

Modelo para sintonización dinámica de aplicaciones guiada por eventos de hardware

Fernando Emmanuel Frati^{1,2}, Katalin Olcoz³, Luis Piñuel³, Marcelo Naiouf²
y Armando De Giusti²

¹ Departamento de Ciencias Básicas y Tecnológicas
Universidad Nacional de Chilecito, Argentina

² Instituto de Investigación en Informática LIDI (III-LIDI)
Facultad de Informática, Universidad Nacional de La Plata, Argentina
{ffrati, mnaiouf, degiusti}@lidi.info.unlp.edu.ar

³ Departamento de Arquitectura de Computadores y Automática
Facultad de Informática, Universidad Complutense de Madrid, España
{katzalin, lpinuel}@ucm.es

Resumen El uso habitual de los contadores de hardware consiste en obtener información sobre lo que ocurre dentro del procesador durante la ejecución de un proceso. Sin embargo, los contadores pueden ser configurados para proveer información *parcial* en tiempo real sobre lo que está ocurriendo, generando interrupciones que pueden ser capturadas por la aplicación. Este modo de operación, denominado *muestreo*, puede ser utilizado para alterar el comportamiento de la aplicación en tiempo real con el fin último de conseguir una sintonización de la aplicación. Este trabajo recoge nuestras experiencias en el planteo de una aplicación que utiliza los contadores de este modo, y propone un modelo general que sirve de base para otros proyectos.

Palabras clave: *contadores hardware, muestreo, sintonización dinámica, paralelismo.*

1. Introducción

La multiplicación de recursos y las posibilidades de paralelismo que ello implica se han convertido en la forma en que la industria del hardware consigue aumentar el rendimiento de sus procesadores. Sin embargo, los problemas que empujaron a la industria en esa dirección (como el exceso de consumo, dificultad para disipar el calor debido a elevadas frecuencias de cómputo y la latencia del sistema de memoria entre otros) siguen igual de vigentes. Un diseño de un algoritmo paralelo que no se ajuste a la arquitectura donde será ejecutado sufrirá de penalizaciones elevadas. Como resultado, para conseguir una eficiencia alta es necesario conocer con detalle la arquitectura de la máquina, y tenerla en cuenta durante el diseño del algoritmo.

Una manera de conocer los aspectos específicos de la ejecución de los programas es a través de los contadores de hardware [1][2]. Los contadores hardware

son un conjunto de registros que pueden ser programados para contar el número de veces que ocurre un evento dentro del procesador durante la ejecución de una aplicación. Tales eventos pueden proveer información sobre diferentes aspectos de la ejecución de un programa (por ejemplo el número de instrucciones ejecutadas, la cantidad de fallos cache en L1, cuántas operaciones en punto flotante se ejecutaron, etc). Los procesadores actuales poseen una gran cantidad de eventos (más de 300) y la capacidad de usar hasta 11 registros simultáneamente [3]. El acceso a estos recursos de monitorización se puede llevar a cabo usando diferentes herramientas en función del nivel de abstracción deseado: por ejemplo, linux provee una aplicación de usuario llamada perf [4] que puede ser utilizada para monitorizar eventos generados durante la ejecución de un programa sin necesidad de recompilar el código; PAPI [5] es una API que permite al programador obtener información sobre segmentos específicos de sus programas, otorgando cierto grado de portabilidad a su código. Por último, para conseguir prestaciones más avanzadas, es necesario recurrir a alguna interfaz de bajo nivel que permita el control total de los registros: en el caso de linux es el subsistema de gestión de eventos `perf_events`[6]. Esta interfaz permite explotar el potencial de los contadores a partir de la información disponible en los manuales del procesador en uso. Esta información resulta muy útil al programador en la tarea de encontrar los motivos que penalizan la ejecución y comparar con datos concretos los beneficios de los cambios que realiza [7].

Por otro lado, la información que proporciona la PMU (Unidad de Monitorización de Performance) de los procesadores puede ser utilizada para modificar el comportamiento o tomar decisiones en tiempo de ejecución [8][9]. Esto permite construir algoritmos dinámicos de gran precisión, que se ajustan a los eventos que ocurren en el hardware como por ejemplo algoritmos de planificación o balance de carga. Esta segunda forma de utilizar los contadores es lo que se ha denominado sintonización dinámica de aplicaciones guiada por eventos de hardware.

El aporte de este trabajo consiste en un modelo general de desarrollo que puede ser adaptado a cualquier algoritmo paralelo que requiera alterar su comportamiento en función de la información provista por los contadores de hardware.

El artículo se organiza de la siguiente manera: la Sección 2 sienta las bases para un modelo simple de sintonización dinámica guiado por eventos de hardware. En la Sección 3 se extiende el modelo simple para trabajar con aplicaciones paralelas. La Sección 4 brinda detalles sobre la configuración del entorno experimental y las pruebas realizadas. La Sección 5 ofrece los resultados de los experimentos. Finalmente, la Sección 6 presenta las conclusiones y las líneas de trabajo futuras.

2. Modelo Simple

Los contadores pueden ser configurados en dos modos de operación: 1) *contar eventos* y 2) *muestreo de eventos*. El primer modo de operación es el modo por

defecto y el que utilizan las aplicaciones como perf, PAPI, etc. En este modo cada vez que ocurra un evento del tipo seleccionado, la PMU incrementará el valor del registro correspondiente. Al final el resultado se puede obtener simplemente leyendo el valor del registro. En el modo de operación de muestreo la PMU generará una interrupción cada N eventos, donde N debe ser configurado por el usuario.

Se puede pensar en un modelo simple donde una aplicación cualquiera debe alterar su comportamiento si ocurre un determinado evento de hardware: accesos a memoria, número de fallos de cache, cantidad de operaciones en punto flotante, predicciones de saltos incumplidas, etc. Para conseguirlo es suficiente configurar un contador en modo de muestreo y agregar el código necesario para gestionar la interrupción en la aplicación. El Listado 1 muestra un prototipo para establecer esta configuración.

Listado 1: Configuración y gestión de interrupciones generadas por un contador en modo de muestreo. Se omitió la inclusión de archivos de cabecera, la declaración de variables y el código necesario para gestionar errores en la configuración de los eventos.

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

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

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

Las funciones `create_eventset` y `start` (líneas 5 y 9) son abstracciones para inicializar el contador hardware y comenzar a contar eventos. La mayoría de las librerías que proveen acceso a los contadores poseen funciones similares. La función `sample_event`¹ (línea 6) se utiliza para configurar el contador en modo de muestreo. A esta función hay que indicarle el contador a utilizar (`tEventSet`), un valor en hexadecimal que representa el evento que será utilizado para el muestreo (`event`), la cantidad de eventos que deben ocurrir antes de generar la interrupción (`sample`) y el código de señal que debe enviar la PMU (`SIGUSR1`). Finalmente, la función `signal`² (línea 7) se utiliza para configurar el manejador de interrupciones, indicando que la función `eventListener` será la encargada de gestionar la señal `SIGUSR1`.

¹ Para una explicación detallada sobre cómo configurar un evento en modo de muestreo se puede consultar la página del manual disponible en [6].

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

Aunque el código del Listado 1 permite alterar el comportamiento de la aplicación en función de un evento, no tiene en cuenta los siguientes aspectos:

1. **Aplicaciones paralelas.** Se debe notar que el modelo propuesto genera interrupciones desde un único contador, o dicho de otra forma, desde un único procesador. En una aplicación paralela probablemente sea necesario tener un contador configurado para cada procesador, ya que el evento de la ejecución que se desea detectar/controlar puede ocurrir en cualquiera de los procesadores donde se esté ejecutando un hilo.
2. **Gestión de múltiples interrupciones.** ¿Qué se debería hacer ante múltiples interrupciones generadas por el contador?. Si la cantidad de eventos requeridos para generar la interrupción es pequeña, probablemente ocurran muchas interrupciones seguidas. Tal vez se desee que el comportamiento de la aplicación sea alterado en función de una combinación de eventos.
3. **Rendimiento.** El modelo simple no tiene en cuenta los aspectos relacionados al rendimiento de la aplicación: si el número de interrupciones generadas por el contador es elevado o el código del manejador del evento degrada notablemente su eficiencia, entonces se pierden las supuestas ventajas de una sintonización dinámica de la aplicación.

3. Modelo para aplicaciones paralelas

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

El primer aspecto a considerar es la configuración de los contadores cuando la aplicación posee más de un proceso. Se debe decidir si el contador estará asociado al proceso, al procesador o a una combinación de ambos. La interfaz de linux `perf_events` permite configurar la PMU en alguno de los siguientes modos:

1. Contar eventos en el proceso/hilo que llama a la interfaz, independientemente del CPU donde se ejecute.
2. Contar eventos en el proceso/hilo que llama a la interfaz, sólo cuando se ejecute en un CPU determinado.
3. Contar eventos en un proceso/hilo especificado como un argumento a la interfaz, independientemente del CPU donde se ejecute.
4. Contar eventos en un proceso/hilo especificado como un argumento a la interfaz, sólo cuando se ejecute en un CPU determinado.
5. Contar eventos en cualquier proceso/hilo que se ejecute en un CPU determinado.

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

En segundo lugar, se debe decidir qué hacer ante múltiples interrupciones, ya sean provocadas por el mismo evento o por varios. Por ejemplo en el trabajo que

dió origen a esta investigación se espera activar una costosa rutina de análisis con la ocurrencia del primer evento detectado: por lo tanto, las siguientes interrupciones a la primera deben ser ignoradas. Par ello la primera instrucción de la función `eventListener` debe reemplazar el manejador de interrupciones por la acción `SIG_IGN`. Luego se utilizó otro código de señal y un nuevo manejador de interrupciones para que la rutina de análisis restablezca `eventListener` como manejador de `SIGUSR1`. Sin embargo, la decisión sobre hasta cuándo ignorar las interrupciones se desprende del problema que se esté tratando. Como en el ejemplo, si se deben considerar interrupciones de distintos eventos (o el mismo evento proveniente de distintos procesadores) se debe configurar un segundo manejador de interrupciones y asociarlo a otro código de señal.

De los tres aspectos, probablemente el más importante es evitar degradar el rendimiento del programa con la gestión de interrupciones. Dado que en el modelo propuesto la gestión de la interrupción se hace *en el mismo proceso*, es muy probable que terminará atenuando su rendimiento.

Por este motivo la propuesta divide el modelo simple en dos procesos con `pid` distintos: uno corresponde al *programa objeto de estudio* (POE), que aún debe ser configurado con los contadores en modo de muestreo. El otro, al que se denominó *gestor de interrupciones* (GI), tiene la única función de recibir las interrupciones y unificar las comunicaciones con el POE.

Este enfoque tiene la ventaja de que el GI sólo interrumpe al POE cuando efectivamente es necesario alterar su comportamiento. Asumiendo que el GI puede ser ejecutado en un procesador distinto que el POE, ya sea que el GI ignore las interrupciones o se sature, no afectará el rendimiento del POE.

3.1. Gestor de Interrupciones (GI)

Para el modelo general se pensó en dividir los procesos en dos programas y establecer las comunicaciones entre ellos a través de señales. El programa que representa al POE será ejecutado desde GI, con lo que se aprovecha la relación natural entre los procesos para conocer sus respectivos `pid` necesarios para enviar las señales. Tomando en cuenta los aspectos resaltados anteriormente, se extendió el modelo para recibir como argumento el POE e iniciarlo a través de las funciones del sistema `fork` y `exec`. El Listado 2 muestra un prototipo para esta aplicación.

La función `fork` (línea 6) crea un nuevo proceso duplicando el proceso que la ejecuta y guarda distintos valores en la variable `child` según se trate del proceso original o del duplicado. La variable `child` valdrá 0 para el proceso que ha sido creado, en este caso el POE. La sentencia `if` (línea 7) clasifica el código a ejecutar en función del valor de `child`, y la función `execvp` (línea 8) reemplaza la imagen del proceso actual por la del nuevo proceso. El POE podrá obtener el `pid` del proceso GI a través de una llamada a la función `getppid`.

Por otro lado, el proceso original (en este caso GI) recibe en la variable `child` el `pid` del proceso que representa al POE. Este valor será luego utilizado para enviar interrupciones al POE cuando sea apropiado. Finalmente se debe configurar el manejador de interrupciones (línea 10), indicando que la función

Listado 2: Prototipo del programa Gestor de Interrupciones.

```
1 pid_t child;
2 // codigo para recibir interrupciones del POE
3 void eventListener (int sig){...}

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

`eventListener` será la encargada de gestionar la señal `SIGUSR1`. Para asegurar que el proceso quedará a la espera de las interrupciones, se itera sobre la función `sleep` (líneas 11 y 12). En este caso el programa GI deberá ser terminado manualmente por el usuario. Una versión optimizada podría detectar cuando el proceso POE ha finalizado y terminar el proceso GI automáticamente.

3.2. Diseño del Programa Objeto de Estudio (POE)

Se asumirá que el POE fue desarrollado para el modelo de memoria compartida con la librería `pthread`s. Sin embargo, las ideas aquí expuestas se pueden trasladar a otros modelos sin mayor dificultad. El Listado 3 muestra los fragmentos de código que hay que agregar en la aplicación paralela.

Al igual que en el Listado 1, las funciones `create_eventset` y `start` (líneas 3 y 5) son abstracciones para inicializar el contador hardware y comenzar a contar eventos. La diferencia con el modelo anterior radica en que cada `thread` inicializará un contador distinto a través del arreglo de contadores `tEventSet`: cada hilo accede a una posición distinta del arreglo usando la variable `id` (se asume que un valor distinto para `id` fue utilizado en la creación de cada hilo). La función `sample_event` (línea 4) fue modificada para recibir el `pid` del proceso que creó a POE, es decir, el `pid` del GI. De esta manera las interrupciones generadas por los eventos serán dirigidas a ese proceso. El resto de los argumentos (`tEventSet`, `event`, `sample` y `SIGUSR1`) tienen el mismo significado que en el Listado 1. Finalmente, la función `signal` (línea 13) se utiliza para configurar el manejador de interrupciones, indicando que la función `handle_GI_signal` será la encargada de gestionar la señal `SIGUSR1` proveniente del GI.

Listado 3: Prototipo del programa Objeto de Estudio.

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

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

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

4. Trabajo Experimental

Para evaluar el modelo se utilizó como POE una herramienta de detección de violaciones de atomicidad llamada AVIO[12]. Esta herramienta analiza las interacciones entre hilos de un programa e informa al usuario sobre las violaciones detectadas. El proceso de análisis que realiza para detectar una violación es muy costoso, por lo que se busca reducir el overhead de la herramienta.

De los resultados obtenidos, se verificó que la variación de los tiempos de ejecución sea lo suficientemente pequeña para asegurar que no haya efectos externos que artificialmente incrementen o decrementen los tiempos de ejecución.

Plataforma. El sistema operativo es un Debian wheezy sid x86_64 GNU/Linux con kernel 3.2.0. El compilador empleado fue gcc en la versión 4.7.0. La arquitectura objetivo de la compilación es la x86_64 y los flags de optimización estandar -O2. Los experimentos se realizaron en una máquina con dos procesadores Xeon X5670, cada uno con 6 núcleos con HT cuya microarquitectura es la denominada Westmere.

Diseño del POE. La implementación de AVIO utilizada en este trabajo fue validada en [15], contrastando sus resultados con los citados en el artículo original[12]. La rutina de análisis de AVIO trabaja con complejas estructuras de datos para llevar la historia de todos los accesos a memoria que realiza el programa monitorizado. Por ello se utilizó el modelo propuesto para activar AVIO sólo cuando se detecte compartición de datos. En trabajos previos [9][10][11] se mostró cómo un contador podía ser utilizado con este fin. AVIO fue instrumentado para detectar cuando se está ejecutando una sección de código que no accede a datos compartidos: en este caso se desactiva y envía una señal SIGUSR2 al GI.

El POE se desarrolló con el framework para instrumentación binaria dinámica PIN [13] en la versión 2.11. Se garantiza que cada hilo del programa será ejecutado en un procesador diferente. El acceso a los contadores se realizó a través de la API `perf_event`. Esta API se incluye por defecto en el kernel de linux desde la versión 2.6.32.

Diseño de GI. El GI fue diseñado de acuerdo al modelo expuesto en la Sección 3, con un manejador de interrupciones para capturar eventos provenientes del POE. Al recibir la primer interrupción `SIGUSR1`, el manejador se reemplaza por `SIGIGN` para ignorar el resto y envía una señal de activado al POE. Posee además otro manejador de interrupciones `SIGUSR2` para volver a activar la recepción de interrupciones generadas por eventos.

Benchmarks. Para evaluar el desempeño de la propuesta se utilizó la suite de benchmarks SPLASH-2[14]. Esta suite es probablemente la más utilizada para estudios científicos en máquinas paralelas con memoria compartida. Está compuesta por 14 aplicaciones paralelas según el modelo de *pthread*s. Cada aplicación se configuró para ser ejecutada con 2 hilos y se garantiza que el GI será ejecutado en un procesador distinto.

5. Resultados

La Tabla 1 resume los resultados de los experimentos. Se realizaron mediciones de tiempo para cada aplicación de la suite ejecutada a través de las dos versiones de AVIO. Para los resultados se utilizó el menor de los tiempos entre 10 ejecuciones. Cada fila de la tabla corresponde a una aplicación distinta de la suite de benchmarks. La primera columna (V1) corresponde a los resultados obtenidos utilizando la implementación original de AVIO, mientras que

Cuadro 1: Relación de overhead de cada versión de AVIO. V1 corresponde a la versión original y V2 a la versión guiada por eventos.

aplicación	V1	V2
BARNES	130×	7×
CHOLESKY	26×	2×
FFT	4×	1×
FMM	62×	5×
LU_CB	37×	3×
LU_NCB	2×	1×
OCEAN_CP	38×	9×
OCEAN_NCP	40×	12×
RADIOŠITY	48×	24×
RADIX	6×	1×
RAYTRACE	35×	37×
VOLREND	18×	16×
WATER_NSQUARED	28×	2×
WATER_SPATIAL	24×	2×
Promedios	36×	9×

Cuadro 2: Cantidad de violaciones informadas por cada versión de AVIO. V1 corresponde a la versión original y V2 a la versión guiada por eventos.

package	V1	V2
BARNES	104	140
CHOLESKY	57	18
FFT	9	18
FMM	139	233
LU_CB	11	10
LU_NCB	13	12
OCEAN_CP	61	86
OCEAN_NCP	58	97
RADIOŠITY	250	276
RADIX	21	17
RAYTRACE	32	24
VOLREND	85	100
WATER_NSQUARED	50	27
WATER_SPATIAL	45	43
Total	935	1101

la segunda columna (V2) corresponde a los resultados obtenidos utilizando la versión modificada de AVIO siguiendo los cambios tratados en este artículo.

Cada resultado está expresado como la relación entre el tiempo (en segundos) que demora una aplicación de la suite ejecutada a través de cada versión sobre el tiempo que demora sin ser ejecutada a través de ella. Esta relación (denominada overhead) expresa cuantas veces más tardará en ser ejecutada una aplicación si se utiliza la herramienta de detección de violaciones de atomicidad. Para facilitar la interpretación de los datos, los resultados se redondearon a su parte entera. La última fila de la tabla muestra los promedios de overhead para cada versión. Debe notar que AVIO provocó en promedio un overhead de $36\times$ contra un $9\times$ de la versión que utiliza el modelo propuesto.

Por otro lado la Tabla 2 muestra los resultados en términos de violaciones de atomicidad detectadas. Se utilizó el máximo de violaciones informadas entre 10 ejecuciones. Estos valores son relativos ya que el número de violaciones detectadas por cada versión depende fundamentalmente del no determinismo implícito de cada ejecución. La última fila de la tabla muestra la suma total de violaciones detectadas por cada versión. Se debe notar que en total la versión V2 detectó más violaciones que V1. De las 14 aplicaciones, sólo en dos (cholesky y water_nsquared) V2 detectó menos de la mitad que V1.

6. Conclusiones y Trabajos Futuros

En este trabajo se presentó un modelo para sintonización dinámica de aplicaciones guiado por contadores hardware configurados en modo de muestreo. Se discutieron las consideraciones a tener en cuenta cuando la aplicación a sintonizar se encuentra paralelizada.

Los experimentos se realizaron sobre una herramienta de detección de atomicidad llamada AVIO. Los resultados mostraron que se consiguió reducir el overhead promedio que introduce de $36\times$ a $9\times$ gracias al proceso de sintonización guiado por eventos de hardware, sin que ello repercuta negativamente en su capacidad de detección.

El modelo propuesto puede ser adaptado para cualquier programa que deba alterar su comportamiento en función de algún evento que ocurra en el procesador de la máquina donde se ejecuta. Aunque el modelo planteado está pensado para una aplicación desarrollada en el modelo de memoria compartida, las ideas propuestas pueden extenderse al modelo de memoria distribuida.

El modelo actual depende de las comunicaciones por señales entre procesos, pero no es posible transmitir información junto a la señal enviada. En trabajos futuros se espera aumentar el potencial del modelo extendiéndolo para comunicar datos entre los procesos y desarrollar una capa de software que permita instrumentar al POE sin necesidad de alterar su código. Con respecto a AVIO, se está evaluando su desempeño con una suite de benchmarks más moderna y con aplicaciones reales como apache y mysql.

Referencias

1. B. Sprunt, «The basics of performance-monitoring hardware», IEEE Micro, vol. 22, n.o 4, pp. 64- 71, ago. 2002.
2. D. H. Bailey, R. F. Lucas, y S. W. Williams, Eds., «Performance Tuning of Scientific Applications». USA: CRC Press, 2011.
3. Intel® 64 and IA-32 Architectures Software Developer's Manual, Intel Corporation, Manual 253669-043US, may 2012. Recuperado a partir de <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
4. Tutorial - Perf Wiki. (s. f.). Recuperado 6 de marzo de 2013, a partir de <https://perf.wiki.kernel.org/index.php/Tutorial>.
5. B. D. Garner, S. Browne, J. Dongarra, N. Garner, G. Ho, y P. Mucci, «A Portable Programming Interface for Performance Evaluation on Modern Processors», The International Journal of High Performance Computing Applications, vol. 14, pp. 189-204, 2000.
6. V. Weaver, «Manpage of PERF_EVENT_OPEN», Linux Programmer's Manual, 17-abr-2014. [En línea]. Disponible en: http://web.eece.maine.edu/~vweaver/projects/perf_events/perf_event_open.html. [Accedido: 22-jul-2014].
7. F. G. Tinetti, S. M. Martin, F. E. Frati, y M. Méndez, «Optimization and Parallelization experiences using hardware performance counters», in Proceedings of the 4th International Supercomputing Conference in México, Manzanillo, Colima, México, 2013, p. 5.
8. J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, y T. Austin, «Demand-driven software race detection using hardware performance counters», SIGARCH Comput. Archit. News, vol. 39, n.º 3, pp. 165-176, jun. 2011.
9. F. E. Frati, K. Olcoz Herrero, L. P. Moreno, D. M. Montezanti, M. Naiouf, y A. De Giusti, «Optimización de herramientas de monitoreo de errores de concurrencia a través de contadores de hardware», in Proceedings del XVII Congreso Argentino de Ciencia de la Computación, La Plata, 2011, vol. XVII, p. 10.
10. F. E. Frati, K. Olcoz Herrero, L. Piñuel Moreno, M. R. Naiouf, y A. De Giusti, «Unserializable Interleaving Detection using Hardware Counters», in Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems, Las Vegas, USA, 2012, pp. 230-236.
11. F. E. Frati, K. Olcoz Herrero, L. Piñuel Moreno, M. Naiouf, y A. E. De Giusti, «Detección de interleavings no serializables usando contadores de hardware», presented at the XVIII Congreso Argentino de Ciencias de la Computación, 2012.
12. S. Lu, J. Tucek, F. Qin, and Y. Zhou, «AVIO: detecting atomicity violations via access interleaving invariants», SIGPLAN Not., vol. 41, no. 11, p. 37-48, 2006.
13. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, «Pin: building customized program analysis tools with dynamic instrumentation», in Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 190-200, ACM, 2005.
14. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, «The splash-2 programs: characterization and methodological considerations», ACM SIGARCH Computer Architecture News, vol. 23, pp. 24-36, 1995.
15. F. E. Frati, «Evaluación de técnicas de detección de errores en programas concurrentes», Trabajo Final de Especialista, Facultad de Informática, 2014, Disponible en <http://hdl.handle.net/10915/36923>