Evaluación de técnicas de detección de errores en programas concurrentes

Fernando Emmanuel FRATI*

Director: Armando E. DE GIUSTI†

Codirector: Marcelo NAIOUF†

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

marzo 2014

*Universidad Nacional de Chilecito, Argentina
†Facultad de Informática, Universidad Nacional de La Plata, Argentina

Índice general

Índice de figuras

Índice de cuadros

Publicaciones

- 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*. Argentina

- 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*. Argentina

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

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

Capítulo 1

Introducción

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, las limitaciones a la integración impuestas por factores físicos como son la temperatura y el consumo de energía, se han traducido en la integración de unidades de cómputo en un único chip, dando lugar a los procesadores de múltiples núcleos. 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 accediendo a datos compartidos. El orden en que ocurren estos accesos a memoria puede variar entre ejecuciones, haciendo que los errores sean más difíciles de detectar y corregir.

En cómputo de altas prestaciones donde los tiempos de ejecución de las aplicaciones pueden variar de un par de horas hasta días, la presencia de un error no detectado en la etapa de desarrollo adquiere una importancia mayor. 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 detec-

tados. En este contexto, la eficiencia de los programas monitorizados se ve comprometida por el overhead que introduce el proceso de monitorización.

Este trabajo forma parte de las investigaciones para la tesis doctoral del autor en el tema *Software para arquitecturas basadas en procesadores de múltiples núcleos. Detección automática de errores de concurrencia*. Como tal, su aporte constituye un estudio de las técnicas y métodos vigentes en la comunidad científica aplicados a la detección y corrección de errores de programación en programas concurrentes.

Las siguientes secciones constituyen una introducción al proceso de detectar, localizar y corregir errores de software en programas secuenciales y se explican las complicaciones introducidas por los programas concurrentes. El Capítulo 2 trata los distintos errores que se pueden manifestar en programas concurrentes. El Capítulo 3 resume los antecedentes en técnicas de detección y corrección de errores de concurrencia y se justifica la elección de las violaciones de atomicidad como caso de error más general. El Capítulo 4 explica las características de un algoritmo de detección de violaciones de atomicidad, y da detalles de su implementación. El Capítulo 5 contiene las características de la plataforma de experimentación y de la metodología empleada. El Capítulo 6 proporciona los resultados del trabajo experimental. Finalmente, se presentan las conclusiones del trabajo y se proponen las líneas de investigación futuras.

1.1. 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 corregirse. 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, permitiendo 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.

1.2. 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. Esto se logra a través de la *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.

Capítulo 2

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.

Semáforos Los semáforos fueron inventados por E. W. Dijkstra hace más de 40 años (Dijkstra, 1968) y aún hoy son el método 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 manipularse 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 asumirá que los semáforos utilizados en los ejemplos son inicializados en 1.

2.1. 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.

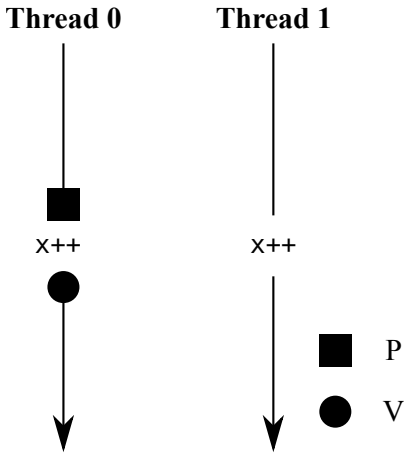

Figura 2.1: Ejemplo de una condición de carrera.

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

[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.

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.

2.2. 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.

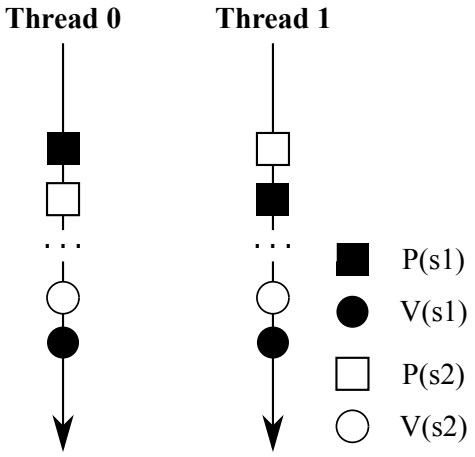

Figura 2.2: Ejemplo de deadlock.

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 sistemas operativos optan por ignorarlos y dejar en manos del desarrollador de la aplicación la responsabilidad de tratar con éstos.

2.3. 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.

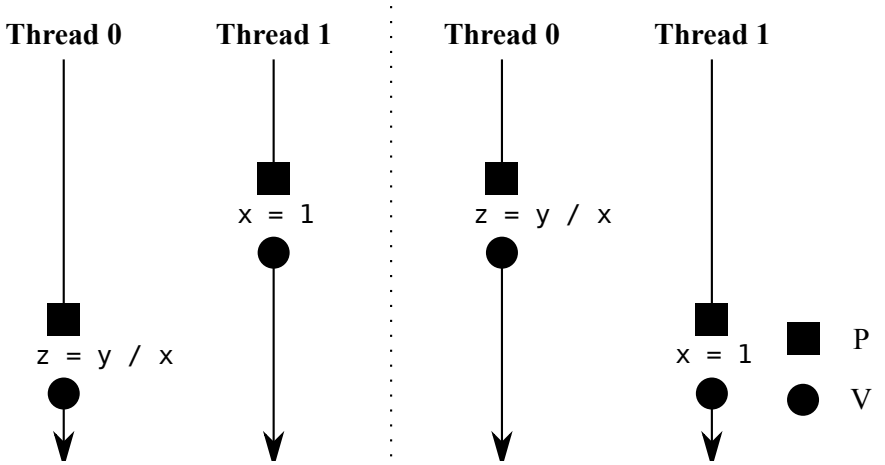

Figura 2.3: Ejemplo de una violación de orden.

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.

2.4. 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.

Como se puede apreciar, la variable compartida *tmp* se encuentra correctamente protegida en sus secciones críticas, eliminando la posibilidad de una condición de carrera.

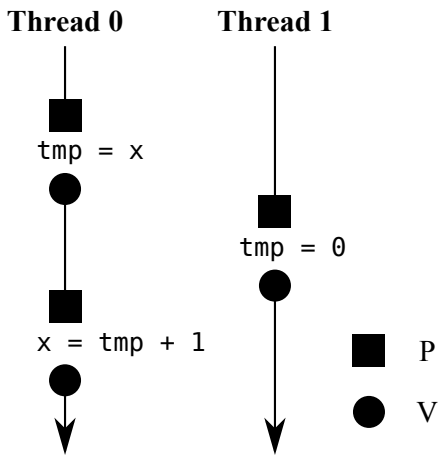

Figura 2.4: Ejemplo de una violación de atomicidad.

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. 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.

2.5. 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.

Supongamos que existe una variable *str* que contiene una cadena, y una variable *length* que contiene el número de caracteres de la cadena. *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, elimiando la posibilidad de una condición de carrera. Sin embargo, *Thread 1* modifica los valores de *str* y *length* antes de que *Thread 0* ingrese a la sección crítica de la variable *len*. Se debe notar que en esta situación, *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

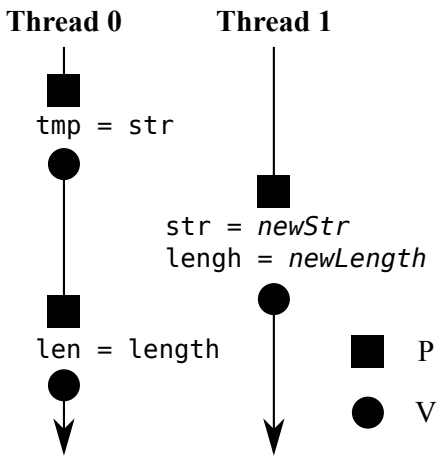

Figura 2.5: Ejemplo de una violación de atomicidad multivariable.

violación de atomicidad: es necesario considerar ambas variables como una unidad para que el error sea evidente.

2.6. ¿Qué tan frecuentes son estos errores?

Cualquiera de estos errores se puede encontrar con frecuencia en programas reales[2].

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, en 2008 un equipo de investigación publicó un trabajo(Lu, Park, Seo, y Zhou, 2008) en el que se estudian más de 100 errores de concurrencia conocidos en aplicaciones ámpliamente utilizadas (Apache, MySQL, Mozilla Firefox y OpenOffice).

Este trabajo reveló los siguientes datos interesantes:

- 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 los errores que no son deadlocks, 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 consituyen el caso más general de error de concurrencia.

[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.

- El 66 % de los errores que no son deadlocks 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.

Por los motivos antes expuestos, el resto del trabajo se enfocará en las técnicas de detección de violaciones de atomicidad, particularmente aquellas que ocurren sobre variables simples.

Capítulo 3

Violaciones de Atomicidad: 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.

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, Tucek, Qin, y Zhou, 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. Por este motivo proponen una serie de extensiones hardware para implementar el algoritmo.

MUVI(Lu y cols., 2008) analiza el caso de accesos correlacionados entre múltiples variables, y propone extensiones para detectores de carreras que usen los algoritmos lockset y happens-before. Es de los primeros trabajos dedicados integramente 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.

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 200x para aplicaciones intensivas como SPLASH-2(Woo, Ohara, Torrie, Singh, y Gupta, 1995) y PARSEC(Bienia, 2011). Por este motivo su propuesta deriva en añadir soporte al procesador para eficientemente validar estas restricciones y evitar que se ejecuten interleavings no probados.

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 multiples variables además de violaciones de atomicidad

y condiciones de carrera. 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(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 multi-variables.

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. Para ejecutar eficientemente el algoritmo, propone una implementación hardware ya que la versión software produce demasiado overhead.

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.

3.1. 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[1]).

Dada la escasez de herramientas disponibles para evaluar su desempeño, en este trabajo se decidió implementar una versión de AVIO. La ventaja de AVIO sobre las otras propuestas es que puede ser implementado completamente en software a un buen rendimiento. Además, como se verá en las conclusiones de este trabajo, el diseño de esta herramienta sienta las bases para trabajos futuros sobre optimización de herramientas de detección. El siguiente capítulo profundiza en el método de análisis de interleavings propuesto por AVIO y da detalles sobre su implementación.

[1]Para una explicación sobre estos algoritmos se puede consultar el Anexo A

Capítulo 4

Análisis de Interleavings

Este método y la herramienta que lo implementa llamada AVIO (Lu y cols., 2006) consiste en 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 4.1. Los accesos locales se encuentran alineados a la izquierda, mientras que el acceso remoto se encuentra desplazado a la derecha.

Cuadro 4.1: 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 4.1.

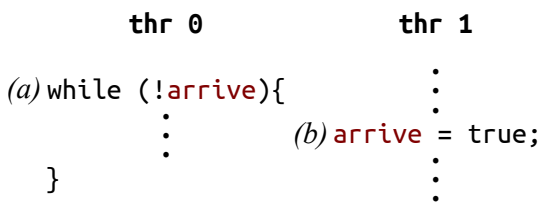

Figura 4.1: Ejemplo de un interleaving no serializable de caso 2 intencional. El interleaving se debe a una sincronización entre procesos.

Aunque se trata de un interleaving de caso 2, es de esperar que la escritura remota ocurra entre dos lecturas locales: 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.

4.1. Implementación de AVIO

Dado que no existe una implementación libre de AVIO, tuvimos que implementar nuestra propia versión siguiendo las especificaciones del artículo orginal. Básicamente el

algoritmo consiste en incluir una llamada a rutinas de análisis por cada lectura o escritura que realiza el programa objetivo.

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 4.2 muestra un modelo de las estructuras de datos que soportan el algortimo.

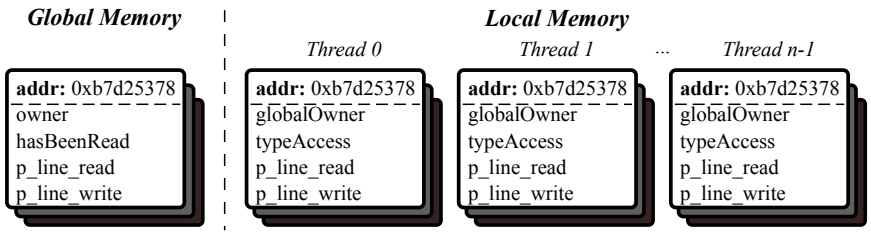

Figura 4.2: Estructuras de datos para soportar el algoritmo de AVIO.

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 4.1, 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.

Capítulo 5

Trabajo Experimental

La implementación de AVIO se desarrolló con el framework para instrumentación binaria dinámica PIN (Luk y cols., 2005) en la versión 2.11, de acuerdo a las especificaciones enunciadas en la sección 4.1.

Para la evaluación de las herramientas se toman en cuenta dos características: *capacidad de detección* y *rendimiento*. En cada caso se utiliza un conjunto de benchmarks diseñados para esos fines. La siguiente sección precisa detalles de cada benchmark. Cada aplicación se configuró con 2 threads y se garantiza que cada thread se ejecuta en un procesador diferente.

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 de el administrador de paquetes de Debian. Para nuestros experimentos, la versión más actual instalable desde los repositorios es la 3.8.1.

Para cada herramienta evaluada se calculó el promedio de 10 ejecuciones. De los resultados obtenidos, se verificó que la variación de los tiempos de ejecución sea lo suficientemente pequeña para asegurar que no hay efectos externos que artificialmente incrementan o decrementan el tiempo de ejecución.

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. 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).

5.1. 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 5.1 resume cada kernel y el caso de interleaving para el que fue diseñado.

Cuadro 5.1: 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 paper 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.

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 5.1 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

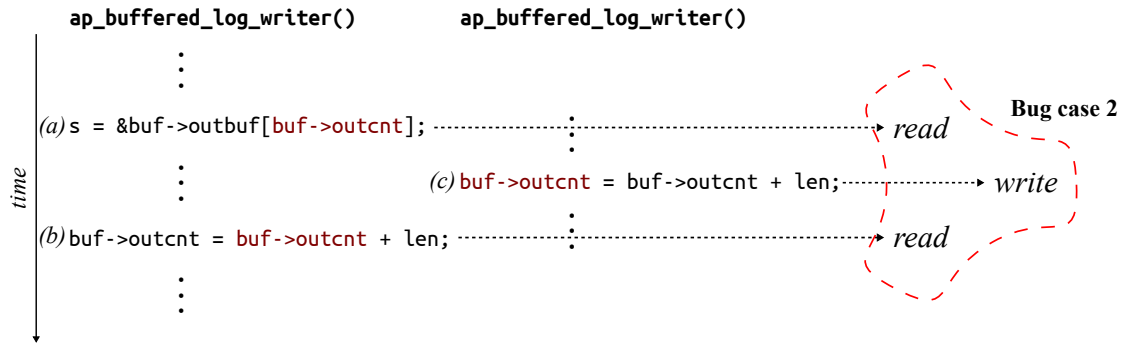

Figura 5.1: Ejemplo de una violación de atomicidad en Apache

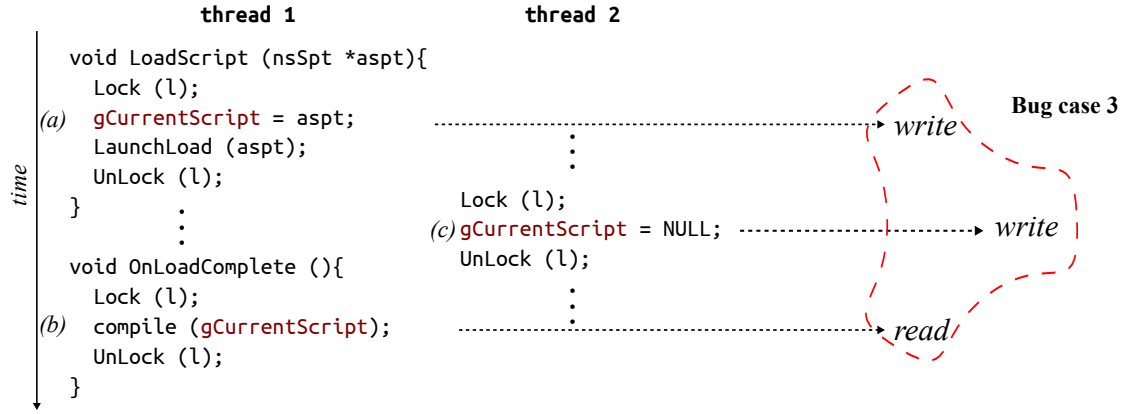

Figura 5.2: Ejemplo de una violación de atomicidad en Mozilla

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.

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 5.2 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.

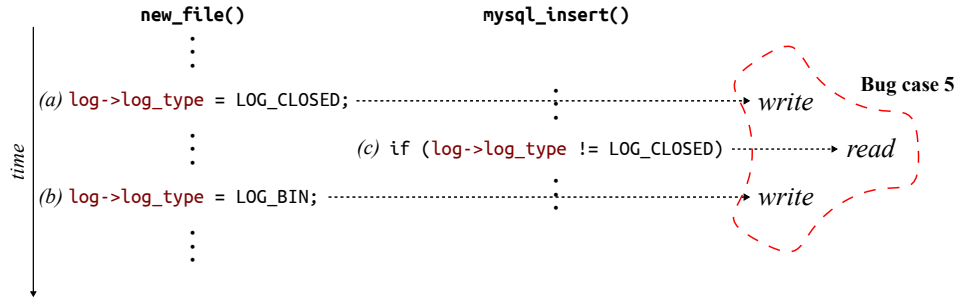

Figura 5.3: Ejemplo de una violación de atomicidad en MySQL

Cabe destacar que en este ejemplo el programador protegió cada acceso a *gCurrentScript* con el semáforo *l*. Esto evita que se produzca una condición de carrera sobre esta variable, pero como queda de manifiesto en el ejmplo, 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 5.3 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 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

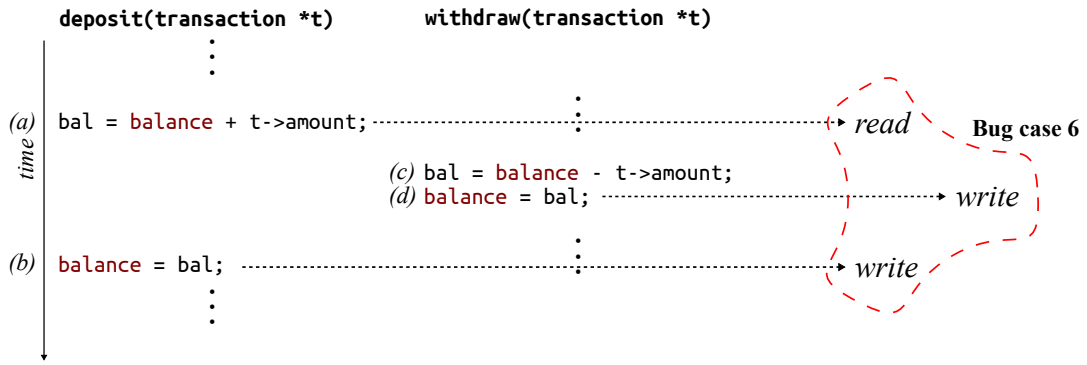

Figura 5.4: Ejemplo de una violación en una cuenta bancaria

y luego asigna el resultado nuevamente en *balance*. La Figura 5.4 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.

5.2. 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 5.2 resume los programas que componen la suite y las configuraciones empleadas en cada caso.

SPLASH-2 es probablemente la suite más utilizada para estudios científicos en máquinas paralelas con memoria comparida. 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.

Cuadro 5.2: 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

Capítulo 6

Resultados

Este Capítulo resume los resultados de evaluar ambas herramientas (AVIO y HELGRIND) a partir de su capacidad de detección de errores y rendimiento.

6.1. Capacidad de Detección de Errores

Para evaluar la capacidad de detección de violaciones de atomicidad, se ejecutaron los kernels de la Sección 5.1 a través de cada herramienta. La Tabla 6.1 resume los resultados indicando para cada kernel y cada herramienta si fue capaz de detectar el bug simulado.

Cuadro 6.1: 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 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 5.2 (descripción del kernel MOZILLA) donde se puede notar que los accesos a la variable compartida *gCurrentScript* se encuentran protegidos por locks. 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.

6.2. 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 6.2 resume los resultados obtenidos.

Cuadro 6.2: 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 estandar 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 6.3 muestra la relación de overhead para cada herramienta.

Cuadro 6.3: 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 *x*	1142 *x*
cholesky	26 *x*	335 *x*
fft	4 *x*	2 *x*
fmm	68 *x*	1108 *x*
Lu-cb	36 *x*	6 *x*
Lu-ncb	2 *x*	1 *x*
Ocean-cp	37 *x*	35 *x*
Ocean-ncp	38 *x*	41 *x*
radiosity	48 *x*	135 *x*
radix	6 *x*	2 *x*
raytrace	35 *x*	54 *x*
volrend	18 *x*	248 *x*
Water-nsq	28 *x*	4 *x*
Water-sp	24 *x*	4 *x*
Promedio 1	**32 *x***	**223 *x***
Promedio 2	**23 *x***	**224 *x***

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 pro-

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

ducto 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 x, mientras que HELGRIND alcanzó un valor de 223 x. 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 x para las aplicaciones de la suite SPLASH2. Sin embargo, en ese trabajo sólo se publicaron los resultados para las aplicaciones fft, fmm, lu y radix. La Tabla 6.3 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 x para esas aplicaciones, lo cual coincide con los resultados reportados en el artículo original.

Capítulo 7

Conclusiones y Líneas de Trabajo Futuras

En programas secuenciales basta con comparar dos ejecuciones con el mismo caso de prueba para determinar si un error ha sido corregido. Sin embargo, esta suposición no es válida para programas concurrentes: el no determinismo implícito en la ejecución de estos programas provoca que sucesivas ejecuciones del mismo programa con el mismo caso de prueba siempre produzcan historias diferentes. Además la mayoría de los errores de concurrencia (a excepción de los deadlocks) pueden ocurrir sin que el usuario lo note hasta que sea tarde. Por esto es necesario el uso de herramientas de detección de errores de concurrencia que ayuden al programador en la tarea de probar y depurar sus programas.

Desafortunadamente los algoritmos de detección de errores introducen un elevado overhead (de uno a varios ordenes de magnitud) en las ejecuciones de los programas. Si consideramos que una parte importante de los recursos de desarrollo se invierten en la prueba del software, es evidente que una prueba rigurosa vuelve al proceso inaceptablemente costoso: inevitablemente una gran cantidad de errores no serán detectados y terminarán en el software liberado para producción.

Una manera de resolver este problema consiste en evitar que los errores ocultos en el código se manifiesten en tiempo de ejecución. Esto se puede conseguir evitando que ocurra la historia que lo produce. Para ello muchas propuestas *restringen* el no-determinismo implicito en las ejecuciones concurrentes forzando algún orden conocido en el que no se manifestaron estos errores durante la prueba del sistema. Se debe notar que aunque se *aleje* la posibilidad de que los errores se manifiesten, no se elimina la posibilidad de que ocurran: después de todo las pruebas no son perfectas y no hay garantías de que las

ejecuciones marcadas como válidas no posean errores[1]. Además, si una historia válida no fue desencadenada durante la etapa de pruebas, estas técnicas podrían evitar que ocurra.

Otro enfoque considera que la única manera de saber si una ejecución fue correcta, es controlando su ejecución. Sin embargo, nuevamente el overhead que introducen los algoritmos de detección constituye un problema fundamental: es demasiado costoso controlar todas las ejecuciones del software en producción.

Un problema común que caracteriza a todos los enfoques es que el objetivo es la eficiencia: no es suficiente con detectar los errores, sino que es necesario hacerlo en el menor tiempo posible. Sin embargo, la mayoría de los trabajos que tienen en cuenta este problema apuntan a desarrollar hardware a medida para dar soporte a sus algoritmos de detección: evidentemente esto restringe su utilidad sólo a entornos experimentales, dejando a los entornos de producción desamparados hasta que algún fabricante incorpore estas propuestas en sus diseños de hardware. Mientras esto no ocurra, es necesario el diseño de herramientas implementables completamente en software, que eviten la necesidad de hardware especializado.

En el Capítulo 2 se caracterizaron los tipos de errores y se justificó la elección de las violaciones de atomicidad como caso general y representativo para el análisis.

Para determinar el overhead de las herramientas, se buscaron en el mercado productos de software que cumplan con esta tarea. Sorprendentemente, aunque existe una gran cantidad de trabajos que abordan el problema de detección de errores de concurrencia, sólo se encontraron implementaciones de las técnicas más robustas para detección de condiciones de carrera. Por este motivo se implementó una versión propia de AVIO, un algoritmo completamente implementable en software para detección de violaciones de atomicidad. Los Capítulos 3 y 4 justifican la elección de este algoritmo y proporcionan detalles sobre su implementación.

En el Capítulo 6 se determinó el overhead que introducen estas herramientas en la ejecución de las aplicaciones. Además, se corroboró que la implementación de AVIO es coherente con los resultados de overhead argumentados en el trabajo original. Aunque AVIO mostró un comportamiento promedio más estable que HELGRIND y con mejor capacidad de detección, los resultados evidencian la necesidad de reducir el overhead de estas herramientas.

[1]En el Capítulo 2 se mostró como un programa considerado libre de condiciones de carrera puede tener otra clase de errores, como violaciones de orden o de atomicidad.

En este sentido, se está trabajando en el desarrollo de un modelo de optimización para herramientas de detección de errores de concurrencia, en particular violaciones de atomicidad sobre variables simples. La propuesta consiste en utilizar la información disponible sobre la ejecución de los programas a través de los contadores de hardware del procesador para conseguir una importante reducción de overhead.

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.

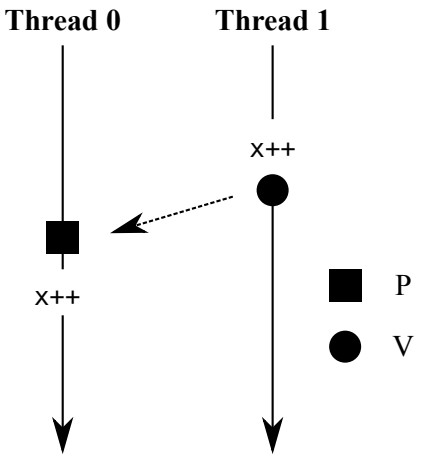

Figura A.1: Noción de orden parcial entre segmentos de procesos.

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.

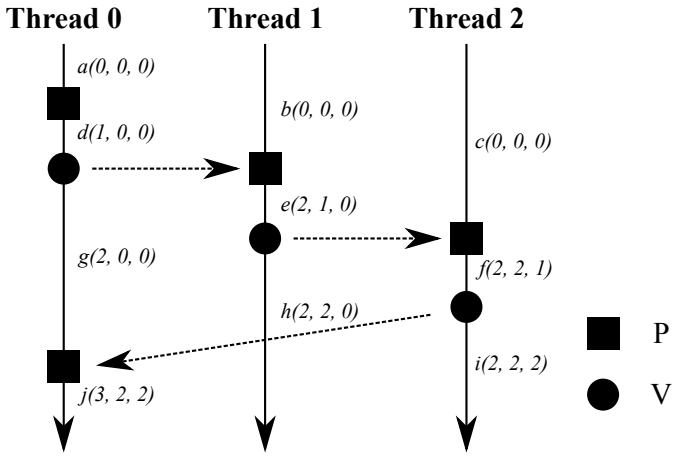

Figura A.2: Detección de segmentos potencialmente concurrentes a través del uso de vectores de relojes lógicos.

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 descripto 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.

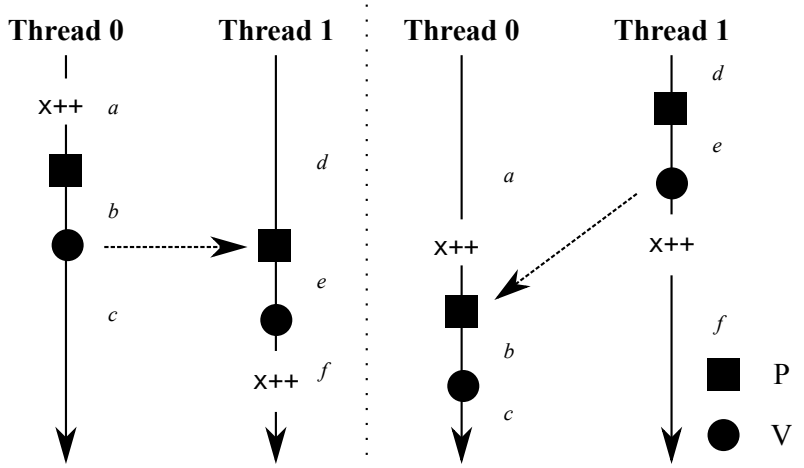

Figura A.3: En el ejemplo se muestran dos casos de ejecución, donde la condición de carrera sólo se presenta en el caso de la derecha.

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 *C(x)* y en él se reflejan las actualizaciones correspondientes ante cada acceso a la variable

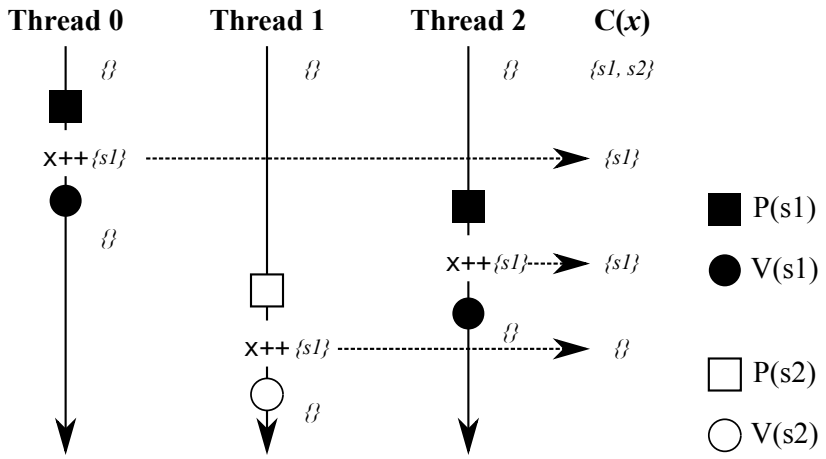

Figura A.4: Detección de condiciones de carrera con la estrategia lockset.

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.

Referencias

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.

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

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

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

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

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

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

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.

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

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.

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.

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

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.

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

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*.

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

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

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