



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Estudio del uso de procesadores en aplicaciones de usuario

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Federico Nicolás Patané

Director: David González Márquez

Codirector: Esteban Mocskos

RESUMEN

Durante los últimos años, la tendencia en la tecnología de los procesadores es incrementar la cantidad de núcleos en cada pastilla. Este escenario viene dado por la imposibilidad de aumentar la frecuencia del procesador debido al aumento de energía que esto implicaría. La expectativa implícita es que los desarrolladores de software deben escribir aplicaciones teniendo en cuenta la concurrencia para poder sacar provecho de este cambio de dirección.

El estudio del uso de la concurrencia en los procesadores data de la aparición de los primeros multicores. En trabajos realizados en los años 2000 y 2010 se concluyó que los desarrolladores de software no generan aplicaciones que hagan uso simultáneo de múltiples núcleos de cómputo. El presente trabajo se enfoca en la misma pregunta: luego de más de diez años de la aparición de los procesadores multicore, ¿Han comenzado los desarrolladores a utilizar eficientemente estos recursos disponibles en la mayoría de los dispositivos y máquinas?

Este trabajo se basa en la toma sistemática de trazas de uso de procesador en un laboratorio de computadoras durante jornadas de uso típico por parte de estudiantes de la carrera de Licenciatura en Ciencias de la Computación durante más de cuatro meses. Estos datos son analizados y discutidos buscando evaluar si los múltiples núcleos tienen uso intensivo.

Se estudia el uso concurrente del procesador tanto de manera global, como individualmente por parte de distintas aplicaciones, detectando patrones interesantes de uso. También se estudia cómo afectan las tareas propias del Sistema Operativo y las actividades de entrada/salida.

La conclusión principal de esta tesis es que tanto de manera global como por aplicación individual, salvo casos muy puntuales, el uso concurrente del procesador sigue siendo considerablemente bajo.

Quiero agradecer a los jurados Matías y Facundo.

A David, por haber sido el director de esta tesis, poniendo un esfuerzo y dedicación increíble en el día a día, estando con este trabajo desde el día cero hasta el último. Nunca me van a alcanzar las palabras para agradecerte. Gracias infinitas.

A Esteban por abrirme las puertas de su laboratorio cuando acudí a él preguntándole desesperado qué era una tesis.

A la Universidad de Buenos Aires por brindarme una educación y formación de primer nivel desde lo humano y académico, y de forma gratuita.

A Lucy y Alfredo por siempre guiarme y mostrarme por donde había que ir. A Agostina, mi hermana y mejor amiga, que siempre estuvo al lado mío.

A Carla, mi compañera y motor de vida. Mi incondicional.

A todos mis compañeros y amigos de cursada y a cada uno de los profesores que tuve el placer de tener.

A los amigos de mi vida, Lobo Agus y Martin.

A Marce, Valen, Fran, Kumal y Pelusa que se unieron a mi familia en la parte final de mi carrera.

A Axel que me dio las dos manos cuando más lo necesitaba.

A Alex, Alejandro y Edgar por haber creado junto conmigo al magnífico Segmentation Fault.

A los chicos de soporte del DC por ayudarnos a instalar los scripts en las máquinas de los labos.

ÍNDICE GENERAL

1	Introducción	6
1.1	Trabajo relacionado	11
2	Metodología	14
2.1	Muestras	14
2.2	Uso del procesador por programa	17
2.3	Uso del procesador para <i>entrada salida</i>	17
2.4	Uso del procesador por el Sistema Operativo	18
2.5	Threads en el análisis de los programas	19
2.6	Recolección de datos	20
2.6.1	Arquitectura	21
2.6.2	Funcionamiento del proceso de recolección de datos	21
2.6.3	Configuración de los parámetros de la captura	22
3	Caracterización de la información	24
3.1	Análisis de muestras por rango horario	26
3.2	Análisis de muestras <i>burst</i>	27
4	Validación de la información	30
4.1	Introducción	30
4.2	Experimento 1	31
4.3	Experimento 2	32
5	Resultados	34
5.1	TLP sobre el total de las muestras	34
5.1.1	Categorización de muestras por TLP	35
5.2	TLP sobre conjunto de programas	37
5.2.1	Comparación de programas con publicaciones anteriores	40
5.3	Entrada salida	42
5.3.1	Tiempo de duración de las tareas de <i>entrada salida</i>	42
5.3.2	Tiempo de demora en entrar en actividades de <i>entrada salida</i>	43
5.4	Tareas del SO	43
5.4.1	Tiempo de duración de las tareas del SO	43
5.4.2	Tiempo en el que tarda el SO en poner a correr una de estas tareas	44
5.4.3	Concurrencia (TLP)	45
5.5	Transiciones del procesador	45
5.5.1	Definición de transición	46

ÍNDICE GENERAL	5
5.5.2 Tiempo y cantidad promedio de las transiciones	46
6 Conclusiones	48
7 Trabajo Futuro	49

INTRODUCCIÓN

Desde hace ya varios años, la tendencia en la tecnología de los procesadores es agregar más núcleos (cores) al procesador¹. Esto se debe a la imposibilidad de aumentar la frecuencia del procesador por el aumento en el consumo que implicaría. El límite impuesto por el consumo se denomina *power wall*.

A pesar de esto, las técnicas de producción de procesadores siguen permitiendo aumentar la cantidad de transistores, siendo éstos cada vez más pequeños. Desafortunadamente, también se incrementa el consumo y la disipación de energía. En algunos casos, se han desarrollado costosos y complejos sistemas de refrigeración, pero no resultan aptos las plataformas de cómputo de uso general y, además, son prohibitivos para entornos móviles.

En la figura 1.1 [Rup19] se ilustra la evolución de cinco características principales de los procesadores: cantidad de transistores, rendimiento de un procesador, frecuencia, potencia típica y número de núcleos (cores).

En esta figura se presenta cómo la cantidad de transistores que es posible integrar dentro de una misma pastilla ha crecido exponencialmente desde los años 70. Esto ha permitido contar, en un mismo circuito integrado, con una mayor cantidad de unidades funcionales. Si bien esto permite implementar técnicas agresivas de paralelismo a nivel de instrucciones, estas técnicas tienen sus límites.

Para el año 2005, la tecnología de integración logró integrar una cantidad suficiente de transistores como para poder colocar más de un procesador en la misma pastilla. Sin embargo, a medida que se aumenta la cantidad de núcleos, el rendimiento individual de cada uno disminuye. Esto se debe a que los núcleos deben competir por los recursos del sistema, tanto en términos de memoria cache, unidades funcionales, como también de acceso a la memoria principal. La potencia total que puede disipar el sistema está relacionada con su frecuencia de funcionamiento, lo que termina imponiendo un límite práctico al máximo que puede alcanzar.

Este aumento en la cantidad disponible de núcleos plantea la siguiente pregunta: *¿Estamos usando eficientemente los recursos computacionales disponibles?* Esta pregunta plantea dos ejes de discusión. Por un lado, buscamos maximizar la utilización de los recursos. Si tenemos varios procesadores, lo más eficiente es utilizar todos los disponibles para resolver un mismo problema buscando minimizar el tiempo de ejecución de la aplicación. Por otro lado, el uso eficiente está relacionado con el consumo energético.

¹En este texto, nos referimos a core como uno de los núcleos de procesamiento de la computadora, y por *procesador* al conjunto de los cores.

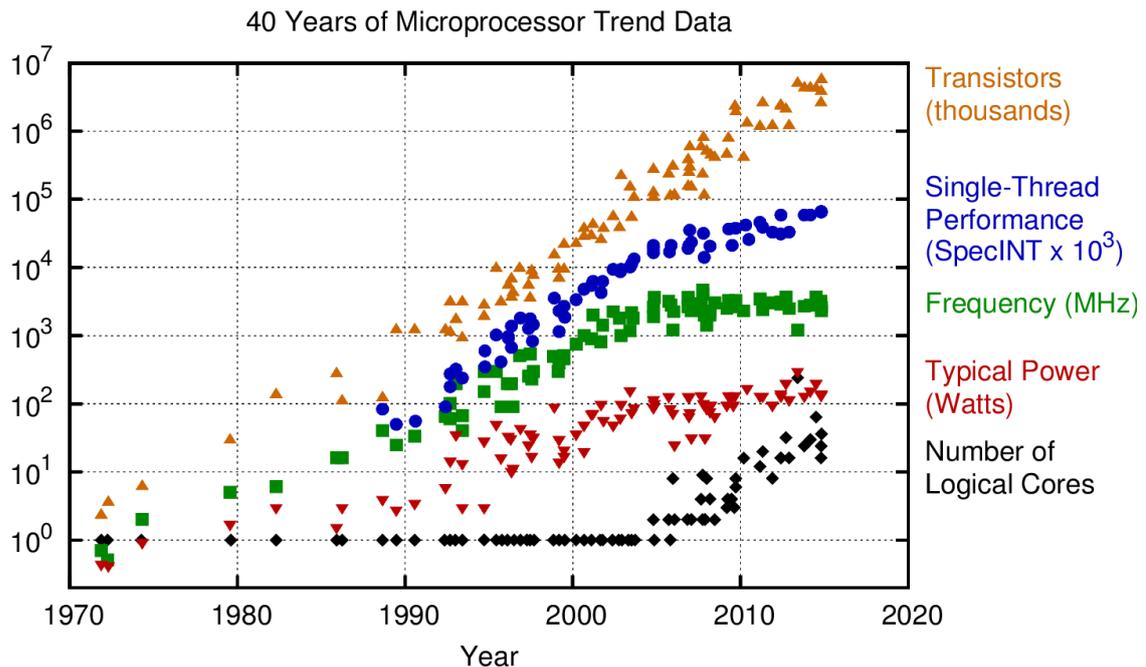


Figura 1.1: Evolución histórica de los microprocesadores. La cantidad de transistores sigue incrementándose, sin embargo la frecuencia de funcionamiento y el desempeño de las aplicaciones *single core* se han detenido en los últimos años.

Utilizar los recursos más adecuados para resolver los problemas, minimizando el consumo de energía. Cualquiera sea el caso, las aplicaciones utilizan los procesadores según su lógica de funcionamiento. Éstas están desarrolladas con una determinada arquitectura que hace uso de la concurrencia entre *threads* con algún criterio. Estudiar cómo las aplicaciones utilizan los procesadores permitiría tener una noción del diseño de las mismas, como así también poder responder la pregunta original.

El diseño de nuevos lenguajes de programación, *frameworks* de desarrollo o arquitecturas de sistemas está reglado por como estas herramientas serán utilizadas en el futuro. Entender cómo las aplicaciones usan los recursos paralelos permitirá construir mejores herramientas para utilizarlos de forma más eficiente.

Además, a nivel del sistema operativo, muchas aplicaciones son ejecutadas de manera concurrente y entre todas comparten los recursos disponibles. Estudiar el uso concurrente a nivel del sistema operativo resulta fundamental para entender cómo los recursos son utilizados por un conjunto de aplicaciones simultáneamente. Lograr una caracterización de este funcionamiento permitiría realizar mediciones para la optimización tanto de políticas de asignación de recursos a nivel de software, como algoritmos implementados en hardware. Éstos últimos controlan las técnicas como *dynamic frequency scaling* [Chu19] o *dynamic voltage scaling* utilizadas para regular el funcionamiento del sistema.

Una *aplicación* es un programa creado con un fin determinado. Un programa puede definirse como una secuencia de pasos escrita en algún lenguaje de programación. Las aplicaciones surgen de alguna necesidad concreta de los usuarios y se utilizan para facilitar o permitir la ejecución de ciertas tareas en las que un analista o un programador ha detectado una cierta necesidad. Una aplicación hace uso de recursos para realizar sus tareas, en tanto que el Sistema Operativo (SO) es el encargado de gestionar y administrar los recursos del sistema entre las distintas aplicaciones que se ejecutan.

Un SO [Tan13] es una pieza de software que se encarga principalmente de dos tareas. La primera es proporcionar a las aplicaciones un conjunto abstracto de recursos, escondiendo los detalles de acceso y proporcionando una interface sencilla para el programador. La segunda es la administración de recursos de una plataforma de cómputo. Las computadoras modernas constan de procesadores, memorias, discos, interfaces de red, impresoras y una amplia variedad de otros dispositivos. El SO proporciona una asignación ordenada y controlada de los procesadores, memorias y dispositivos de *entrada salida*, entre las diversas aplicaciones que compiten por estos recursos.

Supongamos que hubieran tres programas que se ejecutan en cierta computadora. Los tres intentan imprimir sus resultados en forma simultánea en la misma impresora. Las primeras líneas de impresión podrían provenir de la aplicación 1, las siguientes de la aplicación 2, después algunas de la 3, y así en lo sucesivo: el resultado sería un caos y, seguramente, no lo esperado por los usuarios.

El sistema operativo es el encargado de imponer orden al caos potencial, administrando el acceso a la impresora y utilizando *bufers* para almacenar los datos a imprimir a la espera de que el recurso se libere. El SO al detectar que el recurso fue liberado, puede copiar el *buffer* a la impresora, mientras que al mismo tiempo otro programa puede continuar generando más salida, ajeno al hecho de que, en realidad, no se está enviando sus datos a la impresora todavía, sino a un archivo de almacenamiento temporal [Tan13]. Cuando una computadora (o red) tiene varios usuarios, la necesidad de administrar y proteger la memoria, los dispositivos de *entrada salida* y otros recursos es cada vez mayor; de lo contrario, los usuarios podrían interferir unos con otros. Además, los usuarios necesitan, con frecuencia, compartir no sólo el hardware, sino también la información (archivos o bases de datos, por ejemplo). En resumen, esta visión del sistema operativo sostiene que su tarea principal es llevar un registro de qué programa está utilizando qué recursos, de otorgar las peticiones de recursos, de contabilizar su uso y de mediar las peticiones en conflicto provenientes de distintos programas y usuarios [Tan13].

Decimos que un programa se transforma en un proceso cuando se encuentra ejecutando en un *core* asignado. El SO almacena un serie de datos relacionados con los procesos para poder administrarlos. Estos datos incluyen los valores actuales del contador de programa, los registros y las variables. Desde el punto de vista conceptual, el SO hace creer a los procesos que son los únicos utilizando los procesadores. Además, para poder realizar su tarea, un proceso necesita recursos como: tiempo de CPU, memoria, archivos y dispositivos de *entrada salida*.

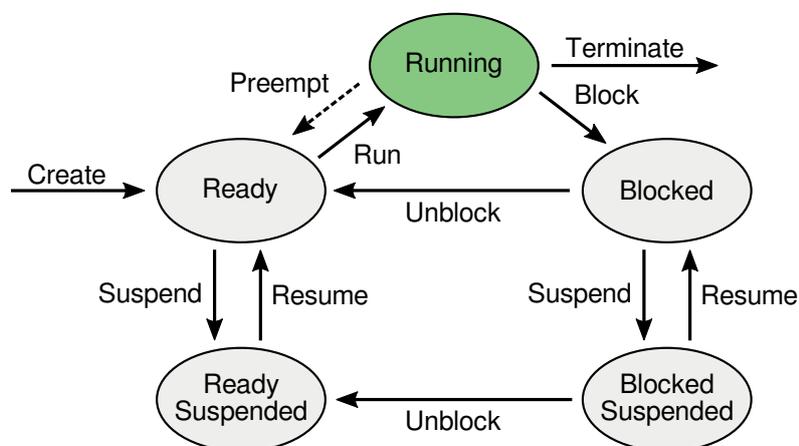


Figura 1.2: [Tan13] Un proceso puede pasar por diferentes estados durante su ejecución. El Sistema Operativo es el encargado de mediar los eventos que generan los distintos cambios.

En la figura 1.2 se presentan los diferentes estados por los que puede atravesar un proceso durante su ejecución. De ellos, vale la pena remarcar los principales:

1. **Ready:** el proceso se encuentra en condiciones de poder ejecutar, el único recurso que le falta es el procesador. A este estado se puede llegar cuando el proceso recién comienza a ejecutar o cuando fue desalojado del procesador por el SO debido a la falta de algún recurso o a que se decidió reemplazarlo por otro proceso (*preemption*).
2. **Running:** el proceso está usando el procesador en ese instante y ejecutando la aplicación.
3. **Blocked:** el proceso no puede ejecutarse, se encuentra bloqueado, ya que está a la espera de cierto evento o recurso (por ejemplo acceder a un dispositivo de *entrada salida*).

El componente del SO que se encarga de la gestión del uso del procesador es el *scheduler*. Éste, a través de aplicar alguna política de *scheduling*, se encarga de decidir cuál es el próximo proceso a ser ejecutado (de los *procesos* que se encuentran en estado *Ready*) y en qué *core* lo hará. También es el encargado de desalojarlos, en caso de que termine su turno de ejecución (tiempo asignado para usar el *core* en forma continua), o en caso de que deba hacerlo por una interrupción de alguna tarea de *entrada salida*. La política de *scheduling* es uno de los componentes clave de un SO y, por lo general, tiene varias políticas entre las cuales seleccionar la que mejor se adecue al uso del sistema. Una optimización en el *scheduler* tiene un potencial impacto considerable en el rendimiento general del sistema.

Algunas de las características principales que se pueden optimizar en un *scheduler* son:

- **Fairness:** se busca que cada proceso reciba una dosis justa del tiempo del procesador.
- **Eficiencia:** se trata de que el procesador esté ocupado la mayor parte del tiempo.
- **Carga del sistema:** el objetivo es minimizar la cantidad de procesos listos que están esperando para acceder al procesador.
- **Rendimiento (*throughput*):** se intenta maximizar el número de procesos terminados por unidad de tiempo.

Para esto, el SO mide cómo el procesador es usado por los procesos generando estadísticas de uso. Éstas, a su vez, son usadas para alimentar al algoritmo del *scheduler*, buscando mejorar la eficiencia del sistema.

Una única aplicación puede utilizar varios procesos. Gracias al SO, las aplicaciones pueden hacer uso de varios *cores* para ejecutar al mismo tiempo. De esta manera, las aplicaciones pueden separar el trabajo a realizar en distintos *cores*, disminuyendo el tiempo total de ejecución.

A pesar de la ventaja de contar con varios recursos disponibles, el desarrollo de aplicaciones concurrentes es una tarea compleja. La etapa de desarrollo puede ser especialmente ardua, ya que pueden aparecer *race conditions* [Tan13] entre los procesos si la sincronización no está bien diseñada. Además, reproducir errores puede ser complicado, visto que éstos pueden aparecer como resultado de una combinación de estados particulares de varios procesos, y normalmente no es factible reproducir exactamente el orden de ejecución y los eventos de una aplicación con varios procesos concurrentes.

A fin de aliviar parte del trabajo de desarrollar programas paralelos, se han desarrollado distintos *frameworks* que ayudan en esta tarea. Ejemplos que han sido adoptados por una gran comunidad de desarrolladores son MPI [MPI19] para memoria distribuida y OpenMP [FG07] para memoria compartida. La memoria distribuida es una abstracción utilizada para compartir datos entre computadoras que no comparten memoria física, es decir, que están conectadas por medio de una red. En nuestro caso, las

aplicaciones consideradas utilizan el paradigma de memoria compartida, con lo que utilizan recursos computacionales en una misma computadora y comparten un único espacio de direccionamiento.

OpenMP se basa en el modelo de *fork-join*, donde la tarea a computar se divide en K *threads* (etapa de *fork*). Luego de realizado el computo por los distintos *threads*, hay una etapa en la que se recolectan sus resultados y se los combina (etapa de *join*).

Un *thread pool* es un patrón de diseño de software para lograr la simultaneidad de ejecución en un programa. Dado que la creación de un *thread* tiene un costo significativo, es usual que una aplicación implemente un *thread pool* que se basa en crear todos los *threads* al comienzo de la ejecución de la aplicación a la espera de que el programa asigne tareas para la ejecución simultánea. Al mantener este conjunto de hilos, el modelo se ahorra el costo de la creación de *threads* aumentando el rendimiento y evitando la latencia en la ejecución debido a la frecuente creación y destrucción de hilos para tareas de corta duración. El número de *threads* disponibles puede ajustarse dinámicamente durante la ejecución de una aplicación en función del número de tareas por realizar. Por ejemplo, un servidor web puede agregar *threads* si llegan varias solicitudes de páginas y puede eliminarlos cuando esas solicitudes disminuyen. Los frameworks para trabajar deben seguir las reglas impuestas por el SO. Como nombramos anteriormente, este tipo de programas opera a nivel de usuario. Por lo tanto no tienen el control de lo que se ejecuta en la máquina, este control pasa por el SO. La aplicación puede crear varios *threads* para acelerar el procesamiento general, pero no puede decidir como asignarlos. Esta tarea pasa exclusivamente por el SO.

La presente investigación se refiere al estudio del uso concurrente de los *cores* de un procesador. Para eso se estudiará el uso del procesador por parte de los procesos y el uso, a nivel global, del mismo por parte del SO.

Lo que intentaremos responder es si, a pesar del aumento del número de *cores*, son realmente usados por los *procesos*, o si se desaprovechan las capacidades de la máquina.

En nuestro trabajo, diremos que un *core* puede estar en dos estados.

- i) *idle*. El *core* no se encuentra siendo utilizado por ningún proceso.
- ii) Corriendo o *running*. El *core* se encuentra siendo utilizado por algún proceso.

Para medir el uso simultáneo del procesador, resulta necesario interactuar con el SO para saber en cada momento qué proceso corre en cada *core*. Esto se realiza a través de la herramienta *perf*. Esta herramienta nos da la posibilidad de acceder a cada evento que sucede sobre el *scheduler*, ordenados por *timestamp*.

Dos trabajos principales son los que dan sustento a la presente tesis: Flautner et al. [FURM00] y Blake et al. [BDMF10]. En ambos se estudió el uso del *procesador* por parte de determinados *procesos* en ambientes controlados de aplicaciones reales. En nuestro caso estudiamos su uso en un ambiente de uso real de la máquina. No nos focalizamos en ningún *proceso* particular, sino que se estudia el uso del *procesador* por cada una de las aplicaciones que utilizan los usuarios. Además, se analiza el uso a nivel global de la cantidad de *cores* ejecutando de manera simultánea.

Además de los dos trabajos mencionados, algunos años más tarde fueron realizados estudios similares sobre dispositivos móviles. En ellos se presenta la misma problemática explicada anteriormente y, de manera similar a lo que ocurre con los equipos de escritorio y servidores, el mercado avanza en incrementar el número de *cores* [BDMF10]. Entre estos trabajos se destacan Gao et al. [GGD⁺14, GGR⁺15] y Halpern et al. [HZR16], quienes se centraron en aplicaciones para dispositivos móviles y cómo utilizan simultáneamente el procesador. En estos trabajos se utilizó la misma metodología que en los trabajos

de Flautner et al. [FURM00] y Blake et al. [BDMF10], realizando distintas mediciones sobre *benchmarks* contruidos manualmente. En la sección 1.1 se explicará con mayor detalle los resultados de estos estudios.

A diferencia de los trabajos anteriores en los que las muestras fueron tomadas sobre *benchmarks* artificiales, en el presente trabajo son tomadas sobre máquinas con un uso real por parte de usuarios reales. Nuestro ambiente experimental está constituido por los laboratorios de alumnos del Departamento de Computación de la Facultad de Exactas de la Universidad de Buenos Aires. La infraestructura disponible totaliza seis laboratorios con alrededor de 20 máquinas cada uno con el sistema operativo Linux.

Los laboratorios son usados, mayoritariamente, por alumnos de las distintas carreras de la Facultad, y no solo de la carrera de computación. Por la mañana es más común el uso de alumnos de carreras como Física, Química o Matemática, quienes usan aplicaciones como R, Octave, entre otras. Luego, por la tarde, es frecuentemente más utilizado por alumnos de la carrera de computación, ya que es horario de cursada de clases en los laboratorios. En este caso, lo que más se utiliza son programas como los editores de texto, IDEs (entorno de desarrollo de aplicaciones), compiladores, etc.

1.1 Trabajo relacionado

Los trabajos de Flautner et al. [FURM00] y Blake et al. [BDMF10] estudian el comportamiento de aplicaciones en sistemas *multicore*. Para el momento del primer trabajo, existían pocos procesadores disponibles con más de un *core*. Los múltiples *threads* se usaban en aplicaciones y contextos muy específicos, además el soporte para desarrollo de aplicaciones paralelas también estaba poco avanzado, siendo un proceso muy artesanal y dificultoso. Ya para el momento en que fue realizado el trabajo de Blake et al., los multicores se habían establecido en el mercado como un estándar. Aún así, este trabajo concluye que “*los desarrolladores de software todavía están en un momento de transición de cambio de pensamiento y diseño en sus programas, dado al abrupto cambio que implicó pasar de máquinas de un solo thread para chips multiprocesadores.*”. Por otro lado, el trabajo de Flautner et al. observa que “*hemos notado que el uso del procesador se encuentran actualmente desbalanceado*”.

Ambos trabajos estudian cómo un conjunto de aplicaciones particulares hacen uso de los procesadores disponibles, exponiendo el nivel de uso de múltiples hilos de ejecución. Flautner et al. introducen el **TLP** para cuantificar la concurrencia de los programas. Esta misma medida es también utilizada luego por Blake años más tarde.

El **TLP** (*thread level parallelism*) se calcula tomando la fracción de tiempo que **exactamente** i *cores* estuvieron ejecutando simultáneamente multiplicado por i . i varía entre 1 y la cantidad de *cores* de la máquina. Por último, se lo divide por la fracción de tiempo en que fue utilizado algún *core*, quedando definido como:

$$\text{TLP} = \frac{\sum_{i=1}^n C_i \cdot i}{(1 - C_0)}$$

donde C_i es la fracción de tiempo que se utilizan exactamente i *cores* simultáneamente y C_0 es la fracción en la que los *cores* no son utilizados. Por ejemplo, en un sistema con cuatro *cores*, si se utilizarán todos durante todo el tiempo, el **TLP** obtenido sería cuatro. Cualquier valor menor a éste significaría que, durante un cierto tiempo, uno o varios *cores* no tuvieron tareas asignadas. **TLP** será explicado en detalle en la sección 2, dónde introduciremos la forma en que se utiliza dicha el **TLP** en la presente tesis.

Los trabajos de Blake et al. y Flautner et al. se basan en el estudio de *benchmarks* de aplicaciones reales realizados a medida y en ambientes controlados. Es decir, los *benchmarks* reflejan algún uso particular de la aplicación establecido por el propio estudio, siendo el mismo estudio el que determina cuál es

el conjunto de acciones típicas que realiza una aplicación. En nuestro caso, el estudio se realiza sobre muestras de uso real de la computadora por parte de usuarios reales.

Años más tarde a los dos trabajos mencionados, fueron realizados estudios similares pero sobre dispositivos móviles. En ellos se presenta la misma problemática explicada anteriormente respecto al crecimiento en el uso de recursos de cómputo y, al igual que con las computadoras de escritorio, la solución del mercado es el incremento del número de *cores*. Los siguientes tres estudios, [GGD⁺14, GGR⁺15, HZR16], se centraron en aplicaciones sobre dispositivos móviles y cómo las mismas utilizan simultáneamente el procesador. En estos trabajos se utilizó la misma metodología que en los trabajos de Flautner et al. y Blake et al, midiendo sobre *benchmarks* construidos manualmente. A pesar del paso del tiempo y del aumento del número de *cores*, en todos los trabajos se pudo constatar un bajo uso paralelo de parte de las aplicaciones.

En Gao et al. [GGD⁺14] se evalúa el **TLP** de varios tipos de aplicaciones, como por ejemplo navegadores de Internet (*browsers*), programas de video, programas de música, juegos, redes sociales, y programas de oficina. Los resultados indican que las aplicaciones presentan un **TLP** limitado. Para la mayor parte de las aplicaciones expuestas en el trabajo, se observa un **TLP** no superior a 1,2, pero considerando que se utiliza una plataforma de cuatro núcleos, resulta bastante decepcionante. En promedio, el **TLP** alcanza 1,4, mientras que para aplicaciones con alto **TLP**, como juegos o *browsers*, el **TLP** logrado va de 1,5 a 1,6. Otras aplicaciones, como programas de reproducción de música y el navegador de archivos, presentan un **TLP** bajo que se encuentra entre 1,2 y 1,3.

Además, sobre los mismos experimentos, se realizan pruebas aumentando el número de *cores*, a fin de observar su impacto sobre el **TLP**. En promedio, el **TLP** aumenta un 4,5 % cuando se incrementa el número de *cores* en un sistema de dos a tres *cores*. Cuando se pasa de un sistema de tres *cores* a otro de cuatro, el incremento es un marginal 3,1 %. Con esto se concluye que el incremento del número de *cores* no impacta de gran manera sobre el **TLP** de las aplicaciones.

Otro de los experimentos fue realizar el mismo conjunto de pruebas sobre procesadores a distintas frecuencias. Con esto se busca cuantificar la variación de rendimiento en función de la frecuencia del procesador.

En todas las aplicaciones se advierte una disminución del **TLP** con procesadores a mayor frecuencia. En el caso de la aplicación de edición de imágenes, el **TLP** permanece sin cambios.

Ninguna de las aplicaciones tiene un **TLP** mayor a 1,5. Esto indica que los procesadores con más frecuencia tienden a lograr menos **TLP**.

Por lo tanto, estos resultados indican que el **TLP** no aumenta en la misma proporción en que lo hacen la cantidad de *cores* disponibles. Los procesadores continuarán siendo subutilizados si los desarrolladores de software no producen programas que utilicen eficientemente los recursos paralelos.

Luego en otro trabajo de Gao et al. [GGR⁺15] al año siguiente, se realizan estudios equivalentes mostrando resultados similares. Además del estudio del **TLP**, este trabajo amplía la propuesta al medir el uso de *GPU* en las aplicaciones que utilizan de forma sistemática la aceleradora gráfica (como por ejemplo los juegos), y el consumo de energía por parte de las aplicaciones.

En Halpern et al. [HZR16] se amplía la segunda parte del estudio realizado por Gao [GGR⁺15]. A medida que el hardware continuó creciendo, se comenzó a consumir una mayor cantidad de energía. Esto impactó directamente en la duración de la carga de la batería del celular, variable que resulta crucial para la satisfacción del usuario final. Por ejemplo, en el trabajo se muestra que el modelo SAMSUNG Galaxy S4 (del año 2013) es uno de los primeros celulares con un gran salto en hardware. En ese momento, los

desarrolladores de software móvil no estaban preparados aún para este cambio, por eso este aumento del hardware también influyó en un gran incremento en el consumo de energía de los celulares por parte de las aplicaciones. A partir de esto, los desarrolladores comenzaron a adaptarse a esta dificultad, mejorando el desempeño energético de las aplicaciones. Esto da por resultado que sin tener que disminuir las bondades del hardware, modelos posteriores comienzan a solucionar este problema.

En todos estos trabajos se pone de manifiesto el bajo uso paralelo que las aplicaciones hacen del procesador y ponen en un primer plano el necesario cambio de paradigma de programación que los desarrolladores deben realizar para poder explotar los recursos disponibles. En este trabajo, se utilizarán varias de las ideas en los trabajos anteriores, especialmente el **TLP** y los resultados obtenidos en Blake y Flautner.

METODOLOGÍA

2.1 Muestras

En este trabajo se miden distintos aspectos del sistema a partir de capturas de uso de procesadores en un ambiente real por parte de usuario reales. Las muestras son capturadas utilizando la herramienta `perf`. La misma existe solamente en entornos Linux a partir del *kernel* versión 2.6.31.

Esta herramienta de análisis de rendimiento permite acceder a diferentes elementos dentro del *kernel*, generando eventos asociados a una marca temporal.

Admite la captura de contadores de rendimiento de hardware, software, y *tracepoints* para todos los módulos del *kernel* con distinto nivel de detalle.

Se usó esta herramienta debido a su capacidad y flexibilidad para capturar eventos, particularmente a nivel de *scheduler*, sin tener que llegar a modificar el *kernel*. En este trabajo resulta fundamental comprender con precisión qué tarea está siendo ejecutada dentro de cada procesador, para esto es necesario generar eventos que describan este comportamiento: cuándo una tarea es asignada a un procesador y cuándo ésta es desalojada. Esta información solo está disponible mediante el acceso a nivel de *kernel*.

Cualquiera de los eventos que se pueden capturar respetan la siguiente estructura:

```
<name> <pid> [<core>] <timestamp> <event>
prev_comm=<name> prev_pid=<pid> prev_prio=<priority> prev_state=<state> ==>
next_comm=<name> next_pid=<pid> next_prio=<priority>
```

- `<name>`: Nombre del proceso que se encuentra corriendo.
- `<pid>`: Identificador numérico del proceso (otorgado por el SO) que se encuentra corriendo.
- `<core>`: identificador del core donde se genera el evento.
- `<time>`: Timestamp del Sistema Operativo.
- `<event>`: Tipo de evento.
- `prev_x` → `next_x`: *estado anterior* y *próximo proceso*. Listas con datos del proceso desalojado y el proceso a ejecutar.

Si bien `perf` permite la captura de distintos eventos, utilizaremos los siguientes:

- `switch`: se interpreta como un intercambio de tareas sobre un procesador. Diremos que en el tiempo `<time>` el procesador `<core>`, pasó de ejecutar la tarea `<name>` a ejecutar una nueva tarea identificada en los datos de `<next>`. Se considera que este intercambio sucede instantáneamente, sin tener en cuenta el tiempo necesario para desalojar y cargar las tareas.
- `iowait`: este evento es utilizado cada vez que una tarea comienza a esperar por *entrada salida*.
- `wakeup`: cuando se termina su tarea asignada de *entrada salida* se genera el evento `wakeup`. En la sección 2.3 se explican en detalle los eventos relacionados a actividades de *entrada salida*.

En el contexto de este trabajo, se considera una **muestra** como una lista ordenada en forma cronológica de todos los eventos sucedidos dentro del *scheduler* durante la ejecución de la herramienta `perf`.

Dependiendo de los eventos sucedidos durante la recolección de las muestras, un procesador se puede encontrar en dos estados posibles: *activo* o *inactivo*. Este estado será definido independientemente para cada procesador en función de qué tarea tenga asignada. El estado *activo* se refiere a cuando el `core` se encuentra con un proceso asignado. El estado *inactivo* es cuando tiene asignado la tarea *idle*.

Existen, además, tareas especiales denominadas *swapper* que figuran como una tarea, pero en realidad su funcionamiento es equivalente a una tarea *idle* que no realiza ningún trabajo. Por lo tanto, en las muestras los procesadores tienen asignadas tareas en todo momento, aunque algunas de éstas representan el “no estar haciendo nada”.

A continuación se presenta un ejemplo de la interpretación de un fragmento de una muestra, en el mismo se identifica cuándo un procesador pasa al estado inactivo y luego al estado activo.

```

...
chrome 4614 [000] 592.954342 segundos sched:sched_switch
prev_comm=chrome prev_pid=4614 prev_prio=120 prev_state=S ==>
next_comm=swapper/0 next_pid=0 next_prio=120
...
swapper 0 [000] 592.954697 segundos sched:sched_switch
prev_comm=swapper/0 prev_pid=0 prev_prio=120 prev_state=R ==>
next_comm=nautilus next_pid=4082 next_prio=120
...

```

En el tiempo 592.954342 segundos se realiza un cambio de contexto en el que se desaloja al proceso `chrome` identificado por el número 4614 del procesador [000], y se comienza a ejecutar el proceso `swapper` indicado por `next_comm`. Este cambio indica que el procesador pasa al estado inactivo.

El siguiente evento sobre el procesador [000], se genera en el tiempo 592.954697. Éste indica que el proceso `swapper` es desalojado y se comienza a ejecutar el proceso `nautilus`.

Entre estos dos eventos se puede calcular que durante 355 μ s el procesador [000] se encontró inactivo. Notar que cuando el `<core>` [000] entra en estado *idle* ejecutando la tarea `swapper0`. Esto es debido a que cada `<core>` posee su tarea *idle* propia, con su número identificatorio correspondiente.

Siguiendo la lógica anterior, es posible obtener cuánto tiempo fue asignado un procesador a cada tarea, incluso cuántos procesadores se encuentran activos durante un intervalo de tiempo. La figura 2.1 presenta un ejemplo de este cálculo realizado directamente por `perf` mediante la opción `map`. En la misma

se identifica el tiempo de cada evento y el momento en que cada tarea es asignada a un procesador. En la columna *current usage* se indica de forma gráfica cuál es el número de procesadores activos simultáneamente. En la sección central, la asignación de tareas a cada procesador en función del tiempo, y además cada evento de la muestra es indicado por el símbolo asterisco (*).

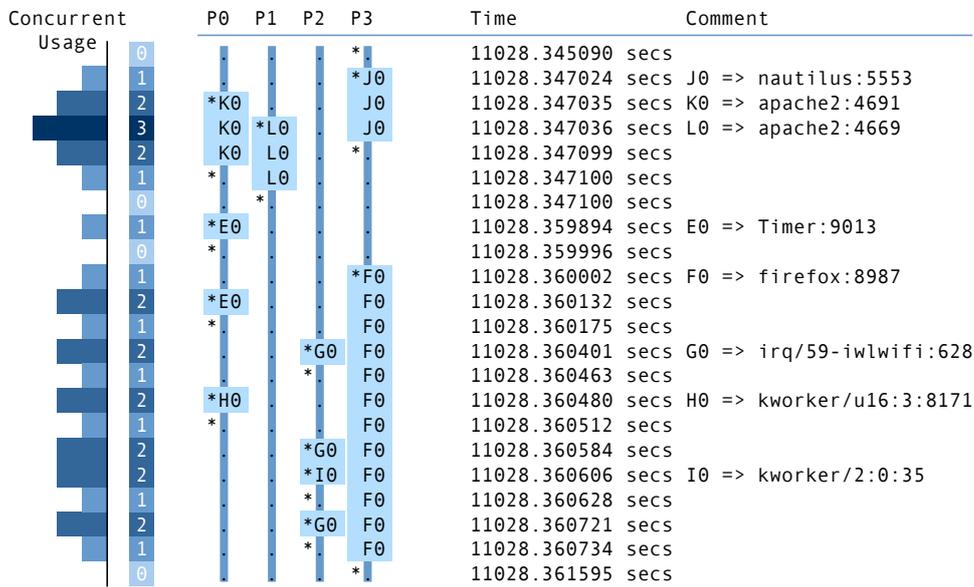


Figura 2.1: Ejemplo de los resultados de `perf` mostrados en forma de *map*. Se puede notar la lista de eventos con su *timestamp* y la indicación de cuál es cada proceso.

Para calcular el uso simultáneo de procesadores se debe determinar durante cuánto tiempo fueron asignados simultáneamente una cantidad dada de procesadores. Es decir, para cada muestra se genera el conjunto de fracciones de uso $CS = \{C_0, \dots, C_n\}$, donde C_i representa la fracción en la que se utilizaron exactamente i procesadores en forma simultánea. Si la muestra fue generada y procesada adecuadamente, $\sum_{i=0}^n C_i = 1$.

El valor de C_i es equivalente a $\frac{t_{C_i}}{t_M}$, donde t_{C_i} es el tiempo en segundos que i *cores* ejecutaron simultáneamente alguna tarea y t_M es el tiempo total de la muestra en segundos. Es importante notar que C_0 corresponde al porcentaje en que la muestra estuvo sin consumir recursos del procesador, es decir, no tuvo ninguna tarea asignada. De forma equivalente, se dice que los procesadores estuvieron ejecutando tareas *idle*: $C_0 = 1 - (C_1 + \dots + C_n)$.

Para evaluar el uso del procesador en una muestra de tiempo de duración t_m , se utilizarán las siguientes magnitudes:

- **TLP_{idle}**: Cuantifica el uso simultáneo de *cores*, incluyendo el tiempo en que ninguno está en uso. Aumenta su valor a medida que todos los *cores* del sistema se usan simultáneamente. Su definición es:

$$\text{TLP}_{\text{idle}} = \sum_{i=1}^n C_i \cdot i \quad (2.1)$$

- **uso compartido**: Indica la cantidad de tiempo en que al menos un *core* tuvo asignado algún proceso, se lo calcula como:

$$\text{uso compartido} = 1 - C_0. \quad (2.2)$$

- **TLP**: En este caso se considera el uso simultáneo de los *cores*, teniendo en cuenta solo el tiempo en que algún *core* estuvo asignado. Se define como:

$$\mathbf{TLP} = \frac{\mathbf{TLP}_{\text{idle}}}{(1 - C_0)} = \frac{\sum_{i=1}^n C_i \cdot i}{(1 - C_0)} \quad (2.3)$$

En el marco de este trabajo, las muestras son tomadas, almacenadas y luego analizadas posteriormente. Se busca clasificar las muestras en base al perfil del uso del procesador. Aquellas muestras que presenten un uso limitado los recursos computacionales serán consideradas como *idle*, es decir, que no se han ejecutado suficientes procesos durante su duración. El umbral de utilización considerado es de 0,05 de **uso compartido**, las muestras que superen este valor serán consideradas como *útiles*, las otras se descartan directamente.

2.2 Uso del procesador por programa

Un aspecto a analizar sobre el uso de procesadores es el comportamiento de los programas de usuario con respecto al nivel de concurrencia alcanzado. Dada una muestra, se obtiene el conjunto de programas que fueron ejecutados durante la misma, conjuntamente con el nivel de paralelismo alcanzado por cada uno de éstos. Es decir, por cada muestra se obtiene el conjunto de programas $PS = \{P_1, \dots, P_i, \dots, P_n\}$ que se ejecutaron al menos una vez. Luego para cada $P_i \in PS$ se calcula el **uso compartido** y **TLP** tal como fueron definidos anteriormente. Notar que esto nos brinda una noción de comportamiento del programa, analizándolo sobre los momentos en el que el programa estuvo siendo ejecutado, no sobre el total de tiempo de la muestra. Generalmente, la cantidad de tiempo en la que un programa P_i se encuentra ejecutando es considerablemente menor que la duración total de la muestra. Entonces, se llevaron adelante dos enfoques para el estudio del uso del procesador:

- Uso Global**: se consideran todas las aplicaciones que son ejecutadas durante la muestra y se analiza la utilización de los recursos por todas las aplicaciones que están en el sistema.
- Por aplicación**: se centra en el comportamiento de cada aplicación que aparece ejecutada en la muestra y se analiza su perfil de uso de los procesadores. Está claro que el comportamiento de las aplicaciones se verá fuertemente afectado por el tipo de uso que hace el usuario, ya que las muestras son obtenidas en base a la interacción real de una persona con el sistema y no por medio de un *benchmark* creado artificialmente.

2.3 Uso del procesador para *entrada salida*

Cada vez que una aplicación recurre a los servicios provistos por el sistema, éste resuelve su pedido respetando una política previamente establecida. Algunos de estos pedidos requieren acceder fuera de la memoria del sistema, generando así eventos de *entrada salida*.

El *scheduler* genera el evento `iowait` cada vez que una tarea comienza a esperar por *entrada salida*. En principio, esta información proporcionada por el *scheduler* puede diferir del momento exacto en que se está realizando el acceso a *entrada salida*, ya que el evento indica el cambio de estado de un proceso y no el momento efectivo en que sucede el evento. Considerando esto, cuando se indica un evento de `iowait` sobre un proceso P en un *core* C en el tiempo T_i , diremos que en C el proceso P comienza a realizar tareas de *entrada salida* en el momento T_i . Luego, cuando recibimos el evento `wakeup` sobre el

proceso P en el core C en el tiempo t_f , diremos que el proceso ejecutó una tarea de *entrada salida* por el lapso de tiempo determinado por la expresión $t_f - t_i$.

...	Del cálculo del tiempo de duración de los eventos tenemos el conjunto 50 s, 110 s y 60 s. Éstos, son los tiempos en que cada acción de <i>entrada salida</i> está consumiendo efectivamente recursos. Utilizando la misma secuencia del ejemplo, medimos la duración de cada evento de <i>entrada salida</i> . En cualquiera de los procesadores, tendremos el conjunto 10 ms, 90 ms y 60 ms, que viene dado por la diferencia {110 - 100; 200 - 110; 260 - 200}. Estos últimos, son los tiempos en los que comienzan a realizar <i>entrada salida</i> alguno de los procesos. Para saber cuánto tiempo se demoró en volver a realizar tareas de <i>entrada salida</i> , le restamos el momento en el que el último proceso comenzó a realizar tareas de esta índole. Notar que para el primer evento de la lista (C1, iowait, chrome, T=100) no es posible saber cuándo fue el último momento en el que algún proceso comenzó a realizar tareas de <i>entrada salida</i> debido a que ese evento no forma parte de la muestra por lo que no lo es considerado para el análisis.
↓	
T=100 C=C1 E=iowait P=chrome	
↓	
T=110 C=C2 E=iowait P=firefox	
↓	
T=150 C=C1 E=wake P=chrome	
↓	
T=200 C=C3 E=iowait P=atom	
↓	
T=220 C=C2 E=wake P=firefox	
↓	
T=260 C=C3 E=iowait P=atom	
↓	
...	

T equivale al timestamp <time> en el que el evento ocurre.

C es el core <core> que se ve afectado por el evento.

E es el nombre <event> del evento.

P es el nombre <name> del programa afectado por el evento.

Figura 2.2: Ejemplo del cálculo de tiempos de actividades de *entrada salida*

Además, una vez que se termina de ejecutar esta espera, el programa vuelve a estar disponible para que el *scheduler* pueda volver a ejecutarlo. El evento de *entrada salida* indica que se está realizando una tarea de *entrada salida*, pero no especifica cuál es. Este evento pudo haber sido, desde leer o escribir en memoria secundaria, hasta esperar por una tecla. Incluso existen eventos que no son de *entrada salida* que son tomados como si lo fuesen. Por ejemplo, si se accede a un *buffer* para traer un dato, como podría ser una tecla, también será considerado como *entrada salida*.

Sobre los eventos de *entrada salida*, nos enfocaremos en dos aspectos:

- Tiempo de duración de los eventos de *entrada salida*.
- Tiempo entre dos eventos de *entrada salida*.

En la figura 2.2 se ilustra una secuencia simplificada de eventos desde los que se extraen las mediciones de actividades de *entrada salida*.

2.4 Uso del procesador por el Sistema Operativo

El SO *linux*, sobre el que estamos trabajando, además de ejecutar como *kernel* del sistema, hace uso de los procesadores por medio de tareas distinguidas denominadas *kworkers*. Uno de los objetivos de este

trabajo es estudiar cómo estas tareas utilizan de manera simultánea al procesador. Como no resulta posible saber qué es lo que está realizando cada una de estas tareas individualmente, se las evaluará como indistinguibles entre sí.

...	
↓	
T=210 C=C4 E=switch P=chrome3	Considerar que los tiempos de ejecución de las tareas del SO vienen dados por el conjunto 20 s, 40 s, 30 s y 50 s. El mismo proviene de la ejecución de los procesos <code>kworker1</code> , <code>kworker2</code> , <code>worker3</code> y <code>kworker4</code> .
↓	
T=120 C=C2 E=switch P=kworker1	Utilizando la misma secuencia del ejemplo, medimos el tiempo entre cada ejecución de los procesos del SO. El mismo es el conjunto 10 ms, 20 ms y 10 ms, que viene dado por la diferencia de $\{130 - 120, 150 - 130, 160 - 150\}$.
↓	
T=130 C=C3 E=switch P=kworker2	Notar que en el caso C2, <code>switch</code> , <code>kworker1</code> , T=120 no sabemos cuándo fue la anterior tarea del SO que comenzó a ejecutar en alguno de los procesadores, por lo tanto no será considerada en el análisis.
↓	
T=140 C=C2 E=switch P=swapper	
↓	
T=150 C=C1 E=switch P=kworker3	
↓	
T=160 C=C4 E=switch P=kworker4	
↓	
T=170 C=C3 E=switch P=swapper	
↓	
T=180 C=C1 E=switch P=atom	
↓	
T=210 C=C4 E=switch P=chrome3	
↓	
...	

Figura 2.3: Ejemplo del cálculo de tiempos de tareas del SO.

De esta forma, nos centraremos en el estudio de cómo el SO utiliza concurrentemente el procesador, en vez de hacerlo sobre cada *kworker*. Sobre este tipo de tareas nos centraremos en tres casos a analizar:

- Cuál es el tiempo neto que corre una tarea del SO.
- Cada cuánto tiempo comienza una tarea del SO a ser ejecutada.
- Concurrencia (**TLP**).

Con la consideración de estos aspectos, podremos analizar cuánto es el tiempo que el SO ejecuta tareas propias y cada cuánto tiempo el *scheduler* les otorga tiempo de procesador. La figura 2.3 reproduce una secuencia simplificada de eventos como ejemplo de lo considerado para este análisis.

2.5 Threads en el análisis de los programas

Se ha encontrado en las muestras que las aplicaciones pueden generar *threads* que no tengan el mismo nombre identificador que el nombre de la aplicación misma. Es decir, que `perf` los identifica de manera diferente. El programa `perf` no tiene parámetros en sus eventos que nos permita hacer esta asociación.

Por ese motivo, para detectar el nombre de los *threads* de un programa, deberá ser ejecutado en un ambiente controlado, y realizar los siguientes pasos:

1. Crear un `cgroups` con un sólo `core`.
2. Ejecutar en el `cgroup` la aplicación a estudiar.
3. Capturar el árbol de dependencias entre procesos.
4. Identificar los nombres de todos los procesos.
5. Simultáneamente ejecutar `perf`.
6. Identificar los nombres que captura `perf` con lo capturados desde el árbol de dependencias entre procesos. Todos los procesos que dependan del programa serán considerados *threads* que el mismo programa crea.

En el capítulo 5 (página 34), utilizaremos este método para determinados programas. Será realizado con el fin de poder estudiar el uso del procesador de una manera más exhaustiva y poder compararlos con los programas estudiados en los trabajos de Blake et al. y Flauner et al.

2.6 Recolección de datos

Para el funcionamiento del proceso de recolección de datos, se construyó una arquitectura capaz de almacenar las muestras, como así también servir de fuente de configuración. Además se desarrolló un *script* encargado de tomar muestras y comunicarse con el servidor central para poder llevar a cabo el almacenamiento de las mismas.

La figura 2.4 izquierda ejemplifica un intercambio de mensajes entre el script corriendo en una máquina y el servidor. Por otro lado, la figura 2.4 derecha muestra una ilustración que representa a la infraestructura de la red de los laboratorios. Cada laboratorio cuenta con un *switch* que interconecta las máquinas instaladas en el mismo. De todos los laboratorios sale un único *link* a un *switch* central que lleva el tráfico hacia los servidores del departamento. A este *switch* central está conectado el servidor utilizado para almacenar las muestras recolectadas.

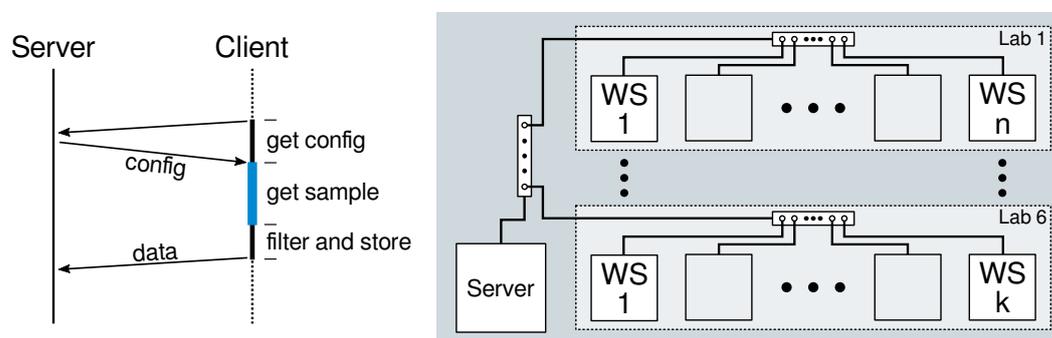


Figura 2.4: Ejemplo del trabajo realizado por parte de cada una de las máquinas de los laboratorios. La parte izquierda representa un intercambio de mensajes entre la aplicación que almacena los datos para posterior análisis y el cliente liviano que interactúa con las aplicaciones ejecutándose en cada máquina. En la parte derecha se representan los distintos laboratorios que fueron utilizados durante la recolección de datos para este trabajo.

2.6.1 Arquitectura

Para la recolección de la información se siguió una arquitectura de tipo cliente/servidor. Las máquinas de los laboratorios están conectadas todas a un único *switch*. En éstas, los usuarios se autentican y ejecutan sus aplicaciones.

Cada máquina ejecutará el *script* de recolección que, principalmente, cumple dos tareas:

1. Conectarse al servidor para descargar los datos de configuración, que incluyen si se debe tomar datos en ese momento y durante cuánto tiempo.
2. En caso de tener que tomar datos, realizar la toma y almacenar la muestra en el servidor central.

Además el servidor central, será el encargado de:

- Almacenar todas las muestras que sean tomadas en las máquinas de los laboratorios.
- Servir como fuente de configuración para las máquinas. En el momento de ejecución del *script* por parte de una máquina, se deberán consultar al servidor dos datos.
 - *Configuración del tiempo de muestreo*. Cantidad de tiempo en segundos que se debe ejecutar el comando `perf` para la muestra.
 - *Configuración del laboratorio*. Variable para indicar si las máquinas de un determinado laboratorio deben recolectar datos en el momento de la ejecución del *script*. Esta variable nos permite elegir los momentos en los cuáles realizar la toma de los datos. Por lo tanto, podremos recolectar información en situaciones de un intenso uso de las máquinas, por ejemplo cuándo hay clases prácticas de laboratorios, o cuándo se está en épocas de entregas de trabajos prácticos.

La arquitectura elegida fue de tipo cliente/servidor ya que el servidor no podría iterar todas las máquinas para comunicarles la acción a realizar. Podría suceder que alguna computadora se encontrase apagada, lo cuál el servidor no lo sabría. También al contar con tantas computadoras, sería muy costoso que el servidor haga esto. Además en nuestro trabajo debía ser fácil actualizar variables en común para la configuración de las tomas. Por este motivo, estos datos debían estar en el servidor, y que las máquinas se conecten a ella. Por último la otra ventaja que nos permite éste modelo de arquitectura, es la facilidad para agregar máquinas a la infraestructura. Con sólo instalar el *script* en la misma, ya puede comunicarse con el servidor para dejar los datos.

2.6.2 Funcionamiento del proceso de recolección de datos

En cada una de las máquinas de los laboratorios, se instaló el *script* encargado de realizar las tareas de recolección de datos. El mismo será el encargado de establecer comunicación con el servidor central para determinar que acción tomar, realizar la muestra, y posteriormente almacenar esta muestra en el servidor.

En la figura 2.5 se ilustran los pasos del *script*.

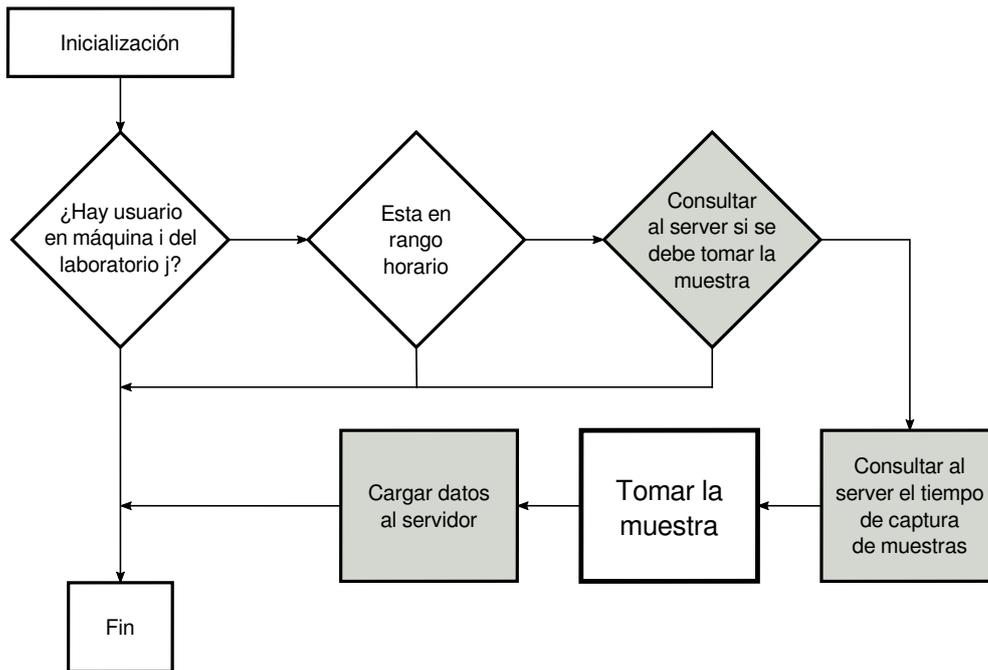


Figura 2.5: Ejemplo del trabajo realizado por parte de cada una de las máquinas de los laboratorios.

1. *¿Hay usuario en máquina i del laboratorio j ?*: Se confirma que en el momento de la toma se encuentra un usuario utilizando la computadora, caso contrario se finaliza.
2. *¿Esta en rango horario?*: Se chequea si la hora actual esta entre las 10 AM y 22 PM, rango horario en los que se encuentran los laboratorios abiertos al público. En caso contrario se finaliza.
3. *Consultar en el server si se debe tomar la muestra*: La máquina deberá chequear en el servidor si su laboratorio está configurado para tomar datos. En caso contrario se finaliza.
4. *Consultar al server el tiempo de toma de muestras*: La cantidad de tiempo de la muestra es configurable, por lo que la máquina deberá chequear cuál es el tiempo de toma de la muestra.
5. *Toma de la muestra*: Ejecución del comando `perf` durante el tiempo conseguido en 4.
6. *Cargar datos al servidor*: Cargar la muestra recientemente tomada al servidor respetando la siguiente nomenclatura: *Laboratorio_i/Maquina_j/Fecha/fecha hora.data*, donde i es el laboratorio al cual pertenece la máquina, y j es la *jesima* computadora del laboratorio i . Esta nomenclatura nos permite conseguir la metadata de las muestras, como por ejemplo laboratorio u hora de la muestra.

2.6.3 Configuración de los parámetros de la captura

El *script* es ejecutado en cada máquina cada cinco minutos, con privilegios de supervisor, que son necesarios para poder ejecutar el programa `perf`.

La frecuencia de tiempo de la toma fue definido para que todas las máquinas alcancen a enviar todas sus muestras por la red hacia el servidor.

Cada muestra, es una traza tomada por el programa `perf`, durante 10 segundos. Los archivos de las muestras pueden llegar a ser muy grandes. Al tomar muestras de 10 segundos, los archivos generados

tuvieron un tamaño promedio de 5,4 MB. Este tamaño es razonable para poder enviar por la red interna al servidor, sin saturarla.

Calculando un total de 20 máquinas por laboratorio, teniendo en cuenta que los scripts fueron instalados en tres laboratorios y, calculando que un archivo tarda 5 s en ser enviado al servidor, tardaríamos un total de cinco minutos en enviar todas las muestras.

Además, este proceso debe ser totalmente imperceptible para el usuario. Para eso fue importante la elección de la frecuencia y duración de la toma. Una vez que el proceso de recolección de datos comenzó a operar, fueron consultados distintos alumnos si se notó algún cambio en el rendimiento y respuesta de las computadoras, para poder así asegurarnos que esto no afectase al correcto funcionamiento de los laboratorios.

CARACTERIZACIÓN DE LA INFORMACIÓN

En el presente capítulo se presenta el estudio de las características de la información capturada en los laboratorios de la Facultad.

Las máquinas de los laboratorios cuentan con hardware diferente, tanto en procesadores, como placa de video o discos. Pero todas las máquinas utilizan procesadores de la marca Intel en distintos modelos. A pesar de esto, en todos los laboratorios se tienen procesadores de cuatro cores reales por máquina. Por eso cada vez que nos refiramos a los mismos en este trabajo, sabremos que siempre se encuentran fijos en cuatro.

El total de las muestras fueron tomadas en el trascurso del 29 de Junio del 2017 al 5 de Octubre del 2017, contabilizando un total de 99 días. En este rango de fechas se pudieron capturar un total de 20524 muestras *útiles*. Éstas representan al 23 % del total de las muestras capturadas, siendo estas aproximadamente 89000 muestras. Considerando que cada muestra se captura cada 5 minutos, las muestras *útiles* representan 280 horas de uso útil de las máquinas por parte de los usuarios, del total de 1200 horas aproximadamente. Teniendo en cuenta que cada muestra equivale a una toma de 10 segundos, la captura de entre todas las muestras equivalen a 56 horas de uso neto de trazas del *scheduler*.

De las muestras *útiles*, el porcentaje de muestras que tienen un TLP_{idle} menor a 1 (considerando el tiempo *idle*) es del 73,63 %. Las muestras con TLP_{idle} entre el 1 y el 2 son el 21,40 %, entre 2 y 3 el 3,05 % y mayor a 3 el 1,95 %.

Además, el promedio del **uso compartido** del procesador alcanza los 0,4, mientras que el promedio del TLP_{idle} logra llegar a 0,57.

En la figura 3.1 se muestra la distribución obtenida del cálculo de las dos medidas mencionadas anteriormente: TLP_{idle} y **uso compartido**.

Teniendo en cuenta que el TLP_{idle} siempre está acotado inferiormente por el **uso compartido**, vemos que el promedio del TLP_{idle} es sólo un poco más grande que el caso del **uso compartido**. Al ser el TLP_{idle} un promedio ponderado del **uso compartido**, la cercanía de estos dos valores en las distribuciones, nos hace ver que en el caso del valor del TLP_{idle} predomina el peso del uso de 1 sólo core. Es decir que el uso paralelo es muy bajo, por eso es que el promedio de TLP_{idle} y **uso compartido** son parecidos. En caso de que no fuera así, el TLP_{idle} promedio debería diferenciarse más del **uso compartido**.

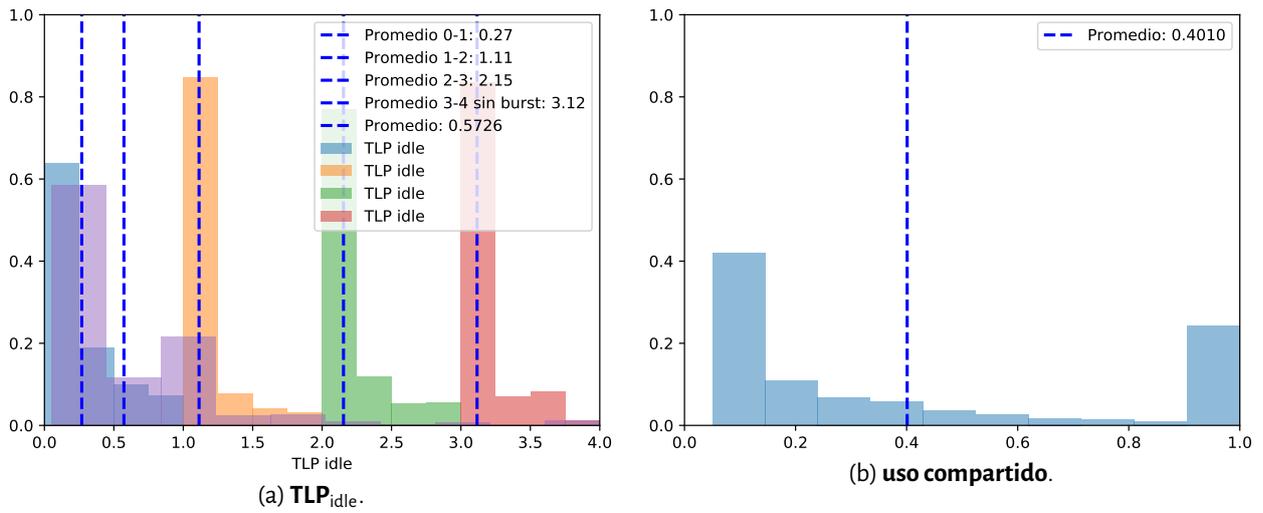


Figura 3.1: Uso del procesador.

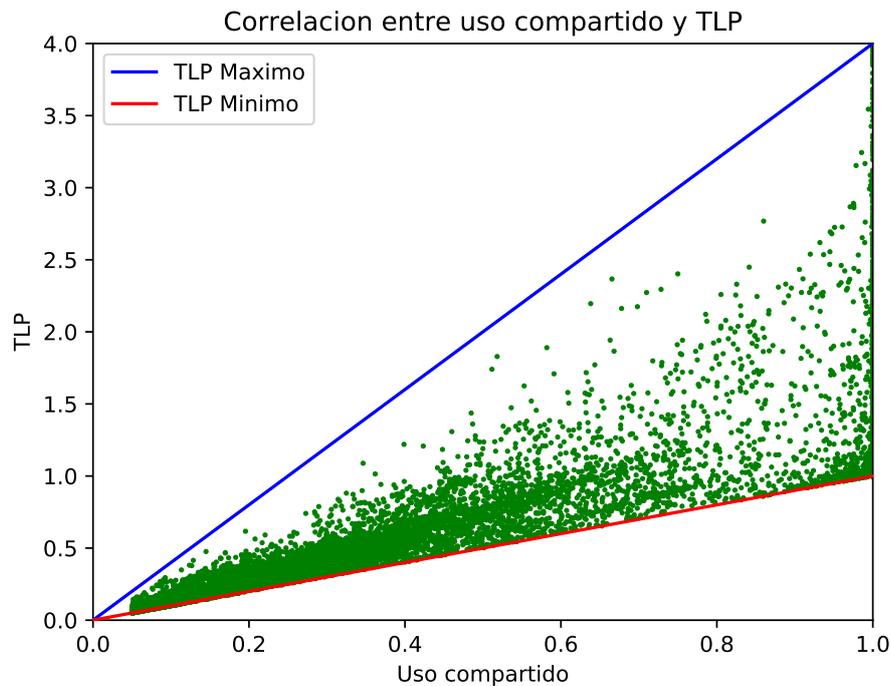


Figura 3.2: Relación entre el **uso compartido** y el **TLP**

La figura 3.2, por otro lado, pone de manifiesto la correlación entre **uso compartido** y TLP_{idle} . Podemos ver como a medida que aumenta el **uso compartido**, comienza a aumentar el TLP_{idle} . Por lo tanto, las muestras comienzan a tener un uso del procesador mucho más paralelo. Las rectas inferior y superior representan cuál es el mínimo y máximo valor de TLP_{idle} que se puede obtener para una muestra y un valor de **uso compartido**. Además, como nombramos anteriormente, el TLP_{idle} de una muestra está acotado superiormente por la cantidad de **cores** de las máquinas.

Notar que para un mismo número de **uso compartido** pueden corresponder a más de un TLP_{idle} . Por ejemplo, para un **uso compartido** de 0,5 puede deberse a que en la muestra el 50 % del tiempo se está ejecutando con un **core**, o que el 50 % del tiempo se corrió con dos **cores**. Para estos casos, el TLP_{idle}

es 0,5 y 1 respectivamente. Podemos observar como en todos los casos, la mayor densidad de puntos se encuentran más cercano al valor del **uso compartido**. Es decir, para cualquier valor de **uso compartido** dado, sus posibles valores de TLP_{idle} se encuentran más próximo al **uso compartido**, que a su cota superior. Esto implica que en su gran mayoría, las muestras tienen un paralelismo muy bajo. A medida que el **uso compartido** crece, los valores de los TLP_{idle} comienzan a dispersarse más.

En las siguientes secciones serán explicadas dos características particulares de las muestras. Por un lado, se realizará la partición de las mismas por rango horario en el que fueron tomadas. Esto nos permitirá saber si en algún momento del día las máquinas son más utilizadas que en otro momento. También, se estudiará un conjunto de muestras que fueron detectadas con un uso excesivo del procesador.

3.1 Análisis de muestras por rango horario

En la figura 3.3 se muestran algunos resultados sobre TLP_{idle} estudiados sobre las muestras, teniendo en cuenta el rango horario en la que fueron tomadas.

Las mismas, han sido divididas en **mañana**, **tarde**, y **noche** dependiendo de la hora la toma.

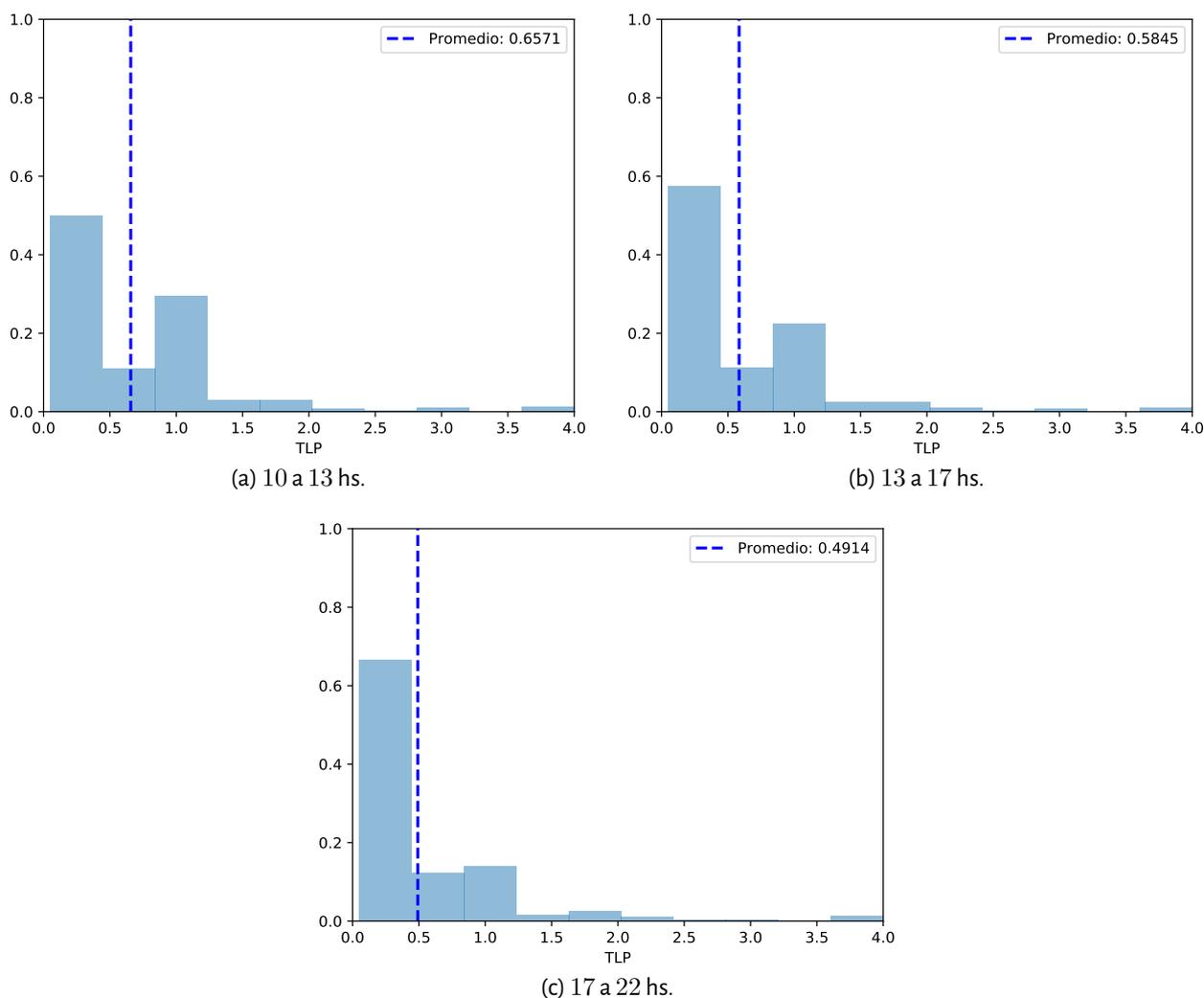


Figura 3.3: Uso del procesador por rango horario

Podemos observar como tienen un compartamiento similar en el caso de las muestras del rango horario **mañana**, 3.3a, y **tarde**, 3.3b, mientras que en el rango horario **noche**, 3.3c, las muestras poseen un tiempo *idle* mayor. Esto repercute en que su TLP_{idle} sea apenas menor.

3.2 Análisis de muestras *burst*

Llamaremos muestras **burst** a aquellas en dónde el TLP_{idle} es máximo. En nuestro caso sucede cuando es igual a 4, debido a que es la cantidad de *cores* con la que trabajamos. Este caso puede darse debido un rango de posibilidades. En uno de los extremos de éste, se encuentra la posibilidad de que haya una gran cantidad de programas utilizando el procesador de manera serial.

Tabla 3.1: Cuadro de **TLP** de los programas en muestras **burst**

Id	Cant. PS distintos	TLP Promedio	TLP Mínimo	TLP Máximo
1	37	1,00	1,00	1,12
2	37	1,01	1,00	1,16
3	35	1,02	1,00	1,19
4	33	1,01	1,00	1,27
5	31	1,01	1,00	1,16
6	31	1,02	1,00	1,21
7	31	1,01	1,00	1,23
8	30	1,01	1,00	1,09
9	30	1,02	1,00	1,66
10	30	1,09	1,00	2,76
11	29	1,00	1,00	1,05
12	29	1,03	1,00	1,48
13	29	1,02	1,00	1,65
14	27	1,04	1,00	1,62
15	27	1,03	1,00	1,86
16	26	1,02	1,00	1,27
17	25	1,01	1,00	1,15
18	24	1,01	1,00	1,08
19	24	1,03	1,00	1,63
20	23	1,00	1,00	1,05
21	23	1,01	1,00	1,27
22	23	1,02	1,00	1,36
23	22	1,01	1,00	1,12
24	22	1,02	1,00	1,19
25	22	1,02	1,00	1,41
26	21	1,01	1,00	1,18
27	21	1,02	1,00	1,18
28	18	1,03	1,00	1,32
29	17	1,06	1,00	1,98
30	8	1,35	1,00	3,77

Tabla 3.2: Cantidad de programas distintos, **TLP** mínimo, máximo y promedio de las muestras *burst*. Nos referimos a PS como programas.

Por el otro lado, puede ocurrir que haya una cantidad mucho menor de programas, pero que uno de ellos ejecute una gran cantidad del tiempo de la muestra con un $TLP_{idle} = 4$ o cercano al mismo.

El porcentaje de muestras **burst** es menos del 1% de las muestras totales. En total son 30 muestras sobre las 20524. Al contar con pocas muestras **burst**, es posible listarlas exhaustivamente. El cuadro 3.1 enumera todas muestras **burst** encontradas.

El cuadro muestra el **TLP** promedio de entre todos los programas que son ejecutados en cada una de las muestras, como a su vez, el máximo y el mínimo de los mismos. También, presenta la cantidad de programas distintos que se encontraron consumiendo el procesador durante la muestra.

Las muestras fueron ordenadas en orden descendiente por la cantidad de programas distintos que fueron ejecutados; y luego por **TLP** máximo de manera ascendente.

Las columnas de **TLP** máximo y mínimo nos muestran cuál fue el programa con mayor y menor **TLP** de la muestra respectivamente.

En particular, se observa que el **TLP** mínimo, siempre es igual a 1 en todas las muestras. Esto quiere decir que siempre existió al menos un programa que ejecutó en un core.

El **TLP** promedio no varía demasiado para todas las muestras con excepción de la muestra número 30. Por otro lado, la cantidad de programas corriendo entre estas muestras varía notablemente. Si bien, los **TLPs** máximos presentan mayor variabilidad, no aparece un **TLP** que sea superior a 2, a excepción de la muestra en fila 10.

Esto da la noción que la muestra alcanza un uso total de sus cores, debido a la gran cantidad de programas que se encuentran corriendo con 1 o 2 cores, y no debido a que los programas estén corriendo de manera paralela.

En la tabla dos muestras son indicadas en color como casos de estudio. El primer caso, la fila 10, se observa que existió un programa con un **TLP** de 2,76, que es significativamente superior al promedio de la misma muestra. Éste último reporta un **TLP** promedio de 1,09.

Además, hay 30 programas distintos corriendo en la muestra, y debido a su bajo promedio de **TLP**, podemos deducir que prácticamente todos los programas corrieron con un **TLP** muy bajo; cercano a 1.

Por último, tenemos marcada la fila 30 como otro caso de estudio aparte. Nuevamente nos encontramos con un caso parecido, en el que existe un programa con un **TLP** muy alto, esta vez de 3,77. A diferencia del caso anterior, tenemos un **TLP** promedio mayor y una cantidad de programas en ejecución considerablemente menor. Aún así, no llega a acercarse al **TLP** máximo del programa de la muestra. Por lo tanto, la distribución de cargas del procesador entre los programas, no es equitativa.

El cuadro 3.3 muestra el **TLP** de cada uno de los programas encontrados en la muestra 30.

Tabla 3.3: Cuadro de **TLP** de cada uno de los programas de la muestra 30

Id	Programa	TLP
1	camino v9	3,77
2	rcu sched	1
3	ksoftirqd/3	1
4	mpirun	1
5	ntpd	1
6	gmain	1
7	salt-minion	1
8	avahi-daemon	1

En el caso de caminos v9 muy probablemente sea de un trabajo práctico de algún alumno, que utiliza mucha concurrencia. Además, se encontró que este proceso fue ejecutado el 100 % del total de la traza. El 1,5 % lo hizo con un core, el 5,9 % con dos cores, el 6,01 % con tres cores y el restante 86,3 % con cuatro cores.

Se destaca la diferencia entre el promedio de los valores de **TLP** de los programas y el de las muestras. Recordemos que al estar hablando de muestras **burst**, todas éstas tienen un **TLP_{idle}** igual a 4. A pesar de eso, el promedio de **TLP** de sus programas es muy inferior al de la muestra en sí.

Con esta información podemos concluir que lo más probable es que una muestra sea **burst** debido a la gran cantidad de programas corriendo con 1 core, que a la utilización paralela del procesador por parte de los programas.

VALIDACIÓN DE LA INFORMACIÓN

4.1 Introducción

Nuestra metodología de trabajo consiste en tomar muestras de 10 segundos de duración cada 5 minutos sobre las máquinas de los laboratorios, que se encuentran siendo utilizadas de manera aleatoria por los usuarios. Luego a partir de las mismas serán calculadas las magnitudes ya presentadas.

En esta sección, se buscará validar el comportamiento del **TLP** en las muestras tomadas. Para eso se realizarán experimentos en el mismo ambiente que se realizó la toma de las muestras, pero de forma controlada. Serán llevados a cabo en momentos en los que los laboratorios no se encuentren abiertos al público, como por ejemplo los fines de semana. Para los mismos se reemplaza el uso real de los usuarios por programas controlados.

La carga sobre los procesadores será utilizando instancias del programa π . π es un programa totalmente serial que se encarga de calcular el número π con una cantidad de dígitos arbitraria. Las cuentas que realiza π son numéricas y sin necesidad de realizar actividades de *entrada salida*; con un consumo de memoria mínimo.

A partir de estas muestras, se obtendrá el **TLP**. Conociendo la carga del procesador asignada, buscaremos constatar que el **TLP** promedio de todas las muestras es similar al resultado que esperamos.

Sobre esta idea se realizarán dos experimentos:

- Modificando la cantidad de instancias de π en ejecución.
- Modificando la cantidad de instancias de π en ejecución, en función del tiempo.

Ambos experimentos son realizados sobre un subconjunto de máquinas de uno de los laboratorios. Se tomarán muestras cada 1 minuto para cada experimento. Consideramos que el SO es justo en la asignación de cores. Esto quiere decir que si hay dos instancias de π siendo ejecutadas, las mismas serán asignadas a dos *cores* distintos.

4.2 Experimento 1

El experimento constará de cuatro mediciones. En cada una de estas se ejecutarán desde una hasta cuatro instancias de π en las máquinas. Cada medición tendrá una duración de dos horas.

Conociendo la carga del procesador asignada en cada medición, para que la validación en este caso sea correcta, necesitamos que el promedio del **TLP** de las muestras tomadas en cada uno de las cuatro mediciones, sea similar a la cantidad de instancias de π que se encuentran siendo ejecutadas en la misma.

El **TLP** es medido para todas las aplicaciones y no solamente para el programa π . Además de las instancias de π , se encuentran siendo ejecutados procesos propios del SO. Esto implica que el promedio de **TLP** de todas las muestras, es mayor a la cantidad de instancias de π ejecutadas.

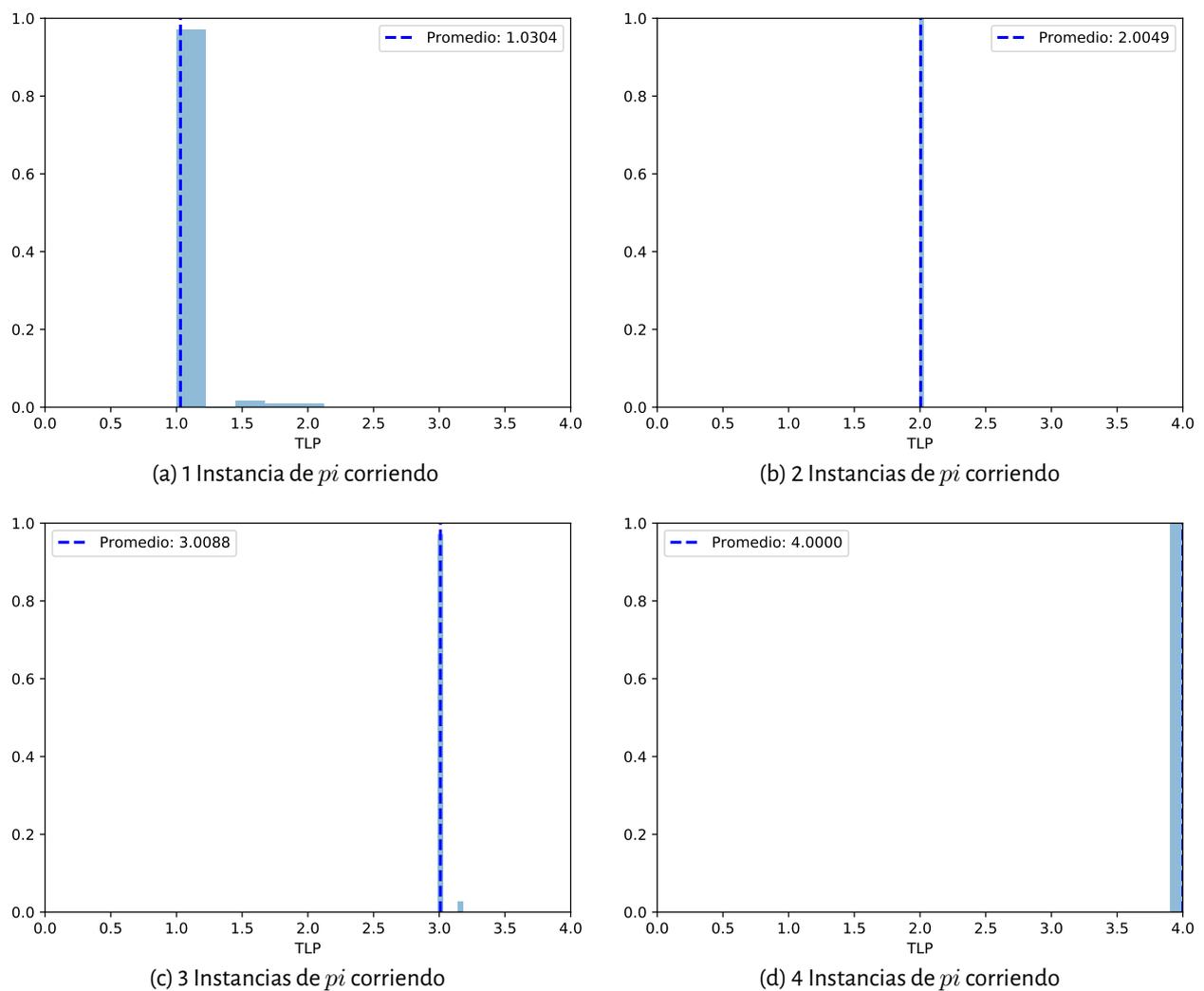
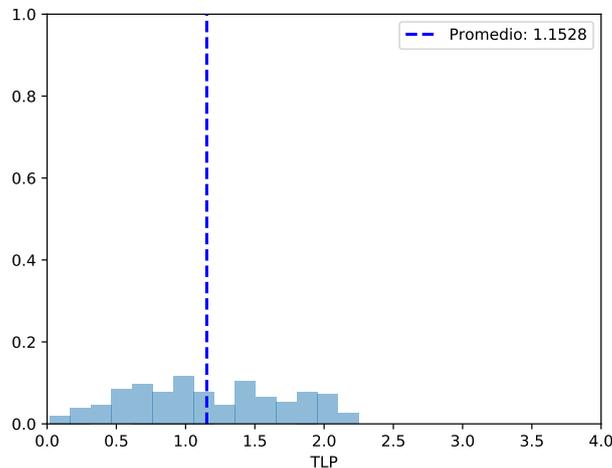


Figura 4.1: Validación de las muestras

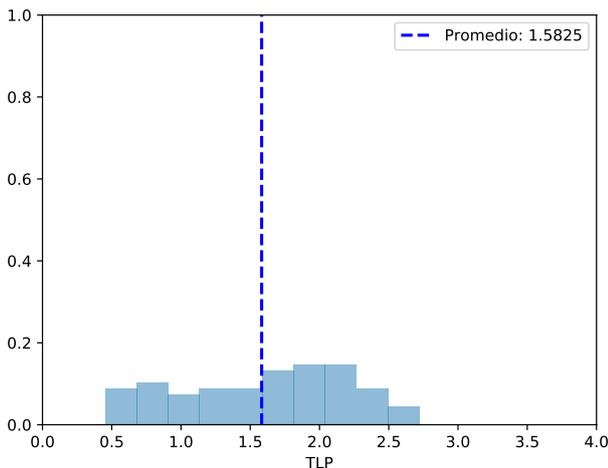
La figura 4.1 presenta las 4 distribuciones de los **TLP** en cada una de las mediciones. Las cuatro figuras muestran el promedio de **TLP** que se obtuvo para cada una de las mismas. En todos los casos, el **TLP** obtenido se corresponde con el valor esperado.

4.3 Experimento 2

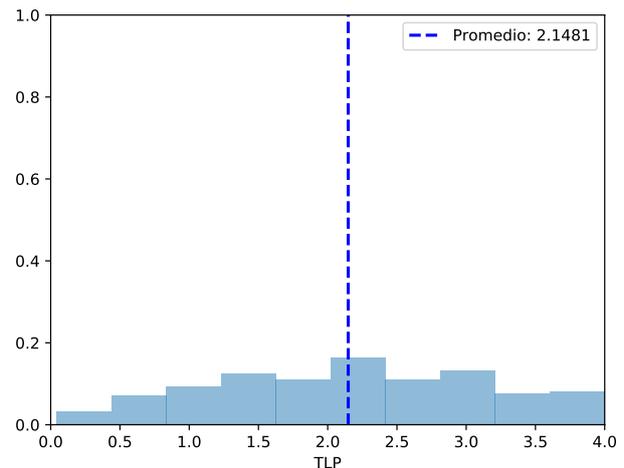
En este experimento modificaremos la carga del procesador en función del tiempo. A diferencia del experimento anterior en el que las instancias de π quedaban fijas en el tiempo, en este caso irán variando de acuerdo a intervalos fijos en el tiempo.



(a) 2 Instancias de π corriendo en tiempos de 1 segundo



(b) 3 Instancias de π corriendo en tiempos de 1 segundo



(c) 4 Instancias de π corriendo en tiempos de 1 segundo

Figura 4.2: Validación de las muestras en tiempo fijo

Se realizarán 3 mediciones en las cuáles se lanzarán de dos a cuatro instancias de π durante un tiempo fijo s . Luego de ese tiempo s se ejecutará un comando para eliminar a todas las instancias de π . A partir de esto, se esperará otro tiempo s sin instancias de π en ejecución, para luego volver a ejecutar las instancias de π que correspondan según la medición y comenzar nuevamente el ciclo.

Es decir, si este tiempo es de s segundos, y la cantidad de instancias de π ejecutadas en la medición es de n , la secuencia será :

s segundos con n instancias de π
 s segundos sin instancias de π
 s segundos con n instancias de π
 s segundos sin instancias de π

...

Para que la validación sea correcta, debemos obtener en cada medición un **TLP** que sea aproximadamente la mitad de las instancias fijas de π ejecutando en cada caso. Esto es debido a que la mitad del tiempo están siendo utilizados esa cantidad de *cores*, y la otra mitad del tiempo se está ejecutando sin un número de instancias de π fijas.

En la figura 4.2 se muestran los gráficos de las distintas mediciones, habiéndose utilizado un tiempo fijo s de 1 segundo.

Como podemos ver en las figuras 4.2a, 4.2b y 4.2c, se muestra una distribución de **TLP** más uniforme que en el primer experimento. Esto es atribuido a que al correr las instancias de π durante un tiempo limitado, en varios casos sucede que el *scheduler* no logra dejar a π corriendo durante todo ese segundo. Además puede ocurrir que durante la ejecución, el *scheduler* quite a π de alguno de los *cores* antes de que termine de ejecutar.

A pesar de esto, en el caso del promedio, el **TLP** es similar a lo que esperábamos, dando así una validación correcta.

RESULTADOS

En este capítulo presentaremos los experimentos realizados sobre el conjunto de muestras *útiles* que han sido tomadas y analizadas de los laboratorios.

Los mismos son:

1. **TLP sobre el total de las muestras.**
2. **TLP sobre un conjunto de programas.**
3. **Análisis de actividades de *entrada salida*.**
4. **Análisis de tareas del SO.**
5. **Transiciones del procesador.**

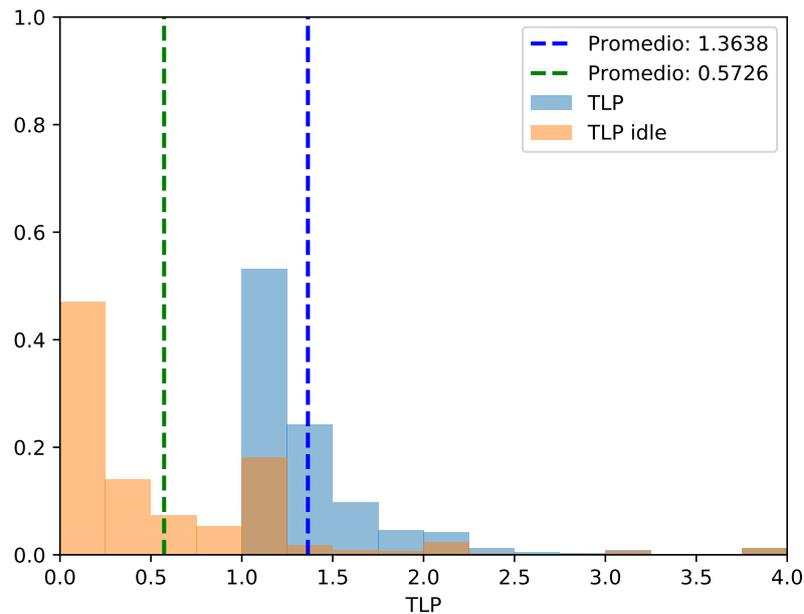
5.1 TLP sobre el total de las muestras

En este experimento se usará **TLP**, que ya fue explicada en detalle en la sección de metodología 2. La misma será utilizada para poder medir la concurrencia de las muestras. Esta medida fue introducida para el análisis del uso del procesador por Flautner et al. y luego utilizada también en Blake et al.

En el trabajo de Flautner et al., se introduce la medida para analizar la concurrencia de los programas. En esta sección será utilizada para estudiar el uso concurrente del procesador en las muestras.

Se analizará el **TLP** desde dos puntos de vista distintos. Considerando el tiempo *idle* (TLP_{idle}) y sin considerarlo para el total de las muestras (**TLP**). Notar que esta distinción nos permitirá estudiar cómo es el uso del procesador sobre la muestra en su totalidad, al mismo tiempo que nos permitirá hacerlo solamente sobre el tiempo en el que la muestra se encuentra en estado *activo*. Para el cálculo del **TLP** sin considerar el tiempo *idle*, notar que siempre va a estar acotado inferiormente por 1.

A continuación se presentan las distribuciones de los dos análisis del **TLP**.

Figura 5.1: Distribución **TLP** *idle* vs no *idle*

En la figura 5.1 superior podemos observar el promedio de ambos **TLP**. En el caso de la distribución del **TLP** sin incluir el tiempo *idle*, presenta un valor promedio bajo además de una gran concentración de sus valores, muy cercanos a 1. Esto evidencia el muy bajo uso promedio del procesador en todas las muestras.

Por otro lado, en la distribución del **TLP** incluyendo el tiempo *idle*, también obtenemos un promedio muy bajo. Este valor nos indica que a nivel general, el tiempo en el que se encuentra ocioso el procesador, es muy grande.

Nos parece interesante para el análisis, realizar una partición de las muestras dependiendo el **TLP** en distintos rangos. En la siguiente subsección, se detallará el estudio y la partición realizada.

5.1.1 Categorización de muestras por TLP

Para un estudio más detallado, serán categorizadas las muestras según su uso del procesador en general, es decir por su **TLP** considerando el tiempo *idle*. Particionaremos a las mismas en los siguientes rangos de valores de \mathbf{TLP}_{idle} : $\{[0 - 1), [1 - 2), [2 - 3), [3 - 4]\}$. Esta partición es propuesta ya que cada vez que se es asignado una cantidad de cores en una muestra, luego esta cantidad de cores se mantiene por un período de tiempo.

Las figura 5.2 expone todas las distribuciones y el \mathbf{TLP}_{idle} promedio de cada uno de los rangos mencionados.

La figura observa la partición en los distintos rangos de valores del \mathbf{TLP}_{idle} del total de las muestras. En la parte superior de los gráficos se puede observar cada promedio correspondiente a la distribución. Esta partición no es uniforme, ya que como podemos observar en la tabla 5.1, hay distintas cantidades de muestras en cada uno de estos rangos.

En los casos $\{[0 - 1), [1 - 2), [2 - 3)\}$ los promedios correspondientes a cada partición, se encuentran cercanos a su mínimo. En el caso $[3 - 4]$ encontramos un promedio no tan cercano a su mínimo valor de \mathbf{TLP}_{idle} como en los casos anteriores. Esto es debido a, como se explicó en la sección 3.2, la existencia de muestras que saturan al procesador. Al pertenecer las mismas a este rango de valores de \mathbf{TLP}_{idle} , se

obtiene un promedio mayor que en los casos anteriores. Para constatar que este comportamiento es debido a las muestras **burst**, en la figura 5.3 se vuelve a graficar la partición [3 – 4] sin este tipo de muestras.

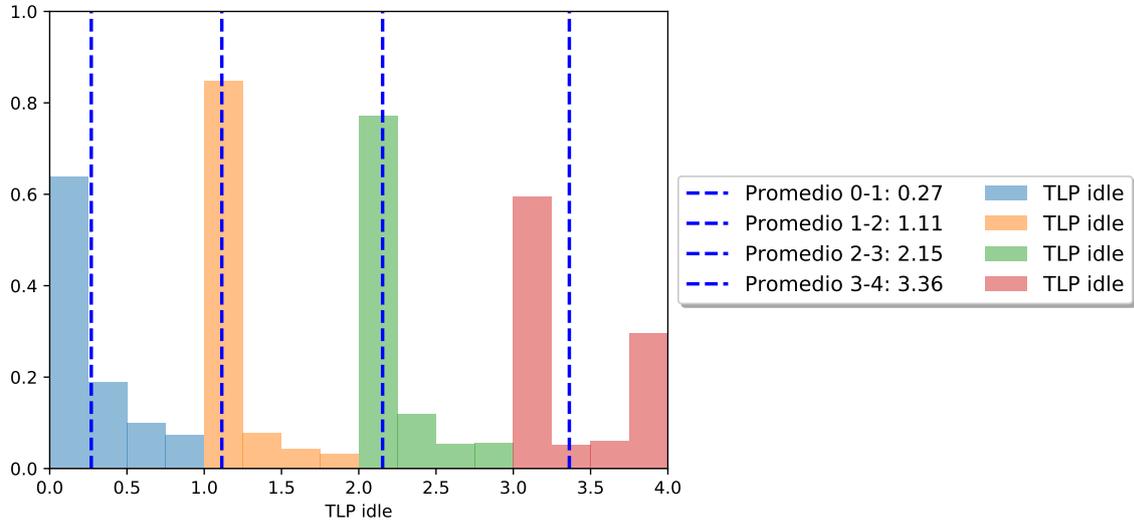


Figura 5.2: Distribución de **TLP_{idle}** por rango de valores de [0 – 1), [1 – 2), [2 – 3) y [3 – 4]

Rango	Valor
[0 – 1)	73,63
[1 – 2)	21,40
[2 – 3)	3,05
[3 – 4]	1,95

Tabla 5.1: Porcentaje de cantidades de muestras por rango de valores del **TLP_{idle}**

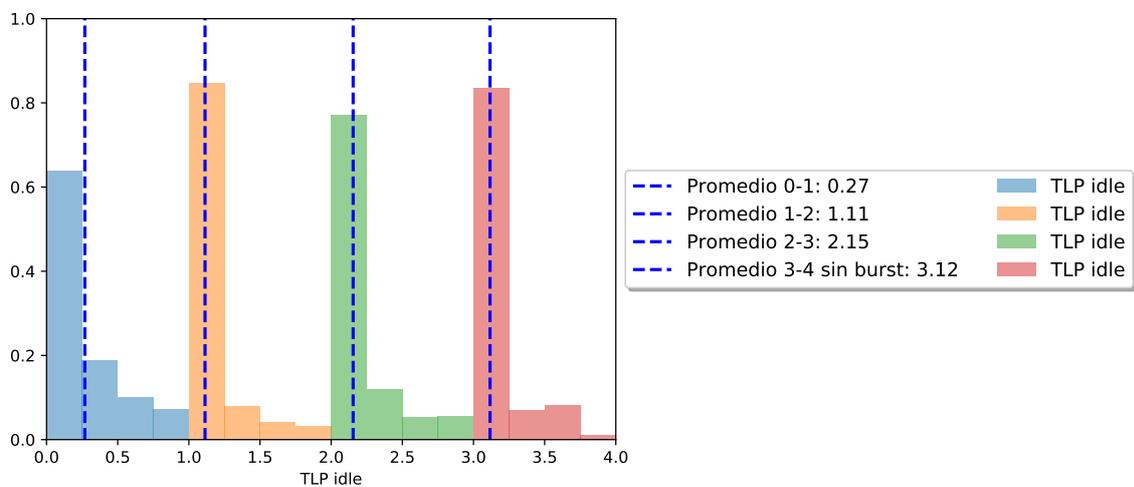


Figura 5.3: Rango de valores de **TLP_{idle}** [3 – 4] sin muestras **burst**

5.2 TLP sobre conjunto de programas

A diferencia de Blake et al. y Flautner et al, no es posible para la metodología de este trabajo utilizar programas específicos, sino que son capturados los programas utilizados en los laboratorios. Si estos resultan muy usados por los usuarios, entonces obtendremos una gran cantidad de muestras con información sobre los mismos. En caso contrario se obtendrían pocas muestras de esos programas.

Los laboratorios donde son tomadas las muestras del trabajo, como mencionamos en capítulos anteriores, pertenecen al Departamento de Computación de la FCEN. Las máquinas de éstos, son usadas mayoritariamente por alumnos de la carrera de Computación. De esta manera se esperaría que en el muestreo se encontrase una considerable cantidad de información sobre programas como compiladores de código, por ejemplo gcc. No obstante, no se han hallado en ninguna muestra. Creemos que esto se debe a dos motivos. El primero, como mencionamos anteriormente, en el momento de toma de la muestra (10 segundos) es difícil conseguir un uso elevado del procesador. Además sumado a que los procesos de compilación suelen ser muy rápidos, entonces la probabilidad de tomar una muestra en ese momento, es muy baja.

Por el contrario, se ha encontrado mucha información sobre uso de programas de categorías como: Browsers, editores de texto, programas de estadística y matemática. También programas como evince o Telegram han presentado un uso muy frecuente por los usuarios. Por lo tanto en el estudio, los categorizaremos dentro de Otros.

En la tabla 5.2 se presentan los **TLP** promedio para cada uno de los programas más usados, segmentado por categoría. En la misma tabla se obtendrá el uso discriminado por porcentaje de uso de cada cantidad de cores.

Categoría	Programa	idle	c1	c2	c3	c4	TLP
Browsers	chrome	51,73	27,07	14,01	6,14	1,04	1,6094
	firefox	46,46	36,13	12,75	4,29	0,38	1,4194
Matemática	rstudio	32,54	41,02	23,64	2,63	0,17	1,4360
	MATLAB	49,57	33,18	13,21	3,81	0,24	1,4271
	octave	63,08	36,40	0,52	0,00	0,00	1,0141
Editores de texto	atom	85,76	11,61	2,60	0,02	0,00	1,1861
	sublime_text	62,97	36,35	0,67	0,01	0,00	1,0188
	geany	86,56	13,37	0,07	0,00	0,00	1,0053
	gedit	91,34	8,66	0,00	0,00	0,00	1,0003
Otros	evince	93,22	6,67	0,11	0,00	0,00	1,0178
	Telegram	57,68	42,21	0,11	0,00	0,00	1,0026

Tabla 5.2: Trabajo actual- 4 cores. Cada valor de las columnas C_i están representadas en porcentaje

Se observa como los programas que mejor **TLP** consiguieron, han sido los browsers, seguidos por los programas de matemáticas, a excepción del Octave que obtuvo un **TLP** muy bajo. Dentro de la categoría de los programas de edición de texto, a excepción del Atom, el promedio de sus **TLP** tiende hacia 1. En el caso del Atom presenta un **TLP** mayor, de 1,18. Además, programas como evince y Telegram obtienen un **TLP** muy bajo, siendo así programas prácticamente seriales.

A continuación se ilustran los gráficos de las distribuciones de los **TLP** de los programas estudiados. Al igual que en el caso de las muestras generales, son analizadas las distribuciones del **TLP** incluyendo y sin incluir al tiempo *idle*. Recordemos que el hecho de no considerar el tiempo *idle*, lo que nos permite pensar el uso del procesador de cada programa en el tiempo en el que estuvo activo en la muestra (con al menos un core ejecutando la aplicación).

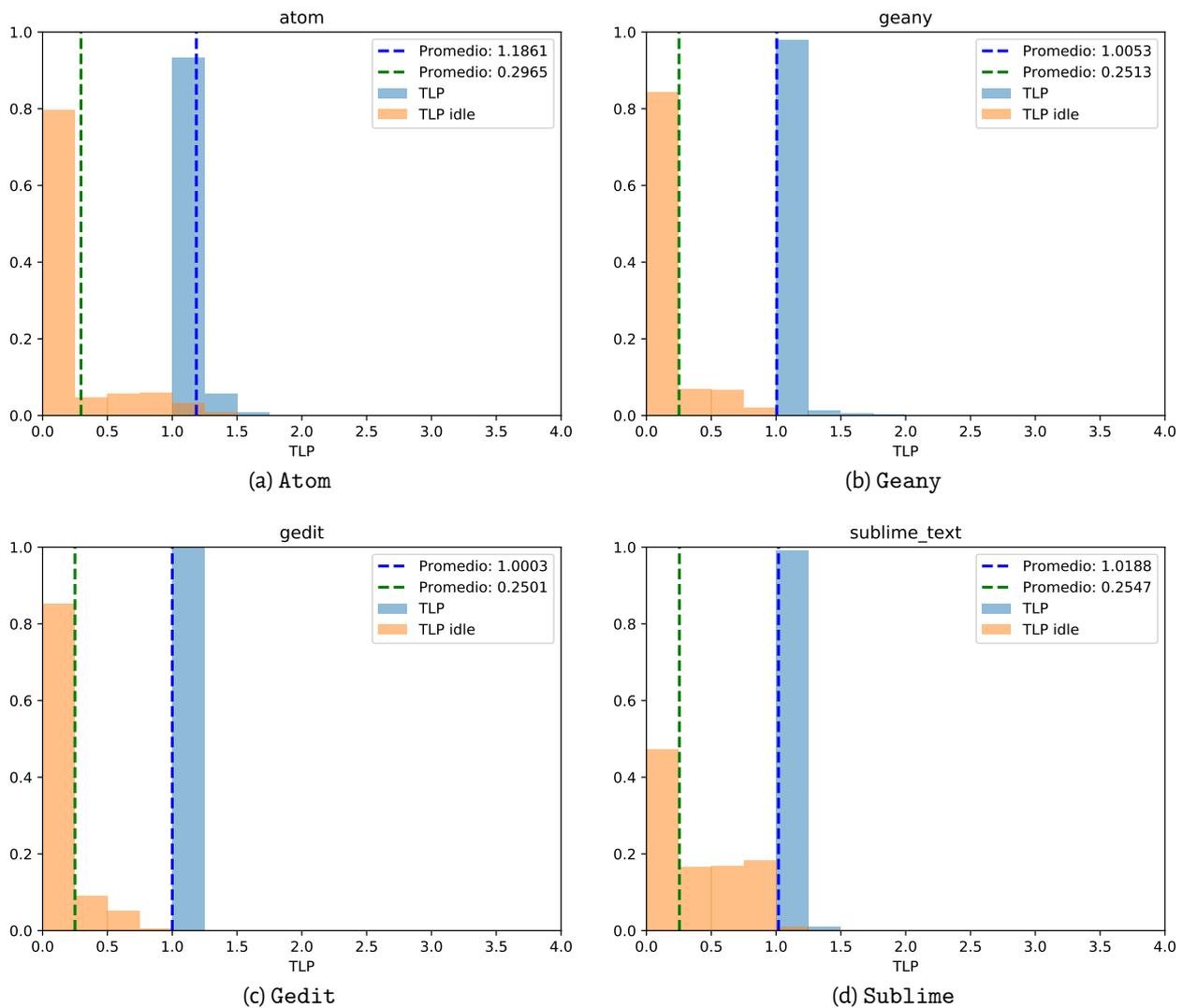


Figura 5.4: Editores de texto

De los gráficos en las figuras 5.4, se observa que los programas de la categoría de editores de texto, tienen un comportamiento claramente serial.

Todos los programas presentan un promedio de **TLP** considerando el tiempo *idle* muy bajo. Esto nos indica que gran parte del tiempo se encuentran en estado *Listo* pero sin ser ejecutados.

En cuanto al **TLP** sin considerar el tiempo *idle*, *atom* muestra un mejor **TLP** que los demás. Sin embargo, como veremos a continuación, dista mucho del promedio de **TLP** mostrado por los browsers.

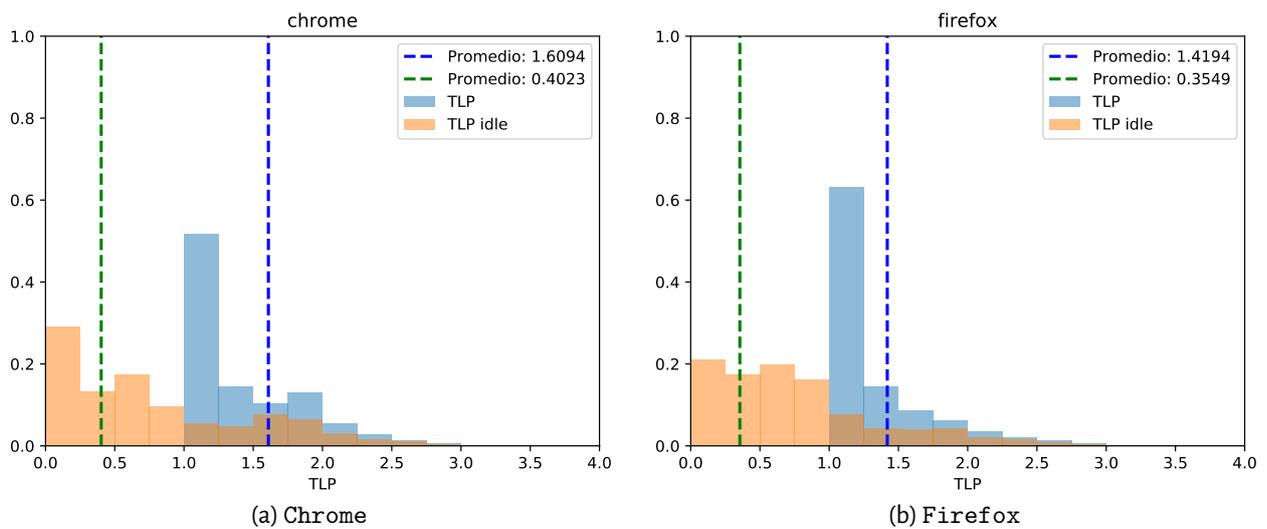


Figura 5.5: Browsers

En la figura 5.5 se observa que los browsers, tanto el Chrome como el Firefox, tienen un **TLP** considerablemente mayor a 1. Se pudo observar en el análisis de las muestras, que este tipo de programas lanzan muchos *threads* que sirven de *helpers* para realizar sus funciones. Por este motivo es que poseen un mejor **TLP** que el resto de las aplicaciones.

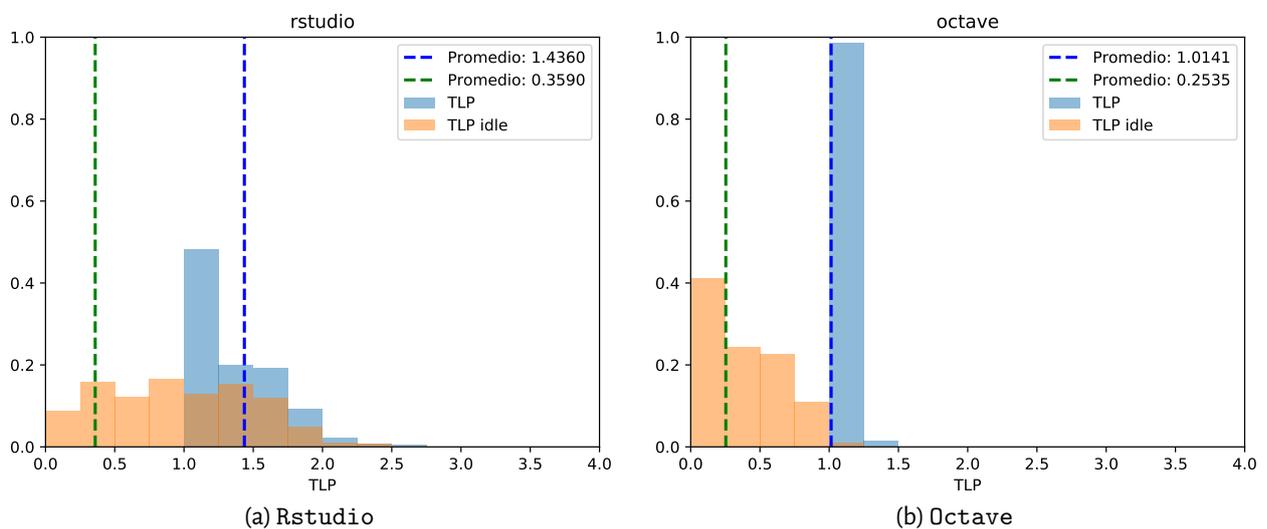


Figura 5.6: Matemática

En la figura 5.6 se muestran las distribuciones de los programas de matemática. Podemos observar una gran diferencia entre el R y el Octave. Octave muestra un comportamiento serial, mientras que el R un **TLP** de 1,4. Esto es debido a la máquina virtual sobre la que corre esta aplicación.

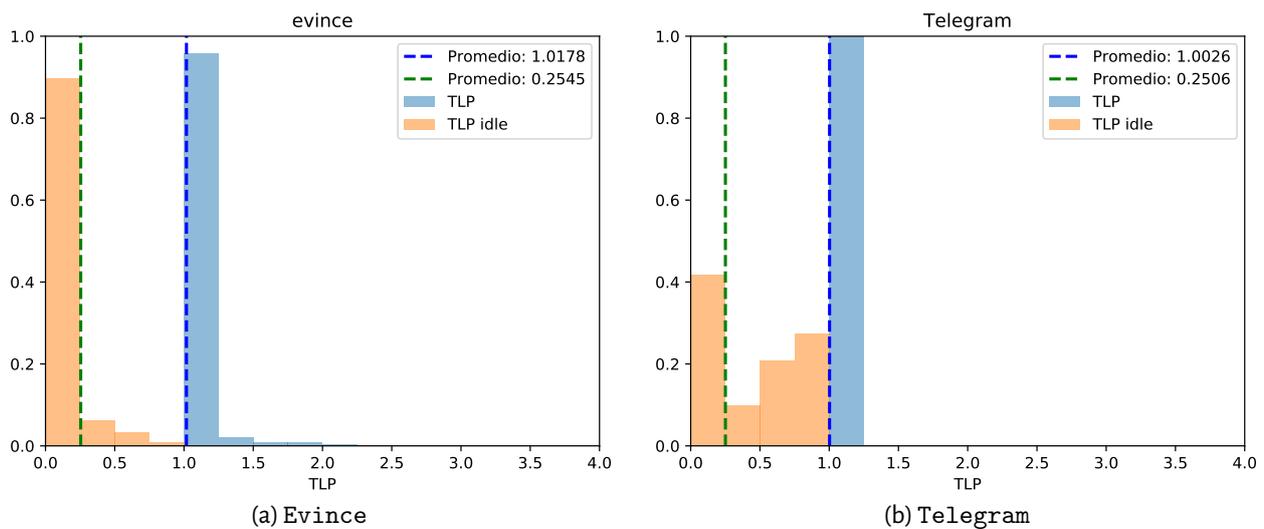


Figura 5.7: Otros programas

Por último en la figura 5.7 se observan las distribuciones del valor del **TLP** para el **evince** y **telegram**, pertenecientes a la categoría de Otros programas. Ambos muestran comportamientos seriales.

5.2.1 Comparación de programas con publicaciones anteriores

En la presente sección buscaremos comparar los resultados obtenidos por Blake et al. y Flautner et al. para un conjunto de programas similares a los utilizados en sus trabajos. La comparación será realizada desde el **TLP** sin incluir al tiempo *idle*.

Nos referimos a *programas similares* como aquellos programas que pertenecen a la misma categoría. Esto es debido a que al trabajar sobre muestras de laboratorios de uso real, las mismas son tomadas sobre los programas que los usuarios estén ejecutando. No pudiendo generar muestras fuera de estos programas. Por este motivo es que no todos los programas que son estudiados en los trabajos anteriores aparecen en las muestras obtenidas. En caso de aparecer, lo hacen en pocas de éstas, por lo que sirve para poder realizar un análisis robusto. Los casos de programas de los que no hemos podido encontrar suficiente información para una comparación de resultados, pero que fueron estudiados en los trabajos de referencia, han sido programas de música, edición de videos, o juegos.

Las categorías que se compararán son:

- Editores de texto o programas de oficina.
- Browsers.

A continuación se presenta una tabla con información sobre la categoría de editores de texto o programas de oficina. En la misma aparecen los programas utilizados por Flautner, Blake y, por último, los de la presente tesis. Se muestra **TLP** promedio de cada programa para el total de las muestras, y discriminación por cantidad de cores. Los trabajos de Flautner y Blake fueron realizados sobre máquinas con 2 y 16 cores respectivamente. En nuestro caso, se realizó sobre procesadores con 4 cores. Notar que en el caso del trabajo de Flautner no fue expuesta esta discriminación por core.

Autor	Programa	Idle	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	TLP
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Flautner 2000	FrameM	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1,35
	Xemacs	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1,26
Blake 2010	Excel	72	23	4	0	0	0	0	0	0	0	0	0	0	0	0	0	1,2
	Word	74	22	4	0	0	0	0	0	0	0	0	0	0	0	0	0	1,2
Tesis	Atom	85	11	2	0	0	-	-	-	-	-	-	-	-	-	-	-	1,18
	Sublime	63	36	0	0	0	-	-	-	-	-	-	-	-	-	-	-	1,01
	Geany	85	13	0	0	0	-	-	-	-	-	-	-	-	-	-	-	1,005
	Gedit	91	8	0	0	0	-	-	-	-	-	-	-	-	-	-	-	1,002

Tabla 5.3: Comparación de **TLP** de los programas de edición de texto y oficina. Cada valor de las columnas C_i están representadas en porcentaje

Se observa de la tabla 5.3 que los programas con mejor **TLP** son los estudiados en el trabajo de Flautner en el 2000. En el caso de los programas de Blake en el 2010, tanto Word como Excel tienen el mismo **TLP**, siendo similar al de Xemacs. Luego, en el caso de los programas del actual trabajo, a excepción de Atom, se presenta un uso serial del procesador. En el caso del programa de edición de texto Atom, se observa un **TLP** apenas mayor que los restantes programas del 2017. Por lo tanto, a pesar del paso de los años, no se ve una mejoría en cuánto al **TLP** por parte de este tipo de aplicaciones.

A continuación se comparará los programas de la categoría de Browsers.

Autor	Programa	Idle	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	TLP
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Flautner 2000	Netscape	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1,3
Blake 2010	Safari	50	34	11	3	1	0	0	0	1	0	0	0	0	0	0	0	1,6
	Firefox	66	24	6	1	0	0	0	0	1	0	0	0	0	0	0	0	1,5
Tesis	Chrome	51	27	14	6	1	-	-	-	-	-	-	-	-	-	-	-	1,6
	Firefox	46	36	12	4	0	-	-	-	-	-	-	-	-	-	-	-	1,4

Tabla 5.4: Comparación de **TLP** de los browsers. Cada valor de las columnas C_i están representadas en porcentaje

Se observa de la tabla 5.4 que, a diferencia de los editores de texto, se puede observar una mejoría del **TLP** entre los años 2000 y 2010. En el 2000, Netscape presenta un **TLP** de 1,34 mientras que en el 2010, en Safari se observa un **TLP** de 1,6. Luego, entre los años 2010 y 2017, no se presentan cambios relevantes en cuanto al **TLP**.

Con respecto a Firefox, vemos que también fue estudiado por Blake en el 2010. En ese año presenta un mejor **TLP** que en la actualidad, pero vemos una leve mejor distribución de la carga del procesador en el año 2017. A pesar de esto, en el año 2000 se observa un mejor **TLP**, dado que el trabajo de Blake fue realizado en máquinas con 16 cores. En la discriminación por uso de core, el 1 % del promedio total se utilizó 8 cores simultáneamente, por lo que termina generando un **TLP** mayor que al del 2017.

5.3 Entrada salida

En el presente experimento analizaremos el comportamiento de las tareas con relación a las actividades de *entrada salida* y como influye el mismo sobre la concurrencia general del sistema.

Como mencionamos anteriormente en la sección 2, `perf` permite identificar cuándo un proceso comienza a realizar actividades de *entrada salida*, y cuando las mismas son concluidas. Recordemos que esto lo hacemos mediante el evento `iowait` y el evento `wakeup` respectivamente.

Enfocaremos este experimento desde dos ángulos:

1. Tiempo de duración de las actividades de *entrada salida*.
2. Tiempo de demora en entrar en actividades de *entrada salida*.

5.3.1 Tiempo de duración de las tareas de *entrada salida*

A continuación se muestra la distribución de los tiempos que un proceso se encuentra realizando actividades de *entrada salida*, es decir cuánto tiempo transcurre desde que lanzó un evento `iowait` hasta su posterior `wakeup`. Notar que, como las muestras son trazas en el tiempo, puede suceder que algún evento `iowait` no posea su `wakeup` correspondiente luego, ya que quedó por fuera de la muestra. En ese caso, no será tenido en cuenta para el experimento.

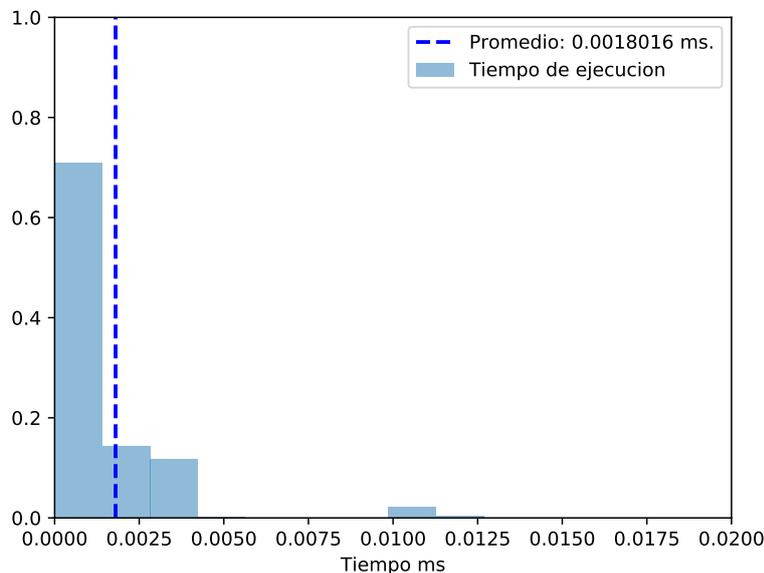


Figura 5.8: Distribución de tiempo de duración de las actividades de *entrada salida*.

Podemos observar en la figura 5.8 que, a excepción de algunos casos en los que las actividades de *entrada salida* de las muestras superan los 0,01 ms, la mayoría de ellas se encuentran entre los 0 ms a 0,001 ms de duración, teniendo un promedio de 0,0018 ms. Con este dato, podemos inferir que, en principio, no es un suceso que afecte el comportamiento concurrente del procesador. Calculando el **TLP** máximo que podrían obtener estas actividades, se llega a 0,004 promedio por muestra, indicando que las mismas no podrían alterar **TLP** global del sistema.

5.3.2 Tiempo de demora en entrar en actividades de *entrada salida*

En la figura 5.9 se ilustra la distribución de tiempos que tarda un proceso en comenzar a realizar actividades de *entrada salida*.

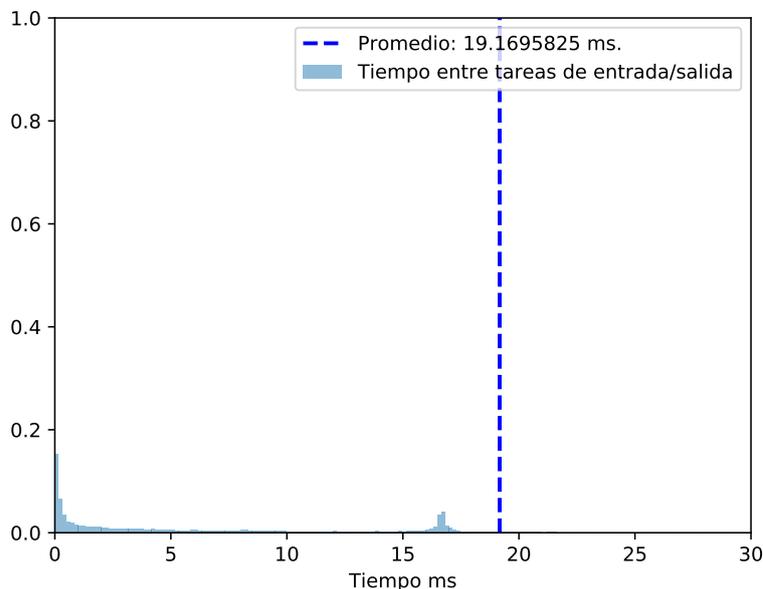


Figura 5.9: Distribución de tiempos cada cuanto un proceso comienza a realizar *entrada salida*.

Como se observa en el gráfico 5.9, las actividades de *entrada salida* tardan un tiempo promedio considerable en comenzar a ser ejecutadas en algunos de los *cores* del procesador. Esta situación sumada al dato anterior acerca de que su duración en el procesador es muy pequeña lleva a que sea esperable que este tipo de tareas no afectan al **TLP** del sistema.

5.4 Tareas del SO

En el presente experimento analizaremos el comportamiento de las tareas del SO y como influye el mismo sobre la concurrencia general del sistema. Es decir, queremos observar si realmente las tareas del SO realizan un trabajo muy fuerte sobre el procesador, pudiendo así afectar al **TLP**.

Como dijimos anteriormente en la sección 2, `perf` permite identificar a las tareas del SO. Las que comienzan el prefijo `worker` son las que pertenecen a este conjunto de tareas. A partir de ahora llamaremos a las tareas del SO como *workers* en forma genérica.

Enfocaremos este experimento desde tres estudios:

1. Tiempo de duración de las tareas del SO.
2. Tiempo en el que tarda el SO en poner a correr una de estas tareas.
3. Concurrencia.

5.4.1 Tiempo de duración de las tareas del SO

Se presenta la distribución de los tiempos que tardan en ser desalojadas las tareas del SO una vez que comienzan a utilizar el procesador.

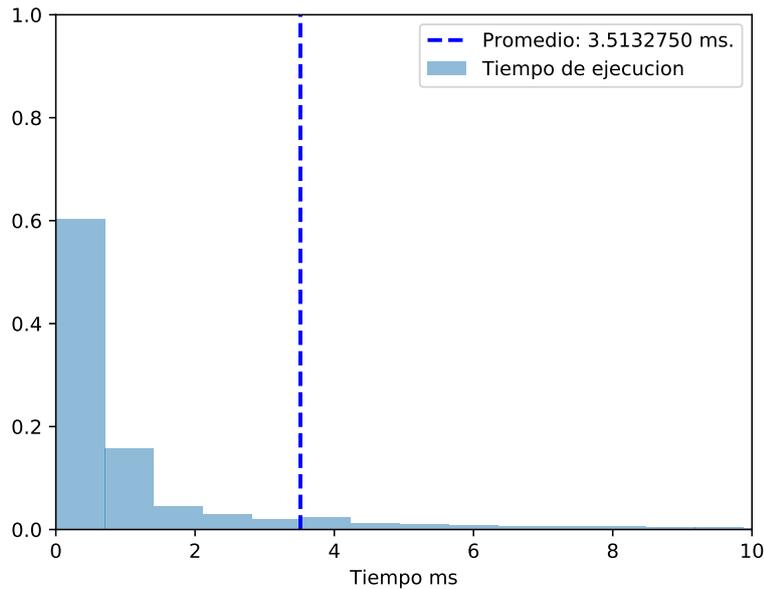


Figura 5.10: Distribución de tiempo de duración de las tareas del SO.

Podemos observar en la figura 5.10 que las tareas del SO tienen una duración bastante prolongada si se compara con las de *entrada salida*. Cuando son alojadas en alguno de los cores, tardan alrededor de 3,5 ms en ser luego desalojadas.

5.4.2 Tiempo en el que tarda el SO en poner a correr una de estas tareas

En el gráfico 5.11 se ilustra la distribución de los tiempos que tarda el SO en alojar un *kworker*.

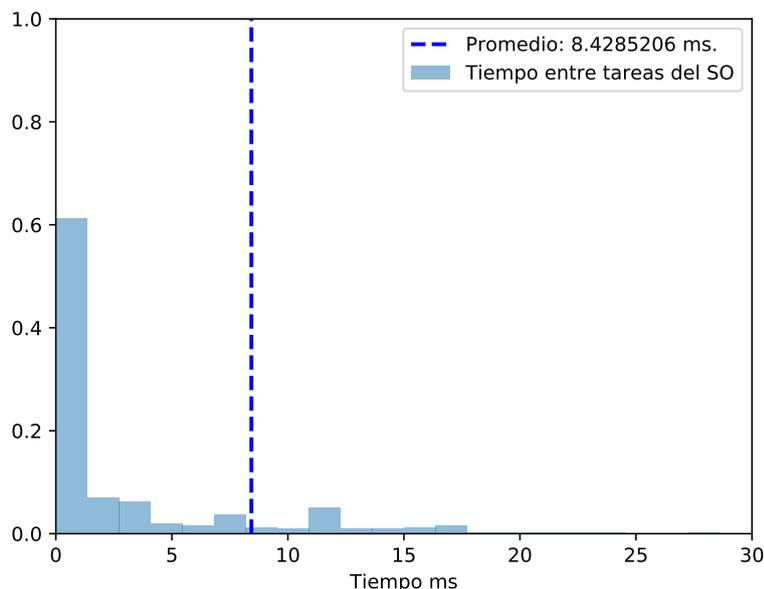


Figura 5.11: Distribución de tiempos cada cuanto el *scheduler* aloja una tarea del SO.

De la figura 5.11 se destaca que, si bien el caso promedio es considerablemente alto, las tareas del SO no tardan mucho tiempo en ser alojadas en el procesador. Esto quiere decir que el SO realiza un uso intenso del procesador para realizar sus tareas.

A diferencia con las actividades de *entrada salida*, las cuales tienen un uso del procesador prácticamente nulo, las tareas del SO parecieran afectar de manera directa al **TLP**. Esto es tanto debido a que comienzan a ejecutar rápidamente, como a que se prolongan una considerable cantidad de tiempo en el uso del core.

A continuación se estudiará la concurrencia de las mismas para poder observar su **TLP**. Con esto podremos determinar si son realmente un caso influyente en el **TLP** del sistema.

5.4.3 Concurrencia (TLP)

En la figura 5.12 se puede ver el gráfico de la distribución del **TLP** para las tareas del SO:

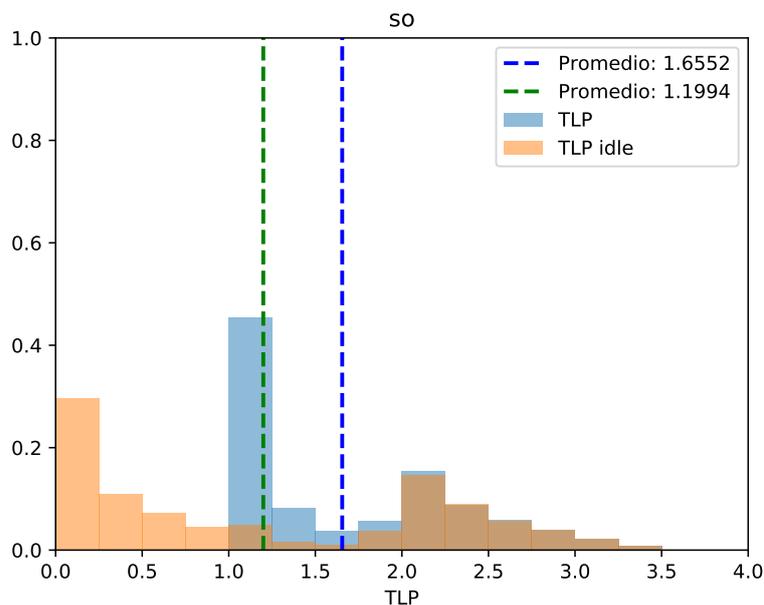


Figura 5.12: **TLP idle** vs **TLP** sin incluir tiempo *idle* para tareas del SO.

Se observa de la figura 5.12 que el **TLP** de estas tareas se comporta mejor que el **TLP** promedio de las aplicaciones de usuario estudiadas en la sección 5.1.1. También el **TLP** incluyendo el tiempo *idle* tiene un promedio alto si lo comparamos con las aplicaciones de usuario estudiadas. Esto significa que el tiempo ocioso de estas tareas es poco. Por lo tanto, podemos afirmar que las tareas del SO realizan un intenso uso del procesador, afectando así al **TLP** general del sistema.

Notar que las tareas del SO no son comunes a una sola aplicación. Es decir son todas tareas independientes identificadas por el mismo nombre, y medidas en términos de **TLP** como una sola.

5.5 Transiciones del procesador

A diferencia de los experimentos anteriores basados en **TLP** y tiempos de aplicaciones, en esta sección se analizará el uso del procesador de una manera global. Para eso se estudiarán las variaciones entre uso de cores simultáneos, introduciendo el concepto de *transición* entre cores.

5.5.1 Definición de transición

Definiremos una *transición* como el cambio de estado en el que se genera una modificación en la cantidad de cores que están ejecutando concurrentemente. Por ejemplo, si en un momento determinado de la muestra se están utilizando i cores, luego de recibir un evento `switch`, en la muestra podrían estar ejecutando con $i - 1$, i o $i + 1$ cores. En el caso en que se esté corriendo una cantidad distinta de cores (o sea $i - 1$ o $i + 1$ cores en este ejemplo) al estado anterior diremos que obtuvimos una *transición* de i a $i + 1$ cores (o de i a $i - 1$ si fuera el caso). La notaremos como $i \rightarrow i + 1$, o $i \rightarrow i - 1$, según sea el caso. Notar que también puede darse un evento de `switch` pero continuar en ejecución la misma cantidad de cores. Esto es que por que un proceso puede ser desalojado por otro proceso.

La medida que nos interesará calcular es el tiempo promedio entre dos transiciones. Es decir, cuánto es el tiempo promedio desde que se utilizan i cores hasta el que comienza a correr con $i + 1$ cores (de la misma manera para i cores a $i - 1$ cores).

Notar en el ejemplo que en el caso de estar en un estado de la muestra en el que solo se están utilizando 0 cores, nos interesará la *transición* de 0 a 1 core. Del mismo modo, en el estado con 4 cores ejecutándose concurrentemente, nos interesará solamente la *transición* de 4 a 3 cores.

En la figura 5.13 se ilustra una secuencia simplificada de eventos `switch`, mostrando las distintas *transiciones* que se generan. Todos los cores empiezan ejecutando la tarea *idle*.

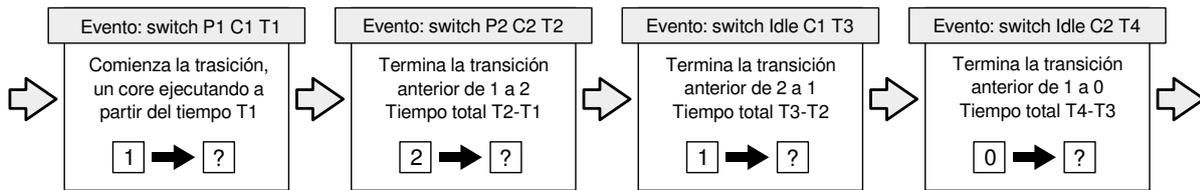


Figura 5.13: Ejemplo de *transición*.

Al llegar el evento `switch` con el proceso P1 en el core C1 en el tiempo T1, comienza una *transición* que puede ser $1 \rightarrow 0$ o $1 \rightarrow 2$. En la próxima *transición* se puede estar corriendo 0 o 2 cores, dependiendo si el proceso P1 es desalojado, o si otro proceso es alojado en otro core que no sea C1. Luego tenemos el evento `switch` con el proceso P2 en el core C2, por lo que, en el tiempo T2 se completa la *transición* $1 \rightarrow 2$ con duración de $T2 - T1$. Esto es debido a que se pasó de ejecutar un core a dos cores.

En el tiempo T3 ocurre el evento `switch` con la tarea *idle* en el core C1, por lo que termina la *transición* $2 \rightarrow 1$ con tiempo de duración $T3 - T2$ seg. Esto es debido a que se pasaron de ejecutar dos cores a un core.

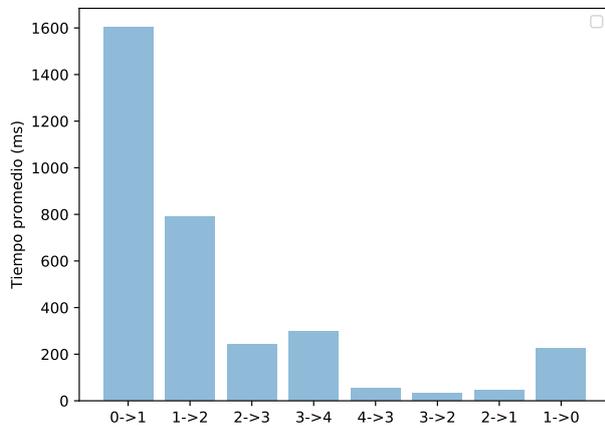
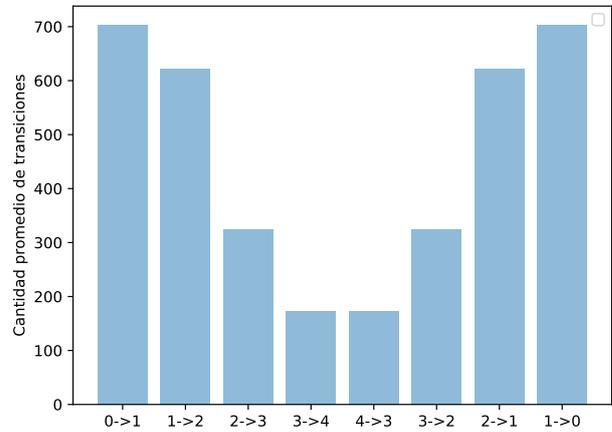
Luego, en el tiempo T4 ocurre el evento `switch` con la tarea *idle* en el core C2, terminando así la *transición* $1 \rightarrow 0$ con tiempo de duración $T4 - T3$.

5.5.2 Tiempo y cantidad promedio de las transiciones

A continuación se expone el gráfico de los tiempos promedios de cada una de las *transiciones* en el total de las muestras y la cantidad de veces promedio que ocurre cada *transición*.

Sobre el eje horizontal de la figura 5.14a, se encuentran cada una de las posibles *transiciones*. Como preveíamos por los resultados anteriores del presente trabajo, el mayor tiempo promedio pertenece a las *transiciones* de $0 \rightarrow 1$ y $1 \rightarrow 2$. Recordemos que la *transición* $1 \rightarrow 2$ nos indica cuánto tiempo pasa en promedio desde que la máquina se encuentra con un core asignado, hasta utilizar con dos cores. La

transición $1 \rightarrow 0$ nos indica el tiempo promedio que se tarda cuando la máquina se encuentra corriendo con un core al estado *idle*. Por lo tanto, sumando el promedio de la *transición* $1 \rightarrow 2$ y $0 \rightarrow 1$, obtenemos cuánto tiempo se ejecutó con un core. Además podemos ver que el gráfico no es simétrico, ya que estamos midiendo tiempos, y éstos pueden ser distintos.

(a) Promedio de tiempo de *transiciones*(b) Promedio de cantidad de *transiciones* por muestra

En el gráfico 5.14b se muestran las cantidades promedio por *transición*. La figura, a diferencia de la anterior, es simétrica, ya que estamos midiendo cantidades. Por lo tanto, es la misma la cantidad de transiciones que ocurren de $0 \rightarrow 1$ que las que ocurren de $1 \rightarrow 0$. La cantidad promedio de transiciones de $0 \rightarrow 1$ es casi tres veces más grande que la cantidad de $3 \rightarrow 4$ y $4 \rightarrow 3$.

CONCLUSIONES

Este trabajo propone un estudio del uso del procesador en términos de uso concurrente de los cores. Fue realizado tanto desde el **uso compartido** como del **TLP** del mismo. Esta última medida fue utilizada por Flautner et al en el año 2000, y por Blake et al en el 2010.

A diferencia de estos trabajos que fueron realizados sobre *benchmarks* de programas específicos, en nuestro trabajo se han tomado muestras de trazas de uso real del procesador de máquinas de los laboratorios de estudiantes universitarios.

Se realizó un estudio de uso de forma global del procesador. Al ser las muestras un conjunto de programas con un uso del procesador determinado, se pudo generalizar el comportamiento de todos los programas al realizar las mediciones sobre todo el conjunto de muestras. Los resultados dan un **TLP_{idle}** promedio de 0,5 y un **uso compartido** de 0,4.

Se estudiaron los programas más usados por los usuarios de las máquinas. Los mismos fueron clasificados en distintas categorías, como por ejemplo Browsers, programas de matemática, editores de texto. Cuando el **TLP** es muy cerca 1, se considera que el mismo representa un uso serial. Esto sucede con los programas pertenecientes a la categoría de edición de texto. De los programas estudiados, los que mejor **TLP** promedio presentan son los browsers, aproximadamente de 1,5.

En Blake et al, el promedio de **TLP** de los browsers es de 1,55, mientras que en el presente trabajo se obtuvo para los browsers un **TLP** promedio de 1,5. Por lo tanto a pesar del paso de los años, estos promedios se mantienen similares. Por otro lado, en Blake, programas de edición de texto u oficina obtienen un **TLP** de 1,2, mientras que en el presente trabajo estos programas muestran un comportamiento serial.

Además, se corroboró que las actividades de *entrada salida* realizan un uso limitado del procesador. Por lo tanto tiene muy poca influencia en el uso final del mismo.

Por otro lado, las tareas propias del SO muestran un buen **TLP** promedio, dando números que evidencian que las mismas podrían afectar al uso concurrente del procesador a nivel general.

De los experimentos realizados, se desglosa que a pesar del paso de los años, aún no se evidencian grandes cambios en el diseño de los programas en términos de paralelización de tareas. Este es un gran problema a corregir por los nuevos desarrolladores, ya que de hacerlo, las bondades del procesador serían cada vez mejor explotadas para un mejor funcionamiento de sus aplicaciones.

TRABAJO FUTURO

La continuación del presente trabajo se plantea desde dos ejes.

Por un lado la recolección de más datos para el análisis, cambiando las características del ambiente sobre el cual se realiza el muestreo. Podría ser en laboratorios universitarios con distinta cantidad de *cores*, o sobre ambientes de oficinas con perfiles de usuarios distintos a alumnos universitarios.

Por el otro, utilizar los datos para un fin específico, produciendo trazas sintéticas.

Por lo tanto, los trabajos futuros podrían ser:

1. Generador de carga sintética del procesador: A partir de todas las trazas conseguidas, reproducir las mismas de manera sintética para ser usadas en otras áreas de investigación dentro de la Arquitectura del procesador.
2. Aplicaciones de otras mediciones o análisis sobre los datos: Se podría evaluar otras magnitudes como por ejemplo, eficiencia del *scheduler* en la asignación de procesos, o estudiar características particulares de algún programa específico hallado en las muestras.
3. Continuación del estudio con un ambiente de muestreo distinto: Esto generaría muestras de características distintas a las del trabajo. Estudiar si los mismos resultados se obtienen cambiando el perfil del usuario, el hardware de la máquina, o los programas específicos que corren en la misma.

BIBLIOGRAFÍA

- [BDMF10] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. 38(3):302–313, June 2010.
- [Chu19] Sudheer Chunduri. Dynamic frequency scaling, 2019.
- [FG07] Karl Furlinger and Michael Gerndt. Analyzing overheads and scalability characteristics of openmp applications. In *Proceedings of the 7th international conference on High performance computing for computational science, VECPAR’06*, pages 39–51, Berlin, Heidelberg, 2007. Springer-Verlag.
- [FURM00] Kristián Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. Thread-level parallelism and interactive performance of desktop applications. 28(5):129–138, November 2000.
- [GGD⁺14] C. Gao, A. Gutierrez, R. G. Dreslinski, T. Mudge, K. Flautner, and G. Blake. A study of thread level parallelism on mobile devices. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 126–127, March 2014.
- [GGR⁺15] C. Gao, A. Gutierrez, M. Rajan, R. G. Dreslinski, T. Mudge, and C. Wu. A study of mobile device utilization. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, volume 00, pages 225–234, March 29-31 2015.
- [HZR16] M. Halpern, Y. Zhu, and V. J. Reddi. Mobile cpu’s rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–76, March 2016.
- [MPI19] Open MPI. Open mpi, 2019.
- [Rup19] Karl Rupp. 40-years-of-microprocessor-trend-data, 2019.
- [Tan13] Andrew Tanenbaum. *Sistemas operativos modernos*, 2013.