Software para arquitecturas basadas en procesadores de múltiples núcleos. Detección automática de errores de concurrencia.

Fernando Emmanuel FRATI*

Director: Armando E. DE GIUSTI†

Codirectores: Marcelo NAIOUF†
y Katzalin OLCOZ HERRERO‡

Asesor científico: Luis PIÑUEL MORENO‡

Tesis presentada para obtener el grado de
Doctor en Ciencias Informáticas

Facultad de Informática
Universidad Nacional de La Plata

marzo 2015

*Universidad Nacional de Chilecito, Argentina
†Facultad de Informática, Universidad Nacional de La Plata, Argentina
‡Facultad de Informática, Universidad Complutense de Madrid, España

Dedicado
a mis dos amores
Patricia y Rosario

Agradecimientos

Esta tesis resume en unas pocas páginas varios años de esfuerzo, compromiso y dedicación.

Inicialmente quiero agradecer a la Universidad Nacional de Chilecito (UNdeC) por confiar en mi y ofrecerme esta oportunidad y al Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), por la financiación obtenida a través del programa de becas de postgrado, sin el que me habría sido imposible seguir este camino.

Aunque en apariencia es un trabajo personal, siento que no habría sido posible sin la intervención de muchas personas que de distintas formas estuvieron ahí para sostenerme y empujarme en los momentos que me hizo falta.

Quiero empezar agradeciendo a mis padres Bety y Fernando, y a través de ellos a toda mi familia, que sin cuestionar mis decisiones siempre me apoyaron, aunque eso significara llevarme lejos de casa.

A Teresa y Orlando, mis segundos padres, que confiaron en mis promesas de regreso cuando con Pato les dijimos que nos íbamos a La Plata. En especial quiero agradecer a Teresa su apoyo cuidando a Rosario para que pudiera completar este trabajo: sin su ayuda habría sido muy difícil terminar este postgrado.

A mi prima Mary, mi tío y toda su familia, por la contención y apoyo que nos dieron durante los cinco años que vivimos en Buenos Aires.

A mis amigos Nely y Edu, que jamás permitieron que la distancia fuera un obstáculo para hacerme llegar su cariño y palabras de aliento.

A mis amigos Diego, Dieguito, Enzo, Andrea, Adriana y José Daniel, todos compañeros de ruta, con quienes compartimos angustias y alegrías: por extraño que parezca, hasta conocerlos pensaba que estaba sólo en este proceso. Gracias por su apoyo incondicional todos estos años!

A todo el grupo de becarios y pasantes del LIDI, por su amistad y compañía, por las risas y reuniones, por todos los mates, almuerzos y cenas compartidos... en fin, por hacer del LIDI un gran lugar para trabajar.

A Franco, Adrián y Fernando, por su generosidad, predisposición y buena voluntad para compartir todo lo que saben. Lo que aprendí de ustedes seguro no puedo encontrarlo en Internet!!!

Al personal de postgrado de la Facultad de Informática de la UNLP, por el excelente trato y buena predisposición para resolver cualquier asunto planteado.

A mis profesores y amigos Pimpo y Fernanda, que desde el primer día en que nos conocimos creyeron en mi y siempre me ayudaron a aprovechar las oportunidades que tuve. Gracias por estar siempre para escucharme y aconsejarme para bien!

A mis directores Tito y Marcelo, quienes me recibieron y aceptaron como becario sin conocerme, sin ninguna experiencia previa en investigación, y confiaron en que sería capaz de completar el doctorado. Gracias por todas las oportunidades que me dieron, por sus consejos y orientación durante todo el proceso.

A Luis y Katza, por su apoyo y orientación para la definición de mi doctorado. En especial quiero agradecer a Katza por su constante apoyo y paciencia: cada correo, conversación o devolución en estos años ha sido tan importante para mi que no habría logrado llegar hasta aquí sin su ayuda.

A Rodrigo (Rekai), por el apoyo recibido cuando comencé a definir mi tema de tesis, y a todo el grupo ArTeCS de la UCM, siempre dispuestos a colaborar. Gracias por su profesionalismo y generosidad para resolver cualquier problema en el que necesité de su ayuda!

Finalmente, quiero agradecer y dedicar las últimas líneas a las dos personas más importantes de mi vida: Pato, sos la gran culpable de que crea en mi; sin tu apoyo, sacrificio y compañía todos estos años, nada de esto sería posible. Solo vos conocés el esfuerzo que nos significó seguir esta aventura... y a Rosario, mi hija: cuando creía que estaba haciendo lo más importante de mi vida, llegaste a poner las cosas en su lugar... así como responsabilizo a tu mamá por animarme a comenzar el doctorado, vos sos la razón por la que quise terminarlo. A ustedes, mis dos amores, les dedico esta tesis.

Resumen

Todos los procesadores disponibles en el mercado (incluso los procesadores utilizados en dispositivos móviles) poseen una arquitectura típica multicore. En consecuencia, el modelo de programación en memoria compartida se impuso sobre el modelo de programación secuencial como modelo por excelencia para obtener el máximo desempeño de estas arquitecturas.

En este modelo de programación las suposiciones de orden de ejecución entre instrucciones y atomicidad en el acceso a las variables heredadas del modelo de programación secuencial ya no son válidas. El no determinismo implícito en la ejecución de los programas concurrentes, obliga al programador a utilizar algún mecanismo de sincronización para asegurar esas propiedades.

Frecuentemente el programador se equivoca al sincronizar los procesos, dando lugar a nuevos errores de programación como son los deadlocks, condiciones de carrera, violaciones de orden, violaciones de atomicidad simple y violaciones de atomicidad multivariable. Los métodos tradicionales de depuración de programas no son aplicables en el contexto de los programas concurrentes, por lo que es necesario disponer de herramientas de depuración que puedan ayudar al programador a detectar esta clase de errores.

De estos errores, los deadlocks y las condiciones de carrera han gozado de mayor popularidad en la comunidad científica. Sin embargo, solo el 29,5 % de los errores son deadlocks: del 70,5 % restante, *las violaciones de atomicidad representan más del 65 % de los errores*, el *96 % ocurren entre dos threads* y *el 66 % involucran una sola variable*. Por eso las violaciones de atomicidad simple se han definido en los últimos años como el caso más general de error de concurrencia y han recibido gran atención por numerosos grupos de investigación.

En 2005 aparecen las primeras propuestas que utilizan métodos de instrumentación dinámicos para la detección de violaciones de atomicidad, mejorando notablemente la capacidad de detección sobre las propuestas anteriores. De estas propuestas, AVIO(Lu,

Tucek, Qin, y Zhou, 2006) se destaca como la propuesta con mejor rendimiento y capacidad de detección. Para detectar una violación de atomicidad, el método de AVIO consiste en monitorizar los accesos a memoria por parte de los procesos concurrentes durante la ejecución, registrando qué procesos acceden a cada variable, en búsqueda de interleavings no serializables. Pese a que AVIO es superior a las propuestas previas, el overhead que introduce (en promedio 25×) es demasiado elevado para ser utilizado en entornos en producción.

Muchas propuestas proponen reducir el overhead de los algoritmos de detección implementándolos directamente en el hardware a través de extensiones (cambios en el procesador, memoria cache, etc.), consiguiendo excelentes resultados. Sin embargo, este enfoque requiere que los fabricantes de procesadores decidieran incorporar esas modificaciones en sus diseños (cosa que no ha sucedido por el momento), por lo que es de esperar que tardarán en llegar al mercado y más aún en reemplazar las plataformas que actualmente están en producción.

Por otro lado, las implementaciones en software aplican métodos de instrumentación de programas. Debido a que requieren agregar llamadas a una *rutina de análisis* a cada instrucción que accede a la memoria, los métodos de detección de errores utilizan instrumentación a nivel de instrucción. Lamentablemente, este granularidad de instrumentación es lenta, penalizando el tiempo de la ejecución con más de un orden de magnitud.

Sin embargo, la posibilidad de error solamente existe si al menos dos threads acceden simultáneamente a datos compartidos. Esto significa que, si de la totalidad de la aplicación que está siendo monitorizada sólo un pequeño porcentaje de las operaciones acceden a datos compartidos, gran parte del tiempo invertido en instrumentar todos los accesos a memoria está siendo desperdiciado.

Para reducir el overhead de la instrumentación a nivel de instrucción restringiéndolo sólo a los accesos a memoria compartida, es necesario detectar el momento preciso en que esos accesos ocurren. La mejor opción para detectar este momento es cuando ocurre algún cambio en la memoria cache compartida entre los núcleos que ejecutan los procesos.

Una herramienta muy útil para esta tarea son los *contadores hardware*, un conjunto de registros especiales disponibles en todos los procesadores actuales. Esos registros pueden ser programados para contar el número de veces que un evento ocurre dentro del procesador durante la ejecución de una aplicación. Los eventos proveen información sobre diferentes aspectos de la ejecución de un programa (por ejemplo el número de instruc-

ciones ejecutadas, el número de fallos en cache L1 o el número de operaciones en punto flotante ejecutadas).

Se plantea como estrategia encontrar un evento que detecte la ocurrencia de interleavings no serializables y en función de ello activar/desactivar AVIO. Lamentablemente, no existe un evento capaz de indicar la ocurrencia de casos de interleavings. Sin embargo, si es posible representar los casos a través de patrones de acceso a memoria.

La búsqueda de eventos asociados a los cambios de estado en el protocolo de coherencia cache reveló que para la arquitectura de pruebas existe un evento, cuya descripción indica que ocurre con uno de los patrones de acceso presentes en los casos de interleavings. El patrón asociado al evento está presente en tres de los cuatro casos de interleavings no serializables que AVIO debe detectar. La experimentación realizada para validar el evento demostró que efectivamente ocurre con precisión con el patrón de acceso, y en consecuencia puede detectar la ocurrencia interleavings no serializables.

Luego de determinar la viabilidad del evento seleccionado, se experimentó con los contadores en un modo de operación llamado *muestreo*, el cual permite configurar los contadores para generar señales dirigidas a un proceso ante la ocurrencia de eventos. En este modo el programador especifica la cantidad de eventos que deben ocurrir antes de que la señal sea generada, permitiendo ajustar esta prestación de acuerdo a los requerimientos de la aplicación.

Este modo de operación fue utilizado para decidir cuándo activar la rutina de análisis de las herramientas de detección y en consecuencia reducir la instrumentación del código. Por otro lado, el desactivado puede ser un poco más complejo. Debido a que no es posible configurar un contador para enviar una señal ante la *no ocurrencia* de eventos, se propone configurar un *timer* para verificar a intervalos regulares de tiempo si es seguro desactivar la rutina de análisis (por ejemplo porque en el último intervalo no se detectaron violaciones de atomicidad).

El modelo propuesto se utilizó para implementar una nueva versión llamada AVIO-SA, la cual inicia la ejecución de las aplicaciones monitorizadas con la rutina de análisis desactivada. En el momento en que detecta un evento la rutina es activada, funcionando por un tiempo como la versión original de AVIO. Eventualmente AVIO deja de detectar interleavings y la rutina de análisis es desactivada.

Debido a que no es posible estimar el valor óptimo para el tiempo del intervalo de muestreo analíticamente, se desarrollaron experimentos para encontrar este valor empíricamente. Se encontró que un intervalo de 5ms permite a AVIO-SA detectar aproxima-

damente la misma cantidad de interleavings que AVIO, pero con un tiempo de ejecución significativamente menor.

Para completar las pruebas de rendimiento se completaron los experimentos con HELGRIND, una herramienta libre de detección de condiciones de carrera y se estimó el overhead de cada herramienta con respecto a cada aplicación. En promedio, HELGRIND demostró un overhead de 223×, AVIO un overhead de 32× y AVIO-SA de 9×.

Aparte del rendimiento, se evaluó la capacidad de detección de errores de AVIO-SA. Para ello se hicieron 3 experimentos:

- Prueba de detección con kernels de bugs conocidos.
- Prueba de detección en aplicaciones reales (Apache).
- Comparación de bugs informados entre AVIO y AVIO-SA (a partir de SPLASH-2).

Afortunadamente AVIO-SA pasó las 3 pruebas satisfactoriamente.

Los resultados obtenidos demuestran que el modelo propuesto no afecta negativamente la capacidad de detección de la herramienta, empleando en el proceso menos del 30 % del tiempo requerido por AVIO. Debido a que AVIO-SA altera menos la historia de ejecución de la aplicación monitorizada, es una mejor opción para ser utilizada en entornos de producción.

Índice general

Publicaciones

Libros

- Frati, F. E., De Giusti, A. E., Naiouf, M. (2014). Evaluación de técnicas de detección de errores en programas concurrentes. *Trabajo Final presentado para obtener el grado de Especialista en Cómputo de Altas Prestaciones y Tecnología GRID*. Facultad de Informática, Universidad Nacional de La Plata, Argentina. Recuperado a partir de http://hdl.handle.net/10915/36923

Revistas

- González-Alberquilla, R., Frati, F. E., Piñuel, L., Strauss, K., & Ceze, L. (2013). Data Race Detection with Minimal Hardware Support. *The Computer Journal*. http://dx.doi.org/10.1093/comjnl/bxt013

- Frati, F. E., Olcoz Herrero, K., Piñuel Moreno, L., Naiouf, M., & De Giusti, A. E. (**EN PREPARACIÓN**). Sharing Awareness Model for concurrency bug detections tools. Para ser enviado a *New Generation Computing*.

Proceedings en Congresos

- Frati, F. E., Olcoz Herrero, K., Piñuel Moreno, L., Montezanti, D. M., Naiouf, M., & De Giusti, A. E. (2011). Optimización de herramientas de monitoreo de errores de concurrencia a través de contadores de hardware. Presentado en *XVII Congreso Argentino de Ciencias de la Computación*. Universidad Nacional de La Plata, Argentina. Recuperado a partir de http://hdl.handle.net/10915/18688

- Frati, F. E., Olcoz Herrero, K., Piñuel Moreno, L., Naiouf, M., & De Giusti, A. E. (2012a). Detección de interleavings no serializables usando contadores de hardware. Presentado en *XVIII Congreso Argentino de Ciencias de la Computación*. Universidad Nacional del Sur, Argentina. Recuperado a partir de http://hdl.handle.net/10915/23631

- Frati, F. E., Olcoz Herrero, K., Piñuel Moreno, L., Naiouf, M. R., & De Giusti, A. (2012b). Unserializable Interleaving Detection using Hardware Counters. En *IASTED International Conference Parallel and Distributed Computing and Systems* (pp. 230-236). Las Vegas, USA: T. Gonzalez, M.H. Hamza. http://dx.doi.org/10.2316/P.2012.789-058

- Frati, F. E., Olcoz Herrero, K., Piñuel Moreno, L., Naiouf, M., & De Giusti, A. E. (2014). Modelo para sintonización dinámica de aplicaciones guiada por eventos de hardware. Presentado en *XX Congreso Argentino de Ciencias de la Computación*. Universidad Nacional de La Matanza, Argentina. Recuperado a partir de http://hdl.handle.net/10915/42392

Proceedings en Workshops

- Naiouf, M., Chichizola, F., De Giusti, L. C., Montezanti, D. M., Rucci, E., Frati, F. E., ... De Giusti, A. E. (2013). Tendencias en arquitecturas y algoritmos paralelos. En Proceedigns del XV Workshop de Investigadores en Ciencias de la Computación (Vol. XV, pp. 690-695). Paraná, Entre Ríos: Red UNCI. Recuperado a partir de http://hdl.handle.net/10915/27294

- Naiouf, M., Chichizola, F., De Giusti, L. C., Montezanti, D. M., Rucci, E., Frati, F. E., ... De Giusti, A. E. (2014). Tendencias en Arquitecturas y Algoritmos Paralelos para HPC. En Proceedings del XVI Workshop de Investigadores en Ciencias de la Computación (Vol. XVI, pp. 712-717). Ushuaia, Tierra del Fuego: Red UNCI. Recuperado a partir de http://hdl.handle.net/10915/42798

Resúmenes en Jornadas

- Frati, F. E. (2013). Software para arquitecturas basadas en procesadores multicore. Detección automática de errores de concurrencia. Presentado en *II Jornadas Científicas de Estudiantes Investigadores*. Universidad Nacional de Chilecito, Argentina

- Frati, F. E. (2014). Sintonización dinámica de aplicaciones guiada por eventos de hardware. Presentado en *III Jornadas Científicas de Estudiantes Investigadores*. Universidad Nacional de Chilecito, Argentina

Publicaciones complementarias

- De Giusti, A. E., & Frati, F. E. (2010). ¿Concurrencia y Paralelismo en el primer curso de Algorítmica? En Proceedings del V Congreso de Tecnología en Educación y Educación en Tecnología (Vol. V, p. 7). El Calafate: Red UNCI. Recuperado a partir de http://hdl.handle.net/10915/20430

- De Giusti, A., Frati, F. E., Leibovich, F., Sanchez, M., De Giusti, L., & Madoz, M. C. (2012). Concurrencia y Paralelismo en CS1: la utilización de un Lenguaje Visual orientado. En Proceedings del VII Congreso de Tecnología en Educación y Educación en Tecnología (Vol. VII, p. 8). Pergamino, Buenos Aires: Red UNCI. Recuperado a partir de http://hdl.handle.net/10915/18451

- De Giusti, A., Frati, F. E., Sanchez, M., & De Giusti, L. (2012). LIDI Multi Robot Environment: Support software for concurrency learning in CS1. En 2012 International Conference on Collaboration Technologies and Systems (CTS) (pp. 294 -298). Denver, CO, USA: Waleed W. Smari and Geoffrey Charles Fox. Recuperado a partir de http://dx.doi.org/10.1109/CTS.2012.6261065

- De Giusti, L., Frati, F. E., Leibovich, F., Sanchez, M., & Madoz, M. C. (2012). LMRE: Un entorno multiprocesador para la enseñanza de conceptos de concurrencia en un curso CS1. Revista Iberoamericana de Tecnología en Educación y Educación en Tecnología, 7, 9. ISSN: 1850- 9959. Recuperado a partir de http://hdl.handle.net/10915/18283

- Leibovich, F., Frati, F. E., & Sanchez, M. (2012). Elaboración de conceptos de concurrencia en CS1 a través de un entorno visual orientado. AUGM 2012, 8.

- Tinetti, F. G., Martin, S. M., Frati, F. E., & Méndez, M. (2013). Optimization and Parallelization experiences using hardware performance counters. En Proceedings of the 4th International Supercomputing Conference in México (p. 5). Manzanillo, Colima, México.

Capítulo 1

Introducción

En este capítulo se presenta una introducción al problema de detectar errores de programación (1.1), en particular aquellos relacionados con programas concurrentes para cómputo de altas prestaciones. Luego se enuncian los objetivos, la metodología (1.2) y las contribuciones (1.3). Finalmente se presenta la organización del resto de la tesis (1.4).

1.1. Fundamentación

Una característica fundamental de los sistemas de software es que se construyen desde el principio sabiendo que deberán incorporar cambios a lo largo de su ciclo de vida. Todos los libros que tratan sobre ingeniería de software coinciden en que los sistemas son evolutivos. Incluso al evaluar el esfuerzo que se debe invertir en un proyecto de software, se considera que un 20 % está en el desarrollo y 80 % se aplica al mantenimiento (Pfleeger y Atlee, 2009). Ian Sommerville estima que el 17 % del esfuerzo de mantenimiento se invierte en localizar y eliminar los posibles defectos de los programas (Sommerville, 2006). Por ello, conseguir programas libres de errores es uno de los principales objetivos que se plantea (o se debería plantear) el desarrollador frente a cualquier proyecto de software.

Por otro lado, los requerimientos computacionales actuales demandan cada vez mayor poder de cómputo. Algunas áreas con requerimientos de este tipo son:

- Simulación y modelos: un modelo climático más preciso que los actuales requiere tomar en cuenta las interacciones de la atmósfera, los océanos, los continentes, el hielo en los polos, etc.

- Computación científica: se cree que el plegamiento de proteínas podría estar involucrado en enfermedades como Parkinson o Alzheimer, pero la habilidad para estudiar configuraciones de moléculas complejas como las proteínas está limitada por el poder computacional.

- Análisis de datos: se estima que la cantidad de datos almacenados en el mundo se dobla cada dos años(Gantz y Reinsel, 2012). Un conjunto de datos sin analizar es inútil: procesar una gran cantidad de datos para producir información útil requiere costosos procesos de análisis.

- Juegos: uno de los mercados que más dinero invierte en el desarrollo tecnológico es el de los juegos. De hecho, las principales consolas de videojuegos (PlayStation 4, XBox One y Wii U) poseen procesadores multicore como los que se pueden encontrar en servidores de alto desempeño.

Esto indudablemente deriva en que la comunidad de usuarios con requerimientos de alto desempeño (HPC por sus siglas en inglés) cada vez sea mayor.

Durante décadas, la industria respondió a la creciente demanda de mayor poder computacional incrementando exponencialmente el rendimiento de los procesadores. Esto fue posible gracias a tres tecnologías clave(Borkar y Chien, 2011):

- *Escalado de transistores*: a principios de los años 70, Robert N. Dennard en IBM ideó un método (conocido como la ley Dennard) para reducir las dimensiones de los transistores un 30 % cada dos años(Dennard, Gaensslen, Rideout, Bassous, y LeBlanc, 1974). Este método permitió que con cada generación nueva de transistores el nivel de integración se doble, los circuitos fueran 40 % más rápidos y el consumo energético del sistema permaneciera constante.

- *Diseño de la microarquitectura*: contar con mayor cantidad de transistores disponibles permitió el desarrollo de técnicas como pipeling, predicción de saltos, ejecución fuera de orden y especulación, que permitieron incrementar aún más el rendimiento de los procesadores.

- *Arquitectura de memoria*: aunque la densidad de la memoria se ha doblado cada dos años durante los últimos 40 años, su rendimiento no ha aumentado a la misma velocidad. Como consecuencia, la memoria se convirtió en un cuello de botella que reduce el rendimiento general del sistema. Sin embargo, esta realidad condujo

a la industria en el diseño de jerarquías de memoria, empleando caches (pequeñas memorias de alta velocidad) en dos o más niveles antes de llegar a la memoria principal(Przybylski, Horowitz, y Hennessy, 1989). Esta jerarquía permitió emular una memoria de alta velocidad que, aprovechando las propiedades de localidad espacial y temporal de los programas, permitió satisfacer las necesidades de un gran ancho de banda y baja latencia de los procesadores.

Desafortunadamente, la ley de escalado de Dennard finalizó hace una década (con la tecnología de 130nm) y con ella el escalado en frecuencia de los procesadores. Aunque la introducción de nuevos materiales y nuevas estructuras 3D (strained SiGe, SOI, dieléctricos hight-k, transistores FinFET, near threshold transistors, etc.) siguen permitiendo escalar el tamaño de los transistores – la ley de Moore continúa –, la velocidad y la energía de los mismos ya no escala a los ritmos anteriores. En este sentido la eficiencia energética se ha convertido en el factor fundamental que limita el rendimiento de los microprocesadores y los sistemas en general (Ahmed y Schuegraf, 2011).

En la actualidad, la eficiencia energética es equivalente al rendimiento, y con esta restricción dejan de ser atractivos muchos de los mecanismos que se propusieron durante las décadas pasadas (Borkar y Chien, 2011; Esmaeilzadeh, Blem, St. Amant, Sankaralingam, y Burger, 2012). La presión que ejerce el requerimiento de eficiencia energética conduce hacia procesadores multicore heterogéneos, con pocos cores complejos para tareas predominantemente secuenciales y un gran número de pequeños y simples núcleos para aumentar productividad. En el futuro las mejoras de rendimiento de los procesadores provendrán de forma predominante del paralelismo a nivel de tarea y no de incrementar la frecuencia de reloj o la complejidad del procesador.

Desde el año 2005, el escalado tecnológico se ha venido aprovechando para aumentar el número de cores dentro del chip, dando lugar a los procesadores de múltiples núcleos. Los recursos tecnológicos para enfrentar problemas de HPC que antes estaban restringidos a selectos grupos con capacidad de inversión en clusters y servidores de alto desempeño, hoy están disponibles en una variedad de configuraciones: multicores, comodity clusters y/o GPUs. Incluso el segmento del mercado destinado a dispositivos móviles (tablets, celulares y televisores inteligentes, dispositivos embebidos, etc.) comienza a valerse del procesamiento multicore para ofrecer nuevos servicios y prestaciones a sus usuarios (Yeap, 2013).

A corto plazo es necesaria una gestión más eficiente de este tipo de arquitecturas, ya que el modelo seguido para su diseño ha tratado de reusar muchos de los mecanismos diseñados en el contexto *single core*.

Es de interés destacar que para obtener la máxima eficiencia de estas arquitecturas, es necesario el desarrollo de programas concurrentes (Grama, Gupta, Karypis, y Kumar, 2003). A diferencia de los programas secuenciales, en un programa concurrente existen múltiples hilos en ejecución que acceden simultáneamente a datos compartidos, dando lugar a nuevos tipos de errores (deadlocks, condiciones de carrera, violaciones de atomicidad entre otros). El orden en que ocurren estos accesos a memoria puede variar entre ejecuciones, haciendo que los errores sean más difíciles de detectar y corregir.

Este contexto representado por múltiples núcleos de procesamiento que operan a distintas frecuencias, jerarquías de memoria que involucran varios niveles de cache y programas concurrentes diseñados para sacar provecho de estas arquitecturas incrementan aún más las probabilidades de ocurrencia de errores no detectados en etapas de desarrollo.

En HPC donde los tiempos de ejecución de las aplicaciones pueden variar de un par de horas hasta días, la ocurrencia de un error adquiere una importancia mayor. Existen diferentes propuestas destinadas a la detección y/o superación de esta clase de errores, pero dadas las características especiales de este tipo de programas es probable que el software sea liberado cuando aún posee errores ocultos. La forma de lidiar con ellos consiste en ejecutar una herramienta de detección de errores durante la etapa de producción de los programas, y emitir una alerta cuando un nuevo error se manifiesta. Sin embargo, la eficiencia de los programas monitorizados se ve comprometida por el overhead que introduce el proceso de detección.

La ingeniería del software enfrenta nuevos desafíos, donde el paralelismo tiene un rol protagónico. Por este motivo, resulta indispensable contar con herramientas que ayuden al programador en la tarea de verificar los algoritmos concurrentes y desarrollar tecnología robusta para tolerar los errores no detectados.

1.2. Objetivos y metodología

El objetivo principal del presente trabajo es proponer un modelo de implementación en software para herramientas de detección de errores de concurrencia, que permita reducir el overhead del proceso sin disminuir su capacidad de detección.

Los objetivos específicos son:

1. Caracterizar los errores que afectan a los programas concurrentes.

2. Describir las propuestas de la comunidad científica para la detección de errores de concurrencia.

3. Comparar las estrategias para reducir el overhead empleadas en las propuestas anteriores.

4. Explicar las causas del overhead en el proceso de detección.

5. Identificar las oportunidades de optimización de las herramientas con base en la comparación realizada.

6. Diseñar un modelo de desarrollo que permita reducir el overhead sin disminuir la capacidad de detección de las herramientas.

7. Evaluar la efectividad del modelo propuesto en base al overhead y la capacidad de detección.

La investigación de esta tesis es de carácter proyectivo, ya que remite a la creación, diseño o propuesta de un modelo con base en un proceso de investigación (Hurtado de Barrera, 2008). Esto significa que el mismo proceso debe proporcionar la información necesaria para desarrollar la investigación, y que se reconoce que la situación planteada amerita ser cambiada para mejorar el proceso generador de los eventos estudiados. En el caso de este trabajo los eventos a modificar están dados por el overhead y la capacidad de detección, mientras que el proceso generador (o causal) que da lugar a esos eventos es el proceso de detección de errores de concurrencia. De acuerdo a la metodología, los objetivos están planteados de manera tal que el cumplimiento de los mismos guían el proceso de investigación a través de los niveles de conocimiento perceptual, aprehensivo, comprensivo y finalmente integrativo, en el que se satisface el objetivo principal.

En particular se realizarán las siguientes acciones:

- Se llevará a cabo un relevamiento de la bibliografía existente en el tema a partir de revistas, publicaciones y tareas de otros grupos de investigación, como pueden ser las publicadas por IEEE y/o ACM.

- Se instalará en un entorno de prueba real las versiones disponibles de herramientas de detección de errores que permitan tomar muestras para evaluar posteriormente el modelo propuesto.
- Se estudiará el uso de herramientas de instrumentación de software empleadas por las propuestas para implementar tareas de análisis de programas como profiling, evaluación de performance y detección de errores.
- Se seleccionará e implementará una versión propia de una de estas herramientas para luego integrarla con las propuestas de mejora. En esta etapa se buscará analizar hipótesis, experimentar y obtener resultados, trabajar sobre la arquitectura real en la investigación experimental y obtener conclusiones.
- Se medirá y compará la herramienta mejorada a partir del overhead y capacidad de detección con respecto a las versiones originales. Para esto se utilizarán las suites de benchmarks presentes en trabajos relacionados. Durante esta etapa se evaluarán los resultados obtenidos y se refinará la propuesta tecnológica a incorporar al software paralelo sobre arquitecturas basadas en multicores.

1.3. Contribuciones

La principal contribución de esta tesis es el modelo de instrumentación dinámica guiado por eventos de hardware, el cual permite implementar herramientas de detección de errores de concurrencia que son activadas dinámicamente sólo cuando se está accediendo a datos compartidos. Este modelo se usó para implementar una versión optimizada de una herramienta de detección de violaciones de atomicidad llamada AVIO (Lu y cols., 2006), consiguiendo una reducción considerable en el overhead sin afectar negativamente su capacidad de detección de errores.

La reducción en el overhead se consiguió empleando los contadores hardware presentes en todos los procesadores actuales para determinar zonas de código donde fuera seguro deshabilitar la herramienta de detección. De manera complementaria a la contribución principal, una segunda contribución es un método para validación de eventos de hardware. Aunque los eventos de hardware se encuentran documentados en los manuales de las empresas que fabrican los procesadores, se encontró que muchas veces no es clara la relación del nombre o incluso la descripción con lo que el usuario desea medir: el método propuesto permite al usuario aproximarse experimentalmente al evento que está evaluando

y en función de los resultados determinar si es útil para sus necesidades. En esta tesis fue necesario aplicar el método para determinar si un evento de hardware en particular podía indicar que el proceso en cuestión había accedido o no a datos compartidos con otro proceso.

Por último, una tercera contribución consiste en una comparación de las técnicas y métodos vigentes en la comunidad científica aplicados a la detección y corrección de errores de programación en programas concurrentes y un marco teórico de referencia sobre esta clase de errores. Esta contribución sienta las bases para estudios futuros en el área de esta investigación.

1.4. Organización de la tesis

El resto de esta tesis está dividido en los siguientes capítulos:

- Capítulo 2: introduce el marco teórico de referencia. En este capítulo se presentan conceptos empleados en el resto de la tesis, se caracterizan los errores de concurrencia y se justifica la elección de violaciones de atomicidad como caso más general.

- Capítulo 3: presenta los antecedentes de la comunidad científica en términos de propuestas de detección de violaciones de atomicidad. Se describen las propuestas de otros autores y se justifica la elección del caso de estudio que guía el trabajo del resto de la tesis.

- Capítulo 4: delimita el problema a ser tratado en el proceso de investigación y discute las ventajas y desventajas de los métodos empleados por las propuestas anteriores. En este capítulo se identifican las causas que dan origen al overhead en el proceso de detección. Al final se enuncian las oportunidades de optimización detectadas.

- Capítulo 5: en este capítulo se presenta el uso de contadores hardware para detectar interleavings no serializables y se propone un método para validar eventos de hardware.

- Capítulo 6: el contenido de este capítulo desarrolla las contribuciones del trabajo de tesis. Propone el modelo de instrumentación dinámica guiado por eventos y discute los aspectos a tener en cuenta para su implementación.

- Capítulo 7: muestra cómo puede ser implementado el modelo propuesto en la herramienta de detección de violaciones de atomicidad elegida como caso de estudio. Proporciona detalles sobre los aspectos de implementación del modelo y los resultados de los experimentos realizados. Posibilita ver los beneficios alcanzados en relación a la reducción de overhead de la herramienta de detección propuesta como caso de estudio. Se utilizan suites de benchmarks conocidas y aplicaciones reales. En este capítulo se evalúa la efectividad del modelo propuesto en base al overhead y la capacidad de detección.

- Capítulo 8: resume y concluye el trabajo de tesis. Aquí se presenta una evaluación general de los procedimientos utilizados y se comentan las dificultades encontradas. Finalmente se mencionan las lineas de investigación futuras derivadas de esta tesis.

- Apéndices: incluyen material complementario a los capítulos de la tesis.

Capítulo 2

Marco teórico: errores de programación

2.1. Introducción

Este capítulo proporciona una serie de conceptos que permiten abordar el problema de detección de errores de concurrencia en programas paralelos y orienta el resto del trabajo de tesis. Las siguientes secciones constituyen una introducción al proceso de detectar, localizar y corregir errores de software en programas secuenciales (2.2) y se explican las complicaciones introducidas por los programas concurrentes (2.3). Finalmente, se tratan los distintos errores que se pueden manifestar en programas concurrentes (2.4).

2.2. Depuración de programas secuenciales

Un programa secuencial no es más que proceso formado por un conjunto de sentencias que son ejecutadas una después de otra, siguiendo el orden en que aparecen. Normalmente el programador sigue distintas estrategias con la intención de encontrar errores que deben ser removidos de sus programas. En este proceso se diferencian dos etapas bien definidas (Myers, 1984): por un lado la *prueba del software*, que consiste en ejecutar el programa con diferentes casos de prueba en los que se busca exponer la presencia de un error; por el otro, si un error es detectado debe ser corregido. La acción de encontrar y corregir un error se conoce como *depuración del programa*.

Afortunadamente los compiladores actuales tienen la capacidad de detectar una gran cantidad de errores de tipo sintáctico y proveer información sobre su ubicación aproximada en el código. Esta capacidad permite que el programador se concentre en detectar, localizar y corregir errores en la lógica del programa.

Las pruebas del programa se pueden realizar a partir de dos estrategias principales, conocidas como prueba de caja negra y prueba de caja blanca, orientadas a verificar que el programa haga lo que se supone que debe hacer y que está libre de errores.

La prueba de caja negra apunta a encontrar circunstancias en las que el programa no se comporta como debería. Para ello se contrastan los resultados que el programa produce contra los resultados esperados para distintos datos de entrada, sin tener en cuenta el comportamiento y estructura del programa. Dado que sería imposible hacer una prueba exhaustiva del programa, el éxito de esta técnica dependerá de la habilidad del programador para elegir casos de prueba representativos. A diferencia de las pruebas de caja negra, el interés en las pruebas de caja blanca es examinar la estructura interna del programa. La principal ventaja de esta técnica es que permite diseñar casos de prueba a partir del examen de la lógica del programa. Sin embargo, al igual que ocurre con la prueba de caja negra, es imposible realizar una prueba perfecta de caja blanca.

Por lo tanto, la estrategia razonable para probar el software se debe conformar uniendo elementos de ambas estrategias. Se debe desarrollar una prueba razonablemente rigurosa usando ciertas metodologías orientadas hacia la prueba de tipo caja negra y complementar los casos de prueba obtenidos examinando la lógica del programa (es decir, usando los métodos de caja blanca). Si un caso de prueba ha sido exitoso (esto es, ha conseguido evidenciar la presencia de un error) se debe determinar la naturaleza del error, su ubicación dentro del código y corregirlo. Como se mencionó anteriormente, a este proceso se lo conoce como *depuración* o *debugging*.

Existen diferentes técnicas empleadas para llevar a cabo este proceso, desde el volcado de memoria hasta el uso de herramientas automáticas, pasando por el esparcido de sentencias a través de todo el programa. También se usan métodos más formales para llevar a cabo esta tarea, que implican procesos de inducción y deducción. En ciertos casos un método muy efectivo para localizar un error es rastreándolo hacia atrás. También se puede localizar el error a través del refinamiento de los casos de prueba, es decir, haciéndolos más específicos para limitar las posibilidades que pueden hacer fallar el programa.

Sin embargo, cualquiera de estas técnicas serán validadas a través de un proceso de reejecución del programa con el mismo caso de prueba que lo hizo fallar originalmente. Si el error no se vuelve a manifestar, entonces el proceso de depuración ha sido exitoso y se puede confiar en que el error se ha corregido.

2.3. Programas concurrentes

En un programa concurrente coexisten dos o más procesos secuenciales que trabajan en conjunto para resolver el problema más rápido. En general estos procesos necesitan algún mecanismo para *comunicar* resultados parciales entre sí. Esto se consigue a través del uso de *variables compartidas* o del *paso de mensajes* entre procesos (Grama y cols., 2003; Andrews, 2000). Una variable compartida es un espacio de memoria que puede ser accedido por dos o más procesos donde al menos uno de estos escribe algo para que otro lo lea. En cambio el pasaje de mensajes consiste en una técnica en la que de manera explícita uno de los procesos envía un mensaje y otro lo recibe.

Al conjunto de valores en todas las variables del programa en un punto en el tiempo se denomina *estado del programa*. El programa comienza con un *estado inicial* que es alterado por los procesos a medida que ejecutan las sentencias que los conforman. Si la sentencia que se ejecuta examina o cambia el estado del programa de manera que no puede ser interrumpida por otra sentencia, entonces se considera que la sentencia es una *acción atómica*. En general estas acciones son lecturas y escrituras a memoria.

La situación en que una acción atómica de un proceso ocurre entre dos acciones atómicas consecutivas de otro proceso se llama *interleaving*. La secuencia cronológica en que ocurren estos interleavings se conoce como *historia del programa*. Dado que cada proceso se ejecuta independientemente de los otros, se dice que existe un *no determinismo* en la ejecución del programa. Por ello ante dos ejecuciones seguidas del mismo programa, el interleaving que producen los procesos puede variar, generando en cada ejecución una historia diferente. Esto significa que el número de historias que puede producir un programa es enorme, ya que se conforma por todas las combinaciones posibles de interleavings.

Es responsabilidad del programador restringir esas historias solamente a las historias deseables. Para conseguirlo, se emplea *sincronización* de procesos: esto significa que el programador intencionalmente provocará que un proceso espere a que ocurra un evento en otro proceso para poder continuar, lo que permite forzar el orden de ejecución de los procesos. Para que ocurra la sincronización necesariamente deben actuar dos procesos: uno a la espera de un evento y otro que señala al primero la ocurrencia de este evento. La sincronización puede ser por *exclusión mutua* o *sincronización por condición*. La exclusión mutua es una técnica que permite al programador definir un conjunto de sentencias que se comportarán como una acción atómica (es decir, no pueden ocurrir interleavings durante la ejecución de las instrucciones que la componen). Este conjunto de instrucciones se

denomina *sección crítica* o *región atómica*. La sincronización por condición es una técnica donde se retrasa la siguiente sentencia de un proceso hasta que el estado del programa satisface una condición booleana.

Aunque las técnicas y métodos utilizados para detección de errores en programas secuenciales siguen vigentes para los programas concurrentes, el diseño de casos de prueba es insuficiente para detectar los errores de este tipo de programas. El problema es que no hay certeza de que la historia en que se manifestó el error se vuelva a repetir en siguientes ejecuciones: qué historia ocurrirá dependerá del orden de ejecución de los procesos, haciendo que los errores de concurrencia sean particularmente difíciles de detectar.

2.4. Errores de concurrencia

Como se comentó anteriormente, es responsabilidad del programador especificar de qué manera se sincronizarán los procesos del programa. En función del modelo de comunicación empleado, existen diferentes métodos para establecer la sincronización. Por ejemplo en un modelo de memoria compartida es común la utilización de semáforos o monitores y en uno de memoria distribuida se suele utilizar pasaje de mensajes. Los errores de concurrencia aparecen cuando el programador se equivoca en la utilización de alguno de estos métodos.

Por otro lado, el inicio de la llamada *era multicore* impone el desarrollo de software en memoria compartida para obtener el máximo desempeño de las nuevas arquitecturas: de hecho aplicaciones ampliamente usadas como apache, mysql o firefox se encuentran desarrolladas siguiendo este modelo, donde el método de sincronización predilecto es con semáforos. Desafortunadamente, los programas que utilizan este mecanismo de sincronización tienden a contener muchos errores. Por los motivos antes expuestos este trabajo se enfoca en los desafíos relacionados a errores de concurrencia, particularmente en programas de memoria compartida que utilizan semáforos como mecanismo para sincronizar procesos.

En las siguiente secciones se presentan los principales errores de concurrencia: *deadlocks*, *violación de orden*, *violación de atomicidad simple* y *violación de atomicidad multivariable*.

Semáforos Los semáforos fueron inventados por E. W. Dijkstra hace más de 40 años (Dijkstra, 1968) y aún hoy son el método de sincronización más utilizado en aplicaciones paralelas que se ejecutan en arquitecturas de memoria compartida. Un semáforo es una clase especial de variable compartida cuyo valor es por definición un entero no negativo, y que sólo puede ser manipulado a través de las operaciones atómicas P y V. La operación V señala que ocurrió un evento, y para ello incrementa el valor del semáforo. La operación P se usa para decrementar el valor del semáforo. Para asegurar que éste nunca sea negativo, si su valor antes de decrementar es cero la operación no se puede completar y el proceso es detenido: el proceso reanudará su ejecución cuando el valor del semáforo sea positivo, en cuyo caso intentará decrementar el valor del semáforo nuevamente. Salvo que se indique lo contrario, se asume que los semáforos utilizados en los ejemplos son inicializados en 1.

2.4.1. Condiciones de carrera

Una condición de carrera es una situación que se presenta en programas de memoria compartida, donde múltiples procesos intentan acceder a una dirección en memoria y al menos uno de los accesos se trata de una escritura [1]. La Figura 2.1 muestra una interacción entre dos threads donde se puede apreciar una condición de carrera.

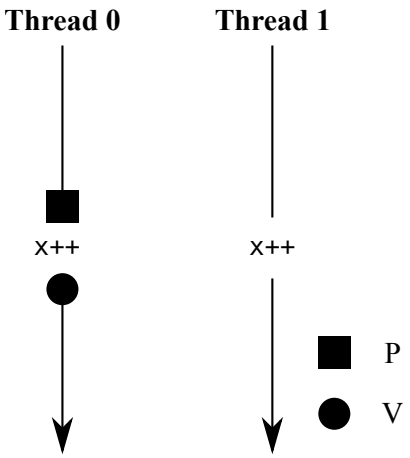


Figura 2.1: Ejemplo de una condición de carrera.

[1] Cuando ambos accesos son lecturas, no se produce modificación al estado del programa, y por lo tanto no hay error. Por lo tanto, aunque estrictamente hablando existe una "condición de carrera" entre dos lecturas, en este trabajo no las consideraremos por no producir errores.

En este ejemplo se puede observar que aunque el *Thread 1* protege adecuadamente el acceso a la variable x, no ocurre lo mismo con el *Thread 2*. Esto provocará que eventualmente ambos threads intenten escribir al mismo tiempo en la variable x.

En general la causa de este error es una sincronización incorrecta donde el programador falló al definir las regiones críticas de los procesos. Debido a que este error sólo se manifiesta para determinadas historias del programa, puede ser muy difícil de detectar. De hecho dependiendo de los casos de prueba, aunque el error ocurra es probable que el programador no se de cuenta de ello. Por este motivo la condición de carrera constituye uno de los errores más tratados en la literatura.

2.4.2. Deadlock

El deadlock es un estado en que dos o más procesos de un programa se encuentran esperando por condiciones que nunca ocurrirán, de manera tal que ninguno consigue finalizar. La figura 2.2 muestra una situación entre dos threads que podría derivar en deadlock.

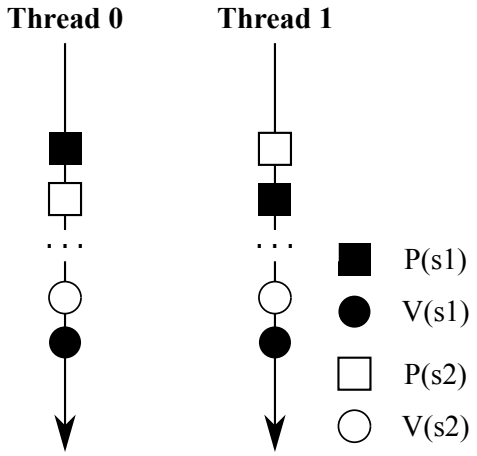


Figura 2.2: Ejemplo de deadlock.

En el ejemplo cada thread hace uso de dos semáforos para proteger su sección crítica. Dependiendo del orden de ejecución de las instrucciones, podría ocurrir que luego de que el *Thread 1* complete la operación P(s1), el *Thread 2* complete la operación P(s2); en este punto, ningún thread podrá completar la siguiente instrucción y en consecuencia quedarán bloqueados.

Debido a la necesidad de compartir recursos entre muchos procesos, los deadlocks son ampliamente estudiados en el ámbito de los sistemas operativos. En cualquiera de estos

textos de estudio se identifican cuatro estrategias comunes para resolver deadlocks: prevenirlos, evitarlos, detectarlos (y resolverlos) o ignorarlos (Silberschatz, Galvin, y Gagne, 2009; Stallings, 2004; Tanenbaum y Woodhull, 2006). Sin embargo, el costo en términos de rendimiento de las tres primeras opciones para la detección de deadlocks provocados por variables compartidas suele ser tan alto, que los diseñadores de los sistemas operativos optan por ignorarlos y dejar en manos del desarrollador de la aplicación la responsabilidad de tratar con éstos.

2.4.3. Violación de orden

Las violaciones de orden ocurren cuando al menos dos accesos a memoria de diferentes procesos suceden temporalmente en un orden inesperado. La Figura 2.3 muestra un ejemplo de esta situación.

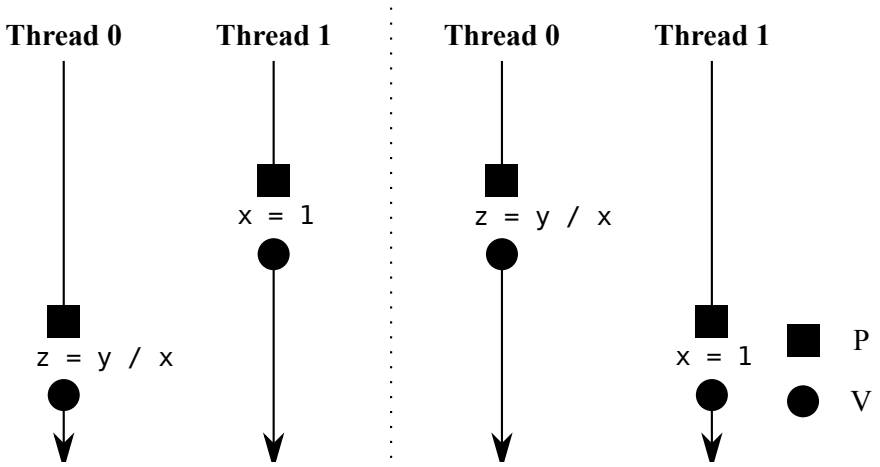


Figura 2.3: Ejemplo de una violación de orden.

El caso de la izquierda representa la intención del programador: el *Thread 1* debe inicializar la variable x antes que el *Thread 0* realice la división. Sin embargo podría ocurrir el caso de la derecha donde el *Thread 0* se adelanta al *Thread 1*, provocando un resultado inesperado.

Esta clase de error es muy frecuente entre los programadores, y como se discutirá al final del capítulo es uno de los más frecuentes. Suelen ocurrir con programadores que están acostumbrados a pensar secuencialmente y asumen un orden determinado entre dos operaciones de diferentes threads, pero olvidan utilizar algún mecanismo de sincronización para forzar ese orden. Luego durante la ejecución, una de las operaciones se ejecuta más

rápido (o más lento) que durante las pruebas realizadas por el programador y el error aparece.

2.4.4. Violación de atomicidad simple

Aún cuando un programa se encuentre libre de los defectos anteriores, puede seguir teniendo errores. Una violación de atomicidad ocurre cuando una secuencia de accesos locales que se espera se ejecuten atómicamente no lo hacen (ocurrió un interleaving no planeado), provocando resultados inesperados en el programa. La Figura 2.4 muestra un ejemplo sencillo donde ocurre esta situación.

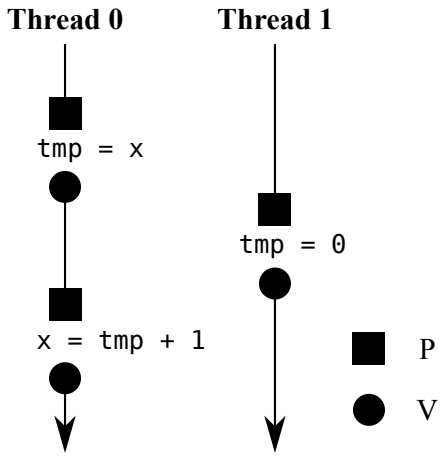


Figura 2.4: Ejemplo de una violación de atomicidad.

Como se puede apreciar, la variable compartida *tmp* se encuentra correctamente protegida en sus secciones críticas, eliminando la posibilidad de una condición de carrera. Tampoco existe la posibilidad de que ocurra un deadlock ya que los threads sólo compiten por acceder a un único recurso compartido que es liberado oportunamente por cada proceso luego de salir de su sección crítica. Sin embargo, el segundo thread puede acceder a un resultado parcial del primero, aunque no fue ésta la intención del programador cuando escribió el código. Una violación de atomicidad ocurre cuando un programador asume que un conjunto de instrucciones serán ejecutado atómicamente y olvida usar un mecanismo para asegurarlo. Esta clase de error es muy común en programas de memoria compartida, y en los últimos años ha adquirido un gran interés su estudio debido a que constituye un caso más general de error que los anteriores.

2.4.5. Violación de atomicidad multivariable

Aunque el caso más común de violaciones de atomicidad ocurre con variables simples (es decir, ambos procesos acceden a la *misma* variable), existe otra clase de error menos común pero más complejo que involucra múltiples variables. La Figura 2.5 muestra un ejemplo de esta situación.

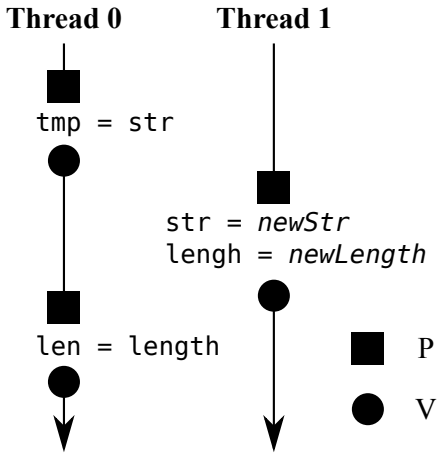


Figura 2.5: Ejemplo de una violación de atomicidad multivariable.

Supóngase que existe una variable *str* que contiene una cadena, y una variable *length* que contiene el número de caracteres de la cadena. El *Thread 0* guarda en una variable temporal *tmp* la cadena *str* y en la variable *len* el valor de *length*. Ambas operaciones se encuentran debidamente protegidas, eliminando la posibilidad de una condición de carrera. Sin embargo, el *Thread 1* modifica los valores de *str* y *length* antes de que el *Thread 0* ingrese a la sección crítica de la variable *len*. Se debe notar que en esta situación, el *Thread 0* tendrá un valor para *len* inconsistente con la cadena guardada en *tmp*. Este error es interesante porque si se analizan las variables compartidas str y length individualmente, no existe violación de atomicidad: es necesario considerar ambas variables como una unidad para que el error sea evidente.

2.4.6. ¿Qué tan frecuentes son estos errores?

Cualquiera de los errores antes mencionados se puede encontrar con frecuencia en programas reales[2].

[2]La expresión *programas reales* se utiliza para diferenciar los programas que están diseñados para evaluar alguna característica de un sistema o herramienta de los programas que son utilizados realmente por usuarios.

De todos los errores, las condiciones de carrera y los deadlocks son los que han gozado de más popularidad en la comunidad científica durante muchos años.

Sin embargo, sólo el 29,5 % de los errores reportados en aplicaciones ampliamente utilizadas son deadlocks(Lu, Park, Seo, y Zhou, 2008). El estudio que reveló esta información consistió en analizar 105 errores de concurrencia conocidos en aplicaciones ampliamente utilizadas (Apache, MySQL, Mozilla Firefox y OpenOffice). Es de interés destacar que el 70,5 % de los errores no son deadlocks. Además, de ellos:

- Cerca del 96 % de los errores evaluados ocurren entre dos threads: esto indica que la mayoría de los errores se pueden detectar estudiando las interacciones entre dos threads, lo que reduce notablemente la complejidad de los algoritmos de detección.

- Aproximadamente un 65 % son violaciones de atomicidad, un 32 % son violaciones de orden, y un 3 % son errores de otra clase: se debe notar que *las violaciones de atomicidad constituyen el caso más general de error de concurrencia.*

- El 66 % de los errores involucran una única variable: enfocarse en accesos concurrentes a una misma variable es una buena simplificación, ya que contempla a la mayoría de los errores.

2.5. Resumen

Los métodos tradicionales para detección de errores en programas secuenciales no son adecuados para la depuración de programas concurrentes, principalmente por el no determinismo implícito en la ejecución de esta clase de programas. El no determinismo implica que dos ejecuciones seguidas pueden tener historias de ejecución distintas, y por lo tanto no hay garantías que un determinado error vuelva a ocurrir.

Las suposiciones de orden de ejecución entre instrucciones y atomicidad en el acceso a las variables heredadas del modelo de programación secuencial ya no son válidas, obligando al programador a utilizar algún mecanismo de sincronización para asegurar esas propiedades. El modelo de comunicación entre procesos paralelos más eficiente para procesadores multicore es el de memoria compartida. El mecanismo de sincronización por excelencia en software para arquitecturas multicore son los semáforos. Muchas aplicaciones ampliamente utilizadas como apache, mysql o firefox se encuentran desarrolladas siguiendo este modelo.

Sin embargo, frecuentemente el programador se equivoca al sincronizar los procesos, dando lugar a nuevos errores de programación como son los deadlocks, condiciones de carrera, violaciones de orden, violaciones de atomicidad simple y violaciones de atomicidad multivariable. Los métodos tradicionales de depuración de programas no son aplicables en el contexto de los programas concurrentes, por lo que es necesario disponer de herramientas de depuración que puedan ayudar al programador a detectar esta clase de errores.

De estos errores, los deadlocks y las condiciones de carrera han gozado de mayor popularidad en la comunidad científica. Sin embargo, solo el 29,5 % de los errores son deadlocks: del 70,5 % restante, *las violaciones de atomicidad representan más del 65 % de los errores*, el *96 % ocurren entre dos threads* y *el 66 % involucran una sola variable*. Las violaciones de atomicidad simple se han definido en los últimos años como el caso más general de error de concurrencia y han recibido gran atención por numerosos grupos de investigación. Por los motivos antes expuestos, el resto del trabajo se enfocará en técnicas de detección de violaciones de atomicidad simples.

Capítulo 3

Caso de estudio: detección de violaciones de atomicidad

3.1. Introducción

Este capítulo presenta los antecedentes sobre técnicas de detección de violaciones de atomicidad en la comunidad científica (3.2). Luego se presenta el método de detección llamado análisis de interleavings (3.3), por ser el método con mejor capacidad de detección y desempeño que puede ser implementado completamente en software. Finalmente, se proporcionan detalles sobre la implementación de este método siguiendo las especificaciones del trabajo donde fue propuesto (3.4) y se valida su desempeño con respecto a esa propuesta (3.5).

3.2. Antecedentes

Las primeras propuestas para detección de violaciones de atomicidad profundizaban en métodos estáticos que requerían anotaciones en el código para detectar las violaciones (Flanagan y Qadeer, 2003; Sasturkar, Agarwal, Wang, y Stoller, 2005; Wang y Stoller, 2005), volviendo el proceso de depuración costoso y complejo. Sin embargo, a lo largo de de estos años, varias propuestas han conseguido muy buenos resultados con estos métodos:

Atomizer(Flanagan y Freund, 2004) evita las anotaciones en el código valiéndose del algoritmo lockset para encontrar regiones atómicas y utiliza heurísticas para validar su atomicidad. Sin embargo, lockset(Savage, Burrows, Nelson, Sobalvarro, y Anderson, 1997) es una estrategia demasiado conservativa: al depender de las sentencias de sincronización

para determinar atomicidad en el acceso a las variables, situaciones donde la exclusión mutua sea garantizada por la sincronización entre procesos en lugar de pares de instrucciones lock/unlock serán informados como bugs.

SVD(Xu, Bodík, y Hill, 2005) es uno de las primeras propuestas en utilizar un método dinámico que no requiere anotaciones en el código: utiliza heurísticas para determinar las regiones de código que deben ser atómicas, y luego verifica que sean ejecutadas serialmente. Analiza patrones de dependencia read-write entre regiones atómicas, por lo que no puede detectar bugs que impliquen dependencias de tipo write-write o write-read.

AVIO(Lu y cols., 2006) propone el método de análisis de interleavings, lo que permite detectar otros casos de violaciones de atomicidad no contemplados por las propuestas anteriores. Fue uno de los primeros en inferir la atomicidad del código a partir de observar invariantes en ejecuciones de entrenamiento, sin conocimiento previo sobre las primitivas de sincronización. Aunque comparado con otras propuestas consigue reducir notablemente el overhead, los autores reconocen que aún es muy elevado para ser usado en entornos en producción.

MUVI(Lu y cols., 2008) analiza el caso de accesos correlacionados entre múltiples variables, y propone extensiones hardware para detectores de carreras que usen los algoritmos lockset y happens-before. Es de los primeros trabajos dedicados íntegramente a la detección de errores que involucran multivariables.

AtomFuzzer(C.-S. Park y Sen, 2008) altera el comportamiento del scheduler de Java para construir escenarios donde se viole la atomicidad de una región atómica. Identifica las regiones atómicas a partir de anotaciones en el código, pero complementa esta técnica con heurísticas similares a las empleadas por Atomizer. Esta técnica se limita a un caso particular de violación de atomicidad donde un mismo thread adquiere y libera un mismo lock en dos ocasiones dentro de la misma función, y es intervenido por una operación lock/unlock de otro thread. Al detectar el primer lock/unlock de un thread, lo demora a la espera de que otro thread intente adquirir el lock. Si esto ocurre, entonces advierte una violación de atomicidad. Los autores argumentan que este es el caso más común de violación de atomicidad en programas Java, y que el resto de violaciones se deben a condiciones de carrera, por lo que pueden ser detectadas por técnicas tradicionales.

Atom-aid(Lucia, Devietti, Strauss, y Ceze, 2008) propone agrupar instrucciones para crear bloques atómicos, lo que reduce la posibilidad e ocurrencia de interleavings no probados. Esta propuesta utiliza un procesador especial para ejecutar esos bloques. Sin

embargo, su capacidad de detección está sujeta al tamaño del bloque, lo que podría provocar que determinadas violaciones de atomicidad no sean detectadas.

ColorSafe(Lucia y Ceze, 2009) mediante anotaciones agrupa variables que están correlacionadas en conjuntos (o colores) e integra el concepto de análisis de interleavings de AVIO con la técnica de Atom-aid para evitar la ocurrencia de interleavings no probados por hardware. Al tratar como unidad a cada conjunto de variables, es capaz de detectar violaciones de atomicidad sobre variables simples y multivariables.

PSet(Yu y Narayanasamy, 2009) rescata la idea de que los interleavings que provocan bugs son poco frecuentes (en caso contrario habrían sido detectados por una buena prueba del software) y propone un sistema de ejecución que fuerza al programa a tomar los interleavings probados en ejecuciones de entrenamiento: de esta manera aleja la posibilidad de que ocurra un error oculto en algún interleaving no probado. Para ello adhiere a cada acceso a memoria de un thread un conjunto de restricciones de acceso de otros threads, recolectados durante el entrenamiento. Los autores comentan que aunque implementaron una versión software de su algoritmo, el overhead alcanzaba más de 200× para aplicaciones intensivas como SPLASH-2(Woo, Ohara, Torrie, Singh, y Gupta, 1995) y PARSEC(Bienia, 2011).

Bugaboo(Lucia y Ceze, 2009) utiliza ejecuciones de entrenamiento para configurar grafos, los que son extendidos con información contextual sobre las comunicaciones entre procesos. La información que proponen incorporar a los grafos permite detectar violaciones de orden y errores que involucran múltiples variables además de violaciones de atomicidad y condiciones de carrera.

AtomTracker(Muzahid, Otsuki, y Torrellas, 2010) propone un método para inferir automáticamente regiones que deben ser atómicas. Para ello se vale de un conjunto de ejecuciones de entrenamiento: cada ejecución genera un archivo de trazas que contiene las regiones atómicas detectadas; la siguiente ejecución parte del archivo generado, y al combinarlo con la información de la ejecución actual probablemente deba romper algunas regiones atómicas en regiones más pequeñas; el proceso continúa hasta que el número de regiones atómicas no cambia entre ejecuciones. El proceso de detección toma el archivo generado y verifica si dos regiones atómicas que deben ejecutarse concurrentemente se pueden ejecutar en secuencia, en cualquiera de los dos órdenes posibles: caso contrario declara una violación de atomicidad.

Métodos para forzar errores Aunque no son técnicas de detección en si mismas, existen propuestas que complementan algunas de las técnicas anteriores con métodos para forzar que los errores se manifiesten.

CTrigger(Lu, Park, y Zhou, 2012; S. Park, Lu, y Zhou, 2009) apunta directamente a la etapa de prueba del software: las herramientas de detección no pueden reportar un bug hasta que este se manifieste en una ejecución. Por este motivo, técnicas de *stress testing* son demasiado costosas y poco eficientes para encontrar errores de concurrencia. El método propuesto analiza la ejecución del programa y selecciona interleavings representativos (aquellos que son poco probables de ocurrir). Luego altera la ejecución del programa para forzarlos a que ocurran.

CFP(Deng, Zhang, y Lu, 2013) (continuando con las ideas de CTrigger) reconoce que las técnicas anteriores son demasiado costosas para ser aplicadas en la etapa de prueba del software. En el contexto de errores de concurrencia, muchos bugs pueden ser expuestos por diferentes casos de prueba, lo que da como resultado que el esfuerzo de detección en muchos casos es malgastado. Por lo anterior proponen un método para guiar la selección de pruebas a realizar con el objeto de acelerar la detección de errores a partir de la reducción del esfuerzo redundante de detección a través de las pruebas.

Las ideas expuestas en estos trabajos claramente contribuyen a la detección de errores. Aunque esta tesis se enfoca en reducir el overhead de métodos de detección, se considera importante mencionarlas en el contexto de estudio.

Herramientas de detección

Aunque hay muchos trabajos sobre detección de errores de concurrencia, hay pocas herramientas que los implementan.

Se destacan Intel Inspector XE (Banerjee, Bliss, Ma, y Petersen, 2006a, 2006b), HELGRIND (Jannesari, Bao, Pankratius, y Tichy, 2009), Google ThreadSanitizer (Serebryany y Iskhodzhanov, 2009), IBM multicore SDK (Qi, Das, Luo, y Trotter, 2009) y Oracle Thread Analyzer (Terboven y cols., s.f.). Sorprendentemente, en todos los casos están diseñados sólo para la detección de condiciones de carrera. Estos productos implementan técnicas que validan la relación happens-before de Lamport (Lamport, 1978) o implementan el algoritmo lockset (Savage y cols., 1997) (en algunos casos combinan ambas

técnicas). El Apéndice A (Detección de condiciones de carrera) ofrece una explicación sobre estos algoritmos y cómo funcionan.

En este trabajo se decidió utilizar AVIO por ser la propuesta con mejor capacidad de detección y menor overhead que, por poseer una versión completamente software, puede ser ejecutada en las plataformas actualmente en producción. Debido a que no existe una versión de AVIO disponible para su descarga e instalación, en este trabajo se decidió implementar una versión acorde a las especificaciones del artículo original. Además, esta versión servirá de punto de partida para analizar las propuestas del presente trabajo de tesis.

3.3. Método de análisis de interleavings

AVIO(Lu y cols., 2006) fue la primera herramienta de detección de errores que propuso analizar en tiempo de ejecución los interleavings que generaban los threads de la aplicación en búsqueda de violaciones de atomicidad. Para ello el método propone clasificar los interleavings en serializables y no serializables: un interleaving es serializable si la ocurrencia del acceso remoto antes o después que los accesos locales no altera el resultado de su ejecución.

Dependiendo del tipo de cada acceso del interleaving (lectura o escritura) se pueden configurar ocho casos diferentes, resumidos en la Tabla 3.1. Los accesos locales se encuentran alineados a la izquierda, mientras que el acceso remoto se encuentra desplazado a la derecha.

Cuadro 3.1: cada caso muestra una configuración diferente de interleaving. Los accesos corresponden a operaciones sobre una misma dirección de memoria entre dos threads. Los interleavings no serializables están resaltados en gris.

	thr_0	thr_1		thr_0	thr_1		thr_0	thr_1		thr_0	thr_1
	read			read			read			read	
0		read	2		write	4		read	6		write
	read			read			write			write	
	write			write			write			write	
1		read	3		write	5		read	7		write
	read			read			write			write	

Los interleavings 0, 1, 4 y 7 son serializables, y por lo tanto no representan riesgo de violaciones de atomicidad. En cambio los interleavings 2, 3, 5 y 6 se deben analizar para

determinar si se deben a un bug o no. Por ejemplo el interleaving de la Figura 2.4 es un bug del caso 3. Sin embargo, se debe notar que a veces un interleaving no serializable se puede deber a la intención del programador. Para aclarar esta idea observe el ejemplo de la Figura 3.1.

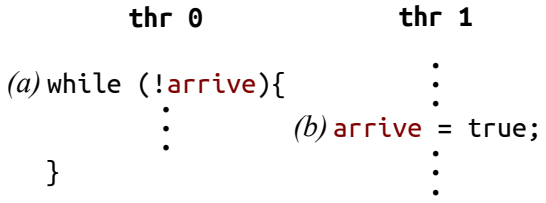


Figura 3.1: Ejemplo de un interleaving no serializable de caso 2 intencional. El interleaving se debe a una sincronización entre procesos.

Se trata de un interleaving de caso 2 debido a que la escritura remota ocurre entre dos lecturas locales. Sin embargo, el programador intencionalmente provocó el interleaving para sincronizar los threads. Los interleavings no serializables como el anterior se denominan *invariantes*: estos se diferencian de los bugs en que suelen ocurrir en todas las ejecuciones.

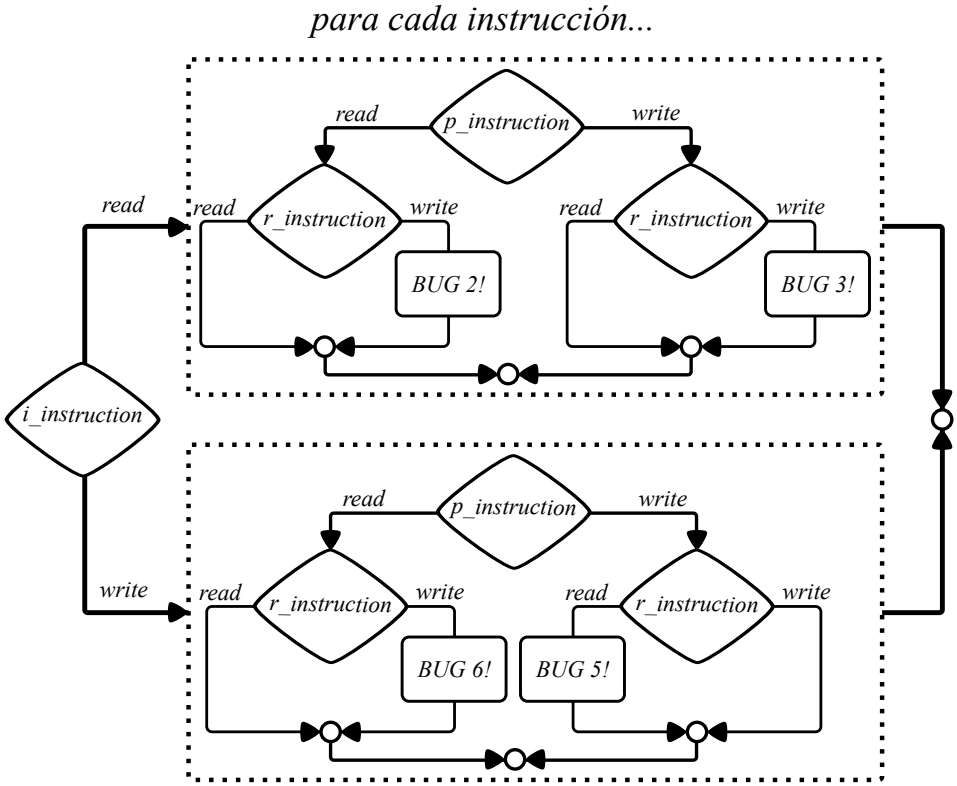
Aprovechando esta característica propia de los invariantes, AVIO emplea una etapa de entrenamiento llamada *extracción de invariantes*. Esta etapa consiste en ejecutar varias veces la aplicación a través de AVIO. Todos los interleavings no serializables detectados serán guardados en un registro de invariantes. Si durante esta etapa uno de estos interleavings no se repite, entonces se lo quita del registro de invariantes.

Dado que existe la posibilidad que un invariante no ocurra en todas las ejecuciones, se puede flexibilizar la restricción anterior: para ello se puede definir un *umbral* de ocurrencias que debe superar un interleaving en el entrenamiento antes de ser clasificado como invariante. Realizar pocas ejecuciones de entrenamiento podría derivar en que AVIO detecte más falsos positivos. Por otro lado muchas ejecuciones de entrenamiento podría derivar en que AVIO marque como invariantes interleavings que no lo son. Tanto el número de ejecuciones de entrenamiento como el umbral se pueden ajustar a cada aplicación. Luego de completar la etapa de entrenamiento, AVIO funciona en modo de detección. En este modo sólo informará como bugs los interleavings no serializables que no se encuentren en el registro de invariantes.

3.4. Implementación de AVIO

El algoritmo utiliza instrumentación a nivel de instrucción para incluir una llamada a rutinas de análisis por cada lectura o escritura que realiza el programa objetivo. La Figura 3.2 resume en un diagrama de flujo el funcionamiento de AVIO.

Figura 3.2: Diagrama de flujo para la rutina de análisis implementada por AVIO.



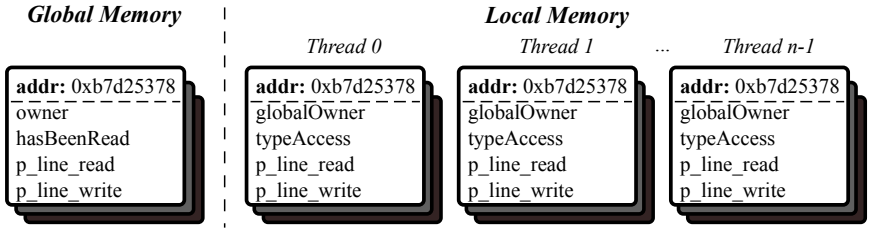
Para determinar la ocurrencia de interleavings y su posterior clasificación en serializables o no serializables, es necesario llevar la historia de accesos a cada dirección de memoria. Debido a que un interleaving está compuesto por dos accesos del mismo thread entrelazados con un acceso de un thread remoto, cada thread solo necesita registrar los últimos dos accesos a cada dirección de memoria: estos accesos corresponden a la vista *local* del interleaving.

Para registrar el acceso entrelazado es necesario tener una vista *global* de la historia de cada dirección de memoria. Cada dirección de memoria puede ser entrelazada con una lectura o una escritura, pero hasta que no se complete el interleaving no se puede saber

cuál será. Por este motivo, la vista global debe registrar la última lectura y escritura e indicar qué thread realizó el acceso.

Para facilitar el acceso a la historia de cada dirección de memoria, se utilizaron tablas de hash para las estructuras de datos. Cada thread tendrá una tabla para sus accesos locales y todos compartirán una tabla global para calcular los accesos remotos. La Figura 3.3 muestra un modelo de las estructuras de datos que soportan el algortimo.

Figura 3.3: Estructuras de datos para soportar el algoritmo de AVIO.



La estructura *Global Memory* se implementa como una tabla de hash indexada por la dirección de memoria a la que acceden los procesos. Esta tabla combinada con *Local Memory* sirve para seguir el rastro de los accesos remotos. Cada dirección posee cuatro propiedades:

- *owner*: un número entero único (id) que identifica al último thread que escribió en esa dirección de memoria.
- *hasBeenRead*: un valor lógico que indica si la dirección de memoria ha sido leída por un thread diferente a *owner* (por defecto vale falso).
- *p_line_read*: número de la última instrucción de lectura sobre la dirección de memoria.
- *p_line_write*: número de la última instrucción de escritura sobre la dirección de memoria.

Además, cada thread posee una tabla de hash denominada *Local Memory*, también indexada por la dirección de memoria a la que acceden los procesos. Esta tabla sirve para seguir el rastro de los accesos locales de cada thread. Cada dirección posee cuatro propiedades:

- *globalOwner*: contiene el valor de *owner* en *Global Memory* al momento de registrar el acceso.
- *typeAccess*: indica si el acceso realizado por el thread es una lectura, una escritura (por defecto vale indefinido).
- *p_line_read*: número de la última instrucción de lectura sobre la dirección de memoria.
- *p_line_write*: número de la última instrucción de escritura sobre la dirección de memoria.

Cada vez que la línea sea ***leída*** se registrará el número de instrucción que lee en las propiedades *p_line_read* de la *Local Memory* del thread que está accediendo y en la *Global Memory*; además, si el id del thread que lee es distinto al *owner*, entonces se asinga el valor verdadero a la variable *hasBeenRead*. Cada vez que la línea sea ***escrita***, se registrará el id del thread que escribe en *owner*, se reinicializará *hasBeenRead* a falso y se registrará el número de instrucción que escribe en las propiedades *p_line_write* de *Global Memory* y de la *Local Memory* del thread que está accediendo.

Sea *I-instruction* la instrucción de acceso que AVIO está evaluando. Para que exista un interleaving se deben haber ejecutado las siguientes instrucciones especiales:

- *P-instruction*. Una instrucción de acceso del mismo thread previa a la misma dirección de memoria.
- *R-instruction*. Una instrucción de acceso de un thread diferente a la misma dirección de memoria, que ocurrió luego de *P-instruction*.

Estas condiciones se verifican consultando las tablas de hash para la dirección de memoria accedida. Si es la primera vez que se accede a esa dirección de memoria, entonces el valor para *typeAccess* en la tabla *Local Memory* sera indefinido, lo que significa que no se ha registrado aún una *P-instruction*: en este caso el algoritmo solamente registrará en las tablas *Global Memory* y *Local Memory* los datos del acceso.

Por el contrario, si *typeAccess* en *Local Memory* tiene un valor definido, se debe averiguar si se ejecutó una *R-instruction*. Para ello debe consultar la estructura *Global Memory*.

En adelante se asume que el algoritmo ya ha registrado una *P-instruction* y que se encuentra analizando una *I-instruction*. Tal como se mencionó en el cuadro 3.1, existen cuatro interleavings que deben ser detectados:

- Caso 2 ($R_P \rightarrow W_R \rightarrow R_I$)
- Caso 3 ($W_P \rightarrow W_R \rightarrow R_I$)
- Caso 5 ($W_P \rightarrow R_R \rightarrow W_I$)
- Caso 6 ($R_P \rightarrow W_R \rightarrow W_I$)

En todos los casos el tipo de la *P-instruction* se obtiene de la propiedad *typeAccess* de la tabla *Local Memory* y el tipo de la *I-instruction* se deduce del acceso que se está evaluando.

Para determinar si hubo un acceso remoto se consultan las propiedades *hasBeenRead* y *owner* de *Global Memory*. En los casos 2, 3 y 6 se debe detectar si ocurrió una escritura remota: esto es, si el valor de *globalOwner* en la *Local Memory* del thread es distinto al valor de *owner* en *Global Memory*. Se debe recordar que *globalOwner* registra el valor que tenía *owner* al momento de ser ejecutada la *P-instruction*, por lo tanto si *owner* cambió desde entonces, significa que un *thread remoto* escribió en esa dirección de memoria.

En el caso 5 se debe detectar si ocurrió una lectura remota: esto es, si el valor de *hasBeenRead* es verdadero. Se debe recordar que esta propiedad sólo será verdadera si ocurre un acceso de lectura por un thread diferente al que la escribió por última vez.

Si el algoritmo determina que ocurrió un interleaving de los casos mencionados, entonces se registra un posible bug compuesto por las instrucciones almacenadas en las propiedades *p_line_read* y *p_line_write* de ambas estructuras de datos.

3.5. Resultados experimentales del capítulo

Esta sección proporciona los resultados de evaluar AVIO a partir de su capacidad de detección de errores y rendimiento. La versión utilizada de AVIO corresponde a la implementación descrita en la sección anterior. Para comparar AVIO se utilizó HELGRIND (Jannesari y cols., 2009), una herramienta comercial ampliamente utilizada para detección de condiciones de carrera. HELGRIND viene incluido con VALGRIND, y se instala a partir del administrador de paquetes de Debian. La versión más actual instalable desde los repositorios es la 3.8.1.

3.5.1. Plataforma de experimentación

El sistema operativo es un Debian wheezysid x86_64 GNULinux con kernel 3.2.0. El compilador empleado fue gcc en la versión 4.7.0. Salvo que se indique lo contrario, la arquitectura objetivo de la compilación es la x86_64 y los flags de optimización estandar -O2. Los experimentos se realizaron en una máquina con dos procesadores Xeon X5670, cada uno con 6 núcleos con HT. En esta plataforma, los procesadores numerados del 0 5 corresponden a los núcleos físicos del primer procesador y los procesadores numerados del 6 al 11 corresponden a los núcleos físicos del segundo procesador. Gracias al hiperthreading, se cuenta además con los procesadores numerados del 12 al 17 del primer procesador y del 18 al 23 del segundo procesador. Sin embargo, los experimentos están diseñados para ser ejecutados sólo con 2 threads, ya que como se comentó al final del Capítulo 2, la mayoría de los errores de concurrencia involucran esta cantidad de procesos. No obstante, para asegurar que los experimentos siempre se ejecutarán en procesadores físicos distintos se evitaron configuraciones con más de 12 threads, y la afinidad de cada thread se fijó acorde a su número, garantizando que los threads serán ejecutados en los procesadores identificados del 0 al 11. La microarquitectura de estos procesadores es Westmere. La frecuencia de cómputo es de 2.93 GHz. Cada procesador tiene tres niveles de cache – L1 (32KB) y L2(256KB) privadas de cada núcleo y L3(12MB) compartida entre todos los núcleos del mismo procesador (*Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2012).

3.5.2. Benchmarks

Para evaluar la capacidad de detección de las herramientas se diseñaron cuatro kernels de ejecución, uno para cada caso de interleaving *no serializable* conocido. La Tabla 3.2 resume cada kernel y el caso de interleaving para el que fue diseñado.

Cuadro 3.2: Kernels diseñados para exponer los casos de interleavings conocidos.

kernel	interleaving
APACHE	Case 2
MOZILLA	Case 3
MYSQL	Case 5
BANKACCOUNT	Case 6

Para evaluar el desempeño de las herramientas propuestas se utilizó la suite de benchmarks SPLASH-2 (Woo y cols., 1995). SPLASH-2 es probablemente la suite más utilizada para estudios científicos en máquinas paralelas con memoria compartida. Esta suite fue

empleada en el artículo original de AVIO y en otros trabajos relacionados (Yu y Narayanasamy, 2009; Muzahid y cols., 2010; S. Park y cols., 2009; Deng y cols., 2013).

Los detalles de diseño de estos kernels y la configuración empleada en los programas que componen la suite SPLASH-2 se pueden consultar en el Anexo B.

3.5.3. Capacidad de Detección de Errores

En este experimento se ejecutaron los kernels de la Sección B.1.1 a través de AVIO y HELGRIND. Debido a que los kernels poseen un único error, sólo hay dos resultados posibles: la herramienta detectó el error o no lo hizo. La Tabla 3.3 resume los resultados indicando para cada kernel y cada herramienta si fue capaz de detectar el bug simulado.

Cuadro 3.3: Resultados de detección para diferentes herramientas usando kernels con errores de concurrencia.

Kernel	AVIO	HELGRIND
APACHE	Si	Si
MOZILLA	Si	No
MYSQL	Si	Si
BANKACCOUNT	Si	No

Tal como se esperaba, AVIO fue capaz de detectar los cuatro casos conflictivos. En el caso de HELGRIND, sólo detectó los bugs de APACHE y MYSQL. Se debe tener en cuenta que esta herramienta está diseñada para detectar condiciones de carrera analizando la relación *happens-before* (Apéndice A) entre los accesos a datos compartidos.

Los kernels APACHE y MYSQL son extractos de las versiones oficiales con bugs, y en ambos casos el error es que el programador omitió proteger los accesos a memoria. Por este motivo, los interleavings se pueden manifestar como pares de condiciones de carrera. Por ejemplo, la detección del bug de APACHE fue informada por HELGRIND como el resultado de dos condiciones de carrera: una carrera $R_L \rightarrow W_R$ entre el primer acceso del *Thread 0* con el acceso del *Thread 1*, y otra carrera $W_R \rightarrow R_L$ entre el *Thread 1* con el segundo acceso del *Thread 0*.

Por otro lado, los kernels MOZILLA y BANKACCOUNT se desarrollaron libres de condiciones de carrera. A diferencia de los casos anteriores, en estos kernels los accesos fueron protegidos con locks, pero hay un error en la granularidad de las secciones críticas.

Para aclarar este punto se puede observar la Figura 3.4 donde se puede notar que los accesos a la variable compartida *gCurrentScript* se encuentran protegidos por locks.

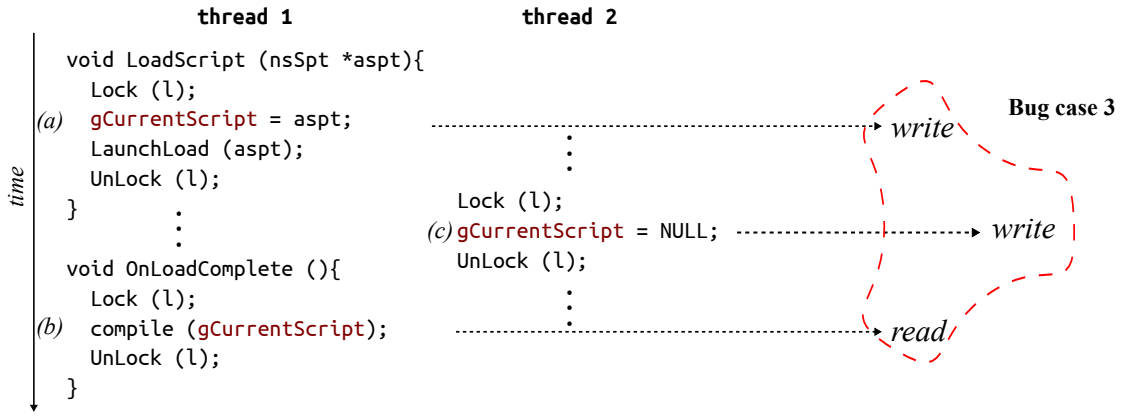


Figura 3.4: Ejemplo de una violación de atomicidad en Mozilla.

Esto evita que ocurran condiciones de carrera entre los threads, pero aún así podría ocurrir una violación de atomicidad si el *Thread 1* alcanza el lock luego de que el *Thread 0* abandone la función *LoadScript()* pero antes que ejecute la función *OnLoadComplete()*. Esta clase de errores no pueden ser detectados por las herramientas que analizan la relación happens-before.

3.5.4. Análisis de Rendimiento

Se ejecutaron los paquetes de la suite de benchmarks SPLASH-2 a través de ambas herramientas. Para cada aplicación se tomó en cuenta el tiempo que se demora en ejecutar a través de cada herramienta. La tabla 3.4 resume los resultados obtenidos.

Cuadro 3.4: Comparación de tiempos de ejecución entre AVIO y HELGRIND. Como referencia, se incluye también el tiempo que demora cada aplicación a través de el framework de instrumentación dinámica PIN.

	PIN		**AVIO**		**HELGRIND**	
	tiempo	***DevNorm***	***tiempo***	***DevNorm***	***tiempo***	***DevNorm***
barnes	1,53	0,48	129,2	3,22	1752,22	775,67
cholesky	1,05	0,01	27,24	0,34	352,78	133,78
fft	0,72	0,01	2,75	0,02	1,37	0,01
fmm	1,25	0,01	84,63	5,08	1378,93	794,42
lu_cb	0,72	0,01	25,86	0,41	4,63	0,02
lu_ncb	0,67	0,01	1,07	0,01	0,54	0,01
ocean_cp	1,19	0,01	44,05	0,43	41,82	2,49
ocean_ncp	1,11	0,01	42,2	0,44	45,03	0,05
radiosity	1,35	0,01	64,96	1,07	182,25	113,77
radix	0,66	0,01	3,78	0,04	1,4	0,01
raytrace	1,34	0,01	46,99	0,56	72,99	1,74
volrend	1,16	0,06	20,92	0,24	287,08	210,77
water_nsq	0,89	0,01	25,19	0,29	3,69	0,09
water_sp	0,93	0,01	21,97	0,22	3,3	0,01

Para cada experimento se realizaron 10 ejecuciones. Los valores de la tabla corresponden a los promedios de tiempo (en segundos) que demoraron en completar la tarea. Las columnas destacadas en gris corresponden a la desviación estándar de la muestra. Cabe destacar que AVIO en todos los casos se comportó con gran estabilidad entre ejecuciones, mientras que HELGRIND en varios casos (barnes, cholesky, fmm, radiosity y volrend) tuvo grandes variaciones. Esto se puede explicar a partir del funcionamiento de cada algoritmo: AVIO lleva un registro de *todos* los accesos a memoria que realiza el programa en búsqueda de interleavings no serializables; esto se traduce en que el overhead de AVIO no depende del no determinismo (que es una propiedad de la ejecución) sino del número total de accesos a memoria que tiene el código (una propiedad del programa). Por otro lado el overhead de HELGRIND esta fuertemente relacionado al no determinismo de la ejecución: las estructuras de datos y los algoritmos que emplea[1] dependen de la ocurrencia de condiciones de carrera (sin discriminar entre falsos positivos o condiciones reales), y por lo tanto del orden en que se ejecuten los threads.

Otra manera de analizar los resultados consiste en expresar el overhead que introduce cada algoritmo como la relación entre el tiempo de ejecución de cada herramienta y el tiempo de ejecución de la aplicación sin instrumentación. La Tabla 3.5 muestra la relación de overhead para cada herramienta.

Cuadro 3.5: Overhead de las herramientas con respecto a la ejecución de los paquetes de la suite SPLASH-2 a través del framework PIN ([AVIO|HELGRIND] / PIN). Para facilitar la interpretación de los datos los resultados se redondearon para eliminar los decimales.

	AVIO	**HELGRIND**
barnes	84×	1142×
cholesky	26×	335×
fft	4×	2×
fmm	68×	1108×
lu_cb	36×	6×
lu_ncb	2×	1×
ocean_cp	37×	35×
ocean_ncp	38×	41×
radiosity	48×	135×
radix	6×	2×
raytrace	35×	54×
volrend	18×	248×
water_nsq	28×	4×
water_sp	24×	4×
Promedio 1	32×	223×
Promedio 2	23×	224×

[1]El Anexo A proporciona una explicación de estos algoritmos

Se utilizó como denominador de la relación el tiempo de cada paquete ejecutado a través del framework PIN con la intención de excluir de los resultados el overhead producto de la instrumentación. Para facilitar la interpretación de los datos, los resultados se redondearon a su parte entera.

En promedio, AVIO provocó un overhead de 32×, mientras que HELGRIND alcanzó un valor de 223×. Cabe destacar que las aplicaciones que demostraron una gran variación en sus resultados para HELGRIND (barnes, cholesky, fmm, radiosity y volrend) poseen un factor de overhead notablemente superior al de AVIO.

En un segundo conjunto de casos (ocean_cp, ocean_ncp y raytrace) no hay diferencias significativas entre ambas herramientas.

También se debe notar que para las aplicaciones fft, lu_cb, lu_ncb, radix, water_nsq y water_sp HELGRIND tuvo mejores resultados que AVIO. Eso se debe al tipo de sincronización empleado en esos benchamrks: la mayoría de sus estructuras de sincronización son barreras. Dado que una barrera es un punto de sincronización entre todos los procesos, cada vez que el algoritmo happens-before se encuentra con una, puede desechar la información recolectada hasta ese momento sobre la ejecución. Esta situación beneficia significativamente a HELGRIND con respecto a AVIO, ya que como se dijo anteriormente el overhead de AVIO depende directamente del número de accesos a memoria del programa.

Se quiere destacar que el artículo original de AVIO reporta un overhead promedio de 25× para las aplicaciones de la suite SPLASH-2. Sin embargo, en ese trabajo sólo se publicaron los resultados para las aplicaciones fft, fmm, lu y radix. La Tabla 3.5 destaca en gris las filas correspondientes a esas aplicaciones. Para calcular el promedio 2 de la tabla sólo se tomaron en cuenta los overheads de estos paquetes. Como se puede observar, la implementación del algoritmo de AVIO en este trabajo reporta un overhead promedio de 23× para esas aplicaciones, lo cual coincide con los resultados reportados en el artículo original.

3.6. Resumen

Durante décadas la comunidad científica ha estudiado problemas de la concurrencia como deadlocks o condiciones de carrera y propuesto métodos para su detección. A principios de la década pasada, con la llegada de la llamada *era multicore*, el modelo de programación en memoria compartida se impone sobre el modelo de programación

secuencial, cobrando especial relevancia los problemas de concurrencia. Así, las primeras propuestas para herramientas de detección de violaciones de atomicidad comienzan a aparecer a finales de los 90. Estas propuestas usaban métodos estáticos de análisis, lo que se traducía en un pobre desempeño y limitada capacidad de detección.

No fue hasta 2005 que aparecieron las primeras propuestas que utilizan métodos dinámicos para la detección de violaciones de atomicidad, mejorando notablemente la capacidad de detección. De estas propuestas, AVIO(Lu y cols., 2006) se destaca como la propuesta con mejor rendimiento y capacidad de detección que puede ser implementada completamente en software. Esta propuesta implementa el método de análisis de interleavings: un interleaving es un conjunto de dos accesos de un proceso a la misma dirección de memoria, pero donde justo en medio de ellos ocurre un acceso a la misma dirección de otro proceso. El método clasifica los interleavings en 4 casos serializables y 4 casos no serializables. Un caso de interleaving es no serializable si al alterar el orden de ocurrencia del acceso del segundo proceso, el resultado final también es alterado. En consecuencia, para encontrar las violaciones de atomicidad el método consiste en monitorizar los accesos a memoria por parte de los procesos concurrentes durante la ejecución, registrando qué procesos acceden a cada variable, en búsqueda de interleavings no serializables. Pese a que AVIO es superior a las propuestas previas, el overhead que introduce (en promedio 25×) es demasiado elevado para ser utilizado en entornos en producción.

Es importante destacar que por las características propias de los errores de concurrencia, es imposible garantizar que una aplicación este libre de errores cuando llegue a la etapa de producción. Por ello es indispensable contar con herramientas que permitan monitorizar las ejecuciones cuando el software ya ha sido liberado para evitar la ocurrencia de errores no detectados.

Debido a que no existe una versión de AVIO disponible para su descarga e instalación, en este trabajo se decidió implementar una versión acorde a las especificaciones del artículo original. La implementación propia fue evaluada con las mismas aplicaciones y casos de prueba usadas con AVIO, y fue comparada con una herramienta de uso comercial llamada HELGRIND. Los resultados mostraron que la versión implementada posee un overhead de 23×, equivalente al overhead de AVIO (25×). También fue capaz de detectar los mismos errores, por lo que se considera que es una versión representativa de la original. Esta versión servirá de punto de partida para analizar las propuestas de mejora del presente trabajo de tesis.

Capítulo 4

Definición del problema

4.1. Introducción

Este capítulo destaca una serie de conceptos sobre la ejecución de programas paralelos (4.2). Presenta los métodos empleados en la comunidad científica para implementar algoritmos de detección de errores de concurrencia (4.3). Luego analiza las oportunidades de optimización y propone la hipótesis del trabajo de tesis (4.4).

4.2. Ejecución de programas concurrentes

Esta sección resume conceptos sobre hardware y software relacionados a la ejecución de programas concurrentes empleados en el resto de la tesis. La plataforma de hardware y software descrita en esta sección corresponde a una abstracción de las plataformas disponibles en el mercado.

4.2.1. Soporte hardware

Tal como se mencionó en la introducción, las limitaciones físicas impusieron un límite al escalado de frecuencia que permitió a la industria responder a la demanda de mayor rendimiento en los procesadores durante las últimas décadas. Sin embargo, la alta disponibilidad de transistores permitió un nuevo tipo de escalado a partir de la integración de múltiples núcleos de procesamiento dentro del mismo chip. Esta nueva forma de conseguir mayor poder computacional se ha consolidado a través de los años, llegando incluso al

segmento de los dispositivos móviles (Yeap, 2013). La Figura 4.1 presenta la arquitectura de un procesador Westmere, utilizada en los experimentos de esta tesis.

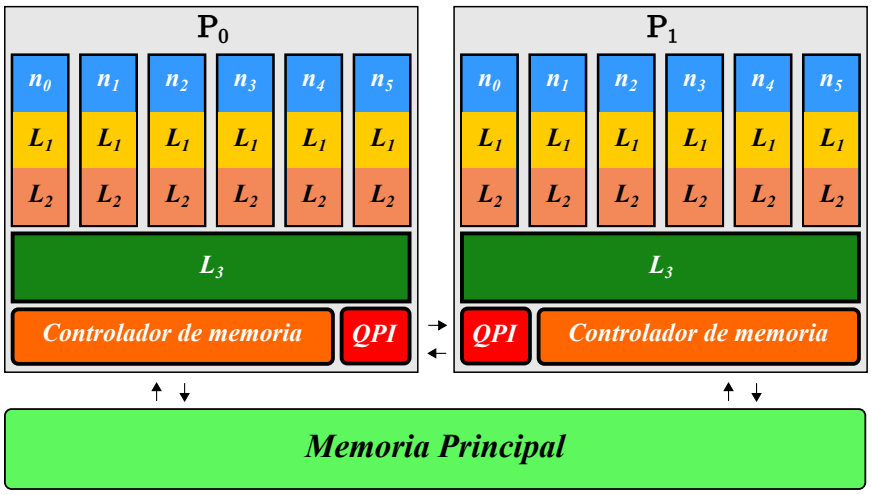


Figura 4.1: Diagrama de estructura de un multicore.

En la figura se puede observar una máquina compuesta por dos procesadores, P_0 y P_1, donde cada uno posee seis núcleos de procesamiento n_0, n_1, n_2, n_3, n_4 y n_5 y un conjunto de memorias cache de alta velocidad L_1, L_2 y L_3. Cada núcleo es una unidad completa de procesamiento (pipeling, predicción de saltos, ejecución fuera de orden y especulación propios), que permiten ejecutar un único flujo de control independiente de los otros núcleos[1]. Si los procesos que se ejecutan en cada núcleo pertenecen al mismo programa, entonces es de esperar que accedan a datos compartidos en memoria principal.

La brecha entre la velocidad de operación de los núcleos y el tiempo de acceso a la memoria principal es mitigada gracias a un conjunto de memorias de alta velocidad denominadas caches, configuradas en una jerarquía en niveles. La jerarquía ejemplificada en la Figura 4.1 presenta tres niveles, donde los primeros niveles L_1 y L_2 son privados a cada núcleo y el nivel L_3 es compartido. La comunicación entre los procesadores se realiza a través del QPI (Quick Path Interconnect), un bus de un interconexión punto a punto de alta velocidad con gran ancho de banda y baja latencia. El acceso a la memoria principal se realiza a través del controlador de memoria. Las zonas de la memoria principal que tienen más accesos se intentan mantener en la cache para aumentar la probabilidad

[1]Existen técnicas como *hiperthreading* que permiten que dos o más hilos coexistan en la misma unidad de ejecución. Sin embargo, aún en esta situación los flujos se ejecutan como si fueran únicos en el procesador.

de aciertos (*hit*) ante una petición futura. Esto significa que con cada acceso a memoria se tendrán dos o más copias del mismo dato: la copia original en la memoria principal y una o varias copias en los distintos niveles de cache. Si además el acceso en cuestión se trata de una escritura, entonces es necesario algún mecanismo para asegurar que las distintas copias de la memoria repartidas en la jerarquía de memoria sean *consistentes*. Esta propiedad se garantiza con la *política de escritura* de la cache.

Por otro lado, un segundo procesador puede intentar acceder a un dato que está siendo accedido por el primer procesador. En esta situación, la respuesta con la copia de memoria puede venir de la memoria principal o de la cache del primer procesador si ésta tiene una copia más reciente (por ejemplo porque ha sido actualizada y esta actualización aún no impactó en memoria principal). Luego, si uno de los procesadores actualiza esa porción de memoria, se debe asegurar que los otros procesadores que poseen copias en sus caches sean conscientes de ello para mantener la *coherencia*.

La coherencia se maneja con un protocolo de invalidación de cache basado en MESI (por ser el más utilizado en los procesadores de uso comercial), donde cada núcleo ejecuta un thread diferente. En este protocolo cada línea de cache puede estar en alguno de los siguientes estados(Sorin, Hill, y Wood, 2011):

- **M**odified (Modificado): la cache tiene la única copia válida y puede ser leída o escrita. Si otro procesador solicita esa línea, entonces la cache debe responder a la petición. Si hay un nivel superior de cache, en ese nivel la línea puede estar actualizada o no.
- **E**xclusive (Exclusivo): la cache tiene la única copia válida, pero sólo puede ser leída. Si hay un nivel superior de cache, en ese nivel la línea se encuentra actualizada. Este estado permite que ante un requerimiento de escritura la línea de cache cambie a estado M sin generar tráfico de coherencia.
- **S**hared (Compartido): la cache tiene una copia válida de solo lectura. Otras caches pueden tener copias de sólo lectura de la línea.
- **I**nvalid (Inválido): la cache no tiene alojada la línea o la copia que posee no es válida.

4.2.2. Soporte software

De igual forma que el hardware, el software se ha visto forzado a evolucionar para dar soporte al cómputo paralelo. De hecho, el proceso constituye el concepto más importante de los sistemas operativos. Un *proceso* es una instancia de un programa que está siendo

ejecutada. Se identifica al proceso como una entidad activa y al programa como una entidad pasiva. Contiene el código del programa y tiene asignado recursos del sistema operativo. Cada proceso tiene su propio espacio de direcciones: aún si el programa tiene un único conjunto de nombres de variables, distintas instancias del mismo programa tendrá diferentes valores para estas variables. Puede tener varios hilos que ejecutan instrucciones concurrentemente.

Por otro lado un *programa* es una colección pasiva de instrucciones. De aquí que un proceso es la ejecución actual de las instrucciones de un programa. Varios procesos pueden ser asociados al mismo programa: ante un pedido de ejecución de un programa, el sistema operativo toma las instrucciones y crea una nueva instancia del proceso.

El sistema operativo ve al proceso como una unidad de planificación, con uno (o varios) hilos secuenciales de control. Para cada hilo debe mantener un registro del contador del programa (que indica la siguiente instrucción a ser ejecutada), valores de los registros del procesador donde se ejecuta y pila de llamadas. También debe estar consciente del espacio de direcciones de memoria asignado al proceso (tanto para código como para datos) y de los archivos que éste ha abierto. Debe facilitar la multiprogramación, permitir compartir e intercambiar datos entre los procesos, proteger sus recursos de accesos no permitidos y establecer mecanismos que permitan sincronizar distintos procesos entre sí.

La forma en que el sistema operativo ve al proceso se conoce como *imagen*. La imagen del proceso es una manifestación física de éste: contiene el programa que está siendo ejecutado, las ubicaciones de los datos en memoria para las variables y constantes, la pila de llamadas a procedimiento y los parámetros pasados y la información usada para gestionarlo denominada PCB (Process Control Block). Dado que en el sistema pueden coexistir miles de procesos, se deben proveer mecanismos que permitan compartir los recursos de cómputo y dispositivos de entrada/salida entre los procesos.

A su vez, en el concepto de proceso se pueden distinguir dos aspectos:

- Los recursos asociados al proceso.
- La unidad o hilo de ejecución (thread).

Esta distinción permite introducir el concepto de *thread* o *proceso liviano*. Un thread es un flujo de ejecución secuencial dentro de un proceso. La ventaja de este concepto es que permite desacoplar los aspectos relacionados a la asignación de recursos (memoria, dispositivos de E/S, etc.) de los aspectos de control de ejecución. De esta manera los procesos se convierten en "contenedores" para uno o más threads, que comparten el mismo espacio

de direcciones. Cada thread replica parte de la estructura de control del proceso en lo que se llama la TCB (Thread Control Block). Distintos threads de un mismo proceso pueden ser ejecutados en múltiples núcleos de procesamiento en paralelo. Estas características hacen de los threads una abstracción útil para representar la concurrencia dentro de un mismo programa.

Otro concepto importante que será utilizado posteriormente en este trabajo es el de *señales*. Una señal permite notificar a un proceso sobre la ocurrencia de algún evento interno o externo (por ejemplo si se intentó ejecutar una instrucción ilegal o si el usuario decidió interrumpir la ejecución del proceso). Las señales son el equivalente en software de las interrupciones en hardware. Cada señal tiene un comportamiento definido por defecto en caso de que el evento ocurra. Sin embargo, el sistema operativo provee mecanismos que permiten al programador cambiar ese comportamiento predefinido ya sea para ignorar o capturar esas señales a través de un procedimiento especial llamado *handler* (manejador).

4.2.3. Modelo de ejecución

A partir de los conceptos antes expuestos, es posible abstraer un modelo de ejecución para programas concurrentes.

El programa en ejecución se representa como un proceso: el sistema operativo se comunica con este a través de señales y el hardware a través de interrupciones.

Las tareas concurrentes del programa se representan como hilos: cada hilo comparte los recursos del proceso. El sistema operativo también provee mecanismos que permiten establecer una *afinidad* (preferencia) a cada hilo sobre en cuál procesador desea ser ejecutado. Si hay núcleos de procesamiento disponibles, entonces cada hilo se ejecuta en un núcleo diferente.

Por último, es interesante destacar que los errores de concurrencia corresponden a actualizaciones sobre datos compartidos entre distintos hilos. Estas interacciones entre hilos se hacen evidentes recién en el primer nivel compartido de cache en la jerarquía de memoria.

4.3. Implementación de algoritmos de detección

El overhead del proceso de detección depende del algoritmo empleado, pero fundamentalmente del método por el cuál se implementa ese algoritmo. Esa implementación se

puede hacer por *software* a través de una herramienta de instrumentación de programas o a través de *hardware* a través de lo que los autores llaman extensiones.

4.3.1. Instrumentación

Aunque las propuestas anteriores se diferencian en las ideas y algoritmos de detección, todas coinciden en un aspecto: requieren *instrumentar* el código que analizan en búsqueda de errores.

Se entiende por instrumentación de un programa al proceso de agregar o remover código del mismo. Normalmente este código se relaciona a algún tipo de rutina de análisis con el objeto de obtener información sobre el comportamiento del programa o realizar modificaciones que tengan impacto en el rendimiento del mismo.

El proceso de instrumentación se puede llevar a cabo de dos maneras:

- *Instrumentación estática*. Consiste en realizar modificaciones al código fuente del programa. Esto significa que es necesario tener acceso al código fuente del proyecto y recompilarlo luego de cada modificación en la instrumentación.

- *Instrumentación dinámica*. Consiste en utilizar una aplicación que literalmente *virtualiza* la ejecución del proceso instrumentado, permitiendo que el código añadido se agregue en tiempo de ejecución. El flujo de control del programa se va descubriendo a medida que este se ejecuta. La ventaja de este método es que no requiere acceso al código fuente del programa, ya que se opera directamente con los binarios.

Todas las propuestas anteriores que implementan versiones software de sus algoritmos utilizan una de estas dos estrategias. Por un lado la instrumentación estática requiere acceso al código fuente del programa que se quiere instrumentar, sus modificaciones se incluyen en el mismo y forman parte del ejecutable final. Por el otro, la instrumentación dinámica se utiliza directamente con procesos (aunque en la práctica es útil tener acceso al código fuente del programa), las modificaciones se agregan en tiempo de ejecución y de igual manera pueden ser retiradas del mismo. En cualquiera de los casos el overhead es tan elevado que hace imposible utilizar las versiones software en entornos en producción.

La propuesta seleccionada como caso de estudio está implementada con el método de instrumentación dinámica. Salvo que se indique lo contrario, en el resto de la tesis se debe entender instrumentación como instrumentación dinámica. Conceptualmente la instrumentación consiste en dos componentes:

- Código de instrumentación: un mecanismo que especifica dónde el código es agregado.

- Rutina de análisis. el código que debe ser ejecutado.

El código de instrumentación define la *granularidad de la instrumentación*, mientras que la rutina de análisis implementa el algoritmo deseado. El código de instrumentación se ejecuta una vez por ejecución, justo antes de que el programa instrumentado comience a ser ejecutado. La rutina de análisis se ejecuta muchas veces dependiendo de la granularidad empleada, por lo que reducir este número de llamadas hace la instrumentación más eficiente. De allí que aunque es importante elegir adecuadamente la granularidad de la instrumentación, es mucho más importante ajustar la rutina de análisis.

Granularidad de instrumentación. Las herramientas de instrumentación dinámica dividen el código en *trazas*. Una traza usualmente comienza con la primera instrucción luego de una bifurcación en el código (branch) y finaliza con un salto incondicional, llamadas a funciones o instrucciones *returns*. Se debe notar que aunque está garantizado el inicio de cada traza, su fin depende de las instrucciones ejecutadas, pudiendo tener múltiples salidas. Cada vez que finaliza una traza, otra comienza con la primera instrucción luego del salto tomado.

En un grano más fino la instrumentación se puede hacer a nivel de instrucción. En este nivel se llamará a la rutina de análisis antes o después de cada instrucción del programa instrumentado.

Además es posible realizar la instrumentación a nivel de imagen del proceso, lo que permite inspeccionarlo como una unidad o a nivel de rutina y agregar código al programa antes o después que la rutina es ejecutada. El programador puede realizar la instrumentación en alguno de estos modos y dividir el código en sus partes más finas, como ser secciones, rutinas o instrucciones. La ventaja de realizar la instrumentación de este modo es que el programador puede inspeccionar el tipo de la parte a instrumentar y decidir si hacerlo o no. Por ejemplo, es posible determinar si la instrucción a la que se agregará la rutina de análisis es un salto, un acceso a memoria o una operación aritmética, y en función de ello decidir si insertar o no la rutina de análisis.

4.3.2. Extensiones hardware

Debido al overhead generado por cualquiera de las técnicas anteriores implementadas en software, muchas de ellas complementan sus trabajos con propuestas de extender el hardware actual y obtener mejoras significativas en el rendimiento:

- AVIO propone una serie de extensiones hardware a nivel de cache y procesador para implementar el algoritmo.
- Pset deriva en añadir soporte al procesador para validar eficientemente las restricciones y evitar que se ejecuten interleavings no probados.
- Bugaboo de manera similar a PSet, utiliza extensiones de hardware para detectar cuando en un grafo aparece un nodo anormal (que no existe en los grafos de entrenamiento) y emite un alerta.
- Atom-aid utiliza un procesador especial para ejecutar esos bloques.
- AtomTracker propone una implementación hardware para ejecutar eficientemente el algoritmo ya que la versión software produce demasiado overhead.

Por ejemplo, la Tabla 4.1 muestra los resultados de ejecutar las versiones software y hardware de AVIO sobre algunas de las aplicaciones de la suite de benchmarks SPLASH-2 (Woo y cols., 1995) [2].

Cuadro 4.1: Overhead de las versiones implementadas en software y hardware de AVIO.

Benchmark	**AVIO (Hardware)**	**AVIO (Software)**
fft	$1,05\times$	$42\times$
fmm	$1,04\times$	$19\times$
lu	$1,04\times$	$23\times$
radix	$1,04\times$	$15\times$
Promedio	$1,04\times$	$25\times$

La x en la tabla representa el tiempo de la aplicación monitorizada sin instrumentación. Mientras que la versión software introduce en promedio un $25\times$ de overhead, la versión hardware apenas llega a introducir un $1,04\times$ de overhead.

[2] Cabe mencionar que los datos fueron extraídos del artículo original de AVIO (Lu y cols., 2006), pero para facilitar su interpretación se unificó la notación empleada por sus autores.

En todos los casos, para implementar estas versiones se requiere hardware dedicado que debe ser incorporado a la arquitectura, y por lo tanto este tipo de soluciones no son viables en las plataformas que se encuentran actualmente en producción. Además sus resultados son menos precisos que la versión equivalente en software: normalmente las extensiones hardware proveen una granularidad mayor que la versión software, por lo que están expuestas a detectar mayor cantidad de falsos positivos. Por ejemplo, una extensión relativamente sencilla de implementar consiste en agregar bits de control a las lineas de cache, pero una línea de cache puede contener varias variables. Un algoritmo implementado en hardware podría detectar que dos procesos escribieron en la misma línea, pero no podría determinar si lo hicieron a la misma variable o a variables distintas. En este caso el algoritmo debería informar que ocurrió un error, aunque sea probable que no haya ocurrido realmente.

4.4. Oportunidades de optimización

En la Tabla 4.1 se observa claramente que la versión hardware es superior a la versión software en términos de rendimiento. Sin embargo, implementar un algoritmo de esta manera requiere hardware dedicado que debe ser incorporado a la arquitectura, lo que hace inviable a este tipo de soluciones en las plataformas que se encuentran actualmente en producción. Por otro lado, las versiones hardware de los algoritmos son menos precisas que sus versiones software. Normalmente las extensiones hardware proveen una granularidad mayor, por lo que están expuestas a detectar mayor cantidad de falsos positivos.

De la misma manera, las versiones software tienen una mayor precisión en la detección de errores, pero el costo debido al overhead que introducen es muy elevado. Es deseable tener una herramienta de detección de errores con la precisión y capacidad de detección de las versiones software pero con el rendimiento de las versiones hardware.

La optimización de esta relación de compromiso entre precisión, rendimiento y posibilidades de implementación de las herramientas de detección se presentan como la motivación principal de esta tesis.

4.4.1. Overhead causado por la instrumentación

La Tabla 4.2 muestra el resultado de ejecutar un conjunto de aplicaciones de la suite SPLASH-2 (Woo y cols., 1995) a través de una herramienta de instrumentación dinámi-

ca llamada PIN (Luk y cols., 2005) con diferente granularidad de instrumentación. Esta herramienta es utilizada para implementar la versión software de la mayoría de las propuestas antes mencionadas. En este caso para determinar el costo en términos de overhead debido a la instrumentación dinámica, en lugar de implementar una costosa rutina de análisis para detectar errores se optó por una función que sólo incrementa un contador. Cada experimento se realizó 10 veces y los datos mostrados corresponden al promedio entre ejecuciones. Como la rutina de análisis es verdaderamente simple, la diferencia en tiempo entre experimentos se debe a las diferentes granularidades de instrumentación.

Cuadro 4.2: Comparación de tiempos de ejecución de diferentes granularidades de instrumentación. Las aplicaciones corresponden a la suite SPLASH-2.

	Granularidad de instrumentación		
programa	***Imagen***	***Rutina***	***Instrucción***
barnes	0,965	1,780	12,767
cholesky	1,018	1,048	2,585
fft	0,691	0,718	1,113
fmm	1,207	1,255	5,117
lu_cb	0,696	0,709	3,790
lu_ncb	0,643	0,658	1,050
ocean_cp	1,170	1,179	3,149
ocean_ncp	1,084	1,095	3,081
radiosity	1,318	2,436	24,301
radix	0,637	0,663	1,038
raytrace	1,306	1,519	5,151
volrend	1,116	1,148	4,700
water_nsq	0,864	0,963	3,437
water_sp	0,896	1,007	3,774
Total	13,611	16,178	75,053

La última fila de la tabla corresponde a la suma de los tiempos para cada nivel de granularidad de instrumentación entre todas las aplicaciones de la suite. Como es de esperarse, a medida que se utiliza una granularidad más fina, el costo en tiempo es mayor.

Dado que los algoritmos de detección están basados en analizar las historias de accesos a memoria, en su implementación se trabaja con granularidad a nivel de instrucción. Es de interés destacar que el tiempo de instrumentación a nivel de instrucción es de 5,51 veces el tiempo con instrumentación a nivel de imagen.

4.4.2. Hipótesis

Idealmente para obtener el mejor rendimiento, los algoritmos de detección para monitorizar ejecuciones de procesos en tiempo de ejecución deberían ser implementados en hardware. Lamentablemente, las modificaciones necesarias para ello son inviables, ya que no solo implica diseñar un procesador nuevo, sino que también requeriría reemplazar to-

dos los procesadores que actualmente se encuentran en producción. Por ello se busca una alternativa viable en software, que pueda ser implementada en las arquitecturas actuales pero con un mejor rendimiento que las propuestas actuales.

En la sección anterior se destacó que una parte importante del overhead del proceso de instrumentación dinámica proviene de la granularidad de instrumentación empleada. También se destacó que esta granularidad es la empleada en los algoritmos de detección, por lo que no es posible pasar a un grano más grueso para obtener mejor rendimiento. En este sentido, se considera que una parte importante del overhead en las versiones software se debe al costo que implica instrumentar cada acceso a memoria que realiza la aplicación monitorizada. Sin embargo, se debe notar que la posibilidad de error solamente existe si al menos dos threads acceden simultáneamente a datos compartidos. Esto significa que, si de la totalidad de la aplicación que está siendo monitorizada sólo un pequeño porcentaje de las operaciones acceden a datos compartidos, gran parte del tiempo invertido en instrumentar todos los accesos a memoria está siendo desperdiciado.

Por lo tanto, la hipótesis de este trabajo de tesis es que el overhead del proceso de detección de errores de concurrencia será disminuido si la instrumentación a nivel de instrucción se restringe sólo a aquellos fragmentos de código que acceden a datos compartidos.

Para probar esta hipótesis se diseñará una variante del algoritmo de detección de violaciones de atomicidad AVIO. Se debe notar que instrumentar el código en búsqueda de fragmentos de código que acceden a datos compartidos sería tan costoso como los métodos actuales (o incluso más). Es de interés sumar las ventajas de implementar en software el algoritmo de detección pero con los aspectos de rendimiento de la implementación en hardware. Un trabajo que sirvió de inspiración para esta tesis fue (Greathouse, Ma, Frank, Peri, y Austin, 2011). En éste se propuso mejorar el rendimiento de una herramienta de detección de condiciones de carrera utilizando contadores de hardware para detectar eventos en el sistema de cache que indiquen compartición de datos. El enfoque de esta propuesta buscará en el sistema de memoria cache eventos que permitan diseñar un modelo de instrumentación híbrido, que puedan ser utilizados para activar la rutina de análisis en tiempo de ejecución sólo cuando se accede a datos compartidos.

4.5. Resumen

En las últimas décadas el hardware evolucionó para dar soporte al cómputo paralelo. Todos los procesadores disponibles en el mercado (incluso los procesadores utilizados en dispositivos móviles) poseen una arquitectura típica multicore. Normalmente, un multicore está compuesto por dos o más unidades de procesamiento donde cada una puede ejecutar un flujo secuencial de instrucciones. Si estos flujos secuenciales de instrucciones corresponden a tareas de un mismo programa concurrente, entonces se habla de una ejecución paralela. Debido a que el acceso a la memoria es muy costoso, implementan una jerarquía de memorias de alta velocidad llamadas cache (organizadas en dos o más niveles) con la que se pretende emular la velocidad de los procesadores. Esta jerarquía aprovecha las propiedades de localidad espacial y temporal de los programas para mantener lo más cerca del procesador los datos con los que trabaja.

Por su parte, el software también evolucionó para dar soporte al cómputo paralelo: el sistema operativo se encarga de planificar la ejecución de los procesos, asignarles recursos y monitorizar su ejecución. También incorpora el concepto de thread (o proceso liviano) y librerías para su manipulación y desarrollo. Un proceso puede tener uno o varios threads en ejecución que comparten el mismo espacio de direcciones y recursos asignados al proceso, haciendo de este modelo una abstracción ideal para expresar la concurrencia de los programas. También provee un mecanismo denominado *señales* para interrumpir la ejecución de los procesos. Esta característica permite terminar con todos los threads asociados a un mismo proceso con una única operación, facilitando la gestión de los mismos.

Tomando en cuenta lo anterior, es que las propuestas de implementación de herramientas de detección se clasifican en dos grupos:

- Las que se implementan en hardware
- Las que se implementan en software

Muchas propuestas proponen para su implementación modificaciones al hardware (cambios en el procesador, memoria cache, etc.) con las que se consiguen excelentes resultados. Sin embargo, este enfoque requiere que los fabricantes de procesadores decidieran incorporar esas modificaciones en sus diseños (cosa que no ha sucedido por el momento), por lo que es de esperar que tardarán en llegar al mercado y más aún en reemplazar las plataformas que actualmente están en producción.

Por otro lado, las implementaciones en software aplican métodos de instrumentación de programas. La instrumentación puede ser estática (cuando se usan anotaciones en el código

fuente del programa) o dinámica (cuando la estructura del programa se descubre durante la ejecución y en consecuencia la instrumentación se realiza en el momento). Debido a que los errores de concurrencia –particularmente las violaciones de atomicidad– dependen del no determinismo de la ejecución para manifestarse, el método de instrumentación dinámica es ideal para ello. Es posible realizar la instrumentación con diferentes grados de granularidad: por ejemplo a nivel de imagen, de rutina o de instrucción. En particular, los métodos de detección de errores utilizan instrumentación a nivel de instrucción ya que requieren agregar código llamado *rutina de análisis* a cada instrucción que accede a la memoria. Esta rutina registra el orden en que ocurren los accesos de los distintos threads sobre cada variable y ejecutan costosos análisis para determinar la ocurrencia de un error. Lamentablemente, esta granularidad de instrumentación es lenta, penalizando el tiempo de la ejecución con más de un orden de magnitud.

Sin embargo, se debe notar que la posibilidad de error solamente existe si al menos dos threads acceden simultáneamente a datos compartidos. Esto significa que, si de la totalidad de la aplicación que está siendo monitorizada sólo un pequeño porcentaje de las operaciones acceden a datos compartidos, gran parte del tiempo invertido en instrumentar todos los accesos a memoria está siendo desperdiciado.

Es interesante destacar que en el modelo de ejecución habitual con procesadores multicore, el sistema operativo asigna recursos de ejecución a un proceso, y este intenta ejecutar un thread en cada núcleo. Se debe notar que en este modelo de ejecución los datos compartidos estarán en las memorias cache de los núcleos que estén ejecutando los respectivos procesos. En este sentido se buscará encontrar una solución a este problema restringiendo la instrumentación a nivel de instrucción sólo a aquellos fragmentos de código que acceden a datos compartidos.

Capítulo 5

Eventos de hardware

5.1. Introducción

Este capítulo introduce al tema de eventos y contadores de hardware (5.2) y se comentan los modos de operación de los mismos (5.3). Luego define las características del evento que se debe poder detectar para los propósitos de este trabajo y se selecciona un evento candidato (5.4). A continuación se propone un método de validación de eventos y se lo utiliza con el evento candidato seleccionado (5.5). Finalmente, se presentan los resultados experimentales del capítulo (5.6).

5.2. Contadores hardware

Todos los procesadores actuales tienen un conjunto especial de registros llamados contadores hardware (Sprunt, 2002). Esos registros pueden ser programados para contar el número de veces que un evento ocurre dentro del procesador durante la ejecución de una aplicación. Los eventos proveen información sobre diferentes aspectos de la ejecución de un programa (por ejemplo el número de instrucciones ejecutadas, el número de fallos en cache L1 o el número de operaciones en punto flotante ejecutadas). Esta información pretende ayudar al programador a comprender mejor el desempeño de sus algoritmos y optimizar su programa. Sin embargo, aunque hoy es relativamente simple hacer uso de los contadores, no siempre fue así: siempre existieron registros ocultos destinados a la depuración de problemas en el hardware, pero no fue hasta 1993 que se hizo pública su existencia y posibilidades para depuración de software a la comunidad de usuarios. En esta sección se presenta un recorrido por la evolución del hardware y software que en las

últimas décadas dieron soporte para acceder a los contadores. La Figura 5.1 compara a través de una línea de tiempo la evolución de ambos aspectos.

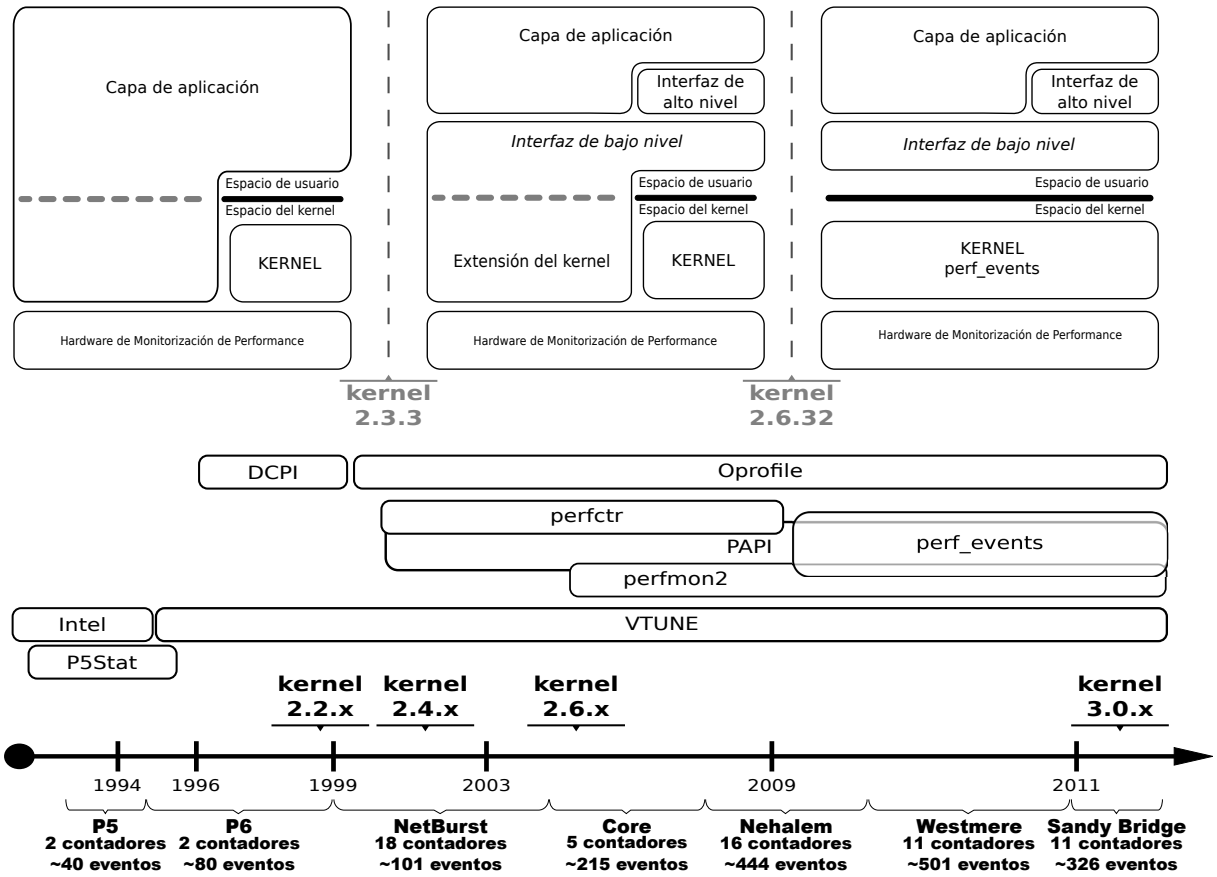


Figura 5.1: Línea de tiempo de evolución de soporte hardware y software para el uso de contadores.

5.2.1. Soporte hardware

Independientemente de los nombres comerciales con los que se conoce a los procesadores, los cambios importantes de un procesador al otro depende de la microarquitectura a la que pertenecen. En esta sección se tratan los cambios entre microarquitecturas Intel para computadoras de escritorio y servidores. Se ha dejado intencionalmente afuera aquellas microarquitecturas que fueron empleadas exclusivamente para notebooks.

Los contadores hardware se introdujeron para uso por primera vez con el pentium en el año 1993, el cual poseía una microarquitectura denominada **P5**. El P5 posee dos registros

especiales de máquina MSR (Machine Specific Register), con los que se puede seleccionar y medir entre aproximadamente 40 eventos distintos.

Con la siguiente generación de procesadores basados en la microarquitectura **P6** se mejoró el mecanismo de monitorización y, aunque aún se tienen sólo dos contadores, se amplió la cantidad de eventos que pueden ser medidos: cuenta con más de 80 eventos diferentes. Estos registros, aparte de poder ser configurados para contar eventos, pueden ser usados para medir el tiempo empleado en el evento (esto es, el número de ciclos reloj empleados mientras el evento ocurre). Además provee una opción para generar una interrupción cuando ocurre un overflow en el registro. Los procesadores Pentium Pro, Pentium II y Pentium III pertenecen a esta generación, y estuvieron presentes desde el año 1995 hasta el 2000 aproximadamente.

En el año 2000 Intel incorporó una gran cantidad de cambios con la microarquitectura que llamó **Netburst**. En esta generación entran el Pentium IV y el Pentium IV HT. Entre los cambios principales, el mecanismo de configuración empleado en las arquitecturas anteriores es incompatible con Netburst. La cantidad de contadores se incrementó a 18 y se amplió la lista de eventos a 100. Estos eventos ahora se clasifican en eventos *Non-retirement* y *At-retirement*. Los primeros pueden ocurrir en cualquier momento durante la ejecución de la instrucción, mientras que los segundos ocurren sólo en la etapa final del pipe. Además se incorporó una característica llamada muestreo basado en evento preciso PEBS (Precise Event Based Sampling). Básicamente permite obtener información sobre el estado arquitectónico a intervalos regulares del evento que se está contando.

Hasta el 2006, todos los eventos de una microarquitectura eran específicos de ella y por lo tanto incompatibles con las demás. A partir de la microarquitectura **Core**, los eventos se dividieron en 7 eventos arquitectónicos y 208 no arquitectónicos. Los registros se clasificaron en contadores para funciones precisas y contadores de propósito general. Los eventos arquitectónicos son eventos que están disponibles en todas las arquitecturas a partir de core, en cambio los eventos no arquitectónicos pueden variar (aún entre modelos de la misma microarquitectura). Además a partir de esta microarquitectura los procesadores poseen varios núcleos, por lo que también se incorporaron mecanismos que permiten configurar los eventos para contar sobre uno o varios procesadores lógicos, soportar tecnología Hyper-Threading o capacidades de multi-threading simultaneo. Con estos cambios se puede configurar un contador para que esté vinculado a un thread y, si este cambia de procesador lógico, la cuenta continúe en el nuevo procesador.

En 2008 con la llegada de la microarquitectura **Nehalem** los procesadores soportan todas las características anteriores. Los registros se organizaron en Unidades de Monitorización de Performance que agrupan las facilidades enumeradas (Performance Monitoring Unit). Se aumentó la precisión de los contadores para que puedan retener cuentas mayores. Además permite establecer prioridades entre diferentes eventos. El sistema de monitorización se vuelve más complejo, identificando como un sistema aparte de monitorización al sistema encargado de gestionar la interacción entre diferentes núcleos de procesamiento. Este sistema llamado *uncore* (llamado así para diferenciarlo de los eventos *core*) provee facilidades adicionales de monitorización fuera del núcleo de procesamiento. La PMU core posee 7 registros, mientras que la PMU uncore posee 9. A los 7 eventos arquitectónicos se suman los no arquitectónicos: 262 core y 175 uncore. El core i7 es un procesador basado en esta tecnología.

La microarquitectura **Westmere** introducida en el mercado en el año 2010 no realizó grandes cambios en el sistema de monitorización de performance. El número de contadores de propósito general se fijó en 8. Además permite configurar hasta tres contadores por thread. Soporta los 7 eventos arquitectónicos y los no arquitectónicos se dividen en: 275 core y 219 uncore. Algunos procesadores que implementan esta tecnología son los Core i3, Core i5 y Core i7. Se debe notar que Core i7 es el mismo nombre con que se comercializaron procesadores basados en Nehalem.

En 2011 Intel presentó su microarquitectura **Sandy Bridge**. Soporta los 7 eventos arquitectónicos, 297 eventos core no arquitectónicos y sólo 22 eventos uncore.

5.2.2. Soporte software

En esta sección se tratan los cambios que atravesó el software para acceder a los contadores, principalmente en sistemas operativos linux. Al igual que el hardware, el software ha evolucionado con el tiempo para adaptarse a los cambios en las arquitecturas. En función del soporte por parte del sistema operativo y la facilidad de uso, se pueden distinguir tres generaciones de software para acceder a los contadores:

- Software sin soporte del Sistema Operativo
- Software con soporte parcial del Sistema Operativo
- Software con soporte total del Sistema Operativo

Las siguientes secciones proveen información sobre cada generación.

Software sin soporte del Sistema Operativo

Esta generación inicia con la aparición del Pentium en el año 1993. Inicialmente no había documentación pública sobre estos registros especiales, y sólo se tenía acceso a ellos a través de herramientas provistas por Intel (como por ejemplo VTune), previa firma de un acuerdo de confidencialidad para poder utilizarlas.

Una de las primeras referencias públicas a estos registros está en la revista *Byte* de julio de 1994. En esa edición se publicó un artículo titulado "Pentium Secrets"(Mathisen, 1994), donde el autor explicaba un método para obtener métricas de rendimiento a partir de los contadores de hardware presentes en el Pentium, útiles para optimizar programas. El artículo relata las dificultades que tuvo el autor para descubrir cómo usarlos, y proporciona un listado no oficial de eventos que podían ser medidos en ese procesador. Incluso proporciona un link de descarga para una herramienta freeware (desarrollada por él) para obtener estas métricas llamada *P5Stat*.

Otra referencia de la época es DCPI (Anderson y cols., 1997), un proyecto de HP diseñado para obtener información sobre diferentes aspectos de rendimiento en sistemas en producción, que luego se transformó en Oprofile, una herramienta disponible para linux que permite generar profiling y gráficos de llamadas por programa. En esta época el sistema operativo no proveía soporte de ningún tipo, y cada proyecto implementaba manualmente la interfaz con el hardware de monitorización de la arquitectura. En esta generación si no se usaba alguna herramienta como las anteriores, había que hacer un gran esfuerzo de programación para acceder a los contadores.

Software con soporte parcial del Sistema Operativo

A principios del año 2000 la comunidad de usuarios comenzó a desarrollar extensiones al kernel de linux, que permitían acceder a estos registros. Con el kernel 2.3.3 surgió un proyecto llamado *perfctr*, un parche para el kernel que proveía una extensión para acceder a los registros en la mayoría de los procesadores de la época. Luego, con el kernel 2.6.x surgió otro proyecto llamado *perfmon2* (Eranian, 2006) que provee una interfaz de monitorización más uniforme que perfctr. Entre las características más destacadas, fue la primera interfaz en ofrecer soporte para muestreo basado en eventos PEBS (Precise Event Based Sampling).

Estos proyectos sirvieron de soporte para herramientas de más alto nivel como *PAPI* (Garner y cols., 2000). Esta herramienta provee una API para acceder a los contadores

desde los programas, permitiendo una granularidad menor en las mediciones que Oprofile. PAPI ofrece un conjunto de constantes para acceder a los registros a través de nombres. Además proporciona soporte multiplataforma y, de manera similar a los eventos arquitectónicos, ofrece una interfaz de alto nivel para lo que denomina eventos predefinidos: estos son eventos que pueden encontrarse en diferentes arquitecturas con diferentes nombres, pero que PAPI ofrece acceso a través de un nombre común (por ejemplo el número de fallos de cache L_1). Para el resto de eventos que PAPI no puede garantizar una configuración equivalente en todas las arquitecturas, también ofrece una interfaz de bajo nivel accesible a través de nombres. PAPI se constituyó en uno de los proyectos más importantes, en el que se apoyan otras interfaces de más alto nivel como *perfSuite*.

Software con soporte total del Sistema Operativo

La tercera generación llega con el kernel 2.6.33. La comunidad que desarrolla el kernel de linux incluye soporte nativo para los eventos a través de la librería *perf_ events*. Esta reemplaza totalmente a perfctr. Perfmon2 evoluciona para ubicarse por encima de perf_events a través de *libpfm4.3*. PAPI se posiciona sobre perf_events y se consolida como aplicación de referencia multiplataforma para el acceso a los contadores. El cambio más importante es que ahora hay una clara separación entre el espacio de usuario y el espacio del kernel. Con esta generación linux incluye una herramienta propia para obtener información desde el modo de usuario llamada *perf* (*Tutorial - Perf Wiki*, s.f.).

5.3. Uso de los contadores

La utilización de contadores hardware requiere un amplio conocimiento sobre la arquitectura del procesador con el que se está trabajando. Aunque en teoría es posible saber cómo es la microarquitecutra de un procesador, en la práctica es difícil conocer con precisión los detalles de cada modelo. Además los eventos pueden variar entre diferentes arquitecturas y modelos, dificultando la tarea de generalizar las conclusiones obtenidas en una arquitectura en particular.

El acceso a estos recursos se puede llevar a cabo usando diferentes herramientas en función del nivel de abstracción deseado: Linux provee una aplicación de usuario llamada perf que puede ser utilizada para monitorizar eventos generados durante la ejecución de un programa sin necesidad de recompilar el código. Este modo de operación permite

obtener una métrica del evento deseado simplemente ejecutando la aplicación a través del programa perf.

Otra alternativa es PAPI, una API que permite al programador obtener información sobre segmentos específicos de sus programas, otorgando cierto grado de portabilidad a su código. En este modo de operación el programador puede especificar sobre qué fragmentos desea realizar las mediciones de los eventos seleccionados. Requiere recompilar el código y enlazarlo con la librería, por lo que es necesario contar con el código fuente del programa monitorizado.

Por último, para conseguir prestaciones más avanzadas, es necesario recurrir a alguna interfaz de bajo nivel que permita el control total de los registros: en el caso de Linux es el subsistema de gestión de eventos perf_events (Weaver, 2014), el cual permite explotar el potencial de los contadores a partir de la información disponible en los manuales del procesador en uso.

Esta información facilita encontrar los motivos que penalizan la ejecución de un programa y comparar con datos concretos los beneficios de los cambios que se realizan.

Documentación de eventos

Cualquiera de las librerías empleadas para manipular los contadores hardware ofrecen algún nivel de abstracción sobre el hardware a través de un subconjunto de eventos que representan comportamientos o situaciones similares en diferentes arquitecturas. Estos eventos se suelen identificar como eventos *predefinidos* o *predeterminados*.

Sin embargo, en proporción a la cantidad de eventos disponibles, los eventos predefinidos o predeterminados son pocos. Para acceder a aspectos más específicos de la ejecución es necesario recurrir a los manuales de los fabricantes y leer las tablas de eventos soportados por cada arquitectura. Desafortunadamente en la mayoría de los casos los eventos están pobremente documentados, y la descripción que acompaña a cada evento no es clara sobre lo que mide exactamente, o incluso posee errores. La figura 5.2 muestra el evento utilizado en el resto del trabajo para dos versiones del mismo manual. Aunque las referencias son de la misma microarquitectura de procesadores Intel, se puede observar que en la versión de 2011 (5.2a) el evento se llamaba LOCAL_HITM, mientras que en la versión de 2012 (5.2b) el evento se llama OTHER_CORE_L2_HIT.

Ya sea por un error de documentación, por desconocimiento de los términos empleados o por un mal entendido de la descripción de los eventos, la documentación que propor-

0FH	02H	MEM_UNCORE_RETIRED.LOCAL_HITM	Load instructions retired that HIT modified data in sibling core (Precise Event).	

(a) Documentación para el evento 0x020f en el manual de intel número 253669-037US publicado en enero de 2011.

0FH	02H	MEM_UNCORE_RETIRED.OHTER_CORE_L2_HIT	Load instructions retired that HIT modified data in sibling core (Precise Event).	Applicable to one and two sockets

(b) Documentación para el evento 0x020f en el manual de intel número 253669-043US publicado en mayo de 2012.

Figura 5.2: Errores de documentación en los manuales de intel.

cionan los fabricantes de procesadores sobre los eventos que pueden ser medidos en cada microarquitectura es insuficiente.

Esta situación derivó en la necesidad de diseñar un método para validación de eventos de hardware basado en un proceso experimental. En las siguientes secciones se presenta el método y cómo se lo utilizó para determinar la utilidad de un evento en particular en la detección de accesos a datos compartidos.

5.4. Identificar el evento que se desea detectar

De acuerdo a la hipótesis planteada en el capítulo anterior, se busca un evento que indique la ocurrencia de un interleaving no serializable para, en función de la ocurrencia de este evento, activar o no la herramienta de detección y en consecuencia reducir el overhead de la herramienta. Los interleavings no serializables corresponden a los casos 2, 3, 5 y 6 de la Tabla 3.1. En el Manual del desarrollador de software para las arquitecturas de Intel (*Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2012) se encuentra la lista de eventos disponibles para todas sus arquitecturas de procesadores.

Debido a que la detección de acceso a un dato compartido recién se hace evidente en el primer nivel de cache compartido entre núcleos, inicialmente se buscaron eventos asociados a la cache que indicaran interleavings. Lamentablemente, no existe un evento capaz de indicar la ocurrencia de estos casos. En consecuencia, se optó por buscar un evento que aunque no fuera preciso, permitiera tener una aproximación a los casos de interleavings que se deseaban detectar.

Un interleaving es una interacción entre tres accesos a la misma dirección de memoria, donde dos de los accesos son realizados por un proceso y el tercer ocurre justo entre los dos primeros. En consecuencia la estrategia fue buscar eventos que describieran patrones de acceso a memoria compartida. Como cada acceso puede ser una lectura o una escritura, se tienen cuatro casos o patrones de acceso posibles a un dato compartido en memoria entre dos procesos, resumidos en la Tabla 5.1.

Cuadro 5.1: Cada caso muestra un patrón de acceso diferente a una misma dirección de memoria entre dos threads.

Caso	**Patrón**		**Caso**	**Patrón**	
	thr_0	thr_1		thr_0	thr_1
0	read	read	2	write	read
1	read	write	3	write	write

Se debe notar que cada patrón de acceso sirve para detectar diferentes interleavings, ya que un interleaving está formado por dos patrones de acceso: uno entre el primer acceso del primer proceso y el acceso del segundo proceso, y otro entre el acceso del segundo proceso y el segundo acceso del primero. La Tabla 5.2 muestra cómo se relacionan los patrones de la Tabla 5.1 con los interleavings de la Tabla 3.1.

Cuadro 5.2: Relación entre patrones de acceso a memoria compartida e interleavings.

		0 $\begin{matrix}R\\R\end{matrix}$ R	1 $\begin{matrix}W\\R\end{matrix}$ R	2 $\begin{matrix}R\\R\end{matrix}$ W	3 $\begin{matrix}W\\R\end{matrix}$ W	4 $\begin{matrix}R\\W\end{matrix}$ R	5 $\begin{matrix}W\\W\end{matrix}$ R	6 $\begin{matrix}R\\W\end{matrix}$ W	7 $\begin{matrix}W\\W\end{matrix}$ W
0	$R \to R$	SI	SI	-	-	SI	-	-	-
1	$R \to W$	-	-	SI	-	SI	SI	SI	-
2	$W \to R$	-	SI	SI	SI	-	SI	-	-
3	$W \to W$	-	-	-	SI	-	-	SI	SI

Es de interés destacar que aunque se esté accediendo a datos compartidos, sólo se busca detectar los cuatro casos de interleavings no serializables (2, 3, 5 y 6). En este sentido, se debe notar que el patrón de acceso 0 no está presente en ninguno de los cuatro casos requeridos, motivo por el cual puede ser desestimado. De los tres patrones restantes, los patrones 1 y 2 están presentes en tres de los cuatro interleavings requeridos, mientras que el patrón 3 sólo en dos.

Nuevamente el mejor lugar para buscar eventos que describan estos patrones es en el sistema de cache. Es importante notar que estos patrones de acceso son operaciones comunes en la jerarquía de memoria cache, por lo que es muy probable que existan eventos relacionados al protocolo de coherencia que sean representativos de estos patrones. Cualquier evento que describa uno de esos patrones será un buen indicador de que están ocurriendo interleavings no serializables, y como tal de que la herramienta de detección debe ser activada.

5.4.1. Protocolo de coherencia cache

Se asume un procesador con dos núcleos donde hay al menos un nivel de cache compartida entre ambos, con las características descritas en 4.2.1. La Figura 5.3 muestra los cambios de estado de coherencia para la línea de cache solicitada por ambos threads en función de los patrones de acceso. Se omitieron las transiciones de estado que no son significativas para el objeto de este trabajo.

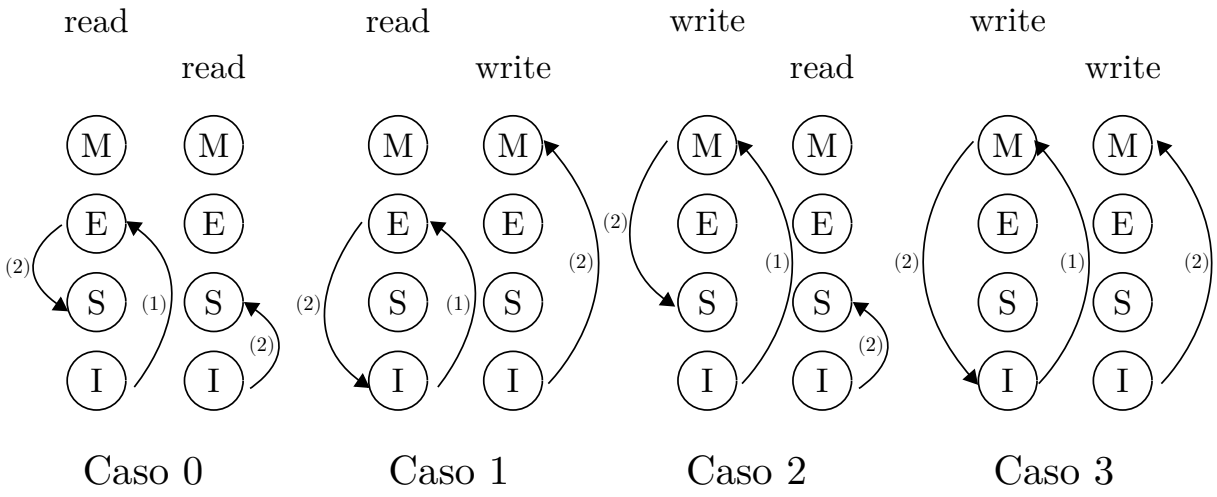


Figura 5.3: Cambios de estado de coherencia de la línea en las caches en función de los patrones de acceso.

Cada caso consiste de dos operaciones de acceso a memoria que pueden ser lecturas o escrituras. Las flechas representan la transición de un estado a otro donde se identifica con el número (1) la transición provocada por el acceso del primer thread, y con el número (2) las transiciones provocadas por el acceso del segundo thread.

Se asume que ante el primer pedido de acceso a memoria, la línea no se encuentra en ninguna cache. En consecuencia, dado que el primer acceso en los casos 0 y 1 es una

lectura, ocurrirá un cambio de estado (I) → (E) en la cache del primer procesador. Si el segundo acceso es una lectura, la línea en la cache del segundo procesador cambiará (I) → (S) y (E) → (S) en la cache del primer procesador. Si el segundo acceso es una escritura, la línea en la cache del segundo procesador cambiará (I) → (M) y (E) → (I) en la cache del primer procesador.

Por otro lado el primer acceso en los casos 2 y 3 se trata de una escritura. Esto provocará un cambio de estado (I) → (M) en la cache del primer procesador. Si el segundo acceso es una lectura, la línea en la cache del segundo procesador cambiará (I) → (S) y (M) → (S) en la cache del primer procesador. Si el segundo acceso es una escritura, la línea en la cache del segundo procesador cambiará (I) → (M) mientras que en la cache del primer procesador cambiará (M) → (I).

Estos cambios de estado en el protocolo de coherencia cache provocan eventos que pueden ser capturados en los contadores hardware.

Selección de eventos candidatos

Una vez comprendidos los cambios de estado en el protocolo de coherencia cache que permiten identificar la ocurrencia de alguno de los patrones de acceso, se volvió a revisar la lista de eventos. En particular el evento MEM_UNCORE_RETIRED:LOCAL_HITM, indica en su descripción que cuenta instrucciones *loads* que aciertan en datos en estado modificado en un *sibling core* (otro núcleo en el mismo procesador). También lo clasifica como un evento preciso, por lo que se asume que el número informado es muy cercano a la cuenta real de eventos. Dado que este evento parece indicar que ocurre luego del *read* del patrón de acceso 2, se lo seleccionó como evento candidato.

Con respecto a los patrones de acceso 1 y 3, no se encontraron eventos en la arquitectura de pruebas que describa su ocurrencia. Aún así la ocurrencia del patrón de acceso 2 es un claro indicador de que se están accediendo a datos compartidos entre los threads, y más importante aún, que esos accesos forman parte de un interleaving no serializable.

Con respecto al interleaving no serializable de caso 6, no genera eventos del tipo seleccionado. Sin embargo, tal como se muestra en el experimento de la Sección 5.6.4, esto no parece ser un problema: la frecuencia de este caso en la mayoría de las aplicaciones es inferior al 3 % (con respecto al total), y a su vez ocurre en medio de interleavings de los otros casos que sí desencadenan el evento.

Tiene sentido pensar que los interleavings que modifican datos compartidos poseen localidad temporal: por ejemplo si la operación que desencadenó el evento en una aplica-

ción real corresponde a una operación de sincronización, seguramente es porque el thread pretende leer/modificar uno o varios datos compartidos. Esta relación también es válida para otras plataformas donde no se encuentre un evento que describa el patrón de acceso 2 pero si se encuentre un evento que describa los patrones 1 y 3.

5.5. Microbenchmarks representativos de eventos

Una vez identificado un evento candidato, es necesario asegurar que se comprende con precisión la situación que desencadena el evento. Para ello se diseñaron programas que simulan escenarios donde el evento a validar pueda ser desencadenado a voluntad. Estos programas están diseñados para tener control sobre el número de eventos teóricos que deberían ocurrir: de esta manera la validación del evento consiste en comprobar que la diferencia entre el número de eventos teóricos y el número de eventos reales es poco significativa. Otra ventaja importante de este método de validación es que permite conocer el grado de precisión con que es posible aproximarse al evento: en la medición podrían aparecer eventos generados por la herramienta de medición en lugar de la aplicación objetivo.

Para verificar que los eventos candidatos se pueden usar para distinguir entre diferentes patrones de acceso, se diseñaron los micro-benchmarks de la Figura 5.4.

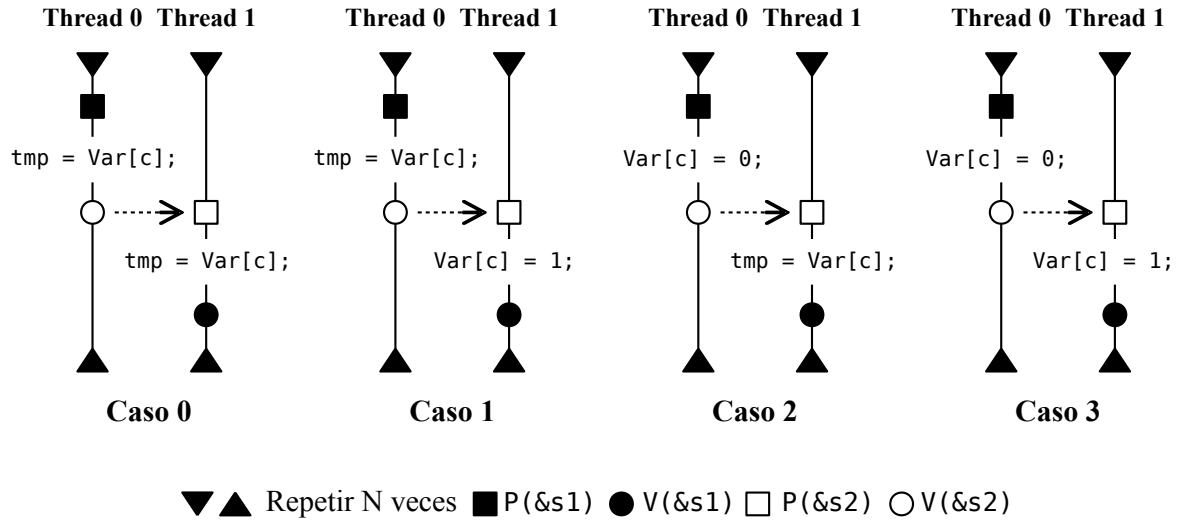


Figura 5.4: Patrones de interacción en memoria entre dos threads

Cada caso corresponde a un programa desarrollado con la librería `Pthreads`, en que se repite la interacción simulada entre dos threads `N` veces (`N` es un parámetro del programa).

Cada thread posee una variable local llamada `tmp` utilizada para simular la lectura. La variable compartida está representada por un arreglo `Var` de `N` elementos, inicializado por el thread principal del programa. Para evitar que alguna optimización sobre la librería `Pthreads` en el uso de semáforos pueda entorpecer el análisis de los resultados, se optó por implementar las funciones `P()` y `V()` con variables compartidas. Las variables `s1` y `s2` cumplen la función de semáforos binarios y se encuentran inicializadas en `1` y `0` respectivamente. En el Listado 5.1 se puede ver el pseudocódigo de estas operaciones.

Listado 5.1: Pseudocodigo de las operaciones P y V. La estructura cache_line completa la línea de cache con un arreglo de char.

```
1  typedef struct
2  {
3      int value;
4      char pad [CACHE_LINE_SIZE - sizeof(int)];
5  } cache_line;

7  P (cache_line *s)
8  {
9      while (*s.value <= 0)
10         ; // esperar...
11     *s.value--;
12 }

14 V (cache_line *s)
15 {
16     *s.value++;
17 }
```

Se debe notar que en esta implementación de `P()` podría ocurrir una violación de atomicidad si entre el `while` y el decremento de `s` ocurriera un `V()`. Esta situación podría ocurrir si el `thread 1` iterara lo suficientemente rápido como para ejecutar un segundo `V(&s1)` antes que el `thread 0` completara el decremento de esta variable. Sin embargo, como antes de que el `thread 1` ejecute el siguiente `V(&s1)` deberá esperar en `P(&s2)` a que el `thread 0` ejecute `V(&s2)`, la violación no ocurrirá nunca. En consecuencia, aunque las operaciones `P()` y `V()` aquí descritas son útiles para este trabajo no son seguras y no deben ser utilizadas en otras aplicaciones.

Debido a que los eventos cambian el estado de la línea completa de cache, se diseñó una estructura que ocupa la línea completa para las variables `s1`, `s2` y los elementos del

arreglo `Var`. Esto permite analizar los cambios de estado asociados a las operaciones de lectura/escritura sobre estas variables.

5.6. Resultados experimentales del capítulo

En esta sección se presentan resultados parciales propios de los experimentos derivados de este capítulo. Los experimentos se realizaron en tres etapas:

- *Validación del evento candidato.*
- *Indicador de accesos inseguros.*
- *Patrones de accesos a datos compartidos.*

5.6.1. Validación del evento candidato

Los experimentos se diseñaron para demostrar que el evento seleccionado puede detectar la ocurrencia del patrón de acceso 2, el cual permitirá detectar interleavings no serializables de los casos 2, 3 y 5. Dado que los micro-benchamrks explicados anteriormente usan una variable compartida para sincronizar los threads, es de esperar (en promedio) que un evento ocurra para cada iteración (se debe recordar que cada evento se genera con una operación de lectura en un procesador, luego de una escritura en la cache de otro procesador). La única diferencia se encuentra en el caso 2, donde el `thread 1` debe provocar un evento extra después de cada iteración causado por la operación de lectura luego de la operación de escritura en la variable `Var[c]`. Los resultados de los experimentos se pueden observar en la Tabla 5.3.

Cada micro-benchmark fue ejecutado con diferentes tamaños de N para mostrar que la relación entre eventos teóricos y reales es independiente del tamaño del problema. Las filas de la tabla muestran la cuenta del evento para el thread 0 y el thread 1 en cada patrón de acceso por cada tamaño de problema, y calcula la relación entre esos valores y el tamaño del problema (celdas resaltadas en gris). Los resultados fueron redondeados a dos decimales. Se debe notar que cada patrón de acceso exhibe el mismo comportamiento independientemente del tamaño del problema. En todos los casos, con cada iteración del bucle el evento es generado una vez. Esto puede ser explicado a partir de la Figura 5.4 y el Listado 5.1. Para operaciones en las que la actividad de lectura/escritura ocurren en el orden apropiado, los threads son sincronizados usando dos variables compartidas, `s1` y

Cuadro 5.3: Resultados de evaluar cada patrón de acceso entre dos hilos para diferentes tamaños de problema.

Kernel	**Thread**	1000		10000		100000		1000000	
case0	thr0	1153,5	1,15	10146,8	1,01	100265,1	1,00	1001851,2	1,00
	thr1	1137,6	1,14	10123,9	1,01	100128,2	1,00	999845,9	1,00
case1	thr0	1149,2	1,15	10173,8	1,02	100265,5	1,00	1000341,8	1,00
	thr1	1119,9	1,12	10145,1	1,01	100331,9	1,00	1001140,3	1,00
case2	thr0	1320,6	1,32	10202,4	1,02	91282,8	0,91	1011695,3	1,01
	thr1	2145,7	2,15	20149,6	2,01	199652,5	2,00	1992097,5	1,99
case3	thr0	1158	1,16	10149,1	1,01	100297,2	1,00	1001510,2	1,00
	thr1	1150,7	1,15	10231,1	1,02	100400,3	1,00	1001767,7	1,00

`s2`. Por ejemplo, el `thread 1` permanece leyendo `s2` hasta que `thread 0` incrementa su valor (observar la flecha en la Figura 5.4).

Debido a que la política de escritura es write-allocate, el procesador que ejecuta el `thread 0` trae la variable `s2` a su propia cache. En ese momento el estado de coherencia de esa línea es actualizado a (M). Luego, el `thread 1` lee la línea que se encuentra en estado (M) en la cache del procesador donde se está ejecutando el `thread 0`, provocando que el evento ocurra.

La misma explicación se aplica al `thread 0`: como se mencionó anteriormente, se puede ver que para el caso 2 en el `thread 1` ocurre un evento extra con respecto a cada iteración, mostrando que el evento puede ser usado para distinguir el patrón de acceso 2 de los otros. En conclusión, el evento seleccionado sirve para identificar tres de los cuatro interleavings no serializables.

5.6.2. Indicador de accesos inseguros

La hipótesis es que el overhead puede ser reducido si la monitorización se restringe a las regiones de código que son inseguras. Aunque es bien conocido que el número de operaciones de acceso a datos compartidos suele ser una fracción pequeña del número total de accesos, este experimento se diseñó para determinar si la técnica propuesta podría mejorar el rendimiento de herramientas de detección de errores de concurrencia. Para ello primero se calculó la relación entre el número de accesos inseguros y el total de accesos. Un acceso inseguro es un patrón de acceso que desencadena el evento. Por lo tanto, una aplicación con un indicador de accesos inseguros (IAI) cercano a cero tendrá un alto

potencial porque esto significa que hay pocos accesos no serializables, mientras que una aplicación con un IAI cercano a uno tiene un bajo potencial.

Se diseño una capa de software ubicada entre la aplicación objetivo y el hardware. Básicamente, esta herramienta detecta cuando un nuevo hilo es creado y configura su afinidad a un núcleo específico (se asegura que cada hilo será ejecutado en un núcleo diferente para aprovechar los cambios de estado en el protocolo de coherencia cache) e inicia un contador de hardware configurado para el evento seleccionado. La herramienta también mantiene un contador software para las instrucciones del programa. Este puede clasificar las instrucciones en lecturas, escrituras, saltos y otros. Esta información será suficiente para conocer el IAI de la aplicación objetivo. La herramienta tiene dos modos de operación:

1. Mostrar estadísticas recolectadas cuando la aplicación finaliza.

2. Mostrar estadísticas parciales recolectadas cada un número predefinido de instrucciones.

La herramienta de monitorización fue usada con cada aplicación de la suite de benchmarks SPLASH-2. Sólo se consideraron las instrucciones de lectura y escritura, ya que el método de detección se basa en análisis de interleavings y para detectar errores no requiere de otro tipo de instrucciones. Los resultados se muestran en la tabla 5.4. En la mayoría de los casos, el IAI es menor que 1 %. Esto significa que el 99 % de las operaciones de lectura y escritura no provocan el evento monitorizado y, en consecuencia, se puede estar seguro de que no hay interleavings no serializables de los casos 2, 3 o 5 en esas regiones de código. Se debe notar que el IAI no es un indicador de número de instrucciones que acceden a datos compartidos, solamente sirve como una medición del número de interleavings no serializables. El programa puede tener un alto porcentaje de operaciones de lectura y escritura a variables compartidas, pero esos accesos pueden ser serializables o interleavings del caso 6.

5.6.3. Patrones de accesos a datos compartidos

Como se mencionó en la sección anterior, el IAI no es suficiente para saber si el enfoque propuesto en este trabajo podrá mejorar el rendimiento de las herramientas de detección. Para saber si la herramienta de monitorización puede ser desactivada es necesario conocer cómo se distribuyen los eventos a lo largo del código. Los experimentos de esta sección

Cuadro 5.4: Comparación de tiempos de ejecución de diferentes granularidades de instrumentación. Las aplicaciones corresponden a la suite SPLASH-2.

	Thread 0			Thread 1		
programa	***Eventos***	***RD+RW(M)***	***IAI***	***Eventos***	***RD+RW(M)***	***UAI***
barnes	471613	578,21	0,0008	479037	575,49	0,0008
cholesky	349135	116,02	0,003	274857	83,59	0,0033
fft	52355	5,26	0,0099	38218	4,01	0,0095
fmm	274997	221,66	0,0012	235089	220,14	0,0011
lu_cb	46271	80,94	0,0006	39468	72,42	0,0005
lu_ncb	33724	1,36	0,0249	27294	1,16	0,0235
ocean_cp	554190	81,2	0,0068	528390	81,32	0,0065
ocean_ncp	514282	74,55	0,0069	461943	74,59	0,0062
radiosity	886884	298,43	0,003	903262	296,89	0,003
radix	51664	14,46	0,0036	46883	14,47	0,0032
raytrace	563691	243,62	0,0023	586121	141,93	0,0041
volrend	304107	1670,51	0,0002	737558	472,17	0,0016
water_nsq	189560	99,02	0,0019	155233	97,01	0,0016
water_sp	254036	88,95	0,0029	140784	88,83	0,0016

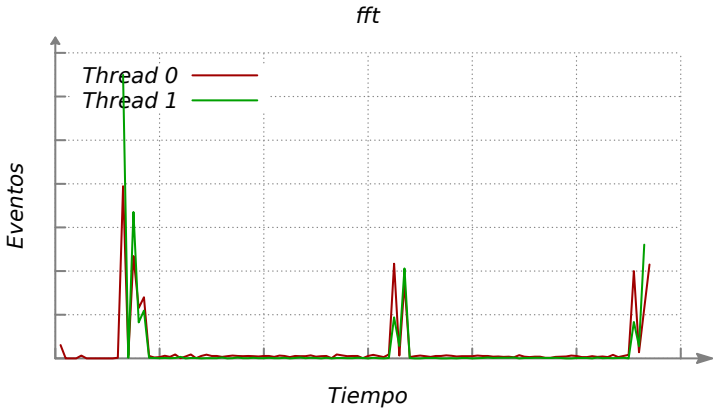
se realizaron con el modo de operación 2 definido en la sección anterior. Este modo de operación permite conocer la distribución de eventos a lo largo de la ejecución del programa. La herramienta fue configurada para tomar muestras cada 100 instrucciones aproximadamente para cada aplicación. Para ello tiene un conjunto de contadores software para cada thread; cada vez que una instrucción es ejecutada, el contador correspondiente es incrementado. Cuando un nuevo thread es creado, su contador se inicializa con la cuenta parcial del thread 0. Cuando el contador alcanza el número de instrucciones definido para la muestra, se lee el contador hardware y se guarda con el contador software en un archivo. el contador hardware se pone a cero y el proceso continúa hasta que la aplicación finaliza.

Las Figuras 5.5, 5.6 y 5.7 muestran la distribución para tres aplicaciones de la suite de benchmarks SPLASH-2. Se eligieron esos casos porque son representativos del resto. En el Anexo C se pueden consultar las figuras correspondientes a todas las aplicaciones de la suite. El eje horizontal corresponde al número de instrucciones ejecutadas, y el eje vertical a la cuenta de eventos. Los threads 0 y 1 están representados por las líneas de color verde y rojo. Las leyendas de los ejes horizontal y vertical fueron intencionalmente dejadas en blanco porque los números grandes hacen difícil la lectura de la figura. Como se puede notar, se encontraron tres comportamientos diferentes entre los programas:

- En *fft*, cada thread tiene varios picos del evento mayores que cero. Se debe notar también que hay una cantidad significativa de código que puede ser ejecutada sin instrumentación: las regiones de código entre los picos que tienen una cuenta de cero eventos. Este tipo de comportamiento es ideal para la propuesta de desactivar

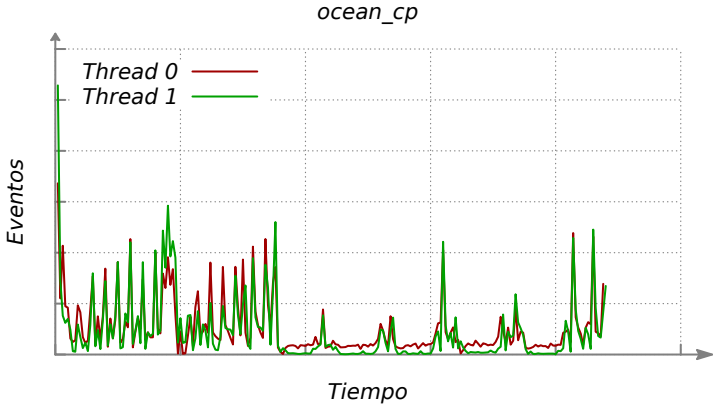
la rutina de análisis, ya que entre los picos hay una gran cantidad de accesos donde es seguro desactivar la herramienta. *fft* y *lu_ncb* se comportan en la misma forma.

Figura 5.5: Distribución ideal de interleavings no serializables.



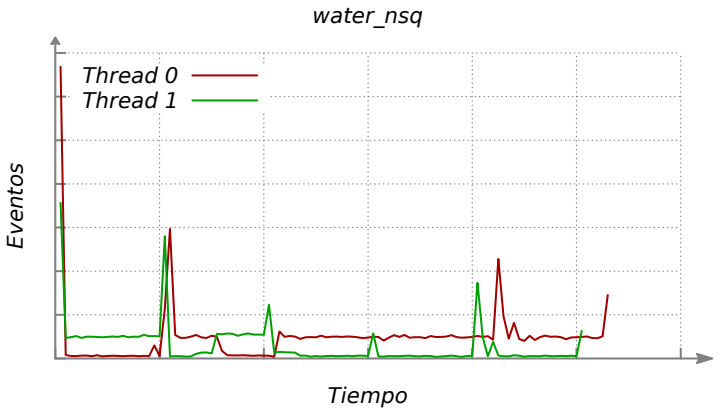
- En *ocean*, parece haber interacciones significativas entre los threads. Este tipo de comportamiento podría perjudicar el desempeño de la propuesta porque en una aplicación con este patrón podría causar que la herramienta de detección constantemente cambie entre los estados activado y desactivado. Las dos versiones de *ocean*, *radiosity*, *raytrace* y *volrend* mostraron este comportamiento.

Figura 5.6: Peor caso de distribución de interleavings no serializables.



- *water_nsq* presenta una situación interesante: aunque el programa registra cuentas de evento superiores a cero todo el tiempo, hay un patrón en la forma en que los eventos ocurren. La mayor parte del tiempo, la cuenta es menor que 300. Esta clase de comportamiento es probablemente causado por interleavings no serializables intencionales, utilizados por el programador para sincronización del código. Este tipo de comportamiento demuestra regularidad en la ocurrencia de eventos, por lo que es de esperar que los eventos son generados por mecanismos de sincronización. En este caso sería posible definir un grado de tolerancia para poder desactivar la herramienta cuando la cuenta de eventos esté por debajo del valor fijado. *barnes*, *cholesky*, *fmm*, *lu_cb*, *radix* y ambas versiones de *water* tienen este comportamiento.

Figura 5.7: Caso especial de distribución de interleavings.



Se debe notar que un programa puede mostrar uno o todos esos comportamientos en diferentes ejecuciones. En el Capítulo 7 se analiza el impacto que puede tener estos patrones de comportamiento en el rendimiento de la herramienta de detección utilizada como caso de estudio.

5.6.4. Frecuencia y distribución de interleavings no serializables

Aunque la distribución de eventos del experimento anterior se puede relacionar con la distribución de interleavings no serializables, es útil conocer esta distribución a partir de lo que puede detectar AVIO. Además, dado que el evento seleccionado ocurre con los casos de interleavings no serializables 2, 3 y 5 pero no el caso 6, es necesario estimar el impacto que podría tener usar sólo este evento en la capacidad de detección de la herramienta.

Para ello se desarrolló una versión modificada de AVIO que genera un reporte de interleavings detectados a intervalos regulares de tiempo. El valor del intervalo se fijó en 10ms, y el experimento se repitió 10 veces con cada aplicación de la suite SPLASH-2. En este caso no se usó el contador hardware ya que el objetivo es conocer el porcentaje de interleavings de cada caso detectados por AVIO sobre el total y su distribución en la ejecución del proceso. La Tabla 5.5 muestra los resultados del experimento.

Cuadro 5.5: porcentaje de interleavings no serializables por aplicación.

programa	***Caso 2***	***Caso 3***	***Caso 5***	***Caso 6***	***Total***
barnes	51 %	36 %	12,4 %	0,6 %	7911
cholesky	0,1 %	99,9 %	–	–	29987
fft	33,4 %	–	66,6 %	–	98253
fmm	21,9 %	35,6 %	39,8 %	2,7 %	62025
lu_cb	27,2 %	72,8 %	–	–	136
lu_ncb	2,7 %	6,5 %	90,8 %	–	414
ocean_cp	30,1 %	68,4 %	–	1,5 %	2166
ocean_ncp	26,7 %	71,8 %	–	1,5 %	2216
radiosity	8,7 %	83 %	7,5 %	0,8 %	129367
radix	0,3 %	0,6 %	99,2 %	–	3099
raytrace	0,4 %	99,6 %	–	–	57455
volrend	0,8 %	95,7 %	2,7 %	0,8 %	7511
water_nsq	28,5 %	71,5 %	–	–	32266
water_sp	97,7 %	2,3 %	–	–	2374

Es importante destacar que la columna *Total* se refiere al número total de interleavings no serializables que ocurrieron durante la ejecución. Este valor no debe ser confundido con el número total de interleavings reportados por la versión original de AVIO: en ese reporte se omiten los interleavings repetidos, es decir, sin importar cuantas veces se repita un interleaving, siempre que los tres accesos sean los mismos, ese interleaving cuenta sólo como uno en la cuenta total de AVIO. Los valores de las celdas corresponden a los valores máximos detectados entre las 10 ejecuciones. Para facilitar su interpretación están expresados como la fracción de cada caso sobre el total, y se redondearon a un decimal.

Se debe notar que con respecto a los interleavings de caso 6, en 8 aplicaciones no se detectó ningún caso, en 3 aplicaciones representan menos del 1 % (*barnes*, *radiosity* y *volrend*) y en 3 aplicaciones se ubican entre el 1 % y el 3 % (*fmm*, *ocean_cp* y *ocean_ncp*). Estos resultados indican que los casos de tipo 6 son muy poco frecuentes.

Sin embargo, es interesante analizar la distribución de estos interleavings de caso 6 con respecto al resto. La Figura 5.8 corresponde a la distribución de interleavings de la aplicación *fmm*, y la Figura 5.9 corresponde a la distribución de la aplicación *ocean_cp*.

El resto de figuras pueden ser consultadas en el Anexo D. Es interesante destacar que los interleavings de caso 6 no ocurren de manera aislada. Esto significa que aunque el

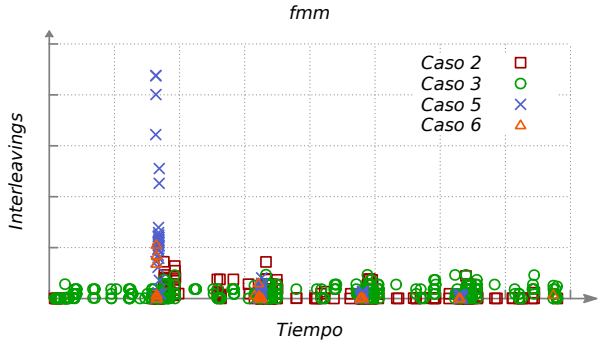


Figura 5.8: distribución de interleavings no serializables para fmm.

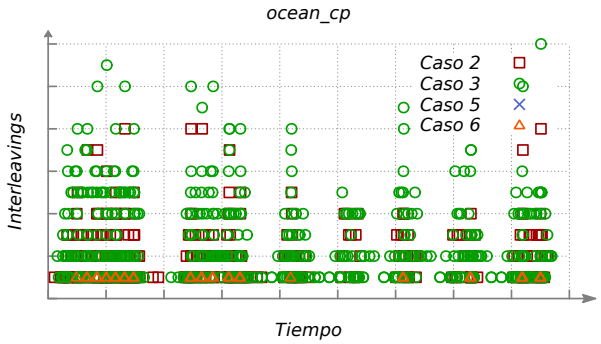


Figura 5.9: distribución de interleavings no serializables para ocean_cp.

interleaving de caso 6 no desencadena eventos del patrón de acceso 2, es muy probable que otras instrucciones que comparten localidad temporal con este provoquen interleavings de los otros casos, y en consecuencia el evento ocurra. En conclusión, el evento seleccionado es un buen indicador de interleavings no serializables.

5.7. Resumen

Para determinar si es posible reducir el overhead de la instrumentación a nivel de instrucción restringiéndolo sólo a los accesos a memoria compartida, es necesario determinar si es posible detectar el momento preciso en que esos accesos ocurren. La mejor opción para detectar este momento es cuando ocurre algún cambio en la memoria cache compartida entre los núcleos que ejecutan los procesos.

Una herramienta muy útil para esta tarea son los *contadores hardware*, un conjunto de registros especiales disponibles en todos los procesadores actuales. Esos registros pueden ser programados para contar el número de veces que un evento ocurre dentro del procesador durante la ejecución de una aplicación. Los eventos proveen información sobre diferentes aspectos de la ejecución de un programa (por ejemplo el número de instrucciones ejecutadas, el número de fallos en cache L1 o el número de operaciones en punto flotante ejecutadas). El soporte hardware/software para el uso de los contadores evolucionó a través de los años permitiendo en la actualidad medir cientos de eventos, ya sea a nivel de usuario (con aplicaciones como perf) o a nivel de fracciones de código (con librerías como PAPI o perf_events).

Sin embargo, aún con el soporte hardware/software actual, para usarlos es necesario tener un amplio conocimiento sobre la arquitectura del procesador. Los procesadores ofrecen cientos de eventos para medir distintos aspectos de la ejecución, pero sólo un pequeño conjunto de estos denominados *predefinidos* o *predeterminados* están presentes en todas las arquitecturas. El resto puede variar de una microarquitectura a la siguiente, o incluso entre distintos procesadores que comparten la misma microarquitectura. Para dificultar aún más las cosas la información que proveen los fabricantes suele ser escasa, con descripciones poco significativas y confiables. En consecuencia no es posible determinar si un evento en particular servirá para los propósitos de este trabajo sólo con leer los manuales del fabricante: es necesario un método de validación que demuestre que los eventos candidatos pueden ser utilizados.

Debido a que la detección de acceso a un dato compartido recién se hace evidente en el primer nivel de cache compartido entre núcleos, inicialmente se buscaron eventos asociados a la cache que indicaran interleavings. Es interesante notar que un interleaving se puede dividir en dos patrones de acceso: el primero entre el primer acceso del thread 0 seguido por el acceso del thread 1, y el segundo entre el acceso del thread 1 seguido por el segundo acceso del thread 0. Al analizar los casos de interleavings, fue evidente que es posible representar los 8 casos con 4 patrones de acceso a memoria. Esto es importante porque mientras un interleaving es un concepto de software, los patrones de acceso ya han sido analizados para resolver problemas en hardware en el protocolo de coherencia cache.

La búsqueda de eventos asociados a los cambios de estado en el protocolo de coherencia cache reveló que para la arquitectura de pruebas existe un evento llamado MEM_UNCORE_RETIRED:LOCAL_HITM, cuya descripción indica que ocurre con uno de los cuatro patrones de acceso. El patrón asociado al evento está presente en tres de los cuatro casos de interleavings no serializables. Para asegurar que el evento puede ser utilizado para detectar interleavings no serializables, se diseñaron 4 kernels de ejecución que simulan los patrones de acceso. Estos kernels permiten establecer el número de eventos teóricos que deberían ocurrir en cada ejecución. Luego se diseñó una herramienta para medir el número de eventos reales generados por cada ejecución de cada kernel. A partir de comparar los resultados con distinta cantidad de eventos teóricos y eventos reales medidos por la herramienta, se concluyó que el evento efectivamente ocurría con precisión con uno de los patrones de acceso, y en consecuencia puede detectar la ocurrencia de tres de los cuatro casos de interleavings no serializables.

Para completar el trabajo experimental del capítulo, se analizaron tres aspectos de las aplicaciones que podrían condicionar el éxito de la propuesta: se estimó el porcentaje de accesos a memoria en que ocurren eventos, se analizó la distribución de los eventos durante la ejecución y se analizó el impacto que podría tener en la capacidad de detección de AVIO el hecho de que el evento seleccionado no ocurre con el interleaving no serializable de caso 6. Los resultados mostraron que los accesos inseguros ocurren en una fracción muy pequeña del código (inferior al 1 % en la mayoría de los casos). En cuanto a la distribución de eventos, se encontraron tres patrones, donde uno por ser muy irregular durante toda la ejecución podría perjudicar el desempeño de la propuesta de mejora. Finalmente, se demostró que los interleavings de caso 6 ocurren con muy poca frecuencia, y cuando ocurren siempre son en medio de interleavings de los otros casos (2, 3 y 5). Esto indica que aunque no sirve para detectar directamente este caso, por su localidad temporal con interleavings de los otros casos es muy probable que la herramienta sea activada y en consecuencia el caso sea detectado.

Por todo lo anterior, se concluye que el evento seleccionado sirve para detectar interleavings no serializables y que puede ser utilizado para detectar momentos en la ejecución donde es apropiado activar la herramienta de detección.

Capítulo 6

Instrumentación dinámica guiada por eventos

6.1. Introducción

Este capítulo comienza con una introducción al tema de muestreo de eventos (6.2). Luego presenta un modelo simple de instrumentación dinámica (6.3). Posteriormente se extiende el modelo simple para trabajar con aplicaciones paralelas (6.4). Finalmente, se presenta el modelo completo de instrumentación dinámica guiada por eventos de hardware (6.5).

6.2. Muestreo de eventos

Los contadores pueden ser configurados en dos modos de operación: 1) *contar eventos* y 2) *muestreo de eventos*. El primer modo de operación es el modo por defecto y el que normalmente utilizan las aplicaciones como perf, PAPI, etc. Todos los experimentos del Capítulo 5 se realizaron con contadores configurados para contar eventos. En este modo cada vez que ocurra un evento del tipo seleccionado, la PMU (Performane Monitoring Unit) incrementará el valor del registro correspondiente al contador. Al final el resultado se puede obtener simplemente leyendo el valor del registro.

El segundo modo de operación llamado muestreo de eventos, permite configurar la PMU para generar una señal cada N eventos, donde N debe ser especificado por el usuario. Esta segunda forma de utilizar los contadores es de interés para este trabajo de tesis ya que la señal generada por la PMU (Unidad de Monitorización de Performance) de los

procesadores puede ser utilizada para modificar el comportamiento o tomar decisiones en tiempo de ejecución. Esto permite construir algoritmos dinámicos de gran precisión, que se ajustan a los eventos que ocurren en el hardware (por ejemplo algoritmos de planificación o balance de carga). En este capítulo se propone un modelo que integra conceptos de instrumentación dinámica con este modo de operación de los contadores para determinar cuándo activar las rutinas de análisis.

6.3. Modelo Simple

Se puede pensar en un modelo simple donde una aplicación debe alterar su comportamiento si ocurre un determinado evento de hardware: accesos a memoria, número de fallos de cache, cantidad de operaciones en punto flotante, predicciones de saltos incumplidas, etc. Para conseguirlo es suficiente configurar un contador en modo de muestreo y agregar el código para gestionar la señal en la aplicación. El Listado 6.1 muestra un prototipo para esta configuración.

Listado 6.1: Configuración y gestión de señales generadas por un contador en modo de muestreo.

```
1  // Código para alterar comportamiento del programa
2  void eventListener (int sig){...}

4  int main(){
5      create_eventset (&tEventSet);
6      sample_event (tEventSet, COD_EVENT, N, SIGUSR1);
7      signal (SIGUSR1, eventListener);

9      start(tEventSet);
10     ... // codigo del programa
11 }
```

Las funciones `create_eventset` y `start` (líneas 5 y 9) son abstracciones para inicializar el contador hardware y comenzar a contar eventos. La mayoría de las librerías que proveen acceso a los contadores poseen funciones similares. La función `sample_event`[1] (línea 6) se utiliza para configurar el contador en modo de muestreo. A esta función hay

[1]Para una explicación detallada sobre cómo configurar un evento en modo de muestreo se puede consultar la página del manual disponible en (Weaver, 2014).

que indicarle el contador a utilizar (`tEventSet`), un valor en hexadecimal que representa el evento que será utilizado para el muestreo (`COD_EVENT`), la cantidad de eventos que deben ocurrir antes de generar la señal (`N`) y el código de señal que debe enviar la PMU (`SIGUSR1`). Finalmente, la función `signal`[2] (línea 7) configura la función `eventListener` como el manejador para la señal `SIGUSR1`.

Señales. Cada señal tiene un procedimiento predefinido en el sistema operativo que es heredado por cada proceso cuando se instancia. Este procedimiento denominado acción (*Action*), determina cómo se debe comportar el proceso en caso de que una señal le sea entregada. Existen 5 acciones predefinidas:

- *Term*. El proceso debe terminar.
- *Ign*. La señal debe ser ignorada.
- *Core*. El proceso debe terminar y hacer un volcado de memoria.
- *Stop*. El proceso debe ser suspendido temporalmente.
- *Cont*. El proceso debe continuar con su ejecución (si fue suspendido previamente).

Un programador puede cambiar la acción predeterminada para una señal usando las rutinas del sistema `sigaction` o `signal`. Usando estas llamadas al sistema es posible elegir cómo el proceso debe comportarse ante una señal entregada:

- Realizar la acción por defecto para el código de señal recibido.
- Ignorar la señal y por lo tanto no realizar ningún cambio.
- Capturar la señal con un *signal handler* (manejador de señales) y ejecutar una función definida por el programador en lugar de la acción por defecto.

El número de señales disponibles puede variar según la plataforma. Sin embargo, todas las plataformas poseen al menos dos señales para ser utilizadas con fines específicos por los programadores denominadas `SIGUSR1` y `SIGUSR2`.

[2] Aunque las páginas del manual de Linux aconsejan utilizar `sigaction` en lugar de `signal`, conceptualmente `signal` es más simple para el ejemplo. No obstante, en la versión definitiva se aconseja utilizar `sigaction`.

Es importante destacar que aunque los conceptos de señales e interrupciones son similares, existen diferencias importantes y no deben ser confundidos. En adelante, donde se emplee el término *interrupción*, debe ser interpretado como la acción de interrumpir la ejecución de un proceso, y no como una interrupción del hardware.

Es importante destacar que el proceso de sintonización dinámica implica pasar por cuatro fases, conocidas como *instrumentación*, *monitorización*, *análisis* y *modificación* o *sintonización* (Morajko, 2003). El prototipo del Listado 6.1 refleja sólo las dos primeras fases del proceso. La rutina de análisis y la modificación de parámetros críticos que afecten el rendimiento de la aplicación son específicos del problema de sintonización que se esté tratando, y corresponden al código de `eventListener`. En el Capítulo 7 se presenta un caso específico que implementa las cuatro fases para reducir el tiempo de ejecución de una herramienta para detección de violaciones de atomicidad.

Aunque el código del Listado 6.1 permite alterar el comportamiento de la aplicación en función de un evento, no tiene en cuenta los siguientes aspectos:

1. **Aplicaciones paralelas**. Se debe notar que el modelo propuesto genera señales desde un único contador, o dicho de otra forma, desde un único procesador. En una aplicación paralela probablemente sea necesario tener un contador configurado para cada procesador, ya que el evento de la ejecución que se desea detectar/controlar puede ocurrir en cualquiera de los procesadores donde se esté ejecutando un hilo.

2. **Gestión de múltiples señales**. ¿Qué se debería hacer ante múltiples señales generadas por el contador? Si la cantidad de eventos requeridos para generar la señal es pequeña, probablemente interrumpa la ejecución del proceso demasiado seguido. Tal vez se desee que el comportamiento de la aplicación sea alterado en función de una combinación de eventos.

3. **Rendimiento**. El modelo simple no tiene en cuenta los aspectos relacionados al rendimiento de la aplicación: si el número de señales generadas por el contador es elevado o el código del manejador del evento degrada notablemente su eficiencia, entonces se pierden las supuestas ventajas de una sintonización dinámica de la aplicación.

6.4. Modelo para aplicaciones paralelas

De acuerdo a lo expuesto al final de la sección anterior, un modelo completo para sintonización dinámica de aplicaciones guiada por contadores de hardware debe: contemplar que la aplicación objeto de estudio esté paralelizada, especificar qué hacer ante múltiples señales y considerar que la gestión de éstas no degrade el rendimiento del programa.

El primer aspecto a considerar es la configuración de los contadores cuando la aplicación posee más de un proceso. Se debe decidir si el contador estará asociado al proceso, al procesador o a una combinación de ambos. La interfaz de Linux `perf_events` permite configurar la PMU especificar qué proceso/s y/o qué procesador se quiere monitorizar. Los modos disponibles son:

1. Contar eventos en el proceso que configura el contador en cualquier procesador (si el proceso se ve obligado a cambiar de procesador durante la ejecución, la cuenta continúa en el nuevo procesador).

2. Contar eventos en el proceso que configura el contador sólo cuando se ejecute en un procesador determinado (si el proceso cambia de procesador, la cuenta no continúa).

3. Contar eventos en un proceso distinto al que configura el contador en cualquier procesador (mismo comportamiento que en el primer modo).

4. Contar eventos en un proceso distinto al que configura el contador sólo cuando se ejecute en un procesador determinado (mismo comportamiento que en el modo 2).

5. Contar eventos en un procesador determinado sin importar qué procesos se ejecuten en él.

De todas las configuraciones posibles, la opción 1) permite adaptar el modelo simple sólo asegurando que cada proceso configurará su propio contador.

En segundo lugar, se debe decidir qué hacer ante múltiples señales, ya sean provocadas por el mismo evento o por varios. En el modo de operación de muestreo de eventos, el contador continuará generando señales de acuerdo a la configuración realizada por el usuario. Por ejemplo si el contador se configuró con un valor de `N=1` (ver Listado 6.1) y ocurren 10 eventos seguidos en la aplicación, entonces el proceso será interrumpido 10 veces. El modelo propuesto utiliza la señal generada por el contador para activar una rutina de análisis que alterará el comportamiento del programa. En este contexto, cualquier señal posterior a la primera sería redundante (la rutina de análisis ya habría

sido activada con la primera señal)[3]. Además, las siguientes señales a la primera podrían interrumpir la rutina de análisis en un momento poco oportuno, creando situaciones no contempladas por el programador.

La forma de lidiar con esta situación propuesta en este trabajo consiste en utilizar un recurso que ya está disponible en el sistema de gestión de señales: las siguientes señales a la primera deben ser ignoradas. Para ello, luego de recibir la primera señal `SIGUSR1` del contador de hardware, se debe reemplazar el manejador de señales por la acción `SIG_IGN`. La acción `SIG_IGN` se usa junto a `signal` para indicar al sistema operativo que la señal asociada será ignorada en lugar de gestionada. Sin embargo, esta solución pone de manifiesto otro problema: una vez concluida la rutina de análisis que modificó el comportamiento del programa, ¿debería el proceso ser capaz de capturar nuevas señales generadas por el contador? Afortunadamente la solución a este problema puede ser igualmente simple: una vez concluida la rutina de análisis, se vuelve a restablecer `eventListener` como manejador de señales para *SIGUSR1*. El Listado 6.2 muestra como modificar `eventListener` para conseguir este comportamiento.

Listado 6.2: Configuración del manejador de señales para ignorarlas luego de detectar la primera.

```
1 void eventListener (int sig){
2     signal (SIGUSR1, SIG_IGN);
3     ...
4     signal (SIGUSR1, eventListener);
5 }
```

De los tres aspectos, el más importante es evitar degradar el rendimiento del programa con la gestión de señales. Dado que en el modelo propuesto la gestión de la señal se hace *en el mismo proceso*, es muy probable que terminará atenuando su rendimiento. Por ello, la propuesta divide el modelo en dos procesos con `pid` distintos: uno corresponde al *programa objeto de estudio* (POE), que aún debe ser configurado con los contadores en modo de muestreo. El otro, al que se denominó *gestor de interrupciones* (GI), tiene la única función de recibir las señales y unificar las comunicaciones con el POE. Este enfoque tiene la ventaja de que el GI sólo interrumpe al POE cuando efectivamente es necesario alterar su comportamiento. Asumiendo que el GI puede ser ejecutado en un procesador distinto

[3]Pueden existir situaciones donde sea deseable capturar todas las señales generadas. Sin embargo, en este trabajo se propone un caso más general. Si éste fuera el caso se puede omitir la configuración propuesta como solución.

que el POE, ya sea que el GI ignore las señales o se sature, no afectará el rendimiento del POE.

6.4.1. Gestor de Interrupciones (GI)

El modelo general divide el programa en dos procesos y establece las comunicaciones entre ellos a través de señales. El programa que representa al POE será ejecutado desde GI, con lo que se aprovecha la relación natural entre los procesos para conocer sus respectivos `pid` necesarios para enviar las señales. Tomando en cuenta los aspectos resaltados anteriormente, se extendió el modelo para recibir como argumento el POE e iniciarlo a través de las llamadas del sistema `fork` y `exec`. El Listado 6.3 muestra un prototipo para esta aplicación.

Listado 6.3: Prototipo del programa Gestor de Interrupciones.

```
1  pid_t child;
2  // Código para recibir señales del POE
3  void eventListener (int sig){...}
4  void endHandler (int sig){ exit(0);}

6  int main(int argc, char *argv[]){
7      child = fork();
8      if (child == 0)
9          execvp (argv[1], &argv[1]);
10     else{
11         signal (SIGUSR1, eventListener);
12         signal (SIGCHLD, endHandler);
13         while (1)
14             pause();
15     }
16 }
```

La función `fork` (línea 7) crea un nuevo proceso duplicando el proceso que la ejecuta y guarda distintos valores en la variable `child` según se trate del proceso original o del duplicado. La variable `child` valdrá 0 para el proceso que ha sido creado, en este caso el POE. La sentencia `if` (línea 8) clasifica el código a ejecutar en función del valor de `child`, y la función `execvp` (línea 9) reemplaza la imagen del proceso actual por la del nuevo proceso. Esta función espera como argumentos la ruta al programa que será ejecutado

(`argv[1]`) y la lista de argumentos para ese proceso (`&argv[1]`). El POE podrá obtener el `pid` del proceso GI a través de una llamada a la función `getppid`.

Por otro lado, el proceso original (en este caso GI) recibe en la variable `child` el `pid` del proceso que representa al POE. Este valor será luego utilizado para enviar señales al POE cuando sea apropiado. Se debe configurar el manejador de señales (línea 11), indicando que la función `eventListener` será la encargada de gestionar la señal `SIGUSR1`.

Una vez finalizado el POE el proceso GI debe finalizar también. Afortunadamente, cuando un proceso hijo termina, el sistema operativo envía al proceso padre una señal `SIGCHLD`. El comportamiento por defecto para esta señal es ignorarla, pero en este caso particular es útil para terminar el proceso GI. Para ello se configura el manejador de la señal de finalización `SIGCHLD` (línea 12) para que sea gestionada por la función `endHandler`. Esta función (línea 4) simplemente indica al proceso POE que debe terminar su ejecución.

Finalmente, para asegurar que el proceso quedará a la espera de las señales, se itera sobre la función `sleep` (líneas 13 y 14).

6.4.2. Programa Objeto de Estudio (POE)

Se asumirá que el POE fue desarrollado para el modelo de memoria compartida con la librería `pthreads`. Sin embargo, las ideas aquí expuestas se pueden trasladar a otros modelos sin mayor dificultad. El Listado 6.4 muestra los fragmentos de código que hay que agregar en la aplicación paralela.

Al igual que en el Listado 6.1, las funciones `create_eventset` y `start` (líneas 3 y 5) son abstracciones para inicializar el contador hardware y comenzar a contar eventos. La diferencia con el modelo anterior radica en que cada `thread` inicializará un contador distinto a través del arreglo de contadores `tEventSet`: cada hilo accede a una posición distinta del arreglo usando la variable `id` (se asume que un valor distinto para `id` fue utilizado en la creación de cada hilo). La función `sample_event` (línea 4) fue modificada para recibir el `pid` del proceso que creó a POE, es decir, el `pid` del GI. De esta manera las señales generadas por los eventos serán dirigidas a ese proceso. El resto de los argumentos (`tEventSet`, `COD_EVENT`, `N` y `SIGUSR1`) tienen el mismo significado que en el Listado 6.1. Finalmente, la función `signal` (línea 13) configura la función `handle_GI_signal` como el manejador para la señal `SIGUSR1` proveniente del GI.

Listado 6.4: Prototipo del programa Objeto de Estudio.

```
1  void *eachThreadOfTheProcess (void *arg){
2      int id = (long) arg;
3      create_eventset(&tEventSet[id]);
4      sample_event (tEventSet[id], COD_EVENT, N, getppid(), SIGUSR1);
5      start(tEventSet[id]);
6      ... // Código normal del thread
7  }

9  // codigo para alterar comportamiento del programa
10 void handle_GI_signal(int signum){...}

12 int main(){
13     signal (SIGUSR1, handle_GI_signal);
14     ... // codigo del programa
15 }
```

6.5. Modelo de Instrumentación Dinámica Guiado por Eventos

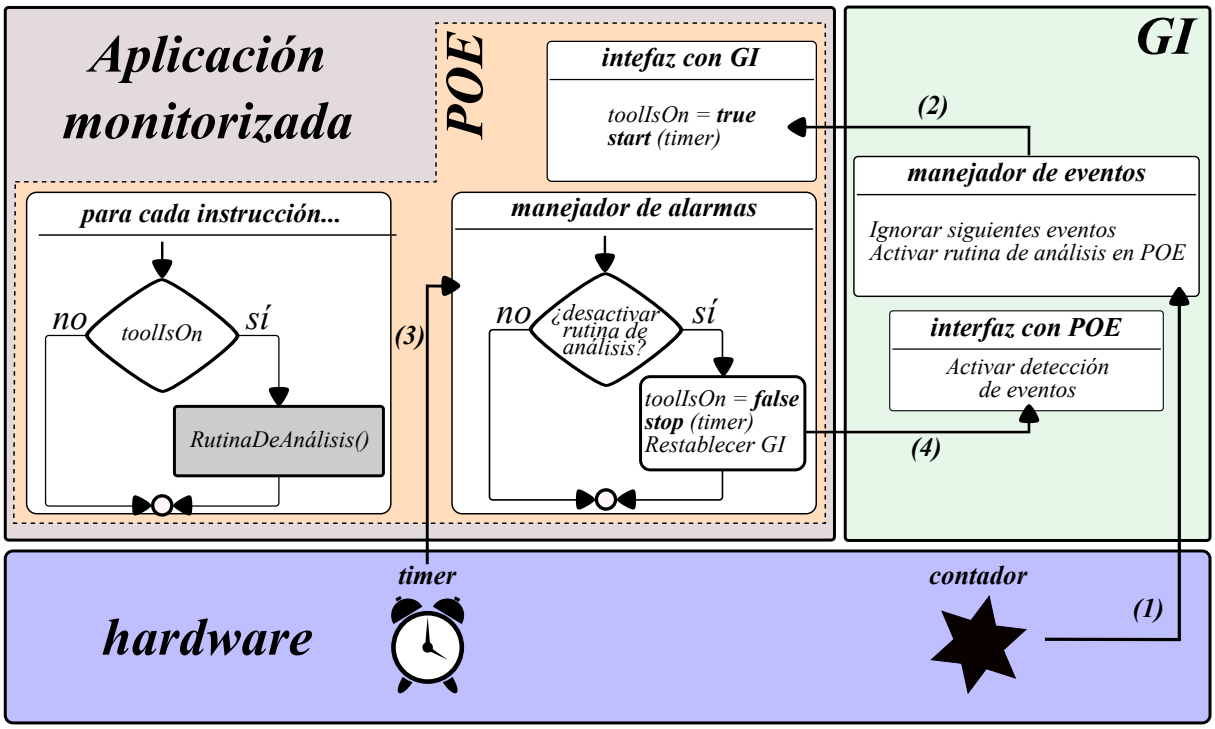
El modelo propuesto para aplicaciones paralelas puede ser empleado en casos donde se desee alterar el comportamiento de la aplicación en función de la ocurrencia de eventos hardware. Sin embargo, se debe notar que requiere acceso al código fuente del programa del POE para hacer las modificaciones necesarias.

En esta sección se extiende ese modelo para ser aplicado en conjunto con herramientas de instrumentación dinámica. Dado que esta tesis está centrada en herramientas de detección de errores de concurrencia y en particular violaciones de atomicidad, este modelo plantea la granularidad de la instrumentación a nivel de instrucción, ya que como se vio en la Sección 4.4.1, el overhead producto de esta granularidad es muy elevado. Sin embargo, para no perder generalidad del modelo, en este capítulo se propone el modelo de instrumentación dinámica guiado por eventos de hardware y se deja para el próximo capítulo su implementación en la herramienta de caso de estudio.

Supóngase una aplicación cualquiera que debe ser monitorizada con una herramienta de instrumentación dinámica. En esta herramienta se utilizó granularidad a nivel de instrucción, por lo que para cada instrucción de la aplicación se incluirá (antes o después de ejecutar la instrucción) una llamada a una rutina de análisis. La rutina aparece en la

Figura 6.1 sombreada en gris, y puede ser desde una operación de incremento como en la Tabla 4.2 o una costosa rutina de análisis de accesos a memoria como la descrita en el Capítulo 3 para detectar violaciones de atomicidad.

Figura 6.1: Diagrama del modelo de instrumentación dinámica guiado por eventos de hardware.



El modelo propone emplear las estrategias de configuración de los contadores de hardware vistas en las secciones anteriores para forzar a que la llamada a la rutina de análisis sólo se ejecute cuando se ha detectado la ocurrencia de un evento del tipo monitorizado. Se debe observar que en la Figura 6.1 la rutina de análisis sólo se ejecuta si el valor del *flag* `toolIsOn` es verdadero: caso contrario, mientras la aplicación esté ejecutando código que no genera eventos, se evita la llamada a la rutina de análisis.

Para conseguirlo se emplea la estrategia propuesta en la Sección 6.4 de dividir la gestión de eventos en dos programas, el POE y el GI. En este caso el POE, destacado con fondo naranja en la Figura 6.1, está integrado en la herramienta de instrumentación.

6.5.1. Activado de la rutina de análisis

Cada vez que se crea un nuevo thread, se configura un contador hardware para generar una señal ante la ocurrencia del evento. Esa señal (1) será enviada al GI (destacado con fondo verde en la Figura 6.1) para ser gestionado por un manejador de eventos. Este manejador configurará a GI de manera que ignore futuras señales generadas por el/los contadores y enviará una señal (2) al POE para activar la rutina de análisis. Mientras tanto, el POE se encuentra ejecutando las instrucciones de la aplicación monitorizada sin pasar por la rutina de análisis, ya que el valor del *flag* `toolIsOn` es falso. Al recibir la señal proveniente del GI (2), el POE cambia el valor de este flag a verdadero e inicia un *timer*[4]. La siguiente sección explica cómo y porqué utilizar este timer para desactivar la rutina de análisis.

6.5.2. Desactivado de la rutina de análisis

En la Sección 6.4, se propone ignorar las señales posteriores a la primera hasta que finaliza la rutina de análisis, momento en el que se vuelve a activar la captura de señales. Sin embargo, se debe notar que en este caso la rutina de análisis deberá ser ejecutada tantas veces como señales deben ser instrumentadas. Si la rutina fuera desactivada y la captura de señales en GI fuera restablecida luego de analizar cada instrucción, entonces probablemente se perderían muchas de las interacciones entre threads que justamente la rutina de análisis se supone que debe monitorizar.

Para el caso particular que se trata en esta sección, la rutina de análisis debe permanecer activa mientras detecte lo que se supone que debe detectar: en el caso de estudio analizado en el siguiente capítulo esto significa dejar activada la rutina de análisis hasta que la herramienta deje de detectar violaciones de atomicidad. Dicho de un modo más general, la rutina debe estar activada hasta que dejen de ocurrir eventos del tipo monitorizado.

Dado que no es posible configurar un contador para enviar una señal ante la *no ocurrencia* de eventos, forzosamente hay que buscar un método alternativo. La propuesta de este trabajo para resolver este problema es emplear un *timer* configurado para interrumpir al POE luego de un intervalo predefinido de tiempo. Luego de transcurrido el intervalo, el sistema operativo enviará una señal `SIGALRM` al POE, quien a su vez tiene un manejador

[4]Término empleado para denominar a un reloj por software que genera una señal dirigida al programa que lo configura luego de una fracción de tiempo.

para esta señal. Este manejador cumple la única función de buscar en las estructuras de datos de la herramienta evidencias de que es seguro desactivar la rutina de análisis. En caso negativo, el *timer* se reinicia automáticamente y la rutina de análisis permanece activada. En caso positivo, el *flag* `toolIsOn` se cambia a falso para evitar ejecutar la rutina de análisis, se detiene el *timer* y se envía una señal (4) a GI para reactivar la detección de eventos.

6.5.3. Aspectos de diseño

En esta sección se discuten aspectos de diseño que pueden afectar al modelo propuesto.

Activación anticipada

Relacionado al aspecto anterior, se debe tener en cuenta que intentar medir en hardware un aspecto de software necesariamente repercute en la precisión de la medición. Por ejemplo, si no se utilizara la propuesta de ignorar las siguientes señales a la primera, el modelo permitiría detectar con precisión la primera señal generada producto de un evento en la ejecución de la aplicación monitorizada. Luego, la rutina de análisis comenzaría a ser ejecutada con cada instrucción de acceso a memoria. Se debe notar que la misma rutina de análisis puede provocar nuevos eventos. En esta situación, sería imposible saber si las siguientes señales entregadas a GI han sido generadas por eventos en el código de la aplicación monitorizada o por eventos en la rutina de análisis. En estas condiciones, aunque se esté ejecutando una fracción del código de la aplicación donde la herramienta debería estar desactivada, el GI podría *activar anticipadamente* la herramienta debido a que detectó un evento producido por la rutina de análisis.

Otra situación común que podría provocar la activación anticipada de la rutina de análisis es la falsa compartición: volviendo al ejemplo del caso de estudio, la rutina de análisis trabaja con granularidad a nivel de variable. Sin embargo, la granularidad del evento que detecta el patrón 2 ($W \rightarrow R$) es a nivel de *línea* de cache. Esto significa que si un proceso acaba de actualizar una variable de una línea de cache y otro proceso lee una variable distinta pero que se encuentra en la misma línea, entonces generará la señal sin importar que el error en realidad no existe. Este problema se conoce como *falsa compartición* y afecta a todas las propuestas que proponen modificaciones en las caches para implementar sus algoritmos. El método más aceptado en la comunidad para lidiar con este problema consiste en aplicar un "relleno" (padding) mediante instrumentación

a cada variable, de manera que cada línea de la cache sea ocupada solamente por una variable. Aunque este método es útil para depuración, claramente sería un problema para una herramienta orientada a entornos de producción ya que degradaría notablemente el rendimiento de la aplicación.

Sin embargo, se debe notar que en el contexto de rutinas de análisis que deben ser activadas con eventos, la activación anticipada no representa un problema. Esta condición podría afectar el rendimiento de la versión que implementa el modelo propuesto, pero no más que tener la rutina activada todo el tiempo. Por otro lado es interesante destacar que no perjudicaría la capacidad análisis de la rutina de instrumentación: la herramienta será activada y la rutina de análisis será ejecutada.

Elección del intervalo de muestreo

En la Sección 6.5.2 se mencionó que no es posible utilizar la información provista por los contadores para desactivar la herramienta y se propuso configurar un *timer* para que genere señales a intervalos regulares de tiempo. El manejador de esta señal revisa si es seguro desactivar la rutina de análisis, actuando en consecuencia.

Elegir el tiempo que debe durar cada intervalo no es un detalle menor. Un intervalo corto permite desactivar la herramienta muy cerca del instante en que los eventos dejan de ocurrir. Por otro lado, si la herramienta se desactiva demasiado pronto, podría comenzar a activarse y desactivarse constantemente, lo cual perjudicaría su rendimiento y capacidad de análisis. Un intervalo largo evita el problema del desactivado temprano, pero si tarda demasiado en ser desactivado penalizaría su rendimiento.

No es posible determinar analíticamente un valor óptimo para el intervalo de muestreo. En cambio, una vez conocida la rutina de análisis y la arquitectura donde la aplicación será ejecutada, si es posible estimar este valor experimentalmente. La solución propuesta en este trabajo consiste en utilizar la rutina de análisis para calcular el tiempo que puede demorar en realizar su trabajo (por ejemplo detectar violaciones de atomicidad). Esto es posible si se usa una etapa de entrenamiento en la que se ejecutan varias aplicaciones a través de la herramienta con diferentes intervalos y se comparan los resultados. Es de esperar que una vez conocido este valor, será seguro utilizarlo con otras aplicaciones.

6.6. Resumen

Habitualmente los contadores hardware son utilizados para obtener métricas sobre eventos que ocurren durante la ejecución de una aplicación o de una fracción de código de ésta. Un segundo modo de operación llamado *muestreo de eventos*, permite configurar los contadores para generar señales dirigidas a un proceso ante la ocurrencia de eventos. En este modo el programador especifica la cantidad de eventos que deben ocurrir antes de que la señal sea generada, permitiendo ajustar esta prestación de acuerdo a los requerimientos de la aplicación.

A los efectos de este trabajo de tesis, este modo de operación puede ser utilizado para decidir cuándo activar la rutina de análisis de las herramientas de detección y en consecuencia reducir la instrumentación del código. En su forma más simple, es posible configurar estos contadores agregando un *manejador de instrucciones* en la aplicación que está siendo monitorizada. Un manejador de señales es una función del usuario que será ejecutada por el proceso cada vez que reciba la señal asociada. Sin embargo, esta forma de gestionar las señales posee los siguientes defectos:

- No contempla que la señal podría ser generada por cualquiera de los procesos que componen la aplicación paralela.
- No define el comportamiento a seguir luego de que el evento sea detectado. Múltiples señales seguidas podrían ser perjudiciales para la aplicación.
- El rendimiento de la aplicación podría ser afectado seriamente si la rutina de análisis es muy costosa o si es interrumpido constantemente para atender las señales.

En una versión más completa que contempla los puntos anteriores se propone dividir la tarea en dos programas, el POE (Programa Objeto de Estudio) y el GI (Gestor de Interrupciones). Esta división permite aislar la gestión de señales en un proceso independiente, de manera que la aplicación monitorizada (POE) sólo sea interrumpida cuando el GI determina que debe activar la rutina de análisis, resolviendo los dos últimos inconvenientes de la primera versión del modelo. Con respecto al primer punto, afortunadamente es posible configurar un contador hardware para cada proceso si estos se ejecutan en procesadores diferentes. Estos cambios implican que habrá que agregar una fracción de código al código fuente del POE para configurar los contadores ante la creación de nuevos threads y configurar un manejador de señales para que el GI pueda comunicarse con este cuando sea apropiado.

No obstante, para que el modelo esté completo es necesario combinar el método propuesto con la instrumentación dinámica para activar/desactivar la rutina de análisis y de esta forma conseguir una mejora de rendimiento en la herramienta de detección. En este caso el POE debe ser interpretado como la herramienta de detección, donde la función del manejador de señales debe cambiar el valor de un *flag* `toolIsOn` (inicialmente configurado como falso) a verdadero. Este flag será utilizado para determinar si la rutina de análisis debe ser ejecutada (si su valor es verdadero) o no (si su valor es falso).

Aunque el activado de la rutina de análisis se resuelve fácilmente, el desactivado puede ser un poco más complejo. Debido a que no es posible configurar un contador para enviar una señales ante la *no ocurrencia* de eventos, se propone configurar un *timer* para interrumpir al POE a intervalos regulares de tiempo. El manejador de estas señales ejecutará una rutina de muy bajo costo cuya única función es buscar en las estructuras de datos de la herramienta de detección evidencias de que es seguro desactivar la rutina de análisis (por ejemplo porque en el último intervalo no se detectaron violaciones de atomicidad). En caso de que la rutina pueda ser desactivada, cambiará el valor del flag `toolIsOn` nuevamente a falso y avisará al GI que la rutina de análisis ha sido desactivada para que vuelva a esperar señales.

En conclusión, se ha propuesto un modelo de instrumentación dinámica de aplicaciones paralelas que utiliza los contadores hardware para detectar el momento preciso en que ha ocurrido un evento en el hardware. Este modelo sirve para activar y desactivar las herramientas de monitorización de manera automática, lo que se espera permitirá reducir el overhead de las herramientas producto de la instrumentación y disminuir su impacto en la ejecución del software durante la etapa de producción.

Capítulo 7

Experimentación y resultados obtenidos

7.1. Introducción

Este capítulo propone una implementación del modelo tratado en el capítulo anterior aplicado a AVIO (7.2). Luego se describe la configuración experimental (7.3), se analiza la mejora en rendimiento alcanzada (7.4) y los efectos de utilizar el modelo sobre la capacidad de detección de la herramienta (7.5). Finalmente, se propone una discusión sobre el método empleado para comparar las versiones de AVIO (7.6).

7.2. AVIO Consciente de Compartición

Esta sección integra los conceptos de instrumentación dinámica guiada por eventos de hardware discutidos en el Capítulo 6 con la herramienta de detección de violaciones de atomicidad AVIO, elegida como caso de estudio en el Capítulo 3. El resultado es una nueva versión de AVIO llamada AVIO-SA, capaz de detectar el momento en que comienzan a ocurrir interleavings no serializables y activar en función de ello las rutinas de análisis. Para la detección de interleavings no serializables se utiliza el evento analizado en el Capítulo 5. Un contador se configura en modo de muestreo para enviar señales al proceso e interrumpir su ejecución cada vez que ocurra el evento en la cache de alguno de los procesadores donde se ejecutan los threads de la aplicación. Esta señal será capturada por AVIO-SA y en consecuencia activará la rutina de análisis. La Figura 7.1 muestra el modelo de ejecución esperado.

Al principio las rutinas de análisis de AVIO se encuentran desactivadas mientras se ejecutan en paralelo dos threads de la misma aplicación. Durante la fracción de tiempo

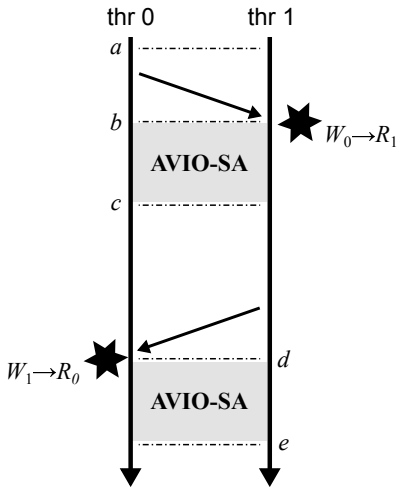


Figura 7.1: Activado de herramienta en función de eventos de cache.

correspondiente al segmento $a-b$ los threads ejecutan instrucciones que no generan eventos del tipo monitorizado. En algún momento de la ejecución, el thread 0 escribe un dato en memoria que luego es leído por el thread 1, provocando el patrón de acceso $W_0 \rightarrow R_1$. Como resultado de esta interacción, el contador del procesador donde se ejecuta el thread 1 detecta la ocurrencia del evento y envía una señal para activar la rutina de análisis.

En el instante de tiempo b se activa la rutina de análisis. Durante la fracción de tiempo correspondiente al segmento $b - c$ AVIO-SA se comporta como la versión original, instrumentando todas las instrucciones que acceden a memoria en búsqueda de interleavings no serializables. Luego, en el instante c llega a la aplicación una señal `SIGALRM`, y al corroborar que no se detectaron interleavings no serializables en el último intervalo de tiempo, AVIO-SA se desactiva y vuelve a esperar la ocurrencia de un evento para ser activado nuevamente.

A partir de aquí la aplicación continuará ejecutando sin instrumentación (segmento $c - d$). Eventualmente uno de los contadores asociados a los procesos volverá a detectar un evento y enviará una nueva señal (d), momento en el que la secuencia se repite.

7.2.1. Detalles de implementación

AVIO-SA se implementó sobre las bases de la versión de AVIO desarrollada en el Capítulo 3. Para ello se utilizó el framework de instrumentación binaria dinámica PIN

(Luk y cols., 2005) en la versión 2.11, y la API de linux para acceso a los contadores perf_events, incluida por defecto en el kernel de Linux desde la versión 2.6.32.

La implementación de la capa de software descrita en la Figura 6.1 se consigue agregando unas pocas variables y añadiendo soporte para manejo de señales a la implementación de AVIO. Para garantizar que la interacción entre dos threads se refleja en los cambios de estado de la cache, se aseguró configurando la afinidad de los threads que cada uno se ejecute en un núcleo de procesamiento diferente.

Diseño del POE. La rutina de análisis de AVIO trabaja con complejas estructuras de datos para llevar la historia de todos los accesos a memoria que realiza el programa monitorizado. Se utilizó el modelo propuesto para activar AVIO sólo cuando se detecte compartición de datos. En el Capítulo 6 se mostró cómo un contador hardware puede ser utilizado con este fin. La Figura 7.2 compara la versión original de AVIO con AVIO-SA. Se puede observar la señal *(1)* (`SIGUSR1`) dirigida a GI generada por el evento. El intervalo de tiempo utilizado para el timer se lee de una variable de entorno, de manera que no es necesario recompilar la herramienta para hacer pruebas con distintos valores. Para la nueva versión fue necesario modificar AVIO de manera que exponga al POE los interleavings no serializables detectados. De esta manera cuando el POE recibe la señal `SIGALRM` puede determinar si se está ejecutando una sección de código que no accede a datos compartidos: en este caso se desactiva y envía una señal `SIGUSR2` al GI, identificada en la Figura 7.2 por *(3)*, para avisar que AVIO fue desactivado. De esta manera, las fase de *análisis* y *modificación* de la instrumentación dinámica consisten en determinar cuándo es conveniente activar / desactivar AVIO con el fin de mejorar el rendimiento general de la aplicación.

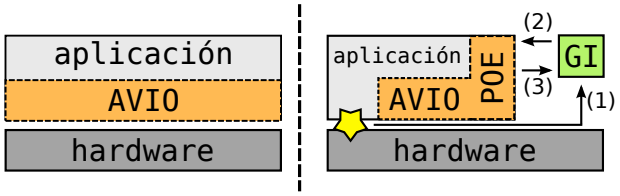


Figura 7.2: Versión original de AVIO (izquierda) y versión optimizada (derecha).

Diseño de GI. El GI fue diseñado de acuerdo al modelo expuesto en la Sección 6.4, con un manejador de señales para capturar eventos provenientes del POE. Al recibir la primera señal `SIGUSR1`, el manejador es reemplazado por `SIGIGN` para ignorar el resto

y envía una señal de activado al POE. Esta señal está identificada en la Figura 7.2 por el código *(2)*. Posee además otro manejador de señales `SIGUSR2` para volver a activar la detección de eventos cuando el POE detecte que no están ocurriendo interleavings no serializables.

7.3. Configuración experimental

El modelo fue evaluado a partir de dos características: *capacidad de detección* y *rendimiento*. En cada caso se utilizó un conjunto de benchmarks diseñados para esos fines.

Para comparar AVIO-SA se utilizó la versión de AVIO del Capítulo 3 y HELGRIND (Jannesari y cols., 2009), una herramienta comercial ampliamente utilizada para detección de condiciones de carrera. HELGRIND viene incluido con VALGRIND, y se instala a partir del administrador de paquetes de Debian. La versión más actual instalable desde los repositorios es la 3.8.1.

Plataforma. El sistema operativo es un Debian wheezysid x86_64 GNULinux con kernel 3.2.0. El compilador empleado fue gcc en la versión 4.7.0. Salvo que se indique lo contrario, la arquitectura objetivo de la compilación es la x86_64 y los flags de optimización estandar -O2. Los experimentos se realizaron en una máquina con dos procesadores Xeon X5670, cada uno con 6 núcleos con HT. En esta plataforma, los procesadores numerados del 0 5 corresponden a los núcleos físicos del primer procesador y los procesadores numerados del 6 al 11 corresponden a los núcleos físicos del segundo procesador. Gracias al hiperthreading, se cuenta además con los procesadores numerados del 12 al 17 del primer procesador y del 18 al 23 del segundo procesador. Sin embargo, los experimentos están diseñados para ser ejecutados sólo con 2 threads, ya que como se comentó al final del Capítulo 2, la mayoría de los errores de concurrencia involucran esta cantidad de procesos. No obstante, para asegurar que los experimentos siempre se ejecutarán en procesadores físicos distintos se evitaron configuraciones con más de 12 threads, y la afinidad de cada thread se fijó acorde a su número, garantizando que los threads serán ejecutados en los procesadores identificados del 0 al 11. La microarquitectura de estos procesadores es Westmere. La frecuencia de cómputo es de 2.93 GHz. Cada procesador tiene tres niveles de cache – L1 (32KB) y L2(256KB) privadas de cada núcleo y L3(12MB) compartida entre todos los núcleos del mismo procesador (*Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2012).

7.3.1. Elección de un intervalo de muestreo óptimo

Para encontrar el intervalo de muestreo óptimo se diseñaron kernels de ejecución de N instrucciones, donde N es el número de instrucciones que representan una sección que *debería* ser atómica. Debido a que se busca un caso general, se decidió simular las instrucciones como una operación de incremento sobre una variable entera. El valor de N se informa en tiempo de compilación, y el preprocesador se encarga de generar la secuencia de N instrucciones. Además, se toma el tiempo antes de procesar la primera instrucción y luego de procesar la *N-ésima*. Al finalizar la ejecución, el kernel imprime un mensaje indicando la diferencia de ambas mediciones.

Para contemplar el overhead de la herramienta de instrumentación, se diseñó una herramienta que recibe el kernel como argumento y simula el proceso de instrumentación de las operaciones de lectura y escritura.

El primero de los experimentos consistió en obtener los tiempos probables para el intervalo de muestreo. Para ello se ejecutaron los kernels de diferentes cantidades de instrucciones a través de la herramienta de PIN. La tabla 7.1 muestra los resultados obtenidos.

Cuadro 7.1: Tiempos expresados en milisegundos para ejecutar diferentes cantidades de instrucciones en la arquitectura de pruebas.

Bench	Nro. instructions	ms
R0010	10	1,717
R0100	100	1,982
R0500	500	3,256
R1000	1000	4,869
R1500	1500	6,433
R2000	2000	7,919
R2500	2500	9,081

Una vez conocidos los valores probables para el intervalo, es necesario decidir cuál será el óptimo. De acuerdo a lo enunciado al final de la sección 6.5.3, el valor óptimo será aquel que minimice el tiempo de ejecución de las aplicaciones y maximice el número de interleavings detectados. Para obtener un conjunto de valores testigo que permitan interpretar los resultados de los experimentos, se utilizaron los paquetes de la suite de benchmarks SPLASH-2 a través de AVIO. Para cada aplicación se tomó en cuenta el tiempo que demora en ejecutarse a través de AVIO y el número de interleavings no serializables detectados. La tabla 7.2 resume los resultados obtenidos.

Para cada experimento se realizaron 10 ejecuciones. Los valores de la tabla corresponden a los promedios de tiempo (en segundos) y de número de interleavings no serializables

Cuadro 7.2: Resultados de ejecutar los paquetes de la suite SPLASH-2 a través de nuestra implementación de AVIO. Los resultados son los promedios de 10 ejecuciones.

Package	Interleavings		Seconds	
	media	DevNorm	media	DevNorm
barnes	98,0	1,247	130,305	3,548
cholesky	57,0	-	28,231	1,608
fft	9,0	-	2,746	0,032
fmm	127,1	3,843	83,039	3,567
lu_cb	11,8	0,422	26,522	0,786
lu_ncb	12,0	0,471	1,075	0,014
ocean_cp	60,2	1,751	43,359	0,466
ocean_ncp	57,2	0,632	41,400	0,277
radiosity	239,5	10,091	65,323	0,855
radix	20,9	1,287	3,808	0,064
raytrace	30,5	1,581	47,468	0,570
volrend	82,1	2,767	21,139	0,342
water_nsq	55,0	4,055	25,410	0,336
water_sp	41,0	2,160	22,034	0,232
Total	**901,3**		**541,859**	

detectados. Las columnas destacadas en gris corresponden a la desviación estándar de la muestra de cada característica. Se debe notar que las variaciones entre ejecuciones no son significativas.

Una vez obtenidos los posibles valores para el intervalo de muestreo y los valores testigo, se repitieron los experimentos a través de AVIO-SA para cada tamaño de bloque. La tabla 7.3 resume los tiempos obtenidos y la tabla 7.4 resume el número de interleavings no serializables detectados en cada caso.

Cuadro 7.3: Tiempo en segundos de ejecutar los paquetes de la suite SPLASH-2 a través de AVIO-SA. Los resultados son los promedios de 10 ejecuciones.

	Number of Instructions						
	10	**100**	**500**	**1000**	**1500**	**2000**	**2500**
barnes	4,146	4,474	5,354	5,919	6,789	7,511	7,699
cholesky	1,861	1,983	1,661	2,173	2,011	2,876	2,994
fft	0,883	0,914	0,886	0,99	0,989	1,066	1,049
fmm	3,01	3,17	3,924	5,41	7,226	8,661	11,707
lu_cb	1,854	2,2	2,707	3,31	4,027	4,047	4,201
lu_ncb	0,866	0,837	0,891	0,931	0,953	0,989	0,971
ocean_cp	1,777	2,043	8,72	13,153	17,516	20,809	19,8
ocean_ncp	6,619	10,441	13,724	15,25	17,819	19,293	19,433
radiosity	12,857	12,35	19,257	20,016	21,37	22,144	23,337
radix	0,849	1,036	1,057	1,36	1,543	1,537	1,451
raytrace	10,453	15,28	20,617	29,867	30,414	25,469	30,324
volrend	7,364	7,344	8,711	9,627	10,333	11,307	11,681
water_nsq	2,321	2,24	2,099	1,909	2,034	1,814	1,803
water_sp	2,39	2,687	3,086	2,644	2,296	2,54	2,199
Total	**57,25**	**66,999**	**92,694**	**112,559**	**125,32**	**130,063**	**138,649**

Para facilitar la comparación entre herramientas se utilizó la suma de las medias. Se debe observar en la tabla 7.3 que a bloques de instrucciones más pequeños, menor es el

Cuadro 7.4: Número de interleavings detectados producto de ejecutar los paquetes de la suite SPLASH-2 a través de AVIO-SA. Los resultados son los promedios de 10 ejecuciones.

	Number of Instructions						
	10	**100**	**500**	**1000**	**1500**	**2000**	**2500**
barnes	111,286	119,571	118,857	117	110,143	113,714	110,429
cholesky	2,571	7,857	0	5,571	4,571	15,143	18,286
fft	14,571	15,429	15	16,571	16,857	16,143	15,714
fmm	46,857	65,429	92,429	142,286	160,571	200	201
lu_cb	4,286	5,857	8,286	8,857	10,143	11,286	11,857
lu_ncb	12,714	10,429	12,571	12,857	12,429	12,286	12,286
ocean_cp	0,286	6,857	61,286	72,714	84,143	90,857	82
ocean_ncp	74	102,429	97,857	88,429	91,286	94	87,429
radiosity	15,429	45,857	246,571	250,143	251,286	258,714	257,143
radix	1,571	7,714	6,714	12	12,286	12,429	11,571
raytrace	8,143	12,857	13,286	19,429	20,143	17,143	19,143
volrend	102,429	95,714	100	103,714	101,429	99,143	100,429
water_nsq	16,857	13,143	19,286	16,286	26,429	22,143	23,857
water_sp	26,429	35,286	42,571	43,143	41,857	42,429	42,429
Total	**437,429**	**544,429**	**834,714**	**909**	**943,573**	**1005,43**	**993,573**

tiempo de ejecución. Esto se debe a que un intervalo de muestreo pequeño favorece el desactivado de AVIO muy cerca del lugar donde se generó el evento que lo activó, lo que a su vez reduce la instrumentación de las lecturas y escrituras del programa repercutiendo en una mejora notable de tiempo: por ejemplo un intervalo para un bloque de 10 instrucciones tarda aproximadamente la mitad que un intervalo para un bloque de 1000 instrucciones.

Sin embargo, si se observa la tabla 7.4 se notará que a bloques de instrucciones más pequeños, también es menor el número de interleavings detectados.

Este comportamiento se podría explicar de dos maneras:

1. El apagado temprano de AVIO puede provocar que algunas instrucciones de lectura/escritura no sean registradas por la herramienta, lo que se traduce en la pérdida de detecciones.

2. Mientras más tiempo permanece AVIO desactivado, menos influyen las rutinas de instrumentación en el no determinismo de la ejecución. Esto significa que algunos interleavings no serán detectados simplemente porque no ocurrieron en esa ejecución.

Por los motivos antes expuestos, se considera que una solución de compromiso (y la que se utilizará en el resto de los experimentos) consiste en elegir un intervalo que detecte aproximadamente la misma cantidad de interleavings que AVIO con el menor tiempo posible. No obstante, al final de este capítulo se retomará este tema para su análisis.

Siguiendo este criterio es que se eligió el intervalo correspondiente a un bloque de 1000 instrucciones (aproximadamente 5ms), ya que es el intervalo con el menor tiempo

que detecta (en la suma total) aproximadamente la misma cantidad de interleavings que AVIO.

7.4. Rendimiento

En esta sección se compara el rendimiento de AVIO-SA con el de AVIO y HELGRIND.

Se realizaron mediciones de tiempo para cada aplicación de la suite ejecutada a través de las dos versiones de AVIO. Para los resultados se utilizó el menor de los tiempos entre 10 ejecuciones. La Tabla 7.5 resume los resultados de los experimentos.

Cuadro 7.5: Relación de overhead para AVIO, AVIO-SA y HELGRIND.

aplicación	**AVIO**	**HELGRIND**	**AVIO-SA**
barnes	84×	1142×	8×
cholesky	26×	335×	2×
fft	4×	2×	1×
fmm	68×	1108×	6×
lu_cb	36×	6×	4×
lu_ncb	2×	1×	2×
ocean_cp	37×	35×	10×
ocean_ncp	38×	41×	12×
radiosity	48×	135×	24×
radix	6×	2×	1×
raytrace	35×	54×	38×
volrend	18×	248×	17×
water_nsq	28×	4×	2×
water_sp	24×	4×	2×
Promedio General	32×	223×	9×
Promedio 1	31×	289×	3×
Promedio 2	35×	102×	20×

Cada fila de la tabla corresponde a una aplicación distinta de la suite de benchmarks. La primera columna corresponde a los resultados obtenidos utilizando la implementación original de AVIO, la segunda columna corresponde a los resultados obtenidos con HELGRIND y la tercera columna corresponde a los resultados obtenidos con la versión AVIO-SA.

Cada resultado está expresado como la relación entre el tiempo (en segundos) que demora una aplicación de la suite ejecutada a través de cada versión sobre el tiempo que demora sin ser ejecutada a través de ella. Esta relación de overhead expresa cuantas veces más tardará en ser ejecutada una aplicación si se utiliza la herramienta de detección de violaciones de atomicidad. Para facilitar la interpretación de los datos, los resultados se redondearon a su parte entera.

Las últimas tres filas de la tabla muestran los promedios de overhead para cada versión: *Promedio General* corresponde al promedio de las 14 aplicaciones de la suite, *Promedio 1* corresponde al promedio de las aplicaciones que demuestran un patrón de eventos favorable para la propuesta y *Promedio 2* corresponde al promedio de las aplicaciones que demuestran un patrón de eventos desfavorable para la propuesta. Las aplicaciones con patrón de eventos favorable son las de caso 1 y 3 indicadas en la Sección 5.6.3, mientras que las aplicaciones con patrón de eventos desfavorable son las de caso 2 de la misma sección.

De las tres herramientas, la que tuvo peor desempeño (en promedio) fue HELGRIND con un overhead de 223×. AVIO provocó en promedio un overhead de 32× contra un 9× de AVIO-SA. Se debe notar que el overhead promedio de AVIO-SA es menos del 30 % del overhead promedio de AVIO.

Es interesante notar que para las aplicaciones con un comportamiento favorable en la ocurrencia de eventos, la penalización de AVIO-SA es significativamente menor que para AVIO y HELGRIND, disminuyendo con respecto a AVIO el overhead en un orden de magnitud. Con respecto a las aplicaciones con un comportamiento desfavorable, es importante destacar que el overhead de AVIO-SA fue equivalente para dos de las aplicaciones (raytrace y volrend) e inferior a la mitad para las otras tres (ocean_cp, ocean_ncp y radiosity). Esto demuestra que aún con una distribución de eventos desfavorable para la propuesta, se confirma que en la mayoría de los casos el modelo de activado por eventos mejora el rendimiento de la herramienta de detección.

7.5. Capacidad de detección

Los experimentos destinados a probar la capacidad de detección de AVIO-SA se realizaron en tres etapas:

1. Prueba de detección en los kernels.

2. Prueba de detección en aplicaciones reales.

3. Comparación de bugs informados entre AVIO y AVIO-SA.

7.5.1. Prueba de detección en los kernels

Para evaluar la capacidad de detección de AVIO-SA se utilizaron los cuatro kernels de ejecución presentados en la Sección B.1.1, uno para cada caso de interleaving conocido. Al igual que los experimentos de la sección anterior, se utilizó un intervalo de muestreo de 5ms. Tal como se muestra en la Tabla 7.6, las modificaciones introducidas con nuestra propuesta no afectaron la capacidad de detección de AVIO: ambas versiones fueron capaces de detectar los cuatro casos conflictivos.

Cuadro 7.6: Resultados de detección para diferentes herramientas usando kernels con errores de concurrencia.

APLICACIÓN	AVIO	AVIO-SA	HELGRIND
APACHE	Si	Si	Si
MOZILLA	Si	Si	No
MYSQL	Si	Si	Si
BANKACCOUNT	Si	Si	No

En el caso de HELGRIND, sólo detectó los bugs de APACHE y MYSQL. En la Sección 3.5.3 se explicó porqué HELGRIND no pudo detectar los bugs de MOZILLA y BANKACCOUNT.

7.5.2. Prueba de detección en aplicaciones reales

Para comprobar que AVIO-SA efectivamente detecta errores en aplicaciones reales, se compiló y utilizó la versión 2.2.29 de la aplicación Apache. Esta versión fue modificada para que la función que posee el error sea equivalente a la de la versión original (2.0.48). El anexo E provee detalles sobre estas modificaciones y la configuración empleada en este experimento.

Debido a la naturaleza de aplicación cliente/servidor, definir un experimento para medir rendimiento y overhead para este programa es muy difícil. Por lo tanto este experimento simplemente consistió en determinar si AVIO-SA era capaz o no de detectar el bug reportado en el trabajo original de AVIO(Lu y cols., 2006).

Al igual que con los kernels del experimento anterior sólo hay dos resultados posibles: detectó el bug o no. Afortunadamente, los experimentos demostraron que AVIO-SA es capaz de detectar el bug mencionado.

7.5.3. Comparación de bugs informados entre AVIO y AVIO-SA

Este experimento consistió en comparar AVIO y AVIO-SA en términos de violaciones de atomicidad detectadas por ambas versiones sobre las aplicaciones de SPLASH-2. La Tabla 7.7 muestra los resultados obtenidos.

Cuadro 7.7: Cantidad de violaciones de atomicidad informadas por cada versión de AVIO.

package	AVIO	AVIO-SA
barnes	104	140
cholesky	57	18
fft	9	18
fmm	139	233
lu_cb	11	10
lu_ncb	13	12
ocean_cp	61	86
ocean_ncp	58	97
radiosity	250	276
radix	21	17
raytrace	32	24
volrend	85	100
water_nsq	50	27
water_sp	45	43
Total	**935**	**1101**

En cada caso se utilizó el máximo de violaciones de atomicidad informadas entre 10 ejecuciones. La última fila de la tabla muestra la suma total de violaciones detectadas por cada versión. Se debe notar que en total la versión AVIO-SA detectó más violaciones de atomicidad que AVIO. De las 14 aplicaciones, sólo en dos (cholesky y water_nsquared) AVIO-SA detectó menos de la mitad que AVIO. Estos valores son relativos ya que el número de violaciones de atomicidad detectadas por cada versión depende fundamentalmente del no determinismo implícito de cada ejecución.

7.6. Discusión

Aunque los métodos empleados hasta ahora reflejan los procedimientos de comparación utilizados en trabajos relacionados, los resultados obtenidos de este trabajo de tesis invitan a hacer dos preguntas importantes:

1. ¿Es realista comparar dos herramientas a partir del número de interleavings detectados?

2. Si no es así, ¿de qué manera se pueden comparar?

7.6.1. Interleavings detectados como criterio de comparación

Dado que el no determinismo inherente de las arquitecturas paralelas provoca que dos ejecuciones seguidas de un mismo programa tengan dos historias distintas de ejecución –generando distintos interleavings– entonces aún *usando la misma herramienta de detección de errores se obtendrán resultados distintos*.

Sin embargo, al ejecutar repetidas veces la misma aplicación con la misma herramienta siempre habrá un subconjunto de interleavings que se repetirán en cada ejecución: este conjunto corresponde a los interleavings que ocurren como producto de la sincronización especificada por el usuario. Partiendo de esta premisa es que AVIO propone una etapa de entrenamiento para capturar ese conjunto de interleavings a los que llama *invariantes*. Finalizada la etapa de entrenamiento, entonces cualquier interleaving detectado por AVIO que no se encuentre en el conjunto de invariantes será considerado un bug y por lo tanto informado al usuario. De hecho una buena parte del éxito de este método se basa en la observación de que normalmente en las ejecuciones los bugs –o interleavings no serializables imprevistos– no ocurren: si ocurrieran entonces habrían sido detectados oportunamente y no serían bugs latentes en el código.

Dado que el éxito de un algoritmo de detección radica en encontrar nuevos errores, es evidente que existe un problema en conseguir que las ejecuciones correspondientes a la etapa de depuración del código no sean *normales* (en el sentido del párrafo anterior). Así es que aparecen en la literatura propuestas complementarias a los algoritmos de detección que buscan forzar historias de ejecución poco probables (S. Park y cols., 2009; Lu y cols., 2012; Deng y cols., 2013).

El punto de discusión no está centrado en los interleavings que son invariantes, ya que es de esperar que sean detectados por cualquier propuesta, sino en el conjunto restante de interleavings que no ocurren en ejecuciones normales. Este segundo conjunto de interleavings está formado por historias que no habrían ocurrido si no hubiera sido alterado el orden natural de ejecución de las instrucciones del programa. ¿Cómo se altera el orden natural de ejecución? Puede ser a través de interrupciones, señales, sobrecarga del sistema o por la misma *instrumentación dinámica*.

Sirva de ejemplo el siguiente experimento realizado con una aplicación llamada bodytrack. bodytrack es una aplicación cliente/servidor paralelizada con phtreads que puede ser configurada para ser ejecutada con varios hilos. La aplicación se ejecutó con 1, 2, 4, 6 y 8 hilos (clientes) a través de las versiones AVIO y AVIO-SA. La Tabla 7.8 muestra los resultados de los experimentos.

Cuadro 7.8: Comparación de AVIO y AVIO-SA a partir del overhead que introducen y del número de interleavings detectados.

Versión	Datos	1	2	4	6	8
AVIO	overhead	84×	138×	296×	466×	466×
	interleavings	900	1594	1690	1686	1719
AVIO-SA	overhead	3×	38×	23×	22×	12×
	interleavings	54	178	238	181	213

La tabla muestra para cada versión el overhead y los interleavings detectados. Si la comparación entre ambas versiones se realiza considerando solamente el overhead, entonces AVIO-SA resulta una propuesta ampliamente superadora sobre AVIO. Sin embargo, al incluir en la comparación el número de interleavings detectados se puede observar que AVIO-SA en el mejor de los casos detecta apenas el 15 % de lo que detecta AVIO.

Estos resultados se pueden explicar de dos maneras:

1. Los interleavings ocurren, pero AVIO-SA falla en detectarlos.

2. Los interleavings no ocurren, y por lo tanto no puede detectarlos.

Por un lado, dados los resultados de los experimentos en las secciones anteriores, parece poco probable que AVIO-SA falle en detectar los interleavings. Se debe considerar que el modelo propuesto sobre el que se desarrolló AVIO-SA evita la instrumentación de las instrucciones a menos que el evento ocurra. Si el código que se está ejecutando posee pocos interleavings en las ejecuciones normales, entonces es natural pensar que AVIO-SA interferirá menos en la ejecución del programa y por lo tanto ocurrirán menos interleavings que con AVIO.

En este contexto, utilizar el número de interleavings detectados por las herramientas no parece ser un buen criterio de comparación ya que perjudica a AVIO-SA en el sentido de que es más probable que las ejecuciones que analiza sean “más normales” (o menos alteradas) que las que analiza AVIO. Por otro lado, dado el no determinismo implícito en las ejecuciones tampoco es sencillo probar esta hipótesis.

7.6.2. Comparación basada en re-ejecución

Dado que cada ejecución es distinta, resulta muy difícil imaginar una forma de probar que los interleavings no detectados no ocurrieron. Es necesario un método más confiable para comparar herramientas de detección.

Una estrategia que puede servir para este fin fue empleada en (Ronsse y De Bosschere, 1999). Esta propuesta estaba destinada a la detección de condiciones de carrera. Para funcionar requería dos ejecuciones: en la primera ejecución registra el orden en que se ejecutan las instrucciones, y en la segunda ejecución fuerza esa historia "grabada" a través de su algoritmo de detección.

Aunque como método de detección de errores este mecanismo es muy costoso, la idea de grabar la historia de ejecución para posteriormente comparar la detección de dos herramientas distintas sobre esa historia resulta prometedora.

La hipótesis es que si se elimina el no determinismo de las ejecuciones garantizando que las historias que analizarán ambas herramientas serán equivalentes, entonces se puede usar el número de interleavings detectado como criterio de comparación.

Re-ejecución y AVIO

Para conseguir que AVIO pueda reproducir una historia, se modificó la versión original para incluir el siguiente fragmento de código del Listado 7.1 a cada lectura o escritura en memoria.

Donde:

- `tableAccess` es un arreglo unidimensional que contiene la historia de accesos a reproducir. Corresponde a una historia de ejecución previamente guardada con otra herramienta diseñada para este fin, y almacena el número de hilo y el número de instrucción que debe ser ejecutado en cada paso para reproducir la historia original.

- `next` es un contador compartido que indica cuál es la siguiente instrucción que debería ser ejecutada.

- `acc` es un puntero a la siguiente instrucción que debe ser ejecutada.

- `delay()` es una función que introduce una demora para dar oportunidad a que otra instrucción sea ejecutada.

- `threadid` e `ins` son dos parámetros que pasa la herramienta de instrumentación dinámica a la función con el número de thread y el número instrucción que está por ejecutar.

Listado 7.1: Fragmento de código empleado antes de cada acceso a memoria para demorar la ejecución la instrucción hasta que sea su turno.

```
1      ...
2       GetLock(&lock);
3       Access *acc = &tableAccess[next];

5       while (threadid != acc->getThread() || (lluint) ins !=
      acc->getIns()){
6            ReleaseLock(&lock);

8            delay();

10           GetLock(&lock);
11           acc = &tableAccess[next];
12      }

14      next++;

16      // llamada al método correspondiente de AVIO
17      ReleaseLock(&lock);
18     ...
19 }
```

El fragmento de código anterior utiliza un bucle `while` para introducir demoras antes de ejecutar una instrucción hasta que sea su turno. Si una instrucción está siendo demorada, para garantizar que otra instrucción podrá ser ejecuta se libera el `lock` antes de ejecutar el `delay()`. Cuando el `delay()` finaliza se vuelve a capturar el `lock` y se verifica nuevamente si es el turno de la instrucción para ser ejecutada. Se supone que el contador `next` ha sido incrementado y en consecuencia actualizó el puntero `acc` para que la comparación del bucle apunte al siguiente acceso en `tableAccess`. Esto continúa hasta que la siguiente instrucción en `tableAccess` corresponde con la que está intentando ejecutar y abandona el bucle. Fuera del bucle se incrementa el contador `next` para que apunte a la siguiente instrucción, se ejecuta la rutina de análisis de AVIO correspondiente (escritura o lectura) y se libera el `lock`.

Re-ejecución y AVIO-SA

Conseguir re-ejecutar una historia en AVIO-SA es un poco más complicado, ya que además del no determinismo producto de la ejecución existen otros dos orígenes a contemplar: los eventos hardware para activar las rutinas de análisis y el timer para desactivarlas.

Se debe considerar que cuando ocurra el evento si la siguiente instrucción no es la que debe ser ejecutada según `tableAccess`, entonces el `delay()` que ejecuta generará una ventana de tiempo, dando lugar a que llegue una señal `SIGALRM` proveniente del timer. En esta situación no se habrán detectado bugs debido a que aún la siguiente instrucción no ha sido ejecutada, pero el mecanismo acabará desactivando la rutina de análisis. Por lo tanto para poder realizar la comparación con AVIO, se propone simular los eventos hardware y el timer.

Para simular la ocurrencia de eventos se desarrolló una clase `Cache`, con métodos para registrar escrituras y lecturas en cada dirección de memoria. Dado que el evento que se está utilizando ocurre con cada lectura a una dirección de memoria que fue escrita por un proceso distinto al que está leyendo, se incluyó un *flag* en cada dirección de memoria para saber si el proceso que lee es el mismo que escribió anteriormente. Si al momento de leer se detecta que el proceso que lee es distinto al último que escribió, entonces se simula el evento llamando al handler de eventos de hardware de GI. En la Sección 5.5 se demostró que el evento utilizado es preciso en términos de diferenciar este patrón de acceso del resto, por lo que se considera equivalente al funcionamiento esperado de AVIO-SA.

Para simular el timer se utilizaron los resultados de la Sección 7.3.1. Dado que el intervalo de muestreo utilizado en los experimentos es de 5ms y que corresponde al tiempo necesario para ejecutar aproximadamente 1000 instrucciones, se agregó una rutina de análisis a cada instrucción del programa (no solamente lecturas y escrituras) que todo lo que hace es incrementar un contador. Cuando ese contador superó las 1000 instrucciones, entonces se llama al handler del timer simulando la señal `SIGALRM` para ejecutar la rutina que analiza si debe o no desactivar la rutina de análisis. Aunque la relación entre tiempo y número de ejecuciones no es lineal, se considera que es una aproximación realista para la simulación.

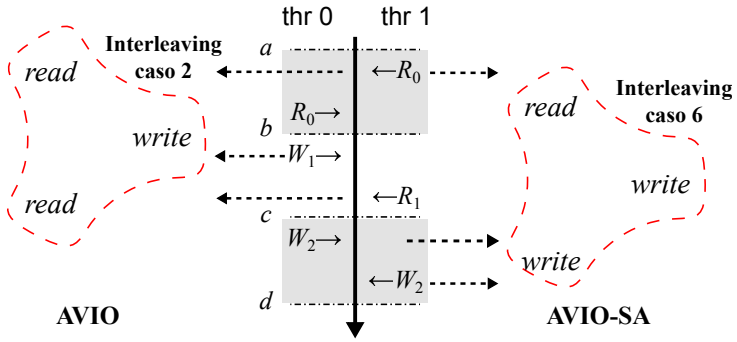
AVIO vs. AVIO-SA basados en re-ejecución

Dado que tanto los eventos como el timer simulados ahora ocurren en función de las instrucciones ejecutadas, se repitieron los experimentos con bodytrack para AVIO y

AVIO-SA pero asegurando que la historia de ejecución es la misma. Se utilizó el paquete de datos TEST y fue configurado para tener 2 threads. Como resultado AVIO detectó 143 interleavings no serializables, mientras que AVIO-SA detectó 149 interleavings no serializables en esa misma historia. Las ejecuciones de la historia grabada fueron repetidas 10 veces, obteniendo los mismos resultados en cada caso.

Resulta interesante que AVIO-SA haya reportado 6 interleavings más que AVIO. Esto se puede explicar a partir del funcionamiento de AVIO-SA. La Figura 7.3 muestra un caso hipotético de la ejecución a través de AVIO y AVIO-SA.

Figura 7.3: Detección de interleavings para AVIO y AVIO-SA.



Cada interleaving está formados por dos operaciones de acceso a memoria de un mismo proceso (P, I) y una instrucción de un segundo proceso que ocurre justo en medio (R). Durante una ejecución donde los accesos a memoria son forzados a ocurrir en un orden determinado, AVIO siempre detectará el interleaving no serializable formado por las operaciones $P_{R_0} \rightarrow R_{W_1} \rightarrow I_{R_1}$ (posible bug caso 2). Sin embargo, debido a que AVIO-SA *desactiva* la rutina de análisis, las instrucciones R_{W_1} y I_{R_1} pueden no ser registradas, dando lugar al interleaving $P_{R_0} \rightarrow R_{W_2} \rightarrow I_{W_2}$ (posible bug caso 6). Es importante destacar que AVIO no tiene posibilidad de detectar este interleaving de caso 6 en una ejecución forzada, ya que al registrar todos los accesos a memoria siempre detectará el caso 2. Por otro lado, si estos interleavings ocurren de manera repetitiva, es probable que AVIO-SA en otra iteración sea activado antes, y en consecuencia registraría también el interleaving de caso 2. En consecuencia, AVIO-SA puede detectar más interleavings porque además de los interleavings detectados por AVIO, el desactivado da lugar a que el algoritmo registre nuevos casos de interleavings.

En función de los resultados, se considera que el modelo propuesto no afecta negativamente la capacidad de detección de AVIO. Debido a que AVIO-SA altera menos la historia de ejecución de la aplicación monitorizada, es una mejor opción para ser utilizada

en entornos de producción. Los resultados anteriores muestran que en caso de ocurrir el interleaving no serializable, el evento será generado y AVIO será activado, lo que permitirá detectarlo oportunamente.

De igual manera, es interesante destacar que para la etapa de depuración sería más conveniente el comportamiento opuesto al de AVIO-SA, es decir, conseguir que las historias de ejecución durante las pruebas sean lo más variadas posibles. Esto permitirá detectar mayor cantidad de bugs antes de que el software sea liberado para su etapa de producción.

7.7. Resumen

El modelo propuesto en el Capítulo 6 puede ser aplicado para reducir el overhead de la herramienta de detección de violaciones de atomicidad AVIO. Para ello se utiliza el evento candidato seleccionado en el Capítulo 5, el cual permite detectar patrones de acceso en cache a datos compartidos que indican la ocurrencia de interleavings no serializables. Esta nueva versión llamada AVIO-SA, inicia la ejecución de las aplicaciones monitorizadas con la rutina de análisis desactivada. En el momento en que detecta un evento la rutina es activada, funcionando por un tiempo como la versión original de AVIO. Eventualmente AVIO deja de detectar interleavings y la rutina de análisis es desactivada.

Para conseguir implementar esta versión, es necesario modificar ligeramente AVIO para exponer una cuenta parcial de los bugs detectados al POE. Se debe recordar que el POE posee un timer que genera señales a intervalos regulares de tiempo. El manejador de estas señales se encarga de buscar en las estructuras de AVIO el número de interleavings no serializables detectados en el último intervalo para decidir si debe desactivar la rutina de análisis o no.

Debido a que no es posible estimar el valor óptimo para el tiempo del intervalo de muestreo analíticamente, se desarrolló una serie de kernels de ejecución con diferente cantidad de instrucciones entre 10 y 2500. Estos valores se eligieron arbitrariamente, partiendo de que es sabido que las instrucciones que acceden a datos compartidos frecuentemente están cerca entre sí. También se desarrolló una herramienta de instrumentación dinámica con granularidad a nivel de instrucción para simular el overhead de AVIO si la rutina de análisis se encontrara desactivada. Luego se tomaron los tiempos de ejecución para cada kernels a través de esta herramienta de instrumentación.

Estos tiempos son considerados intervalos de muestreo potenciales. Para elegir uno de ellos, se ejecutaron las aplicaciones de SPLASH-2 a través de AVIO-SA y con cada uno de los tiempos potenciales para muestreo. Luego se repitió el experimento usando AVIO solamente, y se compararon los tiempos de ambas herramientas y la cantidad de interleavings no serializables reportados. Se encontró que un intervalo de 5ms permite detectar aproximadamente la misma cantidad de interleavings que AVIO, pero con un tiempo de ejecución significativamente menor.

Para completar las pruebas de rendimiento se completaron los experimentos con HELGRIND y se estimó el overhead de cada herramienta con respecto a cada aplicación. En promedio, HELGRIND demostró un overhead de 223×, AVIO un overhead de 32× y AVIO-SA de 9×. Se debe notar que el overhead promedio de AVIO-SA es menos del 30 % del overhead promedio de AVIO. Se analizó el rendimiento de las aplicaciones con distribución favorable de eventos y distribución desfavorable. Para las aplicaciones con un comportamiento favorable en la ocurrencia de eventos, la penalización de AVIO-SA disminuye con respecto a AVIO el overhead en un orden de magnitud, mientras que para las aplicaciones con un comportamiento desfavorable, el overhead de AVIO-SA fue equivalente para dos de las aplicaciones (raytrace y volrend) e inferior a la mitad para las otras tres (ocean_cp, ocean_ncp y radiosity). Esto demuestra que aún con una distribución de eventos desfavorable para la propuesta, se confirma que en la mayoría de los casos el modelo de activado por eventos mejora el rendimiento de la herramienta de detección.

Aparte del rendimiento, se evaluó la capacidad de detección de errores de AVIO-SA. Para ello se hicieron 3 experimentos:

- Prueba de detección con kernels de bugs conocidos.
- Prueba de detección en aplicaciones reales (Apache).
- Comparación de bugs informados entre AVIO y AVIO-SA (a partir de SPLASH-2).

Afortunadamente AVIO-SA pasó las 3 pruebas satisfactoriamente, demostrando que la aplicación del modelo en AVIO no afectó negativamente su capacidad de detección de errores.

Finalmente, se plantean dos preguntas relacionadas al método empleado para comparar herramientas de detección de errores:

1. ¿Es realista comparar dos herramientas a partir del número de interleavings detectados?

2. Si la respuesta es negativa, ¿de qué manera se pueden comparar?

La posición adquirida con respecto a la primera pregunta es que debido al no determinismo implícito en la ejecución de las aplicaciones paralelas, las historias de ejecución en las pruebas siempre serán distintas. Debido a que en las ejecuciones normales (sin instrumentación) estas historias no suelen provocar bugs, el éxito o no de detección de errores no depende del algoritmo de detección sino más bien de cuando la herramienta pueda alterar la ejecución normal de la aplicación para que ocurran historias poco normales. En este sentido, una propuesta como AVIO-SA se encuentra en desventaja ya que la comparación no toma en cuenta que al instrumentar menos, también altera menos la ejecución, y en consecuencia es probable que la historia sea más *normal* que la historia de AVIO (es más probable que en la historia de AVIO-SA ocurran menos interleavings no serializables).

Para dar respuesta a la segunda pregunta se propuso comparar ambas versiones a partir de la misma historia de ejecución. Esto se consiguió desarrollando una herramienta de instrumentación cuya única función fue grabar el orden en que los threads de la aplicación acceden a cada dirección de memoria. Luego se modificó AVIO para forzar ese orden durante la ejecución. Con AVIO-SA fue necesario también simular la generación de eventos y el timer, ya que al incluir las demoras producto de forzar el orden de ejecución de las instrucciones los eventos y el intervalo de muestreo perdían sentido.

Para la experimentación se utilizó una aplicación más moderna llamada bodytrack. Esta aplicación se configuró para ser ejecutada con dos hilos y con el paquete de entrada para pruebas. Los resultados mostraron que ambas versiones detectan aproximadamente la misma cantidad de interleavings no serializables sobre la misma historia de ejecución.

En función de los resultados obtenidos, se concluye que el modelo propuesto no afecta negativamente la capacidad de detección de AVIO. Debido a que AVIO-SA altera menos la historia de ejecución de la aplicación monitorizada, es una mejor opción para ser utilizada en entornos de producción. Los resultados anteriores muestran que en caso de ocurrir el interleaving no serializable, el evento será generado y AVIO será activado, lo que permitirá detectarlo oportunamente.

Capítulo 8

Conclusiones y Líneas de Trabajo Futuras

Esta tesis planteó como objetivo principal *proponer un modelo de implementación en software para herramientas de detección de errores de concurrencia, que permita reducir el overhead del proceso sin disminuir su capacidad de detección.*

Se definió un proceso de investigación guiado por objetivos específicos, a través de los cuales se transitó por diferentes niveles de profundidad del conocimiento, desde el perceptual hasta el integrativo, en el que se dio cumplimiento al objetivo planteado. A continuación, se enumeran los logros alcanzados con este proceso:

- Se definió un marco teórico de referencia sobre depuración de programas. Este marco permitió comprender las características que hacen únicos a los errores de concurrencia, determinar los motivos por los que los métodos tradicionales de depuración no son adecuados para estos errores y definir un grado de importancia a partir la frecuencia con que ocurren. Como resultado de esta etapa se definió que la violación de atomicidad simple sería el caso de estudio que guiaría el resto del trabajo de tesis.

- Se realizó un completo análisis de antecedentes en la comunidad científica sobre detección de violaciones de atomicidad. Se compararon las técnicas y métodos vigentes aplicados a la detección y corrección de estos errores. Este proceso reveló que AVIO era la herramienta con mejor desempeño y capacidad de detección que podía ser ejecutada completamente en software. Es importante destacar que para poder continuar con las siguientes etapas de la tesis fue necesario desarrollar una implementación propia de esta herramienta, debido a que no está disponible su código

fuente ni existe una implementación comercial de la misma. Esta implementación fue validada a partir de la capacidad de detección y rendimiento contra los datos presentes en el trabajo original. Los resultados mostraron que la implementación propia era equivalente a AVIO, ya que se consiguió reproducir los mismos resultados reportados.

- Se compararon y estudiaron los distintos enfoques de implementación utilizados por las propuestas analizadas. Esto permitió dividir las propuestas en dos grandes grupos: las que implementan sus algoritmos en hardware a través de propuestas de modificación y las que implementan sus algoritmos en software. Aunque las implementaciones hardware tienen un gran rendimiento, están limitadas sólo a entornos de investigación, excluyendo a todas las plataformas que actualmente se encuentran en producción. Por su parte las implementaciones software pueden ser utilizadas en cualquier plataforma, pero su rendimiento es muy pobre. Se definió que la propuesta del modelo debía integrar los aspectos más relevantes de ambos métodos: una implementación completamente en software pero haciendo uso del soporte hardware disponible en las plataformas actuales que pueda ayudar a reducir el overhead de las herramientas. En consecuencia, se estudió el modelo de ejecución de procesos en arquitecturas multicore, incluyendo las características de soporte del sistema operativo, en búsqueda de aspectos relevantes que pudieran ser utilizados para mejorar el rendimiento.

- Se estudiaron técnicas de instrumentación dinámica para poder analizar las causas que generan el overhead de las herramientas de detección. Este trabajo permitió identificar como causa principal del overhead a la granularidad de la instrumentación, que en el caso de las herramientas de detección de errores como AVIO es a nivel de instrucción.

- Se identificó como oportunidad de optimización la reducción de la instrumentación a nivel de instrucción. Para ello se estudió el comportamiento de distintas aplicaciones en relación al acceso a datos compartidos. Este proceso requirió el estudio de los contadores hardware como recurso para acceder a información en el procesador sobre la ejecución de los programas. Como resultado se determinó que sería posible acceder a información en el protocolo de coherencia cache para detectar los momentos de la ejecución donde se accede a datos compartidos.

- Se diseñó un modelo general de instrumentación dinámica que permite activar una rutina de análisis (en el caso de estudio un algoritmo de detección de violaciones de atomicidad) a partir de una señal generada por un evento de hardware, el cual indica la posibilidad de ocurrencia de errores. Para ello fue necesario definir un método de validación de eventos que permitiera asegurar que un evento candidato fuera adecuado para el propósito del trabajo. La validación implicó analizar la frecuencia y distribución de eventos y errores en la ejecución de distintas aplicaciones.

- Se desarrolló una versión de AVIO con el modelo propuesto y se evaluó su efectividad en base al overhead y la capacidad de detección de errores. Los resultados mostraron que la nueva versión es capaz de detectar los mismos bugs que AVIO pero empleando para ello (en promedio) sólo la cuarta parte del tiempo que requiere la versión original.

8.1. Líneas futuras

Los resultados finales muestran que el objetivo general ha sido cumplido. Sin embargo, también abren la puerta a nuevas líneas de trabajo derivadas de esta tesis:

- Factibilidad de aplicación del modelo para algoritmos de detección de otra clase de errores. Esta tesis plantea como caso de estudio las violaciones de atomicidad por tratarse del tipo de error más general y frecuente luego de los deadlocks. Es de interés analizar si algoritmos de detección de otros tipos de errores pueden beneficiarse de las técnicas tratadas en este trabajo.

- Métodos de comparación fiables para propuestas de detección de errores. Una dificultad encontrada durante la experimentación fue que el método empleado por la comunidad científica para validar propuestas carece de precisión. Las herramientas de detección se evalúan a partir de bugs conocidos detectados en aplicaciones reales, pero también sobre la base de suites de benchmarks como la empleada en este trabajo. Actualmente el método empleado para esta evaluación consiste en realizar suficientes ejecuciones de las aplicaciones a través de la herramienta como para garantizar que los resultados gozan de cierta *estabilidad estadística*. Sin embargo, en propuestas como la de esta tesis donde el enfoque seguido deriva en reducir la interferencia en las historias de ejecución de las aplicaciones, las diferencias entre la historia que analizan las distintas propuestas pueden ser muy grandes. Al final del

Capítulo 7 se propone una discusión sobre esta observación y un método alternativo para realizar las comparaciones entre herramientas. Sin embargo, este método propuesto requiere aún de análisis, por lo que no se lo incluyó como contribución de la presente tesis.

- Herramientas para forzado de historias potencialmente defectuosas. Derivado del punto anterior, sería muy útil contar con herramientas que ayuden en la depuración de los programas. En particular es de interés plantear algoritmos de planificación que den soporte desde el sistema operativo a las herramientas de detección.

- Métodos alternativos para reducción del overhead de la instrumentación. Algunos experimentos derivados de esta tesis indican que buena parte del overhead introducido por AVIO (cerca del 50 %) se debe al mecanismo utilizado para garantizar atomicidad en la ejecución de las rutinas de análisis. El estudio de mecanismos de sincronización más eficientes para las herramientas de instrumentación dinámica podría significar una mejora importante no solo para AVIO, sino para otras herramientas que compartan con AVIO las características de instrumentación.

- Instrumentación dinámica y perfiles de aplicaciones paralelas. Otra de las dificultades encontradas en esta tesis fue que algunos resultados requieren explicar los patrones de interacción de los threads de las aplicaciones paralelas para sostener o refutar hipótesis. En estos casos el procedimiento seguido fue analizar el código fuente de la aplicación, lo que implicó una cantidad de tiempo y esfuerzo considerable. Las técnicas de instrumentación dinámica permiten descubrir aspectos sobre las estructuras de los programas sin necesidad de recurrir al código fuente de la aplicación. La definición de criterios que ayuden a construir herramientas de este tipo y explicar patrones de las aplicaciones sería de gran utilidad para el análisis de esta clase de propuestas.

Apéndices

Apéndice A

Detección de condiciones de carrera

Todos los algoritmos para detectar condiciones de carrera están basados en dos estrategias principales (o alguna variación de éstas), llamadas *happens before* y *lockset*.

A.1. Happens before

En programas secuenciales, si la instrucción b se encuentra en el código después de la instrucción a, entonces se puede afirmar que a ocurre antes que b. Esta noción de orden que permite estudiar el comportamiento de los programas secuenciales, se pierde si las instrucciones a y b pertenecen a procesos diferentes. Sin embargo, se puede establecer un "orden parcial" entre segmentos de los procesos a través de sus sentencias de sincronización (Lamport, 1978). Considere el caso de la Figura A.1.

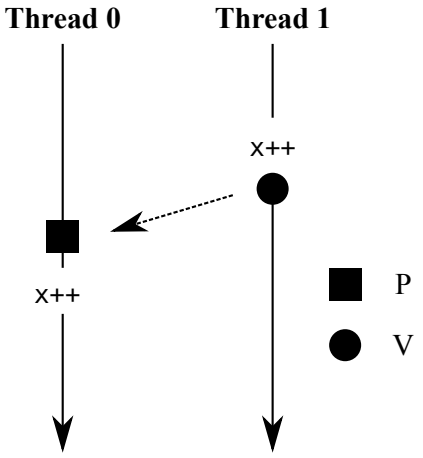


Figura A.1: Noción de orden parcial entre segmentos de procesos.

En este ejemplo, aunque ambos threads escriben en la variable x, se puede estar seguro de que ambas escrituras nunca ocurrirán en paralelo: hasta que el *Thread 1* libere el semáforo, el segmento de código posterior a la instrucción P en el *Thread 0* no se ejecutará. Esta relación identificada por el símbolo "$\rightarrow$" se conoce como *happens before*: la expresión $a \rightarrow b$ significa que a ocurrió antes que b, y por lo tanto se puede estar seguro que las instrucciones que preceden a a no se ejecutarán nunca en paralelo con las instrucciones que siguen a b. La importancia de este concepto radica en que permite diferenciar con seguridad cuáles segmentos de código son *potencialmente concurrentes*[1] de los que no.

La estrategia clásica para identificar estos segmentos consiste en mantener para cada thread un vector de relojes lógicos (Fidge, 1991; Mattern, 1989), el cual tiene tantos elementos como número de threads en el programa. Cada vez que un thread libera un semáforo, incrementa el elemento del vector que representa su reloj; cada vez que un thread adquiere un semáforo, incrementa el elemento del vector que representa su reloj y para cada elemento restante, deja el mayor valor entre el elemento con su equivalente en el vector del thread que liberó el semáforo. Para aclarar esta idea, considere el ejemplo de la Figura A.2.

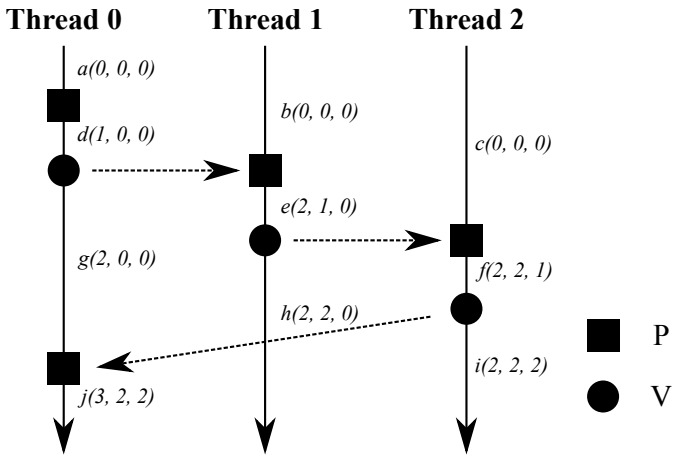


Figura A.2: Detección de segmentos potencialmente concurrentes a través del uso de vectores de relojes lógicos.

En la figura pueden observarse tres threads divididos en segmentos por sus sentencias de sincronización[2]:

[1] Se debe recordar que la ejecución en paralelo de los segmentos dependerá del no determinismo inherente a la ejecución

[2] Para simplificar el ejemplo, las sentencias de sincronización corresponden a operaciones P y V sobre el mismo semáforo

- El *Thread 0* se divide en los segmentos a, d, g, j
- El *Thread 1* se divide en los segmentos b, e, h
- El *Thread 2* se divide en los segmentos c, f, i

A su vez, cada segmento posee un vector de contadores que es actualizado en cada sentencia de sincronización según el método descrito anteriormente. Entonces, para determinar si dos segmentos son potencialmente concurrentes, basta con encontrar aquellos que no cumplen con la relación *happens before*: por ejemplo, la relación $d \rightarrow e$ se cumple porque se verifica que los relojes que corresponden al *Thread 0* y *1* en el segmento d son menores que los relojes que corresponden a los mismos threads en el segmento e. De igual manera, si se compara el segmento e con el segmento c se comprobará que no se cumple la relación: $c \not\rightarrow e$ debido a que aunque *c[2]* es menor que *e[2]*, no ocurre lo mismo entre *c[3]* y *e[3]*, por lo que los segmentos c y e son potencialmente concurrentes.

Este método se ha utilizado con éxito para detectar condiciones de carrera en numerosos trabajos. La técnica consiste en guardar un registro para cada operación de lectura o escritura que ocurre en cada segmento que contiene el vector de relojes de su proceso cuando la operación ocurrió. Luego, para detectar una condición de carrera basta con encontrar escrituras y lecturas sobre las mismas variables en segmentos que no cumplen la relación *happens before*.

Esta estrategia tiene dos problemas principales:

1. Es poco eficiente: requiere mantener información por cada thread sobre los accesos a datos compartidos. Esto vuelve la técnica costosa en tiempo de ejecución y en uso de memoria.

2. Su efectividad depende del planificador del sistema operativo: Debido a que la estrategia está basada en el orden de ejecución de los segmentos, podría ocurrir que en la mayoría de los casos ocurran historias válidas, como el caso de la izquierda la Figura A.3. En este, se puede verificar que los segmentos a y f cumplen la relación *happens before* y, por lo tanto, no hay riesgo de que se produzca una condición de carrera sobre la variable x. Sin embargo, eventualmente podría ocurrir el caso de la derecha donde la relación *happens before* entre los segmentos a y f no se cumple.

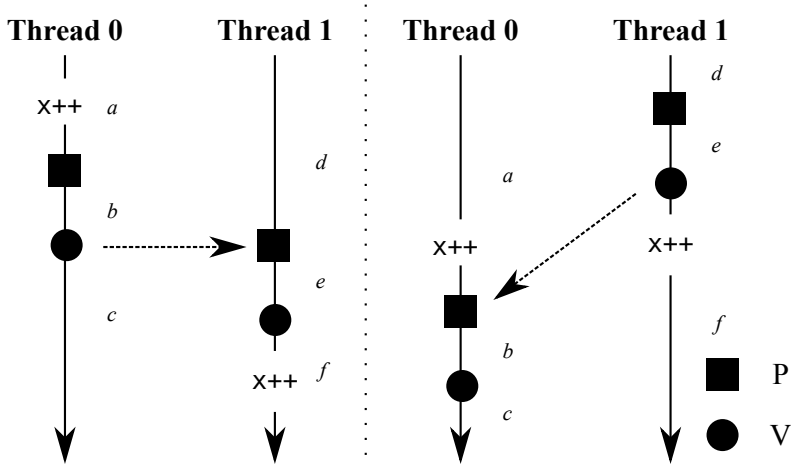


Figura A.3: En el ejemplo se muestran dos casos de ejecución, donde la condición de carrera sólo se presenta en el caso de la derecha.

A.2. Lockset

Las condiciones de carrera sólo pueden ocurrir entre accesos concurrentes a una variable compartida sobre la que no se garantiza exclusión mutua. La estrategia de Lockset (Savage y cols., 1997) consiste en asegurar que toda variable compartida se encuentra protegida ante cualquier acceso de cualquier thread. La manera más simple de proteger una variable es encerrarla en un semáforo s entre las operaciones $P(s)$ y $V(s)$. Debido a que es imposible saber a priori qué semáforo protege a qué variable, el algoritmo prevé que la relación de protección entre semáforos y variables se derivará de la ejecución del programa.

Para cada variable compartida x, se mantiene un conjunto $C(x)$ de semáforos candidatos para x. Este conjunto contiene cada semáforo para el que se hace una operación de P/V sobre la variable x. Cada vez que se accede a la variable x, el algoritmo actualiza el conjunto C con la intersección de $C(x)$ y el conjunto de semáforos activos en el thread actual. Si ocurre algún acceso que no está protegido, entonces el resultado de la intersección será un conjunto vacío, lo que significa que no hay un semáforo que proteja consistentemente a la variable x. Para aclarar esta idea, considere el ejemplo de la Figura A.4.

En la Figura se debe notar la interacción entre tres thread, donde cada uno de ellos actualiza una variable x. También se puede observar que tanto el *Thread 0* como el *Thread 2*, protegen la variable x con el semáforo *s1*. Por otro lado, el *Thread 1* utiliza un semáforo diferente llamado *s2*. Además en la última columna de la figura, se encuentra el conjunto

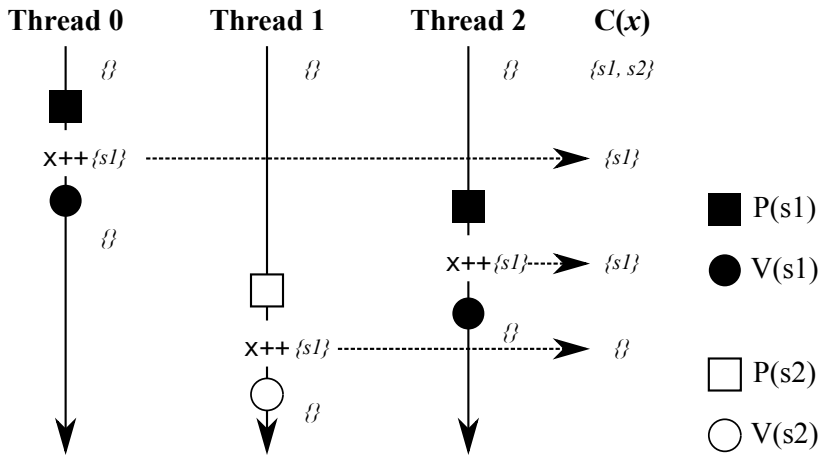


Figura A.4: Detección de condiciones de carrera con la estrategia lockset.

C(x) y en él se reflejan las actualizaciones correspondientes ante cada acceso a la variable x. En este ejemplo se puede notar que tanto el *Thread 0* como el *Thread 2*, protegen adecuadamente a x, ya que *C(x)* permanece lleno; sin embargo, ante la operación que realiza el *Thread 1*, el conjunto se vacía. En este momento, el algoritmo debe notificar una posible condición de carrera sobre la variable x.

Esta estrategia es más general que *happens before*, ya que detecta condiciones de carrera independientemente del orden de ejecución (se debe notar que esta estrategia detectará la condición de carrera de la Figura A.3). Sin embargo, esta independencia convierte este método en una estrategia demasiado conservativa: al no mantener información sobre el orden de ejecución de los segmentos, informará falsos positivos en situaciones donde la exclusión mutua se garantiza por la sincronización entre procesos, en lugar de pares P/V.

Considere el caso de la Figura A.1 aunque la variable x no se encuentra protegida por un par P/V en cada proceso, no importa la planificación que haga el sistema operativo: siempre la escritura del *Thread 1* sobre x ocurrirá antes que la escritura del *Thread 0*. Un caso como éste será detectado como una condición de carrera por la estrategia lockset.

Apéndice B

Benchmaks

B.1. Introducción

Esta sección presenta los benchmarks utilizados para evaluar la capacidad de detección y el desempeño a partir del rendimiento de las herramientas empleadas en esta tesis.

B.1.1. Benchmarks para evaluar capacidad de detección

Para evaluar la capacidad de detección de las herramientas se diseñaron cuatro kernels de ejecución, uno para cada caso de interleaving *no serializable* conocido. La Tabla B.1 resume cada kernel y el caso de interleaving para el que fue diseñado.

Cuadro B.1: Kernels diseñados para exponer los casos de interleavings conocidos.

kernel	**interleaving**
APACHE	Case 2
MOZILLA	Case 3
MYSQL	Case 5
BANKACCOUNT	Case 6

Estos kernels corresponden a los ejemplos de interleavings enumerados en el trabajo de AVIO con dos ligeras modificaciones: 1) poseen instrucciones de *delay* ubicadas de tal manera que aumentan las posibilidades de que el interleaving se manifieste; 2) a través de una variable de entorno se puede activar un mecanismo que asegura la atomicidad de los bloques de instrucciones conflictivos. Estas modificaciones permiten disminuir el número de experimentos necesarios para comprobar la capacidad de detección de las herramientas evaluadas. A continuación se ofrece una descripción de cada kernel y la explicación del bug que representan.

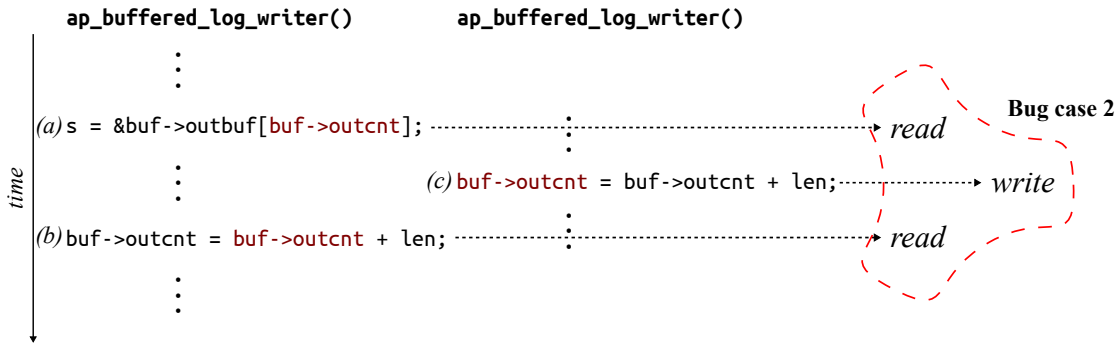


Figura B.1: Ejemplo de una violación de atomicidad en Apache

APACHE. Este kernel simula un error real de la aplicación apache, presente hasta la versión 2.0.48. El error aparece cuando varios threads llaman a la función *ap_ buffered_ log _ writer()* con la intención de preservar sus logs en el archivo *buf*. Esta función luego de verificar que hay espacio suficiente para almacenar los logs, recorre el arreglo *strs* (donde están almacenados) y copia las cadenas al buffer *buf*. La Figura B.1 ilustra la ejecución en el tiempo de la situación que desencadena el error.

La estructura *buf* posee un miembro *outcnt* que es un índice al final del buffer. Cada llamada a la función *ap_ buffered_ log_ writer()* guarda en un puntero privado *s* la dirección del final del buffer (instrucción *a*) y luego agrega a éste los elementos del arreglo de logs *strs*. Finalmente actualiza el valor de *outcnt* (instrucción *b*) para apuntar al final del buffer. Si otro thread está ejecutando la misma función, podría ocurrir que la actualización del índice del final de buffer (instrucción *c*) ocurra entre las instrucciones que deberían haber sido atómicas. Como resultado el archivo de logs puede terminar con registros incompletos de logs. Se debe notar que además de la lectura correspondiente al segundo acceso del primer thread que da lugar al bug, también existe una escritura que daría lugar a otro caso. Sin embargo, como la lectura ocurre antes y provoca en si misma un bug de caso 2, no haremos referencia a este segundo bug.

MOZILLA. Este kernel simula un error real de la aplicación mozilla. Este error se produce por la manipulación entre threads de la variable compartida *gCurrentScript*. La Figura B.2 resume la situación que desencadena el error.

El thread uno se encuentra procesando un script. Para ello, primero lo carga en una variable compartida *gCurrentScript* (instrucción *a*) y luego intenta compilarlo (instrucción *b*). Mientras esto ocurre, el thread dos asigna el valor *NULL* a la variable compartida (instrucción *c*). Como resultado la aplicación se cuelga cuando intenta compilar el script.

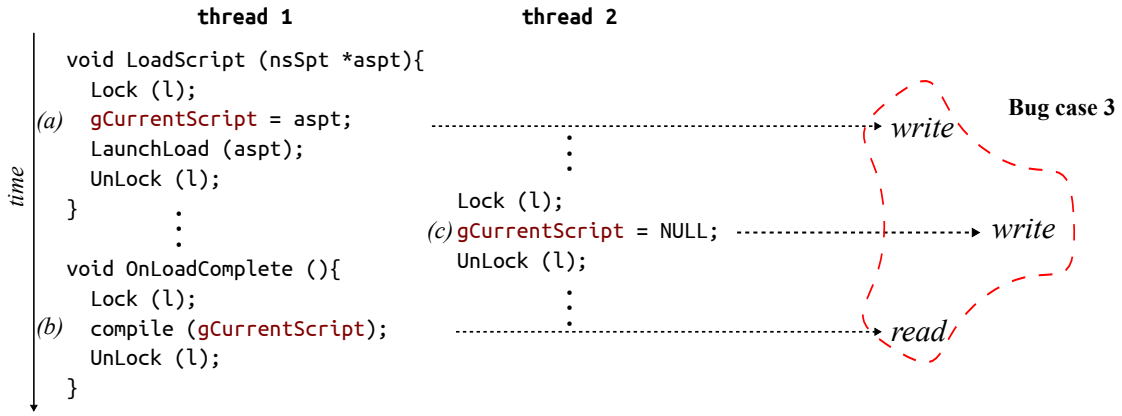


Figura B.2: Ejemplo de una violación de atomicidad en Mozilla.

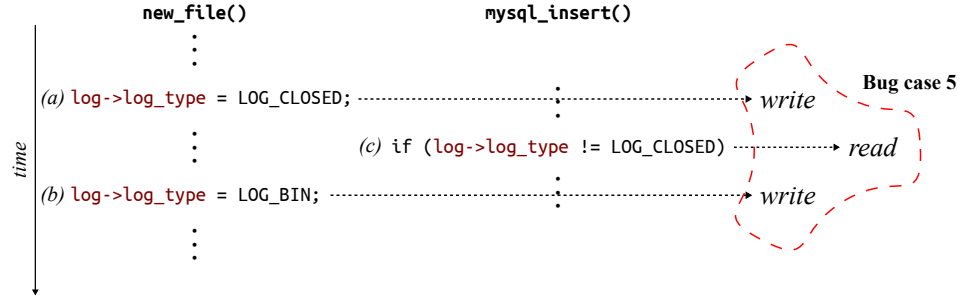


Figura B.3: Ejemplo de una violación de atomicidad en MySQL

Cabe destacar que en este ejemplo el programador protegió cada acceso a *gCurrentScript* con el semáforo *l*. Se debe notar que los accesos se encuentran protegidos, pero la granularidad de las acciones atómicas es incorrecta. Esto evita que se produzca una condición de carrera sobre esta variable, pero como queda de manifiesto en el ejemplo, eso no garantiza la ausencia de errores de concurrencia.

MYSQL. En este kernel se presenta un error detectado en el motor de base de datos mysql. El bug que se describe a continuación fue corregido en la versión 4.0.14. La Figura B.3 resume las instrucciones involucradas.

La situación que desencadena el bug se presenta cuando uno de los threads intenta cerrar el archivo de logs y crear uno nuevo a través de la función *new_file()*. El estado del archivo (ABIERTO o CERRADO) se guarda en un miembro *log_type* de la estructura *log*. La operación se lleva a cabo entre varias operaciones, donde primero se actualiza el valor de *log_type* con *LOG_CLOSED* (instrucción *(a)*) y luego de que el nuevo archivo está listo para su uso se vuelve a actualizar su valor con *LOG_BIN* (instrucción *(b)*). Puede ocurrir

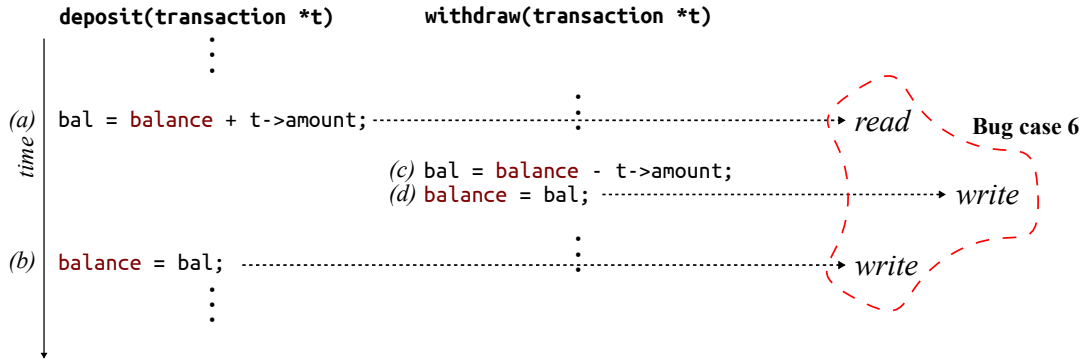


Figura B.4: Ejemplo de una violación en una cuenta bancaria

que en medio de estos eventos otro thread se encuentre ejecutando *mysql_insert()*. Esta función determina si debe registrar un log de la operación a través del *if* de la instrucción *(c)*. Debido a que temporalmente el valor de *log_type* es *LOG_CLOSED*, la operación que modifica la base de datos no será registrada en el archivo de logs, lo que constituye un problema de seguridad.

BANKACCOUNT. El ejemplo de la cuenta bancaria simula realizar varias transacciones en una cuenta bancaria. A tal efecto, el programa está diseñado para ejecutar todas las transacciones que corresponden a depósitos desde un thread, y las que corresponden a extracciones desde otro. La cuenta bancaria es una variable compartida llamada *balance*. Cada thread guarda el valor de *balance* en una variable privada *bal*, ejecuta la transacción y luego asigna el resultado nuevamente en *balance*. La Figura B.4 ilustra la ejecución en el tiempo de varias de estas transacciones.

En la figura las instrucciones *a* y *b* corresponden a la la función *deposit()*, ejecutada por el *Thread 0*. Por otro lado, las instrucciones *c* y *d* corresponden a la función *withdraw()* ejecutada por el *Thread 1*. El bug se manifiesta cuando la escritura de la variable compartida en *withdraw* (instrucción *d*) ocurre en medio de la transacción *deposit* (instrucciones *a* y *b*). Como resultado la transacción correspondiente a *withdraw* se pierde, aunque el programa asume que se completó con éxito.

B.1.2. Benchmarks para desempeño

Para evaluar el desempeño de las herramientas propuestas se utilizó la suite de benchmarks SPLASH-2 (Woo y cols., 1995). La Tabla B.2 resume los programas que componen la suite y las configuraciones empleadas en cada caso.

Cuadro B.2: Programas que componen la suite SPLASH-2 y configuraciones por defecto utilizadas.

program	**type**	**size**
barnes	apps	16K particles
fmm	apps	16K particles
ocean_cp	apps	258 x 258 ocean
ocean_ncp	apps	258 x 258 ocean
radiosity	apps	room, -ae 5000.0 -en 0.050 -bf 0.10
raytrace	apps	car
volrend	apps	head
water_nsq	apps	512 molecules
water_sp	apps	512 molecules
cholesky	kernels	tk15.0
fft	kernels	64K points
lu_cb	kernels	512x512 matrix, 16 x 16 blocks
lu_ncb	kernels	512x512 matrix, 16 x 16 blocks
radix	kernels	1M integer, radix 1024

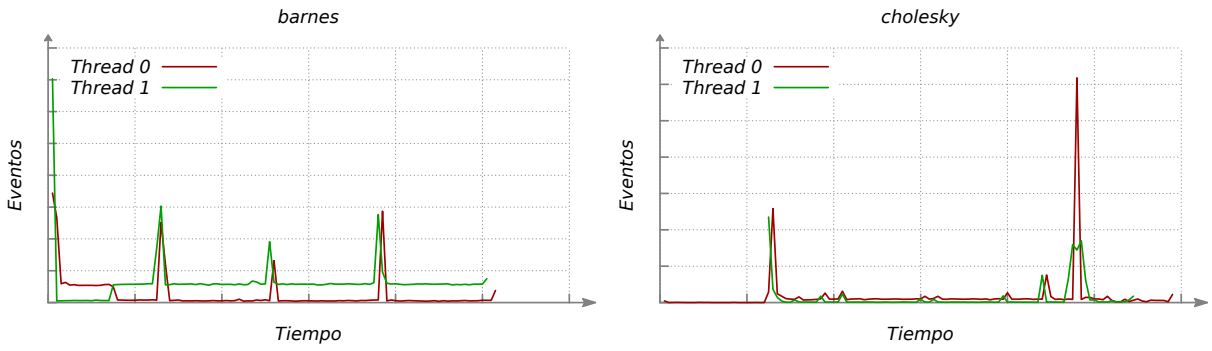
SPLASH-2 es probablemente la suite más utilizada para estudios científicos en máquinas paralelas con memoria compartida. Esta suite fue empleada en el artículo original de AVIO y en otros trabajos relacionados (Yu y Narayanasamy, 2009; Muzahid y cols., 2010; S. Park y cols., 2009; Deng y cols., 2013). Contiene nueve aplicaciones y cinco kernels, todos considerados sin errores.

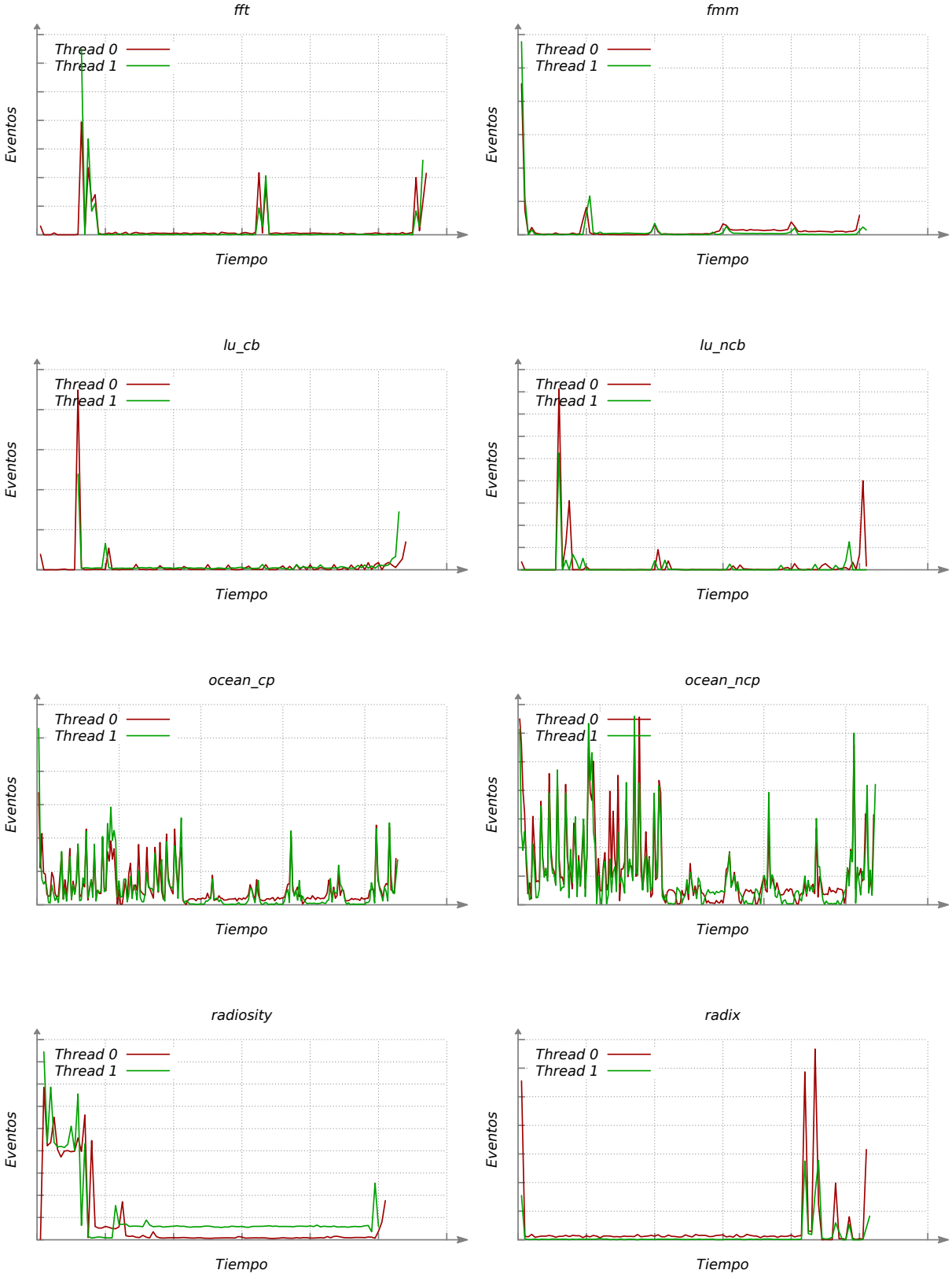
Apéndice C

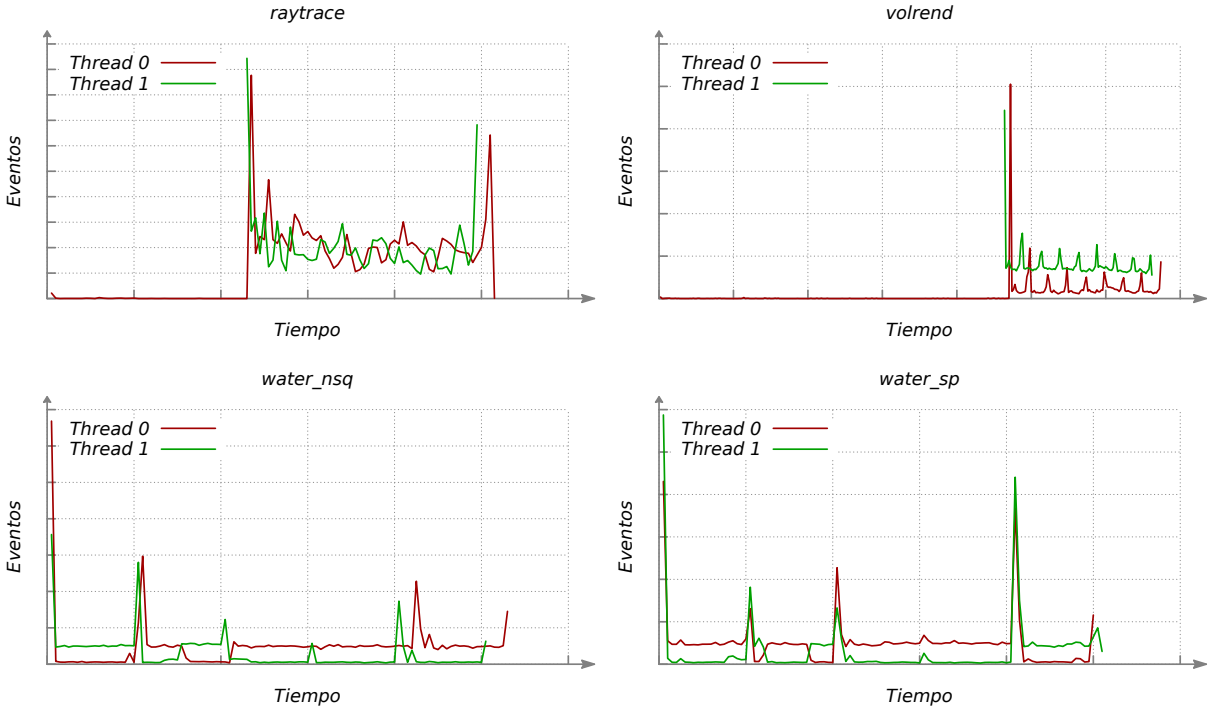
SPLASH-2: distribución de eventos

C.1. Introducción

Este anexo muestra las figuras correspondientes a la distribución de eventos sobre la ejecución de cada una de las aplicaciones de la suite SPLASH-2. Los resultados son complementarios al experimento de la Sección 5.6.3 (Patrones de accesos a datos compartidos). El evento utilizado es MEM_UNCORE_RETIRED:LOCAL_HITM, el cual indica en su descripción que cuenta instrucciones *loads* que aciertan en datos en estado modificado en un *sibling core* (otro núcleo en el mismo procesador). Este evento ocurre con la operación de lectura del patrón de acceso 2 ($W \rightarrow R$) presente en 3 de los cuatro casos de interleavings no serializables.




fft
fmm
lu_cb
lu_ncb
ocean_cp
ocean_ncp
radiosity
radix
Thread 0
Thread 1
Eventos
Tiempo

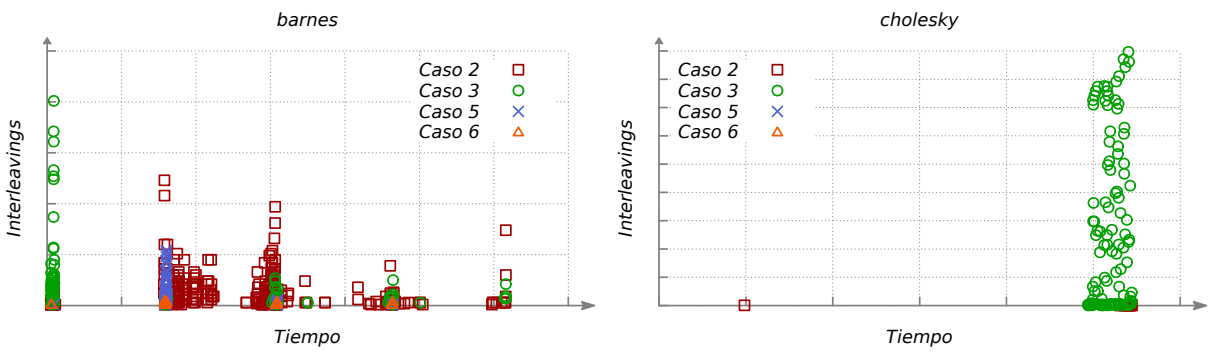

raytrace
Thread 0
Thread 1
Eventos
Tiempo
volrend
Thread 0
Thread 1
Eventos
Tiempo
water_nsq
Thread 0
Thread 1
Eventos
Tiempo
water_sp
Thread 0
Thread 1
Eventos
Tiempo

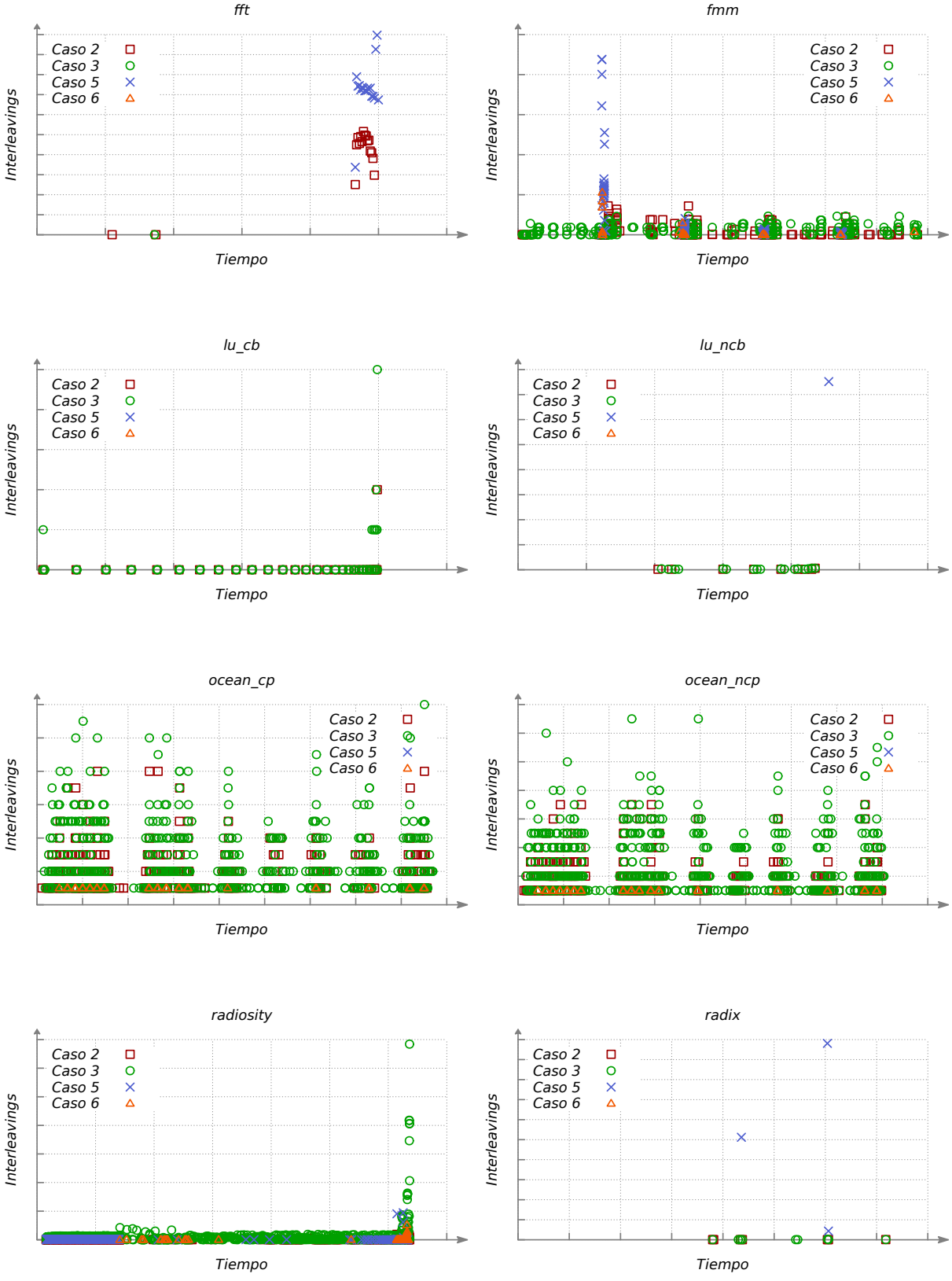
Apéndice D

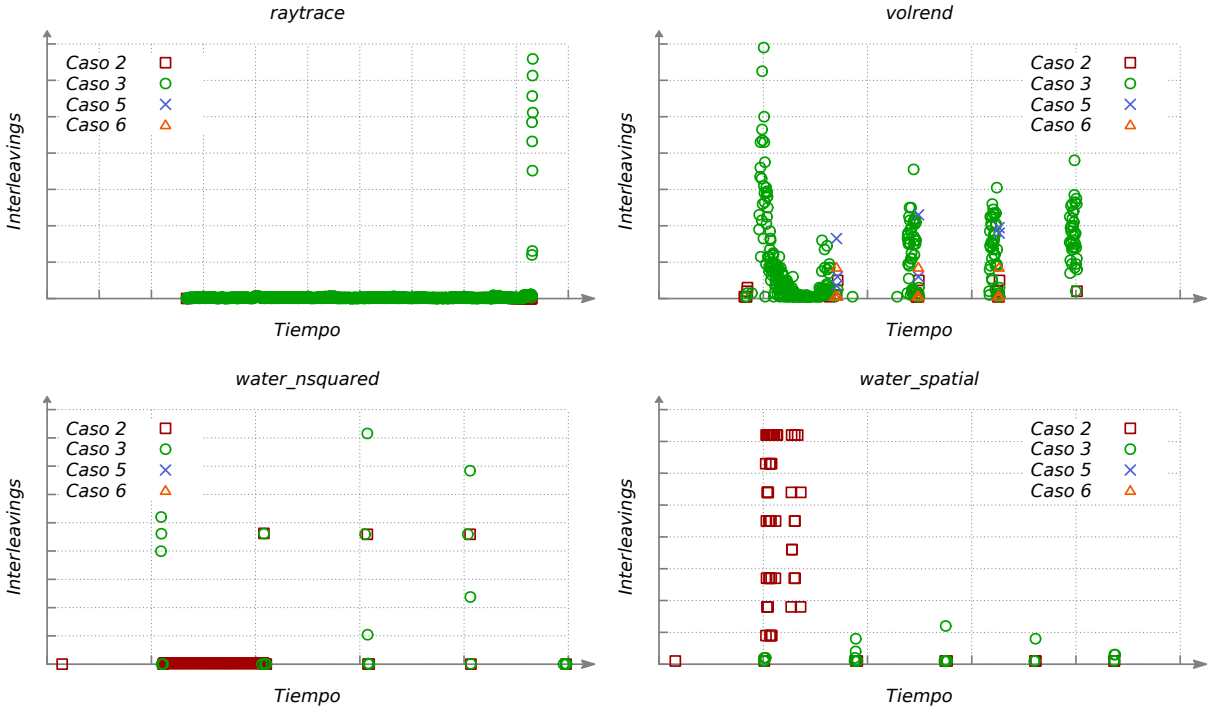
SPLASH-2: distribución de interleavings

D.1. Introducción

Este anexo muestra las figuras correspondientes a la distribución de interleavings no serializables sobre la ejecución de cada una de las aplicaciones de la suite SPLASH-2. Los resultados son complementarios al experimento de la Sección 5.6.4 (Frecuencia y distribución de interleavings no serializables).




fft
fmm
lu_cb
lu_ncb
ocean_cp
ocean_ncp
radiosity
radix
Caso 2
Caso 3
Caso 5
Caso 6
Interleavings
Tiempo


raytrace
volrend
water_nsquared
water_spatial
Caso 2
Caso 3
Caso 5
Caso 6
Interleavings
Tiempo

Apéndice E

Apache

E.1. Introducción

Apache es un servidor web de código abierto y multiplataforma. Es uno de los servidores web más populares entre usuarios, llegando a superar el 40 % de uso en el mercado (netcraft, 2015). Por su condición de software de código abierto ha sido ampliamente utilizado en trabajos de investigación y gracias a ello muchos de sus bugs se encuentran documentados. En particular en este trabajo es de interés el bug explicado en la Figura B.1 (Sección B.1.1), documentado en el sistema de bugtraking de apache bugzilla con el código 25520 (Sussman, 2003). Sin embargo, la versión a la que hace referencia el bug tiene más de 10 años de antigüedad. Debido a que muchas librerías en el sistema operativo han cambiado (incluido el compilador), compilar la versión original del error puede ser muy complicado, e incluso de lograrlo, puede ser más difícil aún ejecutar esa versión en un sistema actual.

Afortunadamente, aunque pasó mucho tiempo desde aquella versión, la rutina con el error documentado prácticamente no ha cambiado desde entonces. De hecho la rutina estudiada en esta tesis de la versión actual (2.2.29) de apache se puede convertir en la rutina defectuosa de la versión (2.0.48) simplemente comentando unas pocas líneas. Este apéndice muestra las modificaciones hechas a la versión actual de apache para reproducir el bug y describe el procedimiento empleado para su compilación y posterior ejecución.

E.2. Compilación e instalación

El código fuente de apache puede ser descargado de www.apache.org. Al momento de escribir esta tesis, la última versión estable disponible es la 2.2.29. Para reproducir el error se modificó la función `ap_buffered_log_writer` ubicada en el archivo `modules/loggers/mod_log_config.c` para ser equivalente a la versión de la misma función documentada en bugzilla. El Listado E.1 muestra la función modificada.

Listado E.1: función modificada de apache para permitir la manifestación del bug.

```
1  static apr_status_t ap_buffered_log_writer(request_rec *r,
2                                             void *handle,
3                                             const char **strs,
4                                             int *strl,
5                                             int nelts,
6                                             apr_size_t len)
7
8  {
9      char *str;
10     char *s;
11     int i;
12     apr_status_t rv;
13     buffered_log *buf = (buffered_log*)handle;
14
15 //     if ((rv = APR_ANYLOCK_LOCK(&buf->mutex)) != APR_SUCCESS) {
16 //         return rv;
17 //     }
18
19     if (len + buf->outcnt > LOG_BUFSIZE) {
20         flush_log(buf);
21     }
22     if (len >= LOG_BUFSIZE) {
23         apr_size_t w;
24
25         str = apr_palloc(r->pool, len + 1);
26         for (i = 0, s = str; i < nelts; ++i) {
```

```
27              memcpy(s, strs[i], strl[i]);
28              s += strl[i];
29          }
30          w = len;
31          rv = apr_file_write(buf->handle, str, &w);

33      }
34      else {
35  ////////////////////////////////////////////////////////////////////
36          int k;
37          for (k = 0; k < 1000000; k++) ;
38  ////////////////////////////////////////////////////////////////////
39          for (i = 0, s = &buf->outbuf[buf->outcnt]; i < nelts; ++i) {
40              memcpy(s, strs[i], strl[i]);
41              s += strl[i];
42          }
43          buf->outcnt += len;
44          rv = APR_SUCCESS;
45      }

47  //      APR_ANYLOCK_UNLOCK(&buf->mutex);
48      return rv;
49  }
```

La modificación consistió en comentar las operaciones `lock` y `unlock` que evitan que el bug ocurra. Además, para facilitar la manifestación del bug se introdujo un *delay* antes de la primera instrucción del `else` de la línea 34. En la carpeta de los fuentes de apache (en cualquiera de sus versiones) se encuentra un script ejecutable llamado configure. Al ejecutar este archivo, el programa se asegura de que en el entorno de desarrollo del usuario existen las librerías necesarias para compilar apache y permite personalizar la compilación de acuerdo a los requerimientos del usuario. Para los propósitos de este trabajo de tesis se usan dos parámetros de configuración:

- `prefix`: permite modificar el directorio de instalación por defecto de apache.

- `with-mpm`: permite seleccionar el módulo de multiprocesamiento que será utilizado por apache. En este caso se utilizará el modo `worker`, que configura apache para implementar un servidor basado en hilos en lugar de procesos.

Además para permitir la depuración posterior del código se deben configurar las variables de entorno `CFLAGS` y `CPPFLAGS` con el valor "-g". Una vez que el proceso de configuración acaba exitosamente, se utiliza el comando `make` para compilar y posteriormente instalar apache en la carpeta indicada. El Listado E.2 muestra un ejemplo de este procedimiento.

Listado E.2: configuración inicial de apache.

```
1 $ cd APACHE_SOURCE_FOLDER
2 $ export CFLAGS=-g
3 $ export CPPFLAGS=-g
4 $ ./configure --prefix=/INSTALATION_FOLDER --with-mpm=worker
5 $ make
6 $ make install
```

E.3. Configuración

Para que la función sea utilizada es necesario configurar apache para que use *buffering* en el proceso de registrar los accesos en el archivo `access_log`. Para ello se debe agregar la directiva correspondiente al archivo `conf/httpd.conf`.

Es necesario configurar el puerto sobre el que atenderá consultas apache: de esta manera se asegura que las peticiones webs realizadas durante las pruebas no interferirán con algún otro servidor que esté actuando en el mismo puerto.

A los efectos de este trabajo también es necesario configurar el módulo MPM. El Listado E.3 muestra las líneas que deben ser modificadas en el archivo `conf/httpd.conf` para realizar estas configuraciones.

Se debe quitar el carácter de comentario "#" de la directiva `Include` de la línea 7.

Finalmente, para configurar el número de procesos e hilos que serán utilizados en las ejecuciones de apache hay que modificar el archivo `conf/extra/httpd-mpm.conf`. La sección correspondiente al modo `worker` debe quedar como en el Listado E.4.

Listado E.3: configuraciones en el archivo http.conf de apache.

```
...
BufferedLogs on
...
Listen 8080
...
# Server-pool management (MPM specific)
Include conf/extra/httpd-mpm.conf
...
```

Listado E.4: configuración necesaria para limitar el número de hilos y procesos creados por apache.

```
<IfModule mpm_worker_module>
    StartServers          1
    MaxClients            4
    MinSpareThreads       1
    MaxSpareThreads       4
    ThreadsPerChild       4
    MaxRequestsPerChild   0
</IfModule>
```

E.4. Ejecución

Para ejecutar apache se usa el programa `httpd` ubicado en la carpeta `bin/` dentro del directorio de instalación. Una vez que el servidor se ha iniciado y ha completado algunas tareas preliminares, lanzará varios procesos hijo que hacen el trabajo de escuchar y atender las peticiones de los clientes. El proceso principal continúa ejecutándose como `root`, pero los procesos hijo se ejecutan con menores privilegios de usuario. El programa `httpd` debe ser ejecutado con los argumentos `-X` y `-k`, tal como se muestra en el Listado E.5.

Listado E.5: modo de iniciar apache para los casos de uso experimentales de este trabajo de tesis.

```
$ ./bin/httpd -X -k start
```

El argumento “k” se utiliza para indicar la acción a realizar con el programa. El argumento “X” fuerza a apache a ser ejecutado en modo depuración. En este modo sólo un proceso `worker` será iniciado y el servidor no perderá la conexión con la consola.

Para realizar las pruebas que provocan que el bug se manifieste, desde otra terminal se debe ejecutar el script del Listado E.6.

Listado E.6: script para generar las consultas al servidor que provocarán la manifestación del bug.

```
1 #!/bin/bash
2 i=0
3 while [[ $i -le 1000 ]]
4   do
5    wget localhost:8080/index.html.es  &
6    wget localhost:8080/index.html.es
7    rm index.html.*
8     let "i=$i+1"
9   done
```

El script realiza dos consultas simultaneas al servidor por una página. El proceso se repite 1000 veces para facilitar la sobrecarga de consultas. Si el error ocurre, el archivo `logs/access_log` estará corrupto y contendrá basura.

Referencias

Ahmed, K., y Schuegraf, K. (2011, noviembre). Transistor wars. *IEEE Spectrum*, *48*(11), 50–66. doi: 10.1109/MSPEC.2011.6056626

Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., ... Weihl, W. E. (1997, noviembre). Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, *15*(4), 357–390. Descargado 2012-08-13, de http://doi.acm.org/10.1145/265924.265925 doi: 10.1145/265924.265925

Andrews, G. R. (2000). *Foundations of multithreaded, parallel, and distributed programming.* Addison Wesley.

Banerjee, U., Bliss, B., Ma, Z., y Petersen, P. (2006a). A theory of data race detection. En *Proceedings of the 2006 workshop on parallel and distributed systems: Testing and debugging* (p. 69–78). New York, NY, USA: ACM. Descargado 2014-01-21, de http://doi.acm.org/10.1145/1147403.1147416 doi: 10.1145/1147403.1147416

Banerjee, U., Bliss, B., Ma, Z., y Petersen, P. (2006b). Unraveling data race detection in the intel thread checker. En *In proceedings of STMCS '06.*

Bienia, C. (2011). *Benchmarking modern multiprocessors.* Tesis Doctoral no publicada, Princeton University, Princeton, New Jersey, USA.

Borkar, S., y Chien, A. A. (2011, mayo). The future of microprocessors. *Commun. ACM*, *54*(5), 67–77. Descargado 2015-02-06, de http://doi.acm.org/10.1145/1941487.1941507 doi: 10.1145/1941487.1941507

Deng, D., Zhang, W., y Lu, S. (2013). Efficient concurrency-bug detection across inputs. En *Proceedings of the 2013 ACM SIGPLAN international conference on object oriented programming systems languages & applications* (p. 785–802). New York, NY, USA:

ACM. Descargado 2014-01-24, de http://doi.acm.org/10.1145/2509136.2509539 doi: 10.1145/2509136.2509539

Dennard, R., Gaensslen, F., Rideout, V., Bassous, E., y LeBlanc, A. (1974, octubre). Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, *9*(5), 256–268. doi: 10.1109/JSSC.1974.1050511

Dijkstra, E. W. (1968). *Cooperating sequential processes* (Inf. Téc. n.º 123). Descargado de http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF

Eranian, S. (2006). Perfmon2: a flexible performance monitoring interface for linux. En *Proc. of the 2006 ottawa linux symposium* (pp. 269–288). Descargado 2012-08-13, de https://www.kernel.org/doc/ols/2006/ols2006v1-pages-269-288.pdf

Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., y Burger, D. (2012, mayo). Dark silicon and the end of multicore scaling. *IEEE Micro*, *32*(3), 122–134. doi: 10.1109/MM.2012.17

Fidge, C. (1991, agosto). Logical time in distributed computing systems. *Computer*, *24*(8), 28 –33. doi: 10.1109/2.84874

Flanagan, C., y Freund, S. N. (2004, enero). Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, *39*(1), 256–267. Descargado 2013-07-08, de http://doi.acm.org/10.1145/982962.964023 doi: 10.1145/982962.964023

Flanagan, C., y Qadeer, S. (2003). A type and effect system for atomicity. En *Proceedings of the acm sigplan 2003 conference on programming language design and implementation* (pp. 338–349). ACM. Descargado de http://doi.acm.org/10.1145/781131.781169

Gantz, J., y Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*. Descargado 2014-11-20, de http://www.emcchannel.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf

Garner, B. D., Browne, S., Dongarra, J., Garner, N., Ho, G., y Mucci, P. (2000). A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, *14*, 189–204.

Grama, A., Gupta, A., Karypis, G., y Kumar, V. (2003). *Introduction to parallel computing - second edition*. Pearson Education and Addison Wesley.

Greathouse, J. L., Ma, Z., Frank, M. I., Peri, R., y Austin, T. (2011). Demand-driven software race detection using hardware performance counters. *SIGARCH Comput. Archit. News*, *39*(3), 165–176. Descargado de http://doi.acm.org/10.1145/2024723.2000084

Hurtado de Barrera, J. (2008). *¿cómo formular objetivos de investigación?* (segunda ed.). Caracas, Venezuela: Quirón S. A.

Intel® 64 and ia-32 architectures optimization reference manual (Inf. Téc.). (2012). Intel Corporation. Descargado de http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

Intel® 64 and ia-32 architectures software developer's manual (Inf. Téc.). (2012). Intel Corporation. Descargado de http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

Jannesari, A., Bao, K., Pankratius, V., y Tichy, W. (2009). Helgrind+: An efficient dynamic race detector. En *IEEE international symposium on parallel distributed processing, 2009. IPDPS 2009* (pp. 1–13). doi: 10.1109/IPDPS.2009.5160998

Kurzak, J., Buttari, A., Luszczek, P., y Dongarra, J. (2008, febrero). *The PlayStation 3 for high performance scientific computing* (MIMS Preprint). Descargado 2014-11-20, de http://eprints.ma.man.ac.uk/1022/

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, *21*, 558–565.

Lu, S., Park, S., Seo, E., y Zhou, Y. (2008). Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGARCH Comput. Archit. News*, *36*(1), 329–339.

Lu, S., Park, S., y Zhou, Y. (2012). Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Transactions on Software Engineering*, *38*(4), 844–860. doi: 10.1109/TSE.2011.35

Lu, S., Tucek, J., Qin, F., y Zhou, Y. (2006). AVIO: detecting atomicity violations via access interleaving invariants. *SIGPLAN Not.*, *41*(11), 37–48. doi: http://doi.acm.org/10.1145/1168918.1168864

Lucia, B., y Ceze, L. (2009). Finding concurrency bugs with context-aware communication graphs. En *42nd annual IEEE/ACM international symposium on microarchitecture, 2009. MICRO-42* (pp. 553–563).

Lucia, B., Devietti, J., Strauss, K., y Ceze, L. (2008). Atom-aid: Detecting and surviving atomicity violations. En *Proceedings of the 35th annual international symposium on computer architecture* (pp. 277–288). Washington, DC, USA: IEEE Computer Society. Descargado de http://dx.doi.org/10.1109/ISCA.2008.4 doi: 10.1109/ISCA.2008.4

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., ... Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. En *Proceedings of the 2005 acm sigplan conference on programming language design and implementation* (pp. 190–200). ACM.

Mathisen, T. (1994, julio). Pentium secrets: Undocumented features of the intel pentium can give you all the information you need to optimize pentium code. *j-BYTE*, *19*(7), 191–192.

Mattern, F. (1989). Virtual time and global states of distributed systems. En *Parallel and distributed algorithms* (p. 215–226). North-Holland.

Morajko, A. (2003). *Dynamic tuning of parallel/distributed applications*. Tesis Doctoral no publicada, Universitat Autònoma de Barcelona, Barcelona, España.

Muzahid, A., Otsuki, N., y Torrellas, J. (2010). AtomTracker: a comprehensive approach to atomic region inference and violation detection. En *Microarchitecture (MICRO), 2010 43rd annual IEEE/ACM international symposium on* (pp. 287–297). doi: 10.1109/MICRO.2010.32

Myers, G. J. (1984). *El arte de probar el software*. Buenos Aires: El Ateneo.

netcraft. (2015, enero). *January 2015 web server survey*. Descargado 2015-01-04, de http://news.netcraft.com/archives/2015/01/15/january-2015-web-server-survey.html

Park, C.-S., y Sen, K. (2008). Randomized active atomicity violation detection in concurrent programs. En *Proceedings of the 16th ACM SIGSOFT international symposium on foundations of software engineering* (p. 135–145). New York, NY, USA: ACM. Descargado 2014-01-23, de http://doi.acm.org/10.1145/1453101.1453121 doi: 10.1145/1453101.1453121

Park, S., Lu, S., y Zhou, Y. (2009, marzo). CTrigger: exposing atomicity violation bugs from their hiding places. *SIGPLAN Not.*, *44*(3), 25–36. (ACM ID: 1508249) doi: 10.1145/1508284.1508249

Pfleeger, S. L., y Atlee, J. M. (2009). *Software engineering: Theory and practice* (4.ª ed.). Prentice Hall.

Przybylski, S., Horowitz, M., y Hennessy, J. (1989). Characteristics of performance-optimal multi-level cache hierarchies. En *Proceedings of the 16th annual international symposium on computer architecture* (pp. 114–121). New York, NY, USA: ACM. Descargado 2015-02-06, de http://doi.acm.org/10.1145/74925.74939 doi: 10.1145/74925.74939

Qi, Y., Das, R., Luo, Z. D., y Trotter, M. (2009). MulticoreSDK: a practical and efficient data race detector for real-world applications. En *Proceedings of the 7th workshop on parallel and distributed systems: Testing, analysis, and debugging* (p. 5:1–5:11). New York, NY, USA: ACM. Descargado 2014-01-21, de http://doi.acm.org/10.1145/1639622.1639627 doi: 10.1145/1639622.1639627

Ronsse, M., y De Bosschere, K. (1999). RecPlay: a fully integrated practical record/-replay system. *ACM Trans. Comput. Syst.*, *17*(2), 133–152. doi: http://doi.acm.org/10.1145/312203.312214

Sasturkar, A., Agarwal, R., Wang, L., y Stoller, S. D. (2005). Automated type-based analysis of data races and atomicity. En *Proceedings of the tenth ACM SIGPLAN symposium on principles and practice of parallel programming* (p. 83–94). New York, NY, USA: ACM. Descargado 2014-01-23, de http://doi.acm.org/10.1145/1065944.1065956 doi: 10.1145/1065944.1065956

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., y Anderson, T. (1997, noviembre). Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput.*

Syst., *15*(4), 391–411. Descargado de http://doi.acm.org/10.1145/265924.265927 doi: http://doi.acm.org/10.1145/265924.265927

Serebryany, K., y Iskhodzhanov, T. (2009). ThreadSanitizer: data race detection in practice. En *Proceedings of the workshop on binary instrumentation and applications* (p. 62–71). New York, NY, USA: ACM. Descargado 2014-01-21, de http://doi.acm.org/10.1145/1791194.1791203 doi: 10.1145/1791194.1791203

Silberschatz, A., Galvin, P. B., y Gagne, G. (2009). *Operating system concepts, 8/E* (Eighth Edition ed.). John Wiley & Sons.

Sommerville, I. (2006). *Software engineering* (8.ª ed.). Addison Wesley.

Sorin, D. J., Hill, M. D., y Wood, D. A. (2011, noviembre). A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, *6*(3), 1–212. Descargado 2014-12-10, de http://www.morganclaypool.com/doi/abs/10.2200/S00346ED1V01Y201104CAC016 doi: 10.2200/S00346ED1V01Y201104CAC016

Sprunt, B. (2002). The basics of performance-monitoring hardware. *IEEE Micro*, *22*(4), 64–71.

Stallings, W. (2004). *Operating systems: Internals and design principles, 5/E* (Fifth Edition ed.). Prentice Hall.

Sussman, A. (2003, diciembre). *Bug 25520 - corrupt log lines at high volumes* [bugtracking]. Descargado 2015-01-04, de https://issues.apache.org/bugzilla/show_bug.cgi?id=25520

Tanenbaum, A. S., y Woodhull, A. S. (2006). *Operating systems design and implementation, 3/E* (Third Edition ed.). Prentice Hall.

Terboven, C., Bischof, C., Bücker, M., Gibbon, P., Joubert, G. R., Mohr, B., y Terboven, C. (s.f.). *Comparing intel thread checker and sun thread analyzer*.

Tutorial - perf wiki. (s.f.). https://perf.wiki.kernel.org/index.php/Tutorial. Descargado 2013-03-06, de https://perf.wiki.kernel.org/index.php/Tutorial

Wang, L., y Stoller, S. D. (2005). Static analysis of atomicity for programs with non-blocking synchronization. En *Proceedings of the tenth ACM SIGPLAN symposium on*

principles and practice of parallel programming (p. 61–71). New York, NY, USA: ACM. Descargado 2014-01-23, de http://doi.acm.org/10.1145/1065944.1065953 doi: 10.1145/1065944.1065953

Weaver, V. (2014, abril). *Manpage of PERF_event_open.* Descargado 2014-07-22, de http://web.eece.maine.edu/~vweaver/projects/perf_events/perf_event_open.html

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., y Gupta, A. (1995). The splash-2 programs: characterization and methodological considerations. *ACM SIGARCH Computer Architecture News*, *23*, 24–36.

Xu, M., Bodík, R., y Hill, M. D. (2005). A serializability violation detector for shared-memory server programs. En *Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation* (p. 1–14). New York, NY, USA: ACM. Descargado 2014-01-22, de http://doi.acm.org/10.1145/1065010.1065013 doi: 10.1145/1065010.1065013

Yeap, G. (2013, diciembre). Smart mobile SoCs driving the semiconductor industry: Technology trend, challenges and opportunities. En *Electron devices meeting (IEDM), 2013 IEEE international* (pp. 1.3.1–1.3.8). doi: 10.1109/IEDM.2013.6724540

Yu, J., y Narayanasamy, S. (2009). A case for an interleaving constrained shared-memory multi-processor. En *Proceedings of the 36th annual international symposium on computer architecture* (p. 325–336). New York, NY, USA: ACM. Descargado 2014-01-23, de http://doi.acm.org/10.1145/1555754.1555796 doi: 10.1145/1555754.1555796