MP Programming and False Sharing

1st Daniel Camacho
Ingeniería en Computadores
Cartago, Costa Rica
ddcamachog1501@estudiantec.cr

2nd Roy Acevedo Méndez
Ingeniería en Computadores
Cartago, Costa Rica
racevedo@estudiantec.cr

3rd Pablo Mora Rivas
Ingeniería en Computadores
San José, Costa Rica
mora.pablojavier@gmail.com

***Abstract*—In the context of multiprocessing, the phenomenon known as "false sharing" has emerged as a significant challenge in optimizing the performance of parallel systems. This study focuses on investigating the effects of false sharing in MP systems and the application of measures to mitigate its impact. Benchmarks were developed in multiple programming languages, including C and C++, to evaluate the behavior of false sharing in a variety of scenarios. Additionally, target hardware parameters, such as cache line size and number of cores, were determined to ensure the portability of the results. The automation process allowed benchmarks to be run multiple times and capture significant performance data. The results revealed the consequences of false sharing in MP systems, showing notable differences between single-threaded and multi-threaded executions. Additionally, effective measures were implemented to mitigate false sharing, resulting in substantial performance improvements. This study demonstrates the importance of understanding and addressing false sharing in multiprocessor systems and highlights the need to consider the characteristics of the target hardware when designing software solutions. Additionally, performance is compared using profiling tools, highlighting the usefulness of multiple approaches to analyze false sharing.**

***Index Terms*—Python, C, C++, Mult-threading, Multi-processing, Single-threading, Perf, Linux, Cache, Cache Misses, Valgrind, Automation.**

I. Introducción

En la era de la computación paralela y el procesamiento de alto rendimiento, la eficiencia y el rendimiento de los sistemas multiprocesadores (MP) son de vital importancia. Sin embargo, dichos sistemas multiprocesadores siempre se han enfrentado a desafíos técnicos que pueden limitar la capacidad para aprovechar al máximo los recursos de hardware disponibles. Es ahí donde donde aparace uno de estos desafíos críticos, el fenómeno conocido como "False Sharing".

En el área del multiprocesamiento, el False Sharing, es un fenómeno en el que multiples hilos o threads que se encuentran en ejecución en un sistema MP, llegan a compartir la misma línea de caché, esto a pesar de que las operaciones respectivas y variables no lelgana interactuar entre si. A simple vista esto puede parecer nada crítico, pero llega a tener un impacto grande en lo que sería el rendimiento de los sistemas paralelos. El problkema con esto es que cuando varios hilos llegan a compartir una línea de caché, cada vez que uno de los hilos realiza una modificación a su porción de datos, toda la linea de caché entra en un estado de invalidación y debe ser recargada, esto da como resultado una sobrecarga muy notoria en lo que a accesos a memoria respecta.

El principal objectivo de este proyecto es profundizar en el concepto de False Sharing, llegar a comprender el fenómeno como tal y entender el impacto que tiene en el desempéño de los sistemas multiprocesadores. Para poder cumplir con esto, se han diseñado una serie de benchmarks en ciertos lenguajes de programación, como lo son C y C++, esto para poder evaluar y medir los efectos del False Sharing en ciertos escenarios donde las especificaciones del hardware pueden inlfuir en dichos resultados.

Con respecto a las especificaciones del hardware, es sumamento importante el poder determinar dichos parámetros específicos del hardware objetivo o lo que viene siendo el sistema donde se ejecutarán los benchmarks, como lo son el tamaño de las líneas de caché, el número de cores o núcleos, número de threads y niveles presentes en la caché. Esto, con el fin de garantizar la portabilidad de los resultados que se obtuvieron. Dichos parámetros medidos, son de suma importancia debido a lo directo que se relacionan con el fenómeno de False Sharing y son críticos para analizar y comprender los diferentes resultados que se obtienen basados en los diferentes entornos de hardware.

Durante este proyecto y el estudio que conlleva, se implementarán medidas cuyo objetivos es el de mitigar, reducir o eliminar el impacto de False Sharing en los sitemas multiprocesadores. Para ello, se creará una etapa de automatización donde se podrán obtener los datos del sistema objetivo que servirán para poder adaptar dichos benchmarks, también para poder ejecutar los benchmarks múltiples veces, permitiendonos capturar los datos del rendimiento que variarán debido al impacto del False Sharing, así como el poder visualizar de manera gráfica los resultados tanto en single threading como multithreading y sus respectivas soluciones para mitifar el efecto.

En última instancia, esta investigación busca arrojar luz sobre el false sharing en sistemas multiprocesadores y proporcionar información valiosa para desarrolladores y diseñadores de sistemas paralelos. A través de un análisis exhaustivo de los efectos del false sharing y la implementación de medidas efectivas, se espera avanzar en la comprensión y la mitigación de este desafío crítico en el campo de la computación paralela.

II. Propuestas

A continuación se describirán las diferentes propuestas que se plantearon para la elaboración de este proyecto, así como las herramientas que se tomaron en cuenta para poder cumplir con cada uno de los objetivos.

A. Propuesta 1

Para esta propuesta de diseño y desarrollo, se contempló el uso de Rust como principal lenguaje de programación para confeccionar nuestras pruebas ya que es un lenguaje de programación moderno y seguro que se ha vuelto cada vez más popular en los últimos años debido a sus características únicas y su énfasis en la seguridad y el rendimiento. Además de que Rust ofrece un alto rendimiento al permitir un control fino sobre la memoria y la ejecución del programa. No tiene el costo de abstracción de alto nivel que algunos otros lenguajes tienen, lo que lo hace adecuado para sistemas y aplicaciones de alto rendimiento, como sistemas operativos, controladores de dispositivos y motores de juegos, sin olvidar que destaca por su enfoque en la seguridad de la memoria. Utiliza un sistema de control de propiedades de memoria que atrapa muchos errores de tiempo de compilación, como referencias nulas y desbordamientos de búfer. Esto hace que sea mucho más difícil introducir errores de seguridad comunes como fugas de memoria o corrupción de datos.

Como herramienta de perfilado se consideró el uso de VTune, la cual es una poderosa herramienta de perfilado y análisis de rendimiento desarrollada por Intel. Está diseñada para ayudar a los desarrolladores y optimizadores de software a entender y mejorar el rendimiento de sus aplicaciones en arquitecturas de procesadores Intel. Nos llamó la atención debido a que cuenta con análisis de rendimiento, permite analizar el rendimiento de aplicaciones en busca de cuellos de botella y áreas que requieren optimización. Proporciona información detallada sobre el tiempo de CPU, el uso de memoria, el rendimiento de las instrucciones y más. Además de que es compatible con una variedad de lenguajes de programación, incluyendo C/C++, Fortran, Python y otros. También se puede utilizar con diversos entornos de desarrollo, incluyendo Visual Studio y herramientas de línea de comandos.

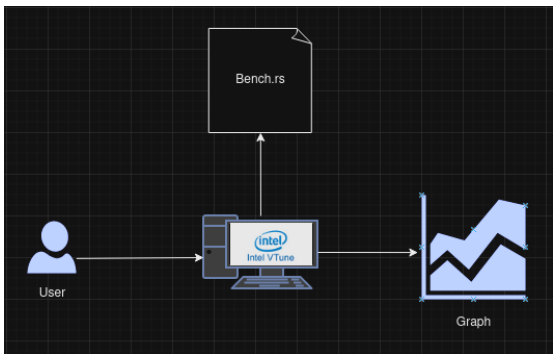

Fig. 1. Propuesta 1 usando VTune y Rust.

B. Propuesta 2

Para esta propuesta se contempló hacer uso de C y C++ para la elaboración de los programas a ejecutar y que se encargarían de ser analizados por una herramienta de perfilado para ver el efecto de False Sharing y su determinado rendimiento. C se caracteriza por su simplicidad y minimalismo. Ofrece un conjunto de características básicas que permiten un alto grado de control sobre el hardware subyacente. Esto lo convierte en una excelente elección para programación de sistemas y desarrollo de bajo nivel. Además de que nos ofrece una gran flexibilidad en términos de gestión de memoria y control de bajo nivel. Con esto se puede administrar directamente la memoria y los recursos del sistema, lo que nos da un alto grado de control pero también requiere una mayor responsabilidad.

Como herramienta de perfilado se pensó en el uso de **Valgrind**, el cual es una herramienta muy útil para el análisis de memoria y la depuración de programas escritos en C y C++. También consta de varias herramientas, siendo una de las más conocidas ”Memcheck”, el cual es una herramienta que realiza un seguimiento exhaustivo de las operaciones de memoria de un programa, lo que permite detectar errores de memoria. Otras herramientas de Valgrind, como ”Cachegrind” y ”Callgrind”, se utilizan para el análisis de rendimiento y la detección de problemas de rendimiento en programas C y C++.

Cabe mencionar que a primera instancia, teniamos la idea de que para poder ver los efectos del False Sharing y de como podríamos llevar al máximo el hardware objetivo, fue la idea de implementar un programa multi’threading algo pesado y que elaborara muchas operaciones simultaneas a elementos en un arreglo, operaciones desde suma, restas, comparaciones, intercamibos de posiciones y ordenamientos mediante Bubblesort.

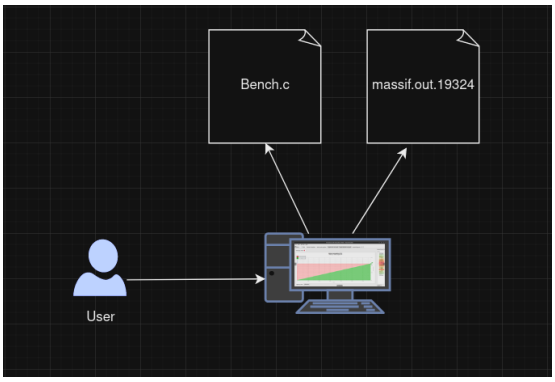

Fig. 2. Propuesta 2 usando Vilgrind y C/C++.

C. Propuesta 3

Para esta propuesta se contempló hacer uso de C y C++ para la elaboración de los programas a ejecutar y que se encargarían de ser analizados por una herramienta de perfilado para ver el efecto de False Sharing y su

determinado rendimiento. C se caracteriza por su simplicidad y minimalismo. Ofrece un conjunto de características básicas que permiten un alto grado de control sobre el hardware subyacente. Esto lo convierte en una excelente elección para programación de sistemas y desarrollo de bajo nivel. Además de que nos ofrece una gran flexibilidad en términos de gestión de memoria y control de bajo nivel. Con esto se puede administrar directamente la memoria y los recursos del sistema, lo que nos da un alto grado de control pero también requiere una mayor responsabilidad.

Como herramienta de perfilado para esta propuesta, se analizó el uso de la herramienta Perf, que es una herramienta de análisis de rendimiento integrada en el kernel de Linux. Proporciona una amplia variedad de herramientas para la monitorización y el análisis del rendimiento del sistema y las aplicaciones. También nos proporciona lecturas sobre las entradas a caché de datos así como la cantidad de caché misses, los cuales son valores que nos ayudan a la idetificación de False Sharing.

También se contempló el uso de Python para realizar la aplicación como tal meidante la unificación de todos benchmarks, así como los respectivos controladores y una respectiva interfaz de usuario para garantizar una buena interacción con la aplicación y las opciones de graficación.

Se hicieron algoritmos más simples, con menos tareas y funciones para no tener un analisis más limpio de lo que sucede a nivel de rendimiento en el sistema objetivo, además de que por recomendaciones de nuestro profesor de usar codigos complejos.

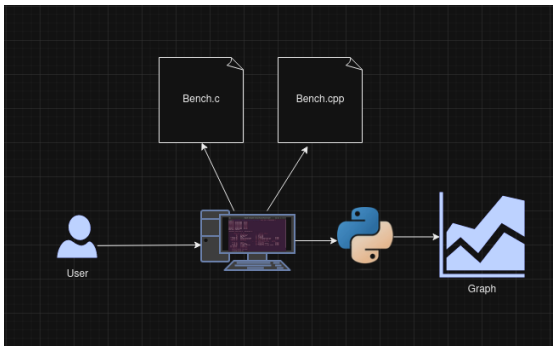

Fig. 3. Propuesta 3 usando Perf, Python y C/C++.

D. Propuesta Final

Como propuesta final se escogió la número 3, esto porque se nos hizo mas simple y entendible al usar las herramientas de perfilado, así como los lenguajes a utilizar, ya que Rust si bien es muy bueno y completo, no tenemos mucha experiencia con el y nos tomaría más tiempo el poder aprenderlo. Otro aspecto fue que con VTune, este presenta muchas compatibilidades con procesadores Intel y que si bien puede funcionar con AMD, algunas caracteristicas puede que esten bloquedas o no esten disponibles o funciones correctamente, es por eso que descartamos el uso de VTune y escogimos Perf como nuestra herramienta de perfilado.

III. Desarrollo

A. Benchmarks

Detalla cómo creaste los benchmarks para explorar el efecto del false sharing en múltiples lenguajes de programación.

Para la elaboración de los benchmarks, se realizaron dos tipos de benchmarks, uno usando C y el otro en C++. Cada uno de ellos capaz de ejecutar una versión single-threading y una multi-threading, esto para poder ver los efectos directamente en los sitemas multiprocesamiento.
Ambas pruebas son muy similares, se buscó el poder realizar un programa el cual pudiera replicar los efectos del False Sharing en nuestros sistemas objetivos y para ello, fue necesario el poder hacer uso de datos que estuvieran en la misma línea de caché, para así tener a varios hilos peleando por la misma línea de caché. Para la versión en C, se acudió a elabroar un programa que usara threads en base al sistema objetivo, dichos threads se encargarían de realizar modificaciones a una variable en específica, una variable respectiva por cada thread, utilizando un arreglo de contadores para poder ser accedido mediante el identificador de los threads, donde dicho arreglo iba a tener un tamaño determinado en base al tamaño de la línea de caché del sistema objetivo, donde por lo general en sistemas modernos, va alrededor de los 64 Bytes y en algunos casos 128 Bytes. Para las modificaciones, se creó una función de incremento, el cual iba a ser usada por cada thread para realizar la modificación respectiva. Dichos contadores mencionados anteriormente, son del tipo **atomic_int**, esto para exonerar los datos del fenómeno de ”Data Races”, además de que son operaciones que se garantiza que se ejecutarán sin interrupción de otros subprocesos o procesos, lo que garantiza que la operación se complete en un paso único e ininterrumpido. Esto es particularmente importante en escenarios de programación concurrente o de subprocesos múltiples donde varios subprocesos pueden acceder y modificar los mismos datos simultáneamente. Por último se definió una cantidad de iteraciones que vendría siendo en relación a la cantidad de incrementos que se deseaba realizar a los contadores y para asegurar que los incrementos fueran en base a la cantidad de threads, se dividió la cantidad de incrementos entre la cantidad de threads, esto para poder tener un mejor efecto de False Sharing al incrementar los Caché Misses que van de la mano del False Sharing donde al aumentar los threads, se tendrán más procesos peleando por la misma linea de caché, creando invalidaciones de la misma en muchas ocaciones y así, obtener el efecto en nuestras métricas.

Para el segundo benchmark elaborado en C++, se acudió a haer uso de un **struct** en lugar de una arreglo, esto crea un confinamiento para los contadores que si bien reduce un poco la cantidad de Caché Misses, el efecto de False Sharing se mantiene, ya que el comportamiento es similar. Además de

que se añadió una arreglo de dichas estructuras para poder posicionar una en cada posición de la línea de caché para que de igual manera que en el programa de C, estos estuvieran alineados en la misma línea de caché.

B. Hardware Objetivo

Para obtener los datos del sistema objetivo, se usó una combinación de **lscpu** y **sysconf**, esto debido a que cierta información era más accesible con una herramienta o solo era obtenida mediante la otra. Para unificar todo, se hizo un script en C el cual se encargara de la ejecución de todas las funciones de extracción de las especificaciones. También para que de manera dinámica y automática, el poder usar dichos parametros como la cantidad de cores para poder correr los test de manera personalizada en base a dichjos parametros.

C. Herramienta de Perfilado

A como se mencionó en la sección de Propuestas, se hizo uso de **Perf**, el cual es una herramienta de análisis de rendimiento integrada en el kernel de Linux. Esta herramientas nos permitiói analizar ciertos parámetros que nos ayudaría en la detección de False Sharing, como la cantidad de entradas a memoria de datos, caché misses y accesos a un mapeo de memoria para poder ver las variables siendo almacenadas en una misma línea de caché, así como el tiempo de ejecución de cada uno de los benchmarks.

D. Mitigación de False Sharing

Para poder dar una ejemplificación de como se podrían mitigar estos fenómenos de False Sharing, se acudió a realizar dos soluciones para los diferentes benchmarks, esto con el fin de poder graficar y realizar las respectivas comparaciones con respecto a su versiones sin solucionar.
Para la primera solución, se implementó Alineamiendo de Datos, el cual se agregan un padding a cada contador, para asegurar que cada contador quede almacenado en una linea de caché por aparte y así evitar que los threads se peleen por esa misma liena de caché.

Como segunda solución, se hizo uso de **TLS**, esto es un tipo de almacenamiento local a nivel de threads, donde cada thread posee una instancia de ese contador mediante el uso de un thread a dicho contador **__thread**, cada thread ejecuta su función respectiva sin necesidad de realizar accesos a línea de caché compartida donde no se ve alterada o afectada por False Sharing.

E. Automatización

Para esta etapa, se hizo un script en Python que unificara todos los benchmarks y controladores, brindando una interfaz al usuario para poder interactuar de manera fácil e intuitiva a la hroa de escoger y ejecutar los tests. Se brindan opciones de ejecución de los benchmarks, obtención de especificaciones de sistema, ejecución de soluciones, comparaciones entre graficas y accesos a resultados almanacenados en ejecuciones anteriores.

IV. Resultados

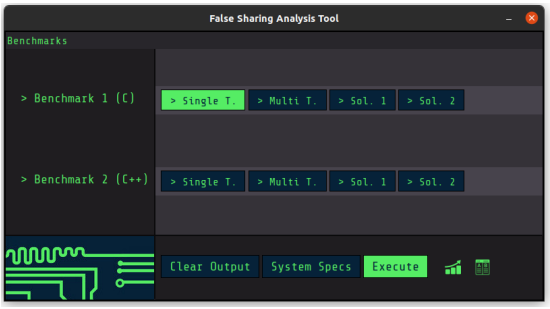

Fig. 4. Aplicación con Python.

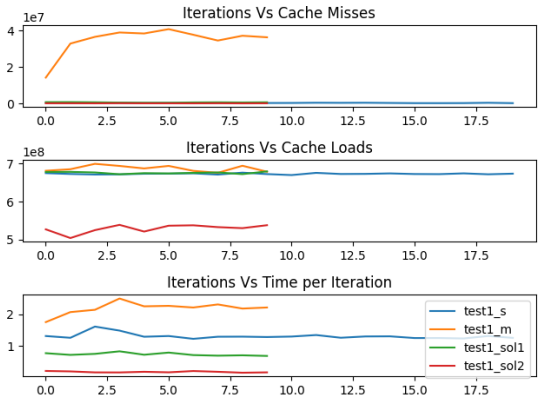

Fig. 5. Gráficos con los valores de Perf.

```
Thread(s) per core:                2
Core(s) per socket:                2
L1d cache:                         64 KiB
L2 cache:                          512 KiB
L3 cache:                          3 MiB
Model name:                        Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
This: t
----- Especificaciones del sistema para t -----

Number of Cores:                    4
RAM Memory:                        19.40 GB
Cache L1:
Line size: 64 bytes
Instance size: 32768 bytes
Cache L2:
Line size: 64 bytes
Instance size: 262144 bytes
Cache L3:
Tamaño de línea: 64 bytes
Instance size: 3145728 bytes
```

Fig. 6. Especificaciones de sistema objetivo.

V. Análisis de Resultados

En la figura **Fig 4** se aprecia la interfaz gráfica de nuestra aplicación, la cual fue hecha en Python, con una interfaz intuitiva donde se le brindan todas las opciones disponibles y necesarias para poder analizar los efectos de False Sharing.

En la figura **Fig 5** Se observan un ejemplo de los graficos que se pueden obtener mediante nuestra aplicación, donde también es posible realizar comparaciones entre las diferentes versiones de los benchmarks, tanto single-threading, muilti-threading y sus respectivas soluciones. Donde se puede ver una

notoria diferencia entre un single thread, donde no hay muchas variaciones con respecto a las métricas obtenidas ya que no se dan problemas de lineas de caché compartida como sucede en multi threading(Línea naranja), además de brindarnos también una comparación son la respectiva solución para ver como se puede mitigar el efecto del False Sharing.

VI. Conclusiones

- El efecto de False Sharing es bastante perjudicial en términos de rendimiento, puesto que su presencia en sistemas de múltiples nucleos conlleva a múltiples invalidaciones innecesarias en memoria y la visualización de este fenomeno no termina siendo sencilla a pesar de las herramientas con las que se cuenta actualmente, por lo que el desarrollo correcto de codigo es de vital importancia para evitar este tipo de problemas de rendimiento en nuestras aplicaciones.
- El uso de herramientas de perfilado es de gran utilidad para lograr ubicar y atacar problemas de rendimiento como el False Sharing, puesto que permiten tener métricas adecuadas a un programa en específico, permitiendo así la realización de benchmarks especializados.
- La gestión correcta de memoria y el uso de datos alineados puede ayudar a la disminución de fenómenos como el false sharing, esto debido a que estas medidas de mitigación reducen los accessos a una misma línea de caché por parte de múltiples hilos durante la ejecución de programas.
- Conocer el hardware objetivo es de vital importancia para la correcta generación de codigo que impida fenomenos como el false sharing que disminuyan el rendimiento de nuestras aplicaciones.

References

[1] https://en.wikipedia.org/wiki/False_sharing
[2] https://www.gem5.org/documentation/learning_gem5/part3/MSIintro/
[3] Hennessy, J., Patterson, D. (2017). Computer Architecture: A Quantitative Approach (6th ed.). Elsevier Science.
[4] Sorin, J Hil, Mark D. (2011) A Primer on Memory Consistency and Cache Coherence.
[5] https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/caches/MESI.htm
[6] https://www.gem5.org/documentation/learning_gem5/part3/MSIdebugging/
[7] Perf wiki. (2023) https://perf.wiki.kernel.org/index.php/TutorialCounting_with_perf_stat