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

DoublePlay: Parallelizing Sequential Logging and Replay

CARLOS MARTÍN FLORES GONZÁLEZ, Carné: 2015183528

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

Se describe una nueva forma de garantizar repetición determinista en multiprocesadores accesibles. Este método combina sencillez y el bajo *overhead* en la grabación de registro de un programa *multithreaded* en un uniprocesador con la velocidad y la escalabilidad de ejecución de un programa en un multiprocesador. La idea básica es que se peude usar un mecanismo más simple y rápido de grabación y repetición en un solo procesador y aún así lograr la escalabilidad ofrecida por múltiples núcleos, utilizando una ejecución adicional para paralelizar la grabación y la repetición de una aplicación. La meta es que una ejecución en un solo procesador sea tan rápida como una ejecución en paralelo tradicional, pero mantener la facilidad de registro de la ejecución *multithreaded* de un solo procesador.

Uniparallelism. Es una técnica para lograr los beneficiones de ejecución en un solo procesador al mismo tiempo que se permite a la aplicación escalar con el aumento de procesadores. Una ejecución uniparalela consiste de ejecución *thread-parallel* y ejecución *epoch-parallel* del mismo programa; esto permite el uso de técnicas que corren solo en un uniprocesador. A diferencia de una ejecución tradicional *thread-parallel* que escala con el número de núcleos corriendo diferentes *threads* en diferentes núcleos, una ejecución *epoch-parallel* logra escalabilidad de una manera diferente, corriendo diferentes epochs (segmentos de tiempo) de la ejecución en múltiples núcleos. Por lo tanto, ejecución *epoch-parallel* requiere de la habilidad de predecir el estados del programa futuros; tales predicciones son generadas corriendo la segunda ejecución *thread-parallalel* concurrentemente. Consecuentemente, para cargas de trabajo intensivas en CPU, la ejecución uniparalela requiere del doble del número de núclos de una ejecución tradicional, conduciendo a un aumento de aproximadamente 100 % de la utilización de CPU y energía. Uniparalalelismo mejora el rendimiento en dos casos: (1) si una carga de trabajo no es intensiva en CPU o si la aplicación no puede escalar y utilizar todos los núcleos eficientemente en una computadora *multicore*, el costo de uniparalelismo puede ser mucho menos que del 100 %(típicamente menos de 20 %). (2) cuando la propiedad que se proporciona es más del doble de costosa en un multiprocesador que en un uniprocesador, es más eficiente en términos de tiempo y energía correr la aplicación dos veces con uniparalelismo que proporcionar la propiedad directamente en la versión multiprocesador de la aplicación.

Design Overview. La meta de DoublePlay es la de grabar eficientemente la ejecución de un proceso o group de procesos que corren en un multiprocesador tal que la ejecución pueda ser repetida de forma determinista luego todas las veces necesarias. DoublePlay corre dos ejecuciones simultaneas del programa que se está grabando. La ejecución *thread-parallel*, corre múltiples *threads* concurrentemente en los núcleos asignados para ello. DoublePlay particiona la ejecución *thread-parallel* en segmetos de tiempo llamados *epochs*. Al inicio de cada epoch, DoublePlay crea un punto de verificación *copy-on-write* del estado de la ejecución *thread-parallel*. La ejecución *epoch-parallel* corre cada epoch en uno de los núcleos asignados para ello. Todos los *threads* de un epoch dado corren en el mismo núcleo, lo que simplifica la tarea de repetir de forma determinista el resultado de la ejecución. Durante la grabación, DoublePlay guarda tres elementos que son suficientes para garantizar que la ejecución de un proceso puede ser repetido de forma determinista en el futuro. (1) Graba el estado inicial del proceso al inicio de la grabación. (2) DoublePlay graba el orden y los resultado de los *system calls*, señales y sincronizaciones de bajo nivel en GNU `libc`. (3) graba la planificación de la ejecución del *thread* de la ejecución *thread-parallel*. Las dos ejecuciones puede ser vistas de la siguiente forma. La ejecución *epoch-parallel* es la ejecución real del program que está siendo grabado. La ejecución *epoch-parallel* corre en un solo núcleo. DoublePlay puede usar el mecanismo de repetición determinista uniprocesador para grabar y reproducir su ejecución. La ejecución *thread-parallel*

permite a la ejecución *epoch-parallel* escalar de acuerdo al número de núcleos. La ejecución *thread-parallel* proporciona un *hint* en cuanto a lo que el estado futuro de la ejecución del proceso será en cada transición de epoch. Mientras que este *hint* sea correcto, el estado del proceso al inicio de cada epoch en la ejecución *epoch-parallel* coincidirá con el estado del proceso al final del epoch previo. Esto significa que los epochs se pueden juntar para forma una sola y natural ejecución del proceso. Si el estado de la ejecución *epoch-parallel* ha divergido del estado de la ejecución *thread-parallel*, DoublePlay aplasta (*squaches*) la ejecución de todos los epochs subsecuentes para ambas ejecuciones. Restaura el estado de cada ejecución *thread-parallel* al punto de verificación en el inicio de cada epoch y reinicia ambas ejecuciones desde este punto. Las salidas no pueden ser externalizadas hastas que todos los epochs previos han sido encontrados como libres de divergencia. DoublePlay usa *online replay* para reducir la frecuencia de tales divergencias y sus resultantes atrasos. La única situacio4n en que la ejecución *thread-parallel* y la ejecución *epoch-parallel* podrían divergir es cuando el programa contiene un *data race* y dos *threads* se ejecutan sin sincronizar, haciendo que operaciones entren en conflicto en un espacio de memoria compartido. En cualquier momento subsecuente, DoublePlay puede repetir una ejecución grabada al (1) restaurar el estado inicial del proceso grabado, (2) repitiéndolo en un sólo núcleo usando los *sistem calls* registrados, señales y operaciones de sincronización y (3) usa la misma planificación de la ejecución del *thread* que fue usada durante la ejecución *epoch-parallel*.

Implementación. La implementación de DoublePlay aprovecha el trabajo previo de Respec para proveer repetición *online* determinista. Respec asegura que dos ejecuciones concurrentes del mismo proceso son externamente deterministas, lo que significa que las dos ejecuciones ejecutan los mismo *system calls* en el mismo orden y que el estado del programa de los dos procesos es idéntico al final de cada epoch de ejecución. Respec sólo asegura determinismo externo mientras dos procesos se ejecutan al mismo tiempo. No aborda repetición determinista de un proceso luego de que la ejecución original se completa. **Supporting offline replay:** DoublePlay hace varias mejoras en la infraestructura básica de Respec con el fin de soportar repetición *offline*. **Enabling concurrent epoch execution:** DoublePlay hace múltiples copias de la ejecución *thread-parallel* llamando la primitiva de *multi-threaded fork* antes de inicar la ejecución la ejecución de cada epoch individual. Esta primitiva crea un nuevo proceso cuyo estado es idéntico al de la ejecución *thread-parallel* en el punto de su ejecución. Cada vez que un nuevo proceso es creado, DoublePlay no le permite iniciar su ejecución, sino que en lugar de esto, lo pone en un *epoch queue* ordenada por el tiempo de creación del proceso. El *scheduler* de DoublePlay es el responsable de decidir cuando y donde cada proceso va a correr. No hay garantía que la ejecución de un epoch finalizará en orden, debido a que algunos epochs son más cortos que otros . DoublePlay usa un algoritmo adaptativo para variar la longitud de los epochs. Aunque los epochs pueden finalizar no ordenados, DoublePlay asegura que ellos hacen *commit* o *rollback* en orden secuencial. Cuando un epoch completa su ejecución, DoublePlay realiza un *divergence check* comparando su memoria y registros de estado con los del punto de verificación asociado con el próximo epoch. Cuando la verificación pasa, DoublePlay le permite al proceso salir y asigna su procesador a otro epoch. Cuando falla, DoublePlay finaliza todos los *threads* que ejecutan el epoch actual y cualquier epoch futuro en la ejecución *epoch-parallel*, así como todos los *threads* de la ejecución *thread-parallel*. **Replaying thread schedules:** DobulePlay garantiza repetición *offline* determinista al ejecutar cada epoch en un solo núcleo usando la misma planificación de *thread* que fue usada durante la grabación de la ejecución *epoch-parallel*. Una estrategia es usar el mismo planificador determinista durante la ejecución *epoch parallel* y la repetición *offline*. La segunda estrategia es la de registrar las decisiones de planificación hechas durante la ejecución *epoch parallel* y repetir aquellas deciciones de manera determinista durante la repetición *offline*. Para la primera estrategia, se asigna una prioridad estricta a cada *thread* basado en el orden en que los *threads* fueron creados. DoublePlay siempre escoge el *thread* con la prioridad más alta que preserve el orden total de los *system calls* y el orden parcial de las operaciones de sincronización. Para permitir apropiaciones *preemptions* se implementa la segunda estrategia para repetir la planificación de los *threads* de forma determinista, la cual es la de registrar las apropiaciones que ocurren durante la ejecución *epoch parallel* y repetir esas decisiones de forma determinista durante la repetición *offline*. **Offline replay:** DoublePlay graba los *system calls* y las operaciones de sincronización ejecutadas durando un epoch en un conjunto de archivos de bitácora. Luego de que cada epoch es *committed*, DoublePlay marca las entradas que pertenecen a cada epoch como elegibles para ser escritas en disco. Luego escribe los registros marcados de forma asincrónica mientras otros epochs se ejecutan. Nótese que DoublePlay solo tiene que grabar los resultados de operaciones de sincronización o *system calls*, ya que los argumentos a esas llamadas serán reproducidas de forma determinista por cualquier proceso de repetición. DoublePlay guarda el estado inicial del proceso cuando la grabación inicia. Para realizar una repetición *offline*, inicia desde esta copia inicial. DoublePlay corre la repetición *offline* en un solo procesador y usa un algoritimo de planificación (el de la sección anterior) para restringuir el orden de la ejecución del *thread* para el proceso de repetición *offline*. Cuando un *thread* de repetición *offline* ejecuta un *system call* o una operación de sincronización, DoublePlay retorna los resultados grabados en su archivo de bitácora. DoublePlay solamente ejecuta *system calls*

que modifican el espacio de direcciones del processo como `mmap2` y `clone`. DoublePlay entrega las señales grabadas en el mismo punto en la ejecución del proceso en la que fueron entregadas durante la grabación. **Forward recovery:** Una vez que se vió que la ejecución *epoch-parallel* como la que se tenía que ser grabada, fue claro que el estado de su ejecución en el momento de un fallo de verificación de divergencia es un estado válido de ejecución que puede ser reproducido *offline* de forma determinista. Por lo tanto, DoublerPlay puede usar el estado del proceso *epoch-parallel* en el momento de un fallo de verificación de divergencia como un punto de verificación de donde reiniciar la ejecución. A este proceso se le llama recuperación hacia adelante (*forward recovery*). DoublePlay registra una operación para deshacer cada *system call* que modifica el estado del kernel, así recuperación hacia adelante (o normal) puede devolver el estado del kernel al aplicar las operaciones para deshacer. *Forward recovery* garantiza que DoublePlay haga progreso hacia adelante incluso cuando una verificación de divergencia falla. Todo el trabajo hecho por la ejecución *epoch-parallel* hasta la verificación de divergencia, incluyendo al menos un *data race*, será preservado. En el peor caso, cada epoch puede contener *data races* frecuentes y verificaciones de divergencia que siempre fallen. *Looser divergence checks:* con la implementación de *forward recovery*, se decidió que realizar verificaciones estrictas de divergencia no era lo mejor. Se modificó DoublePlay par soportar tres tipos de formas más relajadas de verificación de divergencia en un epoch. (1) si el búfer circular de un *thread* contiene un *system call* sin ningún efecto hacia el exterior de este *thread*, se permite omitir el registro extra en el búfer. (2) si un *thread* ejecuta un *system call* que produce una salida para un dispositivo externo, como una pantalla o una red, se permite que la ejecución *epoch-parallel* ejecute el mismo *system call* con diferente salida. (3) si un *thread* ejecuta un conjunto de operaciones de sincronización *self-canceling* o *system calls* durante la ejecución *thread-parallel*, entonces todos los miembros de ese conjunto pueden ser saltados. Beneficios de *looser divergence*. (1) la ejecución *epoch parallel* puede correr por más tiempo antes que se necesite un *rollback*. (2) a veces un *rollback* puede ser evitado en conjunto cuando la divergencia del estado proceso del proceso es transitoria. **Reverse scheduling:** la idea detrás de *reverse scheduling* es la de intentar evitar *rollbacks* cuando una verificación de divergencia falla al encontrar una planificación diferente cuya ejecución cause que la verificación pase. Así, cuando una verificación falle, DoublePlay corre otra ejecución *epoch-parallel* del epoch fallido en el procesador asignado para ese epoch, pero usa las prioridades de planificación opuestas de las que se usaron por la ejecución fallida. Esto es, el último *thread* creado tiene la prioridad más alta, no la más baja. La idea es que si hay un solo *data race* en el epoch, entonces tanto la planificación normal o la reversa son muy probables a reproducir el mismo estado del programa producido por la ejecución *thread parallel*. Si una segunda ejecución pasa la verificación de divergencia, el epoch es *committed* y ningún *rollback* ocurre. La planificación reversa del *thread* es grabada en la bitácora de DoublePlay asi puede ser usada también durante la repetición.

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

La repetión determinista en multiprocesadores es una tarea retadora de implementar eficientemente debido a la necesidad de reproducir el orden o los valores que son leídos por operaciones de memoria compartida en múltiples *threads*.

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

La repetición determinista funciona registrando todos los eventos no-deterministas durante la fase de grabación, luego reproduce estos eventos durante la etapa de repetición. En uniprocesadores, los eventos no-deterministas ocurren relativamente con poca frecuencia, por este motivo registrarlos y repetirlos añade poco *overhead*. Si hay múltiples *threads* corriendo en el uniprocesador, estos puede ser repetidos a un bajo *overhead* registrando y reproduciendo la planificación del *thread*. Sin embargo, en multiprocesadores los accesos a memoria compartida añaden una fuente de no-determinismo de alta frecuencia, y registrar y repetir estos accesos puede reducir drásticamente el rendimiento. Aunque se han propuesto varias alternativas para reducir el *overhead* de la grabación y la repetición de programas *multithreaded*, en memoria compartida, y con múltiples procesadores, muchos se quedan cortos en alguna forma u otra. Algunos enfoques requiren hardware personalizado, otras son muy lentos y otros sacrifican la garantía de poder reproducir la ejecución grabada sin la posibilidad de una búsqueda prohibitivamente larga.

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

- Muchos sistemas de repetición uniprocesador: IGOR, Hypervisor, Mach 3.0 Replay, DejaVu, ReVirt y Flashback.
- SMP-ReVirt: repetición en multiprocesador. Usa protección de página para registrar solo el orden de accesos conflictivos a páginas de memoria.

- Una forma de reducir el *overhead* de registro en multiprocesadores es añadiendo soporte de hardware. La estrategia más común es la de modificar el mecanismo de coherencia de caché para registrar la información necesaria para inferir el orden de accessos de memoria compartida.
- Investigadores también han intentado reducir el *overhead* de registro relajando la definición de repetición determinista. En lugar de requerir que todas las instrucciones retornen los mismos datos que se retornaron en la corrida original, estos enfoques proporcionan garantías ligeramente más débiles las cuales aún soportan los usos propuestos para repetición.
- Repetición determinista basada en búsqueda. En lugar de registrar suficientes datos para repetir rápidamente una ejecución, estos sistemas graban un subconjunto de información, luego usan esa información para guiar la búsqueda de una ejecución equivalente.
- Asegurar que toda la comunicación inter-*thread* sea determinista para una entrada dada. Este enfoque elimina la necesidad de registrar el orden de accesos a memoria compartida.
- DoublePlay aplica ideas de investigaciones en uso de especulación para correr aplicaciones en paralelo.

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

DoublePlay, una nueva forma de garantizar repetición eficiente en multiprocesadores de fácil acceso. La idea clave es que se puede usar los mecanismos simples y eficientes de grabación y repetición de un solo procesador y aún así lograr la escalabilidad ofrecida por múltiples núcleos, usando una ejecución adicional para paralelizar la grabación y repetición de la aplicación. DoublePlay divide múltiples *threads* en segmentos de tiempo en un solo procesador, luego corre múltiples intervalosdel programa concurrentemente en procesadores separados. Esta estrategia, a la que se llama uniparalelismo, hace que el registro sea mucho más fácil porque cada epoch corre en un solo procesador y diferentes epochs operan en diferentes copias de la memoria. Asi, en lugar de registrar el orden de accesos a la memoria compartida, se necesita solo registrar el order en que los *threads* fueron divididos en segmentos de tiempos en el procesador. DoublePlay corre una ejecución adicional del programa en múltiples procesadores para generar puntos de verificación de manera que los epochs corran en paralelo.

5. ¿QUÉ TAN EXITOSA ES ESTA SOLUCIÓN?

Record and replay performance: La disponibilidad de núcleos no utilizados impacta significativamente el *overhead* usado por DoublePlay durante la grabación. Si hay suficientes núcleos sin utilizar, DoublePlay añade poco *overhead* en el momento ejecución de la grabación de una aplicación. En promedio, 2 *worker threads* añaden alrededor de 15 % *overhead*. El *overhead* aumenta gradualmente a 28 % con 4 *threads*. Si todos los núcleos puede usarse productivamente por la aplicación, entonces DoublePlay incurre en un mayor *overhead* debido a que ejecuta la aplicación dos veces durante la grabación, y esto usa los núcleos que pudieron haber sido usados por la aplicación. Cuando se compara con una aplicación configurada para usar el mismo número de núcleos que DoublePlay, DoublePlay añade aproximadamente 100 % de *overhead* a aplicaciones intensivas en CPI que puede escalar a 8 núcleos. Usos de DoublePlay: (1) Aplicaciones que no pueden escalar eficientemente para usar todos los núcleos en una máquina. (2) Sitios que están dispuestos a dedicar núcleos extra para proporcionar repetición determinista. (3) Aplicaciones que comparten datos freuentemente entre núcleos. En comparaciones (*benchmarks*) intensivas en CPU, la repetición *offline* de DoublePlay toma al menos el mismo tiempo que la ejecución de la aplicación en un solo *thread*.

Forward recovery and loose replay: la optimización de *loose replay* reduce la frecuencia de *rollbacks*. La eliminación de *rollbacks* mejora el tiempo de ejecución de 6 % en la ejecución original con 2 *threads* a 13 % con 4 *threads*. Mientras que *forward recovery* y *loose replay* fueron usualmente exitosos en reducir los *rollbacks* y preservar el trabajo hecho durante un epoch fallido, parecen ser más beneficiosos para aplicaciones intensivas en CPU con duraciones más largas epoch.