Optimización de herramientas de monitoreo de errores de concurrencia a través de contadores de hardware

Fernando Emmanuel Frati[1], Katzalin Olcoz Herrero[2], Luis Piñuel Moreno[2], Diego Miguel Montezanti[1], Marcelo Naiouf[1], Armando De Giusti[1]

[1] Instituto de Investigación en Informática LIDI (III-LIDI),
Facultad de Informática, Universidad Nacional de La Plata, Argentina
{fefrati, dmontezanti, mnaiouf, degiusti}@lidi.info.unlp.edu.ar
[2] Departamento de Arquitectura de Computadores y Automática,
Facultad de Informática, Universidad Complutense de Madrid, España
{katzalin, lpinuel}@dacya.ucm.es

Resumen. Ejecutar eficientemente aplicaciones sobre arquitecturas paralelas requiere desarrollar programas concurrentes. Los errores que pueden aparecer en estos programas son de los más difíciles de detectar, debido a la aleatoridad en el orden de ejecución de los procesos. La mayoría de los trabajos sobre detección de estos errores se basan en la instrumentación del programa, lo que introduce un elevado overhead. Para mejorar la eficiencia del proceso de monitorización, algunos autores proponen extensiones de hardware que realizan la tarea, con lo que consiguen reducir sustancialmente el overhead. Sin embargo, resulta inviable su aplicación en entornos de producción debido a que no se cuenta en la actualidad con esa tecnología. Este trabajo propone una alternativa de optimización a este enfoque a través del uso de los contadores de hardware que sí están disponibles en los procesadores actuales. Se presenta un análisis de la propuesta utilizando un algoritmo de multiplicación de matrices paralelo.

Palabras clave: arquitecturas paralelas, programa concurrente, error de concurrencia, detección de errores, depuración, contadores de hardware.

1 Introducción

La tecnología de hardware ha evolucionado desde computadoras monoprocesador hacia computadoras con múltiples procesadores que trabajan en forma paralela [1-4]. La aparición de los procesadores de múltiples núcleos con capacidad de procesamiento paralelo y una jerarquía compleja de memoria (que incluye zonas privadas y otras parcial o totalmente compartidas) impacta directamente sobre todos los niveles de software, especialmente los lenguajes, compiladores y sistema operativo [5].

Ejecutar eficientemente una aplicación sobre estas arquitecturas requiere el desarrollo de programas concurrentes [6]. De hecho la programación secuencial "pura" resulta ineficiente por el número de núcleos ociosos que implica.
Si el programador comete errores al definir las directivas de sincronización y exclusión mutua en el código de la aplicación concurrente, el compilador difícilmente los detectará. Estos sólo se manifestarán bajo determinadas condiciones de ejecución, donde el orden en que los procesos escriben en memoria eventualmente provocará una condición de carrera, un deadlock o una violación de atomicidad. Si estos errores no se manifiestan durante el período de prueba, el programa llegará a formar parte de los sistemas en producción para los que fue pensado volviéndolos vulnerables [7]. Resulta indispensable contar con herramientas de monitoreo (que ayuden al programador en la tarea de verificar los algoritmos concurrentes) y desarrollar tecnología robusta para tolerar los errores no detectados. En este contexto, la eficiencia de los programas monitorizados se ve comprometida por el overhead que introduce el proceso de monitorización.
El aporte de este trabajo consiste en evidenciar la oportunidad de optimización de las herramientas de detección de errores de concurrencia actuales a través del uso de los contadores de hardware.
El resto del trabajo se organiza de la siguiente forma: en la Sección 2 se presentan y comparan las dos principales estrategias utilizadas para depurar programas concurrentes. La Sección 3 introduce los contadores de hardware. La Sección 4 presenta un caso práctico que muestra su utilidad para obtener información en tiempo de ejecución de la aplicación monitorizada. Finalmente, en la Sección 5 se presentan las conclusiones y se mencionan las lineas de trabajo futuras.

2 Depuración de programas concurrentes

Existen diferentes propuestas destinadas a la detección y/o superación de esta clase de errores. En función del tipo de la técnica que emplean pueden clasificarse en estrategias basadas en reejecución y estrategias basadas en interleaving de procesos.

2.1 Estrategias basadas en reejecución

La estrategia clásica de depuración cíclica de programas asume que éstos pueden reejecutarse tantas veces como sea necesario, y que los errores (en caso de existir), se manifestarán en cada ejecución. Sin embargo, este enfoque no es directamente aplicable a las aplicaciones concurrentes, debido a la naturaleza no determinística de éstas. En efecto, el comportamiento del programa puede variar a lo largo de múltiples ejecuciones, aún para el mismo conjunto de datos de entrada. Por lo tanto, para poder aplicar la depuración cíclica, es necesario que el depurador almacene el orden en que los procesos acceden a la memoria. De esta manera, ante la ocurrencia de un error durante la ejecución, se puede reproducir fielmente el comportamiento del programa cuando éste se manifestó. El ejemplo más destacado que utiliza una estrategia basada

en reejecución es la herramienta RecPlay [8], la cual sólo es capaz de detectar condiciones de carrera.

2.2 Estrategias basadas en interleaving de procesos

Otro enfoque propone un análisis de todos los accesos (lecturas y escrituras) que varios procesos hacen a cada dirección de memoria. La idea surge de la observación de que, ante dos accesos consecutivos de un mismo thread a una variable compartida, solamente existe la posibilidad de error si estos son intervenidos por un acceso de un thread diferente. Cada secuencia de dos accesos locales intervenidos por un acceso remoto se denomina *interleaving*. Dependiendo de como se presente el interleaving se lo clasifica como *serializable* o *no serializable*. Un caso de interleaving serializable es aquel donde su ocurrencia produce el mismo efecto que si no hubiese ocurrido.
Las propuestas de este tipo buscan la ocurrencia de interleavings no serializables, y cuando los detectan emiten una alerta o inician un procedimiento correctivo (inmediatamente después que el error se manifiesta), llegando incluso a determinar cuáles son las instrucciones que provocaron el error.
Esta técnica fue propuesta por Shan Lu para su herramienta AVIO [9]. A diferencia de RecPlay, además de detectar condiciones de carrera contempla también la detección de violaciones de atomicidad.

2.3 Consideraciones de rendimiento

Aunque la transparencia y tolerancia a errores son características deseables, el aumento en el tiempo de ejecución debido a la instrumentación es elevado. Para mejorar este aspecto, se han realizado varias propuestas[10-13] que incluyen extensiones de hardware para implementar estos algoritmos de detección. La Tabla 1 muestra una comparación entre las versiones hardware y software de AVIO.

Benchmark	**Overhead introducido por la herramienta de detección de bugs**	
	AVIO (Hardware)	**AVIO (Software)**
fft	0,5 %	42X
fmm	0,4 %	19X
lu	0,4 %	23X
radix	0,4 %	15X
Promedio	**0,4 %**	**25X**

Tabla 1: Comparación de overhead entre herramientas de detección de bugs sobre la suite de benchmarks SPLASH-2[9]. La X en la tabla representa el tiempo de la aplicación monitorizada sin instrumentación.

En la tabla puede observarse claramente que la versión hardware es superior a la versión software en términos de rendimiento. No obstante deben notarse las siguientes desventajas:

- Requiere hardware dedicado que debe ser incorporado a la arquitectura. Este tipo de soluciones no son viables en las plataformas que se encuentran actualmente en producción.
- Es menos precisa que la versión software. Normalmente las extensiones hardware proveen una granularidad mayor que la versión software (una línea de cache puede contener varias variables), por lo que están expuestas a detectar mayor cantidad de falsos positivos.

Aunque la versión software tiene una mayor precisión en la detección de errores, el overhead que introduce es muy elevado. Esto se debe principalmente al costo que implica instrumentar cada acceso a memoria que realiza la aplicación monitorizada. Sin embargo debe notarse que la posibilidad de error solamente existe si se presenta un caso de interleaving no serializable. Para que el interleaving se lleve a cabo, al menos dos threads deben acceder simultáneamente a datos compartidos. Esto significa que, si de la totalidad de la aplicación que está siendo monitorizada sólo un pequeño porcentaje de las operaciones acceden a datos compartidos, gran parte del tiempo invertido en instrumentar todos los accesos a memoria está siendo desperdiciado.

Por ello se considera que si se contara con información sobre cómo la aplicación comparte los datos, sería posible habilitar o deshabilitar la herramienta de detección de errores en función de qué esté ocurriendo. Afortunadamente los procesadores actuales proveen mecanismos para conocer en tiempo de ejecución qué sucede con la aplicación a través de los contadores de hardware. Recientemente, Joseph L. Greathouse [14] empleó una estrategia similar para mejorar una herramienta de Intel llamada Inspector XE. Sin embargo, esta herramienta sólo se ocupa de la detección de condiciones de carrera. En este artículo se estudia la utilidad de emplear los contadores de hardware, con el objeto de integrarlo a herramientas como AVIO que contemplan otra clase de errores.

3 Contadores de hardware

Todos los procesadores actuales están dotados de un conjunto de registros especiales denominados contadores de hardware [15]. Estos registros o contadores son utilizados para recolectar información sobre los eventos que ocurren durante la ejecución de una aplicación. Pueden consultarse eventos para obtener información sobre el programa, pipeline stalls (retraso causado en la linea de ejecución del procesador), accesos a memoria, predicción de saltos y utilización de recursos. A través del análisis de estos datos, el programador puede comprender mejor el desempeño de sus algoritmos y mejorar su programa.

Estos registros pueden configurarse para contar sólo los eventos causados por la aplicación. El principal inconveniente con estos eventos es que suelen estar pobremente documentados, varían entre diferentes procesadores e incluso pueden ser inaccesibles a nivel de usuario.
PAPI [16] es una API que provee un conjunto de librerías comunes y fáciles de usar para acceder a los contadores de hardware sobre la mayoría de las plataformas. Esta API cuenta con dos interfaces de acceso a esos contadores: una interfaz de alto nivel para obtener mediciones sobre eventos simples y comunes a la mayoría de las arquitecturas (preset-events), y una interfaz de bajo nivel pensada para eventos más complejos y dependientes de la arquitectura (native-events).
Los contadores de hardware pueden usarse para crear un perfil de la aplicación, permitiendo identificar los fragmentos de código que comparten datos de los que no lo hacen. El perfil puede hacerse interrumpiendo la aplicación a intervalos regulares de tiempo (time-based profile) o a intervalos regulares de eventos (event-based profile). Esta información sobre la aplicación permitiría habilitar la herramienta de detección de errores solamente cuando se estén ejecutando fragmentos de código que comparten datos.

4 Trabajo experimental

Los experimentos realizados tienen por objetivo determinar si pueden usarse los contadores de hardware para discriminar regiones de código de la aplicación que comparten datos de las que no. La metodología empleada en este trabajo consiste en los siguientes pasos:

1. Identificar eventos que podrían servir para la tarea de análisis.
2. Monitorizar con cada evento la ejecución de las dos versiones del programa objetivo (secuencial y paralela) y determinar si el evento sirve para detectar código del programa que comparte datos.
3. En caso de tener un evento candidato en el paso anterior, ponderar la fracción del programa objetivo que ejecuta instrucciones que efectivamente comparten datos.

4.1 Aplicación objeto de estudio

Para los experimentos se utilizó una versión optimizada por bloques de un programa de multiplicación de matrices cuadradas. Este programa recibe como parámetros dos valores enteros, que corresponden al tamaño de la matriz y al tamaño de bloque. El programa carga el valor 1 en todos los elementos de las matrices *A* y *B*, realiza la multiplicación en una matriz *C* y verifica el resultado (todos los elementos de *C* deben valer *n*, donde *n* es el tamaño de la matriz).
También se desarrolló una versión paralela de este programa usando OpenMP, configurada para distribuir la tarea entre dos threads. El Algoritmo 1 muestra la fracción del programa paralelo que corresponde a la multiplicación de matrices.

```
void matmulblks (double *a, double *b, double *c, int n, int bs)
{
  int i, j, k;
  initvalmat (c, n, 0.0);
  #pragma omp parallel for private(i, j, k) schedule(static)
                           shared (c, a, b, n, bs) num_threads(2)
  for (i = 0; i < n; i += bs)
    for (j = 0; j < n; j += bs)
      for (k = 0; k < n; k += bs)
        blkmul (&a[i * n + k], &b[k * n + j], &c[i * n + j], n, bs);
}

void blkmul (double *ablk, double *bblk, double *cblk, int n, int bs)
{
  int i, j, k;
  for (i = 0; i < bs; i++)
    for (j = 0; j < bs; j++)
      for (k = 0; k < bs; k++)
        cblk[i * n + j] += ablk[i * n + k] * bblk[j * n + k];
}
```

Algoritmo 1: fragmento de código fuente del programa de multiplicación de matrices paralelizado utilizado durante los experimentos.

4.2 Plataforma de experimentación

La plataforma de pruebas está formada por 4 procesadores AMD Opteron^{tm} 6172 (48 núcleos en total), 47GB de memoria RAM, corriendo un Linux Debian wheezy/sid de 64 bits con kernel 2.6.38-2-amd64. El compilador usado es gcc (Debian 4.5.3-1) 4.5.3 con las opciones de optimización por defecto. Para acceder a los contadores de hardware se utilizó la librería PAPI en la versión 4.1.2.1. La instrumentación de la aplicación objetivo se realizó con PIN 2.8 [17](versión 37287).

4.3 Resultados

Se preseleccionaron tres eventos, tomando como base la guía de desarrolladores para procesadores AMD [18]:

- *Data Cache Refills from L2 or System* (DCRL2): ocurre cada vez que un reemplazo en la cache de datos L1 es satisfecho por la cache L2 o el sistema. Este evento puede configurarse de acuerdo al estado de coherencia de las lineas reemplazadas.
- *Data Cache Refills from System* (DCRS): ocurre cada vez que un reemplazo en la cache de datos L1 es satisfecho por la memoria del sistema o por otra cache diferente a la L2. Este evento también puede configurarse de acuerdo al estado de coherencia de las líneas reemplazadas. Cada ocurrencia refleja una transferencia de 64 bytes.

- *System Read Responses by Coherence* (SRRC): cada ocurrencia representa una línea de 64 bytes transferida del sistema (DRAM o memoria caché de otro núcleo) a la cache de instrucciones, cache de datos, o cache L2. Las respuestas pueden ser compartidas por cualquiera de los que encontraron una línea limpia en otra cache.

Con el objeto de obtener muestras representativas de las mediciones, se promediaron los resultados de 100 ejecuciones del experimento para cada evento y tamaño de matriz. El tamaño seleccionado para las matrices inicia con un valor *n* de 900 y llega hasta 1900. El tamaño de bloque se configuró en 50 para todos los casos. La Figura 1 muestra un gráfico para cada evento, donde las mediciones corresponden a la cuenta de ocurrencias de cada evento. Cada medición incluye la ejecución completa del programa.

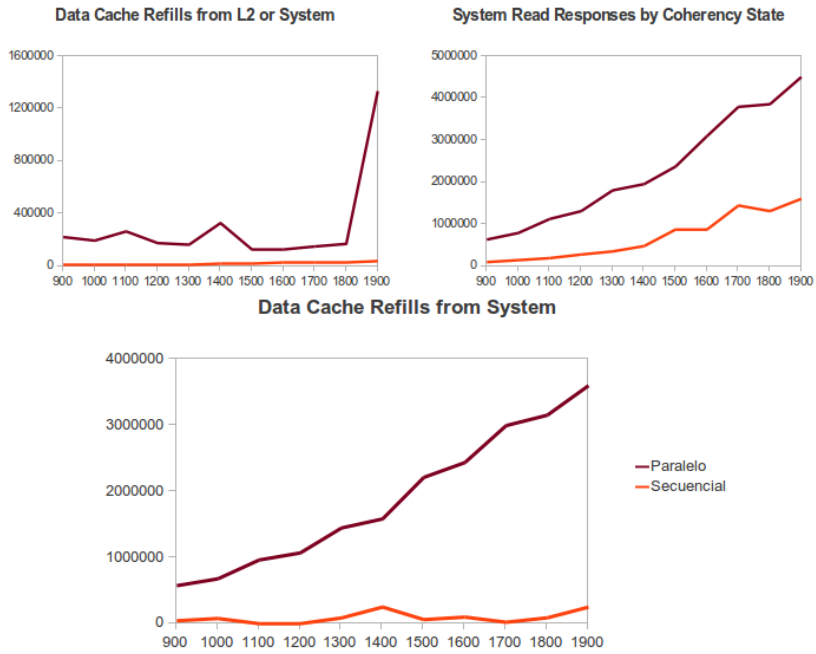

Figura 1: en cada gráfico el eje de ordenadas representa la cuenta del evento y el eje de abscisas representa diferentes tamaños del problema.

Se observa que, para cada uno de los tres eventos, la cantidad de ocurrencias sobre la versión paralela del programa es claramente superior a la cantidad de ocurrencias sobre la versión secuencial. Sin embargo, mientras que la cuenta para la versión paralela del programa objetivo del evento DCRL2 no tiene una relación con respecto al tamaño del problema, los eventos DCRS y SRRC manifiestan un incremento casi lineal con respecto a este valor. Cabe destacar que la cuenta para la versión secuencial del evento SRRC demuestra el mismo comportamiento que para la versión paralela, a diferencia de las mediciones del evento DCRS que tienden al eje de abscisas. Por este

motivo se considera al evento DCRS como el que más adecuadamente refleja la actividad de los threads respecto de compartir datos.
El análisis anterior muestra que puede usarse el evento DCRS para determinar si un programa está compartiendo datos entre sus threads. Sin embargo, se debe determinar si puede obtenerse alguna ventaja al usarlo, ya que si todas las instrucciones del programa deben monitorizarse no tiene sentido introducir la complejidad de medición de eventos de hardware en la herramienta. Con este objetivo se diseñó el siguiente experimento para determinar qué fracción de código comparte datos.
El experimento consistió en desarrollar una herramienta de instrumentación que divide las instrucciones en dos grupos: las que comparten datos y las que no lo hacen. Para ello, cada 100 instrucciones (en promedio) la herramienta lee el valor del contador del evento; si éste ha cambiado desde el último control, acumula su valor en la variable *Comparten*, mientras que si ha permanecido inalterado, las instrucciones se acumulan en la variable *noComparten*, dado que no ha ocurrido tráfico de coherencia entre los threads. En la Figura 2 pueden observarse los resultados de los experimentos. Como es de esperar, el número de instrucciones totales del programa aumenta con respecto al tamaño del problema. No obstante, se debe notar que la fracción de código que comparte datos se encuentra entre el 12% y el 14% en todos los casos.

Figura 2: el eje de ordenadas representa la cantidad de instrucciones y el eje de absisas representa diferentes tamaños del problema.

5 Conclusiones y trabajo futuro

Los resultados muestran que se pueden utilizar los contadores de hardware en tiempo de ejecución para saber qué está ocurriendo con el programa. En particular se observa que en una herramienta de detección de errores en programas concurrentes no tiene sentido monitorizar más allá de una fracción del total de instrucciones del programa, que en el caso de la multiplicación de matrices utilizada de ejemplo en este trabajo no supera el 15%. Aunque este valor depende de la aplicación que se monitorice y de la interacción entre sus threads, queda claro que existe una oportunidad de optimizar las estrategias de detección por software con el fin de acercarlas a los niveles de performance de sus versiones hardware.
En trabajos futuros se prevé integrar la capacidad de detección de tráfico de coherencia con un algoritmo de monitoreo para detección de errores de concurrencia como AVIO. Se espera que al desactivar el módulo de detección en regiones de código que no generan tráfico de coherencia se mejore el rendimiento de estas herramientas.

6 Referencias

1. Burger, T.W.: Intel multi-core processors: Quick reference guide. Intel Corporation (2005).
2. Held, J., Bautista, J., Koehl, S.: From a few cores to many: A tera-scale computing research overview. Intel Corporation (2006).
3. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer. 41, 33–38 (2008).
4. Marowka, A.: Parallel computing on any desktop. Commun. ACM. 50, 74–78 (2007).
5. Pankratius, V., Schaefer, C., Jannesari, A., Tichy, W.F.: Software engineering for multicore systems: an experience report. Proceedings of the 1st international workshop on Multicore software engineering. 53–60 (2008).
6. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing - Second Edition. Pearson Education and Addison Wesley (2003).
7. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. SIGARCH Comput. Archit. News. 36, 329–339 (2008).
8. Ronsse, M., De Bosschere, K.: RecPlay: a fully integrated practical record/replay system. ACM Trans. Comput. Syst. 17, 133–152 (1999).
9. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. SIGPLAN Not. 41, 37–48 (2006).
10. Ceze, L., Tuck, J., Montesinos, P., Torrellas, J.: BulkSC: bulk enforcement of sequential consistency. ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture. pp. 278–289. ACM, New York, NY, USA (2007).

11. Gupta, S., Sultan, F., Cadambi, S., Ivancic, F., Rotteler, M.: Using hardware transactional memory for data race detection. IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing. pp. 1–11. IEEE Computer Society, Washington, DC, USA (2009).
12. Lucia, B., Ceze, L., Strauss, K.: ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. SIGARCH Comput. Archit. News. 38, 222–233 (2010).
13. Nagarajan, V., Gupta, R.: ECMon: exposing cache events for monitoring. ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture. pp. 349–360. ACM, New York, NY, USA (2009).
14. Greathouse, J.L., Ma, Z., Frank, M.I., Peri, R., Austin, T.: Demand-Driven Software Race Detection using Hardware Performance Counters. (2011).
15. Sprunt, B.: The basics of performance-monitoring hardware. IEEE Micro. 22, 64-71 (2002).
16. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. Supercomputing, ACM/IEEE 2000 Conference. p. 42 (2000).
17. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 190–200. ACM, New York, NY, USA (2005).
18. A.M.D.: Kernel developer's guide for AMD Athlon 64 and AMD Opteron processors. Technical Report Pub. 26094, AMD (2006).