Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter Chen, Jason Flinn. University of Michigan

# Respec: Efficient Online Multiprocessor Replay via Speculation an External Determinism

CARLOS MARTÍN FLORES GONZÁLEZ, Carné: 2015183528

**Instituto Tecnológico de Costa Rica**
**Maestría en Computación**
**Sistemas Operativos Avanzados**
**Profesor: Francisco Torres Rojas, Ph.D**

Sistemas de repetición determinista (*deterministic replay systems*) son usados para grabar y reproducir la ejecución de un sistema de hardware o software. Esta habilidad puede ser usada para mejorar sistemas a lo largo de muchas dimensiones, incluyendo confianza, seguridad y debugabilidad (*debuggability*). La idea general detrás de repetición determinista es la de registrar (*log*) todos los eventos no deterministas durante la grabación y reproducir estos eventos durante la repetición. Repetición determinista para uniprocesadores puede ser proporcionado a un bajo *overhead* porque los eventos no deterministas ocurren en frecuencias relativamente bajas, por esto es que el registrar eventos añade poco *overhead*.

Respec se basa en dos ideas: (1) puede registrar de manera optimista el orden de las operaciones en memoria de forma menos precisa que el nivel necesario para garantizar repetición determinista, mientras ejecuta la ejecución grabada especulativamente para garantizar seguridad. Luego de un número configurable de faltas de especulación (*misspeculations* - esto es cuando la información registrada no es suficiente para asegurar repetición determinista para un intervalo), Respec devuelve la ejecución al inicio del intervalo actual y se re-ejecuta con un *logger* más preciso. (2) Respec puede detectar *misspeculations* para un intervalo al repetir concurrentemente el intervalo grabado en *spare cores* y verificando si la salida del sistema y los estado final del programa coincide con los de la ejecución grabada. Fases de Respec: (1) Registra la mayoría de las operaciones de sincronización ejecutadas por un programa *multithreaded* de memoria compartida. (2) Detecta cuando operaciones de sincronización registradas son insuficientes para reproducir un intervalo de la corrida original. Respec repite concurrentemente un intervalo grabado en *spare cores* y lo compara con la corrida original. (3) Ejecución especulativa para ocultar los efectos de intervalos de repetición fallidos y para retroceder de forma transparente ambas ejecuciones, la grabada y la repetida. (4) Luego de devolverse, Respec reintenta la ejecución de un intervalo fallido de ejecución serializando los *threads* y registrando el orden, lo que garantiza que la repetición tendrá éxito para ese intervalo.

**Replay guarantees:** *Fidelity level:* sistemas de repetición difieren en su fidelidad de repetición y en el costo resultante de proporcionar esta fidelidad. Un ejemplo de fidelidades diferentes está en el nivel de abstracción en que una repetición está definida (nivel de máquina, de proceso, de programa). Repetición multiproceso añade otra dimensión de fidelidad: cómo debería la ejecución repetida reproducir las instrucciones de intercalación de diferentes *threads*. No se ha propuesto ninguna aplicación de repetición que requiera de ordenamiento basado en tiempo exacto de todas las instrucciones que se van a reproducir. La dificultad e ineficiencia de métodos de registro pesimistas han conducido a explorar un nuevo nivel de fidelidad para repetición, llamado repetición externa determinista (*externally deterministic replay*). Garantiza que: (1) la ejecución repetida sea indistinguible de la ejecución original desde la perspectiva de un observador externo y (2) la ejecución repetida es una ejecución natural del programa de interés. El primer criterio implica que una secuencia de instrucciones ejecutada durante la repetición no puede ser probada que difiera de la secuencia de instrucciones ejecutadas durante la ejecución original porque todas las salidas observables de las dos ejecuciones son las mismas. El segundo criterio implica que cada estado visto durante la repetición estuvo fue capaz de ser producido por el programa de interés. *Online versus offline replay: Offline*: usos forenses o *debugging*, la repetición se realiza luego que la ejecución original haya finalizado. El sistema de repetición se ejecuta más lento que el original. *Online:* verificaciones paralelas, tolerancia a fallos. La ejecución repetida procede en paralelo con la original. La velocidad de la ejecución repetida es importante porque puede limitar el rendimiento del sistema. Respec está diseñado para usarse en escenarios *online*, busca minimizar el *overhead* de registro y repetición para que pueda ser usado en entornos de producción con garantías sincrónicas de tolerancia a fallos o verificación de errores de programa.

**Design:** Respec proporciona repetición determinista para uno o más procesos. Se repite en la abstracción del proceso al registrar los resultados de *system calls* y de operaciones de sincronización de bajo nivel ejecutadas por el proceso de grabación y proporcionando los resultados registrados a los procesos de repetición en lugar de re-ejecutar el correspondiente *system call* y operaciones de sincronización. Respec crea puntos de verificación (*checkpoints*) para el proceso de grabación a intervalos semi-regulares llamados *epochs*. El proceso de repetición inicia y finaliza un epoch en exactamente el mismo punto en la ejecución que el proceso de grabación. Durante un epoch, cada *thread* grabado registra la entrada y salida de sus *system calls*. Cuando un *thread* de repetición encuentra un *system call*, en lugar de ejecutarlo, lo emula leyendo el registro para producir valores de retorno y modificaciones de espacio de direcciones idénticos a aquellos vistos por el *thread* de grabación. La repetición, podría falla cuando un epoch ejecuta un *unlogged synchronization* o *data race*. Respec realiza un *divergence check* para detectar esos fallos en la repetición. *Divergence Checks:* verificar el estado del programa al final de cada epoch no es estrictamente necesario para garantizar repetición externa determinista. Sin embargo, verificar el estado intermedio trae beneficios: (1) permite a Respec hacer *commit* en epochs y entregar la salida del sistema. (2) reduce la cantidad de ejecución que debe ser devuelta cuando una verificación falla. (3) permiten a otras aplicaciones paralelizar verificaciones de confianza. Los *divergence checks* de Respec garantiza encontrar todas las instancias cudno la repeticion no es externamente determinista con respecto a la ejecución grabado. Pero, esto no significa que la ejecución *unlogged race* siempre va a causar que el *divergence check* falle. Para cierto tipos de *unlogged races*, el *divergence check* de Respec va a tener éxito. Esto reduce el número de retrocesos necesarios para producir repetición externa determinista.

**Implementación**. *Checkpoint and multithreaded fork:* se creó una nueva primitiva de Linux llamada *multithreaded fork*, la cual crea un proceso hijo con el mismo número de *threads* que su padre. Respec usa la primitiva de *multithreaded fork* en dos circunstancias: (1) para crear un proceso repetido idéntico al que se está grabando y (2) para crear puntos de verificación en el proceso de grabación que pueden ser restaurados en un *rollback*. Respec borra los puntos de verificación una vez que el siguiente punto de verificación ha sido verificado en que coincide con proceso de grabado y de repetición. Respec pone puntos de verificación en el proceso de grabación en intervalos semi-regulares llamados epochs. El punto de verificación inicial se pone cuando el proceso de repetición es creado por primera vez. Entonces, espera que pase una cantidad predeterminada de tiempo. Luego que transcurre el intervalo del epoch, el siguiente *system call* de cualquier *thread* de grabación dispara un punto de verificación. Luego de que los restantes *threads* alcancen el *multithreaded fork barrier* y el punto de verificacio4n es creado, todos los *threads* continuan su ejecución. El proceso de grabación se puede ejecutar varios epochs adelante del proceso de repetición. Continua hasta que el proceso se devuelva o termine. *Speculative execution:* Al proceso de grabación no se le permite externalizar una salida[1], hasta que ambos procesos, el de grabación y repetición, completen el epoch durante el cual la salida fue intentada y los estados de ambos procesos coincidan. Enfoque: usar el soporte del sistema operativo para ejecución especulativa. Se puede ejecutar el *thread* de grabación de manera especulativa y poner en un búfer la salida o bien permitir que el estado especulativeo se propague más allá del proceso de grabación mientras que el sistema operativo garantice que el estado especulativo pueda ser devuelto y el estado especulativo no va a afectar causalmente ninguna salida. Se usa Speculator para hacer esto. Speculator permite que un estado especulativo se propague a través de varias formas de IPC. De esta form estructuras adicionales del kernel se vuelven especulativas sin bloquear el proceso de grabación. La salida externa se pone en un búfer dentro del kernel cuando es posible y solo es liberada cuando los puntos de verificación de los cuales la salida depende fueron *committed*. Entradas externas son guardadas como parte del estado del punto de verificación así puede ser restauradas luego de un *rollback*. Si la propagación de un estado especulativo o el *buffering* de una salida no es posible, el *thread* de grabación finaliza el epoch actual, se bloquea hasta que el *thread* de repetición lo alcanza y compara los estados, inicia un nuevo epoch y libera la salida. *Logging and replay:* Una vez que el proceso de repetición es creado, se ejecuta concurrentemente con su proceso de grabación. Cada *thread* grabado registra los *system calls* y operaciones de sincronización de nivel de usuario que realiza, mientras el corresponiente *thread* de repetición consume los registros para recrear los resultados y el orden parcial de ejecución para las operaciones registradas. Se usa un registro en la memoria del kernel que contiene información de los *system calls* y un registro de nivel de usuario que contiene operaciones de sincronización. *User-level logging:* Respect registra la entrada y la salida de cada operación de sincronización. Cada registro conteiene el tipo de operación, su resultado y su orden parcial con respecto a otras operaciones registradas. El order parcial captura el orden total de todas las operaciones de sincronización que acceden a la misma variable de sincronización y el orden del programa de las operaciones sincronizadas ejecutadas en el mismo *thread*. Para grabar el orden parcial, se aplica una función de hash a la dirección del *lock*, futex u otra estructura de datos que se opera sobre un número fijo de relojes de registro glocal (512). Cada operación grabada incrementa atómicamente un reloj y graba el valor del reloj un búfer circular productor-consumidor compartido entre el *thread* de grabación y su correspondiente *thread* de repetición. De esta forma, grabar un

[1] enviar un paquete de red, escribir en consola, etc

registro requere a lo sumo de dos operaciones atómicas. Esto permite alcanzar un *overhead* razonable incluso cuando las operaciones de sincronización que no requieren de un *system call*. *Kernel logging:* Respec usa un solo reloj para asegurar que el proceso de grabación y repetición siguen el mismo orden total para entradas y salidas de *system calls*. Con este mecanismo el proceso de repetición usualmente no ejecuta el *system call* grabado, solo reproduce los resultados de la llamada. Sin embargo, ciertos *system calls* que afectan el espacio de direcciones de la aplicación tiene que ser re-ejecutados por el proceso que invoca[2]. *Detecting divergent replay:* Cuando Respec determina que un proceso de grabación y repetición han divergido, devuelve la ejecución al último punto de verificación en que el proceso de grabación y repetición coincidieron. Un *rollback* debe ser realizado cuando el proceso de repetición intenta realizar una salida externa que difiere de la salida producida por el proceso de grabación. Hasta que ese desajuste no se presente no hay necesidad de realizar un *rollback*. Respec verifica que que el *thread* repetido haga *system calls* y operaciones de sincronización con los mismos argumentos que el *thread* de grabación. Si ambos no coinciden, los procesos se devuelven. Además, final de cada epoch, Respec compara que el espacio de direcciones y los registros de los procesos grabados y repetidos. Respec va a hacer que los procesos se devuelvan si difieren en algún valor. *Rollback:* el *rollback* se dispara cuando los estados de memora difieren al final de un epoch o cuando hay un desajuste en el orden de los argumentos de los *system calls* o en la operaciones de sincronización ocurren. Estos desajustes son siempre detectados por el proceso de repetición, debido a que se ejecuta detrás del proceso de grabación. Respec usa Speculator para devolver el proceso de grabación al último punto de verificación en que el programa coincidió. Speculator, cambia el proceso, *thread* y otros identificadores del proceso que se devuelve con los que tiene el punto de verificación, permitiendo al punto de verificación asumir la identidad del proceso que se ha devuelto. Luego induce a los *threads* del procesos grabado a salir. Luego que se devuelve, el proceso repetido también sale. Inmediatamente luego que un punto de verificación es restaurado, el *thread* grabado crea un nuevo proceso de repetición. También crea un nuevo punto de verificación usando Speculator. Ambos *threads*, el de grabación y el de repetición retoman su ejecución. *Offline replay support:* Cuando se solicita, Respec puede opcionalmente guardar información para habilitar repetión *offline* del proceso grabado. Esta información incluye el registro del kernel de los *system calls*, registros de operaciones de sincronización de nivel de usuario y sumas de verificación MD5 de espacio de direcciones y registros de estado al final de cada *committed rollback*. Debido a que no todos los *races* son registrado, repetición *offline* del proceso grabado no garantiza éxito en el primer intento. Sin embargo, como el proceso grabado ha sido repetido exitosamente al menos una vez, es probable que la repetición *offline* tendrá éxito eventualmente, aunque puede requerir de un número de *rollbacks* y reintentos. *Security considerations:* al cumplir las metas de repetición externa determinista se asegura que el proceso repetido pasa todos las verificaciones de seguridad, una ejecución natural que pasa todas las verificaciones de seguridad existe, y así el salida y el estado producido por el proceso original y el repetido son seguros con respecto a esas verificaciones. Lo más que puede hacer un atacante es escoger cuál ejecución natural va a ejecutar el proceso de repetición, pero esta ejecución natural debe aún coincidir con la salida y el estado del proceso original. Se asume también que el software que se graga y se repite no puede corromper los datos del kernel. Por lo tambien, se pone Respec lo más cerca posible dentro del kerel. Speculator, el kernel log y los puntos de verificacio4n de memoria se ponen en el kernel. Los únicos datos de repetición que software malicioso puede corromper es el registro de nivel de usuario compartido entre el proceso que graba y el que repite; este registro contiene el orden de operaciones de sincronización de nivel de usuario. Por lo tanto, Respec tiene que tratar el registro de nivel de usuario de forma sospechosa. Para protegerse de estos ataques, Respec tiene un modo opcional de verificación que puede ser utilizado durante la repetición. Cuando el modo de verificación se habilita, el proceso de repetición copia cada registro del registro compartido de nivel de usuario a una región de memoria no compartida, luego verifica que el registro representa una posible ruta de ejecución.

## 1. ¿CUÁL ES EL PROBLEMA QUE PLANTEA EL *PAPER*?

Sistemas de repetición determinista graban y reproducen la ejecución de un sistema de hardware o software. Mientras es bien sabido cómo repetir en sistemas uniprocesador, la repetición en sistemas multiprocesador de memoria compartida con bajo *overhead* en hardware básico es todavía un problema abierto.

## 2. ¿POR QUÉ EL PROBLEMA ES INTERESANTE O IMPORTANTE?

Desafortunadamente, es más difícil proporcionar repetición determinista de programas *multithreaded* de memoria compartida en multiprocesadores porque los accesos a la memoria compartida añaden una fuente de no determinismo de alta frecuencia. Una variedad de enfoques se han propuesto para reproducir este no determinismo al registrar el orden preciso de los accesos a memoria, pero estos enfoques requieren o hardware personalizado o son prohibitivamente lentos para muchas aplicaciones en paralelo. Otros enfoques garantizan determinismo solamente en programas *race-free* o solo pueden reproducir el estado parcial del sistema original.

[2] `clone, exit, mmap2, mprotect`

## 3. ¿QUÉ OTRAS SOLUCIONES SE HAN INTENTADO PARA RESOLVER ESTE PROBLEMA?

- IGOR: uno de los primeros grabadores (*recorders*), usa soporte de puntos de verificación *copy-on-write* en el sistema oeperativo para grabar y reproducir el estado intermedio.
- Hyperviser, Boothe y Flashback, instmentan el sistema operativo para grabar y repetir eventos no determistas.
- DejaVu y jRapture: graban la mayoría de los eventos no deterministas usando la máquina virtual de Java.
- ReVirt y ReTrace: usan el soporte del monitor de máquinas virtuales el cual brinda una interface entre los sistemas operativos huésped y anfitrión.

Ninguno de los anteriores soporta repetición en multiprocesadores porque no pueden grabar y repetir el orden no determinista entre los accesos a memoria compartida y accesos ejecutados por *threads* concurrentes.

InstantReplay instrumenta cada acesso a memoria en un objeto compartido al grabar el orden en que diferentes *threads* lo acceden. Otras soluciones como PinSel y Microsoft DNA hacen lo mismo. En lugar de grabar el orden de todos los accesos a memoria RecPlay y JaRec instrumentan solo las operaciones de sincronización y graban el orden de su ejecución. Este enfoque solamente asegura repetición determinista de programa hasta que el primer *data race*, lo que limita el uso del reproductor de muchas formas.

Netzer propuso un algoritmo de reducción transitiva para reducir el número de registros de orden de memoria necesarios a grabar.

ODR y PRES, son similares a Respec pero se concentran en repetición *offline*.

## 4. ¿CUÁL ES LA SOLUCIÓN PROPUESTA POR LOS AUTORES?

Respec, una nueva forma de dar soporte a repetición determinista en programas *multithreaded* de memoria compartida en hardware multiprocesador básico. Respec apunta a repetición *online* en la cual los procesos de grabación y repetición son ejecutados concurrentemente. Respec usa dos estrategias para reducir el *overhead* manteniendo la correctitud: *logging* especulativo y repetición externa determinista. *Logging* especulativo registra menos información acerca de las dependencias de memoria compartida que lo que se necesita para garantizar repetición determinista, entonces recupera y hace reintentos si el proceso de repetición diverge del proceso grabado. Repetición externa determinista relaja el grado en qeu dos ejecuciones deben coincidir al requerir solo que la salida de sistema y los estados finales de programa coincidan. Se muestra que estas dos técnicas dan como resultado bajo *overhead* en la grabación y en la repetición para el caso común de ejecución de intervalos *data-race-free* y aun asegurar una correcta repetición para la ejecución de intervalos que tienen *data races*.

## 5. ¿QUÉ TAN EXITOSA ES ESTA SOLUCIÓN?

*Record and replay performance:* Los resultados mostraron que el *overhead* de Respec es generalmente bajo. Para dos *worker threads*, Respec tiene un *overhead* promedio con respecto a la ejecución original de solo 18 % a través de todas las comparaciones. Este *overhead* incrementa gradualmente con el número de *threads*: con 4 *threads*, el *overhead* promedio de Respec es de 55 %. Comparado con el límite inferior de ejecución redundante, el promedio de Respec es de 16 % con 2 *threads* y 40 % con 4 *threads*. Se cree que este incremento se deriva en su mayoría del incremento de sincronización entre *threads* de repetición.

*Rollback frequency:* Se corrió el programa pbzip2 100 veces. Se pudo ver que 13-16 % de las ejecucuiones con más de un *worker thread* contenían un *rollback*. En general, el costo del *rollback* fue razonable. Los *rollbacks* contribuyen 8 % al *overhead* total de Respec cuando pbzip2 usa múltiples *worker threads*. Par el programa aget, se muestra que la salida diverge del estado de la memoria lo que conduce a *rollbacks*. Sin embargo, el impacto en el rendimiento de estos *rollbacks* es despreciable porque Respec pone puntos de verificación en aget muy frecuentemente.

*The cost of rollback:* Para entender mejor el costo de los *rollbacks*, se insertaron *rollbacks* artificialmente dentro de una de los *benchmarks* emulando la falla de una verificación de divergencia durante la ejecución del *benchmarks*. Se desabilitó el algoritmo de epoch adaptativo de Respec y se puso el intervalo de epoch manualmente. Los resultados muestras que el *overhead* del *rollback* es aproximadamente proporcional a la longitud del intervalo del epoch. Intuitivamente, el tiempo de ejecución incrementa con el cantidad de trabajo que se debe rehacer luego de un *rollback*. Como experimiento final, se varió el número de *rollbacks* durante la ejecución del *benchmark* mientras se mantenía el invervalo del epoch fijado en 100ms. Se pudo ver que el costo de los *rollbacks* es proporcional al número de *rollbacks*. Las aplicaciones fluidanimate y bodytrack mostraron un costo aproximado de 160-180ms por *rollback* (69-80ms más grande que el intervalo del epoch). Luego de estudiar esto se encontró que la razón de esto es la implementación de la barrera para puntos de verificación; puede tomar varias decenas de milisegundos para que todos los *threads* alcancen la barrera para estos dos *benchmarks*.