

Sistemas Operativos:

Concurrencia/Sincronización y uso de profiler

**Alumnos:** Fabián Cid Escobar

Rodolfo Fariña Reisenegger

**Profesora:** Cecilia Hernández Rivas

8 de Enero del 2021, Concepción

**Introducción**

Es importante saber aprovechar al máximo los recursos de un computador al momento de programar, ya que de esta manera podemos generar algoritmos y códigos que se ejecuten de manera más rápida considerando la arquitectura del sistema.

A continuación, se presentan 2 implementaciones, la primera es una simulación de un paso vehicular con una sola vía que debe compartirse para pasar en 2 direcciones utilizando hebras cuidando de no caer en bloqueos mortales y condiciones de carrera, y la otra es una comparación de algoritmos de multiplicación de matrices cuadradas con el fin de ilustrar cómo aprovechan los caché del CPU.

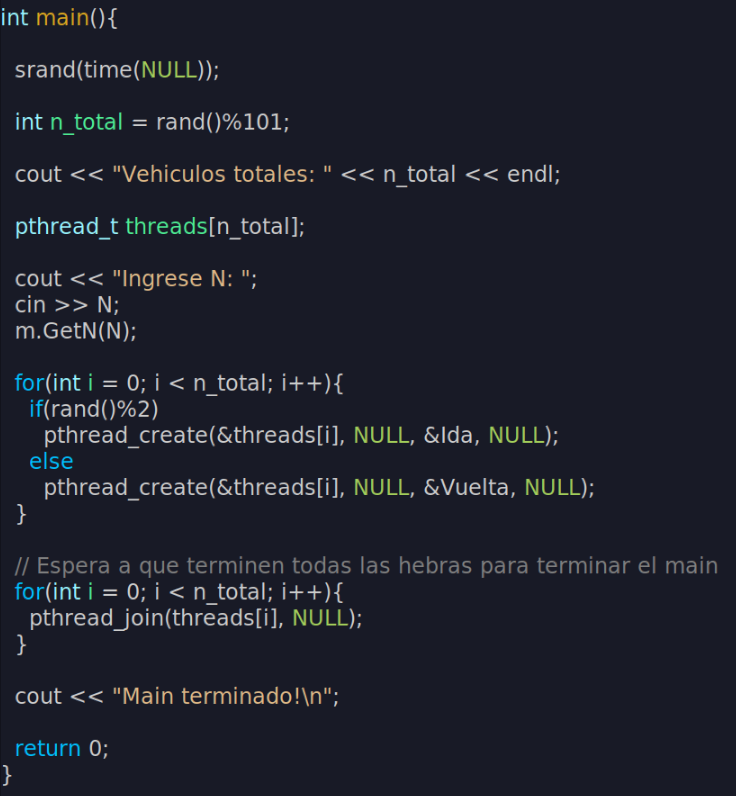
Finalmente, se hará un pequeño análisis sobre los tipos de fallos que ocurren al utilizar un browser, en particular, firefox.

**Simulador de Tráfico**

*El código para este ejercicio se encuentra en la carpeta “item a” en los archivos “main.cpp” y “Monitor.h”*

En el archivo “main.cpp” se crea un Monitor como variable global, se crean “n\_total” hebras con una semilla aleatoria que se inician con las funciones “Ida” y “Vuelta” que representan las direcciones de las hebras, las cuales son asignadas también de manera aleatoria para generar comportamientos inesperados y probar la consistencia del monitor.

La función main:

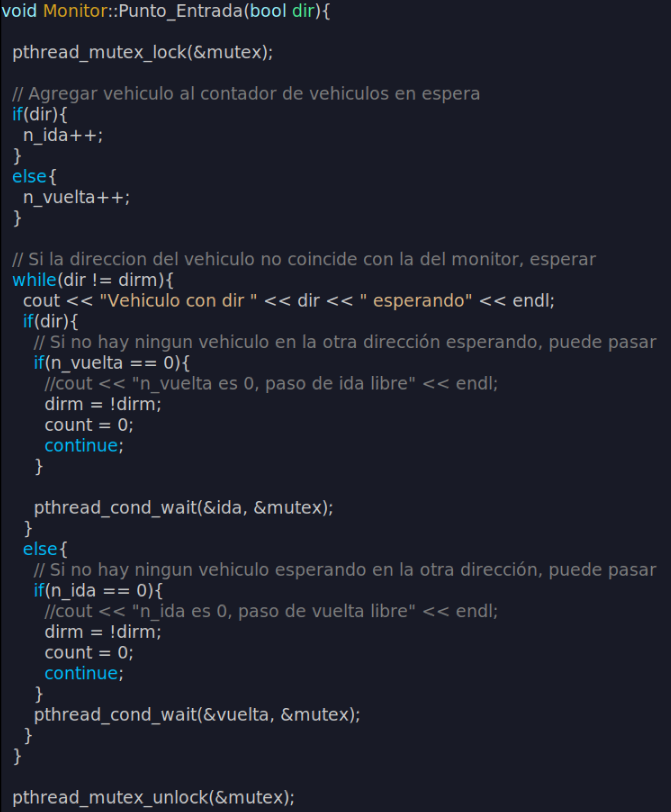


El monitor:

Es una clase de C++ con 1 constructor y 3 métodos: *Punto\_Entrada(bool dir)*, *Punto\_Salida()* y *GetN(int N)*.

A continuación solo se explicarán los métodos de entrada y salida para los vehículos a través de imágenes ya que considero que es más cómodo leer el propio código con sus comentarios que el pseudocódigo. Se omiten el constructor y el método *GetN*.

Método *Punto\_Entrada(bool dir)*:



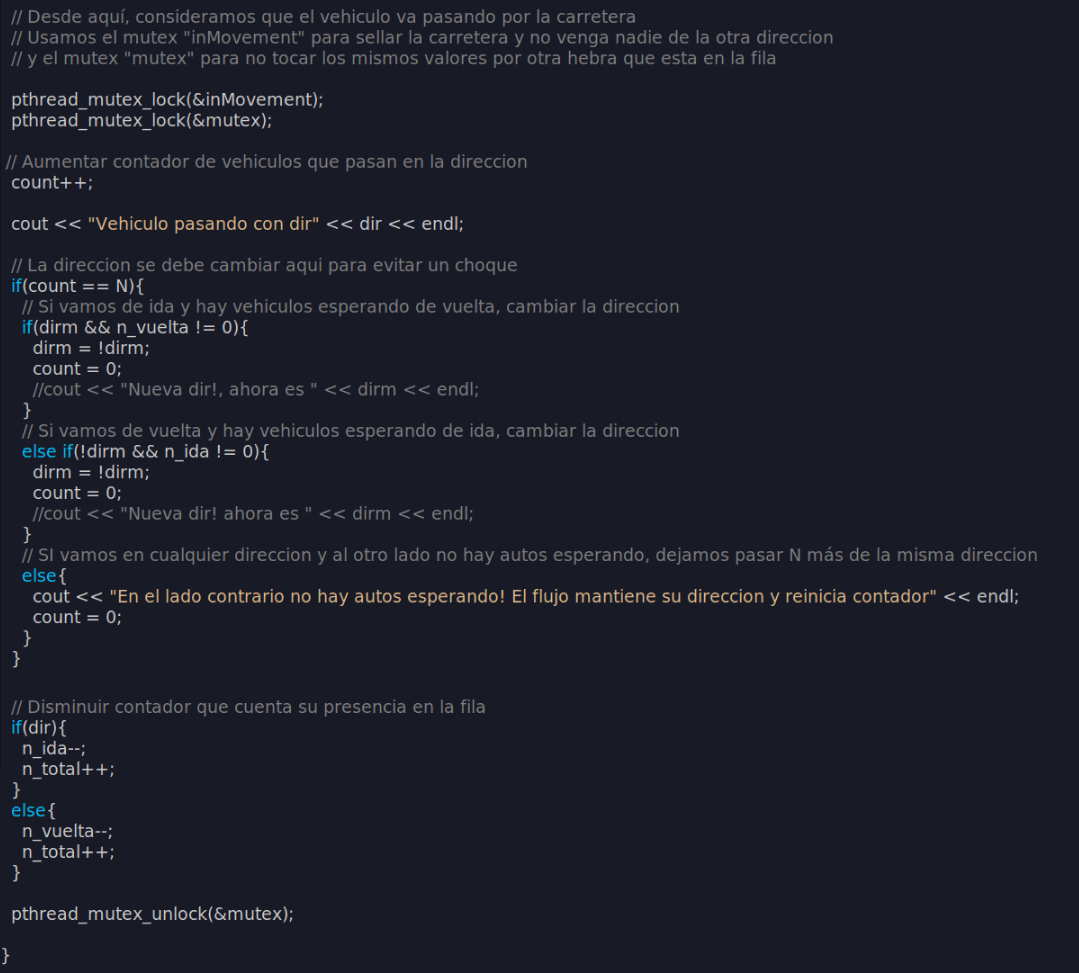
La variable *bool dirm* es la dirección del monitor.

Los enteros *n\_ida* y *n\_vuelta* llevan la cuenta de los vehículos que están en la cola esperando en cada dirección, las variables de condición *ida* y *vuelta* diferencian la dirección de la hebra al momento de hacer signal.

Aquí, además de prevenir el paso de una hebra viene en una dirección que no coincide con la del monitor, también nos deshacemos de un bloqueo mortal en el caso en que una hebra con dirección opuesta a la del monitor deba dormir pero del otro lado no haya ninguna hebra, en ese caso, se cambia la dirección del monitor para dejarla pasar y se reinicia el contador.

También se usa el mutex “mutex” encolar vehículos uno a la vez evitando condiciones de carrera. Luego se suelta para simular la entrada a la calle.

A continuación, se muestra la segunda mitad del método:

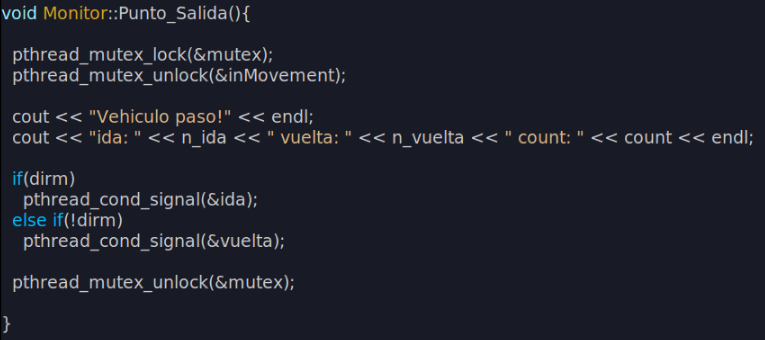


El vehículo una vez en camino, se usa el mutex *inMovement* el cual se asegura de que no entre ninguna hebra al camino una vez alguno esté dentro evitando así condiciones de carrera con autos de ambos sentidos, también se asegura de que lleguen en el orden de entrada. Una vez tomado *inMovement*, también toma a *mutex* ya que también lee y modifica valores que se utilizan en la parte en la que los autos se encolan evitando más inconsistencias.

En esta sección del método también se cambia la dirección ya que en el método de salida solo se avisa al siguiente auto para entrar según la dirección del monitor. También se prevé un bloqueo mortal si no hay autos esperando en la dirección contraria manteniendo la misma dirección. Si la dirección en esta parte no se cambia y la próxima hebra en llegar viene en dirección contraria, pasa de igual manera con la prevención de bloqueo mortal explicada en la primera parte del método.

Al final, se descuenta al vehículo de la fila y se suelta al mutex, dando paso a la llamada de *Punto\_salida()*.

Método *Punto\_Salida()*:



Aquí, al comienzo se toma a *mutex* para evitar condiciones de carrera y se suelta *inMovement* para dejar pasar al vehículo siguiente en la carretera, luego según la dirección se hace un signal a la siguiente hebra dormida.

Y así será para cada hebra hasta que logren salir todas y el programa termine.

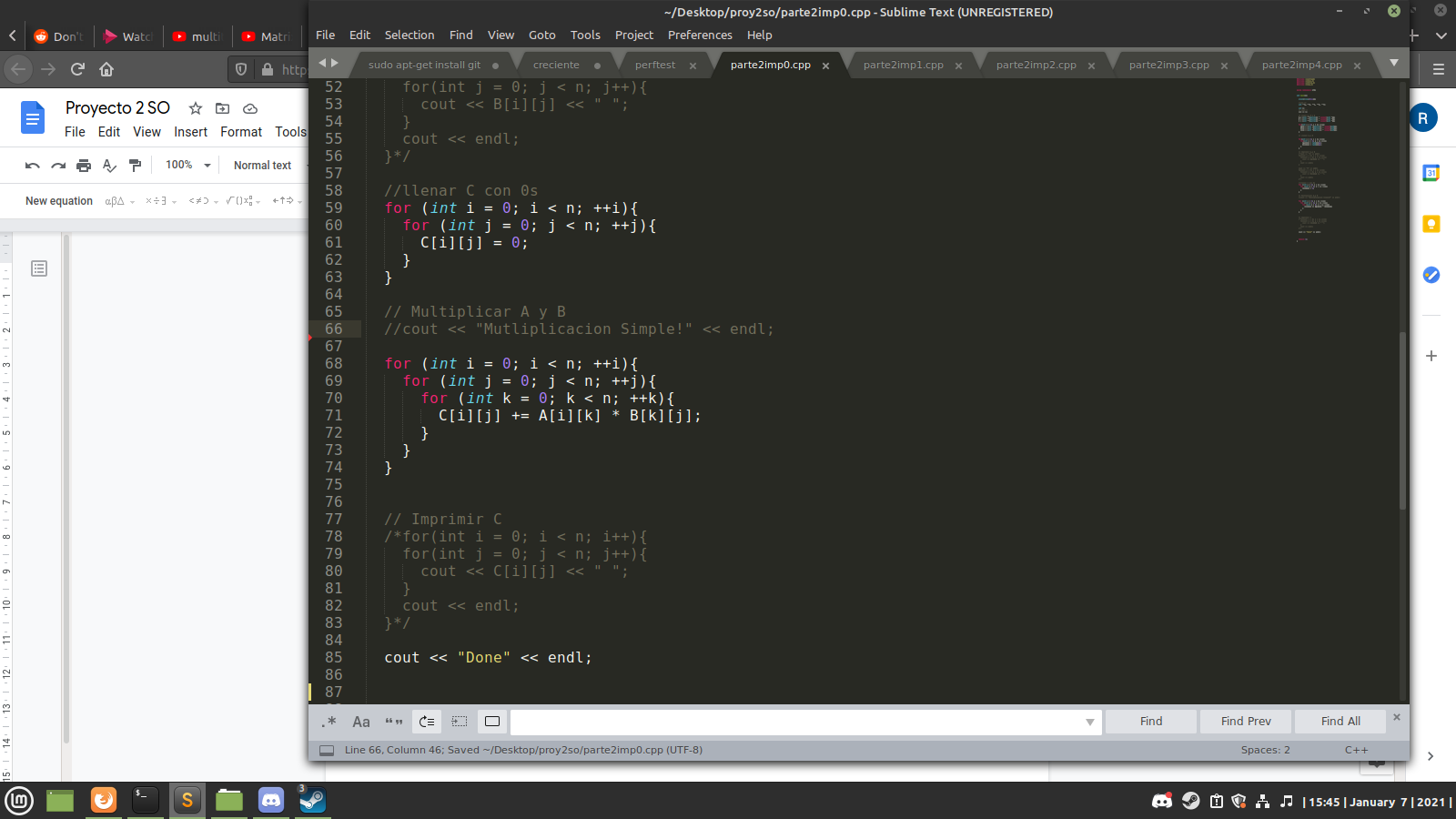
Con las medidas mencionadas anteriormente, jamás se ha obtenido algún bloqueo mortal o inconsistencias dentro de la ejecución.

**Multiplicación de matrices con perf**

Es evidente que hoy en día la multiplicación de matrices es uno de los problemas más comunes en casi todos los campos donde se analizan grandes cantidades de datos, por esto, es necesario poder implementar algoritmos que utilicen los procesadores actuales de manera correcta. Uno de los mejores algoritmos disponibles hoy en día es el de Strassens, sin embargo, no veremos ese, sino que analizaremos 3 implementaciones, la “naive” (implementación 0), la “transpuesta” (implementación 1) y la “copia” (implementación 2).

Se solicitó programar la multiplicación de matrices con 3 implementaciones distintas, cada una debía ser , luego de esto se realizó un análisis de cómo actuó el CPU con cada una de estas. A continuación, se muestran las alternativas:

Naive:



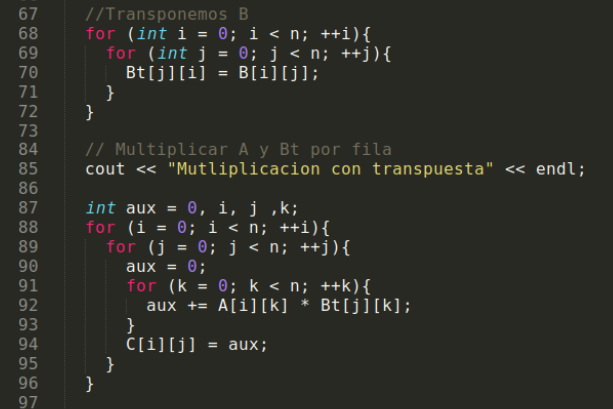
*Imagen: Implementación “naive”*

Este corresponde al algoritmo “naive” de multiplicación de matrices, es el más lento de los que implementamos y este constantemente tendrá misses en el L1 cache.

La implementación de este código está en el archivo “**parte2imp0.cpp**”, para compilar use “g++ parte2imp0.cpp -O3 -o p2i0.out”. Notar que estos archivos se ejecutan individualmente.

Este es el más lento de los códigos y más adelante veremos el rendimiento de él. Nos referiremos a esta implementación de ahora en adelante como la “naive”.

El siguiente snippet de código corresponde a la implementación 2, donde se transpone la matriz B y se realiza la multiplicación ajustada a esto para obtener el resultado correcto. Este código tiene muchos menos fallos en el Cache L1, debido a la organización de la matriz en esta. Esta implementación la nombraremos como “transpuesta”, debido al procedimiento que realiza.

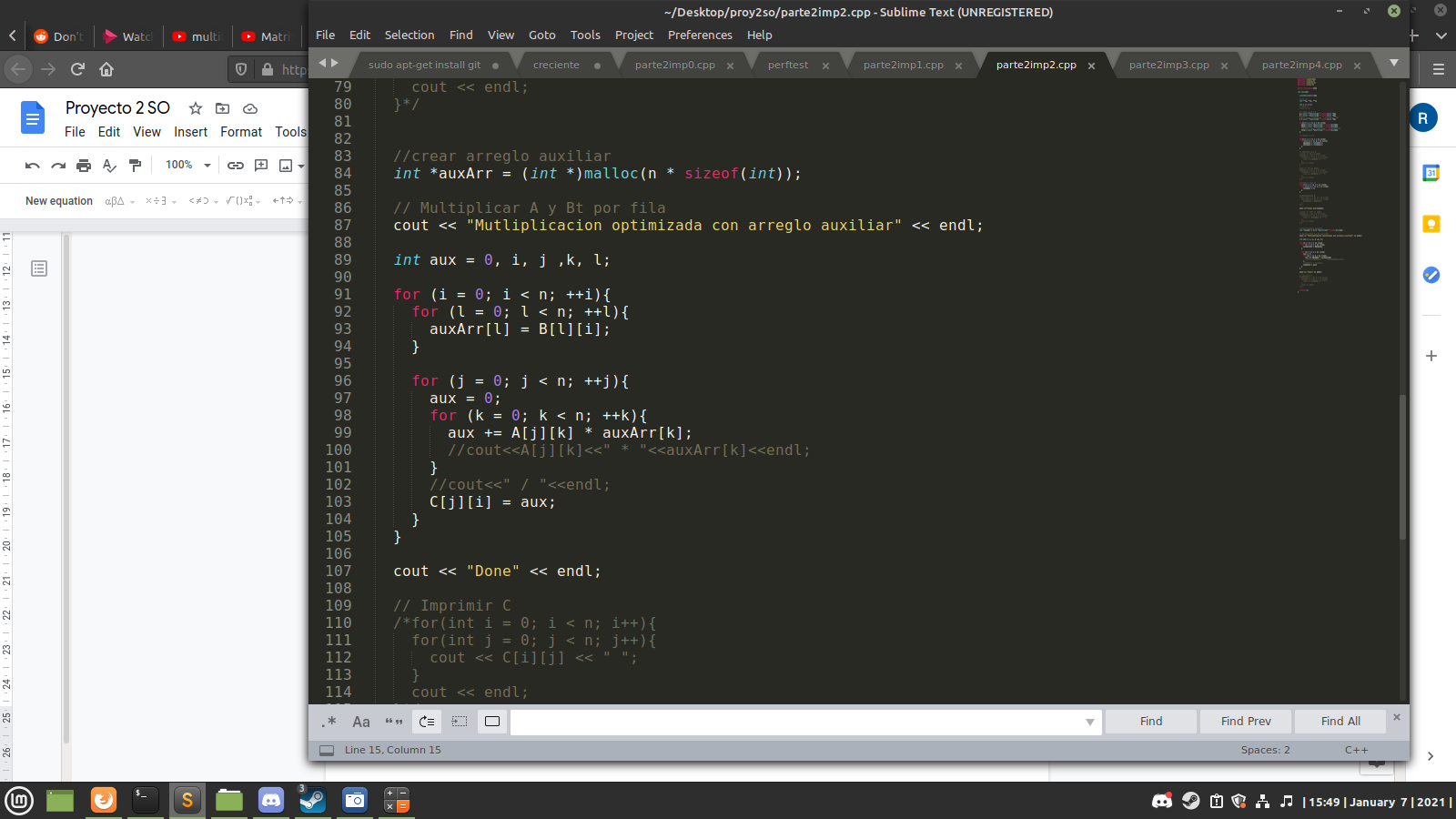


*Imagen: Implementación “transpuesta”*

A diferencia de la implementación anterior, esta si recorre la matriz B como está dispuesta en el caché, sin tener que ir haciendo saltos constantemente, por esto, es mucho más rápido y esto lo veremos en los resultados más adelante.

La implementación de este código está en el archivo “**parte2imp1.cpp**”, para compilar use “g++ parte2imp1.cpp -O3 -o p2i1.out”.

Finalmente, la última implementación, a la cual llamaremos “copia”, toma la idea de copiar una columna de la matriz B, para así acceder a un conjunto más pequeño (arreglo) y tener la información más a mano. A pesar de que implica copiar, es mas rapida y lo veremos en los resultados.



*Imagen: implementación “copia”*

A continuación, analizaremos los resultados de los tiempos de ejecución de multiplicación de matrices aleatorias, desde tamaño 128 a 2048, en potencias de 2. Omitimos los resultados más pequeños puesto que las diferencias son muy pequeñas, de hecho, en general no habían muchas diferencias en los resultados entre las 3 implementaciones, esto debido a que el tiempo en copiar la matriz quitaba los beneficios de transponer la matriz o ir copiando al arreglo auxiliar. La implementación de este código está en el archivo “**parte2imp2.cpp**”, para compilar use “g++ parte2imp2.cpp -O3 -o p2i2.out”.

Para el análisis de perf, se presentaron algunos inconvenientes, puesto que algunos de los eventos del programa no eran soportados en la configuración de ninguno de los 2 integrantes.

Especificaciones del computador en que se realizaron las pruebas:

*CPU: AMD Ryzen 7 PRO 1700 Eight-Core Processor 3.7 GHz 768KB L1, 4MB L2, 16MB L3, 8 Core/16 Thread*

*RAM: 2x8GB 3200MHz CL16*

*GPU: XFX RX580 Ellesmere XT Core 1411 Mhz 8GB Mem 2000 Mhz*

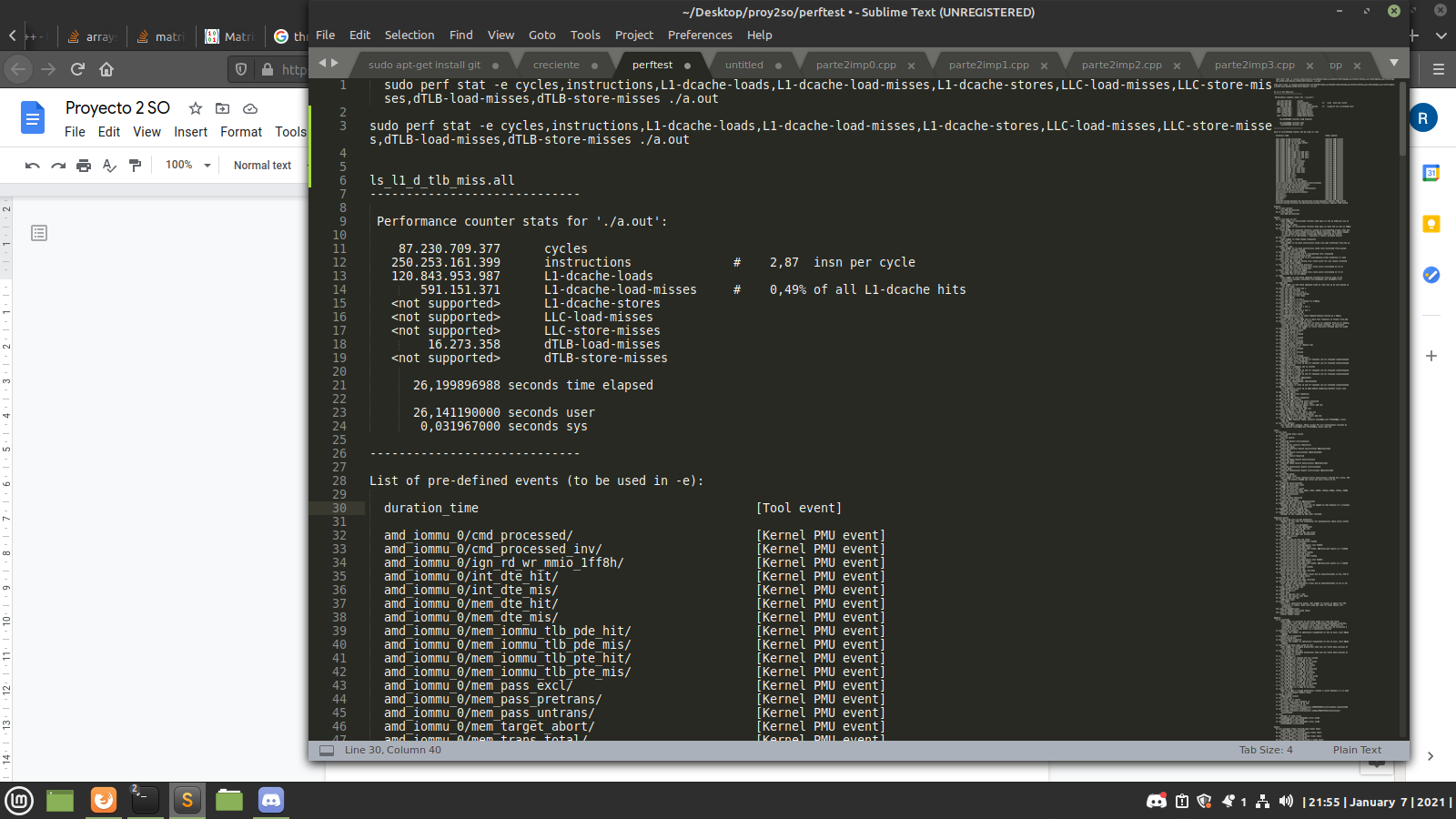
*SSD: Crucial BX500 480GB*

*HDD: WD 1TB 7200 RPM*

Al utilizar perf, se comenzó con la siguiente línea de comando, basándonos en lo aprendido en las prácticas:

*sudo perf stat -e cycles,instructions,L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores,LLC-load-misses,LLC-store-misses,dTLB-load-misses,dTLB-store-misses ./a.out*

Al utilizar esto, obtuvimos el siguiente resultado:



Donde se ve que hay varios eventos *“<not supported>”* por la configuración actual, se intentó utilizar distintos eventos similares que se obtuvieron de *“perf list”,* tales como:

* amd\_iommu\_0/mem\_dte\_hit/
* amd\_iommu\_0/mem\_dte\_mis/
* ls\_l1\_d\_tlb\_miss.all
* bp\_l1\_tlb\_miss\_l2\_hit

Los cuales tampoco estaban soportados, o arrojaban valores extraños, por esto, se ha decidido omitir dichos valores, de esto, nuestro análisis estará limitado a los siguientes eventos:

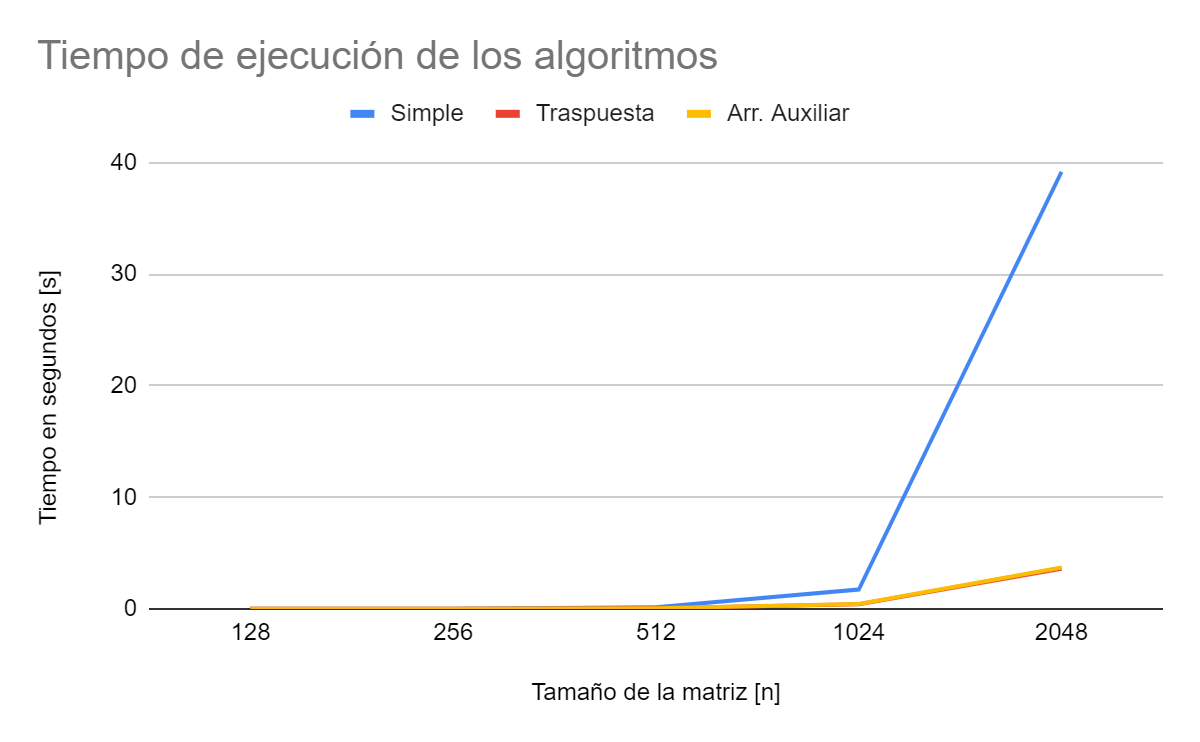
* cycles
* instructions
* L1-dcache-loads
* L1-dcache-load-misses
* dTLB-load-misses
* dTLB-loads (este fue uno que logre encontrar que sirve)

Adicionalmente, se vio la alternativa de utilizar un programa distinto a perf para obtener información, pero dichas alternativas eran de pago.

Se obtuvieron los siguientes resultados:

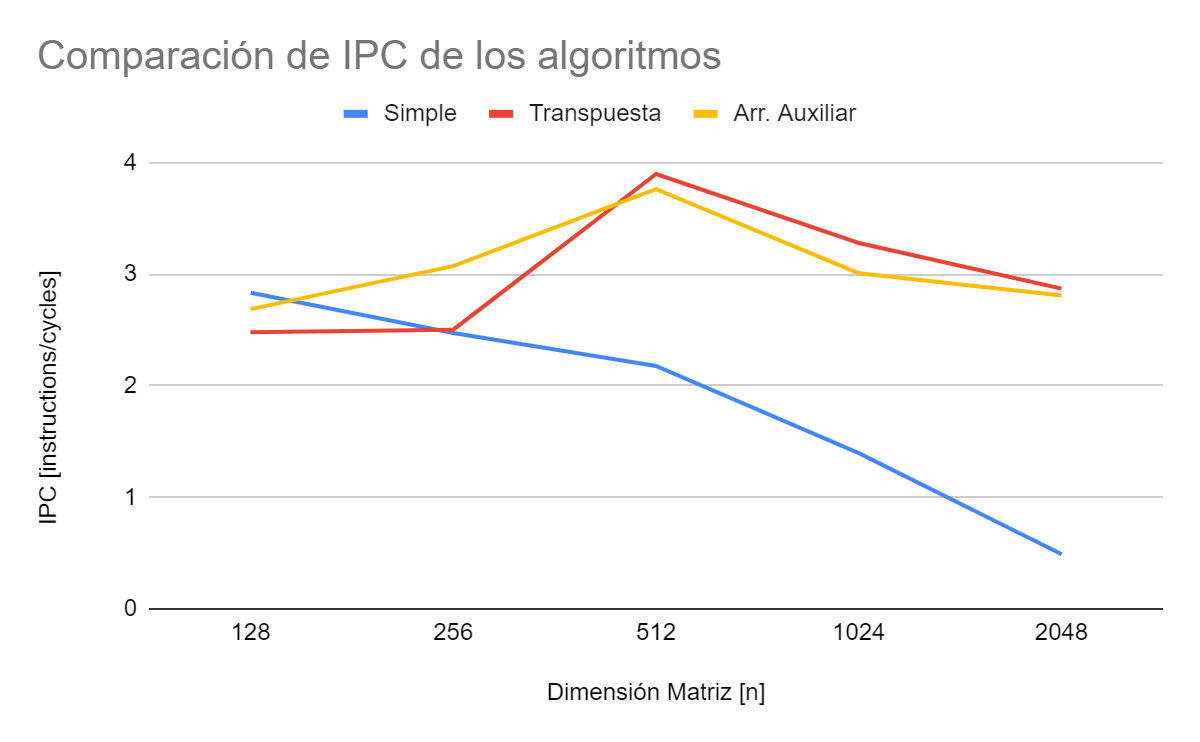
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***Multiplicación simple*** | | | | | | | |
|  | **cycles** | **inst** | **L1-dcache-loads** | **L1-dcache-load-misses** | **dTLB-load-misses** | **dTLB-loads** | **Time elapsed** |
| **128** | 7344550 | 20804743 | 8737923 | 198157 | 1405 | 1794 | 0.005766378 |
| **256** | 52955001 | 130930928 | 52888338 | 1668811 | 26895 | 15073745 | 0.022714834 |
| **512** | 451482744 | 982988117 | 421382094 | 141561726 | 477217 | 137487421 | 0.146807722 |
| **1024** | 5513116124 | 7695571002 | 3493561729 | 1286244011 | 193835512 | 1080488875 | 1.728929354 |
| **2048** | 124378861265 | 60954498926 | 32672660644 | 10258497246 | 6441651040 | 8637860168 | 39.1587705 |
| ***Multiplicación transpuesta*** | | | | | | | |
|  | **cycles** | **inst** | **L1-dcache-loads** | **L1-dcache-load-misses** | **dTLB-load-misses** | **dTLB-loads** | **Time elapsed** |
| **128** | 5769959 | 14300570 | 4698832 | 297375 | 1646 | 14409 | 0.004604734 |
| **256** | 25572250 | 63995693 | 21169328 | 1614127 | 2656 | 180711 | 0.013411133 |
| **512** | 147118066 | 573371504 | 159207239 | 10681260 | 8808 | 774505 | 0.055708131 |
| **1024** | 1292809545 | 4239006580 | 1145828742 | 77798908 | 1905802 | 3740141 | 0.410006475 |
| **2048** | 11508665174 | 33033722302 | 8934032427 | 586818578 | 16334349 | 18153130 | 3.598842539 |
| ***Multiplicación arreglo auxiliar*** | | | | | | | |
|  | **cycles** | **inst** | **L1-dcache-loads** | **L1-dcache-load-misses** | **dTLB-load-misses** | **dTLB-loads** | **Time elapsed** |
| **128** | 5292982 | 14220391 | 4606298 | 292319 | 1528 | 5993 | 0.004265128 |
| **256** | 25601214 | 78639176 | 19595861 | 1638925 | 3267 | 13165 | 0.015353559 |
| **512** | 149505889 | 562470743 | 149985773 | 10660280 | 14392 | 700027 | 0.050276572 |
| **1024** | 1406251803 | 4230955633 | 1155929097 | 78693605 | 3269589 | 3980090 | 0.435634543 |
| **2048** | 11762231343 | 33038736908 | 8924281468 | 597670245 | 20501794 | 21426972 | 3.712286177 |

Notar que cada valor se calculó 3 veces y se obtuvo la media de dichos resultados, para así tener más precisión.



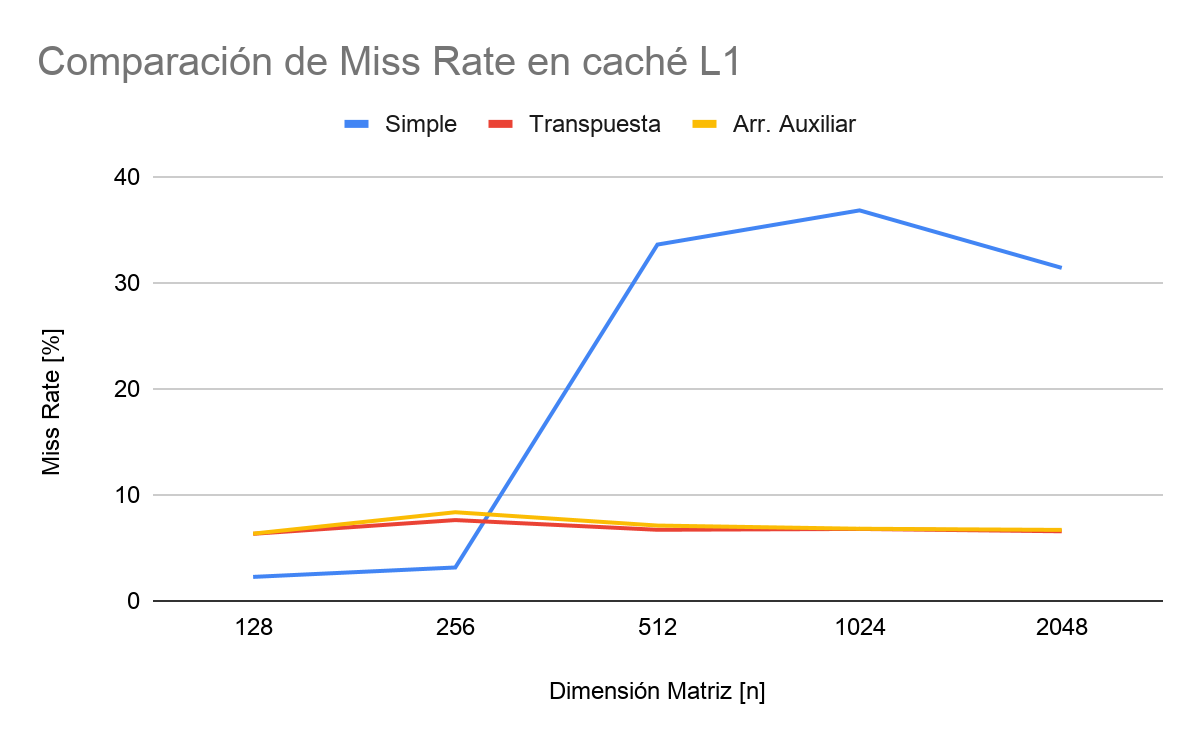
*Gráfico 1: Tiempo de ejecución promedio de los algoritmos de multiplicación de matrices*

En el Gráfico 1 se puede observar claramente que la implementación transpuesta y la del arreglo auxiliar son casi iguales, con la transpuesta siendo levemente mejor, esto es interesante, puesto que según la literatura y resultados en otros sistemas Intel, debería ser levemente la implementación de arreglo auxiliar. Creemos que esto puede ser debido a las diferencias de arquitectura entre los procesadores AMD con Intel. Evidentemente, la implementación naive es mucho peor, demorando casi 10 veces más que las anteriores, mostrando claramente porque es necesario optimizar algunos códigos.



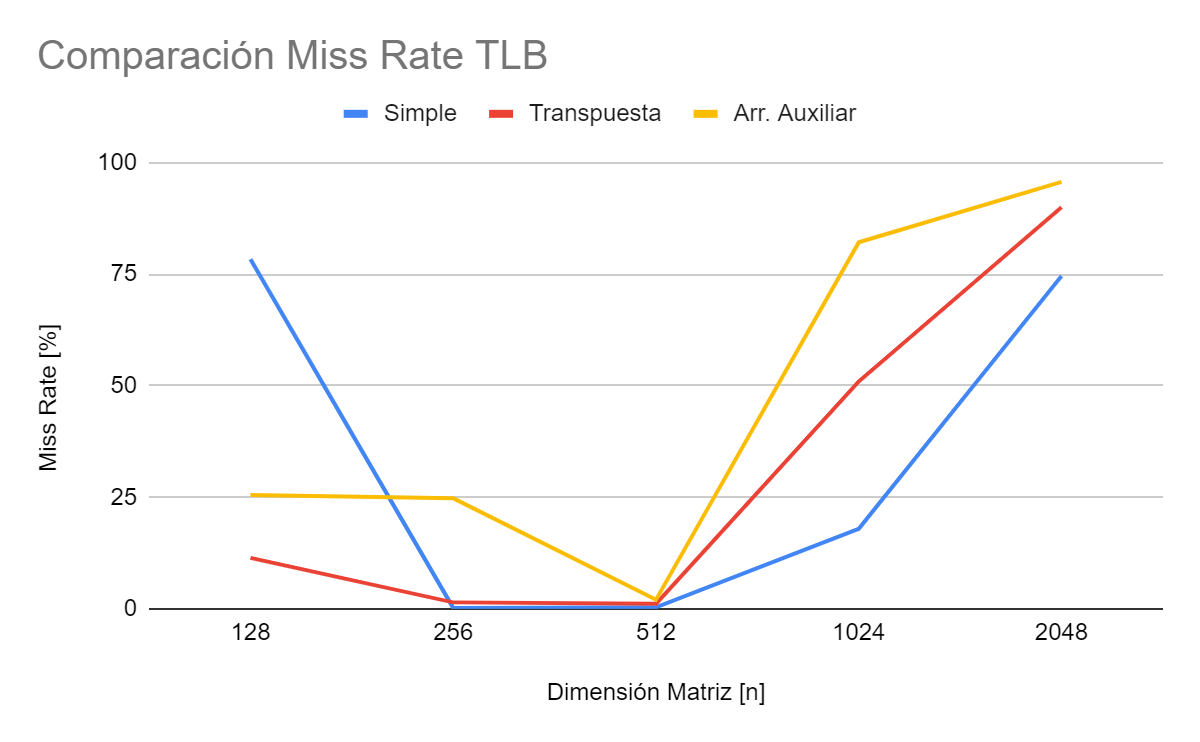
*Gráfico 2: Comparación de instrucciones por ciclos de los 3 algoritmos*

Otro detalle que consideramos interesante, eran las instrucciones por ciclo (IPC), es interesante notar que el IPC permanece alto para la implementación transpuesta y la del arreglo auxiliar, sin embargo, para la implementación naive uno puede ver que el IPC disminuye considerablemente al aumentar el tamaño de la matriz. Esto es porque en la implementación naive los saltos entre las columnas de la matriz consumen mucho tiempo y se demoran mucho en cada ciclo, por esto baja el IPC.



*Gráfico 3: Comparación de los miss rate en caché L1 de los 3 algoritmos*

Lo siguiente es analizar los loads y misses del caché L1, para esto, analizaremos el porcentaje de precisión de los loads y los misses (10 loads con 8 misses = 80% fallo). Notamos que inicialmente la implementación naive tiene menos misses (debido a que no copia la transpuesta o arreglo auxiliar), pero a medida que aumenta el tamaño, el “missrate” aumenta considerablemente. Las otras 2 implementaciones permanecen relativamente constantes con el tamaño de la matriz, y son mucho mejores, puesto que presentan el comportamiento esperado del cache según la literatura, que es un “missrate” menor a 10%.



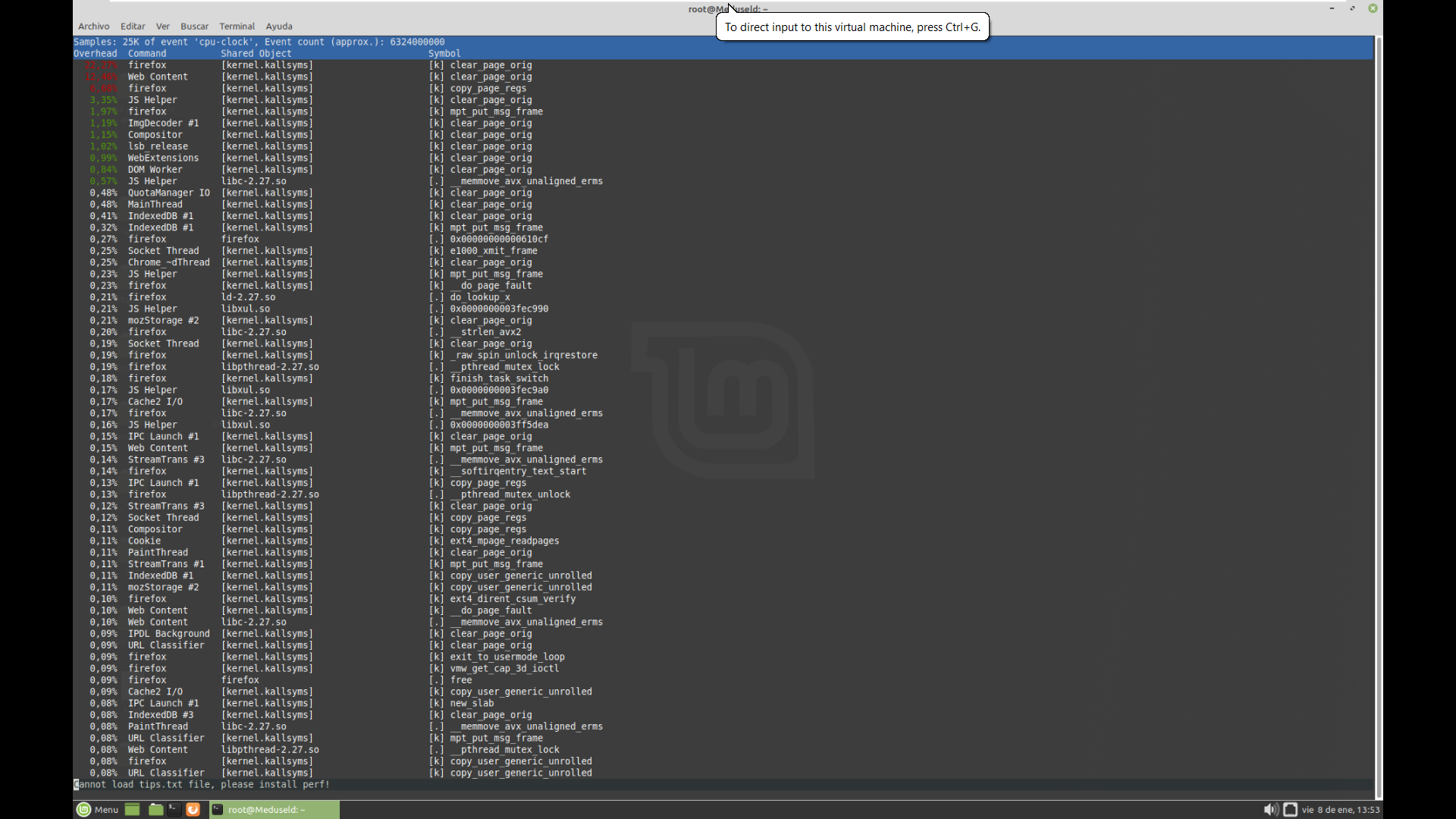
*Gráfico 4: Comparación de Miss Rate de TLB*

En el miss rate de la TLB, se presenta una anomalía extraña para las matrices de tamaño 256 y 512, resulta difícil encontrar una explicación para esto, pero creemos que esto puede ser porque la asociación de tablas de marcos y páginas para esos tamaños funciona mejor. Sin embargo, nos gustaría encontrar una explicación más precisa, puesto que investigamos y no logramos encontrar mucho al respecto.

Creemos que para obtener más precisión se deberían hacer pruebas con tamaños 4096 y 8192 de matrices.

**Linux perf y browser internet**

Lo siguiente era investigar sobre cuáles eran las funciones que utilizaban más el CPU al usar firefox. Para esto, hemos utilizado el comando *“perf record firefox”* y luego *“perf report”*, esto nos entrega las funciones junto al porcentaje de uso del cpu que corresponden. A continuación, presentamos nuestro resultado de navegar la web un poco:

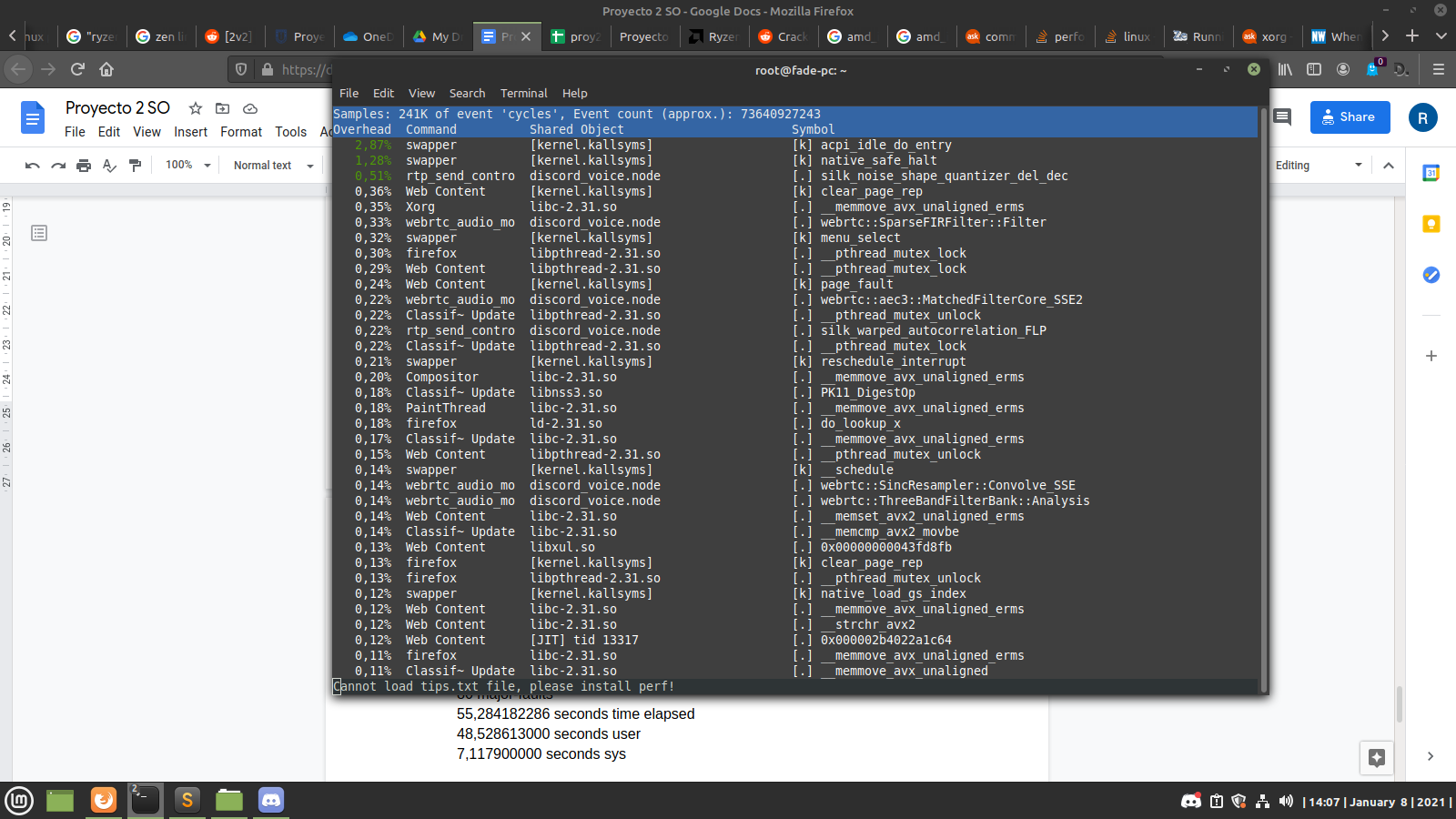


Nos podemos dar cuenta que firefox es quien más está utilizando el CPU, junto con “web content” y otras partes de “js helper”, todos partes de firefox y probablemente llamadas entre sí. Nos damos cuenta de que “clear\_page\_orig” es la función que más consume tiempo, junto con “copy\_page\_regs” y “mpt\_put\_msg\_frame”, las 3 para generar las páginas web que uno puede ver con el browser.

Investigamos de clear\_page\_orig y es una función perteneciente a un repositorio de Linus Tovardls, está inicializa las páginas web. “copy\_page\_regs” resulta difícil de analizar, puesto que la encontramos en ASM y “mpt\_put\_msg\_frame” envía una request MPT, también fue difícil encontrar info de esta.

Es interesante notar que las funciones más usadas dependen mucho del tipo de browsing de web que uno hace, por ejemplo, al ver youtube o reddit, cambia considerablemente el output del report.

Además, si uno usa la opción “-a” para hacer record, graba todas las funciones llamadas por todo el procesador



Resulta difícil entender exactamente qué ocurre acá, pero, se da a conocer que varias de estas funciones corresponden a multithreading como “\_pthread\_mutex\_lock” por firefox, manejo de páginas y swaps de procesos. Aca se logra ver porque es importante el manejo correcto de múltiples hebras. Además, es importante notar que esto se realizó en el computador del Ryzen 7 1700, mientras que el reporte de perf anterior fue en un i5-8300H, notamos que hay diferencia entre los reportes de los distintos procesadores.

Sobre las *minor faults* y *major faults* hemos descubierto lo siguiente:

*Minor Faults*: Ocurre cuando se debe asignar una página.

*Major Faults:* Ocurre cuando se debe recurrir al disco duro para recuperar información de una página.

En la primera ejecución, se puede ver que se registran muchos “major-faults”, esto porque es necesario ir a recopilar del disco la información necesaria para la ejecución de Firefox, del mismo modo, hay muchas minor faults, porque se deben asignar las páginas necesarias para que el programa pueda funcionar correctamente.

Performance counter stats for 'firefox':

889.902 minor-faults

80 major-faults

55,284182286 seconds time elapsed

48,528613000 seconds user

7,117900000 seconds sys

Para la ejecución concurrente de firefox ocurre algo interesante, solo 1 major-fault, mientras que hay minor faults de todos modos. Esto es porque el CPU ya no necesita ir al disco para recopilar la información, sino que ya está guardada en memoria y puede recuperarse más rápido. Notar que dichas partes en memoria pueden o no ser compartidas. También notar que hay más minor-faults porque se ejecuta durante una mayor cantidad de tiempo.

Performance counter stats for 'firefox':

1.326.405 minor-faults

1 major-faults

70,788155713 seconds time elapsed

54,317637000 seconds user

10,443119000 seconds sys

**Conclusión**

En el camino a la versión final y funcional del monitor, dimos con muchos problemas de concurrencia, tales como vehículos que “chocaban” en el camino al no implementar de manera correcta los *mutex* o no se utilizaban correctamente las *variables de condición* provocando bloqueos mortales. Aquello nos hizo dar cuenta de la importancia que tiene la sincronización de hebras y su trabajo en paralelo. También en la parte de *perf* para firefox, notamos que usaba *pthread\_mutex\_lock* lo que nos lleva a concluir que lo que aprendimos en este trabajo está muy presente en la programación actual.

Además, se ha podido ver de manera muy evidente el impacto de programar con la arquitectura del CPU en mente, una mejora de 40 segundos a 4 segundos es extremadamente importante. Esto nos lleva a que en las siguientes instancias en que se programe, se tenga en mente la manera en que se accede a las matrices si uno está manejando matrices demasiado grandes.

Además, fue posible ver que el CPU consume una cantidad de tiempo considerable en el manejo de procesos e hilos, sin embargo, los beneficios de ejecutar múltiples procesos siempre están ahí, y es indispensable aprovecharse de las arquitecturas modernas que poseen muchos núcleos.

Algo que no se remarcó en este informe, fue las diferencias que conlleva el hardware, Fabián posee un i5-8300H, procesador de 4 núcleos y 8 hilos, 4 GHz, mientras que Rodolfo posee un R7 1700, procesador de 8 núcleos y 16 threads, 3.7 GHz. Estos procesadores demoran cantidades de tiempo distinto en la parte de las matrices, y seria interesante mas adelante ver como la cantidad de cache distinto afecta a las matrices (1700 posee 20MB cache, 8300H 8MB). Esto no se analizó con más detalle debido a temas de tiempo.

Tuvimos algunos problemas con perf, debido a que no todas los eventos estaban soportados en el sistema del R7 1700, esto nos quitó un poco de tiempo, pero no fue mayor.

Finalmente, es interesante que en la parte de las matrices el algoritmo del arreglo auxiliar era un poco más lento que el de la matriz transpuesta, puesto que según lo investigado debería ser un poco más rápido. Esto puede ser por diferencias de la implementación, o el hardware.