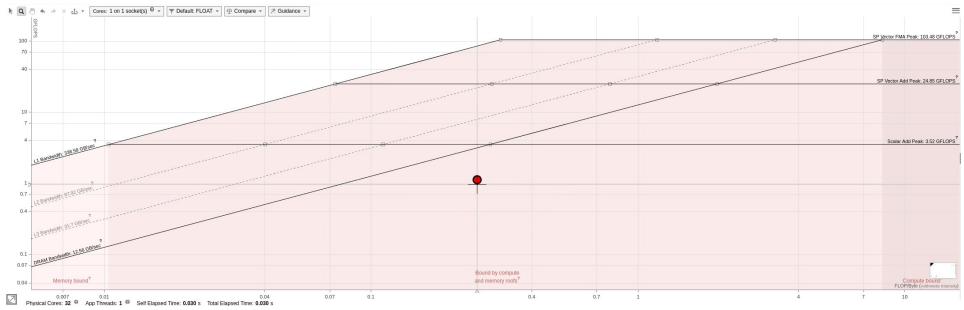
El modelo roofline nos permite ver como estamos aprovechando el ancho de banda de cada bus de memoria y la capacidad de computación del procesador. Así, podemos ver cómo podemos optimizar el código y sacarle el máximo provecho al hardware. Para ello, vamos a ver cómo es la gráfica para cada opción de compilación:

- En primer lugar tenemos O0:



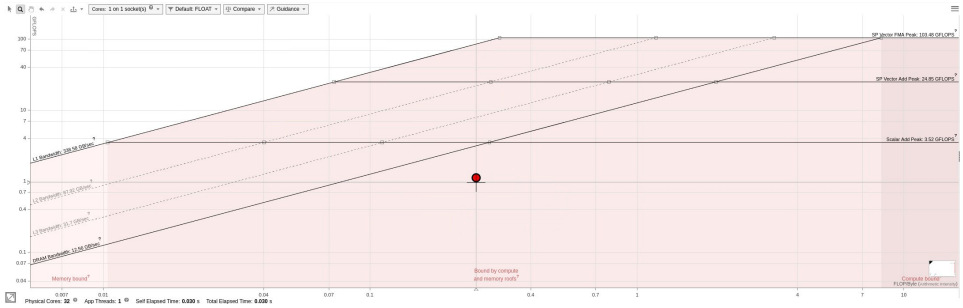
En la gráfica de la opción de compilación -O0 podemos ver que hay un problema de limitación de memoria, concretamente el techo de la DRAM. Este techo representa el ancho de banda máximo de la DRAM y, por lo tanto, nos dice que nuestro programa no tiene un manejo de datos que permita el uso de los niveles de la caché.

- En segundo lugar tenemos O1:



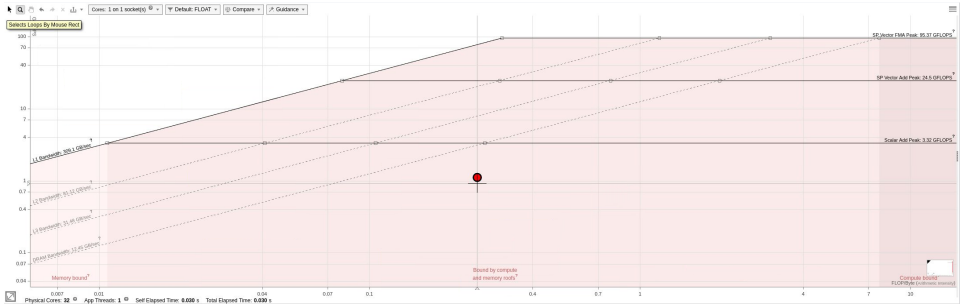
En este caso podemos ver que si subimos vertical, el techo sigue siendo una limitación de memoria debido a lo que hemos comentado en el aprtado anterior; pero justo pegado a él se cruza el techo de la limitación computacional, el Scalar Add Peak, que nos dice el número máximo de operaciones escalares de suma por segundo que se pueden procesar sin vectorizar. Por lo tanto, para poder pasar ese techo, se debe vectorizar el programa.

- En tercer lugar tenemos O2:



La compilación O2 obtiene un resultado muy parecido a la opción O1 debido a que O2 es una mejora de la optimización de O1, pero sigue sin ser vectorizada y sin manejar los datos para poder aprovechar el ancho de banda de la caché, lo que lleva a que tengan la misma limitación.

- Por último, obtenemos el roofline de la opción de compilación O3:



Con esta opción ocurre lo mismo que con O2. O3 optimiza un poco más el código para que ocupe lo mínimo en memoria, pero esta optimización no se aprecia en el roofline ya que no hay un cambio del manejo de los datos y tampoco se ha vectorizado, por lo tanto, el resultado que se obtiene es muy parecido.