Proyecto I - False Sharing

1st Jonathan Guzmán Araya
Ingeniería en Computadores
Instituto Tecnológico de Costa Rica
Costa Rica
jonathan11@estudiantec.cr

2nd Gerald Mora Mora
Ingeniería en Computadores
Instituto Tecnológico de Costa Rica
Costa Rica
gemora@estudiantec.cr

***Resumen*—En este documento, se aborda el problema del "False Sharing.[en] sistemas multiprocesador y se presenta una propuesta de diseño para mitigar este problema en el contexto de la suma de filas de matrices. El "False Sharing.[es] un desafío crítico que puede degradar significativamente el rendimiento cuando múltiples hilos de ejecución acceden a variables o datos que comparten una línea de caché. Se discuten dos enfoques diferentes para resolver este problema: uno que utiliza múltiples hilos y otro que optimiza la utilización de la caché de la CPU. Se evalúa el desempeño de ambas soluciones utilizando herramientas de análisis y se presentan los resultados obtenidos. Este trabajo proporciona una visión profunda de cómo abordar y resolver problemas de "False Sharing.[en] aplicaciones multiproceso.**

***Index Terms*—CPU, False Sharing, hilos de ejecución, sistemas multiprocesador, caché, rendimiento, optimización.**

I. Propuesta de diseño

I-A. Suma de Filas de Matrices

El problema de la suma de filas de matrices consiste en calcular la suma de cada fila individual de una matriz bidimensional. Esto puede ser útil en diversas aplicaciones, como procesamiento de imágenes, análisis de datos y cálculos matriciales en general. La tarea se vuelve más desafiante cuando se trabaja con grandes conjuntos de datos y se busca optimizar el rendimiento.

En esta propuesta de diseño, abordaremos el problema y presentaremos una solución eficiente implementada en C y C++. Discutiremos las ventajas y desventajas de dos enfoques diferentes: uno que utiliza múltiples hilos para la suma de filas y otro que aprovecha la optimización de la caché para maximizar el rendimiento de la CPU.

- Evaluación 1: Suma de filas con hilos
  - Diseño
    1. Inicialización de la matriz: Se crea una matriz bidimensional de tamaño definido de 100x100, donde en cada uno de sus espacios almacenará el valor 1.
    2. Creación de hilos: Se crea un número de hilos igual al número de núcleos de filas de disponibles. Cada hilo ejecuta una función que calcula la suma de la fila asignada.
    3. Sincronización: Se utiliza sincronización para garantizar que todos los hilos completen sus cálculos antes de continuar.
    4. Suma Total: Se suma el resultado parcial de cada hilo para obtener la suma total de las filas.
  - Ventajas
    1. Utiliza eficazmente la capacidad de procesamiento de múltiples núcleos de CPU.
    2. Adecuado para matrices grandes y procesamiento paralelo.
    3. Puede mejorar significativamente el rendimiento en sistemas multiprocesador.
  - Desventajas
    1. Requiere sincronización entre hilos, por lo que puede agregar complejidad y riesgo de errores.
    2. No garantiza una mejor utilización de la caché de CPU, lo que puede generar más fallos de caché en comparación con otros enfoques.
- Evaluación 2: Suma de filas optimizada para la caché
  - Diseño
    1. Inicialización de la matriz: Se crea una matriz bidimensional de tamaño definido de 100x100, donde en cada uno de sus espacios almacenará el valor 1. Se da una alineación de 64 bytes que se utiliza para mejorar el rendimiento en sistemas con múltiples núcleos de CPU para reducir el efecto de falsa intercalación en el acceso a los datos de la matriz por parte de múltiples hilos.
    2. Creación de hilos: Se crea un número de hilos igual al número de núcleos de CPU disponibles. Cada hilo ejecuta una función que calcula la suma de la fila asignada.
    3. Sincronización: Se utiliza sincronización para garantizar que todos los hilos completen sus cálculos antes de continuar.
    4. Suma Total: Se suma el resultado parcial de cada hilo para obtener la suma total de las filas.
  - Ventajas
    1. Maximiza el uso eficiente de la caché de CPU, minimizando los fallos de caché.
    2. Puede ser más rápido para matrices pequeñas, medianas y grandes debido a la optimización de la caché.
  - Desventajas
    1. No aprovecha la capacidad de procesamiento de múltiples núcleos de CPU.

2. No garantiza una mejor utilización de la caché de CPU, lo que puede generar más fallos de caché en comparación con otros enfoques.

II. Especificación de componentes del sistema

II-A. BackEnd

Se tiene una serie de programas en C++ y C, los cuales contienen el código de solución al problema propuesto. Los mismos ya tienen integrado el benchmark, por lo cual, al correr el ejecutable se obtienen alguno de los resultados esperados. Por simplicidad, los nombres de los archivos se detallan a continuación:

- mSST: Matrix Sum Simple Thread
- mSFS: Matrix Sum False Sharing
- mSNFS1: Matrix Sum No False Sharing (Case 1)
- mSNFS2: Matrix Sum No False Sharing (Case 2)
- mSSTC: Matrix Sum Simple Thread (C code)
- mSFSC: Matrix Sum False Sharing (C code)
- mSNFS1C: Matrix Sum No False Sharing (Case 1, C code)
- mSNFS2C: Matrix Sum No False Sharing (Case 2, C code)

II-B. FrontEnd

Se creó una interfaz simple para los resultados obtenidos desde el código de C++ y otra interfaz para los resultados provenientes de la solución obtenida en C. La misma contiene una serie de gráficos como los de la Figura 1 en los cuales se muestran los programas ejecutados y sus resultados.

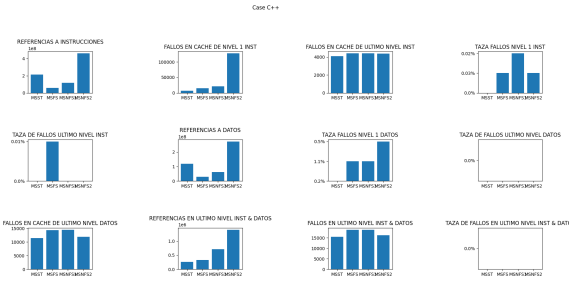

Figura 1: Interfaz gráfica para visualizar los resultados de C++ utilizando cachegrind

II-C. Benchmark

Se genera por cada compilación de los archivos de código mencionados anteriormente, un archivo TXT con algunos datos necesarios para analizar resultados. Estos archivos se pueden encontrar dentro de la carpeta raíz del proyecto con un nombre similar a este Program_BenchCPP_results.txt o Program_BenchC_results.txt.

II-D. Hardware detection code

Para esto se utiliza la herramienta de perfilado Cachegrind, que pertenece a Valgrind, esta herramienta es capaz de leer varios datos de hardware necesarios relacionados con la caché,

```
-------------------------------------------------------------------
Benchmark                         Time             CPU   Iterations
-------------------------------------------------------------------
SumRowsSingleThreadBenchmark    2965270 ns      2965099 ns          243
Compilando mSFS...
Guardando resultado de Benchmark para mSFS...
```

Figura 2: Ejemplo resultado benchmark utilizando C++

los tamaños de línea y la información de consumo de las mismas. Esta herramienta es muy poderosa y versátil, además de que es una herramienta multiplataforma que puede ser ejecutada en diferentes sistemas gracias a su portabilidad. Su facilidad de uso y su análisis detallado de la caché, proporciona una opción sólida y confiable para el análisis del rendimiento y comportamiento de la caché. Además, cabe destacar que esta herramienta es gratis y de código abierto, y que no va a necesitar que dispongas de un procesador de una marca exclusiva para ejecutar todas sus características como sucede con (Vtune de Intel). Otra herramienta que estamos utilizando para comparar resultados con los obtenidos por Cachegrind es Perf, cuenta con muchísimas de las características de la herramienta anterior, sus resultados son muy confiables y es verdaderamente fácil de utilizar.

III. Explicación de proceso del diseño

III-A. Definición del problema

El fenómeno conocido como ”False Sharing.[es] un desafío crítico en sistemas multiprocesador (MP) y se refiere a la degradación del rendimiento que ocurre cuando múltiples hilos de ejecución (threads) acceden a variables o datos que comparten una línea de caché, incluso si esos threads no están modificando las mismas variables. Esto ocurre debido a la naturaleza de cómo funcionan las cachés en los sistemas modernos.

Cuando dos o más threads acceden a variables que residen en la misma línea de caché, la caché debe mantener la coherencia de esas líneas de caché entre los núcleos o procesadores. Esto requiere un intercambio constante de datos entre los núcleos, incluso si no están modificando las mismas variables. Esto provoca una sobrecarga significativa en el sistema, lo que resulta en una degradación del rendimiento.

El problema de la suma de filas de matrices consiste en calcular la suma de cada fila individual de una matriz bastante grande bidimensional y con este esperamos obtener los resultados necesarios para evidenciar el False Sharing.

III-B. Idea, conceptualización y diseño inicial

Se piensa en la idea más básica de solución que sería recorrer cada fila y sumar sus valores y cuando se termina de recorrer toda, se sumarían sus resultados parciales. Ahora, esto en single thread suena fácil, pues es un ciclo sencillo que recorre la matriz y hará los cálculos, pero se debe pensar en mejorar el rendimiento y la utilización de los datos. Si se utiliza uno de los conceptos más comunes visto en nuestra carrera como lo son los hilos, se podría pensar en múltiples soluciones al problema, pero más allá de solo empezar a lanzar código sin pensar en las consecuencias, se debe ir un paso

más adelante y encontrar soluciones que eviten problemas que puedan a aparecer en medio desarrollo del código, como lo son el False Sharing o el Direct Sharing.

Para este caso aplicamos estos dos conceptos:

- Alineación de datos: Cuando se utilizan matrices debemos asegurarnos que estas esten alineadas correctamente, entonces podemos especificar la alineación deseada para estas matrices. Esto asegura que los elementos de la matriz se almacenen en ubicaciones de memoria que eviten compartir líneas de caché con otros datos. Por otro lado, cuando utilizamos estructuras estas pueden compartir datos entre subprocesos, debemos asegurarnos en estos casos de que estén alineadas en la memoria. Esto significa que los elementos de la estructura deben comenzar en posiciones de memoria que sean múltiplos de cierto valor, como 64 bytes.
- Padding: Para este método, si tenemos varias estructuras o variables que podrían estar sujetas al ”False Sharing”, lo importante es asegurarnos de que estén separadas por suficiente espacio en la memoria. Esto significa que debemos diseñar el código de manera que las estructuras y variables no compartan líneas de caché. Esto se puede lograr mediante la alineación y el relleno (padding).

IV. Evaluación de desempeño

En esta sección evaluaremos el rendimiento de nuestros programas diseñados para detectar y mitigar el problema del ”false sharing”. Para llevar a cabo esta evaluación, utilizamos dos herramientas de análisis: perf y Valgrind con su herramienta Cachegrind. Estas herramientas nos proporcionarán información detallada sobre el comportamiento de nuestros programas en términos de referencias a caché, fallos de caché, ciclos de CPU e instrucciones ejecutadas. A través de estas métricas, podremos medir el impacto del ”false sharingz evaluar la efectividad de las soluciones implementadas.

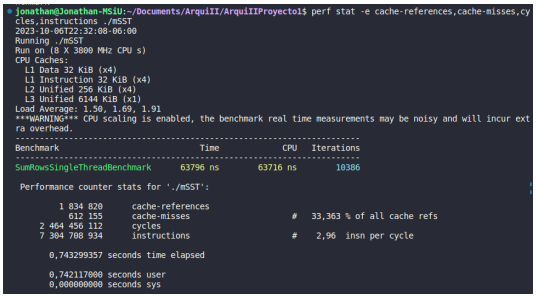

Figura 3: Análisis con Perf para SST

Entonces tenemos lo siguiente:

- Caso 1 - (mSST): Se puede observar de la Figura 3 que Este caso muestra un alto número de referencias a la caché y un porcentaje significativo de fallos en la caché (33,363 %). Además, la relación entre instrucciones y ciclos (2,96 insn por ciclo) sugiere que podría haber oportunidades de mejora en la eficiencia de la ejecución del programa.

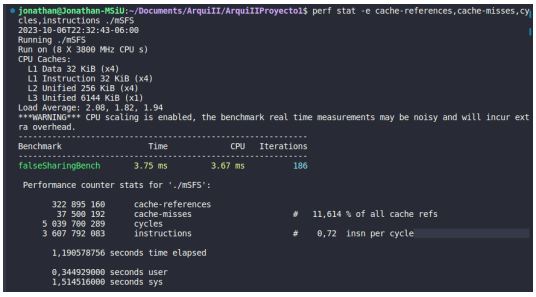

Figura 4: Análisis con Perf para SFS

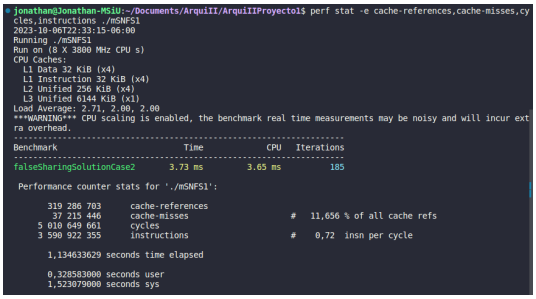

Figura 5: Análisis con Perf para SNFSC1

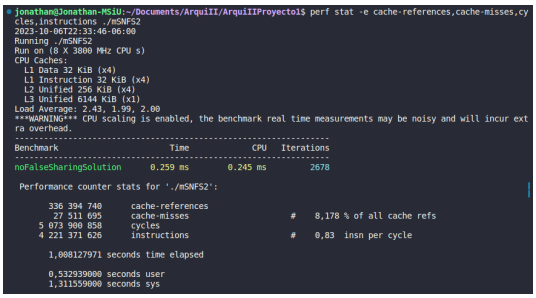

Figura 6: Análisis con Perf para SNFSC2

- Caso 2 - (mSFS): En este caso, se puede observar en la Figura 4 que el número de referencias a la caché es bastante alto, y el porcentaje de fallos en la caché es significativo. La relación entre instrucciones y ciclos (0,72 insn por ciclo) sugiere que puede haber ineficiencias en la ejecución debido a los fallos en la caché lo que se debe a la presencia de ”false sharing”.
- Caso 3 - (mSNFS1): En este caso, podemos notar en la Figura 5 que los resultados son similares al caso anterior, con un alto número de referencias a la caché y un porcentaje significativo de fallos en la caché. La relación entre instrucciones y ciclos (0,72 insn por ciclo) sugiere posibles ineficiencias relacionadas con la caché.
- Caso 4 - (mSNFS2): Este caso muestra se puede observar en la Figura 6 un menor número de fallos en la caché en comparación con los casos anteriores. La relación entre instrucciones y ciclos (0,83 insn por ciclo) sugiere una mejor eficiencia en la ejecución en comparación con los casos anteriores, lo que puede deberse a la ausencia de ”false sharing.en este programa.

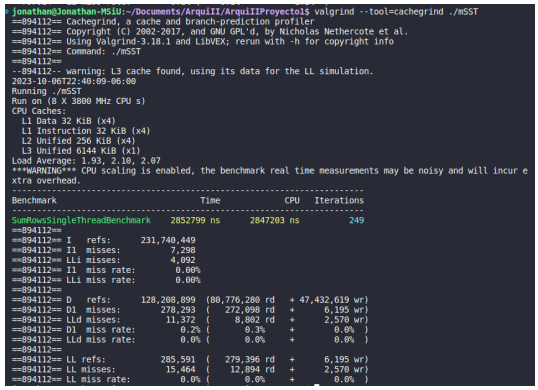

Figura 7: Analisis con Cachegrind para SST

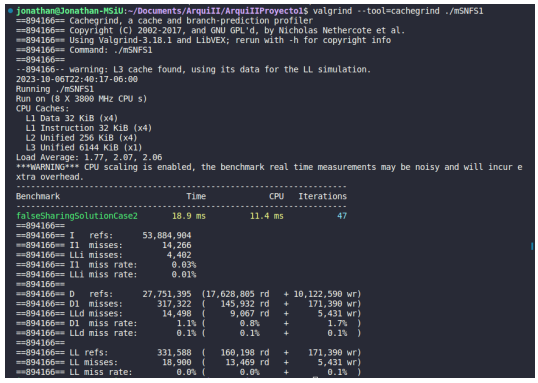

Figura 8: Analisis con Cachegrind para SFS

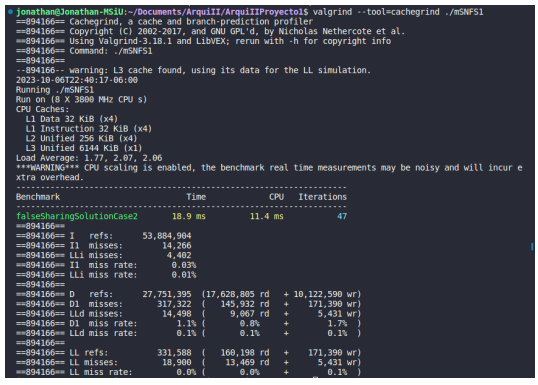

Figura 9: Analisis con Cachegrind para SNFSC1

Ahora realizando el análisis para cachegrind tenemos que:

- Caso 1 - (mSST): En este caso, podemos ver en la Figura **??** que hubo muy pocos fallos de caché de instrucción (casi ninguno), lo que sugiere un buen comportamiento de la caché de instrucción. También hubo pocos fallos de caché de datos en L1 y prácticamente ninguno en L3. Por ultimo, miss rate son las tasas de fallos de caché de datos en L1 y L3, respectivamente, y son muy bajas. Esto indica una alta eficiencia en el acceso a la caché de datos.
- Caso 2 - (mSFS): En este caso, la Figura 8 nos muestra que los resultados muestran un mayor número de fallos de caché en comparación con el caso 1. Los fallos de caché de instrucción y datos en L1 y L3 son más altos, y las tasas de fallos de caché de datos en L1 y L3 también son más altas. Esto indica que el programa mSFS experimenta un mayor impacto en la caché debido a posiblemente problemas de ”false sharing”. Los datos están siendo compartidos entre hilos de manera ineficiente, lo que resulta en un mayor número de fallos de caché.
- Caso 3 - (mSNFS1): Podemos observar en la Figura 9 que Los resultados en este caso son similares a los del caso 2, con un alto número de fallos de caché en comparación con el caso 1. Sin embargo, las tasas de fallos de caché de datos en L1 y L3 son un poco más bajas en comparación con el caso 2. Aunque en el codigo del programa intentamos solucionar el problema de ”false sharing.[en] comparación con el caso 2, todavía se observan impactos significativos en la caché debido a la compartición de datos ineficiente entre hilos.
- Caso 4 - (mSNFS2): En este caso, podemos evidenciar en la Figura 10 los resultados muestran una reducción significativa en el número de fallos de caché en comparación con el caso 2 y el caso 3. Los fallos de caché de instrucción y datos en L1 y L3 son mucho menores, y las tasas de fallos de caché de datos en L1 y L3 también son muy bajas. Esto indica que el código del programa mSNFS2 ha logrado evitar en gran medida el problema de ”false sharing”. Los datos se están compartiendo de manera eficiente entre hilos, lo que resulta en un rendimiento mucho mejor en términos de acceso a la caché.

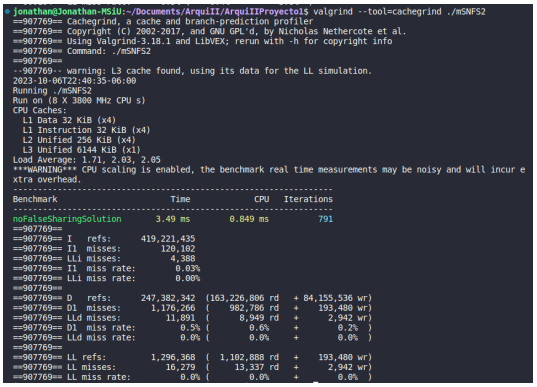

Figura 10: Análisis con Cachegrind para SNFSC2

IV-A. Reporte de resultados

Basado en los resultados del comportamiento, rendimiento y tiempos de ejecución, concluimos con los siguientes resultados generales: La suma de filas de matrices en un solo hilo es sumamente rápida en comparación a un par de los casos donde se esta utilizando múltiples hilos, esto puede ser debido a varias razones como por ejemplo una sobrecarga de hilos, en la que la administración de los mismos implica una penalización en el rendimiento, cosa que no sucede en este caso de single thread. También podría darse porque el false sharing presente en dos

de los casos propuestos afecta demasiado el rendimiento, o que simplemente haya un problema de competencia por el acceso a la caché por parte de los hilos. Por lo que, tener un solo hilo para resolver un problema de este tipo, no implica que el rendimiento sea peor.

En el caso de la suma de matrices con False Sharing, los tiempos de ejecución tienden a ser mayores, los misses de acceso a caché de instrucciones y de datos suelen ser más altos por los conflictos que tienen los hilos al intentar interactuar con una línea de caché al mismo tiempo. Por lo que es de esperarse que el rendimiento sea pésimo aunque disponga de hilos para manejar la tarea. Los problemas vistos en este caso que generan el false sharing son muy conocidos. El primero es que todos los hilos están leyendo la misma matriz, adicionalmente, todos los datos de la matriz están acomodados de una manera en la que los hilos eventualmente llegan a solicitar las mismas líneas de caché simultáneamente. No hay una distribución de filas para una cantidad óptima de hilos según la arquitectura del computador.

En el caso 1 de la suma de matrices sin False Sharing, en realidad se descubrió que de igual manera existe FS, esto porque lo que se intentó mitigar fue la escritura en un vector compartido pero aplicando un alineamiento según el tamaño de línea de caché, para evitar que cuando cada hilo escribiera, no intentara ocupar una línea de cache ya solicitada por otro hilo. Pero, lo que no se tomó en cuenta, es que al igual que en el caso anterior de False Sharing, se seguía administrando la misma matriz de datos, lo que sugiere lectura de la misma por todos los hilos al mismo tiempo. Estos motivos, son los que causan que aunque se intente resolver uno de los problemas de este caso, el no atacar todos los problemas de lectura y escritura en caché, puede causar que el rendimiento siga siendo pobre.

Por último, el caso 2 de la suma de matrices sin False Sharing, muestra un comportamiento óptimo y esperado, sus tiempos de ejecución muestran una gran ventaja respecto a los otros casos analizados, en varias ocasiones fue muy aproximado al tiempo de ejecución en Single Thread. Todo esto nos demuestra una mejora en rendimiento y uso apropiado de la capacidad multi-thread del procesador. En este caso de atacado el problema de raíz dividiendo la matriz en secciones según la cantidad de hilos, para que estos se encargaran de un conjunto de filas cada uno, adicionalmente se realizó un alineamiento entre los datos de la matriz, para que no exista posibilidad de que un hilo llegue a requerir la línea de caché que esté siendo ocupada por otro. Todo esto amortigua un montón los problemas por False Sharing y nos permite trabajar problemas grandes sin exigir mucho del computador y aprovechando al máximo sus características.

REFERENCIAS

[1] CoffeeBeforeArch, "False Sharing Tutorial."[En línea]. Disponible en: https://coffeebeforearch.github.io/2019/12/28/false-sharing-tutorial.html

[2] CodeGuru, "False Sharing in Multithreaded Programming and a Glance at the C++11 std::thread."[En línea]. Disponible en: https://www.codeguru.com/cplusplus/c-programming-false-sharing-in-multi-threaded-programming-and-a-glance-at-the-c0x-stdthread/

[3] Level Up, "False Sharing in Multithreaded Programming: Profiling and Analysis with Perf."[En línea]. Disponible en: https://levelup.gitconnected.com/false-sharing-in-multithreaded-programming-profiling-analysis-with-perf-ba2aff67a30b