Laboratorio 2:
Análisis de microarquitecturas

Objetivos

- Utilizar una herramienta de simulación que permita analizar la performance de un microprocesador con distintas microarquitecturas.
- Analizar cómo impactan en la performance de un microprocesador variaciones en los tamaños y características de caché y predictores de saltos.
- Comparar el rendimiento de un microprocesador con ejecución en orden respecto a uno fuera de orden.

Condiciones

- Realizar el trabajo práctico en grupos de **3 personas**.
- La fecha límite de entrega es el **martes 19 de noviembre** (inclusive) a través del github asignado. No se aceptan trabajos presentados después de esta fecha.
- Se verificará que todos los integrantes hayan realizado la misma cantidad de commits, aproximadamente.

Formato de entrega

- Debe entregarse mediante el repositorio de github, creándose un tag que indique la versión final.
- En el repositorio deben estar los archivos con los códigos en assembler, sin modificar los nombres de los archivos y las configuraciones de los distintos microprocesadores (en caso que se solicite). Además, en el archivo Informe.md se debe escribir el informe con las partes solicitadas en cada ejercicio.
- No está permitido compartir código o informes entre grupos.
- No está permitido subir el código en repositorios públicos.

Calificación

El ejercicio 1 es obligatorio. Su resolución debe estar aprobada para obtener la regularidad de la materia. La resolución del ejercicio 2 es opcional, siendo condición necesaria para acceder a la promoción de la materia. Aquellos que opten por la resolución de los 3 ejercicios completos obtendrán 1 punto adicional, que se suma automáticamente a la nota del segundo parcial.

DESARROLLO

La realización del presente laboratorio pretende que el alumno sea capaz de comparar y analizar el impacto de modificar características de la microarquitectura de un procesador en la velocidad de ejecución de distintos códigos. Para esto se utiliza el simulador **gem5,** el cual tiene una precisión de, al menos, el 5% en la determinación de la cantidad de ciclos de CLK consumidos en la ejecución de un código en procesadores ARM[(1)].

Para simplificar el proceso de instalación, se provee el simulador instalado en un contenedor Docker y scripts para ejecutar la simulación en forma sencilla. La estructura de archivos se

(1) Endo, Fernando A., Damien Couroussé, and Henri-Pierre Charles. "Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5." *2014 international conference on embedded computer systems: Architectures, modeling, and simulation (SAMOS XIV)*. IEEE, 2014.

basa en dos carpetas principales, la primera llamada "benchmarks", donde se encuentran tres templates donde resolver los tres ejercicios planteados. La segunda se llama "scripts", en la cual están los archivos necesarios para configurar y ejecutar la simulación. Dentro de esta carpeta, los archivos de configuración son:

- **cpu_config.toml**: es posible elegir características de la memoria principal y algunas globales del procesador.
- **in_order.py** y **out_of_order.py**: template con la configuración del predictor de salto y la caché.

Los comandos para correr la simulación y para editar las características se detallan en el archivo **README.me**.

EJERCICIO 1

Daxpy es la rutina principal del LINPACK Benchmark, utilizado para determinar la performance de microprocesadores, principalmente debido a su alta carga de cálculo de punto flotante. Sea *alpha* una constante, *X, Y* y *Z* vectores de longitud *N* (todos ellos expresados como valores en punto flotante), el cálculo se realiza como se muestra a continuación:

```
const int N;
double X[N], Y[N], Z[N], alpha;

for (int i = 0; i < N; ++i)
{
  Z[i] = alpha * X[i] + Y[i];
}
```

Considerando que N = 4096, se pide:

a) Escribir el código del microbenchmark daxpy en assembler ARMv8 y verificar su funcionamiento con qemu.
b) Correr la simulación para los siguientes tamaños de caché de datos: [8KB, 16KB, 32KB] de mapeo directo (1 vía) y obtener las siguientes métricas: número de ciclos (numCycles), ciclos ociosos (idleCycles), hits totales en la caché de datos (overallHits) y hits de lectura en la caché de datos (ReadReq.hits).
c) Repetir el punto b) pero para cachés asociativas por conjuntos de 2, 4 y 8 vías (parámetro “asoc” en el archivo in_order.py)
d) Graficar las métricas obtenidas en los puntos b) y c), realizar un análisis y justificar estos resultados respecto al código escrito en el punto a).
e) Reescribir el código utilizando técnicas estáticas de mejora, como loop unrolling, instrucciones condicionales, etc., para mejorar el rendimiento utilizando una caché de mapeo directo (1 vía) y el tamaño de 32KB para obtener rendimientos similares a los de 2 vías.
f) Ejecutar la simulación utilizando el procesador out-of-order con las características de la caché utilizada en el punto e) y comparar los resultados.

En el informe se deben incluir los gráficos del punto d) con el respectivo análisis, una breve explicación de las técnicas de mejoras aplicadas en el punto e) y una comparación de

resultados con los obtenidos en los puntos c) y d). Se deben entregar en adjunto los códigos assembler generados para los puntos a) y e).

EJERCICIO 2

Se pretende simular el flujo de calor en una placa de un material uniforme. Si bien este fenómeno es continuo, es usualmente resuelto mediante métodos de discretización, lo que genera grandes matrices a procesar. Considerar que se dispone de una placa cuadrada plana de un material uniforme (ej: una plancha de metal) y en algunos puntos de esta placa se aplica calor con una fuente de calor (ej: una llama) de modo de mantener una alta temperatura de manera constante en ese punto. Si bien la aplicación de calor es puntual, el calor tiende a distribuirse alrededor de la fuente y, luego de cierto tiempo, los lugares de la placa cercanos a la fuente estarán más calientes que los lejanos. En esta simulación modelamos como varía la temperatura en distintos puntos del material a lo largo del tiempo.

Como se mencionó anteriormente, la temperatura varía continuamente a lo largo de la placa, sin embargo, nuestro modelo va a ser discreto. Es decir, se divide la placa en una cantidad finita de áreas, donde mediremos la temperatura promedio en cada una de esas áreas. La división de la placa se hará en una grilla uniforme de *N*×*N* bloques, donde para cada área se almacenará la temperatura en un *double* conformándose una matriz de NxN elementos. La placa está inicialmente a temperatura ambiente (*t_amb*) lo que implica que debe ser inicializada a este valor. La temperatura de la fuente de calor es constante y por lo tanto, no debe ser modificada en el proceso de simulación. La temperatura de todos los demás puntos se calcula promediando las temperaturas de los puntos adyacentes a una posición *(x, y)* (arriba, abajo, izquierda, derecha), pero teniendo cuidado con los bordes. El borde no debe ser parte de la matriz de temperaturas. Por ejemplo, para *N* = 10, el cálculo de la temperatura para cada tipo de celda se realizaría de la siguiente manera :

- $t_{(0, 0)} = (t_{1,0} + t_{0,1} + t\_amb + t\_amb) / 4$
- $t_{(9, 3)} = (t_{8,3} + t_{9,2} + t_{9,4} + t\_amb) / 4$
- $t_{(6, 8)} = (t_{6,7} + t_{7,8} + t_{6,9} + t_{5,8}) / 4$

Con el fin de unificar las implementaciones del algoritmo de simulación física, en la siguiente página se muestra su codificación en C.

Considerando N = 64 y n_iter = 10, se pide:

a) A partir del código dado, reescribir el algoritmo de la simulación física en assembler ARMv8 y verificar su funcionamiento en qemu.
b) Ejecutar en gem5 la simulación considerando una caché de datos de mapeo directo de 32KB y predictor de saltos local (configurado por defecto).
c) Evaluar la cantidad de ciclos que toma su ejecución utilizando cachés asociativa por conjuntos de 2, 4 y 8 vías. Determinar en qué caso se obtiene la mejor performance y explicar por qué.
d) En este punto se pretende analizar la diferencia al usar dos predictores de saltos distintos: local y predictor por torneos (que está compuesto por un predictor local y uno global). En primer lugar se debe analizar el código e intentar deducir qué tipo de predictor (local o global) funcionará mejor en cada tipo de salto y cuánto podría

mejorar usar el de torneo. Correr el código con el predator local por defecto y obtener el miss rate calculado como:

condIncorrect / (condPredicted + condIncorrect)

Luego elegir el predictor por torneos (similar al utilizado en el procesador alpha 21264) y obtener nuevamente el miss rate. En ambos casos utilizar las características de la caché que obtuvo la mejor performance en el punto c). Analizar si los resultados se corresponden con lo esperado y justificar.

e) Ejecutar la simulación utilizando el procesador out-of-order con las características de la caché que obtuvo la mejor performance en el punto c) y un predictor de saltos por torneos. Comparar los resultados obtenidos con el punto d).

En el informe se deben incluir los resultados de los puntos b) al e), con las respectivas justificaciones. Entregar en adjunto el código de assembler generado en el punto a).

Codificación del algoritmo de simulación física:

```
const int n_iter, fc_x, fc_y;
float fc_temp,sum, x[N*N], x_tmp[N*N], t_amb;

// Esta parte inicializa la matriz, solo es necesaria para verificar el código
for (int i = 0; i < N*N; ++i)
    x[i] = t_amb;
x[fc_x*N+fc_y] = fc_temp;
// ——------------------------------------------------------------------------
for(int k = 0; k < n_iter; ++k) {
      for(int i = 0; i < N; ++i) {
                for(int j = 0; j < N; ++j) {
                        if((i*N+j) != (fc_x*N+fc_y)){
                                sum = 0;
                                if(i + 1 < N)
                                        sum = sum + x[(i+1)*N + j];
                                else
                                        sum = sum + t_amb;
                                if(i - 1 >= 0)
                                        sum = sum + x[(i-1)*N + j];
                                else
                                        sum = sum + t_amb;
                                if(j + 1 < N)
                                        sum = sum + x[i*N + j+1];
                                else
                                        sum = sum + t_amb;
                                if(j - 1 >= 0)
                                        sum = sum + x[i*N + j-1];
                                else
```

```
                                sum = sum + t_amb;
                        x_tmp[i*N + j] = sum / 4;
                        }
                }
        }
for (int i = 0; i < N*N; ++i)
        if(i != (fc_x*N+fc_y))
                x[i] = x_tmp[i];
}
```

EJERCICIO 3

Se tomará como caso de análisis para este ejercicio el método de ordenamiento por burbuja. Este algoritmo es ampliamente difundido por lo que no se incluirá un código en C con su implementación.

a) Escribir un código en assembler ARMv8 donde se implemente un algoritmo de ordenamiento por burbuja de un vector de N=1000 elementos (utilizar el vector aleatoriamente inicializado que se provee en el template). El ordenamiento debe ser ascendente (los números menores deben quedar al principio y los mayores al final del arreglo). Verificar su correcto funcionamiento en qemu.
b) Reescribir el código utilizando instrucciones condicionales (cset, csel, etc.) para evitar la ocurrencia de saltos y verificar su funcionamiento en qemu. Utilizar una caché de 32kB, de mapeo directo y un predictor de saltos por torneo.
c) Simular ambos códigos en gem5 y comparar su rendimiento. Analizar los resultados y explicar a qué se debe la variación de rendimiento. Utilizar una caché de 32kB, de mapeo directo y un predictor de saltos por torneo.
d) Simular ambos códigos nuevamente, en este caso utilizando un procesador out-of-order. Realizar un análisis de estos resultados.

En el informe se deben incluir los resultados de los puntos c) y d) y en archivos adjuntos los códigos generados en los puntos a) y b).