Proyecto Fin de Máster
en Ingeniería de Computadores.
Curso 2009-2010.

Mecanismo de detección de violaciones de atomicidad de reducido coste hardware

Autor:
Jorge Quintás Rodríguez

Director del proyecto:
Luis Piñuel Moreno

Máster en Investigación en informática,
Facultad de Informática,
Universidad Complutense de Madrid.

Resumen

Con la difusión de los procesadores multicore los fallos de concurrencia se han convertido en un problema desafiante. De entre ellos son especialmente problemáticas las violaciones de atomicidad, que suponen un 66 % de los fallos de concurrencia que no son interbloqueos. En este trabajo presentamos un algoritmo que es capaz de detectar violaciones de atomicidad que afectan a una variable compartida. Para llevar a cabo la detección empleamos el mínimo soporte hardware, utilizando la información que nos ofrece el protocolo de coherencia cache.

Abstract

With multicore processors becoming widespread, concurrency bugs have become a challenging problem. Betweem them, atomicity violations are specially problematic and they account for the 66 % of all the concurrency bugs, excluding deadlocks. In this work we introduce an algorithm wich can detect atomicity violations concerning one shared variable. To achieve the detection we use the minimal hardware support, using information collected from the cache coherence protocol.

Palabras clave

Errores de concurrencia, coherencia cache, depuración, detección de errores, violaciones de atomicidad, carreras de datos, arquitecturas multicore, soporte *hardware*

Keywords

Concurrency bugs, cache coherence, debugging, bug detection, atomicity violations, data races, multicore arquitectures, hardware support

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Mecanismo de detección de violaciones de atomicidad de reducido coste *hardware*", realizado durante el curso académico 2009-2010 bajo la dirección de Luis Piñuel Moreno en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Madrid, a 13 de Septiembre de 2010.

Jorge Quintás Rodríguez

Agradecimientos

A mi familia y amigos. A becerrada(isto témolo que celebrar como sabemos), al restaurante O Pote. Al grupo ArTeCS y a sus técnicos Roberto, Luis, Marco y especialmente a Tabas. A los que me habéis acompañado hasta aquí.

Índice general

Capítulo 1

Introducción

En este capítulo inicial primero se define el problema al que nos queremos enfrentar. Después se exponen la motivación y los objetivos concretos del proyecto, y se finaliza con una descripción de la organización de esta memoria.

1.1. *Bugs* de concurrencia

Escribir programas concurrentes es una tarea difícil y propensa a errores. Los programadores están habituados pensar de una forma secuencial, y resulta costoso tener en consideración todas las posibles interacciones entre distintas tareas que se ejecutan a la vez. Esto hace que escribir un programa concurrente erróneo sea sencillo. Sin embargo, la detección de estos errores está lejos de ser fácil. El elevado indeterminismo de la ejecución de los programas concurrentes hace que la manifestación del bug sea impredecible. Esto tiene dos consecuencias directas: el *testeo* de estos programas se complica, ya que son necesarias varias ejecuciones para cubrir todos los posibles entrelazados, y el

diagnóstico *postmortem* es casi imposible ya que sería necesario repetir el entrelazado causante. Para poder hacer esto sería necesario poder fijar el orden de los accesos de memoria, para lo cual existen propuestas, pero el espacio de búsqueda sería demasiado grande como para comprobar todos los posibles casos. Existen tres tipos de bug de concurrencia:

***deadlocks* o interbloqueos:** Se produce un interbloqueo cuando en un grupo de dos o más procesos cada uno está esperando que el resto libere un recurso, o cuando más de dos operaciones esperan circularmente a la liberación de un recurso.

***Data Race* o carrera de datos:** Se produce cuando el acceso a una variable compartida no se efectúa con la sincronización adecuada, de este modo varios hilos pueden acceder a la variable compartida al mismo tiempo.

Violación de atomicidad: Existe una violación de atomicidad en el código cuando un programador asume, incorrectamente, que un fragmento del código se ejecutará de forma atómica, y no encierra los accesos que deberían ejecutarse atómicamente en la misma sección crítica. De esta forma, aunque cada uno de los accesos esté debidamente sincronizado, y por tanto, libre de carreras de datos, se pueden producir accesos de otros *threads* que violan la atomicidad de los accesos supuesta por el programador. Las violaciones de atomicidad también existen en programas que emplean sincronización basada en memoria transaccional en lugar de cerrojos, al contrario de lo que sucede con las carreras de datos.

Además, las arquitecturas de un solo *core* se evolucionaron hasta que la disipación energética y el consumo de potencia fueron tan elevados que el escalado de frecuencia dejó de ser viable. Este problema, conocido como *power wall*, tuvo como respuesta de los fabricantes las arquitecturas *multicore*. En éstas, se incorporan varios *cores* más sencillos en un solo *chip*. De esta forma se logra contener el consumo, pero contrariamente a lo que sucedía en la anterior generación, en la que duplicar la frecuencia de un microprocesador suponía duplicar el rendimiento, duplicar el número de núcleos en un *chip* no supone la duplicación instantánea del rendimiento por norma general. Es necesario que las aplicaciones sean capaces de explotar el paralelismo que ofrece el *hardware*, de forma que cada núcleo tenga siempre tareas listas para ejecutar. Con el escalado tecnológico los fabricantes pueden incorporar cada vez más núcleos, con lo que las aplicaciones deben exponer mayor paralelismo, lo cual hace que los *bugs* de concurrencia sean cada vez más frecuentes en el código, y no solo eso, al aumentar el número de procesadores y por tanto de tareas que se ejecutan de forma simultánea, aumentan los entrelazados de acceso a memoria, y con ello aumentan las posibilidades de que manifieste uno de estos *bugs*.

1.2. Motivación

En la actualidad los microprocesadores con varios núcleos son una realidad ubicua en el mundo de la computación, y la tendencia es introducir cada vez más núcleos en un *chip*. Estas medidas exigen a los programadores el esfuerzo de paralelizar su código en virtud de un mejor aprovechamiento de los recursos *hardware* para obtener un mayor rendimiento, una tarea complicada y,

como hemos comentado, propensa a errores. Dada la dependencia actual de las aplicaciones informáticas en todos los campos, es preciso que tales aplicaciones sean no solo eficientes, sino robustas y confiables. Para ello deben estar libres de *bugs*, por lo que es necesario aportar herramientas a los programadores para que sean capaces de comprobar sus programas y de detectar y arreglar estos fallos. La mayor parte de los estudios previos se centran en detectar carreras de datos, y recientemente se está poniendo interés en las violaciones de atomicidad, ya que, sin contar con los interbloqueos, las violaciones de atomicidad suponen dos tercios de los bugs de concurrencia. Además, con los sistemas de memoria transaccional las carreras de datos dejan de ser un problema ya que el sistema subyacente se encarga de detectar y resolver los conflictos en los accesos a memoria. Sin embargo, las violaciones de atomicidad pueden seguir ocurriendo en sistemas transaccionales si los programadores no agrupan los accesos que deberían ser atómicos en la misma transacción.

1.3. Objetivos

Este trabajo tiene como objetivo el desarrollo de una herramienta que permita detectar las violaciones de atomicidad empleando la información proporcionada por el protocolo de coherencia cache y el mínimo soporte *hardware*. Para esto nos apoyaremos en el protocolo de coherencia cache basado en invalidación presente en los procesadores de Intel, MESI. Se pretende implementar un sistema que sea capaz de detectar las violaciones de atomicidad que involucran a una variable para ayudar a la detección y depuración de este tipo de errores en aplicaciones concurrentes.

1.4. Organización del resto del documento

A continuación se presenta la organización del resto de la presente memoria de proyecto.

- Este Capítulo presenta una breve introducción, junto con la motivación y los objetivos de este trabajo.

- El Capítulo 2 contiene un análisis del marco que ha propiciado que los *bugs* de concurrencia sean un problema a tener en cuenta, se caracterizan las violaciones de atomicidad, y finalmente, se presentan ciertos conceptos del protocolo de coherencia cache que serán empleados para llevar a cabo la detección.

- En el Capítulo 3 se aborda el objetivo del proyecto, que consiste en la implementación, con el mínimo apoyo *hardware*, de un algoritmo que sea capaz de detectar las violaciones de atomicidad que se producen durante la ejecución de un programa.

- En el Capítulo 4 se presentan los resultados obtenidos al someter a diversos programas de ejemplo y aplicaciones reales al algoritmo de detección desarrollado.

- En el Capítulo 5 se exponen las principales conclusiones y aportaciones de este trabajo, así como el trabajo futuro.

- Finalmente, en el Capítulo 6 se recapitulan los principales trabajos relacionados, centrándonos en los dos tipos de *bugs* de concurrencia en

los se centran las investigaciones actuales, las carreras de datos y las violaciones de atomicidad.

Capítulo 2

Violaciones de atomicidad

Durante años la tendencia fue mejorar los procesadores, mediante técnicas que permitían explotar el paralelismo a nivel de instrucción(ILP), algunas de ellas muy agresivas y muy ineficientes desde el punto de vista energético, y a base de aumentar la frecuencia de reloj a la que operaba el microprocesador. El consumo energético causado por estas técnicas acabó en lo que se conoce como *power wall*, no era viable aumentar el rendimiento de un microprocesador con estas técnicas debido al consumo de potencia, lo cual acarrea otros problemas como la disipación del calor producido.

A raíz de este problema los fabricantes optaron por aumentar el número de *cores*, pasando de procesadores monolíticos a procesadores *multicore* o chips multiprocesador (CMP). Debido a que los CMPs explotan el paralelismo a nivel de thread (TLP) explícito, sus *cores* pueden ser más sencillos y no necesitan *hardware* adicional para extraer ILP, dicho de otra manera, los CMPs permiten explotar el paralelismo de una manera eficiente desde el punto de vista energético.

Sin embargo, esta solución plantea nuevos problemas. En la anterior generación de procesadores el rendimiento escalaba linealmente con la frecuencia a la que operaba el procesador. De este modo, si disponemos de dos chips que implementan la misma microarquitectura, pero uno de ellos es capaz de operar al doble de frecuencia del otro, un programa se ejecutará en la mitad de tiempo en él, y esto ocurría de forma totalmente transparente al desarrollador de *software*. Con los CMPs esto no es así, aumentar el número de cores no supone, en general, un incremento inmediato del rendimiento para un determinado programa, por lo que los desarrolladores deben escribir aplicaciones paralelas que sean capaces de aprovechar el *hardware* extra que los fabricantes ponen a su disposición. Por si fuera poco, esta tarea es compleja y propensa a una serie de errores que son difíciles de detectar y depurar, ya que dada la naturaleza aleatoria de los accesos a memoria en estos procesadores, la manifestación de estos fallos es imprevisible.

Como hemos comentado previamente existen tres tipos de errores de concurrencia, *deadlocks*, carreras de datos, y violaciones de atomicidad, siendo este último el más común a parte de los *deadlocks*.

El ejemplo de la figura 2.1 muestra una violación de atomicidad real extraída de la Mozilla Application Suite. Como se observa, los accesos a la variable compartida están correctamente protegidos por cerrojos y tienen relaciones *happened before* estrictas entre ellos. Sin embargo el fragmento sigue siendo incorrecto, `gCurrentScript` es un manejador compartido del *script* que está siendo procesado, el *thread* 1 asigna el *script* y comienza su carga como un evento asíncrono, antes de que la carga finalice, el *thread* 1 continuará el procesado del *script* en base al manejador previo, Mientras tanto, el *thread* 2

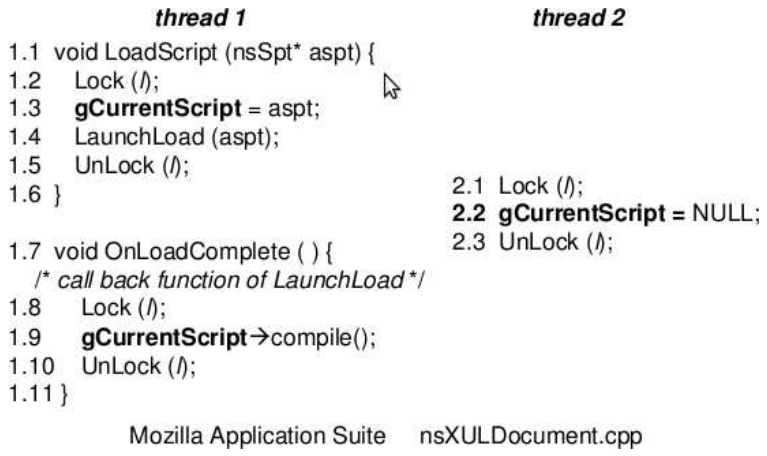

FIGURA 2.1: Ejemplo de violación de atomicidad.

puede anular ese manejador, provocando un error en la aplicación.

La atomicidad es la propiedad por la cual el efecto de la manipulación de datos llevada a cabo por varias acciones que se ejecutan concurrentemente es equivalente a la que se obtendría con una ejecución secuencial de las mismas (serializabilidad). Para el ejemplo de la figura 2.1, las dos partes del procestado del *script* del *thread* 1 deberían ser atómicas. Lo que en muchas ocasiones esperan los programadores es la atomicidad del código, pero al escribir asumen erróneamente que las operaciones cercanas en un *thread* se ejecutarán atómicamente. Si esta asunción no es satisfecha por la implementación, algunas ejecuciones manifestarán entrelazados de acceso a memoria no serializables, lo cual viola las suposiciones del programador y puede manifestarse como un *bug* de concurrencia. Emplear cerrojos es una forma de asegurar la atomicidad si están correctamente ubicados, pero, como hemos visto, la ausencia de carreras de datos no garantiza la atomicidad. Además, propuestas actuales como la memoria transaccional son capaces de evitar las carreras de datos, ya que el

sistema subyacente detecta los accesos en conflicto de distintas transacciones y retrocederá para resolver los problemas encontrados, pero las violaciones de atomicidad seguirán ocurriendo si los programadores no agrupan las operaciones que deberían ser atómicas en la misma transacción.

2.1. Análisis de serializabilidad

Cuando varias tareas se ejecutan concurrentemente, se dice que son serializables si el resultado de las operaciones que llevan a cabo es el mismo que el que se obtendría si se ejecutaran de forma secuencial sin entrelazados.

Para poder detectar las violaciones de atomicidad es preciso caracterizar que entrelazados de accesos a memoria son serializables y cuales no. Existen ocho formas en las que dos accesos locales consecutivos a la misma variable compartida pueden ser entrelazados por un acceso remoto.

La tabla 2.1 resume los ocho entrelazados posibles, explicando para cada caso por qué es serializable, con el acceso secuencial equivalente, o no serializable. De los ocho casos, cuatro son entrelazados serializables (1, 2, 5, 8) y los otros cuatro no lo son (2, 3, 5, 6). Los entrelazados serializables no causan violaciones de atomicidad, mientras que los entrelazados no serializables pueden causarlas si violan las suposiciones del programador. Estas condiciones de no serializabilidad para un solo acceso remoto entrelazado se puede extender para obtener cuatro casos similares de no serializabilidad teniendo en cuenta múltiples accesos remotos a la misma variable compartida:

TABLA 2.1: Ocho casos de entrelazados de acceso

Entrelazado	Descripción	Serializabilidad	Acceso secuencial equivalente	Problemas
$read^p$ $read_r$ $read^i$	Dos lecturas entrelazadas por una lectura	serializable	$read^p$ $read^i$ $read_r$	N/A
$write^p$ $read_r$ $read^i$	Lectura tras escritura entrelazadas por una lectura	serializable	$write^p$ $read^i$ $read_r$	N/A
$read^p$ $write_r$ $read^i$	Dos lecturas entrelazadas por una escritura	no serializable	N/A	La escritura entrelazada hace que las dos lecturas tengan vistas diferentes de la misma posición de memoria
$write^p$ $write_r$ $read^i$	Lectura tras escritura entrelazada por una escritura	no serializable	N/A	La lectura local no toma el resultado local esperado
$read^p$ $read_r$ $write^i$	Escritura tras lectura entrelazada por una lectura	serializable	$read_r$ $read^p$ $write^i$	N/A
$write^p$ $read_r$ $write^i$	Dos escrituras entrelazadas por una lectura	no serializable	N/A	El resultado intermedio, que debería ser invisible para otros *threads* es leído por un acceso remoto
$read^p$ $write_r$ $write^i$	Lectura después de escritura entrelazada por una escritura	no serializable	N/A	La escritura local depende del valor de la lectura local precedente, pero éste es sobreescrito por la escritura remota
$write^p$ $write_r$ $write^i$	Dos escrituras entrelazadas por una escritura	serializable	$write_r$ $write^p$ $write^i$	N/A

- Caso 2: $r[*w_r*]r$, dos lecturas locales son entrelazadas por al menos una escritura remota, de modo que tienen vistas distintas.
- Caso 3: $w[*w_r*]r$, una lectura tras escritura local es entrelazada por al menos una escritura remota. Debido a esta escritura remota, la lectura local puede no obtener el resultado local esperado.
- Caso 5: $w[r_r*]w$, una escritura tras escritura local es entrelazada por una secuencia de acceso remoto que comienza por una lectura, haciendo visible a otros *threads* el resultado intermedio local.
- Caso 6: $r[*w_r*]w$ una escritura tras lectura local es entrelazada por al menos una escritura remota. Esto hace que el valor previamente leído quede obsoleto.

dónde * indica cero o varios accesos de lectura o escritura remotos que se entrelazan con los accesos locales, y los accesos remotos están encerrados en corchetes.

2.2. Coherencia cache

La jerarquía de memoria de los CMPs suele constar de caches privadas para cada *core* junto con caches compartidas por todos ellos. En un esquema de este estilo pueden producirse problemas de inconsistencia de datos. Si un procesador lee un dato de una determinada dirección y otro procesador escribe en esa misma posición posteriormente, el primer procesador tendrá un dato inválido en su cache privada. Para evitar este tipo de problemas se emplean los protocolos de coherencia cache. La coherencia define el comportamiento de

las lecturas y escrituras a la misma posición de memoria. La coherencia de las caches se obtiene si se reúnen las siguientes condiciones:

1. Una lectura efectuada por un procesador P a una dirección D que sigue a una escritura realizada por el mismo procesador P a la dirección D, sin que ningún otro procesador escriba en D entre la escritura y la lectura hechas por P, siempre devolverá el valor escrito por P. Esta condición está relacionada con la preservación del orden del programa.

2. Una lectura hecha por el procesador P1 a la dirección D que es seguida por una escritura del procesador P2 a D debe devolver el valor escrito por P2 si no se realizan otras escrituras a D por parte de otros procesadores entre los dos accesos. Esta condición define el concepto de visión coherente de memoria. Si los procesadores pueden leer el antiguo valor previo a la escritura efectuada por P2, el estado de la memoria es incoherente

3. Escrituras a la misma posición deben efectuarse en secuencia. Si la dirección D recibe dos valores A y B distintos, en este orden, de dos procesadores cualesquiera, los procesadores no pueden leer la dirección D como B y luego como A. La dirección D debe ser actualizada con los valores A y B en ese orden.

2.2.1. Mecanismos de coherencia cache

Existen diversas técnicas sobre las que se construyen los protocolos de coherencia cache:

coherencia basada en directorio: en un sistema basado en directorio, los datos compartidos se almacenan en un directorio compartido que mantiene la coherencia entre las caches. El directorio actúa como un filtro a través del cual el procesador debe obtener permiso para cargar una entrada de la memoria principal a su cache. Cuando una entrada es modificada, el directorio actualiza o invalida el resto de caches que contienen esa entrada.

snooping: es el proceso en el que las caches monitorizan las lineas de dirección en busca de accesos a direcciones que tienen cacheadas. Cuando se observa una operación de escritura a una dirección de la que la cache tiene copia, el controlador de cache invalida su propia copia de esa dirección de memoria.

snarfing: el controlador de cache monitoriza tanto las direcciones como los datos. Cuando se observa una operación de escritura a una dirección de la que la cache tiene copia, el controlador de cache actualiza su propia copia de esa posición de memoria con los nuevos datos.

2.2.2. Protocolo de coherencia cache

El protocolo de coherencia cache es el encargado de mantener la consistencia entre todas las caches en un sistema de memoria compartida distribuida. Existen varios modelos y protocolos para mantener la coherencia cache, como:

- MSI
- MESI o Illinois

- MOSI
- MOESI
- MERSI
- MESIF
- Firefly
- Dragon
- Write-once
- Synapse
- Berkeley

A continuación profundizaremos en el protocolo MESI, ya que es el protocolo de coherencia cache sobre el que hemos implementado el mecanismo de detección de violaciones de atomicidad.

2.2.3. Protocolo MESI

El protocolo MESI fue desarrollado en la Universidad de Illinois en Urbana-Champaign. Es el protocolo más común que soporta caches *write-back*. Su uso en computadores personales es mayoritario debido a su introducción en los procesadores Pentium de Intel cuando, por motivos de eficiencia, cambiaron la política de escritura de cache a *write-back*, en detrimento de la política *write-through* empleada previamente en el procesador 486.

En el protocolo de coherencia cache MESI cada linea de cache está marcada con uno de los cuatro estados siguientes (codificados con dos bits):

Modified La línea de cache está presente solo en la cache local, y está *sucia*, ha sido modificada con respecto al valor existente en memoria principal. Es preciso que la cache escriba los datos de nuevo a la memoria principal en algún momento, antes de permitir cualquier lectura del estado de la memoria principal, ya que no es válido. La escritura cambia la línea al estado *Exclusive*.

Exclusive La línea de cache está presente sólo en la cache actual, pero está *limpia*, su valor coincide con el valor almacenado en memoria principal. Puede cambiar al estado *Shared* en respuesta a una petición de lectura. Alternativamente, puede cambiar al estado *Modified* si se escribe la línea.

Shared Indica que esta línea de cache puede estar almacenada en otras caches y que está *limpia*. La línea puede ser invalidada (el estado cambia a *Invalid*) en cualquier momento.

Invalid Indica que la línea de cache es inválida

En un sistema típico, varias caches comparten un bus común a la memoria principal. Cada una de ellas estará conectada a una CPU que lanza peticiones de lectura y escritura. La labor colectiva de las caches es minimizar el acceso a la memoria principal compartida.

Una cache puede satisfacer una lectura desde cualquier estado no inválido. Cuando los datos son recibidos, la línea de memoria pasará al estado *Exclusive* si ninguna otra cache tiene copia o al estado *Shared* en caso contrario.

Para realizar una escritura, la línea de cache debe estar en estado *Modified* o *Exclusive*, si está en estado *Shared* las copias del resto de caches deben ser invalidadas, para lo que se enviará un mensaje de invalidación a todas las caches(operación *broadcast* conocida como *Read For Ownership*) y se esperará la respuesta de las mismas. Si el estado de partida antes de una escritura es *Exclusive*, tras realizar la operación de escritura la línea pasará al estado *Modified*.

Una cache puede descartar una línea no modificada en cualquier momento, cambiando su estado a *Invalid*. Una línea en estado *Modified* debe ser debidamente actualizada en los niveles inferiores de la jerarquía de memoria antes de proceder a su invalidación. Una cache que contenga una línea en el estado *Shared* debe escuchar las peticiones de invalidación y los *broadcast Read For Ownership* de otras caches y descartar la línea (pasándola al estado *Invalid*) si hay coincidencia. Si la línea está en estado *Exclusive*, debe escuchar todas las peticiones de lectura del resto de caches, y cambiar el estado de la línea a *Shared* si hay coincidencia.

Capítulo 3

Implementación

En este capítulo se aborda la implementación del algoritmo de detección de violaciones de atomicidad. Primero se realiza un análisis de la información que se puede extraer del protocolo de coherencia cache, y posteriormente se dan los detalles de la implementación.

3.1. Análisis previo

Como vimos en el capítulo 2 el protocolo de coherencia cache añade información a cada línea de cache para realizar su labor. Éstos *tags* pueden emplearse para obtener información acerca de los entrelazados que se han producido en los accesos a memoria, y como veremos, con estos datos podemos detectar si durante la ejecución de un programa se han producido violaciones de atomicidad. La información que podemos extraer del estado de un bloque de cache es la siguiente:

- Si un bloque está en estado *Modified* en una cache, podemos concluir

que esa cache fue la última en escribir el bloque.

- Si un bloque está en estado *Exclusive* en una cache, es seguro que esa cache tiene el bloque de forma exclusiva.
- Si un bloque está en estado *Shared*, en una cache, esa cache ha leído el bloque con anterioridad.
- Si un bloque está en estado *Invalid*, en una cache, una cache remota ha invalidado y modificado el bloque.

Con esta información, y conociendo el tipo de acceso que se está llevando a cabo, podemos realizar la detección de violaciones de atomicidad, tal y como se ve en la tabla 3.1

A continuación comentamos los cuatro casos positivos:

RWR Los accesos locales son ambos de escritura. Existe una petición de *downgrade* entre los accesos locales y el estado de la línea antes del último acceso local es *Invalid*, un *thread* remoto ha escrito el bloque entre los accesos locales.

WWR EL primer acceso local es una escritura, y el segundo una lectura. Existe una petición de *downgrade* entre los accesos locales y el estado de la línea de cache antes del acceso local de lectura es *Invalid*, un *thread* remoto ha escrito la línea entre los accesos locales.

RWW Tenemos dos accesos locales, el primero de lectura y el segundo des escritura. Entre los dos accesos locales existe una petición de *downgrade*

TABLA 3.1: Información necesaria para la detección

Último acceso local	*Downgraded*	*evicted*	Estado previo al acceso	Acceso actual	Entrelazado	Resultado
R	N	N	M	R	Ninguno	Ok
R	N	N	E	R	Ninguno	Ok
R	N	N	S	R	Ninguno	Ok
R	N	S	I	R	R*R o ninguno	no concluyente
R	S	N	I	R	RWR	Violación
R	S	N	S	R	RRR	Ok
R	S	N	I	R	RWR	Violación
W	N	N	M	R	Ninguno	Ok
W	N	S	I	R	W*R o ninguno	Ok
W	S	N	I	R	WRR	Ok
W	S	N	I	R	WWR	Violación
R	N	N	M	W	Ninguno	Ok
R	N	N	E	W	Ninguno	Ok
R	N	N	S	W	Ninguno	Ok
R	N	S	I	W	R*W o ninguno	No concluyente
R	S	N	I	W	RWW	Violación
R	S	N	S	W	RRW	Ok
W	N	N	M	W	Ninguno	Ok
W	N	S	I	W	W*W o ninguno	Ok
W	S	N	S	W	WRW	Violación

y el estado de la línea previo al acceso de escritura local es *Invalid*, entre los accesos locales ha existido una escritura efectuada por un *thread* remoto.

WRW Los accesos locales son de escritura. Existe una petición de *downgrade* entre los accesos locales, y el estado de la línea de cache antes de la segunda escritura local es *Shared*, un *thread* remoto ha leído la línea de cache entre los accesos locales.

Como vimos en la sección 2, estos accesos no son serializables. Si la intención del programador era que esos accesos fueran atómicos, el código no garantiza que el resultado de los accesos sea equivalente a una ejecución secuencial, y por tanto podemos detectar que se ha producido una violación de atomicidad.

3.2. Implementación

La información del protocolo de coherencia cache no se encuentra disponible al programador, es necesario añadir una instrucción al repertorio que permita leer estos *tags*. La introducción de esta nueva instrucción no complica el diseño de la cache ni aumenta su tiempo de acceso. Además, introducimos un nuevo bit en cada línea de cache, el bit D (*downgrade*). Éste bit se pone a uno cuando se detecta una petición de *Downgrade* realizada por el protocolo de coherencia cache, y se borra tras cada acceso local.

Para implementar el algoritmo de detección necesitamos esa nueva instrucción, por lo que utilizamos un simulador de cache que nos permite leer el

estado del protocolo de coherencia cache y aumentar la información de estado con el bit D, y Pin [kLCM+05], que nos permite enlazar la información que recogemos del simulador de cache con el marcado realizado en el código.

Para comprobar que la detección se puede llevar a cabo, y como primera aproximación, será necesario marcar el código con los accesos que pueden ser entrelazados por otro *thread*. Para esto será necesario marcar las dos instrucciones que pueden sufrir un entrelazado no serializable. Utilizando la misma nomenclatura que en [LTQZ06], denominamos instrucción I a la instrucción que espera que no se hayan producido entrelazados no serializables entre el acceso local anterior y el actual (el efectuado por la instrucción I), e instrucción P a la la instrucción que realiza ese acceso local anterior. Dicho de otra forma, vamos a marcar los accesos que deberían ser atómicos. Una vez anotado el código, puede ejecutarse el programa sobre el algoritmo de detección, que actúa en dos fases:

- Cuando se alcanza una instrucción de tipo P, se almacena el tipo de acceso (lectura o escritura) y el contador de programa de la instrucción.

- Cuando se ejecuta una instrucción de tipo I, se recoge la información necesaria del protocolo de coherencia cache (el estado del bloque accedido, justo antes de que se ejecute la instrucción I), el bit D y el tipo de acceso realizado por la instrucción P a la misma variable compartida. Con esta información, como hemos visto previamente, podemos detectar si se ha producido una violación de atomicidad. Si esto ocurre, se reportan los contadores de programa de la instrucción P y de la instrucción I involucradas, el tipo de entrelazado no serializable detectado

y los identificadores de *thread* y proceso[1] del programa en el que se ha manifestado el *bug*.

Hacemos notar que el algoritmo de detección solo interrumpe la ejecución normal del programa cuando se detecta una de estas instrucciones marcadas.

El marcado en el código se realiza con las siguientes funciones:

```
__attribute__ ((noinline)) void p_begin(void * p){asm("");}
__attribute__ ((noinline)) void i_begin(void * p){asm("");}
__attribute__ ((noinline)) void p_end(void * p){asm("");}
__attribute__ ((noinline)) void i_end(void * p){asm("");}
```

El objetivo de estas funciones es delimitar las instrucciones I y P en el código, e informar a la herramienta de detección de la variable compartida sobre la que se tiene que efectuar la comprobación acerca de la serializabilidad de los accesos que se realizan a su posición de memoria.

Mediante PIN, empleamos estas funciones vacías para detener la ejecución del programa y poder inyectar el código que anota el tipo de acceso efectuado por la instrucción P y que efectúa las comprobaciones pertinentes para detectar los entrelazados no serializables. El código inyectado se ejecutará antes de ejecutar el acceso, ya que, para la detección, es necesario saber el estado del bloque de cache al que va a acceder la instrucción I antes de que se realice el acceso. Además, la función recibe un parámetro, que es la dirección de la variable compartida sobre la que se realizarán la detección de violaciones de atomicidad.

[1] Algunas aplicaciones, como Apache, lanzan varios procesos y es útil saber en cual de ellos ha tenido lugar el error

3.2.1. Pin

Pin es una herramienta para la instrumentación binaria de programas que soporta ejecutables de Linux y Windows para las arquitecturas IA-32, Intel(R) 64 e IA-64 que posibilita la creación de herramientas para el análisis dinámico de programas.

Como herramienta de instrumentación binaria dinámica, la instrumentación se lleva a cabo en tiempo de ejecución de los binarios, de esta forma no es necesario recompilarlos y soporta herramientas de instrumentación que generan código dinámicamente.

Pin realiza la instrumentación tomando el control del programa justo antes de que se cargue en memoria. Tras esto realiza una recompilación *Just-in-time*(JIT) de pequeños fragmentos del código binario antes de que sean ejecutados. Las nuevas instrucciones que realizan el análisis se añaden al código recompilado. Estas nuevas instrucciones provienen de la herramienta creada para el análisis, a la que llamaremos *pintool*. Una *pintool* contiene rutinas de instrumentación, análisis y *callback*. Las rutinas de instrumentación son llamadas cuando el código que aun no ha sido recompilado está a punto de ejecutarse, y posibilitan la inserción de las rutinas de análisis. Éstas son llamadas cuando el código al que están asociadas se ejecuta. Las rutinas *callback* se invocan cuando se dan ciertas condiciones, o cuando sucede un determinado evento. Pin proporciona una extensa API para la instrumentación a diversos niveles de abstracción, desde una sola instrucción hasta todo un módulo binario. Además soporta *callbacks* para muchos eventos, como la carga de librerías, llamadas al sistema, señales/excepciones y eventos de creación de *threads*.

Nuestro mecanismo de detección de violaciones de atomicidad se implementa como una *pintool* que busca las instrucciones anotadas en el programa. Una vez que se encuentra una de estas instrucciones, Pin inyectará el código de las funciones de análisis que almacenan la información necesaria si la instrucción es de tipo P, o comprueban si se han producido accesos no serializables, en caso de encontrar una instrucción I. Además, debido a que empleamos un simulador de cache para incorporar los cambios anteriormente comentados, utilizaremos Pin para hacer pasar todos los accesos de memoria del programa sometido a la detección de violaciones de atomicidad por dicho simulador.

Capítulo 4

Resultados

A continuación exponemos y analizamos los resultados obtenidos de la evaluación del algoritmo de detección de violaciones de atomicidad implementado.

Para probar el algoritmo de detección de violaciones de atomicidad empleamos tipos de cargas de trabajo, *kernels* y aplicaciones reales. Los *kernels* generan condiciones extremas para que las violaciones de atomicidad se manifiesten más frecuentemente que en aplicaciones reales. De este modo podemos comprobar si el algoritmo es capaz de detectar las violaciones de atomicidad que se producen en menos tiempo. Además, sometemos al algoritmo de detección dos versiones del servidor web de Apache(http://httpd.apache.org/) y una de MySQL(http://www.mysql.com/) con violaciones de atomicidad conocidas.

Los resultados obtenidos se resumen en la tabla 4.1.

TABLA 4.1: Resultados de las pruebas de detección

Nombre de la carga	*threads*	Violación detectada	Descripción

Apache-extract	2	RWR	Versión kernel del fallo del sistema de *log* de apache
BankAccount	2	RWW	Fallo de una estructura compartida en una cuenta bancaria. Retiradas e ingresos de efectivo simultáneos mal sincronizados pueden causar un balance final inconsistente.
CircularList	2	RWR	Error en el ordenamiento de los datos de una lista de trabajo. Eliminar, procesar y añadir unidades de trabajo a la lista de forma no atómica puede causar el reordenamiento de las unidades de trabajo en la lista.
LogProc&Sweep	5	RWR	Deferenciación a NULL de una estructura de datos compartida. Los *threads* manipulan de forma inconsistente el *log* compartido. Un *thread* hace apuntar el *log* a NULL, el que lo lee falla.
MysSQL-extract	2	WRW	Versión kernel del fallo del sistema de *log* binario de Mysql
StringBuffer	2	RWR	Error de desbordamiento en java.lang.StringBuffer [FQ03]. Al añadir un elemento, no se mantienen todos los *locks*, otro *thread* puede cambiar el *buffer* durante la inserción, haciendo inconsistente el estado

Apache 2.0.48 1	25	RWR	Fallo del sistema de log de apache. Los accesos al tamaño del *buffer* no son atómicos y provocan la corrupción del *log*
Apache 2.0.48 2	25	RWR	Mismo fallo que Apache 2.0.48 1 en otra parte del código
Apache 2.0.5	25	RWR	*dangling pointer* en un contador de referencias del módulo de cache de apache
MySQL 4.0.12	20	WRW	Error en el *log* binario de MySQL. Un *thread* está en proceso de cerrar el anterior *log* binario de la base de datos y abrir el nuevo. Durante este periodo el estado del *log* binario es CLOSE. Otro *thread* necesita anotar en el *log* la última acción de la base de datos, pero lee el valor intermedio del estado CLOSE y no realiza la actualización del *log*. La actualización de la base de datos no es reflejada en el *log* binario, lo que puede causar un problema de seguridad.

Se realizaron pruebas sobre regiones en las que la atomicidad estaba garantizada por la implementación. El algoritmo de detección, como cabía esperar, no fue capaz de detectar entrelazados de acceso a memoria no serializables en estas regiones.

Durante las pruebas no detectamos pérdidas de la capacidad de detección debido a líneas desplazadas de la cache ya que el tamaño de los datos necesarios para realizar las comprobaciones entraban en la cache simulada. Sin embargo, en aplicaciones con cargas reales esto puede suponer un problema. Para evitarlo la mejor opción es incorporar una cache de víctimas que almacene las variables compartidas que son desplazadas de la cache, e incluso podría considerarse gestionar esta cache de víctimas de forma que sólo se almacenen en ésta las variables compartidas que están sometidas al algoritmo de detección.

Capítulo 5

Conclusiones y trabajo futuro

Como hemos visto, es posible realizar la detección de violaciones de atomicidad sobre una variable compartida conociendo únicamente el tipo de los accesos locales y la información que proporciona el protocolo de coherencia cache, ya que gracias a éste podemos obtener información de los accesos remotos que se han efectuado sobre la variable compartida en cuestión.

Realizar el marcado de código de forma manual hace que el número de falsos positivos se reduzca a los posibles casos en los que puedan producirse debido al *false sharing*. Esto puede mitigarse modificando la cache para que el protocolo de coherencia trabaje con un grano más fino, palabras en lugar de bloques, pero la penalización en el rendimiento y la contención generada hacen que sea preferible tolerar cierta imprecisión en la detección. Además, lo que se pretende con los algoritmos de detección de violaciones de atomicidad no es contabilizar cuantas ocurren, podemos asumir cierta imprecisión ya que el objetivo es detectar que entrelazados de acceso a memoria no serlializables se producen durante la ejecución de un programa.

Como hemos comentado en el capítulo 4, las regiones cuya implementación garantizaba la atomicidad pasaban desapercibidas ante el algoritmo de detección. Esto supone una ventaja sobre ciertas herramientas de detección de fallos de concurrencia existentes, que en ocasiones producen falsos positivos en regiones que son totalmente correctas (al contrario de lo que comentábamos previamente, que suponía detectar, en una región no atómica, más fallos de los que realmente se producen).

Durante la elaboración de este trabajo se implementó una versión rudimentaria del algoritmo de detección que prescindía del marcado y que monitorizaba todos los accesos a variables compartidas para detectar entrelazados no serializables. Esta implementación no se optimizó ya que no formaba parte de los objetivos iniciales realizar la detección de violaciones de atomicidad sin anotaciones en el código, pero consideramos que lo más útil para un programador es una herramienta que funcione de la forma más autónoma posible, ya que anotar el código es una tarea tediosa. La principal desventaja de esta versión automática es que produce una cantidad elevada de falsos positivos en todas las aplicaciones probadas(las mismas que se comentaron en el apartado 4). Creemos que el algoritmo puede ser mejorado para minimizar la detección de falsos positivos, por lo que planteamos como trabajo futuro incorporar esta característica.

Las violaciones de atomicidad que involucran accesos a varias variables suponen $\frac{1}{3}$ de todas las violaciones que se producen [LPSZ08] y suponen un campo muy interesante de investigación. Las ideas presentadas en este trabajo pueden ser empleadas para la implementación de un algoritmo que sea capad de detectar tanto violaciones de atomicidad sobre una variable como sobre

varias. Este aspecto tampoco estaba contemplado en los objetivos iniciales del trabajo, y por tanto lo planteamos como trabajo futuro.

Capítulo 6

Trabajo relacionado

En este capítulo se van a describir las técnicas propuestas para enfrentarse a los *bugs* de concurrencia. Dividimos este capítulo en dos secciones que cubren los dos tipos de *bugs* de concurrencia que se han venido investigando en los últimos años, las carreras de datos y las violaciones de atomicidad.

6.1. Carreras de datos

Cuando se escribe código para procesadores multicore de memoria compartida, los programadores deben emplear mecanismos de sincronización para restringir el orden en que los distintos hilos realizan ciertas operaciones y así controlar el acceso a las variables compartidas. Se produce una carrera de datos cuando un programador omite las operaciones de sincronización, permitiendo a más de un hilo acceder a una variable de forma no sincronizada cuando al menos uno de los accesos es de escritura. Existen dos mecanismos básicos para detectar carreras de datos: los algoritmos basados en *happened-*

before y los algoritmos basados en *lockset*.

Los algoritmos basados en *lockset* comprueban que la colocación de los *locks* es correcta monitorizando todas las adquisiciones y liberaciones de los mismos, así como de todas las operaciones de memoria. Para cualquier acceso de memoria a una variable compartida, al menos debe haber un *lock* que se mantiene de forma consistente en el resto de accesos a la misma variable. Para cada variable v, el algoritmo *lockset* mantiene un conjunto de *locks* que han protegido a v hasta el momento en una *conjunto candidato* asociado con v, o *C(v)*. Para cada *thread* t, el algoritmo *lockset* también mantiene un conjunto de *locks* actualmente mantenidos por el *thread* t en el conjunto de *locks* de t, o *L(t)* (*lockset* de t). Un *lock* l se añade o se elimina del *lockset* del *thread* *L(t)* cuando el *thread* t adquiere o libera el *lock* l. Cuando una nueva variable v se inicializa, su conjunto candidato *C(v)* mantiene todos los *locks* posibles. Cuando la variable es accedida por el *thread* t, *C(v)* se actualiza con la intersección de *C(v)* y el *lockset* actual del *thread* *L(t)*. De esta forma se asegura que cualquier *lock* que siempre protege a v estará contenido en *C(v)*. Si *C(v)* está vacío, no hay ningún *lock* protegiendo a v, lo que indica una carrera de datos en potencia.

La relación *happened-before* de Lamport es una relación binaria definida entre eventos de un sistema concurrente. En el marco de la detección de carreras de datos un evento será una época (grupo de instrucciones dinámicas de un *thread* ejecutadas entre dos operaciones consecutivas de sincronización llevadas a cabo por dicho *thread*), esto implica que las mismas restricciones de orden que se aplican a una época también se aplican a todas las instrucciones pertenecientes a ella. La definición de esta relación es la siguiente:

- Si las épocas e_1 y e_2 tienen lugar en el mismo *thread* y e_1 precede a e_2 en el orden del programa, entonces $e_1 \rightarrow e_2$, o lo que es lo mismo, la época e_1 sucedió antes que la época e_2.

- Si la última operación de e_1 está en el origen de una sincronización, y e_2 comienza en el destino de esa sincronización, entonces $e_1 \rightarrow e_2$, la época e_1 ocurrió antes que la época e_2.

- La relación es transitiva: $e_1 \rightarrow e_2, e_2 \rightarrow e_3 \Rightarrow e_1 \rightarrow e_3$

Los mecanismos de detección de carreras basados en esta relación asocian relojes lógicos a cada *thread* para codificar el orden establecido por la relación *happened-before*. Los relojes locales se incrementan cada vez que los *threads* participan en una operación de sincronización. Cada *thread* almacena también el último reloj lógico observado de cada uno de los otros *threads*. De este modo, cada *thread* administra su propio vector de relojes, formado por un componente local y otros tantos componentes como *threads* remotos. cuando dos *threads* sincronizan, el origen manda su vector de relojes al destino, en donde se calcula el máximo elemento a elemento entre el vector recibido y el propio. Alternativamente, el mismo algoritmo puede ser implementado almacenando las direcciones de todas las variables leídas y escritas en cada época en conjuntos de lectura y escritura, junto con el vector de relojes de la época. Se intersecan individualmente estos conjuntos de una época con los conjuntos de las otras épocas que no estén ordenados con respecto a la primera. Si la intersección no es vacía y al menos uno de los conjuntos intersecados es de escritura, se detecta una carrera.

Estas dos perspectivas tienen implementaciones tanto *software* como *hard-*

ware. Entre las propuestas *software* podemos destacar RecPlay [RDB99], que realiza la detección de carreras mediante *happens-before*, Eraser [SBN+97], que emplea el esquema *lockset*, y RaceTrack [YRC05], que es una forma simplificada de *happens-before* que reduce la sobrecarga de tiempo y espacio.
Dado el elevado coste de la detección de carreras empleando únicamente *software*, en trabajos recientes se han desarrollado mecanismos *hardware* para mejorar el rendimiento. Propuestas anteriores se centraron en soluciones híbridas en las que la mayor parte del trabajo se realizaba vía *hardware*. Por ejemplo, HARD [ZTZ07] es una implementación *hardware* del algoritmo *lockset*; SigRace [MSQT09] es una implementación basada en signaturas de *happens-before* que emplea ejecución especulativa para reducir falsos positivos. ReEnact[ref] mejora el soporte de la especulación a nivel de *thread* (TLS, siglas del inglés *Thread Level Speculation*) para detectar y potencialmente corregir en tiempo de ejecución las carreras de datos. A pesar de que estos trabajos reducen la sobrecarga introducida por la detección de carreras significativamente, requieren una cantidad elevada de *hardware* complejo.

6.2. Violaciones de atomicidad

Estudios recientes [LPSZ08] muestran que dos tercios de los fallos de concurrencia, dejando a parte los *deadlocks*, son violaciones de atomicidad. Esto ha causado un creciente interés por este tipo de *bugs*. Las mayor parte de las propuestas existentes se centran en la detección de violaciones de atomicidad que afectan a una sola variable, como AVIO [LTQZ06]. En AVIO se implementa un algoritmo que es capaz de detectar dinámicamente los entrelazados que

dan lugar a violaciones de atomicidad. Para evitar los falsos positivos precisa de entrenamiento, para lo que necesita marcar las ejecuciones como correctas o incorrectas, lo cual no es trivial. Para ayudar a este proceso se definen los invariantes de entrelazado de atomicidad o invariantes AI's. Si denominamos a una instrucción instrucción-I y a la instrucción previa que accede localmente(en el mismo *thread*)a la misma variable compartida instrucción-P, la instrucción-I mantiene un invariante AI si entre los accesos de las instrucciones I y P nunca se ejecuta un acceso remoto a la misma variable compartida de forma no serializable. Durante el entrenamiento se obtienen las instrucciones I a partir de ejecuciones correctas. Con esto se consigue evitar que violaciones de atomicidad intencionadas se detecten posteriormente en ejecuciones normales como errores. El funcionamiento de AVIO sobre un programa de producción simplemente comprueba que las instrucciones I no sean entrelazadas con accesos remotos a la misma variable compartida de forma no serializable, para ello se proponen dos implementaciones: una *hardware*, AVIO-H, que realiza leves modificaciones en la cache y en el protocolo de coherencia de la misma; y otra *software*, AVIO-S que es más precisa que AVIO-H pero penaliza más el rendimiento. Otros trabajos intentan, no solo detectar, sino tratar de evitar las violaciones de atomicidad. Atom-Aid [LDSC08] se apoya en propuestas arquitectónicas que agrupan estáticamente los accesos de memoria en *chunks*, de forma que el ordenamiento de los accesos de memoria tenga un grano más grueso. Esto reduce la probabilidad de que se produzcan entrelazados de operaciones de memoria de distintos *threads*, ocultando violaciones de atomicidad. Atom-Aid mejora esta idea creando bloques atómicos de forma inteligente en lugar de emplear una política estática, disminuyendo la manifes-

tación de violaciones de atomicidad, al tiempo que informa de donde podrían producirse éstas en el código. Esta propuesta precisa que la arquitectura subyacente soporte atomicidad implícita o, de forma más general, un sistema que permita formar bloques atómicos a partir del flujo dinámico de instrucciones, modificaciones de esta arquitectura y cambios en la cache, por lo que su implementación es costosa. Finalmente, Color-Safe [LCS10] contempla el caso de las violaciones de atomicidad que involucran accesos a más de una variable. La idea fundamental es agrupar aquellos datos que estén relacionados en colores y monitorizar los entrelazados de acceso en el "espacio de colores". Las violaciones de atomicidad se evitan insertando transacciones efímeras que impiden entrelazados erróneos. La propuesta de ColorSafe consiste en asociar colores con las variables compartidas, de modo que las variables que están relacionadas tengan el mismo color, así el problema de detectar violaciones de atomicidad en varias variables se simplifica a realizar la detección en un color, lo cual es similar a detectar violaciones de atomicidad en una sola variable. El coloreado de las variables puede realizarse de forma manual, mediante anotaciones en el código, o automáticamente, mediante heurísticas(p.e. asociar el mismo color a todos los bloques de memoria reservados por un mismo `malloc()`). Para llevar a cabo la detección ColorSafe emplea el historial de accesos llevados a cabo por el procesador local, el historial de accesos llevados a cabo por los procesadores remotos(se construye con la información del protocolo de coherencia cache) y las reglas que determinan que entrelazados de accesos en el historial son serializables. ColorSafe tiene dos modos de operación: depuración y despliegue. En el modo depuración se detectan y reportan de forma precisa los entrelazados no serializables entre la historia local y la remota que se producen durante la

ejecución. En el modo despliegue se detectan violaciones que pueden llegar a producirse y dinámicamente se lanzan transacciones efímeras que evitan que estos entrelazados no serializables lleguen a producirse. De forma más precisa, en el modo despliegue se define un entrelazado potencialmente no serializable como un par de accesos a memoria en la historia local que, de ser entrelazados por cualquier acceso de la historia remota, podría ser no serializable. La intuición tras esta definición es que si se observa que casi se ha producido una entrelazado no serializable, es probable que en un punto futuro de la ejecución llegue a producirse, manifestándose una violación de atomicidad.

Bibliografía

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, 2003.

[kLCM+05] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[LCS10] Brandon Lucia, Luis Ceze, and Karin Strauss. ColorSafe: Arquitectural Support for Debugging and Dynamically Avoiding Multi-variable Atomicity Violations. In *International Symposium on Computer Architecture*, Saint-Malo, France, June 2010.

[LDSC08] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and Survivinfg Atomicity Violations. In *International Symposium on Computer Arquitecture*, pages 277–288, Beijing, China, June 2008.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Not.*, 43(3):329–339, 2008.

[LTQZ06] Shan Lu, Jopseph Tucek, Feng Qin, and YuanYuan Zhou. AVIO:Detecting Atomicity Violations via Access Interleaving Invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 2006.

[MSQT09] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. Sigrace: signature-based data race detection. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 337–348, New York, NY, USA, 2009. ACM.

[RDB99] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[YRC05] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating*

systems principles, pages 221–234, New York, NY, USA, 2005. ACM.

[ZTZ07] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 121–132, Washington, DC, USA, 2007. IEEE Computer Society.

Índice de figuras

Índice de tablas