Propuesta de Tesis: Tratamiento de Fallos Transitorios en Entornos de Cluster de Multicores

Autor: Diego Montezanti[1]

Directores: Dolores Rexachs[2], Armando De Giusti[1], Marcelo Naiouf[1], Emilio Luque[2]

[1]Instituto de Investigación en Informática LIDI, Facultad de Informática, UNLP
[2]Departamento de Arquitectura de Computadores y Sistemas Operativos, UAB
dmontezanti@lidi.info.unlp.edu.ar
dolores.rexachs@uab.es, degiusti@lidi.info.unlp.edu.ar,
mnaiouf@lidi.info.unlp.edu.ar, emilio.luque@uab.es

1 Resumen

El objetivo de mejorar el rendimiento en las computadoras actuales ha producido el reto de utilizar mayor cantidad de transistores (mayor densidad) y aumentar la frecuencia de operación, además de una disminución en la tensión de alimentación. Todo esto se traduce en un aumento en la temperatura y una mayor cantidad de interferencias, provenientes del entorno, que afectan a los procesadores. Además, con el advenimiento de los *multicores* y los *manycores*, se han integrado varios núcleos de procesamiento en el mismo *chip*. La combinación de todos estos factores tiene como consecuencia que las computadoras sean cada vez menos robustas frente a la ocurrencia de fallos transitorios [1,2,4,5].

El presente trabajo de Tesis se enfoca en el tratamiento de fallos transitorios que ocurren en los registros internos de los *cores* que conforman un procesador actual, en el contexto de un *cluster* de *multicores* en el que se está ejecutando una aplicación científica, de cómputo intensivo. Estos fallos pueden afectar tanto a datos como a instrucciones o direcciones.

El centro de atención está puesto en los fallos silenciosos, aquellos que producen corrupciones de datos que alteran la ejecución del programa, pero sin provocar violaciones detectables a nivel del sistema operativo. La ocurrencia de estos fallos se traduce en la ejecución del programa con parámetros erróneos, de modo que proporciona resultados incorrectos.

En este contexto, el objetivo del trabajo de Tesis es el diseño y desarrollo de un sistema de *middleware* que detecte y tolere los fallos transitorios en un entorno de *cluster* de *multicores*, de manera transparente al usuario, manteniendo un nivel de robustez especificado y optimizando la utilización de recursos en los *multicores* para minimizar la ineficiencia que implica replicar y comparar toda la ejecución.

2 Problema

Dado que la optimización de la ejecución de aplicaciones en los *multicores* es un tema de suma relevancia en la actualidad, es importante el conocimiento sobre el modo en que los fallos transitorios afectan la ejecución sobre estas arquitecturas.

Los fallos pueden clasificarse, de acuerdo a su duración, en permanentes, intermitentes o transitorios. Estos últimos son los que ocurren una vez y no vuelven a repetirse de la misma manera durante la vida útil del sistema [3]. Los fallos transitorios pueden ocurrir en el procesador, la memoria, los buses internos o los periféricos, y normalmente consisten en la inversión del estado de un bit (*bit-flip*) en la ubicación mencionada.

Los errores producidos por los fallos transitorios se denominan *soft errors*. La tasa de *soft errors* (SER, *soft error rate*) de un componente se expresa como la suma de los índices de errores detectados irrecuperables (DUE, *detected unrecoverable error*) y corrupciones silenciosas de datos (SDC, *silent data corruption*) [3]. Los errores que producen SDC son los más peligrosos que pueden presentarse, por lo que serán el objeto del trabajo de investigación propuesto.

En cuanto a la ubicación de los fallos, en este trabajo nos centraremos en los que ocurren dentro de los registros internos de los *cores* que conforman el nodo *multicore*. Este interés especial en los registros radica en que los demás elementos del sistema (memoria, buses y almacenamiento) ya incluyen mecanismos de protección frente a los fallos transitorios [4,13].

La Tolerancia a Fallos (TF) implica tres fases: detección, protección y recuperación. En una primera etapa del trabajo, el foco será la detección. La idea básica para detectar un fallo es duplicar la ejecución de un proceso alojado en un *core* determinado, utilizando para ello otro *core* que funcione como redundancia (DMR – *Dual Modular Redundance*). Ambas réplicas operan sobre los mismos datos de entrada, comparan sus resultados parciales cada cierto intervalo de tiempo y sólo una de ellas escribe en memoria o envía un mensaje a otro proceso [5,6,7,8,9]. Esta técnica implica un alto grado de ineficiencia, ya que, por cada par de *cores*, sólo uno de ellos se encuentra realizando cómputo útil.

El trabajo de Tesis debe conducir a la elaboración de estrategias que permitan reducir esa ineficiencia, buscando que el *core* destinado a detección pueda detectar fallos en más de un *core*. Para ello se propone que el *core* designado ejecute en modo redundante sólo partes significativas de la aplicación, obtenidas automáticamente. En un futuro se contempla la posibilidad de configurar el nivel de tolerancia de acuerdo a las necesidades de cobertura ante fallos o degradación máxima permitida.

En una primera etapa del trabajo se buscará que la aplicación no proporcione resultados incorrectos, de forma que si se detecta un fallo transitorio se produzca una parada segura.

El objetivo final del trabajo es que la aplicación pueda concluir y que sus resultados sean correctos dentro de los rangos de validez que se determinen para cada caso.

Las soluciones propuestas en este camino deberán ser validadas mediante pruebas adecuadas en distintos ambientes de experimentación. La sobrecarga de procesamiento (*overhead*) introducida por ellas deberá ser caracterizado y medido.

3 Trabajos Relacionados

La duplicación de código, con diferentes variantes, ha sido la idea más utilizada en el campo de la detección de fallos transitorios. SRT (*Simultaneous & Redundant Threading*) [5] es una primera aproximación, que consiste en la ejecución simultánea de dos copias de un programa como hilos independientes, haciendo planificación dinámica de los recursos *hardware* entre ellos, y proveyendo detección mediante la replicación de las entradas y la comparación de las salidas. En [6] se propone CRT (*Chip-level Redundant Threading*), la aplicación de esta técnica a entornos de CMP´s, mejorando las prestaciones respecto a hacer *"lockstepping"* de los procesadores. SRTR (*SRT with Recovery*) [7] propone mejoras del mecanismo de detección respecto de SRT y proporciona recuperación mediante reejecución (*rollback*) en el *pipeline*. CRTR (*CRT with Recovery*) [8] mejora la detección separando la ejecución de los hilos para enmascarar la latencia de comunicación entre *cores*, y aplica los mecanismos de recuperación propuestos en [7] en un entorno de CMP´s.

En [9] se propone DDMR (*Dynamic DMR*), una técnica en que los *cores* que ejecutan en modo redundante se asocian dinámicamente para evitar que los *cores* defectuosos afecten la fiabilidad, tratando las asimetrías en el procesamiento y mejorando la escalabilidad. Se introduce la posibilidad de configurar si el sistema opera en modo redundante o si los *cores* se usan independientemente para procesamiento.

En la misma línea, en [4] se propone MMM (*Mixed Mode Multicore*), un modo de proporcionar soporte para que las aplicaciones que requieren fiabilidad (incluyendo el *software* del sistema) se ejecuten en modo redundante y, simultáneamente, las aplicaciones que necesitan alto rendimiento puedan evitar esa penalización. Inclusive, un *core* que está ejecutando una aplicación que requiere prestaciones puede ser utilizado, en un momento dado, para funcionar como redundancia de una aplicación fiable. Se abordan las dificultades propias de los cambios de modo mediante técnicas de virtualización.

En [10] se propone la obtención de una versión reducida de la aplicación quitando cómputo ineficiente y relacionado con control de flujo muy predecible. La aplicación completa y la versión reducida se ejecutan en hilos independientes, proveyendo redundancia y adelantando resultados que aceleran la ejecución de la aplicación. Esta idea puede combinarse con la de seleccionar un *core* para realizar tareas de monitorización sobre los procesos que se ejecutan en los demás, verificando cíclicamente sus estados, como se propone en [11]. Se permite la alternativa de utilizar más de un *core* para operaciones de diagnóstico, y de configurar el nivel de cobertura ante los fallos, así como también el *overhead* permitido mediante un nivel

de agresividad. De esta manera no sería imprescindible la duplicación completa del programa para detección y se reduciría el nivel de ineficiencia introducido por la TF.

4 Marco de Trabajo

El desarrollo de la presente propuesta de Tesis implica el análisis de las consecuencias de los fallos transitorios durante la ejecución de aplicaciones científicas en entornos *multicore*. Las repercusiones de los fallos producidos en diferentes momentos de la ejecución y la magnitud en que los resultados se ven afectados son los factores que nos permitirán elaborar las estrategias de detección más apropiadas, analizando las diversas metodologías de detección desarrolladas en la literatura, sus contextos y condiciones de aplicabilidad y sus influencias en las posibilidades de protección y recuperación. En la siguiente etapa se implementarán dichas estrategias.

El estudio de los diferentes casos de fallos que pueden presentarse debe conducir a la obtención de reglas, de las cuales pueda derivarse cuáles son las partes significativas en la ejecución de aplicaciones científicas. La implementación de dichas reglas debe permitirnos la extracción automática de las secciones del programa que deberán duplicarse con el objetivo de detectar los fallos transitorios, con un grado de tolerancia y *overhead* introducido previamente especificados.

La inyección de fallos es fundamental, tanto en el proceso de análisis de las consecuencias de los fallos como en el de validación de los métodos de detección desarrollados. En general, ha tomado gran importancia al momento de evaluar las propiedades de confiabilidad de un sistema tolerante a fallos [12].

La inyección de fallos, tanto con fines de análisis como de validación de estrategias de detección, se llevará a cabo en dos ambientes de experimentación:

1. Un entorno *multicore* real, consistente en un servidor Blade, conformado por ocho hojas con ocho *cores* cada una. Los *cores* en cada hoja están distribuidos en dos grupos de cuatro *cores*, que comparten las cachés de L2 de a pares. Las hojas están conectadas por medio de una red Gigabit Ethernet.
2. Un entorno *multicore* simulado, utilizando el simulador de máquinas paralelas COTSon [14], desarrollado por HP y AMD, para el cual existe una herramienta de inyección de fallos diseñada en la UAB [12].

Respecto de las aplicaciones de prueba, se utilizará en primer lugar la multiplicación de matrices, por ser una operación bien conocida y que forma parte de la mayoría de las aplicaciones científicas. Luego se realizarán pruebas de los mecanismos desarrollados haciendo uso de los *benchmarks* NAS, los cuales son ampliamente utilizados por la comunidad científica para medir rendimiento de máquinas paralelas, ya que son representativos de las operaciones de cómputo más frecuentemente realizadas.

5 Desafíos

En el presente trabajo se propone hallar una solución que pueda implementarse puramente por *software*, utilizando de ese modo los procesadores *multicore* tal y como se comercializan. En cambio, varios de los trabajos previos [8,9,11,13] proponen soluciones que implican modificaciones al *hardware* de los procesadores.

Como se ha mencionado, el objetivo es utilizar uno o más *cores* designados para replicar partes significativas del cómputo que realizan otros *cores*, lo que implica acotar la TF a algunos tipos de fallos. En este contexto, la mayor dificultad reside en decidir cuáles son estas partes significativas, de acuerdo a la influencia que tendrían en los resultados fallos producidos durante la ejecución.

Otra dificultad consiste en determinar cuándo y cómo se obtendrán las mencionadas partes significativas, tomando en cuenta el *overhead* introducido.

Si la detección se realiza mediante la comparación de los resultados de la ejecución redundante en dos *cores*, se deben resolver varios problemas. Uno de ellos es dónde se realiza la comparación (puede hacerla uno de los dos procesos que se están ejecutando en modo redundante o un proceso distinto que se ejecuta en alguno de los dos *cores*). Otra cuestión consiste, frente a la detección de un fallo, en la necesidad de determinar cuál de los *cores* ha fallado, y en ese caso, cómo realizar esa determinación. Se debe determinar también cuál es el mejor desplazamiento temporal entre el proceso que realiza el cómputo y el que actúa como redundancia.

Un aspecto crítico a resolver es el momento en el que se realizan las comparaciones, considerando fiabilidad y rendimiento. Al respecto, en la literatura se han manejado distintas opciones. [5,7,8,13].

Una dificultad que deberá enfrentarse en la implementación será la posibilidad de configurar la cantidad de *cores* destinados a hacer cómputo redundante cuando las aplicaciones así lo requieran. Por lo tanto, desde el punto de vista del *middleware* desarrollado, un *core* particular podrá, en un momento dado, estar disponible para el procesamiento de la aplicación o realizando tareas de monitorización [11].

6 Próximos Pasos

La etapa de experimentación tendrá como punto de partida la implementación de las soluciones existentes con el fin de utilizarlas como referencias para evaluar las futuras propuestas. Los diferentes mecanismos de tolerancia se valorarán mediante la comparación de los siguientes factores: tipo de implementación, eficiencia de detección, tipo de fallos que detecta, utilización de recursos, grado de transparencia, *overhead* introducido en ausencia de fallos, degradación en presencia de fallos y nivel de tolerancia proporcionado (detección/protección/recuperación).

La primera fase consistirá en la realización de pruebas sobre el *multicore* real. Utilizando un depurador de código, se inyectarán fallos en los registros de los *cores* durante la ejecución de las aplicaciones de prueba, para verificar el alcance de los métodos de detección propuestos (la duplicación del programa, en una primera instancia). Esto permitirá el refinamiento de dichos métodos, en la búsqueda de

ampliar su eficiencia y minimizar el *overhead* introducido por ellos. El objetivo de esta primera experimentación es evitar corrupciones en los datos de salida.

En una segunda etapa, se realizará inyección de fallos en la ejecución de aplicaciones sobre el entorno *multicore* simulado, con objetivos de análisis de repercusiones de fallos y validación de métodos de detección. Además, se instalarán los *benchmarks* NAS, con el objetivo de realizar pruebas de su ejecución en diferentes cantidades de *cores*, tanto utilizando todos ellos para cómputo como en modo redundante. Se realizará un análisis del *overhead* debido a la redundancia, en la búsqueda de optimizar los mecanismos de detección, manteniendo la robustez frente a los fallos, y mejorar la eficiencia en la utilización de los recursos.

En la última etapa, se propondrán e implementarán estrategias adecuadas de recuperación ante los fallos. Finalmente se realizarán pruebas del sistema tolerante a fallos con el fin de validarlo.

7 Referencias

1. Mukherjee, S. S., Emer, J., & Reinhardt, S. K. (2005). The Soft Error Problem: An Architectural Perspective. HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, 243-247.
2. Wang, N. J., Quek, J., Rafacz, T. M., & Patel, S. J. (2004). Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks, 61.
3. Mukherjee, S. S. (2008). Architecture Design for Soft Errors. Morgan Kaufmann.
4. Wells, P. M., Chacraborty K. & Sohi G. S. (2009). Mixed-Mode Multicore Reliability. ASPLOS 2009. SESSION: Reliable systems II, 169-180.
5. Reinhardt, S. K. & Mukherjee S. S. (2000). Transient Fault Detection via Simultaneous Multithreading. Proceedings of the 27th annual International Symposium on Computer Architecture, Vancouver, British Columbia, Canada, 25-36.
6. Kontz M., Reinhardt S. K. & Mukherjee S. S. (2002). Detailed Design and Evaluation of Redundant Multithreading Alternatives. 29th Annual International Symposium on Computer Architecture (ISCA'02).
7. Vijaykumar T. N., Pomeranz, I. & Cheng, K. (2002). Transient-Fault Recovery using Simultaneous Multithreading. Proceedings of the 29th Annual International Symposium on Computer Architecture, Anchorage, Alaska. Session 3: Safety and Reliability, 87-98.
8. Gomaa M., Scarbrough C., Vijaykumar T. N & Pomeranz, I. (2003). Transient-Fault Recovery for Chip Multiprocessors. Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA ´03).
9. Golander A., Weiss S. & Ronen R. (2009). Synchronizing Redundant Cores in a Dynamic DMR Multicore Architecture. IEEE Transactions on Circuits and Systems II: Express Briefs Volume 56, Issue 6, 474-478.
10. Sundaramoorthy K., Purser Z. & Rotenberg E. (2000). Slipstream Processor: Improving both Performance and Fault-tolerance. ACM SIGPLAN Notices Volume 35, Issue 11, 257- 268.
11. Barr A. H., Pomaranski K. G. & Shidla D. J. (2005) United States Patent Application Publication US 2005/0102565 A1: Fault Tolerant Multicore Microprocessing.

12. Gramacho, J. (2009). Analyzing the effects of transient faults into applications. Universitat Autònoma de Barcelona. Departament d'Arquitectura de Computadors i Sistemes Operatius. (www.recercat.net/handle/2072/41798)
13. Ray J., Hoe J.C. & Falsafi, B. (2001). Dual Use of Superscalar Datapath for Transient-fault Detection and Recovery. Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture Austin, Texas. SESSION: Superscalar architectures, 214-224.
14. Argollo, E., Falcon, A., Faraboschi, P., & Ortega, D. (2008). COTSon: infrastructure for system-level simulation of clustered multicores.