Una Metodología de Detección de Fallos Transitorios en Aplicaciones Paralelas sobre Cluster de Multicores

Diego Montezanti

Facultad de Informática

Universidad Nacional de La Plata

Director: Ing. Armando E. De Giusti

Co-director: Dr. Ricardo M. Naiouf

Trabajo Final Integrador presentado para obtener el grado de

Especialista en Cómputo de Altas Prestaciones y Tecnologías Grid

Octubre 2014

Objetivo

El objetivo general de este trabajo consiste en presentar el análisis de una metodología que permite detectar fallos transitorios que pueden producirse en componentes de hardware de las arquitecturas multicore actuales, y que afectan especialmente la ejecución de aplicaciones paralelas de cómputo intensivo.

La creciente vulnerabilidad a los fallos transitorios se ha vuelto crítica, a causa de la capacidad de estos fallos de corromper los resultados de las aplicaciones. En particular, el impacto de los fallos transitorios es más notorio en el ámbito de HPC, donde típicamente un cluster de multicores ejecuta aplicaciones paralelas de cómputo intensivo y de gran duración. El alto costo de relanzar una ejecución desde el comienzo, en caso de que un fallo transitorio produzca la finalización de la aplicación con resultados incorrectos, provoca la necesidad de contar con estrategias para mejorar la robustez de estos sistemas. En particular, resulta relevante poder detectar la ocurrencia de fallos que no son interceptados por el sistema operativo y, por lo tanto, no producen la finalización abrupta de la ejecución.

En este contexto, los temas a abordar en este trabajo comprenden el análisis del problema y de las distintas soluciones que se han propuesto hasta el momento, con sus ámbitos específicos de aplicación y sus limitaciones. Partiendo de esta base, se analizará detalladamente una metodología particular, llamada SMCV [1,2], que es capaz de proveer detección de fallos transitorios que producen ejecuciones incorrectas. Está diseñada específicamente para aplicaciones científicas paralelas de alto coste de ejecución, y utiliza técnicas de software en la búsqueda de aprovechar la redundancia intrínseca de recursos hardware de las arquitecturas multicore.

Resumen

El aumento en la escala de integración, con el objetivo de mejorar las prestaciones en los procesadores actuales, sumado al crecimiento de los sistemas de cómputo, han producido que la fiabilidad se haya vuelto un aspecto relevante. En particular, la creciente vulnerabilidad a los fallos transitorios se ha vuelto crítica, a causa de la capacidad de estos fallos de corromper los resultados de las aplicaciones.

Históricamente, los fallos transitorios han sido una preocupación en el diseño de sistemas críticos, como sistemas de vuelo o servidores de alta disponibilidad, en los que las consecuencias del fallo pueden resultar desastrosas. Pese a ser fallos temporarios, tienen la capacidad de alterar el comportamiento del sistema de cómputo. A partir del año 2000 se han vuelto más frecuentes los reportes de desperfectos significativos en distintas supercomputadoras, debidos a los fallos transitorios.

El impacto de los fallos transitorios se vuelve más relevante en el contexto del Cómputo de Altas Prestaciones (HPC). Aun cuando el tiempo medio entre fallos (MTBF) es del orden de 2 años para un procesador comercial, en el caso de una supercomputadora con cientos o miles de procesadores que cooperan para resolver una tarea, el MTBF disminuye cuanto mayor es la cantidad de procesadores. Esta situación se agrava con el advenimiento de los procesadores multicore y las arquitecturas de cluster de multicores, que incorporan un alto grado de paralelismo a nivel de hardware. La incidencia de los fallos transitorios es aún mayor en el caso de aplicaciones de gran duración, que manejan elevados volúmenes de datos, dado el alto costo (en términos de tiempo y utilización de recursos) que implica volver a lanzar la ejecución desde el comienzo, en caso de obtener resultados incorrectos debido a la ocurrencia del fallo.

Estos factores justifican la necesidad de desarrollar estrategias específicas para mejorar la confiabilidad en sistemas de HPC; en este sentido, es crucial poder detectar los fallos llamados silenciosos, que alteran los resultados de las aplicaciones pero que no son interceptados por el sistema operativo ni ninguna otra capa de software del sistema, por lo que no causan la finalización abrupta de la ejecución.

En este contexto, el trabajo analizará una metodología distribuida basada en software, diseñada para aplicaciones paralelas científicas que utilizan paso de mensajes, capaz de detectar fallos transitorios mediante la validación de contenidos de los mensajes que se van a enviar a otro proceso de la aplicación. Esta metodología, previamente publicada [1,2], intenta abordar un problema no cubierto por las propuestas existentes, detectando los fallos transitorios que permiten la continuidad de la

ejecución pero que son capaces de corromper los resultados finales, mejorando la confiabilidad del sistema y disminuyendo el tiempo luego del cual se puede relanzar la aplicación, lo cual es especialmente útil en ejecuciones prolongadas.

Publicaciones relacionadas a este trabajo

- **"SMCV: a Methodology for Detecting Transient Faults in Multicore Clusters"**
  Autores: Diego Montezanti, Fernando Emmanuel Frati, Dolores Rexachs, Emilio Luque, Marcelo Naiouf, Armando De Giusti
  Evento: V HPCLatam 2012 (presentado como *full paper*)
  Lugar de realización: Ciudad Autónoma de Buenos Aires, Argentina
  Fecha de realización: 23 y 24 de Julio de 2012.
  Publicado: CLEI Electronic Journal, Volume 15, Number 3 (December 2012), Paper 5. ISSN 0717- 5000

- **"A tool for detecting transient faults in execution of parallel scientific applications on multicore clusters"**
  Autores: Diego Montezanti, Enzo Rucci, Dolores Rexachs, Emilio Luque, Marcelo Naiouf, Armando De Giusti
  Publicado: Journal of Computer Science & Technology (JCS&T), Volumen 14, Número 1 (Abril 2014). Páginas: 32 a 38. ISSN: 1666-6038.

- **"Propuesta de Tesis: Tratamiento de Fallos Transitorios en Entornos de Cluster de Multicores"**
  Autores: Diego M. Montezanti
  Directores: Ing. Armando De Giusti (UNLP), Dr. Marcelo Naiouf (UNLP), Dra. Dolores Rexachs (UAB), Dr. Emilio Luque (UAB)
  Evento: XVI CACIC 2010 (Congreso de Ciencias de la Computación).
  Enviado al I Coloquio de Doctorandos – XIII Encuentro de Tesistas de Postgrado.
  Publicado: en los proceedings del Congreso. ISBN: 978-950-9474-49-9. CD-Rom. Páginas 1046 a 1052.
  Lugar de realización: Morón, Buenos Aires, Argentina
  Fecha de realización: 18 al 22 de Octubre de 2010.

Índice general

Capítulo 1

Fallos transitorios

Introducción

La tolerancia a fallos (TF) se ha convertido en un requerimiento importante en el ámbito del cómputo de altas prestaciones (HPC), debido Estos componentes, trabajando cerca de sus límites tecnológicos, son cada vez más propensos a la ocurrencia de fallos. ve Si las aplicaciones son de relevancia significativa, es necesario ejecutarlas sobre un sistema que cuente tanto con una alta disponibilidad (es decir, que se mantienen funcionando un alto porcentaje de tiempo) como con una alta fiabilidad (es decir, cuyo comportamiento no se aparta del especificado, proporcionando, por lo tanto, resultados correctos [3]).

Para obtener un sistema paralelo altamente disponible y fiable se deben aplicar estrategias de tolerancia a fallos, ya que son capaces de proveer detección de fallos, protección y recuperación frente a ellos. La tolerancia a fallos se puede implementar en tres niveles diferentes: a nivel de hardware, a nivel arquitectural o a nivel de aplicación o software del sistema. La técnica particular que se analizará en este trabajo y otras con las cuales se comparará trabajan a nivel de software.

Por otra parte, la detección y la protección/recuperación de los fallos pueden verse como problemas relativamente independientes, ya que una vez que el fallo se detecta, la recuperación puede realizarse de una variedad de formas diferentes; la estrategia de recuperación se ejecuta sólo en caso de la ocurrencia del fallo, en tanto que la detección se encuentra activa continuamente. En el marco de este trabajo, el foco se hará en la detección, aprovechando la característica mencionada [4].

Dependiendo de su duración, los fallos pueden clasificarse en permanentes (que permanecen hasta que son reparados), intermitentes (que aparecen y desaparecen, bajo una combinación específica de circunstancias en el sistema; nuevamente, siguen apareciendo hasta que se tome alguna acción correctiva) o transitorios (que aparecen aleatoriamente y desaparecen solos al cabo de un tiempo) [5]. En este del trabajo, se tratarán únicamente los fallos transitorios.

El resto del capítulo tiene por objetivo definir y analizar los fallos transitorios, las métricas utilizadas en su estudio y sus consecuencias, en especial sobre sistemas paralelos en los que se ejecutan aplicaciones científicas intensivas en cómputo.

Concepto. Causas de ocurrencia En general, se define un fallo como una imperfección o defecto físico que afecta a cualquier componente de hardware o software del sistema. Contrariamente a los fallos llamados permanentes, los fallos transitorios no se reflejan en una disfuncionalidad permanente del sistema, ni ocurren de una manera consistente (como los fallos de diseño o fabricación). Un fallo permanente es capaz de causar errores o comportamientos inesperados cada vez que el componente afectado es utilizado, y permanece en el sistema hasta que dicho componente es reparado o reemplazado [6].

En cambio, un fallo transitorio ocurre una vez y de una manera única, no volviendo a repetirse de la misma manera durante el resto de la vida útil del sistema. Son fallos temporales y de corta duración, por lo que no son capaces de afectar la operación regular del sistema ni causar daño físico permanente a ningún componente. Sin embargo, dependiendo de su ubicación específica y su momento de ocurrencia, pueden corromper el cómputo, resultando en modificaciones en el flujo de control de un programa o en datos alterados que pueden propagarse, causando la ejecución incorrecta de una aplicación [4,7]. De hecho, han causado averías costosas en sistemas de altas prestaciones en los últimos años [8,9].

Un fallo transitorio es consecuencia de alguna forma de interferencia, ya sea interna al sistema o externa (proveniente del entorno) que afecta a uno o varios componentes de hardware de un sistema de cómputo.

La principal causa externa de fallos transitorios es la radiación electromagnética. Ésta ocurre cuando partículas de alta energía inciden sobre zonas sensibles de los circuitos digitales, causando pulsos de tensión que interfieren con el funcionamiento normal. Un ejemplo son los neutrones de la atmósfera o las partículas alfa, que al incidir sobre dispositivos semiconductores, generan pares electrón-hueco, situación que se esquematiza en la figura 1.1). Las fuentes de los transistores y los nodos de difusión pueden almacenar esas cargas, acumulando la cantidad suficiente para invertir el estado de un dispositivo lógico, inyectando de esa forma un fallo en la operación del circuito (por ejemplo, invirtiendo un bit de una posición de memoria o de un registro del procesador) [10].

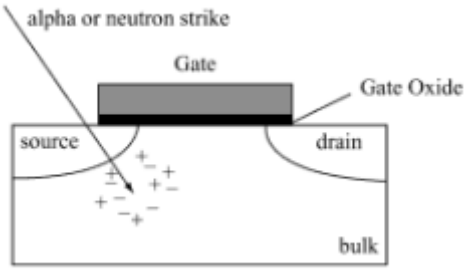

Figura 1.1: Incidencia de partículas cargadas sobre dispositivos semiconductores

Entre las principales causas internas al sistema se encuentran las variaciones en la tensión de alimentación y el sobrecalentamiento de los circuitos integrados (chips) que componen el computador [11].

Físicamente, la ubicación donde se produce el fallo puede ser cualquiera dentro del sistema; de esa forma, pueden ocurrir dentro del procesador, en el subsistema de memoria, los buses internos o los dispositivos de Entrada/Salida. Normalmente, se manifiestan como la inversión temporal de uno o varios bits (*single bit-flip* o *multiple bit-flip*) del componente donde se produce el fallo.

Históricamente, los fallos transitorios comenzaron siendo un problema para los diseñadores de sistemas de alta disponibilidad, sistemas críticos (como pueden ser computadores a bordo de sistemas de vuelo) o que operan en ambientes hostiles desde el punto de vista de la radiación electromagnética (ej: el espacio exterior), donde las consecuencias de un fallo pueden ser desastrosas [11,12,13].

Sin embargo, la situación ha cambiado. A medida que aumenta la escala de integración y progresa el proceso de miniaturización de los componentes, disminuyen los umbrales de ruido en la tensión de alimentación y crece la frecuencia de operación, lo que trae aparejado un incremento en la temperatura interna de los chips. Todos estos factores tienen como consecuencia que los componentes sean cada vez más vulnerables a los fallos transitorios [5,14]. Por la creciente

influencia de la radiación terrestre, en varios componentes de los sistemas de cómputo se han implementado técnicas extensivas de detección y/o corrección de errores. Por ejemplo, los buses internos, las memorias y los dispositivos de almacenamiento tienen mecanismos incorporados basados en redundancia de hardware, como los códigos correctores de errores (ECC´s – *Error Correcting Codes*) o los bits de paridad [13].

A pesar de esto, para escalas de miniaturización inferiores a 65 nanómetros, la protección de la memoria únicamente es insuficiente. La necesidad de protección contra los efectos de los fallos transitorios, tanto en el cómputo como en las comunicaciones, ha motivado la aparición de nuevos mecanismos integrados de protección de *latches* y *flip-flops*. Con el tiempo, incluso será necesaria la protección de parte de la lógica combinacional dentro del procesador, a medida que la tecnología permita la inclusión de cantidades cada vez mayores de transistores dentro de los chips [13].

De esto se desprende que los fallos que afectan a los registros y a la lógica del procesador son los más críticos, ya que son capaces de producir problemas de fiabilidad en la operación del computador. Como consecuencia, incluso los procesadores de propósito general (especialmente los que forman parte de sistemas de alta disponibilidad) deben recurrir a técnicas de tolerancia a fallos en la búsqueda de asegurar la correcta operación [2].

Con el advenimiento de los chips multicore y *manycore*, la cantidad de transistores y buses dentro del procesador se ha vuelto tan grande que se espera que los niveles superiores del sistema de cómputo (como el sistema operativo o incluso la aplicación) deban tratar con los fallos transitorios cada vez más frecuentemente.

1.1 Efectos de los fallos transitorios. Terminología

Un fallo determinado es capaz de producir uno o más errores latentes. Un error es la manifestación de un fallo en el sistema. Un error latente se hace efectivo cuando el recurso en el que ocurre el error es utilizado para realizar algún cómputo. El error efectivo puede propagarse de un componente a otro, causando nuevos errores. Dicho de otra forma, el fallo es una imperfección a nivel físico, mientras que un error es un estado interno inconsistente del sistema, a nivel de información [10].

Así como un error ocurre como consecuencia de un fallo, puede a su vez ser causa de una avería. Una avería es la manifestación de un error en el servicio provisto por el sistema, observable a nivel externo. La avería ocurre cuando el sistema se comporta de un modo diferente del especificado. En general, cualquier estrategia de tolerancia a fallos tiene por objetivo, no impedir completamente los fallos (esto es imposible), sino evitar que éstos lleguen a transformarse en averías.

A continuación, se presenta un ejemplo que ilustra estos conceptos, que se encuentra esquematizado en la figura 1.2:

1. Si una partícula cargada de energía incide sobre una celda de una memoria DRAM puede producir un fallo.
2. Una vez que este fallo altera el estado de la celda de memoria se genera un error latente.
3. El error permanece latente hasta que la posición de memoria afectada es leída por algún proceso, transformándose en un error efectivo.
4. La avería ocurre si la lectura de la posición de memoria alterada por el error afecta la operación del sistema modificando su comportamiento.

Figura 1.2: Propagación de un fallo que resulta en una avería

Los errores producidos por los fallos transitorios son llamados *soft errors*[1]. El hecho de observar un *soft error* no implica que el sistema sea menos fiable que antes de su observación. Los *soft errors* alteran datos pero no modifican físicamente al circuito afectado. Si los datos vuelven a escribirse, el circuito funcionará perfectamente otra vez.

Cabe aclarar que la expresión *soft error*, utilizada en el contexto de fallos transitorios, no debe confundirse con errores en la programación de las aplicaciones o de cualquier nivel del software (*software errors*).

1.2 Métricas utilizadas

Actualmente, en los trabajos sobre fallos transitorios y *soft errors* se utilizan métricas comúnmente usadas en tolerancia a fallos, e incorporan algunas otras que facilitan la estimación de la probabilidad de fallos de un sistema.

El tiempo medio hasta la ocurrencia de una avería (MTTF – *Mean Time To Failure*) expresa, en promedio, el tiempo transcurrido entre el último arranque (o reinicio) del sistema hasta el próximo error del componente, como se ve en la figura 1.3. El MTTF de un componente se expresa normalmente en años y se obtiene en base a un promedio estimativo de predicción de fallos realizado por el proveedor del componente.

El MTTF de un sistema completo (conjunto de componentes) se puede obtener combinando los MTTF de todos sus componentes, como se muestra en la ecuación 1 [10].

$$MTTF_{system} = \frac{1}{\sum_{i=0}^{n} \frac{1}{MTTF_i}}$$

Ecuación 1: MTTF de un sistema

La utilización del término averías en el tiempo (FIT – *Failures in Time*) es más útil, a causa de su propiedad aditiva. Un FIT representa un error en un billón de horas. Para evaluar el FIT de un sistema, sólo es necesario sumar los FIT de todos sus componentes, como se muestra en la ecuación 2 [10]. La estimación del valor FIT debido a los fallos transitorios recibe la denominación SER (*Soft error Rate*).

$$FIT\ rate_{system} = \sum_{i=0}^{n} FIT\ rate_i$$

Ecuación 2: FIT de un sistema

[1] En algunas publicaciones [4,7], los términos *transient fault* y *soft error* está dados como sinónimos. En este trabajo se intenta utilizar rigurosamente la terminología, según la cual el fallo es el problema a nivel físico, que, dependiendo de la forma de utilización del recurso afectado, puede o no manifestarse en la forma de *soft error*.

La tasa FIT y el MTTF de un componente son inversamente proporcionales bajo ciertas condiciones, como se muestra en la ecuación 3:

$$MTTF \text{ (years)} = \frac{10^9}{FIT\ rate \text{ x } 24\ hs \text{ x } 365\ días}$$

Ecuación 3: Relación entre FIT y MTTF

Existen dos métricas adicionales que son utilizadas frecuentemente en el área de tolerancia a fallos: el tiempo medio de reparación (MTTR – *Mean Time To Repair*) y tiempo medio entre fallos (MTBF – *Mean Time Between Failures*). MTTR es el tiempo requerido para reparar un error una vez detectado. MTBF representa el tiempo promedio entre las ocurrencias de dos errores [10]. MTBF se puede expresar como MTBF = MTTF + MTTR, como se muestra en la figura 1.3 (adaptada de [10]).

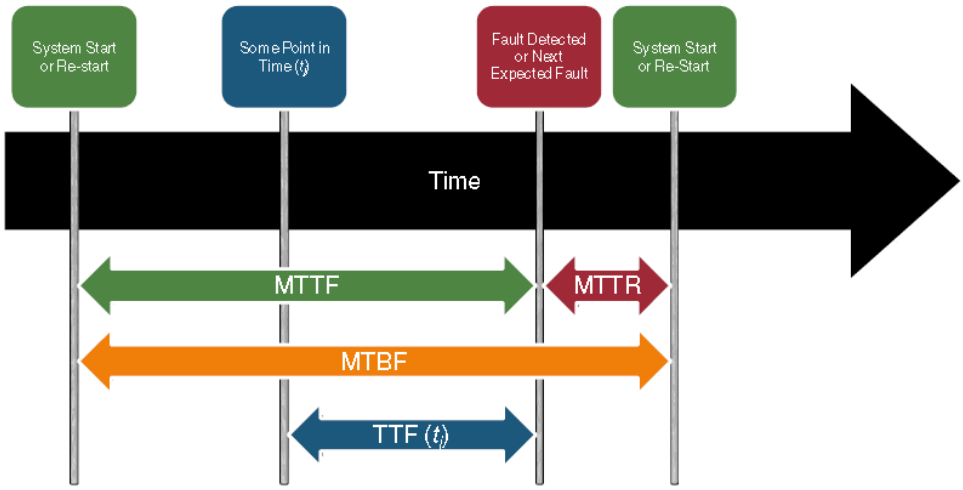

Figura 1.3: Relación entre las métricas de tolerancia a fallos

1.3 Algunos casos reales

Como se mencionó anteriormente, los fallos transitorios comenzaron siendo un problema para los diseñadores de sistemas críticos o que operan en ambientes hostiles desde el punto de vista de la radiación electromagnética (cuya intensidad se incrementa mucho en la altura), donde las consecuencias de un fallo pueden ser considerables. A modo de ejemplo, el satélite meteorológico chino FengYun 1(B), lanzado en 1990, quedó fuera de servicio antes de lo previsto, a causa de que el sistema de control de altitud perdió el control debido a fallos transitorios [12]. Hacia finales de los años `90, Boeing diseñó su aeronave 777 con tres procesadores y buses de datos diferentes, utilizando un esquema de votación por mayoría para lograr tanto detección como recuperación de fallos [15,16]. Más recientemente, en marzo del 2013, el *rover* marciano *Curiosity* sufrió la incidencia de la rayos cósmicos en su sistema de memoria y debió activar su computador de respaldo para continuar su operación en modo seguro (en el cual sólo se realizan actividades mínimas de mantenimiento y control, pero permanece interrumpida la operación de los instrumentos científicos de a bordo), retrasando por lo tanto la misión.

Sin embargo, hoy en día, los fallos transitorios constituyen una considerable fuente de errores no sólo para circuitos que se desempeñan en el espacio exterior, sino también para equipamiento que opera dentro de la atmósfera a altitudes elevadas e incluso a nivel del suelo.

En general, no hay demasiadas publicaciones que reporten la ocurrencia de *soft errors*. Las primeras evidencias provienen de la contaminación de la producción de circuitos integrados a fines de los años `70 y en los `80. A partir del año 2000, los reportes de *soft errors* en supercomputadoras o granjas de servidores se han vuelto más frecuentes, y han causado averías costosas durante los últimos años [8,9]. Esto ocurre porque la cantidad de componentes en estas instalaciones es muy grande (con decenas de miles de procesadores y de terabytes de memoria), y porque los procesadores multicore y *manycore* tienen un alto grado de miniaturización, por lo que el sistema es, potencialmente, más vulnerable a los fallos transitorios [3].

La susceptibilidad de un sistema a los fallos transitorios es normalmente impredecible durante el proceso de diseño y fabricación. Por ejemplo, durante la puesta en producción del supercomputador ASC Q (2003), los científicos del Laboratorio Nacional de Los Alamos documentaron una alta proporción de averías causadas por fallos transitorios [9,13]. En el 2000, Sun Microsystems admitió que rayos cósmicos habían interferido con la operación de memorias caché, causando serios inconvenientes en servidores de sitios de sus principales clientes, como America Online, eBay y muchos otros [8]. En la figura 1.4 se esquematiza la evidencia de *soft errors* en sistemas reales.

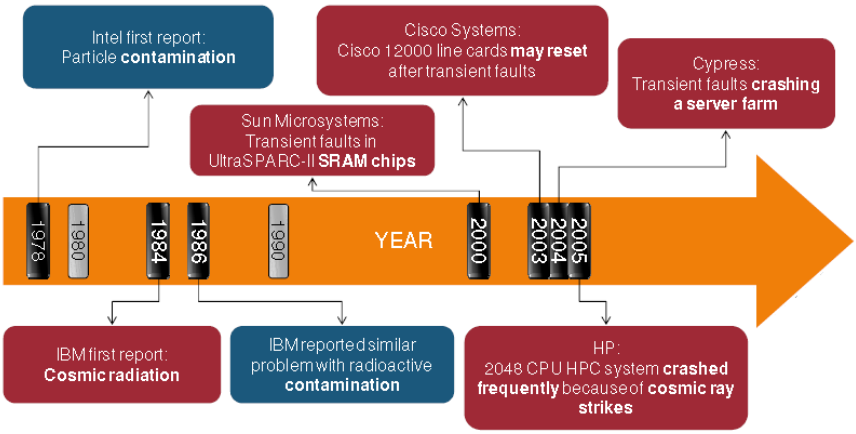

Figura 1.4: Evidencia de *soft errors* en sistemas reales

Desafortunadamente, las tendencias actuales de diseño del hardware sugieren que las tasas de fallos irán en aumento. A esto contribuyen las altas frecuencias de operación, densidades de transistores en los chips y temperaturas internas, así como las pequeñas tensiones de alimentación y sus consecuentes bajos umbrales de ruido [5,7,14]. Debido a una combinación de estos factores, la tasa de *soft errors* (SER) se viene incrementando aproximadamente un 8% en cada generación de procesadores modernos [17]. En particular, los *latches*, que son circuitos potencialmente vulnerables a los fallos transitorios, son utilizados en varias estructuras lógicas y de datos internas, constituyendo una fracción significativa del área del procesador [18].

Los sistemas de alta disponibilidad requieren mucha más redundancia de hardware que la que proveen los ECC´s y los bits de paridad. Por ejemplo, históricamente, IBM ha añadido 20-30% de lógica adicional en sus procesadores para mainframes. En el diseño del S/390 G5, IBM incorporó aún más redundancia, replicando completamente las unidades de ejecución del procesador para evitar varios problemas que tenía su enfoque anterior de tolerancia a fallos. Para mi-

nimizar el efecto de los fallos transitorios, en 2003 Fujitsu lanzó la quinta generación de procesadores SPARC64 con el 80% de sus 200.000 *latches* con alguna forma de protección, incluyendo la generación de paridad en la ALU y una verificación de residuos en las operaciones de multiplicación y división [4].

Además, los *soft errors* son críticos en la operación de sistemas de gran escala, en los cientos de miles de procesadores trabajan juntos para resolver un problema. En el año 2008, el supercomputador BlueGene/L contaba con 128.000 nodos, y experimentaba un *soft error* en su caché L1 cada 4-6 horas, debido a la desintegración radiactiva en las soldaduras de plomo. El supercomputador ASCI Q tenía una tasa de fallos de CPU inducidos por radiación de 26.1 por semana, mientras que para un computador de un tamaño similar, el Cray XD1, se estimaban 109 fallos semanales entre CPUs, memorias y FPGAs [18].

En los últimos años, una serie de estudios [19] han permitido llegar a la conclusión de que el MTBF depende principalmente de la cantidad de procesadores, resultando inversamente proporcional al tamaño del sistema. Para dar una idea del ritmo de crecimiento de los sistemas actuales, en junio del 2010 la cantidad de procesadores (en promedio) de las computadoras del Top 500 era de 10.267 [20], mientras que para junio de 2013 esa cifra había aumentado a 38.700. En 2010, sólo 42 computadoras de la lista tenían más de 16.000 cores (8.4% del total), mientras que, en 2013, ya 274 (54.8 %) sistemas superaban esa cantidad. Se proyecta de los sistemas de exa-escala contengan del orden de decenas o centenares de millones de cores dentro de la década actual; de hecho, el supercomputador que ocupaba en primer lugar del Top500 en junio de 2013 tenía 3.120.000 cores. En estas circunstancias, se espera que el MTBF de los principales supercomputadores llegue a caer por debajo de los 10 minutos en los próximos años [21].

En particular, en el ámbito del HPC, donde estos sistemas ejecutan aplicaciones paralelas intensivas en cómputo y de alta duración, el impacto de relanzar la ejecución a causa de haber obtenido resultados incorrectos a causa de los fallos, justifica la necesidad de adoptar estrategias de tolerancia a fallos para mejorar la robustez de estos sistemas.

1.4 Consecuencias de los fallos transitorios

La figura 1.5 (adaptada de [3]) describe las todas las posibles consecuencias de la incidencia de una partícula energética en el procesador o la memoria de una computadora.

La salida 1 de la figura 1.5 indica que la energía de la partícula no fue suficiente para generar un fallo.

El error latente ocurre cuando la energía de la partícula tiene energía suficiente para invertir un bit en la memoria, en un registro del procesador o en algún *latch* utilizado con algún propósito.

Si este bit alterado no es leído o se sobreescribe en algún momento antes de ser utilizado, el error latente no tiene consecuencias, y su ocurrencia no será notada (salida 2 de la figura 1.5). En tanto, si el bit alterado es leído por algún componente del sistema, el *soft error* se hace efectivo.

El *soft error* efectivo puede pasar desapercibido para las capas superiores del componente que lee si el bit alterado está protegido con detección y corrección (salida 3 de la figura 1.5). Este es el caso, por ejemplo, de las memorias que utilizan ECCs[2].

[2] Las memorias dinámicas de acceso aleatorio (DRAM) son dispositivos estructuralmente simples y, por eso, más vulnerables a los fallos transitorios, por lo que utilizan ECCs para mejorar su confiabilidad. Poseen bits extra que son utilizados por los controladores de memoria para almacenar la paridad de segmentos de bits.

Si el bit afectado tiene sólo detección de errores, produce un estado llamado "error detectado irrecuperable" (DUE), evitando que se generen salidas incorrectas. En el caso de un DUE, el componente que lee el bit alterado sabe que existe un error pero no hay un mecanismo capaz de corregirlo. La salida 4 de la figura 1.5 representa el caso en el que el *soft error* no afecta el resultado generado por el programa que se ejecuta sobre el sistema, llamado DUE falso. En este caso, es preferible evitar el mecanismo de detección para mejorar la performance del sistema, ya que agrega *over-head* cuando el error detectado no afecta realmente la salida del programa.

En cambio, si el *soft error* afecta los resultados del programa que se ejecuta en el sistema (salida 5 de la figura 1.5), el sistema debe informar a la capa superior (por ejemplo, el sistema operativo) de que existe un error efectivo, evitando que el programa continúe bajo esa condición. En ese caso el error recibe el nombre de DUE verdadero. Cuando el bit afectado es asignado a una aplicación, normalmente el sistema operativo produce la interrupción de la aplicación por comportamiento anormal (mata el proceso), pero el resto del sistema operativo y las demás aplicaciones continúan normalmente sus ejecuciones. En el caso de que el bit alterado sea utilizado por el sistema operativo, se provoca una situación, en la que la parte afectada del sistema sólo puede recuperarse por medio de un reinicio del sistema, para lo cual deben interrumpirse todas las aplicaciones en ejecución.

El mayor inconveniente se produce cuando un sistema es afectado por un *soft error* que ocurre en un componente que no cuenta con ningún nivel de protección. El bit alterado es leído y puede ser utilizado por la aplicación que se ejecuta sobre el sistema. La salida 6 de la figura 1.5 representa la situación en la que el *soft error* no detectado no afecta el resultado generado por el programa en ejecución.

Si el sistema utiliza en su operación el bit afectado sin apercibirse de su alteración, el sistema experimenta una "corrupción silenciosa de datos" (SDC), la más peligrosa de las consecuencias de un fallo transitorio. El SDC (salida 6 de la figura 1.5) es procesado por la aplicación o por el sistema operativo y puede causar efectos impredecibles sobre el comportamiento del sistema.

Actualmente, en la industria se especifica la tasa SER de los componentes en términos de las cantidades de SDC y DUE, como se expresa en la ecuación 4.

$$SER = DUE + SDC$$

Ecuación 4: tasa de *soft errors* de un componente o sistema

1.5 Posibles errores debidos a fallos transitorios

Los *soft errors* (errores causados por los fallos transitorios) tienen cuatro posibles consecuencias para el software que se ejecuta sobre el sistema, en términos de DUE y SDC: una excepción por instrucción inválida, un error de paridad durante un ciclo de lectura, una violación en un acceso a memoria [22] y un cambio en un valor producido por algún componente o por algún cómputo hecho por el programa [23]. Estas situaciones se muestran esquemáticamente en la figura 1.6.

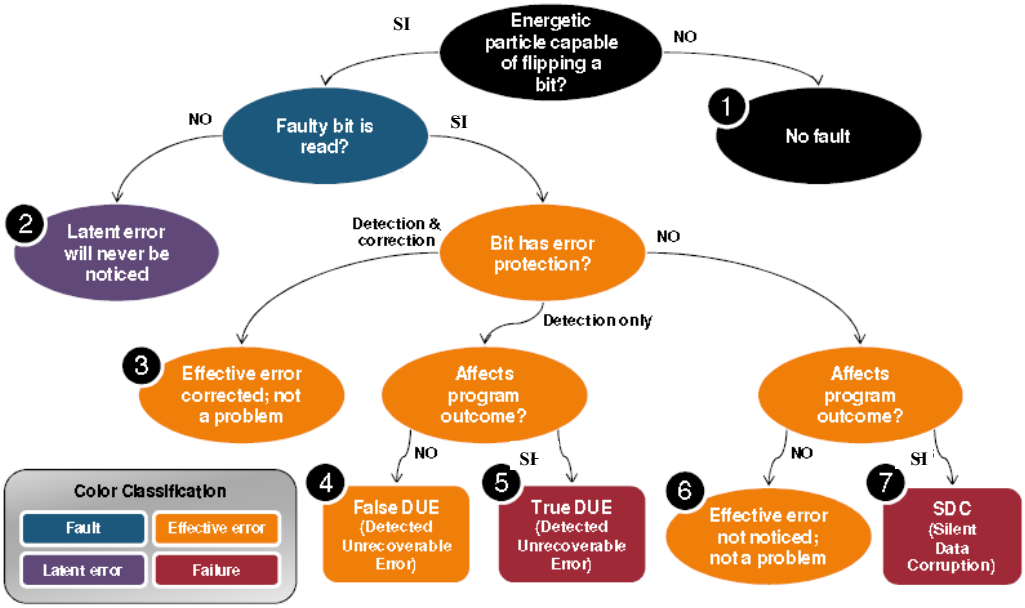

Figura 1.5: Posibles consecuencias de los fallos transitorios

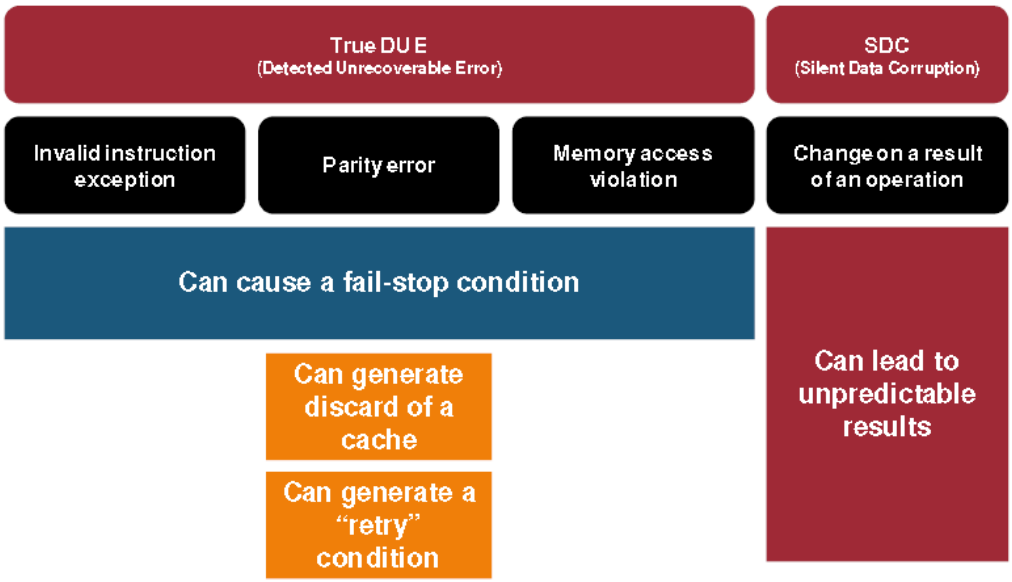

Figura 1.6: Posibles consecuencias de los *soft errors*

1.5.1 Excepción por instrucción inválida

Si se produce un cambio en un bit de una instrucción del programa, se puede generar una excepción por instrucción inválida (DUE) si el procesador es incapaz de decodificar la instrucción resultante. Pero puede darse también la situación en la que la alteración de la instrucción resulte en otra instrucción válida, por lo cual el *soft error* no será detectado por el procesador, que probablemente seguirá adelante con la ejecución luego de haber procesado una instrucción indeseada. En este caso se produce entonces un SDC.

De todas formas, si la instrucción afectada se transforma en otra instrucción válida, es posible que los parámetros de la instrucción original no se ajusten a los que requiere la instrucción modificada, por lo que ese caso resultará en una excepción por instrucción inválida.

Algunas arquitecturas pueden ser más vulnerables que otras a los cambios en las instrucciones. Por ejemplo, en la arquitectura x86 [24], la diferencia entre un salto condicional en el que se testea si un valor es mayor a otro y un salto condicional en el que se testea si un valor es menor o igual que otro (condiciones inversas) es de un sólo bit.

1.5.2 Error de paridad durante un ciclo de lectura

Un error de paridad generado por un fallo transitorio puede ocurrir en dispositivos de memoria (principal o caché) o en los buses.

Es común que los buses que implementan control de paridad en las transmisiones también tengan implementada la posibilidad de retransmisión de los datos afectados. Esto tiene por efecto un retardo en la transmisión, pero el sistema continúa operando normalmente. Cuando un *soft error* modifica una posición de memoria y el componente afectado usa paridad para detectar esa alteración, las consecuencias del error dependen de la ubicación de la posición afectada en la jerarquía de memoria.

Si el error de paridad está en una posición de la memoria principal, es posible recuperarlo si la porción afectada no ha sido modificada por el cómputo normal, y existe una copia de la página en el archivo de *swap*. En este caso, el sistema operativo puede restaurar la página afectada a su estado previo desde el archivo de *swap*. En cambio, si la memoria ha sido modificada y no hay copia actualizada, el sistema operativo puede abortar la aplicación, o elevar el error a la aplicación, permitiendo que ésta intente recuperarse.

En el caso de que el error de paridad ocurra en una memoria caché, la situación es muy similar a la explicada anteriormente: si la porción afectada no ha sido modificada, el controlador de memoria puede buscar una copia en la memoria principal (o el nivel superior de caché) y restaurar el estado anterior; si ha habido modificación, el sistema operativo puede interrumpir la aplicación o permitirle tratar con el error.

1.5.3 Violación en acceso a memoria

Un *soft error* que afecte a un puntero a memoria puede producir que la aplicación intente leer o escribir datos, o producir un salto, a direcciones que se encuentran en un espacio de memoria que se encuentra fuera del alcance de la aplicación. Los procesadores modernos cuentan con protección ante esta situación, y elevan al sistema operativo un error de violación de acceso. Una vez notificado de que una aplicación está intentando acceder a una dirección inexistente o que pertenece al espacio de otro proceso, el sistema operativo detiene la ejecución, evitando que el error se propague a otras partes del sistema.

1.5.4 Cambio de un valor

Un único bit que se haya alterado debido a un fallo transitorio es suficiente para generar un *soft error* que afecte la operación de un componente, cambiando un resultado esperado por uno inesperado.

Este es el caso de los errores que afectan a los componentes internos del procesador. Los registros, el *pipeline*, la ALU, las unidades de punto flotante y casi todos los elementos de un procesador moderno tienen algún tipo de memoria (para almacenar resultados intermedios de las operaciones) y alguna forma de bus para comunicarse con los demás componentes del procesador. Todas estas memorias auxiliares y buses internos son blancos posibles de los fallos transitorios.

Un *soft error* en un componente interno del procesador puede pasar desapercibido si el componente no cuenta con protección y si el resultado no viola el espacio de direcciones de la aplicación ni produce ninguna operación inválida.

La SDC es el efecto más frecuente de los *soft errors* en este tipo de resultados. La ejecución no se detiene, y el usuario sólo podrá apercibirse de su ocurrencia si los resultados generados por la aplicación difieren significativamente de los usuales.

1.6 Fallos transitorios en sistemas paralelos

En las secciones 1.1 y 1.4 se justificó la necesidad de implementar estrategias de tolerancia a fallos transitorios para mejorar la robustez en el ámbito de HPC, donde aplicaciones paralelas intensivas en cómputo y de alta duración se ejecutan sobre grandes clusters de multicores, que consisten en un conjunto de procesadores de múltiples núcleos interconectados mediante una red, y trabajan cooperativamente como un único recurso de cómputo [25].

1.6.1 Concepto de sistema paralelo

Existe una continua demanda para aumentar el poder de procesamiento de los sistemas informáticos, proveniente, especialmente, de áreas en las que se requieren realizar grandes cantidades de cálculos o trabajar con considerables volúmenes de datos, realizando el trabajo en un período de tiempo razonable, de acuerdo al problema a resolver [26].

A partir de las limitaciones físicas que impiden mejorar indefinidamente el rendimiento de las computadoras secuenciales, una manera natural de incrementar el poder de cómputo ha sido la utilización de múltiples procesadores (en una o más máquinas) trabajando de manera conjunta para resolver el problema. En este sentido, el paralelismo es un concepto intuitivo porque el mundo real es esencialmente paralelo [27].

Una arquitectura paralela es una colección de elementos de procesamiento que se comunican y cooperan. Una aplicación paralela busca aprovechar las características de la arquitectura con el objetivo de resolver un problema común, tratando de reducir el tiempo de ejecución de la aplicación, y/o resolver problemas de mayor tamaño.

Los requerimientos de evolución han conducido a un gran esfuerzo por transformar el procesamiento secuencial en paralelo. La creación de algoritmos paralelos/distribuidos, o la transformación de un algoritmo secuencial en paralelo, son tareas complejas y están influidas por la arquitectura de soporte. Un sistema paralelo es la combinación de un algoritmo paralelo y la máquina paralela sobre la cual éste se ejecuta [28].

Existen diversas razones que justifican la importancia que ha adquirido el procesamiento paralelo y distribuido. Algunas de ellas son [29]:

- La evolución de la tecnología de los componentes y las arquitecturas de procesamiento, con el consecuente crecimiento de la potencia de cómputo.
- La transformación y creación de algoritmos que explotan la concurrencia implícita del problema a resolver, de modo de distribuir el procesamiento minimizando el tiempo de respuesta.
- La capacidad del cómputo distribuido/paralelo de reducir el tiempo de procesamiento en problemas de cálculo intensivo (simulaciones, búsquedas, cómputo científico, etc.) o de grandes volúmenes de información (bases de datos, imágenes, etc.).
- Las posibilidades que el paradigma paralelo ofrece en términos de investigación de técnicas para análisis, diseño y evaluación de algoritmos.

En la figura 1.7 se muestra esquemáticamente la operación de un sistema paralelo.

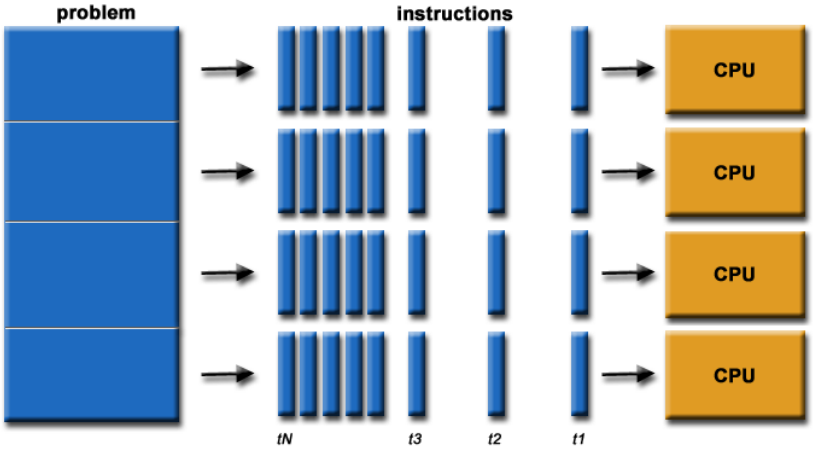

Figura 1.7: Sistema paralelo conformado por un algoritmo paralelo ejecutándose sobre una arquitectura paralela

1.6.2 Características de aplicaciones paralelas científicas de paso de mensajes

En la sección 1.1 se señalaba que la incidencia de los fallos transitorios se vuelve más significativa en el ámbito del HPC, especialmente en el caso de las aplicaciones de cómputo intensivo y de gran duración, debido al alto costo que implica volver a lanzar la ejecución desde el comienzo. La metodología de detección que se analizará en el marco de este trabajo está diseñada para aplicaciones paralelas científicas que utilizan paso de mensajes. En esta sección se revisan las características principales de esta clase de aplicaciones.

La computación científica estudia la construcción de modelos matemáticos y técnicas numéricas para resolver problemas científicos, de ciencias sociales y de ingeniería. Típicamente, consiste en la utilización de modelos numéricos y otras formas de cálculo, implementadas sobre computadoras, para ganar entendimiento sobre diversas disciplinas científicas. Se desarrollan aplicaciones informáticas que responden a modelos matemáticos de los sistemas en estudio, de forma de poder ejecutarlas con diferentes conjuntos de datos de entrada. Actualmente, la computación científica es considerada como el tercer modo de ciencia, complementando a la teoría y a la experimentación/observación.

Generalmente, las aplicaciones científicas modelan situaciones del mundo real o cambios en las condiciones de diversos fenómenos naturales. Algunas aplicaciones relevantes son:

I. Estudio de fenómenos naturales

a. Predicción meteorológica
b. Simulación para estudio del cambio climático
c. Reconstrucción de desastres naturales (terremotos, tsunamis, etc.)
d. Comportamiento de partículas subatómicas

II. Biología molecular (simulación, visualización en 3D)
a. Modelado de estructuras de ADN
b. Alineamiento de secuencias de ADN
c. Modelado de mecanismos enzimáticos
d. Reconocimiento de ácidos nucleicos en proteínas
e. Comparación de secuencias moleculares

III. Medicina
a. Diseño de fármacos
b. Búsqueda de genomas completos
c. Vínculos de anticuerpos/antígenos

IV. Astronomía
a. Movimiento de cuerpos celestes por el espacio

V. Ingeniería
a. Dinámica de fluidos
i. Transporte aéreo y turbulencias
b. Simulaciones de comportamiento de vehículos
c. Estudios de ciencia de los materiales
i. Diseño de semiconductores

Muchas de estas aplicaciones científicas operan creando una malla lógica en la memoria de la computadora, donde cada elemento corresponde a un área en el espacio y contiene información sobre factores que son relevantes para el modelo. Un conocido ejemplo, tomado de [27], es el del pronóstico meteorológico global. En este caso, la atmósfera se modela dividiéndola en celdas tridimensionales. Los diversos efectos que afectan a la atmósfera se describen mediante complejas ecuaciones matemáticas. Las condiciones en cada celda (como temperatura, presión, humedad, velocidad y dirección del viento, etc.) se computan a intervalos de tiempo regulares utilizando los valores del intervalo anterior en la misma celda y las celdas vecinas. Los cálculos para cada celda son repetidos muchas veces para modelar el paso del tiempo. Para predicciones de varios días, es necesario tomar en cuenta que la atmósfera es afectada por eventos muy distantes, por lo que se requiere una gran cantidad de celdas. Si la atmósfera se divide en celdas de $1km^3$, hasta una altura de 16km (16 celdas de altura), resultan necesarias más de 20×10^8 celdas. Si se supone que el cálculo de una celda para un instante de tiempo conlleva 200 operaciones de punto flotante, se requieren 4×10^{11} operaciones para la simulación global de un intervalo de tiempo. Para predecir el tiempo durante una semana con intervalos de 1 minuto, se

requieren 10^4 pasos de simulación y 4 x 10^{15} operaciones de punto flotante en total. Además, para que el resultado de las predicciones climáticas resulte útil, los cálculos deben resolverse rápidamente, lo cual constituye una restricción considerable. Sin embargo, el tiempo necesario para las simulaciones se incrementa significativamente a medida que el sistema simulado se vuelve más complejo.

Este ejemplo es una muestra de que las aplicaciones del cómputo científico demandan una gran potencia computacional, debida a la complejidad de los modelos y a la gran cantidad de datos que deben procesar.

El aumento de la potencia computacional, conjuntamente con la introducción del procesamiento en paralelo, han beneficiado enormemente al cómputo científico, facilitando el manejo de grandes cantidades de datos de forma más rápida y eficiente, y consiguiendo resolver problemas cada vez más complejos. La mayoría de los modelos utilizados permiten un alto grado de paralelización, tanto de tareas como de datos, de forma que el desarrollo de aplicaciones paralelas representa un gran avance para las diversas disciplinas científicas. Por supuesto, resulta necesario diseñar estas aplicaciones paralelas de manera de que puedan funcionar sobre sistemas de cómputo paralelos/distribuidos, obteniendo el mayor provecho posible de las prestaciones que ofrecen estos sistemas. Teniendo en cuenta todo esto, se puede afirmar que el cómputo científico es el principal beneficiario del HPC [27].

En general, una aplicación paralela es aquella en la que un conjunto de procesos trabajan simultáneamente, cooperando para realizar una tarea. Cada uno de los procesos realiza localmente una cantidad de cómputo, y se comunica con los demás procesos de la aplicación para intercambiar resultados.

Las aplicaciones científicas paralelas que resuelven diferentes problemas específicos, comparten, sin embargo, algunas características comunes. Todas ellas, en general, deben manejar grandes volúmenes de información (por ejemplo, en forma de matrices o vectores), que incluso pueden provenir de diferentes medios y dispositivos de entrada/salida. Respecto de las tareas que realizan, todas involucran alguna clase de filtrado y reducción de los datos, y su presentación de forma inteligible. Todas deben resolver grandes sistemas de ecuaciones con adecuada velocidad y precisión numérica, encargándose de la correcta distribución de los datos y de la gestión de los resultados que se generan.

En el modelo de programación de pasaje de mensajes, cada proceso tiene su propio espacio de direcciones, por lo que los datos son vistos como asociados a un proceso en particular. La comunicación y sincronización entre los procesos se da a través del envío y la recepción de mensajes, de forma que el acceso a un dato remoto requiere de una comunicación explícita. En este modelo, las primitivas de envío y recepción son las encargadas de manejar la sincronización [25].

El modelo de pasaje de mensajes tiene la ventaja de que los programas son altamente portables, pudiendo ejecutarse sobre prácticamente cualquier computadora o sistema paralelo. Otra ventaja consiste en el programador tiene control explícito sobre la ubicación de los datos en la memoria. El acceso y el manejo de la memoria inciden directamente sobre el rendimiento de la aplicación, por lo que este control explícito le permite al programador la posibilidad de obtener un alto desempeño en ejecución. Sin embargo, esto mismo constituye la mayor desventaja de este modelo, ya que el programador no puede desligarse de los detalles (como la ubicación de los datos en la memoria) y debe ocuparse del orden de las sentencias de comunicación [30].

1.6.3 Consecuencias de fallos transitorios en sistemas paralelos

Como se explicó en la sección 1.3, los fallos transitorios pueden afectar a diferentes componentes del hardware de una computadora (en particular, de una computadora paralela), mientras que los *soft errors* son manifestaciones externas que resultan de ese estado interno inconsistente, observables a nivel de la ejecución (bajo la presunción de que los programas son determinísticos). La cantidad total de *soft errors* que ocurren la ejecución puede dividirse de la siguiente manera, según sus efectos sobre la ejecución [2]:

$$SER = LE + DUE + TO + SDC$$

Ecuación 5: contribuciones a la tasa total de *soft errors* desde el punto de vista de la aplicación

Dentro de las contribuciones al SER que aparecen en la ecuación 5, un LE (*Latent Error*, también llamado fallo benigno) es un fallo que causa la alteración de datos que no volverán a ser utilizados por la aplicación, por lo que, a pesar de que efectivamente ocurre, no se propaga ni tiene impacto en los resultados de la ejecución.

De manera similar a la explicada en la sección 1.6, un DUE es un error detectado sin posibilidad de recuperación. Son consecuencia de fallos que causan condiciones anormales detectables a nivel de alguna capa de intermedia del software del sistema (por ejemplo, el sistema operativo o la librería de comunicaciones). Normalmente, causan la finalización abrupta de la aplicación. Ejemplos de esto son los intentos de acceso a direcciones ilegales de memoria (*segmentation faults*) o ejecución de instrucciones no permitidas, como una división por cero.

Un error TO (*Time Out*) se produce cuando, debido a la ocurrencia de un fallo, el programa no logra finalizar dentro de un determinado lapso de tiempo. Un ejemplo de esto es la alteración del valor límite de un bucle, que produce que el programa entre en un lazo infinito.

Finalmente, como se mencionó en la sección 1.6, se produce SDC cuando el dato alterado por el fallo no causa ninguna condición anormal, por lo que no es detectado por el software del sistema. El error se propaga silenciosamente, afectando el resultado final. Desde la perspectiva del hardware, la SDC es el resultado de la inversión de uno o más bits de un registro del procesador que es utilizado por la aplicación. Los errores que producen SDC son los más peligrosos desde el punto de vista de la fiabilidad del sistema, dado que la aplicación aparenta ejecutarse correctamente pero, silenciosamente, genera salidas incorrectas. Debido a esto, resulta particularmente importante el desarrollo de estrategias de tolerancia a fallos que sean capaces de interceptar los errores que derivan en SDC.

Existen dos formas de que la ocurrencia de un fallo transitorio en un procesador sobre el que se ejecuta uno de los procesos de una aplicación paralela de paso de mensajes resulte en una SDC. Esto se muestra en la ecuación 6:

$$SDC_{paralelo} = TDC + FSC$$

Ecuación 6: contribuciones a la tasa total de SDC en aplicaciones paralelas

El término TDC (*Transmitted Data Corruption*) representa la situación en la que el dato afectado por el fallo forma parte del contenido de un mensaje que debe ser enviado. Si el error no es detectado, la corrupción se propaga a otros procesos de la aplicación paralela.

En cambio, en el caso de FSC (*Final Status Corruption*), el fallo afecta un dato que no será enviado a otros procesos, pero que es utilizado por el proceso local, propagando el error a lo largo de su ejecución y corrompiendo su estado final. En este caso, el error se comporta de la misma manera que en el caso de un programa secuencial.

Esta división posibilita el desarrollo de estrategias de detección diferentes para cada caso.

Como se comentó en la sección 1.8.2, una aplicación paralela consiste en un conjunto de procesos que colaboran para realizar una tarea. Por lo tanto, el éxito de una aplicación paralela depende fuertemente de la comunicación de los resultados del cómputo local de un proceso a los demás. Debido a esto, todos los fallos que conducen a TDC tienen un alto impacto sobre los resultados finales de la aplicación paralela. Por otra parte, los fallos que provocan FSC están relacionados con la fracción centralizada del cómputo, por lo que pueden ser detectados mediante una comparación de los resultados finales de la aplicación con los generados por una réplica. Extendiendo este razonamiento, si la tarea se divide en un mayor número de procesos, en general, circularán más mensajes, por lo que la tendencia es al crecimiento de la proporción de TDC.

La metodología SMCV, que se explica en detalle en el capítulo 6, es capaz de interceptar todos los fallos que producen SDC y TO en una aplicación paralela, proponiendo un esquema de detección basado en el control de los contenidos de los mensajes antes de enviarlos (para detectar TDC), incorporando una comparación final de resultados que asegura la fiabilidad del sistema y permitiendo la configuración de retardos programables de sincronización entre réplicas para evitar que las aplicaciones ingresen en esperas infinitas.

Capítulo 2

Detección de fallos transitorios

2.1 Modelo de Fallo

A lo largo del resto del trabajo, se asumirá el modelo de fallo SEU (*Single Event Upset*), en el que exactamente un bit resulta invertido a lo largo de la ejecución. La técnica propuesta, y la mayor parte de aquellas que se examinan en este capítulo, tienen por objetivo detectar este tipo de fallos, pero no son tan eficientes en caso de que se presenten fallos múltiples. Sin embargo, como se justifica en la literatura [4], los casos en los que múltiples fallos pueden combinarse para escapar de los mecanismos de detección son muy pocos, por lo que la probabilidad de que se produzcan esas combinaciones es extremadamente baja, de manera que la detección de fallos SEU es la meta principal. Más adelante en este capítulo se detallará un poco más este punto (sección 2.7).

Por otra parte, también se asume que el subsistema de memoria, incluyendo los diferentes niveles de caché, están adecuadamente protegidos mediante la utilización de técnicas como los bits de paridad o los ECC´s. Esta afirmación se cumple para la gran mayoría de los sistemas de cómputo modernos [4].

Los fallos transitorios representan un problema sólo cuando modifican resultados de cómputo que son observables desde la óptica de un usuario externo. El único evento observable desde el exterior de la máquina es la secuencia de escrituras sobre los dispositivos de salida. Por lo tanto, el modelo especifica que, independientemente de la operación interna de la computadora, la única forma de que las consecuencias de un fallo se hagan explícitas es que el programa escriba los resultados alterados en memoria, donde un dispositivo mapeado en memoria pueda leerlos y procesarlos [7]. Aunque el comportamiento del procesador se aparte drásticamente del esperado, se considera que el programa se ejecuta correctamente si la secuencia de valores escritos a memoria resulta inalterada.

Un fallo transitorio puede afectar a un bit cualquiera dentro del procesador. Este bit puede pertenecer a una instrucción, un dato o una dirección. En la sección 1.7.1 se explicó el caso de los fallos que afectan a los códigos de instrucción, que normalmente se resuelven en excepciones por instrucción inválida.

Si un fallo afecta a una dirección, es decir, altera el flujo de control, en última instancia se podrá observar una modificación en la secuencia de valores que el programa almacena en memoria. En consecuencia, el hardware provee mecanismos para detectar fallos en las direcciones que sirven como destinos de saltos.

Las instrucciones de carga de datos (*load*) no son peligrosas en este modelo (en el sentido de que pueden afectar la fiabilidad de la ejecución). A pesar de que un fallo puede corromper un dato que luego será leído, la consecuencia de dicho fallo será observable al momento de escribir en memoria un resultado calculado en base a dicho valor erróneo. Debido a esto, el fallo será, en definitiva, detectado (aún a costa de cierta latencia adicional en la detección) si se verifican los valores que se escriben en memoria. Tampoco es necesario agregar ningún mecanismo para comprobar las direcciones utilizadas en las instrucciones load. Un fallo puede producir una dirección inválida, que en la práctica inducirá una excepción de hardware (como un *segmentation fault*) o resultará en la carga de un valor arbitrario. Dos instancias de ejecución redundante deben escribir valores idénticos en ubicaciones de almacenamiento idénticas y deben emitir órdenes para transferir el control a direcciones idénticas.

En general, en los modelos no se contemplan los casos de fallos que ocurren durante la ejecución de las instrucciones. Esto se debe a que estos fallos se pueden equiparar con una ejecución correcta compuesta con un fallo que ocurre inmediatamente antes o después. Por ejemplo, en una sencilla instrucción de suma aritmética entre valores alojados en registros, una falla en el hardware del sumador durante la ejecución es equivalente a la ejecución correcta de la suma, seguida de un fallo en el registro de destino. De esta misma manera, la mayor parte de los fallos intra-instrucción se pueden modelar modificando el archivo de registros antes o después de la instrucción. Debido a esta equivalencia, las instrucciones aritmético-lógicas básicas no se consideran operaciones peligrosas desde el punto de vista de la fiabilidad.

En conclusión, las instrucciones de almacenamiento de datos en memoria (*store*) son las únicas que influyen sobre la fiabilidad de la ejecución, ya que hacen visibles los resultados hacia el exterior. Sin un hardware especial es imposible garantizar que las instrucciones de almacenamiento acceden correctamente a memoria. Una verificación realizada por software justo antes de una instrucción store convencional, por sofisticada que sea, resultará inútil si el fallo ocurre justo entre la comprobación y la escritura en memoria [7].

2.2 Objetivos de la detección

La estrategia general del cualquier programa que incorpora detección de fallos es mantener dos hilos de cómputo redundantes e independientes. La ejecución de uno de ellos va levemente por delante de la del otro, pero existe una gran flexibilidad respecto cómo pueden intercalarse las instrucciones para cada paso del cómputo. Previo a la escritura de un valor en un dispositivo mapeado en memoria, los resultados de cada cómputo se comparan para ver si hay equivalencia. Si los resultados parciales son diferentes, el sistema debe ser notificado de la ocurrencia de un fallo por el módulo de detección. También deben verificarse los argumentos de todas las transferencias de flujo de control. Esta metodología se ha mostrado en la literatura como una forma efectiva de implementar la detección [31,32].

Se debe comenzar por definir la detección de fallos para un paso de la ejecución del programa. La definición dice que, si existen dos computaciones similares, una con un fallo y otra sin fallo, entonces la computación con fallo debe dar un paso indistinguible de la que no tiene fallo, o la computación defectuosa debe alcanzar el estado de fallo [7].

Abstractamente, todas las soluciones que existen para detectar fallos involucran el agregado de algún tipo de redundancia, pero los detalles varían significativamente. Tradicionalmente, las propuestas se han dividido en las que atacan el problema de la detección desde la perspectiva del hardware y las que lo hacen desde la óptica de las aplicaciones. Existen propuestas de solución basadas únicamente en agregados al hardware, como los ECC´s, coprocesadores *watchdog* [33] e hilos redundantes de hardware [34,35,36]; y técnicas de software puro, diseñadas tanto para monoprocesadores como para multicores [31,32,37,38,39]. En términos generales, las soluciones por hardware son más eficientes para la aplicación de una política de confiabilidad, pero las soluciones de software puro son más flexibles (se pueden aplicar y configurar de acuerdo a las necesidades del entorno concreto) y menos costosas en términos de hardware [4,7].

Un aspecto importante, vinculado a la detección, es el intervalo de comparación. Si los resultados parciales son validados demasiado frecuentemente (por ejemplo, en cada escritura), se introduce una alta sobrecarga de trabajo, que redunda en una gran cantidad de tiempo adicional en la ejecución de las aplicaciones, ligada al mecanismo de detección. Como ventaja, se acota la latencia de detección del fallo. Una vez detectado, el mecanismo de recuperación que se utilice podrá partir de un estado consistente reciente (ya que la mayor parte del cómputo realizado antes de la detección estará validado).

En el otro extremo, si sólo se comparan los resultados finales de las aplicaciones, la sobrecarga de trabajo introducida por el mecanismo de detección es baja, pero la latencia de detección y el costo de la recuperación son altos, ya que no existe un último estado consistente y hay que volver al comienzo de la ejecución. Por lo tanto, debe lograrse un compromiso entre el intervalo de comparación y la sobrecarga introducida [1].

Una vez que se ha detectado un fallo transitorio, se puede utilizar una estrategia de recuperación arbitraria, definida por el usuario, permitiéndose muchas variantes en este aspecto. Gracias a que es posible desacoplar la detección y la recuperación en esta forma, usualmente se las estudia por separado. Inclusive, debido a la (relativa) poca frecuencia de los fallos, el mecanismo de recuperación se ejecuta mucho menos que el de detección, que debe estar activo continuamente [4]. Más aún, el beneficio de ejecutar un mecanismo de recuperación depende del momento de ocurrencia del fallo respecto del ciclo de vida de la aplicación. Una aplicación puede decidir no realizar la recuperación si llevarla a cabo implica volver tan atrás que el costo es el mismo que el de relanzar desde el principio, por lo que no se obtendría ningún beneficio apreciable [40]. Debido a esto, una estrategia de detección diseñada de forma compatible con la mayoría de los mecanismos de reporte y recuperación puede ser extendida con relativa sencillez para proveer tolerancia a fallos completa.

En el resto de este capítulo se revisan las principales propuestas que se han realizado en la literatura en el campo de la detección de fallos transitorios. Para clarificar la discusión, se han dividido en: las propuestas basadas en la realización de modificaciones al hardware para agregar redundancia en algunas computaciones significativas; las propuestas basadas en el agregado de redundancia a nivel del software del sistema o de la aplicación; y la propuestas híbridas que combinan las dos estrategias anteriores. Finalmente, se realiza un breve análisis de propuestas existentes de tolerancia a fallos que se aplican a librerías de comunicaciones para paso de mensajes.

2.3 Propuestas basadas en hardware

Las técnicas basadas en hardware [35,41,42,43] buscan proteger los distintos elementos del procesador agregando lógica adicional para proveer redundancia. Son las más utilizadas en ámbitos críticos, como los sistemas de vuelo o los servidores de alta disponibilidad.

Hace largo tiempo que se utilizan técnicas muy localizadas a nivel de bit (como los ECC´s y las sumas de paridad) de forma eficiente en estructuras de almacenamiento como la memoria. Cuando se requiere proteger zonas grandes del procesador se utilizan técnicas de más alto nivel, que incluyen la duplicación de unidades funcionales, cores de procesamiento o contextos de hardware [7]. La idea principal es la duplicación de la ejecución, operando ambas réplicas sobre los mismos datos de entrada y comparando sus salidas con una frecuencia determinada. Sólo una de las réplicas realiza las escrituras en memoria o envía mensajes a otros procesos [1].

A pesar de que todas las propuestas basadas en hardware involucran el agregado de redundancia al cómputo, los detalles varían mucho, desde coprocesadores *watchdog* [33] a hilos redundantes por hardware [34,35,38,41,44,45].

En 1998, Saxena y McCluskey [46] fueron los primeros en proponer la utilización de hilos redundantes para mitigar el efecto de los fallos transitorios, mediante una técnica denominada SMT (*Simultaneous Multithreading*). Rotenberg [45] expandió el concepto de SMT con AR-SMT (*Active-stream/Redundant-stream Simultaneous Multithreading*). En esta técnica, dos copias explícitas de la aplicación se ejecutan concurrentemente utilizando los mismos recursos del procesador. Las dos copias son tratadas como programas independientes, teniendo cada una su propio contexto de ejecución. Aplicada en un procesador superescalar, esta técnica combina la redundancia espacial con la temporal.

En [38], los autores propusieron un procesador SRT (*Simultaneous and Redundant Threaded*), capaz de ejecutar simultáneamente dos réplicas de un programa en la forma de hilos independientes, pero planificando los recursos de hardware dinámicamente entre ellos, por lo que lograba una mejora en performance respecto de AR-SMT. Los datos de entrada debían ser duplicados, y la detección se lograba verificando cada escritura a memoria y cada lectura de datos no-cacheados, o cada actualización de un registro. Su principal desventaja es la dificultad de implementar el *lockstepping* entre las réplicas (es decir, la sincronización ciclo a ciclo para la comparación de salidas y duplicación de entradas), en un entorno de *scheduling* dinámico y ejecución especulativa.

En [35] se propuso CRT (*Chip-level Redundantly Threaded multiprocessor*), que es la adaptación de SRT a entornos multicore. Se mantuvo la cobertura ante los fallos del *lockstepping* pero mejorando la performance, ya que en este caso las réplicas se ejecutan en procesadores diferentes, permitiendo la ejecución de dos programas simultáneos con detección de fallos. En esta línea, diferentes autores han hecho uso de la multiplicidad de bloques de hardware disponibles en arquitecturas multicore y *multithread* para implementar redundancia en el cómputo.

SRTR (SRT con recuperación) [41] es una técnica que incorpora la recuperación. Propone retardar el *commit* de las instrucciones para que las verificaciones se realicen antes de escribir valores erróneos en memoria, de forma de evitar "deshacer" instrucciones. La recuperación utiliza la capacidad de "rebobinar" la ejecución que proveen los pipelines modernos, hasta un estado conocido. En tanto, CRTR (CRT con recuperación) [34] propone mejoras al mecanismo de detección, separando la ejecución de los *threads* para enmascarar la latencia de comunicación entre los cores y realizando *commits* asimétricos entre ellos (uno de los *threads* lo hace antes de la verificación y el otro después), de forma que el estado del *thread* que se ejecuta por detrás se puede utilizar para la recuperación. En resumen, utiliza el mecanismo mejorado de detección de CRT para multicores e incorpora la recuperación propuesta en SRTR.

En [47] los autores proponen algunas modificaciones a la microarquitectura de un procesador superescalar con ejecución fuera de orden para obtener una versión tolerante a fallos.

En [42] se proponen mejoras a la técnica DDMR (*Dynamic Double Modular Redundancy*), en la cual pares de cores se asocian dinámicamente por software para proveer redundancia, de forma de evitar que cores defectuosos afecten la fiabilidad de la ejecución y permitiendo asociaciones entre cores similares para obtener velocidades de ejecución semejantes. Además, provee soluciones a la sincronización entre cores y al procesamiento asimétrico de interrupciones o excepciones (es decir, que ocurren en un core pero no en el otro), así como también mejoras a la escalabilidad. También permite configurar el sistema para operar en modo redundante o utilizando los cores por separado para procesamiento.

En la misma línea, en [48] 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 ejecuta 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, y se permiten opciones de configuración para proveer flexibilidad.

Los autores de [44] proponen la obtención de una versión reducida de la aplicación, quitando cómputo ineficaz y flujo de control predecible. La aplicación original y su versión reducida se ejecutan en *threads* separados, proveyendo redundancia parcial y adelantando resultados parciales que aceleran la ejecución. En tanto, en [43] proponen la selección de un core que lleve a cabo tareas de monitorización sobre los procesos que se ejecutan en los otros cores, verificando cíclicamente sus estados. Se plantea el problema del método de selección del core de monitorización y los cam-

bios de contexto inter-core. Alternativamente, se puede utilizar más de un core para diagnóstico, permitiendo la configuración de un nivel de cobertura, así como un nivel máximo de *overhead* permitido.

Estas no son las únicas propuestas basadas en redundancia de hardware; el análisis previo no pretende ser exhaustivo, pero sin embargo proporciona una idea del trabajo que se ha realizado en este campo.

Ciertamente, estas propuestas están diseñadas para funcionar en sistemas que utilizan procesadores de altas prestaciones, pero no son suficientes en ámbitos críticos. Por ejemplo, una aeronave del tipo del Boeing 777 ha utilizado tres procesadores y buses de datos redundantes diferentes, con un esquema de votación de mayoría para lograr tanto detección como recuperación en tiempo real [4,12].

2.4 Propuestas basadas en software

Las técnicas basadas en redundancia de hardware son ineficientes en computadoras de propósito general. El diseño y la verificación de elementos de hardware agregados *ad-hoc* (y equipados con mecanismos de detección de errores que verifican la operación correcta en tiempo de ejecución) son procesos costosos. Además, las condiciones ambientales en las que se desempeñan las computadoras, así como el envejecimiento de los componentes, son causas de fallo que no pueden ser predichas durante la etapa de desarrollo [13]. Por otro lado, la evolución arquitectural hacia los multicores ha producido un gran interés en la adaptación de los recursos paralelos que proporcionan para utilizarlos en pro de tolerar fallos transitorios [2].

Conceptualmente, la lógica combinatoria interna al procesador se puede proteger por medio de duplicación, pero, en la práctica, esta técnica es imposible de aplicar a elementos como *latches* o unidades aritméticas [7]; como consecuencia, la tarea de proteger la lógica interna resulta compleja, siendo los registros la preocupación principal por su propensión a los fallos [13]. En todos los procesadores modernos (y no sólo los que se usan en aplicaciones de alto desempeño o disponibilidad) se contempla la vulnerabilidad a los fallos transitorios como un problema a tratar en un diseño agresivo [4].

Por otra parte, en muchas aplicaciones (como audio o video bajo demanda), las consecuencias de los fallos no son tan severas, por lo que no es crítico incorporar mecanismos exhaustivos de tolerancia a fallos [13].

Bajo estas circunstancias, en las que los costos de desarrollo son elevados, y aún en aplicaciones de seguridad crítica, los diseñadores tienden a adoptar el hardware disponible en el mercado. En este contexto, se han propuesto aproximaciones basadas en software puro [4,7], que resultan atractivas ya que permiten implementar sistemas fiables frente a los fallos transitorios sin requerir ningún hardware adicional, siendo esta su principal ventaja y el factor que las hace las más apropiadas para los sistemas de cómputo de propósito general [7,12]. La tolerancia a fallos basada en replicación de software es un campo de estudio bien conocido con algunas décadas de historia.

Las técnicas de software representan un compromiso entre el nivel de cobertura frente a fallos que pueden lograr, el bajo costo de diseño (en términos de hardware) y la flexibilidad en el despliegue (pueden ser configuradas de diferentes formas para adaptarse a las necesidades específicas de las aplicaciones [48]). A pesar de no ser capaces de lograr el mismo desempeño de las técnicas de hardware (ya que deben ejecutar instrucciones adicionales y no pueden examinar el estado de la microarquitectura), se han mostrado como alternativas promisorias, ya que logran niveles significativos de fiabilidad introduciendo un *overhead* razonable a la ejecución [31,32,37].

La idea básica que hay detrás de las propuestas basadas en software para la detección de fallos es la duplicación del cómputo de la aplicación. Las dos réplicas deben operar sobre copias de los mismos datos de entrada y deben comparar sus salidas periódicamente; si los resultados difieren, el sistema ha experimentado un fallo transitorio [4].

Los autores de [49] proponen una técnica para habilitar por software los códigos ECC para datos en memoria. Oh y Mc Cluskey [50] analizan diferentes opciones de duplicación de procedimientos y argumentos a nivel de código fuente para proveer tolerancia a fallos por software y minimizar el consumo energético. En tanto, en [51] se propone una pre-compilación de código fuente para generar código de detección de fallos en lenguaje de alto nivel. Esta técnica aumenta entre 3 y 5 veces el *overhead* y es incapaz de detectar de 1 a 5% de los fallos.

Los fallos se traducen en errores que afectan al flujo de control o a los datos de la aplicación. Los errores en el flujo de control son aquellos que causan una divergencia en la secuencia de valores del contador de programa respecto de los que se observarían en una ejecución libre de fallos. Para esta clase de fallos, las técnicas de software propuestas se basan principalmente en el análisis de firmas, donde una firma única está asociada a un bloque básico de programa que previamente ha sido compilado y alojado en memoria. Durante la ejecución, se computa la firma y se compara con una de referencia [12].

Por otro lado, los errores que afectan a los datos impactan sobre valores de variables, registros o posiciones de memoria utilizados por la aplicación. Para tratar con ellos, las aproximaciones existentes almacenan copias redundantes de la misma información e introducen verificaciones de consistencia entre las réplicas [12].

Algunas de las técnicas propuestas en la literatura utilizan el compilador para insertar instrucciones redundantes y verificaciones del cómputo [31], comprobaciones de flujo de control [37] o una combinación de ambas [4]. Durante la compilación, las instrucciones del programa son duplicadas e intercaladas con las instrucciones originales. Sin embargo, cada copia utiliza diferentes registros y posiciones de memoria para no interferir con la otra. En ciertos puntos de sincronización del programa combinado resultante, el compilador inserta instrucciones de comprobación para asegurar que las instrucciones originales y sus copias redundantes concuerdan en los valores que han computado.

Como se explicó en la sección 2.1, si se asume un esquema de Entrada/Salida mapeada en memoria, un programa se ejecuta correctamente si todas las escrituras en memoria que realiza almacenan los valores correctos en el orden correcto. Por lo tanto, resulta natural utilizar las instrucciones de almacenamiento como puntos de sincronización y comparación. Sin embargo, no alcanza sólo con esto; debido a posibles errores en direcciones de destino de instrucciones de salto, se puede producir que algunas escrituras sean salteadas, que se ejecuten escrituras que no deberían ejecutarse o que se produzcan valores incorrectos que posteriormente sean escritos. Por lo tanto, las instrucciones de salto también constituyen puntos de sincronización en los que deben compararse los valores redundantes.

EDDI (*Error Detection by Duplicated Instructions in Superscalar processors*) [31] es una técnica de detección que utiliza el compilador para duplicar todas las instrucciones del programa, insertando comprobaciones de validación apropiadas. En tanto que CFCSS (*Control-Flow Checking by Software Signatures*) [37] verifica explícitamente el flujo de control durante la ejecución. Cada transferencia de control genera una firma en tiempo de ejecución, que es validada por el código de comprobación generado por el compilador para cada bloque básico. Los autores de [36] desarrollaron la técnica ACFC (*Assertions for Control Flow Checking*), que asigna paridad a cada bloque básico en ejecución y detecta los fallos como errores de paridad. En [52] se propone una estrategia para monitorear las firmas de software sin crear un grafo de flujo de control, pero utiliza hardware adicional. Un coprocesador computa dinámicamente la firma del flujo de instrucciones en ejecución y se usa un temporizador *watchdog* para detectar la ausencia de firmas.

SWIFT (*Software Implemented Fault Tolerance*) [4] es una transformación basada en el compilador que utiliza un solo hilo. SWIFT duplica las instrucciones de la aplicación e inserta comparaciones en puntos estratégicos durante la generación del código. En ejecución, los valores se computan dos veces y se comparan por equivalencia antes de que alguna diferencia debida a un fallo transitorio pueda afectar la salida del programa. Esta estrategia permite a los programadores tener flexibilidad en la política de tratamiento de los fallos. Por ejemplo, se podrían verificar sólo los segmentos de código esenciales o modificar el modo en que se manejan los fallos detectados. Además, la relación entre las instrucciones duplicadas, coordinada por el compilador, facilita la implementación de métodos para manejar las excepciones, las interrupciones y la memoria compartida, lo cual lo hace adecuado para utilizar tanto en monoprocesadores como en multiprocesadores.

SWIFT es una de las propuestas de detección más avanzadas de las que utilizan el compilador para implementar redundancia y detección de fallos transitorios. Aprovecha la existencia de ECC´s en el subsistema de memoria (principal y caches) para no duplicar el uso de memoria, y no requiere ningún hardware adicional. Mejora el mecanismo de comprobación de flujo de control, lo cual le permite eliminar validaciones innecesarias y reducir el *overhead*. De esta forma provee un alto grado de cobertura frente a fallos que producen errores tanto en los datos como en el flujo de control.

Los autores analizan una posible solución para evitar la duplicación de cargas de datos desde la memoria, la cual consiste en que el compilador realice una única lectura y luego duplique el valor leído para ser utilizado por las versiones original y redundante del programa. Esta es una alternativa de implementación sencilla, pero tiene el inconveniente de quitar la redundancia a las instrucciones de carga, transformándolas en puntos de falla centralizados.

Podría esperarse que, debido a la duplicación de instrucciones de un código con detección de fallos respecto del código original (equivalente a ejecutar el programa dos veces), el programa se ejecutara a la mitad de velocidad. Inclusive, la degradación debiera ser aún mayor, dado que se requiere código adicional para comparar y validar las salidas. Sin embargo, los resultados que obtienen los autores muestran que la eficiencia del procesador para planificar la ejecución de instrucciones, asignar recursos (que no son utilizados en la ejecución de la versión original) y explotar el paralelismo a nivel de instrucción para gestionar la detección, produce un *overhead* en el tiempo de ejecución de sólo un 41% en promedio respecto de la versión sin detección de fallos. Sus simulaciones son concordantes con los resultados de otros trabajos sobre técnicas de fiabilidad por software [7] que muestran que la degradación debida a la redundancia puede ser menor al doble esperable.

De esta manera, se puede integrar una técnica de fiabilidad por software al compilador para proveer detección utilizando el hardware comercialmente disponible.

Como desventaja, todas las técnicas basadas en el compilador impactan negativamente en la *performance* y requieren recompilación de las aplicaciones; por otra parte, es indispensable contar con el código fuente, lo que no se cumple en todos los casos de aplicaciones de interés [13].

Otra propuesta basada puramente en software es la que se encuentra en [12]. Los autores buscan una técnica posible de utilizar en aplicaciones espaciales, en las que se requiere un alto grado de cobertura pero también respuesta en tiempo real, por lo que su objetivo es minimizar el *overhead* introducido en el tiempo de ejecución sin sacrificar capacidad de detección. Para esto, proponen métodos alternativos para detección de errores de flujo de control y de errores de datos, obteniendo cobertura ante la mayoría de los fallos a costa de incrementos menores al 200%, tanto en el tiempo de ejecución como en la ocupación de memoria. Cabe aclarar que adoptan un modelo de fallo particular: los *bit-flips* pueden producirse en los segmentos de código, pila y datos, además de algunos registros.

La técnica propuesta para detección de errores de flujo de control se denomina RSCFC (*Relationship Signatures for Control Flow Checking*). Al igual que otros mecanismos de verificación del flujo de control, se basa en la división del programa en bloques básicos. A cada bloque básico se le asigna una firma, en la que se encuentra codificada la relación entre bloques (es decir, las transferencias de información entre los bloques). La ocurrencia de errores de flujo de control se detecta realizando operaciones entre la firma calculada en tiempo de ejecución y la información de ubicación del bloque actual, con instrucciones adicionales introducidas al principio y al final de cada bloque.

La estrategia de detección de errores en los datos consiste en duplicar las variables y las instrucciones que operan sobre los datos, sin tomar en cuenta las ubicaciones relativas en memoria entre los originales y las réplicas y sin añadir instrucciones de neutralización (esta es la causa de que el grado de cobertura frente a fallos simples no es 100%). La idea básica de este mecanismo es definir relaciones de interdependencia entre las variables del programa y clasificarlas, según su rol, en variables intermedias, que son utilizadas para el cálculo de otras variables, y variables finales, que no forman parte de ningún cálculo posterior. Una vez establecidas las relaciones, se duplican todas las variables y cada operación realizada sobre una variable original es repetida sobre su réplica. Después de cada escritura de una variable final se introduce una verificación de consistencia entre las variables duplicadas. En tanto, para los fallos que afectan la condición que se evalúa en cada instrucción de salto del programa, se vuelve a comprobar la condición en el bloque resultante de destino del salto. En los llamados a funciones, todos los parámetros se duplican, e internamente a la función se adoptan las mismas reglas descriptas.

La evaluación que realizan los autores de esta propuesta de detección por software resulta en un *overhead* aproximado de 1.53 (en promedio) respecto del tiempo de ejecución del programa original y un factor de 1.83 respecto de la cantidad de memoria utilizada. En tanto, la evaluación realizada respecto del grado de cobertura muestra una capacidad de detección del 99.7% de los fallos inyectados en el segmento de datos y del 98.8% de los fallos inyectados en el segmento de código.

Un parámetro significativo es la cantidad de salidas incorrectas no detectadas durante las pruebas (que es la suma de los errores que causan SDC y TO). En esta propuesta, para los fallos inyectados en el segmento de datos, las salidas incorrectas no detectadas son el 0.12% en promedio, mientras que para el segmento de código son el 0.78% en promedio. El sistema operativo detecta una gran cantidad de fallos inyectados (en especial en el segmento de código) debido a que esos fallos resultan en instrucciones inválidas, saltos a direcciones ilegales o divisiones por cero, que activan mecanismos de protección del hardware o el sistema operativo; esta activación es siempre anterior al mecanismo de detección.

Otra clase de técnicas de detección de fallos transitorios son las que utilizan replicación de procesos. En ellas, se crea un conjunto redundante de procesos por cada proceso de la aplicación original y se comparan sus salidas para garantizar una correcta ejecución.

Entre estas técnicas, se destaca la propuesta PLR (*Process-Level Redundancy*) [13]. En ella, se permite al sistema operativo planificar libremente los procesos redundantes, por lo que se aprovecha inteligentemente el paralelismo de recursos de hardware disponibles. Por lo tanto, PLR escala con la tendencia arquitectural hacia las grandes máquinas multicore y *manycore* de propósito general, logrando mejorar la cobertura y el rendimiento sin agregar ningún hardware ni modificar el sistema. Frente al problema de la utilización ineficiente de un gran número de cores, por parte de aplicaciones que no pueden explotar todos ellos para obtener mayores prestaciones, PLR provee una alternativa de aprovechamiento en beneficio del sistema. Además, se presenta un prototipo real que se ejecuta en conjunto con programas

serie de forma transparente a la aplicación, sin realizarle ningún tipo de modificación, incorporando la detección de fallos a costa de introducir un 16.9 % de *overhead* en el tiempo de ejecución.

Uno de los aportes significativos de esta propuesta es que se centra en garantizar la ejecución correcta de las aplicaciones, y no en garantizar la ejecución sin errores que afecten componentes de hardware. En concordancia con [4], se muestra que una alta proporción de los fallos que ocurren (y que serían detectados por técnicas basadas en hardware) resultan en errores latentes, por lo que pueden ser ignorados (desde el punto de vista de la detección) sin riesgo alguno. Cuando un error latente es detectado, se produce un DUE falso (ver figura 1.5, sección 1.6), que produce una invocación inútil al mecanismo de recuperación o una parada segura. Los autores de [13] muestran que su herramienta es capaz de ignorar los fallos que producen errores latentes, mediante una campaña de inyección de fallos.

En particular, el trabajo muestra la manera en que los fallos que afectan a los registros de CPU se propagan desde la perspectiva de la aplicación. Mientras que muchos de ellos resultan en errores latentes, otros se propagan a través de cientos o miles de instrucciones.

Como se mencionó anteriormente, una de las características salientes de PLR es la transparencia al usuario y a la aplicación (no para el sistema, que es consciente de la existencia de las réplicas). A pesar de que se crean procesos redundantes para cada proceso original de la aplicación, se mantiene la semántica para un proceso esperada por el usuario y por los demás procesos (en el caso de comunicación entre procesos o IPC); los procesos redundantes interactúan con el sistema como si sólo se ejecutara el proceso original, por lo que la incorporación de PLR pasa desapercibida para la aplicación, la cual no tiene que ser modificada ni recompilada. PLR se ejecuta en espacio de usuario, por lo que no se requiere modificación del sistema operativo ni ninguna capa inferior. Esta característica lo hace flexible, de modo que las aplicaciones con requerimientos de fiabilidad pueden utilizarlo y otras no, de manera similar a lo propuesto en [48].

PLR replica el espacio de direcciones virtuales completo, así como también los metadatos de un proceso (p. ej. los descriptores de archivo); el mecanismo debe coordinar los procesos redundantes para mantener el determinismo entre réplicas, por lo que se proponen estrategias para manejar determinísticamente señales asíncronas y memoria compartida, y se evalúa su impacto sobre el desempeño del sistema. Los procesos redundantes emulan las *system calls* y las operaciones de E/S (sólo el proceso original las realiza efectivamente); previamente, se verifica que todas las réplicas deben realizar la misma *system call*, de forma de detectar posibles errores en el flujo de control. Todas las entradas del proceso original se replican y comunican a los procesos redundantes por memoria compartida; de la misma forma, la comparación de valores de salida se realiza colocándolos previamente en un segmento de memoria compartida. Para disminuir el *overhead*, se calculan CRC´s y se comparan en memoria compartida cuando los *buffers* de escritura son muy grandes. Se incorpora un *watchdog timer* para detectar los fallos que producen que uno de los procesos redundantes no responda dentro de un lapso de tiempo definido por el usuario (errores TO, sección 1.8.3).

El conjunto de procesos redundantes consta de dos procesos para poder detectar un fallo simple, o de tres procesos para obtener capacidad de recuperación; en este caso, al detectar un fallo entre dos procesos, se compara con el tercero para obtener una mayoría. Una vez garantizado el resultado correcto, el proceso que falló se elimina y se crea otro redundante para reemplazarlo.

Los resultados de una campaña de inyección de fallos muestran la eficacia de PLR para detectar fallos transitorios por software. Existen 6 posibles salidas para el programa con los fallos inyectados cuando se ejecuta PLR:

1. Correcto: el fallo es benigno, es decir, permanece latente y el programa se ejecuta correctamente.

2. Incorrecto: el fallo produce SDC, por lo que el programa finaliza con un código de retorno correcto, pero sus resultados son incorrectos.
3. Abortado: el fallo produce un DUE, por lo que el programa finaliza con un código de retorno inválido.
4. Fracaso: el fallo produce un DUE que provoca la finalización abrupta de la ejecución (p. ej. *segmentation fault*).
5. Discordancia: ocurre cuando PLR detecta una diferencia al comparar las salidas.
6. SigHandler: un manejador de señales de PLR detecta la condición de finalización del programa.

Los autores ignoran deliberadamente las salidas que se detectan por *time-out*, ya que sólo ocurren en un 0.05% de las pruebas.

PLR detecta todos los fallos que producen resultados Incorrectos y Abortados mediante comparación de las salidas, transformándolos en Discordancias. También detecta los fallos que producen Fracasos y los transforma en SigHandler. El enfoque centrado en la ejecución correcta a nivel del software produce que la detección esté basada en los efectos de los fallos durante la ejecución. Por el contrario, SWIFT detecta aproximadamente un 70% de los fallos que producen salidas Correctas, de manera inútil. Sin embargo, en algunas circunstancias particulares, PLR detecta algunos pocos falsos positivos. En la figura 2.1 se muestran resultados obtenidos de inyección de fallos y detección para diferentes aplicaciones de prueba (*benchmarks*)

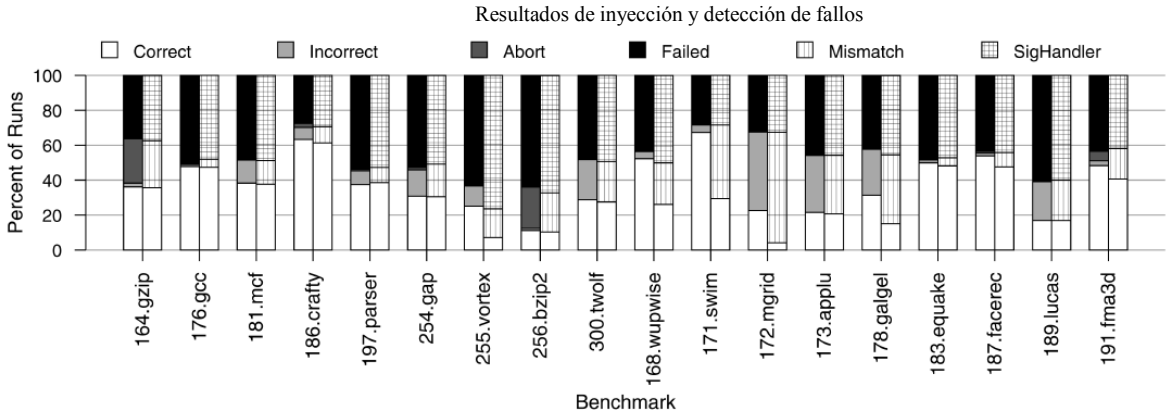

Fig. 2.1: Resultados de una campaña de inyección. La barra izquierda de cada grupo muestra las salidas sólo con inyección de fallos y la barra derecha muestra el desglose de cómo PLR detecta los fallos

Los autores realizan también un análisis sobre la latencia de detección, realizando mediciones de la cantidad de instrucciones que se ejecutan entre la inyección del fallo y su detección. La tendencia muestra que los fallos que se detectan por Discordancia tienen latencias elevadas, en tanto que los que se detectan con el manejador de señales tienen mayor probabilidad de ser detectados rápidamente. De todas formas, la perspectiva de detección basada en la ejecución correcta a nivel de software produce que los fallos permanezcan en estado latente un lapso de tiempo no acotado.

También se realiza un análisis sobre los factores que contribuyen al *overhead*. En primer lugar, se concluye que el nivel de optimización realizado durante la compilación incide negativamente en la performance: los binarios optimizados estresan más al sistema, por lo que producen valores de *overhead* mayores. Las dos fuentes que contribuyen al *overhead* de PLR son la competencia por los recursos de hardware entre los procesos redundantes y el *overhead* debido

a la emulación de llamadas al sistema, sincronización entre réplicas, copias de datos a memoria compartida para verificación e introducción de comprobaciones para detectar los fallos.

La solución propuesta en [13] es similar a un conjunto de trabajos que analizan la utilización de réplicas para tolerancia a fallos [15,53,54]. El enfoque de estos trabajos está centrado en fallos severos, como fallos permanentes de hardware o fallos en la fuente de alimentación, y asume una estrategia de parada segura [55], en la que el procesador se detiene ante la ocurrencia del fallo. Estas presunciones no se aplican al caso de los fallos transitorios.

2.5 Esfera de Replicación (SoR)

La esfera de replicación [38] (SoR, *Sphere of Replication*) es un concepto comúnmente aceptado para describir el dominio lógico de ejecución redundante de una técnica particular, y especificar los límites para la detección de fallos. El concepto de SoR abstrae la redundancia física de un sistema con *lockstepping* y la redundancia lógica de un procesador con hilos redundantes en el hardware. Todos los datos que ingresan en la SoR son replicados y toda la ejecución dentro del ámbito de la SoR es redundante de alguna forma. Antes de abandonar la SoR, todos los datos de salida son comparados para asegurar su corrección. Toda la ejecución por fuera de la SoR no está cubierta por la técnica particular de detección de fallos transitorios y debe ser protegida de otra manera (por ejemplo la utilización de ECC´s). Así, los fallos quedan confinados dentro de la frontera de la SoR y son detectados en los datos que dejan la SoR. En la figura 2.2 se muestra gráficamente el concepto de SoR.

La correcta identificación de la SoR es útil para determinar el conjunto de mecanismos de replicación y comparación necesarios y el alcance de la cobertura frente a fallos. Desde el punto de vista del diseñador del procesador, el hecho de variar la extensión de la SoR afecta la complejidad y el costo del hardware requerido.

El concepto original de SoR se utilizó para definir los límites de la fiabilidad en diseños de hardware redundante. Este modelo tradicional es denominado modelo de detección de fallos centrado en el hardware. Este modelo ve al sistema como una colección de componentes de hardware que deben ser protegidos de los fallos transitorios. La SoR está situada alrededor de las unidades de hardware especificadas. Todos los componentes dentro de la SoR están cubiertos por la ejecución redundante. Los valores que ingresan y salen de la SoR requieren de replicación y comparación, respectivamente.

El modelo de detección centrado en el hardware es apropiado para las técnicas basadas en redundancia de hardware; sin embargo, resulta incómoda la aplicación de una SoR centrada en el hardware para las propuestas de detección implementadas puramente en software. El software opera naturalmente a un nivel diferente, a pesar de lo cual algunas soluciones que utilizan el compilador para insertar instrucciones redundantes han intentado imitar una SoR centrada en el hardware. Por ejemplo, SWIFT [4] coloca su SoR alrededor del procesador, como se muestra en la figura 2.3(a), mientras las SoR de EDDI [31] está alrededor del procesador y el subsistema de memoria completo. De hecho, esta es la principal diferencia entre ambas propuestas. SWIFT coloca la memoria afuera de la SoR porque sus estructuras cuentan con esquemas de protección por hardware, como los bits de paridad y los ECC´s. Esto tiene la ventaja de utilizar la mitad de la memoria y de reducir a la mitad la cantidad de escrituras a memoria, lo cual, sumado a las restantes optimizaciones, justifican las ventajas de SWIFT sobre EDDI.

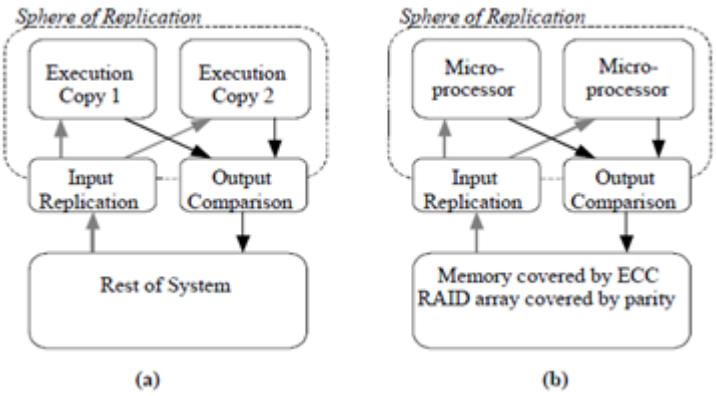

Fig. 2.2: Esfera de replicación: (a) General (b) Del sistema Compaq Non-Stop Himalaya

Sin embargo, SWIFT no puede controlar la duplicación a nivel del hardware, sino que sólo duplica las instrucciones a ejecutar. Cada lectura desde la memoria se realiza dos veces para replicar las entradas y todo el cómputo se realiza dos veces sobre las entradas replicadas. Se comparan los datos de salida antes de ejecutar cada instrucción de escritura en memoria para verificar su corrección. Esta solución funciona porque se puede emular la redundancia del procesador con redundancia de instrucciones. Sin embargo, otras SoR centradas en hardware no se pueden emular en software. Por ejemplo, no se puede implementar una SoR alrededor de las cachés sólo por medio de software. En general, este inconveniente se produce en las estrategias de detección que se implementan por software pero cuyo enfoque está centrado en la protección del hardware frente a los fallos.

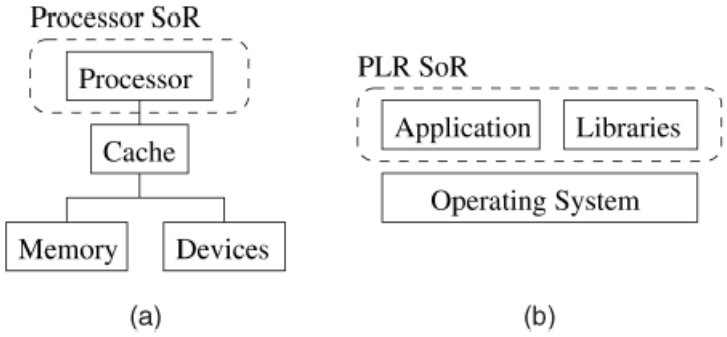

Fig. 2.3(a): Esfera de replicación de SWIFT (centrada en el hardware) - (b): Esfera de replicación de PLR (centrada en el software)

El paradigma de detección de fallos centrado en el software ve al sistema como un conjunto de capas de software que deben ejecutarse correctamente, por lo que debe utilizar una SoR ubicada alrededor de estas capas de software, en lugar de alrededor de componentes de hardware. El software permite analizar los fallos desde una perspectiva más amplia, determinando sus efectos sobre la ejecución de las aplicaciones. PLR [13], como técnica implementada en software, define su SoR en términos de software porque esto provee límites más naturales para la detección. Por otro lado, una SoR centrada en el software pone de manifiesto el hecho de que, aunque los fallos ocurren a nivel de hardware, los únicos fallos que tienen importancia son los que afectan la corrección del software que se ejecuta. De esta forma, los fallos

que permanecen latentes se pueden ignorar sin correr ningún riesgo. Por lo tanto, un sistema de detección (sin recuperación) con una SoR centrada en software se protege de los fallos que se propagarían como salidas incorrectas de las aplicaciones (DUE´s). En tanto, un sistema centrado en software con detección y recuperación no invocará al mecanismo de recuperación para los fallos que no afecten la corrección del software.

Una SoR centrada en software actúa de la misma forma que una centrada en hardware, es decir, todas las entradas son replicadas, la ejecución dentro de los límites de la SoR es redundante, y todos los datos que salen de ella son verificados. La figura 2.3(b) muestra la SoR utilizada por PLR, que está ubicada alrededor del espacio de direcciones de las aplicaciones del usuario y las bibliotecas compartidas asociadas a ellas, proveyendo redundancia a nivel de procesos. Por lo tanto, toda la ejecución que se realiza en espacio de usuario es duplicada, y los fallos se detectan sólo si resultan en datos incorrectos saliendo del espacio de usuario, lo que permite ignorar la gran mayoría de los fallos benignos. PLR replica el código de la aplicación, de las bibliotecas, las variables globales, la *heap*, la pila, los descriptores de archivos y, en general, toda la información asociada al proceso. Todo lo que está por fuera de los límites de la SoR (por ejemplo, el Sistema Operativo) debe ser protegido por otros medios.

PLR utiliza un modelo en el cual hay un único proceso visible ("maestro") para mantener la semántica de procesos esperada por el usuario, y que recibe los datos de los demás, y un conjunto de procesos "esclavos", que son las réplicas que realizan el cómputo útil. La utilización de una SoR centrada en el software implica que todos los datos que se comunican entre los procesos son recibidos una única vez por el proceso maestro. Durante la emulación de llamadas al sistema, todos los datos leídos (como una lectura a un descriptor de archivo) se replican y copian a los procesos esclavos. Los valores de retorno de las llamadas al sistema se consideran datos de entrada y también se replican.

Por otra parte, todos los datos de salida de los procesos redundantes se deben verificar antes de permitir su salida de la SoR. Si las comparaciones arrojan diferencias entre los datos, se asume que se ha detectado un fallo y se invoca a una rutina de recuperación. Durante la emulación de llamadas al sistema, todos los *buffers* de escritura cuyos datos van a trasponer las fronteras de la SoR deben ser verificados. Todos los datos que se pasan como parámetros a las llamadas al sistema se consideran salidas que abandonan la SoR y también deben ser verificados para garantizar la corrección del programa.

Sin embargo, el modelo de detección centrado en el software tiene la desventaja de retardar la detección de los fallos hasta que se produce certeza del error por ejecución incorrecta del programa o por datos inválidos que salen de la SoR. La detección retardada implica que un fallo puede permanecer latente durante la ejecución durante un período indeterminado de tiempo. Los autores proponen, como trabajo futuro, caracterizar la propagación de los fallos y explorar métodos para limitar los lapsos en los que los fallos permanecen sin ser detectados.

2.6 Ventanas de vulnerabilidad

Todas las técnicas de tolerancia a fallos tienen ventanas de vulnerabilidad, es decir, circunstancias en las cuales los fallos (que influyen efectivamente sobre la ejecución, no los fallos latentes) pasan desapercibidos. Las características de diseño de una estrategia de detección, sumadas a las pruebas y evaluaciones a las que son sometidos los sistemas (usualmente mediante inyección de fallos) deben conducir a los autores de las propuestas a describir explícitamente esas ventanas de vulnerabilidad, es decir, los casos en que es conocido que la técnica no será capaz de detectar los fallos.

Las ventanas de vulnerabilidad están normalmente asociadas con fallos que afectan al mecanismo de detección propiamente dicho. Dado que la detección se realiza mediante comparaciones, los sitios en los cuales estas comparacio-

nes se realizan constituyen puntos de falla centralizados. Sin embargo, a pesar de no ser completamente fiable, la redundancia parcial [44,56] en general es suficiente para mejorar la fiabilidad y cubrir los requerimientos de los usuarios.

La técnica SWIFT tiene dos puntos primarios de falla que resultan en ventanas de vulnerabilidad. Debido a que la redundancia se obtiene únicamente por medio de duplicar instrucciones, puede existir un retardo entre la validación y el momento de utilización de los valores de los registros validados. Por lo tanto, cualquier interferencia que ocurra durante ese lapso puede corromper el estado. Mientras que todas las demás instrucciones tienen alguna forma de redundancia para protegerse contra los fallos, los *bit-flips* que ocurren en las direcciones de escritura y en registros de datos no se detectan. Esto puede causar ejecución incorrecta de los programas debido a que implica que escrituras incorrectas salgan de la SoR, ya sea por valores de escritura incorrectos o por direcciones de escritura incorrectas.

El segundo punto de falla ocurre si el código de operación de una instrucción es cambiado al de una escritura a causa de un fallo transitorio. Estas escrituras no están protegidas debido a que el compilador no ve esa instrucción. La escritura resultante se ejecutará libremente y el valor almacenado por ella abandonará la SoR.

Debido a que los llamados a funciones afectan la salida de los programas, los valores inválidos de parámetros pueden resultar en ejecuciones incorrectas. SWIFT resuelve esto haciendo que los llamados a funciones sean puntos de sincronización. Antes de cada llamado a función, todos los operandos de entrada se verifican con sus copias redundantes de la manera habitual. Si no hay diferencia, las versiones originales se pasan como parámetros a la función. Al comienzo de la función, los parámetros deben ser duplicados nuevamente, debido a que la función también se computa por duplicado. Al terminar, sólo se devuelve una versión de los valores de retorno. Estos valores deben duplicarse para el código redundante restante por fuera de la función.

Todo esto produce *overhead* adicional, pero también introduce puntos de vulnerabilidad. Como sólo una versión de los parámetros se le pasa a la función, un fallo que afecta a un parámetro luego de la verificación realizada por el programa que llama a la función, pero antes de la duplicación que se realiza en la función llamada no será detectado.

Para manejar de forma eficiente los llamados a funciones, la convención puede modificarse para pasar varios conjuntos de argumentos a una función y retornar varios valores desde la función. Sólo los argumentos que se pasan por registro necesitan ser duplicados (debido a que los argumentos que se pasan por memoria están fuera de los límites de la SoR). El hecho de existan varios valores de retorno requiere de la reserva de un registro adicional para el valor de retorno replicado. Lograr esto tiene el costo de requerir el doble de registros para valores de entrada y de salida, pero garantiza que se mantiene la detección de fallos a través de los llamados a función.

En tanto, PLR tiene asociadas sus ventanas de vulnerabilidad a los fallos que ocurren durante la ejecución del código de PLR, que pueden causar errores irrecuperables. Por otra parte, si un fallo causa un salto erróneo al código de PLR puede obtenerse una conducta indefinida. Finalmente, PLR no protege al sistema operativo, por lo que cualquier fallo que afecte la ejecución del sistema operativo resultará en un error. La primera y la tercera de las ventanas de vulnerabilidad pueden reducirse si el sistema operativo y el código de PLR se compilan con alguna solución para tolerancia a fallos transitorios basada en el compilador. En tanto, el requerimiento de mantener la semántica de los procesos, el proceso líder debe mantenerse activo durante toda la ejecución de la aplicación. A pesar de representar un punto de falla centralizado, el proceso líder no realiza trabajo real útil, por lo que la probabilidad de que de un fallo transitorio afecte su ejecución es muy baja.

La tabla 2.1 muestra una comparación entre varias soluciones redundantes para detección de fallos. Los encabezados de las columnas muestran las diferentes entidades lógicas que deben protegerse. Las filas muestran contienen detalles de cada técnica. La palabra “todo” que aparece en algunas celdas significa que la técnica de una fila determinada

protege completamente al estado lógico de la columna correspondiente. La palabra "ninguno" significa que la técnica no brinda ninguna protección, por lo que se asume que se debe proteger de alguna forma desde afuera. Las palabras "alguno" y "la mayoría" son niveles de protección intermedios, en los que la técnica protege un subconjunto del estado durante una fracción del tiempo. Las notas al pie brindan detalles adicionales.

Técnica	Categoría	Código de Operación	Loads	Stores	Transferencias de Control	Otras Instrucciones	Estado de Memoria	Costo de Hardware
DIVA	HW	Todo	Todo	Todo	Todo	Todo	Ninguno	Procesador adicional
Himalaya	HW	Todo	Todo	Todo	Todo	Todo	Ninguno	Dualcore, comparador
RMT	HW	Todo	Todo	Todo	Todo	Todo	Ninguno	SMT, comparador, lógica de sincronización
CRT	HW	Todo	Todo	Todo	Todo	Todo	Ninguno	CMP, comparador, lógica de sincronización
Superscalar	HW	La mayoría[a]	La mayoría[a]	La mayoría[a]	La mayoría[a]	La mayoría[a]	Ninguno	Replicador, lógica extra
CFCSS	SW	Alguno[b]	Ninguno	Ninguno	La mayoría[c]	Ninguno	Ninguno	Ninguno
EDDI	SW	La mayoría[d]	Todo	Todo	La mayoría[e]	Todo	Todo	Ninguno
ACFC	SW	Alguno[b]	Ninguno	Ninguno	La mayoría[c]	Ninguno	Ninguno	Ninguno
SWIFT	SW	La mayoría[f]	Todo	La mayoría[c]	La mayoría[c]	Todo	Ninguno	Ninguno

[a] Los fallos en el replicador de instrucciones y en los registros no se detectan

[b] Sólo cobertura para códigos de operación de instrucciones de salto

[c] Transferencias de control incorrectas dentro de un bloque de control pueden no ser detectadas bajo ciertas circunstancias infrecuentes

[d] No hay cobertura para los códigos de operación de instrucciones de salto ni para códigos de operación que difieren de los de saltos en una distancia de Hamming de 1

[e] No se detectan alteraciones a operandos entre la validación y su utilización por una unidad funcional

[f] No hay cobertura para los códigos de operación de instrucciones de almacenamiento ni para códigos de operación que difieren de los de almacenamiento en una distancia de Hamming de 1

Tabla 2.1: comparación entre soluciones redundantes para detección de fallos

2.7 Fallos múltiples

En la sección 2.1 se mencionó el hecho de que la técnica que se presenta en este trabajo, y también el resto de las que se evalúan a lo largo de él, son capaces de detectar fallos si se asume el modelo SEU, en el que ocurre un único *bit-flip* a lo largo de la ejecución. Sin embargo, no son tan efectivas si ocurren fallos que afectan a múltiples bits.

Los autores de [4] realizan un análisis sobre los fallos multibit, en el cual se describen las dos únicas situaciones en las que los fallos múltiples se pueden combinar para causar inconvenientes.

La primera de ellas es aquella en la que el mismo bit resulta alterado tanto en el cómputo original como en el redundante, por lo que la comparación resulta correcta y el fallo no se detecta. La segunda se da cuando ocurre un fallo que afecta a uno de ambos hilos de cómputo redundante, y el resultado de la operación de verificación también es alterado, por lo que no se ejecuta el código de tratamiento del error. Por fortuna, estas combinaciones tienen una probabilidad de ocurrencia sumamente baja, por lo que pueden ser ignorados sin riesgos serios. Todas las demás combinaciones de fallos múltiples son detectadas como fallos simples, ni bien se realice la primera comprobación que contenga al menos una diferencia.

En el trabajo [4] se realiza una estimación de la probabilidad de que ocurran las combinaciones de fallos multibit que escaparían de los mecanismos de detección. Si se utiliza un modelo en el que pueden ocurrir dos fallos (uno en el cómputo original y otro en el redundante), y se asume que el mismo fallo debe ocurrir en el mismo bit de la misma instrucción de ambas ejecuciones redundantes para no ser detectado, la probabilidad viene dada por la ecuación 7:

$$P\ (error_{redundante}\ |error_{original}\ ) = \frac{1}{64} * \frac{1}{nº\ instrucciones}$$

Ecuación 7: probabilidad de que un fallo doble en ambos hilos de cómputo redundantes no sea detectado

Esta es simplemente la probabilidad de que resulte elegida la misma instrucción (en una inyección de dos fallos distribuidos de forma uniforme y aleatoria) combinada con la probabilidad de que resulte elegido el mismo bit en ambas ejecuciones (asumiendo registros de 64 bits). En el caso de prueba del trabajo, en el que se usa la suite de *benchmarks* SPEC, la cantidad promedio de instrucciones dinámicas está en el orden de 10^9 a 10^{11}, por lo que la probabilidad de que ocurra este caso particular de fallo doble es cercana a uno en un trillón.

El otro caso es que un bit resulte alterado, afectando sólo a uno de los hilos de cómputo redundante, y otro bit resulte invertido durante la operación de comprobación. Si se asume que se realiza una única comparación para cada fallo posible, la probabilidad de error viene dada por la ecuación 8:

$$P\ (error_{redundante}\ |error_{original}\ ) = \frac{1}{nº\ instrucciones}$$

Ecuación 8: probabilidad de que un fallo doble, en un hilo de cómputo y en la comprobación, no sea detectado

Para los *benchmarks* SPEC, esta probabilidad es cercana a uno en 10 billones en promedio. Esta es una sobreestimación grosera, debido a que se asume que hay una única verificación para cada fallo, cuando en la práctica pueden existir varias comprobaciones sobre un valor erróneo que sea utilizado en el cálculo de distintas cantidades a almacenar o direcciones de saltos.

2.8 Memoria compartida

Cuando múltiples procesos se comunican utilizando memoria compartida, el compilador no puede forzar un ordenamiento de las lecturas y las escrituras entre los procesos. Por lo tanto, las dos lecturas de una lectura duplicada no devuelven necesariamente el mismo valor, debido a que siempre existe la posibilidad de que se intercalen escrituras de otros procesos. La situación es similar cuando ocurre una interrupción o una excepción entre el par de lecturas de una duplicación, y el manejador de interrupciones o excepciones modifica el contenido en la dirección de la lectura. Estas circunstancias no afectan la cobertura frente a fallos de ningún sistema, pero son capaces de incrementar la cantidad de fallos detectados que no hubieran causado ninguna avería (tanto para monoprocesador como para multiprocesador).

Para solucionar esto, se puede recurrir a alguna técnica segura (basada en hardware) de duplicación de valores leídos, como las que se utilizan en máquinas RMT [10], y adaptarla a los esquemas basados en software, pero introducir estructuras de hardware redundantes tiene las desventajas y los costos mencionados en la sección 2.4.

2.9 Propuestas híbridas

En un intento de combinar las mejores características de las propuestas basadas en hardware y las técnicas de software puro, algunos autores han propuesto soluciones híbridas entre el hardware y el software, que consisten en mecanismos robustos de tolerancia implementados en hardware, pero controlados por el software que se ejecuta en el procesador [7].

Existen diferentes aproximaciones híbridas a la tolerancia a fallos transitorios. Algunas de ellas en enfocan exclusivamente en la protección del flujo de control [52], mientras que otras buscan la protección completa del procesador [39].

En [7], los autores proponen TALFT, un lenguaje ensamblador escrito para ser tolerante a fallos. El trabajo se enfoca en proveer un marco teórico y probar formal y rigurosamente las propiedades que garantizan la corrección y fiabilidad del código máquina.

La solución propuesta utiliza una combinación de software y hardware para lograr la protección para el flujo de control, similar a los procesadores *watchdog* [33], pero que hace explícitas ambas versiones de la protección, como en las estrategias basadas únicamente en software [32,37]

El objetivo principal de un sistema de tipo TALFT es asegurar que los programas escritos correctamente (siguiendo las reglas propuestas en el trabajo) mantienen un comportamiento seguro en presencia de fallos transitorios. En otras palabras, los programas escritos con un lenguaje ensamblador tolerante a fallos deben garantizar que los dispositivos de Entrada/Salida mapeados en memoria nunca pueden leer un valor erróneo y hacerlo visible para el usuario. De hecho, los autores hacen foco en que, independientemente del tipo de implementación (software puro, hardware o híbrida), ninguna otra solución comprueba, mediante un marco teórico formal, la robustez del código máquina para garantizar la fiabilidad y tolerancia a fallos, la cual es la principal diferencia con todo el trabajo relacionado restante.

Capítulo 3

Arquitectura cluster de multicores

En este capítulo se describen las arquitecturas paralelas de clusters y su correspondiente evolución a cluster de multicores.

3.1 Clusters

El término cluster se aplica a un conjunto de computadoras, construido mediante la utilización de componentes de hardware estándar, y que se encuentran interconectadas de manera de que se comportan como si fuesen una sola computadora, trabajando cooperativamente como un único recurso de cómputo [57]. Estas arquitecturas se utilizan en el ámbito del cómputo paralelo y distribuido, y juegan un papel importante en la solución de problemas de las ciencias, las ingenierías y del comercio moderno [58]. La tecnología de los clusters ha evolucionado debido a los requerimientos de diversas actividades que se apoyan en ellos, y que van desde aplicaciones de supercómputo y software de misión crítica hasta servidores web, de comercio electrónico y hasta bases de datos de alto rendimiento, entre otros usos.

Los clusters constituyen una plataforma de cómputo paralelo muy utilizada por sus ventajas en cuanto a la relación costo/rendimiento, lo que significa que se puede obtener una potencia de cómputo significativa a un costo relativamente bajo.

La construcción de los nodos de un cluster es relativamente fácil y económica debido a su flexibilidad, lográndose una buena relación costo/rendimiento y al mismo tiempo una considerable escalabilidad [59]. Todos los nodos del cluster pueden tener la misma configuración de hardware y sistema operativo (cluster homogéneo), o tener diferente hardware y/o sistema operativo (cluster heterogéneo). Esta característica constituye un factor importante en el análisis del rendimiento que se puede obtener de un cluster como máquina paralela [60].

El cómputo con clusters surge como resultado de la convergencia de varias tendencias, que incluyen la disponibilidad de microprocesadores económicos de alto rendimiento y redes de alta velocidad, el desarrollo de herramientas de software para cómputo distribuido de alto rendimiento, y la creciente demanda de potencia computacional para determinados tipos de aplicaciones [58]. Se espera que un cluster sea capaz de proveer servicios como alto rendimiento (*High Performance*), alta disponibilidad (*High Availability*), balance de carga (*Load Balancing*) y escalabilidad (*Scalability*).

Para que un cluster funcione como tal, además de la interconexión entre las computadoras que forman parte de él, es necesario un sistema de gestión del cluster, el cual se encargue de interactuar con el usuario y con los procesos que se ejecutan para optimizar el funcionamiento. Los componentes de un cluster son los nodos, el sistema operativo, la red de interconexión, el middleware (capa de abstracción entre el usuario y el sistema operativo), los protocolos de comunicación y servicios, y las aplicaciones (que pueden ser paralelas o no).

Cada procesador que forma parte de un cluster tiene su propia memoria local. La memoria del sistema se encuentra distribuida entre todos los procesadores y cada procesador puede acceder directamente sólo a su memoria local. El acceso a las memorias de los demás procesadores es indirecto, realizándose normalmente mediante paso de mensajes.

Este acceso local y privado a la memoria es el principal aspecto que diferencia los multicomputadores de los multiprocesadores. Las transferencias de datos se realizan a través de la red de interconexión que conecta un subconjunto de procesadores con otro subconjunto.

De lo anterior se puede concluir que las arquitecturas de cluster presentan dos características importantes que deben ser tenidas en cuenta en el momento de utilizarlas para realizar cómputo paralelo:

I. Los nodos que forman el cluster están débilmente acoplados. Esto tiene las siguientes implicaciones:

   i. La memoria física de las computadoras en la red local está totalmente distribuida y no hay ningún medio de hardware que facilite compartirla.
   ii. El procesamiento en una red local es totalmente asincrónico en cada máquina, y no hay ningún medio de hardware que facilite la sincronización.
   iii. La única forma de hacer que un procesador acceda a información de otro nodo es por medio de la red de comunicaciones. De la misma manera, la única forma de sincronización entre los procesadores de cada computadora es utilizando la red de comunicaciones.
   iv. Descartar el uso de un espacio de direcciones común implica replantear algunos algoritmos diseñados para computadoras paralelas con memoria compartida.

II. El rendimiento de la red de interconexión se puede ver afectado por diferentes aspectos. Entre ellos se pueden mencionar:

   i. El tiempo propio de las comunicaciones es alto frente al tiempo de acceso a memoria.
   ii. El inicio de la comunicación entre dos procesadores tiene un tiempo de latencia (*startup*) que conlleva un costo.
   iii. El ancho de banda (normalmente total o parcialmente compartido) disponible para los procesadores de cada cluster es bajo.

Estas características producen que las aplicaciones que más se benefician de su ejecución en clusters sean aquellas que utilizan más la capacidad de cómputo local y que están diseñadas para solapar comunicaciones entre los procesadores y procesamiento de datos sobre cada procesador.

3.2 Clusters de multicores

En las últimas décadas, los procesadores mejoraron su rendimiento según la ley de Moore, fundamentalmente debido a dos factores tecnológicos. En primer lugar, el aumento de la frecuencia alcanzable por el reloj, y en segundo lugar, el creciente número de transistores que se pueden integrar en un chip. Estas circunstancias, sumadas a ciertas mejoras en los compiladores, incrementaron el número de instrucciones ejecutables por unidad de tiempo. Sin embargo, esta mejora se vio limitada por el incremento del calor generado y el consumo de energía, que resultaron insostenibles. La solución a este problema consistió en cambiar a un paradigma distinto en el diseño de microprocesadores: el procesador de múltiples núcleos [61,62].

Un procesador de múltiples núcleos (multicore) integra dos o más núcleos computacionales dentro de un único chip. Si bien estos núcleos son más simples y menos veloces, al combinarlos permiten mejorar el rendimiento global del procesador y al mismo tiempo hacerlo más eficiente energéticamente [63,64].

Una computadora multicore se puede ver como una máquina paralela compuesta por varios cores interconectados que comparten un mismo sistema de memoria. Un multicore está generalmente formado por **n** cores (o núcleos de procesamiento) y **m** módulos de memoria. La red de interconexión, permite a los diferentes cores acceder a los distintos módulos de memoria.

La incorporación de los procesadores multicore a las arquitecturas de clusters tradicionales ha llevado a una evolución reciente dentro de las arquitecturas paralelas, conocida como cluster de multicores [65]. En este tipo de arquitecturas, la comunicación entre las diferentes unidades de procesamiento es heterogénea [66]. Las comunicaciones pueden clasificarse en comunicaciones inter-nodo y comunicaciones intra-nodo.

Las comunicaciones inter-nodo se dan entre aquellos núcleos de procesamiento que residen en distintos nodos, mediante el envío de mensajes a través de la red de interconexión que une los nodos. Respecto de un procesador multicore, el cluster de multicores introduce a la memoria distribuida accesible vía red como un nivel más en la jerarquía de memoria.

Las comunicaciones intra-nodo se producen entre los núcleos que se encuentran dentro del mismo nodo a través de los diferentes niveles de memoria que comparten. De acuerdo a la ubicación de los núcleos, las comunicaciones intra-nodo pueden a su vez clasificarse en tres grupos: caché-compartida (*shared-cache*), *intra-socket* e *inter-socket*.

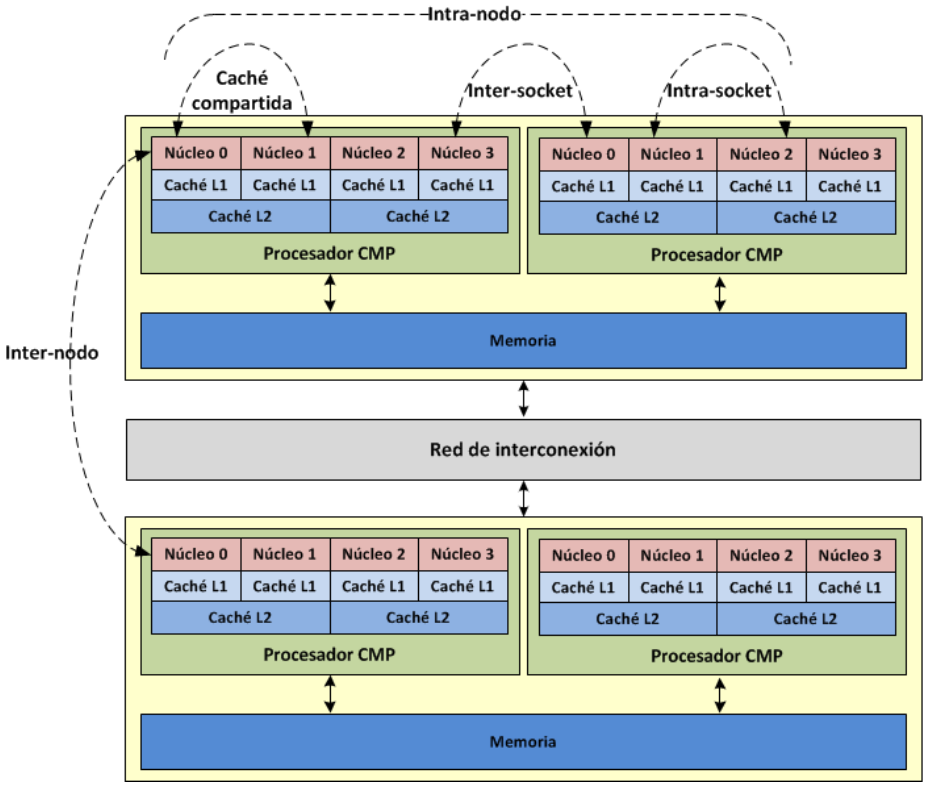

Figura 3.1: Esquema de un cluster de multicores con los diferentes tipos de comunicación que se pueden presentar.

Las comunicaciones caché-compartida se realizan entre núcleos que comparten algún nivel de memoria caché (por ejemplo: L2). Las comunicaciones *intra-socket* se producen entre núcleos que están dentro de un mismo socket pero que no comparten ningún nivel de caché. Por último, las comunicaciones inter-socket se dan entre núcleos que residen en el mismo nodo pero en diferentes sockets. En la Figura 3.1 se muestra un esquema de un cluster de multicores que incluye los diferentes tipos de comunicaciones que pueden presentarse.

Los clusters de multicores son arquitecturas híbridas [67], ya que combinan memoria compartida con memoria distribuida. Teniendo en cuenta que el aprovechamiento de la jerarquía de memoria presente en estas arquitecturas inci-de directamente sobre el rendimiento alcanzable por los mismos, se desarrolló un nuevo paradigma de programación paralela: el paradigma híbrido. Este paradigma combina características del modelo de memoria compartida con el de pasaje de mensajes. En la actualidad, combinaciones de la librería MPI junto a las librerías Pthreads u OpenMP son utilizadas para programar según este paradigma. El modelo de pasaje de mensajes, a través del estándar MPI, también puede ser utilizado para la programación de clusters de multicores. En este caso, los procesos que se ejecutan en el mismo nodo simulan memoria distribuida sobre memoria compartida.

Capítulo 4

Programación con paso de mensajes. Estándar MPI

En este capítulo se define el modelo de programación paralela de pasaje de mensajes. Además, se describe el estándar MPI que se utiliza para programar de acuerdo a dicho modelo.

4.1 Modelo de programación basado en paso de mensajes

En el ámbito del procesamiento paralelo, la elección de un modelo de programación tiene injerencia sobre las decisiones acerca del lenguaje de programación y de la biblioteca de comunicaciones que se van a utilizar. Tradicionalmente, han existido dos modelos principales: la memoria compartida y el pasaje de mensajes [67].

En el modelo de memoria compartida, todos los datos requeridos por la aplicación se encuentran en una memoria global accesible por todos los procesos que se ejecutan sobre la máquina paralela. Esto significa que cada proceso puede buscar y almacenar datos de cualquier posición de memoria independientemente de su ubicación física. Este modelo se caracteriza por requerir de sincronización para preservar la integridad de las estructuras de datos compartidas.

En cambio, en el modelo de pasaje de mensajes, los datos son vistos como si estuvieran asociados a un proceso particular. Los procesos no tienen acceso a ninguna forma de memoria compartida, sino que operan sobre espacios de direcciones disjuntos (cada uno tiene su propio espacio de direcciones privado). De esta manera, se necesita de intercambio explícito de mensajes entre procesadores para acceder a un dato remoto, es decir, el procesador dueño de ese dato debe enviar el dato y el procesador que lo requiere debe recibirlo. En este modelo, las primitivas de envío y recepción son las encargadas de manejar las comunicaciones y la sincronización. El modelo de pasaje de mensajes es el paradigma de programación paralela más natural para la utilización en clusters [67].

El intercambio de mensajes puede servir para diversos propósitos. El más evidente de ellos es el de compartir datos entre un emisor y un receptor que se conocen y cuya interacción fue planeada por el programador. Otra función es la de establecer una conexión entre un emisor y un receptor que no haya sido planeada de antemano (mediante una recepción anónima realizada por el receptor). Un tercer objetivo es la sincronización, que es un caso especial de comunicación. En las rutinas de emisión y recepción, el programador debe especificar detalles de bajo nivel, tales como la identidad del receptor, la dirección y tamaño de un *buffer* de memoria donde se depositan o del cual se extraen los datos, la longitud del mensaje, etc. [29].

El modelo de pasaje de mensajes es fácil de implementar en arquitecturas de memoria distribuida. Además, cuenta con dos grandes ventajas. La primera de ellas es la portabilidad. Debido a que el paso de mensajes fue establecido tempranamente para su operación en máquinas paralelas, se han desarrollado estándares bien definidos, lo que resulta en que prácticamente cualquier conjunto de máquinas puede ser utilizado para ejecutar programas desarrollados siguiendo este modelo. Esto redunda en una alta portabilidad del código, aunque no garantiza la portabilidad del rendimiento.

La segunda ventaja reside en el control explícito que tiene el programador sobre la ubicación de los datos en la memoria. Debido a que el acceso y el manejo de la memoria inciden directamente sobre el desempeño, el programador puede lograr rendimiento cercano al óptimo si dedica suficiente tiempo al ajuste de la aplicación a la arquitectura.

La principal desventaja del modelo de paso de mensajes es que impone una pesada carga al programador, debido a que éste es el responsable de manejar todos los detalles de ubicación de datos en la memoria, distribución de datos entre los procesos y ordenamiento de las sentencias de comunicación entre ellos [67].

La primera biblioteca de funciones de comunicación por pasaje de mensajes utilizada a gran escala fue PVM (*Parallel Virtual Machine*), la cual constituyó una pieza clave en el éxito de las arquitecturas de cluster como máquinas paralelas. Sin embargo, la dificultad para obtener un alto rendimiento y la no definición de un estándar terminaron produciendo su caída en desuso. Subsecuentemente, se definió la biblioteca MPI (*Message-Passing Interface*), la cual se ha convertido en la biblioteca estándar y, por lo tanto, la más utilizada [27]. Ambas son bibliotecas para utilizar en lenguajes comunes como C, C++ y Fortran, lo que las hace sencillas de manejar.

4.2 Estándar de programación MPI

MPI [68,69] es un protocolo de comunicaciones independiente del lenguaje, que se utiliza en la programación de aplicaciones paralelas. Se basa en el modelo de paso de mensajes entre un proceso emisor y un proceso receptor, y soporta tanto comunicaciones punto a punto como colectivas. Permite la concurrencia, a través de la sincronización y la exclusión mutua entre los procesos.

Además, MPI es una API de programación que especifica tanto el protocolo como la semántica de funcionamiento, y define el comportamiento para cualquier implementación. MPI apunta a la obtención de un alto rendimiento en las comunicaciones, escalabilidad y portabilidad. Estas características hacen que sea el modelo más utilizado en la computación de alto rendimiento hoy en día.

MPI se creó para resolver el inconveniente que surgió luego de que cada fabricante de hardware definiera su propia interfaz de comunicación, generalmente incompatible con las demás. El objetivo del surgimiento de MPI fue solucionar este problema, definiendo para ello un estándar [70].

MPI es la estandarización de una especificación de biblioteca que define tanto la sintaxis como la semántica de un conjunto de operaciones de comunicación que pueden ser utilizadas con los lenguajes C, C++, Fortran-77 y Fortran-95. En la actualidad, existen dos versiones del estándar MPI: MPI-1 define operaciones de comunicación estándar y se basa en un modelo estático de procesos. MPI-2 es una extensión de MPI-1 y provee soporte adicional para manejo dinámico de procesos, comunicaciones unidireccionales y E/S paralela. Se debe resaltar que MPI es una especificación de una interfaz bien definida para la sintaxis y la semántica de un conjunto de operaciones de comunicación, pero que no da detalles de implementación de las mismas. Por lo tanto, diferentes bibliotecas MPI emplean distintas implementaciones, utilizando optimizaciones específicas para cada plataforma de hardware.

De todas maneras, como la interfaz es estándar, la portabilidad de los programas está garantizada [62]. Entre las implementaciones de MPI más utilizadas se encuentran MPICH [71], LAM/MPI [72] y OpenMPI [73], que provee mecanismos para realizar el *mapping* (asignación) de procesos a procesadores para ser ejecutados manualmente.

MPI es una interfaz amplia y compleja, que brinda más de 125 rutinas. Sin embargo, posee unas pocas claves conceptuales que facilitan notablemente su utilización. De hecho, con sólo seis funciones se puede resolver una gran cantidad de problemas [67]. En la tabla 4.1 se detallan las seis rutinas MPI básicas de manejo del entorno de ejecución.

Función MPI	Objetivo
MPI_INIT	Inicializa el entorno MPI. Esta función debe utilizarse antes de llamar a cualquier otra función MPI.
MPI_FINALIZE	Finaliza la ejecución de los procesos que forman una sesión MPI
MPI_COMM_SIZE	Devuelve el número de procesos de la sesión MPI actual (asociados a un comunicador)
MPI_COMM_RANK	Permite determinar el identificador (*rank*) del proceso que invoca a la función
MPI_SEND	Realiza el envío de un mensaje a un proceso de forma bloqueante (punto a punto)
MPI_RECV	Realiza la recepción de un mensaje de forma bloqueante (punto a punto)

Tabla 4.1: Funciones MPI básicas

4.2.1 Comunicaciones no bloqueantes

Dependiendo de si los procesos implicados en la comunicación esperan o no a que el mensaje haya sido entregado, se definen el paso de mensajes bloqueantes o no bloqueantes.

En el paso de mensajes bloqueante, el proceso que envía el mensaje espera a que el proceso receptor lo reciba para continuar su ejecución. Asimismo, el proceso receptor se queda bloqueado en la llamada de recepción del mensaje hasta que éste se haya recibido.

En la versión no bloqueante, el proceso que realiza el envío del mensaje no espera a que el proceso receptor reciba dicho mensaje para continuar. El receptor tampoco permanece a la espera de recibir el mensaje; si el mensaje está listo, lo recibe; si no, prepara la recepción pero continúa su ejecución. Posteriormente, existen llamadas para comprobar si el mensaje ha sido efectivamente recibido.

Las operaciones de comunicación descritas en la Sección 4.2 son bloqueantes. Esto significa que el control sólo retorna al proceso que realiza la llamada cuando todos los recursos involucrados están en condiciones de ser reutilizados, como por ejemplo los *buffers* especificados en la invocación. Los protocolos bloqueantes incurren en cierto *overhead* para poder garantizar esta característica semántica. Si, en lugar de quedar bloqueado a la espera de que se complete la operación, el proceso que realiza la llamada recupera el control inmediatamente, entonces puede proseguir realizando cualquier cómputo que no dependa de los datos de la operación de comunicación pendiente. Luego, este proceso puede verificar si la operación de comunicación se ha completado y, si es necesario, esperar a que lo haga. Estos protocolos no bloqueantes proveen operaciones de envío y recepción rápidas, a costa de que la corrección semántica debe ser verificada por el programador [26].

MPI provee variantes no bloqueantes de las funciones de envío y recepción: MPI_ISEND y MPI_IRECV. Los llamados a estas funciones no esperan a que la comunicación se complete para retornar el control. Para poder verificar si una operación se ha completado, MPI proporciona la función MPI_WAIT. En la tabla 4.2 se describen estas funciones.

Función MPI	Objetivo
MPI_ISEND	Inicia el envío de un mensaje realiza el envío de un mensaje de forma no bloqueante (punto a punto)
MPI_IRECV	Inicia la recepción de un mensaje de forma no bloqueante (punto a punto)
MPI_WAIT	Bloquea al proceso que llama a la función hasta que se complete cierta tarea de envío o recepción. La tarea se indica mediante un número de petición (*Request*) que el sistema asignó previamente a dicha tarea.
MPI_TEST	Comprueba si una operación de envío o recepción no bloqueante se ha completado

Tabla 4.2: Funciones MPI para comunicaciones no bloqueantes

4.2.2 Comunicadores

MPI permite formar grupos de procesos y gestionarlos para facilitar las comunicaciones entre ellos. Además, permite identificar cada uno de los procesos en ejecución.

Un comunicador es una estructura que se emplea para definir un conjunto de procesos que se comunican entre sí. Este conjunto de procesos determina lo que se conoce como un dominio de comunicación. En general, todos los procesos pueden necesitar comunicarse entre ellos, por lo que MPI define un comunicador predeterminado llamado MPI_COMM_WORLD, que involucra a todos los procesos de la aplicación. Sin embargo, muchos patrones de comunicación requieren que los procesos sólo se comuniquen por grupos, posiblemente en forma solapada. Una manera de garantizar que los mensajes dentro de un grupo nunca interfieran con los de otro grupo es utilizar un comunicador para cada grupo particular [26].

4.2.3 Comunicaciones colectivas

MPI provee un conjunto extensivo de funciones que realizan operaciones de comunicación colectivas de utilización frecuente. Todas estas funciones toman como argumento un comunicador que define el grupo de procesos involucrados en la operación de comunicación. Todos los procesos pertenecientes a este comunicador participan en la operación colectiva y es necesario que todos ellos invoquen a la función correspondiente [26].

En la tabla 4.3 se describen las funciones MPI más relevantes para comunicaciones colectivas.

Función MPI	Objetivo
MPI_BARRIER	Crea una barrera de sincronización. Cada proceso que llega a la barrera se bloquea hasta que lleguen todos los procesos del comunicador

MPI_BCAST	Envía un mensaje de un proceso a todos los demás procesos del grupo.
MPI_SCATTER	Reparte datos de un proceso a todos los procesos de un grupo, para lo cual envía un mensaje diferente a cada uno de los demás
MPI_GATHER	Agrupa datos de todos los procesos del grupo en un único proceso. Realiza la operación inversa a MPI_ Scatter, recibe un mensaje compuesto por submensajes generados por los demás procesos del comunicador
MPI_REDUCE	Aplica una operación de reducción sobre los datos enviados por todos los procesos de un grupo y los envía a otro proceso.

Tabla 4.3: Funciones MPI más relevantes para comunicaciones colectivas

4.2.4 Tipos de datos

MPI define una serie de tipos de datos a los que denomina tipos básicos. Los tipos básicos encapsulan los tipos de datos definidos por el lenguaje de programación con el cual está desarrollada la aplicación MPI.

Además de los tipos básicos, MPI define tipos derivados de datos con el propósito de facilitar el intercambio de datos de más complejos (estructuras de datos con tipos heterogéneos, vectores, matrices, etc.), disponiendo de rutinas para su creación.

Otra de las funcionalidades que permite MPI es la de empaquetar datos en un mensaje, independientemente de su tipo y de la cantidad de datos enviados. Para ello, los datos se almacenan en un contenedor definido por el tipo de datos MPI_PACKED.

4.2.5 Ventajas y desventajas de MPI

A modo de resumen, se enumeran las principales ventajas de MPI:

- Es estándar
- Es portable a múltiples plataformas hardware y software
- Es compatible con configuraciones tanto de memoria distribuida como de memoria compartida
- Ofrece alto rendimiento
- Dispone de varias implementaciones diferentes (OpenMPI, MPICH, LAM/MPI, etc.)
- La API está en constante evolución, apoyada por una gran cantidad de empresas y entidades del ámbito del hardware y el software
- Es flexible, ya que permite utilizar redes heterogéneas
- Dispone de una amplia documentación

Entre las desventajas, la principal es que el mecanismo de paso de mensajes conlleva una sobrecarga debido al tiempo de envío del mensaje y al tráfico en la red.

Capítulo 5

Detección de fallos transitorios en cómputo paralelo

Las técnicas de detección de fallos transitorios descriptas en el capítulo 2, basadas en redundancia de software, están diseñadas para aplicaciones secuenciales. Desde el punto de vista de la tolerancia a fallos, una aplicación paralela determinística puede verse como un conjunto de procesos secuenciales que deben ser protegidos de las consecuencias de los fallos transitorios por medio de las estrategias que se adopten para ello.

Como se analizó en detalle en el capítulo 4, MPI [69] es actualmente el estándar de facto que define una API para programación paralela con pasaje de mensajes. MPI está diseñado para obtener comunicaciones de alta performance en aplicaciones científicas paralelas.

En el contexto de la popularidad alcanzada por los clusters y las aplicaciones paralelas que utilizan MPI, la fiabilidad de las aplicaciones y las capas de middleware se ha vuelto un aspecto relevante. La disponibilidad de los clusters ha aumentado inherentemente a nivel del hardware mediante el agregado de componentes redundantes y reemplazables en caliente (*hot-swappable*). Sin embargo, esa disponibilidad y fiabilidad no se transfieren automáticamente a las capas superiores. En aplicaciones de gran escala y larga duración, o en entornos con condiciones ambientales severas (ej: aplicaciones espaciales) es inevitable la ocurrencia de fallos externos, no relacionados con defectos del software.

Existe una discrepancia entre las características de alto desempeño de los sistemas de cómputo actuales y el paradigma más utilizado para la programación paralela. Las máquinas han incrementado su robustez, mediante mejoras en las redes, los sistemas operativos y los sistemas de archivos, pero la especificación de MPI no permite explotar estas cualidades de las arquitecturas modernas.

Dado que agregar características que mejoren la confiabilidad en las comunicaciones aumenta el *overhead*, el uso de MPI está limitado a entornos en los que la ocurrencia de fallos externos es abordada desde las aplicaciones, o en los cuales el hardware y los servicios de bajo nivel enmascaran todos los fallos. No obstante el hecho de que los procesos MPI pueden fallar debido a factores externos (fallas del procesador, la red, la tensión de alimentación, etc.), la detección de estos fallos no está definida en el estándar. En lugar de esto, MPI plantea un modelo estático que requiere que todos los procesos terminen exitosamente, lo cual implica que una falla que afecta a un único proceso resulta en la caída de toda la aplicación (de hecho, este es el comportamiento que exige el estándar). En particular, si un único proceso MPI de una aplicación paralela deja de responder (por ej: entra en un interbloqueo o en un lazo infinito) o se aborta prematuramente, se requiere la finalización completa de la aplicación (ya sea de forma “elegante” o anormal), o, en la práctica, toda la aplicación deja de responder. Como consecuencia, la aplicación no puede lanzar un nuevo proceso MPI para reemplazar al que ha fallado, ni ignorarlo para continuar la ejecución en un modo degradado. Estas limitaciones del estándar y de las implementaciones dificultan la detección y recuperación automática basada en software [40,75].

MPI proporciona al usuario dos posibles maneras para lidiar con los fallos. La primera (el modo por defecto) es abortar inmediatamente la aplicación, como se mencionó. La segunda posibilidad, algo más flexible, consiste en devolver el control a la aplicación de usuario, sin garantizar el éxito de ninguna comunicación más allá del límite actual. Esta segunda opción busca darle a la aplicación la chance de realizar operaciones locales antes de finalizar, como por ejemplo cerrar todos los archivos o almacenar un estado consistente para la recuperación posterior.

A continuación, y a modo de resumen, se listan las principales deficiencias de MPI respecto de la tolerancia a fallos externos:

- **Modelo de fallo:** la especificación asume que la infraestructura subyacente es confiable, y por lo tanto no trata con los errores que ocurren en tiempo de ejecución. Por lo tanto, el estándar las implementaciones no proveen a las aplicaciones mecanismos para tratar con fallas en los nodos o mensajes perdidos. Por ejemplo, en la API de usuario de MPI no hay un mecanismo de *time-out*, por lo que una sentencia de recepción bloqueante puede esperar indefinidamente si el proceso MPI que envía falla, o si una red de comunicaciones no fiable pierde el mensaje entrante. Si esto ocurre, la implementación de MPI debe asumir que la aplicación tiene la responsabilidad de proveer un pasaje de mensajes confiable.

- **Detección y notificación de fallos:** la detección de fallos externos no está definida. MPI sólo provee la detección de algunos casos de errores internos locales, como el reporte de argumentos incorrectos de funciones o errores de recursos (ej: intento de exceder el espacio de un *buffer*). La notificación de los errores se lleva a cabo mediante la invocación de una función de manejo del error especificada por la aplicación. Si el manejador de error no detiene la aplicación, la llamada MPI retorna un código de error. El estándar no especifica todos los tipos de errores que se detectan ni los códigos de error [40].

- **Recuperación de errores:** en las aplicaciones MPI, la recuperación de la ocurrencia de fallos externos se encuentra inhibida debido a la cantidad restringida de información contenida en los códigos de error no estandarizados que devuelven las llamadas MPI. Las opciones de recuperación también se ven obstaculizadas debido a que el estándar MPI define un comunicador estático, y además exige la finalización de todas las tareas cuando un proceso o un nodo falla.

La popularidad en la utilización de MPI ha conducido a una cantidad de esfuerzos de investigación para robustecer la especificación, mediante la incorporación de características de fiabilidad. La mayoría de las soluciones propuestas en la literatura (y los middlewares que resultan de ellas) se basan en un modelo de "caja negra", según el cual proveen abstracciones estáticas para las características de tolerancia a fallos, y un conjunto fijo de servicios (por ej: *checkpointing* transparente, *logging* de mensajes y recuperación por *roll-back*) para todas las clases de aplicaciones paralelas [76,77,78]. Sin embargo, la rigidez de estas soluciones resulta en *overheads* innecesariamente altos que se contraponen con el objetivo de MPI de lograr alto rendimiento. Por otra parte, para ciertas aplicaciones, limita la flexibilidad alcanzable, no permitiendo sacrificar fiabilidad para obtener un mejor desempeño.

Las estrategias de tolerancia a fallas que se usan típicamente en entornos MPI almacenan periódicamente el progreso de la aplicación y sus resultados parciales en *checkpoints* y utilizan los datos del último *checkpoint* para recuperar las aplicaciones que fallan prematuramente. En la situación de que una falla externa afecte un proceso MPI, causando la caída de toda la aplicación, el usuario debe volver a lanzar la ejecución; sin embargo, la necesidad de intervención humana introduce un factor de ineficiencia considerable. En el escenario más desfavorable, si un fallo transitorio corrompe el resultado de una aplicación pero sin causar su finalización abrupta, el usuario debe esperar el término de la ejecución para hallar que los resultados son incorrectos o, peor aún, recolectarlos y utilizarlos como si fueran correctos.

Además, como el fallo no produce la caída de la aplicación, los *checkpoints* almacenados no se utilizan como elemento de recuperación [40,75].

Por otra parte, si se consideran máquinas con decenas de miles de procesadores, la técnica de *checkpoint/roll-back* para tolerancia a fallos tiene limitaciones, tanto conceptuales como de rendimiento [75].

Otra clase de soluciones, que no soportan *checkpoint/roll-back*, se basan en técnicas de replicación y manejo dinámico de procesos (como MPI-2) para mantener la "buena salud" de los comunicadores.

Existen algunas aproximaciones que extienden MPI para implementar réplicas de los procesos de la aplicación para tolerar fallos permanentes. Para limitar el *overhead*, sólo brindan cobertura a los fallos externos que producen la caída de un proceso, que se manifiesta como falta de respuesta. Como consecuencia, los fallos transitorios que provocan resultados incorrectos pero permiten que la aplicación MPI continúe su ejecución son ignorados. La detección de esos fallos debe ser provista por la aplicación por medio del uso explícito de técnicas tradicionales. En el capítulo 6 se analiza con detalle SMCV, una técnica desarrollada especialmente para proveer detección de fallos transitorios en aplicaciones paralelas científicas de paso de mensajes que utilizan MPI como librería de comunicaciones.

A continuación se analizarán brevemente algunas propuestas existentes en la literatura, que agregan aspectos de detección y recuperación de fallos a MPI con el objetivo de brindar robustez y fiabilidad a las aplicaciones que lo utilizan.

5.1 MPI/FT

MPI/FT [40] es un middleware basado en MPI que proporciona servicios adicionales de detección de fallos y recuperación de procesos MPI fallidos. Estos servicios para tolerancia a fallos están adaptados para diferentes modelos específicos de ejecución de aplicaciones, con el objetivo de optimizar el rendimiento y minimizar el *overhead* introducido. Esto resulta en una implementación del middleware diferente para cada modelo, por lo que la elección de servicios de tolerancia a fallos depende de la aplicación en particular. Los modelos de ejecución de las aplicaciones son abstracciones obtenidas a partir de ciertas combinaciones de topologías de comunicación y modelos de programación paralela que son susceptibles de proporcionar servicios de tolerancia a fallos introduciendo un *overhead* menor.

MPI/FT proporciona tolerancia a fallos introduciendo un coordinador central y/o replicando procesos MPI. De esta forma, la biblioteca puede detectar mensajes erróneos, utilizando un algoritmo de votación entre réplicas, y es capaz de sobrevivir a la caída de los procesos. El inconveniente que presenta es la necesidad de una mayor cantidad de recursos y la degradación parcial de rendimiento.

En MPI/FT, la API de MPI se extiende para proveer servicios de detección de fallos, notificación, recuperación, redundancia de procesos y *checkpointing* controlado por el usuario. La decisión sobre la conveniencia de los servicios y su implementación dependen del modelo de ejecución de la aplicación particular, permitiendo la flexibilidad necesaria para, si se requiere, mejorar el rendimiento a costa de perder fiabilidad.

El modelo de fallo de MPI/FT asume que las aplicaciones no contienen errores internos (es decir, defectos en el software). Esto es consecuencia de que si un defecto en el software causa la caída de un proceso MPI, una vez que la ejecución se recupere desde un *checkpoint*, el defecto del software causará un nuevo error. Esta secuencia puede repetirse indefinidamente sin que la aplicación avance.

Además, para limitar el *overhead* introducido, MPI/FT sólo ofrece cobertura frente a los fallos externos que llevan a una caída del proceso que se manifiesta como una falta de respuesta por parte del proceso involucrado. En cambio, no

se trata con los fallos transitorios que producen corrupción silenciosa de datos (permitiendo que la aplicación continúe su ejecución). La detección y cobertura frente a estos fallos debe ser provista por la aplicación mediante el uso explícito de algoritmos de detección y recuperación tradicionales [79,80].

La estrategia de diseño de MPI/FT se basa en implementar un middleware con servicios de tolerancia a fallos adaptados a los requerimientos particulares de la aplicación. Para equilibrar el costo de desarrollar un middleware a medida para cada aplicación paralela frente al *overhead* introducido por un middleware general, MPI/FT proporciona diferentes implementaciones para conjuntos de aplicaciones agrupadas según modelos de ejecución que se muestran en la tabla 5.1. MPI/FT también provee mecanismos de notificación y re-ejecución pero delega en la aplicación la responsabilidad de hacer *checkpointing* y retomar la ejecución desde el último *checkpoint* válido.

Estilo de programación	**Topología de comunicación**	**Middleware de redundancia y *checkpointing***	**Implementado**
Maestro/Esclavo	Estrella	Ninguno	Sí
		Sólo en el *master*- activo	No
		Sólo en el *master*- pasivo	No
SPMD	Malla (todos con todos)	Ninguno	Sí
		Solo coordinador en el *rank* 0 - activo	No
		Solo coordinador en el *rank* 0 - pasivo	No

Tabla 5.1: Modelos de ejecución de aplicaciones

MPI/FT toma en cuenta los siguientes parámetros para establecer los diferentes modelos de ejecución de las aplicaciones:

- ✓ Topología de comunicaciones virtual: las aplicaciones MPI tienen una topología virtual de comunicaciones definida por un grafo dirigido en el que los nodos representan procesos y los arcos representan mensajes. Las más comunes, en aplicaciones paralelas, son el hipercubo, la matriz rectangular, la estrella y la conexión completa (*all-to-all*). Esta topología influye directamente sobre el diseño y la implementación de las comunicaciones colectivas de bajo *overhead*, la detección basada en la aplicación y los servicios de recuperación a nivel de middleware.

- ✓ Estilo de programación: hasta la versión del estándar MPI-1.2, las aplicaciones típicamente seguían uno de los dos patrones más comunes: maestro-esclavo o SPMD. En estos modelos, el flujo del programa consiste en iteraciones de cómputo intercaladas con comunicaciones. El estilo influye en la apropiada ubicación de las llamadas a los servicios de tolerancia a fallos provistos por MPI/TF. El análisis del estilo del programa también permite sugerir optimizaciones (por ej: tal vez sólo un subconjunto de los procesos de la aplicación requieren realizar un *roll-back* para recuperarse de un fallo).

- ✓ Nivel de redundancia en el middleware: para lograr fiabilidad se requiere la replicación tanto de los procesos como de los datos. Para reducir el *overhead* introducido y mantener un alto rendimiento, se debe eliminar cuidadosamente cualquier redundancia innecesaria. Por ejemplo, siempre que los requerimientos

de fiabilidad de la aplicación lo permitan, se debe utilizar redundancia pasiva en lugar de redundancia activa. En la redundancia pasiva, los procesos sobrantes creados se encuentran ociosos o están asignados a otras tareas, pudiendo ser inicializados para reemplazar procesos que han fallado. En la redundancia activa, los procesos redundantes duplican el trabajo realizado por los procesos que van a ser reemplazados, por lo que no requerirán inicialización cuando uno de los procesos originales falle.

5.1.1 El modelo de ejecución de aplicaciones Maestro - Esclavo

En este modelo, basado en una topología virtual de comunicaciones estrella, el nodo maestro se comunica con los esclavos y los esclavos con el maestro, pero un esclavo no se comunica directamente con otros. Debido a que no hay interacción ni sincronización explícita entre esclavos no se implementan llamadas colectivas en este modelo. El nodo maestro envía trabajo a los esclavos y recibe resultados de ellos.

Una falla que afecta a un esclavo produce que el middleware retorne un código de error al proceso maestro, que involucre sólo las llamadas MPI del proceso fallido; la comunicación entre el maestro y los demás procesos no resulta normalmente afectada. El proceso maestro utiliza los servicios de MPI/FT para relanzar al esclavo que ha fallado y reenviarle el trabajo que no pudo completarse. MPI/FT repara el comunicador MPI_COMM_WORLD para reflejar el proceso que ha sido reemplazado.

En tanto, si la falla afecta al nodo maestro, se requiere que la aplicación completa sea relanzada desde un *checkpoint*. Esto resulta en un punto de falla centralizado, en contraposición a los múltiples puntos de falla centralizados que presenta la aplicación MPI original.

5.1.2 El modelo de ejecución de aplicaciones SPMD

En este modelo, todos los procesos pueden comunicarse con cualquiera de los demás (topología *all-to-all*). Los procesos consisten en lazos iterativos sincrónicos y las comunicaciones ocurren al comienzo o final de cada iteración. En este modelo, una falla que afecta a un proceso MPI conduce a la propagación sincrónica de notificaciones de error a los demás procesos. Como consecuencia, todos los demás procesos hacen *roll-back* al último *checkpoint* almacenado. El middleware proporciona servicios para recuperar y relanzar el proceso fallido desde el *check point*, y también repara el comunicador MPI_COMM_WORLD para reflejar los procesos que han sido reemplazados.

5.1.3 Detección de fallos y notificación

Una buena estrategia de detección de fallos debe balancear la precisión (es decir, minimizar los falsos positivos), la velocidad (minimizar la latencia entre la ocurrencia del fallo y el instante de su detección) y su costo asociado (que introduzca un bajo *overhead* en ausencia de fallos). En MPI/FT, el mecanismo de detección es el factor que contribuye más significativamente al *overhead* total a la ejecución libre de fallos. Para limitar ese *overhead*, la implementación está restringida a la detección de los procesos MPI que dejan de responder; la integración de mecanismos en tiempo de ejecución que soporten la detección y corrección de otros tipos de error es una tarea pendiente para implementaciones futuras.

La falta de respuesta de los procesos se detecta mediante el uso de mensajes de *heartbeat* internos y externos, generados y monitorizados por los hilos de autocomprobación (*Self-Checking Threads*, SCT´s) provistos por MPI/FT. Los SCT´s utilizan los mensajes internos de *heartbeat* para monitorizar el progreso de los hilos de envío y recepción en los procesos MPI/FT. El coordinador utiliza los mensajes externos de *heartbeat* para recolectar información de avance obtenida por SCT´s remotos.

De esta forma, todas las tareas asociadas a la detección que realiza MPI/FT asumen que si un SCT no recibe un mensaje de *heartbeat* interno dentro de un intervalo de tiempo especificado, o si el coordinador no recibe un mensaje de *heartbeat* externo dentro de un intervalo de tiempo especificado, entonces un proceso de la aplicación ha fallado y debe iniciarse la recuperación. En otras palabras, toda la detección se realiza por *time-out*.

La validación experimental del prototipo de MPI/FT se realiza evaluando la latencia y el ancho de banda del middleware, así como también el tiempo total de ejecución de las aplicaciones hasta su finalización. El rendimiento obtenido se compara con la implementación original de MPI para determinar el *overhead* introducido.

Una conclusión generalizable útil es que el beneficio de invocar al mecanismo de recuperación depende claramente del momento en el que ocurre el fallo dentro del ciclo de vida de la aplicación. Si una aplicación está cercana a su finalización, y el proceso de recuperación no produce una mejora en el tiempo total de ejecución, o extiende ese tiempo, la aplicación puede decidir no realizar la recuperación y continuar, en modo degradado, sin el proceso que ha fallado.

La realización de *checkpointing* agrega un *overhead* considerable al tiempo total de ejecución, por lo que el usuario de MPI/FT debe determinar la frecuencia de *checkpointing* adecuada para la aplicación particular y su entorno de ejecución. Realizar *checkpoints* más frecuentemente de lo conveniente aumenta innecesariamente el *overhead* en ausencia de fallos; en cambio, si no se realizan con la suficiente frecuencia, la ocurrencia de un fallo puede producir que una gran cantidad de trabajo deba ser repetido. Ambas condiciones evitan o retardan el avance del cómputo.

5.2 FT-MPI

FT-MPI (*Fault-Tolerant MPI*) [75] consiste en una extensión de MPI para proporcionar tolerancia frente a fallos de los procesos. Especifica la semántica para una versión de MPI tolerante a fallos, la arquitectura de la biblioteca de comunicaciones y provee los detalles de una implementación de la nueva especificación, que proporciona funcionalidad extendida para un modelo de recuperación frente a la ocurrencia de fallos.

El comunicador de MPI es una importante estructura de datos que define un conjunto de procesos y el contexto de comunicaciones entre ellos. Si un proceso termina prematuramente, los comunicadores que contienen a dicho proceso quedan en un estado inválido. FT-MPI permite que la aplicación continúe, utilizando el comunicador que contiene al proceso fallido (pero excluyendo explícitamente las comunicaciones con ese proceso), o bien reduce el comunicador excluyendo el proceso fallido, o bien genera un nuevo proceso para reemplazar el que ha fallado. Por lo tanto, FT-MPI no es un sistema automático de *checkpoint/recovery*, pero proporciona a la aplicación la posibilidad de continuar a pesar de la ocurrencia de fallos en los nodos o en los canales de comunicación. Los comunicadores se reorganizan y la aplicación retoma su ejecución desde un punto bien definido a nivel de usuario. Sin embargo, definir e implementar un estado consistente en la aplicación es responsabilidad del desarrollador.

La capacidad del middleware de reparar un comunicador inválido produce que existan opciones de recuperación basadas en la aplicación, distintas de re-ejecutar desde el último *checkpoint*. Esta flexibilidad permite construir aplicaciones con mecanismos de recuperación de grano fino. Sin embargo, FT-MPI no elabora estrategias de detección de

fallos, ni proporciona una API para notificación ni para *checkpoint*. FT-MPI tiene menor *overhead* que MPI/FT (y por lo tanto mayor rendimiento potencial), pero las aplicaciones deben ser diseñadas específicamente para beneficiarse de sus características de tolerancia a fallos. Esta tarea puede ser trivial, dependiendo de la estructura de la aplicación. Si una aplicación requiere una gran cobertura frente a fallos, debe ser diseñada para realizar *checkpointing* controlado por el usuario. Una ventaja de esto es que no se requiere detener toda la aplicación y volver a planificarla, lo que ocurre en la mayoría de los sistemas de *checkpoint* a nivel de procesos.

La ocurrencia de fallos se maneja típicamente mediante tres pasos: detección, notificación y recuperación. La especificación de FT-MPI no asume nada respecto de los dos primeros, excepto que el entorno de tiempo de ejecución es capaz de descubrir la ocurrencia del fallo y que todos los demás procesos de la aplicación paralela son notificados. La notificación de que un proceso ha fallado se le pasa a la aplicación MPI mediante un código especial de error. Un proceso que ha recibido esta notificación puede ejecutar una cantidad de acciones que dependen de varios parámetros.

El procedimiento de recuperación consiste en dos pasos: recuperación de la biblioteca MPI y del entorno de tiempo de ejecución, y recuperación de la aplicación (que es considerada responsabilidad de la misma aplicación).

Existen tres modos de recuperación que definen la forma en la que se inicia el procedimiento de recuperación:

- Modo automático: el procedimiento de recuperación es lanzado automáticamente por la biblioteca MPI ni bien se reconoce que ha ocurrido una avería
- Modo manual: la aplicación debe iniciar el procedimiento de recuperación mediante el uso de una función especial de MPI.
- Modo de recuperación: el procedimiento no es iniciado. Cualquier comunicación con el proceso fallido resulta en un error.

Además, la especificación define dos modos para tratar con los mensajes cuando ocurre un error. En el primero, todos los mensajes en tránsito son cancelados por el sistema. Esto es particularmente útil para las aplicaciones que ante un error regresan al último estado consistente. El segundo modo completa la transferencia de todos los mensajes luego de la operación de recuperación, exceptuando los mensajes desde y hacia el proceso fallido. Este modo requiere que la aplicación mantenga información detallada del estado de cada proceso. Existen modos similares para las comunicaciones colectivas.

Las aplicaciones pueden detectar y manejar los eventos asociados con los fallos de dos maneras posibles: verificando los códigos de retorno de cada función de MPI, o utilizando los manejadores de error de MPI. Este segundo modo brinda a los usuarios la posibilidad de incluir una función, en la biblioteca MPI, que es llamada en caso de que ocurra un error. De esta forma, no se debe modificar el código fuente existente para introducir funcionalidad detallada de verificación de errores para cada función MPI que se utilice. La utilización de manejadores de error de la especificación MPI mejora significativamente la legibilidad y mantenibilidad de las aplicaciones tolerantes a fallos.

La figura 5.1 muestra la arquitectura de FT-MPI. Los autores describen diferentes escenarios de aplicaciones que muestran la factibilidad de la especificación en entornos de cómputo científico de altas prestaciones. La validación experimental consiste en medidas de ancho de banda y latencia de en comunicaciones punto a punto y colectivas, comparando a FT-MPI con las implementaciones de MPI no tolerantes a fallos más utilizadas del momento (LAM/MPI 7 y MPICH 1.2.5). Además, proporcionan ejemplos útiles para el desarrollo de aplicaciones tolerantes a fallos (que requie-

ren modificaciones a las aplicaciones paralelas existentes), tanto para un patrón de comunicaciones Maestro/Esclavo como para una aplicación fuertemente acoplada.

5.3 Evaluación de la viabilidad de la replicación de procesos en HPC

La replicación de procesos es una técnica bien conocida para tolerar fallos en sistemas que apuntan a la alta disponibilidad. En este tipo de solución, el estado de un proceso se replica de modo que si el proceso original falla, su réplica está disponible (o puede ser generada) para tomar el lugar del proceso original sin afectar a los demás procesos de la aplicación.

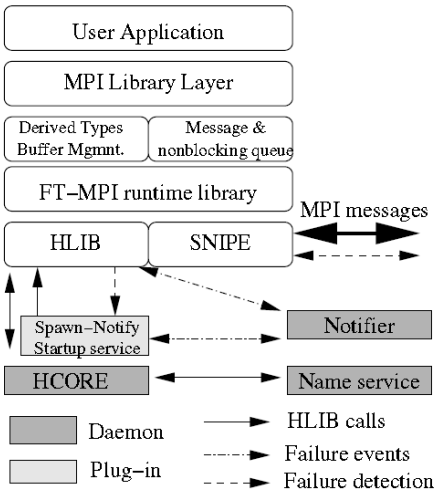

Fig 5.1: Arquitectura de FT-MPI

La replicación de procesos puede ser costosa en términos espaciales (si las réplicas utilizan recursos dedicados) o temporales (si las réplicas son alocadas junto con los procesos primarios). Sin embargo, la replicación de procesos es capaz de incrementar significativamente el tiempo promedio hasta la ocurrencia de una interrupción en la ejecución (MTTI - *Mean Time To Interrupt*) de una aplicación. Algunas variantes de esta técnica pueden utilizarse para detectar fallos que producen SDC [81].

Principalmente debido a sus costos asociados, la replicación de procesos ha sido poco aplicada en el ámbito de HPC. En general, las aplicaciones de HPC confían en mecanismos de *checkpoint* coordinado a un almacenamiento estable y *rollback recovery* a un estado previo consistente ante la ocurrencia del fallo. Sin embargo, se espera que los sistemas de gran escala presenten escenarios mucho más complejos para las aplicaciones respecto de la tolerancia a fallos. Estudios recientes han concluido que las altas tasas de fallos en sistemas de exa-escala, en conjunto con los niveles de *overhead* que implican los mecanismos de *checkpoint/restart* son factores que harán inviable la aplicación de soluciones basadas en estas estrategias. Por ejemplo, estudios independientes han llegado a la conclusión de que un sistema de exa-escala potencial podría pasar más de la mitad de su tiempo leyendo y escribiendo *checkpoints* [77,82].

Estos inconvenientes han sido la motivación para diferentes investigaciones que apuntan a mejorar la escalabilidad de los mecanismos de *checkpoint/restart* [83,84,85]. La mayoría de las propuestas involucran recursos adicionales, como almacenamiento local, enlaces de comunicaciones entre nodos o capacidad de memoria. Sin embargo, a la luz de

los requerimientos de fiabilidad de los sistemas de exa-escala que se avecinan, las investigaciones también se han volcado a examinar la aplicabilidad de mecanismos existentes, como la replicación de procesos, para estos sistemas [82,86,87].

Los autores de [53] examinan la viabilidad de la replicación de procesos como mecanismo primario de tolerancia a fallos, manteniendo el *checkpoint/restart* para proporcionar un mecanismo secundario en caso de ser necesario, en la búsqueda de las ventajas y limitaciones de ese esquema. El trabajo está enfocado a las aplicaciones MPI de gran escala. Las réplicas de los procesos permiten la conmutación entre ellas en caso de error, lo que implica que las aplicaciones puedan atravesar la ocurrencia de fallos de manera transparente sin necesidad de volver atrás. El esquema es mejorado mediante el mecanismo de *checkpoint/restart*, que se utiliza en los casos en los que la replicación de procesos es insuficiente (por ejemplo, si todas las réplicas de un proceso fallan simultáneamente o se vuelven inconsistentes debido a fallos que corrompen el estado del sistema).

En [53] se presenta un estudio sobre cómputo redundante para aplicaciones a exa-escala, de donde concluyen sobre los problemas concernientes a la escalabilidad de las técnicas de *checkpoint/restart* (en sistemas de exa-escala tienen requerimientos de hardware comparables a los de la replicación) y su incapacidad para tolerar fallos de hardware no detectados y errores que no provocan la interrupción de la ejecución. En particular, la utilización del mecanismo de *checkpoint/restart* resulta problemática frente a la ocurrencia de fallos transitorios, debido a que los *checkpoints* conservan el impacto de los errores que corrompen el estado de la aplicación. Abordar estos errores requeriría re-ejecutar las aplicaciones desde el comienzo o analizar los contenidos de los *checkpoints*, en la búsqueda de uno que haya sido almacenado previamente al momento de la ocurrencia del fallo que corrompió el estado de la aplicación.

El análisis basado en el modelo que se realiza permite evaluar la redundancia de cómputo como una alternativa viable, ya que, aún para sistemas con escalas menores que la exa-escala, la replicación de procesos conlleva un *overhead* en hardware menor que el mecanismo de *checkpoint/restart* tradicional.

Por otra parte, los autores diseñan e implementan rMPI, una implementación completa de MPI con replicación de procesos, y la evalúan en un sistema de HPC, obteniendo como resultado que los *overheads* introducidos por la replicación son mínimos para aplicaciones reales.

5.3.1 Replicación de procesos para aplicaciones de HPC con paso de mensajes

La replicación de procesos es conceptualmente directa en aplicaciones de HPC con paso de mensajes. Para cada proceso de la aplicación original (en el cual no se puedan tolerar fácilmente los fallos) se genera una réplica que se ejecuta en un hardware independiente. No siempre es necesario replicar todos los identificadores (*ranks*). Por ejemplo, en una aplicación con un patrón de comunicaciones Maestro/Esclavo, en la que el Maestro es capaz de recuperar la pérdida de un proceso Esclavo, sólo se requiere replicar el proceso Maestro.

El sistema de replicación garantiza que cada réplica recibe los mismos mensajes en el mismo orden y que una copia de cada mensaje de un *rank* se envía a cada réplica en el *rank* de destino. Por otra parte, el sistema de replicación debe detectar errores en las réplicas, reparar nodos caídos cuando sea posible y relanzar nodos que han fallado desde sus réplicas activas. Además, debe verificar periódicamente que los *ranks* replicados tengan el mismo estado. En tanto, el mecanismo de *checkpoint/restart* aún resulta necesario para lograr la recuperación ante los fallos que afectan a todas las réplicas de un *rank* particular, así como para recuperar situaciones en las que el estado se vuelve inconsistente, como por ejemplo debido a fallos silenciosos no detectados.

La solución basada en replicación requiere de una gran cantidad de recursos computacionales adicionales (al menos el doble del hardware para los *ranks* replicados). Si es necesario replicar sólo una parte de la aplicación, estos requerimientos pueden ser mucho menores. Para la gran mayoría de las aplicaciones de HPC, el hardware debe duplicarse (se requieren 2N nodos de cómputo para replicar completamente una tarea que de otro modo se ejecutaría en N nodos). Además, se introduce un *overhead* en tiempo de ejecución para mantener la consistencia entre las réplicas.

Sin embargo, estos costos traen aparejadas ciertas ventajas significativas:

- El MTTI aumenta considerablemente, debido a que la replicación reduce la cantidad de errores visibles para las aplicaciones. Específicamente, la aplicación sólo es consciente de los errores que hacen fallar todas las réplicas de un *rank* particular.
- Los requerimientos de Entrada/Salida se reducen significativamente, debido a que el incremento del MTTI reduce la velocidad a la que se deben escribir los *checkpoints* para permitir a las aplicaciones que utilicen eficientemente el sistema.
- La comparación de los estados de las múltiples réplicas (por ejemplo, utilizando sumas de comprobación de la memoria), previo a la escritura de un *checkpoint*, permite al sistema de replicación detectar si el estado de la aplicación ha sido corrompido (es decir, detectar un *soft error*) y relanzar la aplicación desde un *checkpoint* anterior.
- Los nodos extra que se utilizan como redundancia pueden utilizarse como potencia de cómputo adicional cuando el sistema ejecuta muchas tareas menores, en las que la tolerancia a fallos no es un problema mayor.

Como se mencionó en la sección 5.3, para estudiar el *overhead* introducido por la replicación de procesos, los autores de [53] diseñaron rMPI, una biblioteca MPI de nivel de usuario portable, que proporciona ejecución redundante para aplicaciones MPI determinísticas de manera transparente. rMPI está implementada sobre una versión de MPI existente, utilizando funcionalidad de *profiling*, y se utiliza en un sistema de gran escala (Cray XT-3/4), con algunas aplicaciones reales, para medir el *overhead* en tiempo de ejecución de la implementación.

La idea básica de diseño de rMPI es replicar cada *rank* de la aplicación y permitir a las réplicas que continúen en caso de que un *rank* original falle. Para garantizar que las réplicas mantienen estados consistentes, rMPI implementa protocolos que aseguran que los mensajes están ordenados idénticamente entre las réplicas. A diferencia de otros protocolos de replicación más generales [81,88], estos protocolos son específicos para las necesidades de MPI, con el objetivo de reducir el *overhead* en tiempo de ejecución. Una descripción completa de rMPI, que incluye los protocolos de bajo nivel y detalles de la implementación se encuentra en [89,90].

Los autores de [53] utilizan una combinación de modelado, experimentación y simulación para evaluar las ventajas y desventajas de la replicación de procesos para un rango de parámetros del sistema, incluyendo los costos de hardware y software para aplicaciones MPI, diferentes distribuciones de fallos, anchos de banda de E/S y rangos de MTTI´s. Sus resultados muestran que, en un amplio rango de diseño de sistemas de exa-escala (aunque no para todos los diseños), la replicación de procesos mejora a las soluciones basadas en *checkpoint/restart*. En particular, la replicación es viable en los casos en los que el número de procesadores es grande pero el ancho de banda de E/S es limitado. Sin embargo, en diseños con MTBF´s por procesador mayor a 50 años, con menos de 50.000 procesadores y con ancho de banda de *checkpoint* de 30 TB/s, la replicación no tiene tan buen desempeño.

Como trabajo pendiente, se requiere cuantificar el costo de software que implica detectar errores silenciosos (algo que no está claro en aplicaciones de HPC), debido a que las pruebas realizadas en [53] sólo buscan medir el costo de utilizar la replicación para proteger al sistema de errores que provocan la caída de la operación.

Todos los mecanismos descriptos en este capítulo se enfocan en el soporte para fallos que provocan que los procesos se aborten o finalicen abruptamente. En el capítulo 6 se introduce SMCV, una estrategia de detección de fallos diseñada específicamente para detectar fallos transitorios en aplicaciones paralelas científicas que utilizan paso de mensajes.

Capítulo 6

Metodología SMCV para detección de fallos transitorios

En este capítulo se describe detalladamente SMCV (*Sent Message Content Validation*) [1,2], una técnica completamente distribuida que proporciona detección de fallos para aplicaciones paralelas de paso de mensajes mediante la validación de los contenidos de los mensajes que se van a enviar. De esta forma, evita que los fallos que se producen en el ámbito de un proceso se propaguen a los demás procesos de la aplicación y restringe la latencia de detección y notificación de la ocurrencia del error. Por otra parte, aprovecha la redundancia intrínseca de hardware que es propia de la arquitectura de cluster de multicores. La técnica SMCV alcanza un alto nivel de robustez frente a fallos introduciendo un *overhead* reducido. Además, logra un compromiso entre una latencia de detección moderada y una baja sobrecarga de operación.

6.1 Fundamentación

SMCV es una propuesta diseñada específicamente para detectar fallos transitorios en aplicaciones científicas paralelas determinísticas que utilizan paso de mensajes, y que se ejecutan sobre los nodos de un cluster de multicores. SMCV es una técnica basada puramente en software, que busca utilizar la redundancia de hardware propia de los multicores en beneficio del sistema, ejecutando las réplicas de los procesos de la aplicación paralela en cores del mismo procesador. La detección se realiza mediante la validación de los contenidos de los mensajes que se van a enviar a otros procesos de la aplicación, utilizando un intervalo de validación moderado, e introduciendo una reducida sobrecarga de trabajo y un bajo *overhead* respecto del tiempo de ejecución. Es una estrategia distribuida, que mejora la fiabilidad del sistema compuesto por el cluster y la aplicación paralela que se ejecuta sobre él, confinando el efecto de los fallos que ocurren en el contexto de un proceso y evitando que se transmita hacia los demás procesos. El objetivo principal es que las aplicaciones que logren finalizar lo hagan con resultados correctos. En particular, SMCV es capaz de detectar todos los fallos que producen TO y SDC (tanto TDC como FSC).

A diferencia de otras propuestas basadas puramente en software, descriptas en el capítulo 2 (como por ejemplo SWIFT [4] o PLR [13]), que están diseñadas para programas secuenciales, SMCV es específica para aplicaciones paralelas de paso de mensajes. Por otra parte, en comparación a las propuestas que se utilizan en entornos MPI, descriptas en el capítulo 5 (como por ejemplo MPI/FT [40] o FT-MPI [75]), que soportan fallos que producen la caída de los procesos, SMCV provee un mecanismo para detectar fallos transitorios. Al momento de la investigación que llevó al desarrollo de la metodología que da origen a este trabajo, no se conocía ninguna propuesta específica para detectar fallos transitorios en aplicaciones paralelas científicas de paso de mensajes.

En las siguientes subsecciones se describen los fundamentos sobre los que se basa SMCV, la hipótesis de partida y la solución propuesta. En primer lugar, se explica la utilidad de validar los contenidos de los mensajes y de comparar los resultados finales, y luego, se describe el aprovechamiento de recursos redundantes de hardware para mejorar la fiabilidad del sistema.

6.1.1 Validación de contenidos de mensajes antes de enviar

La metodología de detección propuesta en este trabajo se basa esencialmente en la hipótesis de que, en un sistema formado por un cluster de multicores sobre el que se ejecuta una aplicación paralela de paso de mensajes, la mayor parte del cómputo significativo (entendido como el que incide sobre los resultados de la aplicación) forma parte del contenido de un mensaje que se envía a otro proceso de la aplicación en algún momento de la ejecución.

Como se explicó en el capítulo 1, los fallos pueden corromper datos, direcciones, códigos de operación o *flags*. Sin embargo, si el valor alterado es relevante para el resultado de la aplicación, la situación eventualmente se reflejará en la incorrección de un mensaje. Por lo tanto, de todos los fallos que pueden producir SDC, la mayor parte caen en la categoría de TDC (ver sección 1.8.3). En consecuencia, es necesario monitorizar el contenido de los mensajes para detectar errores que afectan datos importantes.

Tomando este hecho en cuenta, SMCV es una estrategia de que se basa en validar los contenidos de los mensajes que se van a enviar. Para ello, cada proceso de la aplicación se duplica, y ambas réplicas comparan todos los campos que conforman el mensaje previo al envío; el mensaje se envía sólo si la comparación resulta exitosa. De esta manera, la utilización de SMCV permite detectar todos los fallos que causan TDC; desde el punto de vista de la aplicación paralela, garantiza que la ocurrencia de un fallo que afecta el estado de un proceso no se propaga a ningún otro proceso de la aplicación, confinando su efecto al proceso local. Cuando ocurre un error, SMCV (en el estado actual de desarrollo) notifica a la aplicación y conduce a una parada segura.

Como se mencionó, los contenidos de los mensajes son validados previamente al envío. Por ende, sólo una de las réplicas envía efectivamente el mensaje, lo que implica que no se consume ancho de banda de red adicional. Si se considera que las redes actuales utilizan protocolos que garantizan comunicaciones confiables, no es necesario validar los contenidos de los mensajes al momento de la recepción (lo que provocaría tener que transmitir dos copias del mensaje).

6.1.2 Comparación de resultados finales

En la sección 1.8.3 se mencionó que, en una aplicación paralela, una fracción de los fallos que ocurren alteran datos que no se comunican a otros procesos, sino que se propagan localmente produciendo resultados incorrectos sólo en el proceso afectado. La consecuencia de estos fallos es la FSC, debido a que se comportan de la misma forma que los fallos que ocurren durante la ejecución de aplicaciones secuenciales.

Por lo tanto, si se agrega una instancia de validación de los resultados finales del programa, es posible detectar los fallos que afectan a la fase serie de la aplicación, en la que no se producen comunicaciones. SMCV incorpora esta verificación final para garantizar la fiabilidad del sistema y, por lo tanto, que los resultados de todas las aplicaciones que consiguen llegar al final de su ejecución (sin que se produzca parada segura) son los correctos. Dicho de otra forma, ninguna aplicación paralela, que cumpla con los requisitos necesarios para poder utilizar SMCV, finalizará su ejecución con resultados erróneos.

6.1.3 Aprovechamiento de recursos redundantes de hardware

La tendencia en la fabricación de hardware es a incorporar un mayor número de cores a los procesadores. Sin embargo, una gran cantidad de aplicaciones no son capaces de sacar provecho a todos los recursos de cómputo en forma eficiente. Por otra parte, el incremento de la cantidad de fallos transitorios va de la mano con el crecimiento del número de núcleos de procesamiento. Como consecuencia, el foco que se ha hecho tradicionalmente en la *performance* del procesador se ha corrido, y factores como la fiabilidad y la disponibilidad se han vuelto más relevantes. Por lo tanto, el uso de cores de procesamiento para realizar tareas relacionadas con la tolerancia a fallos tiene dos grandes ventajas: mejora la utilización de los recursos e incorpora una característica que es beneficiosa para el sistema.

En este contexto, SMCV aprovecha la redundancia de hardware que existe intrínsecamente en los multicores, utilizando cores del CMP para ejecutar las réplicas de los procesos que realizan cómputo útil para la aplicación. De esta forma, la mitad de los recursos de cómputo se utilizan como redundancia. El acceso a memoria principal es el aspecto crítico al momento de elegir los cores que se van a utilizar para detectar los fallos que ocurren en los demás. SMCV busca explotar la jerarquía de memoria del CMP, de forma que la replicación del cómputo que se lleva a cabo en un core determinado se asigna a otro core con el que existe algún nivel de caché compartido. Por lo tanto, muchas operaciones de comparación entre réplicas se resuelven en el último nivel de caché, de manera que se minimiza el acceso a la memoria principal. En la figura 6.1 se muestra esta forma de utilizar el hardware para detectar fallos.

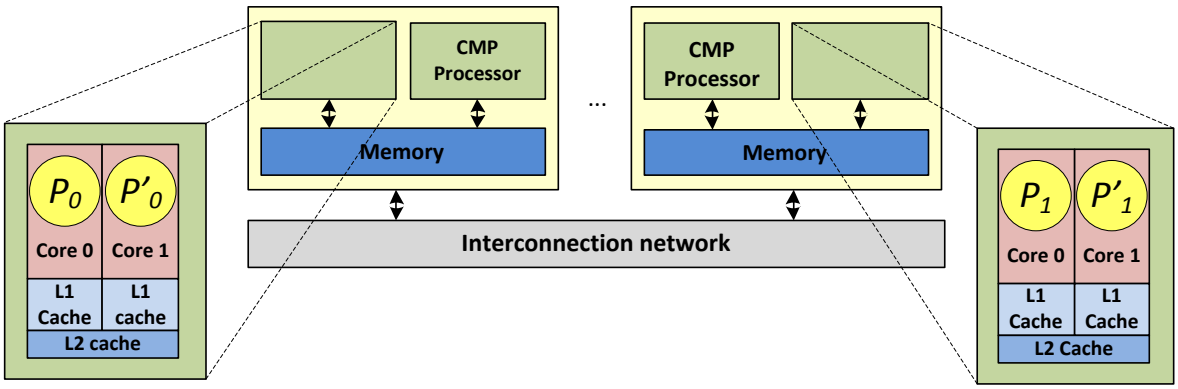

Fig. 6.1: ubicación de las réplicas de los procesos de la aplicación para aprovechar el hardware redundante

6.2 Descripción de la operación

Como se mencionó anteriormente, SMCV es una estrategia basada en software, capaz de detectar fallos transitorios que ocurren en los procesadores de clusters de multicores sobre los que se ejecutan aplicaciones paralelas científicas de paso de mensajes. Cuando se detecta un fallo, se reporta al usuario y se detiene la aplicación, mejorando la fiabilidad del sistema.

En la figura 6.2(a) muestra un esquema de la metodología de detección propuesta, mientras que en la 6.2(b) se observa el comportamiento frente a la ocurrencia de un fallo. Cada proceso de la aplicación paralela se ejecuta sobre un core del CMP, y el cómputo que realiza es replicado en un *thread*, que a su vez se ejecuta en un core que comparte algún nivel de caché con el core en el que se ejecuta el proceso original. Por ende, no es necesario el acceso a la memoria principal, aprovechándose la jerarquía de memoria para resolver las comparaciones.

Cada proceso se ejecuta concurrentemente con su réplica, lo que implica la necesidad de un mecanismo de sincronización entre ellas. Cuando se va a realizar una operación de comunicación (ya sea punto a punto o colectiva), el proceso se detiene temporariamente y espera a que su réplica alcance el mismo punto de su ejecución. Una vez allí, todos los campos del mensaje que se va a enviar son verificados, byte a byte, para validar que los contenidos calculados por ambas réplicas son los mismos. Sólo si se cumple esta condición, una de las réplicas envía el mensaje, asegurando de esta forma que ningún dato corrompido se propaga a los demás procesos.

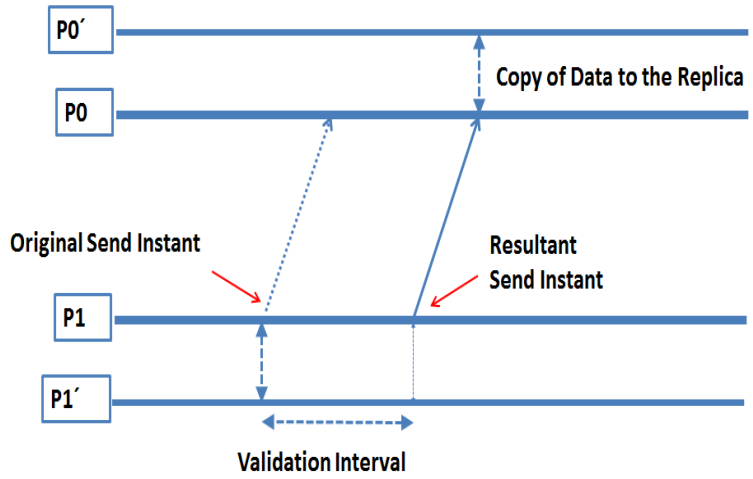

Fig. 6.2(a): Esquema de detección de SMCV

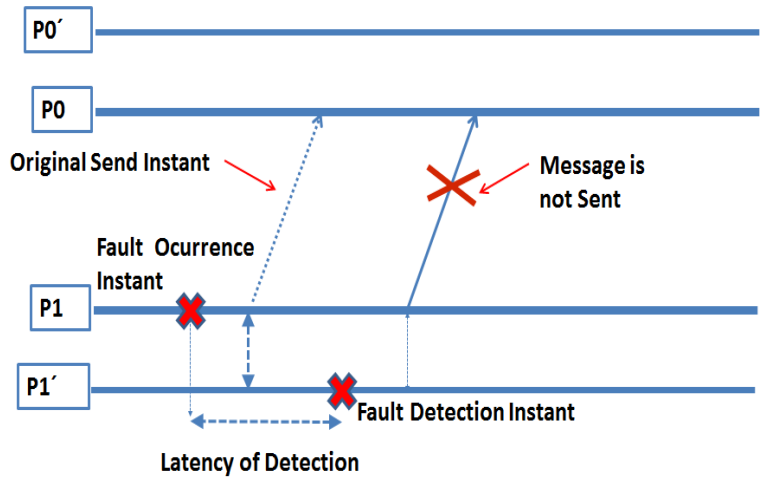

Fig. 6.2(b): comportamiento de SMCV en presencia de un fallo transitorio

El proceso que recibe el mensaje se detiene en el instante de la detección y se mantiene en espera hasta que su réplica llegue al mismo punto. Una vez allí, recibe el mensaje, copia el contenido a la réplica (valiéndose también de la jerarquía de memoria) y ambas réplicas continúan con su cómputo. Si se asume que los errores de la red son detectados y corregidos por el software de red, los mensajes validados llegan a destino sin alteraciones. Al verificar los mensajes antes de enviarlos, éstos pueden enviarse sólo una vez, mientras que si fueran validados al momento de la recepción se

requeriría el envío de dos copias por la red, lo cual es perjudicial para el ancho de banda y para la vulnerabilidad de la red.

Finalmente, cuando la ejecución concluye, los resultados obtenidos son verificados para detectar fallos que pudieran haber ocurrido en la fracción serie de la aplicación, o luego de la finalización de todas las comunicaciones.

6.2.1 Caracterización de la sobrecarga de trabajo

La carga de trabajo adicional está relacionada con la cantidad de cómputo agregada por la inclusión del mecanismo de detección de fallos. Esta métrica es útil para comparar la metodología SMCV con otras alternativas. Para tener una aproximación, se analiza una estrategia conservativa, similar a las que se utilizan para aplicaciones secuenciales [7], que se basa en la validación de todas las operaciones de escritura en memoria. En este caso, los procesos de aplicación paralela también son duplicados en *threads* de la manera descripta, pero se verifican los resultados de todas las operaciones de escritura en memoria (a diferencia de SMCV, donde sólo se verifican los contenidos de los mensajes). Esta estrategia es capaz de detectar todos los fallos, pero a costa de un incremento significativo en la cantidad de cómputo.

La sobrecarga de operación W_{WV} introducida por la técnica de validación de escrituras está dada por:

$$W_{WV} = (S + M.k).(C_{sync} + C_{comp})$$

Ecuación 9: sobrecarga de trabajo introducida por la validación de escrituras

En la ecuación 9, S representa la cantidad de operaciones de escritura realizadas por la aplicación, sin contar aquellas que corresponden a los mensajes que envía. Se asume que la aplicación envía M mensajes de k elementos (en promedio) cada uno. C_{sync} y C_{comp} representan los costos de una operación de sincronización y de una operación de comparación respectivamente. Por lo tanto, el primer factor de la ecuación 9 es la cantidad total de operaciones de escritura que realiza la aplicación. Si se verifican todas las escrituras, cada una de ellas conlleva una operación de sincronización y una de comparación.

En tanto, la sobrecarga de trabajo W_{MV} introducida por la validación de mensajes está dada por:

$$W_{MV} = M.(C_{sync} + k.C_{comp})$$

Ecuación 10: sobrecarga de trabajo introducida por la validación de mensajes

En la ecuación 10, para cada mensaje hay una única operación de sincronización y k operaciones de comparación (una para cada elemento del mensaje).

La relación entre la sobrecarga introducida por SMCV y la estrategia que valida todas las operaciones de escritura, entonces, viene dada por:

$$\frac{W_{MV}}{W_{WV}} = \frac{M\ .\ C_{sync}\ +\ M\ .\ k\ .\ C_{comp}}{S\ .\ \left(C_{sync}\ +\ C_{comp}\right)\ +\ M\ .\ k\ .\ C_{sync}\ +\ M\ .\ k\ .\ C_{comp}}$$

Ecuación 11: relación entre la sobrecarga de trabajo introducida por la validación de mensajes y la introducida por la validación de escrituras

El cociente de la ecuación 11 es siempre un número menor que 1, lo que significa que la técnica de validación de mensajes introduce una cantidad menor de cómputo adicional que la de validación de todas las escrituras.

El análisis anterior es válido para uno de los procesos que comunican sus resultados. En el caso de un proceso que realiza cómputo secuencial, se debe agregar la sobrecarga debida a la comparación de los resultados finales; sin embargo, ésta es la misma para ambas técnicas. Por lo tanto, el análisis es suficientemente general y representativo de varias situaciones. A partir de esto, puede afirmarse que SMCV es una técnica liviana, que agrega una sobrecarga de operación reducida frente a estrategias más conservadoras que detectan fallos que no tienen impacto en los resultados de la aplicación.

6.2.2 SoR de SMCV y vulnerabilidad

Antes de abordar directamente el nivel de ejecución redundante de SMCV, conviene repasar y tener en claro dos ideas fundamentales: en primer lugar, SMCV es capaz de detectar cualquier fallo transitorio simple que cause SDC (ya sea TDC o FSC) o TO, como se mencionó en la sección 6.1. SMCV no soporta fallos múltiples, en el sentido en el que se describen en la sección 2.7: si ocurren dos fallos que afectan al mismo bit del mismo dato en ambas réplicas, o si ocurre un fallo durante la verificación que oculta el efecto de un fallo anterior en el cómputo, no se detectarán estos fallos. En segundo lugar, aunque SMCV es una técnica de software puro, y por lo tanto se adopta una SoR centrada en software, de manera similar a [13] (ver sección 2.5), se debe tener presente que los fallos ocurren en el hardware, aunque sólo tienen relevancia los que repercuten sobre la ejecución de la aplicación. El objetivo de SMCV es detectar fallos que ocurren durante el cómputo sobre los datos que se manipulan dentro del procesador, específicamente en los registros, que como se menciona en las en las secciones 1.2 y 2.4, constituyen la parte más vulnerable del sistema de cómputo, debido a la dificultad de implementar mecanismos de protección por hardware para ellos. Esto implica, entre otras cosas, que la memoria se encuentra por fuera de la SoR de SMCV.

Como se explicó en la sección 6.2, SMCV replica en un *thread* todo el cómputo que realiza cada proceso de la aplicación paralela. Cada *thread* opera sobre una copia local de los datos, es decir, se genera una copia del subconjunto de los datos de entrada para que el *thread* realice el cómputo de forma independiente del proceso al cual replica. Por lo tanto, la SoR se encuentra alrededor de la aplicación del usuario y sus datos. No incluye al sistema operativo ni a la biblioteca de comunicaciones.

Un caso particular lo constituyen los fallos que afectan a variables compartidas por varios procesos de la aplicación paralela. En este caso, el proceso y su réplica leen y utilizan el mismo valor erróneo para sus cálculos, por lo que, si la ejecución subsiguiente es correcta y libre de fallos, SMCV no detecta ningún error. Sin embargo, un fallo que afecta a una variable compartida es un fallo de memoria, y se asume que la memoria está protegida eficientemente por mecanismos como los ECC´s, por lo que se encuentra fuera de la SoR, como se mencionó anteriormente. Sin embargo,

dentro de lo posible, es aconsejable evitar el uso de variables compartidas para disminuir la probabilidad de fallos en puntos centralizados.

Respecto de la vulnerabilidad de la técnica SMCV, de manera similar al caso de [13], está asociada a la ejecución del mecanismo de detección. SMCV minimiza los retardos entre la comprobación y la utilización de los valores validados, debido a que la verificación se realiza al momento de utilizar los datos para enviar un mensaje. Una vez que los datos que se van a enviar están en el *buffer* de salida ya se encuentran por fuera de la SoR.

En tanto, la comprobación de valores a enviar resulta un punto centralizado de falla. Las situaciones que se pueden producir, derivadas de este hecho, son las siguientes:

i) Ejecución correcta, comprobación correcta: en este caso no se ha producido hasta el momento ningún fallo durante la ejecución (salvo quizás algún fallo que aún permanezca latente).
ii) Ejecución correcta, comprobación incorrecta: en este caso ha ocurrido una falla durante la comprobación, por lo que se ha producido un falso positivo. La aplicación va a una parada segura cuando en realidad no ha ocurrido ningún fallo en ejecución, sino que ha sido introducido por el mismo mecanismo de detección. Esta es una vulnerabilidad clara, que puede solucionarse si se duplica la comparación, lo cual introduce un *overhead* que es necesario determinar. La técnica SMCV, en su estado actual, no contempla esta duplicación.
iii) Ejecución incorrecta, comprobación incorrecta: este caso corresponde a la operación normal, en la cual el fallo ocurrido durante la ejecución es detectado en el momento de la comparación.
iv) Ejecución incorrecta, comprobación correcta: en esta situación, un fallo ocurrido durante la ejecución resulta oculto tras la comprobación, debido a un segundo fallo que "compensa" o "anula" al primero. Como se mencionó en esta misma sección, SMCV no puede lidiar con un doble fallo de este tipo, sino que sólo soporta fallos simples. De todas formas, como se comenta en la sección 2.7, la probabilidad de que ocurran dos fallos que se combinen de esta forma tan particular es extremadamente baja.

Es importante tomar en consideración el hecho de que SMCV es capaz de detectar como TO´s otros fallos que entrarían en la ventana de vulnerabilidad si no se contara con dicho mecanismo. Si el código de una instrucción fuera modificado por un fallo, de modo que la instrucción resultante fuera el envío de un mensaje, o si ocurre un fallo durante la ejecución del código de la herramienta de detección, o si un fallo produjera que se ejecute código de la herramienta en un momento erróneo, ambas réplicas separarán sus flujos de ejecución y, ante el envío de un mensaje por parte de una de ellas, no se sincronizarán adecuadamente, por lo que el fallo se detectará al transcurrir un lapso mayor al determinado.

6.2.3 Comportamiento frente a fallos

En esta sección se define y formaliza completamente el comportamiento de la metodología de detección SMCV. En la figura 6.3(a) se muestra un diagrama de los estados posibles de la ejecución de una aplicación cuando no hay implementada ninguna estrategia de detección, mientras que en la figura 6.3(b) se observa el mismo diagrama pero cuando se aplica la metodología SMCV para detectar fallos transitorios. Las elipses representan estados posibles, mientras que los arcos representan eventos que producen transiciones de un estado a otro. Por cuestiones de claridad, las

transiciones aparecen numeradas en los diagramas. En consecuencia, se muestra una descripción de cada evento posible que produce un cambio de estado.

1. El bit afectado no se utiliza
2. El bit afectado es utilizado por la aplicación
3. El bit alterado afecta datos controlados por el Sistema Operativo
4. El bit alterado afecta datos de la aplicación del usuario
5. El bit alterado produce que la aplicación no responda al cabo de un tiempo máximo
6. El Sistema Operativo detecta el fallo y aborta la aplicación
7. El dato afectado se transmite a otro proceso de la aplicación paralela
8. El dato afectado sólo es utilizado por el proceso local
9. Transcurso de tiempo
10. SMCV detecta el fallo al cabo de un tiempo y conduce a una parada segura

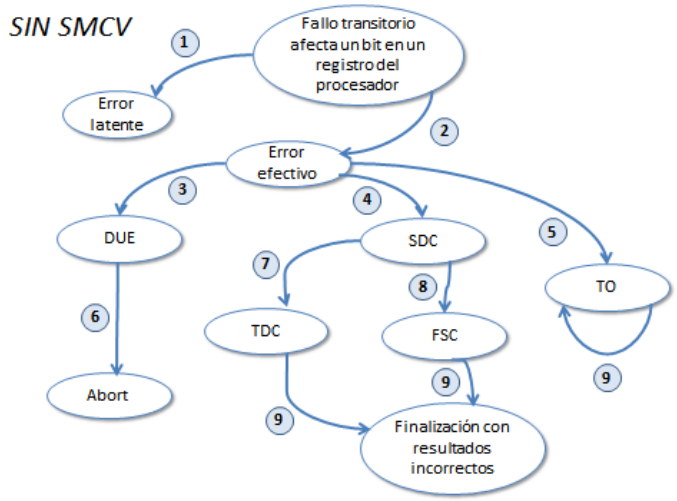

Fig. 6.3(a): diagrama de estados de la ejecución sin estrategia de detección de fallos

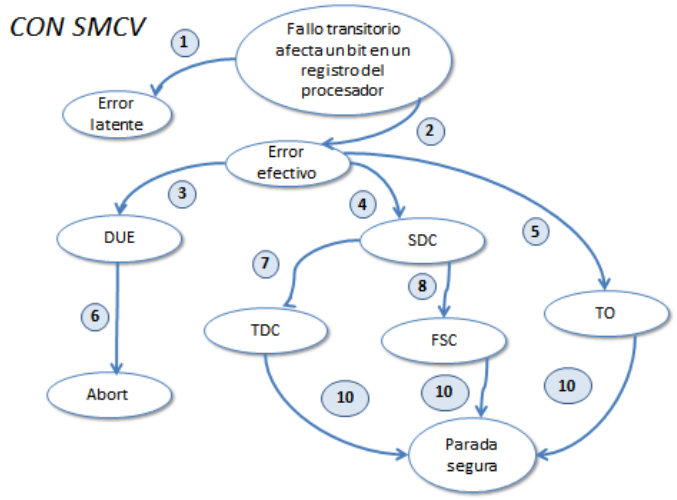

Fig. 6.3(a): diagrama de estados de la ejecución aplicando la estrategia de detección SMCV

6.3 Implementación de la herramienta de detección SMCV

En esta sección se describe una herramienta desarrollada para implementar la metodología SMCV, con el objetivo de facilitar su usabilidad. Consiste en una biblioteca de funciones y tipos de datos MPI modificados, con funcionalidad extendida para detectar fallos mediante comparación al momento del envío, duplicación de contenidos de los mensajes en la recepción y control de concurrencia entre las réplicas. La herramienta SMCV intenta ayudar tanto al programador como al usuario de aplicaciones científicas paralelas a lograr ejecuciones fiables, obteniendo resultados correctos o, al menos, reportando la ocurrencia de fallos silenciosos e impidiendo sus consecuencias, conduciendo a una parada segura. Esto permite evitar la innecesaria y costosa espera hasta la finalización de la ejecución, posibilitando el relanzamiento de la ejecución luego de un retardo acotado debido a la latencia de detección. Esta es una característica importante, a causa de que este tipo de aplicaciones suelen tener tiempos de ejecución muy prolongados.

En su versión actual, la biblioteca SMCV puede ser utilizada con aplicaciones MPI desarrolladas en lenguaje C. Redefine una parte de las funciones y tipos de datos de la biblioteca MPI, realizando un único cambio sintáctico: el prefijo `MPI` se reemplaza por el prefijo `SMCV`. Además, agrega dos funciones nuevas: `SMCV_Call` y `SMCV_Validate`. Utiliza funciones de la biblioteca Pthreads para la replicación de *threads*, y la sincronización entre ellos se realiza por medio de semáforos de la biblioteca. La redefinición de las funciones de MPI es necesaria para proporcionar funcionalidad de detección de fallos de forma transparente al código de las aplicaciones y a sus programadores. Este hecho implica la necesidad de modificación del código fuente de la aplicación y su recompilación.

6.3.1 Funciones básicas

Como se mencionó en la sección 4.2, el estándar MPI provee seis funciones básicas [67]. El núcleo de la biblioteca SMCV consiste en las redefiniciones de estas seis funciones básicas y dos funciones adicionales. Las funciones básicas de SMCV se describen en la tabla 6.1.

Función SMCV	Objetivo
SMCV_INIT	Inicializa el entorno SMCV. Esta función debe utilizarse antes de llamar a cualquier otra función SMCV.
SMCV_FINALIZE	Finaliza la ejecución de los procesos que forman una sesión SMCV
SMCV_COMM_SIZE	Devuelve el número de procesos de la sesión SMCV actual (asociados a un comunicador)
SMCV_COMM_RANK	Permite determinar el identificador (*rank*) del proceso que invoca a la función
SMCV_CALL	Crea un nuevo *thread* que ejecuta el código que debe validarse
SMCV_SEND	Sincroniza un proceso y su réplica. El que llega después al punto de sincronización compara todos los campos del mensaje a enviar (byte a byte). Si todos coinciden, el primer *thread* envía el mensaje. Una vez enviado, ambos *threads* continúan su ejecución. Si algún campo difiere,

	se produce una parada segura. Además, existe un lapso de tiempo (configurable) para que el segundo *thread* llegue al punto de sincronización, de forma de interceptar los TO´s.
SMCV_RECV	Sincroniza un proceso y su réplica. El que llega primero al punto de sincronización recibe el mensaje y permanece en espera. Cuando el segundo *thread* llega, se copia el contenido del mensaje recibido. Luego, ambos *threads* continúan su ejecución. También existe un lapso de tiempo (configurable) para que el segundo *thread* llegue al punto de sincronización, de forma de interceptar los TO´s.
SMCV_VALIDATE	Sincroniza un proceso y su réplica. El que llega después al punto de sincronización compara los resultados finales de ambos *threads* (byte a byte). Si coinciden, ambos *threads* continúan su ejecución. Sino, se produce una parada segura. También existe un lapso de tiempo (configurable) para que el segundo *thread* llegue al punto de sincronización, de forma de interceptar los TO´s.

Tabla 6.1: Funciones SMCV básicas

6.3.2 Utilización

Para integrar la funcionalidad de SMCV al código de una aplicación MPI, se deben seguir los pasos siguientes:

1. Reemplazar el encabezado de MPI por el encabezado de SMCV.
2. Encapsular el código que se debe validar (datos e instrucciones) en una función `void *`.
3. Realizar una llamada a la función `SMCV_Call`, pasándole la función definida en el paso anterior como argumento.
4. Reemplazar el prefijo `MPI` por `SMCV` en todas las funciones y tipos de datos MPI.
5. Realizar una llamada a la función `SMCV_Validate` para validar el resultado final de la aplicación.

La figura 6.4 muestra un ejemplo de cómo debe realizarse esta adaptación.

```
#include <mpi.h>
int main (int argc, char **argv)
{
   MPI_Init();
   /* Process data, instructions
and MPI functions */
   MPI_Finalize();
   return 0;
}
```

```
#include <smcv.h>
int main (int argc, char **argv)
{
   SMCV_Init();
   SMCV_Call(&smcv_process)
   SMCV_Finalize();
   return 0;
}
void * smcv_process () {
   /* Thread data, instructions
and SMCV functions */
   SMCV_Validate();
}
```

Fig. 6.4: ejemplo de adaptación de una aplicación MPI para incorporar la funcionalidad SMCV. Izquierda: código fuente de la aplicación MPI. Derecha: código fuente de la aplicación MPI adaptado para SMCV

6.4 Validación experimental

La metodología SMCV ha sido validada experimentalmente para determinar su eficacia en la detección de fallos transitorios que ocurren cuando se ejecutan aplicaciones paralelas científicas de paso de mensajes en un cluster de multicores. Los fallos detectados son los que causan TDC, FSC y TO. Además, se ha evaluado el *overhead* que introduce respecto del tiempo de ejecución de la aplicación. En esta sección se muestran las pruebas realizadas y los resultados obtenidos.

6.4.1 Arquitectura de prueba

Todo el trabajo experimental fue llevado a cabo en un cluster de multicores Blade con cuatro hojas. Cada hoja tiene dos procesadores de 4 núcleos (*Quad-Core*) Intel Xeon e5405 que operan a 2GHz, con 64KB de memoria caché L1 privada, 6 MB de memoria caché L2 compartida entre pares de cores, 10 GB de memoria RAM compartida entre los dos procesadores que componen la hoja y 250 GB de disco local. El sistema operativo es GNU/Linux Debian 6.0.7 de 64 bits. La versión de *kernel* es la 2.6.32 y la biblioteca de comunicaciones es OpenMPI versión 1.6.4.

6.4.2 Verificación de la eficacia de detección

Se realizaron una serie de pruebas para verificar la eficacia de detección de SMCV. La aplicación utilizada para ello fue una multiplicación de matrices paralela (C = A * B), programada según el paradigma *Master/Worker* con 5 procesos (el *Master* y 4 *Workers*), en la que el proceso *Master* también toma parte en el cómputo de la matriz resultado C [25]. Todas las comunicaciones MPI utilizadas son bloqueantes. La descripción de la operación de la aplicación es la siguiente:

- El proceso *Master* divide la matriz A entre todos los *Workers*, y envía a cada uno su trozo correspondiente de ella, manteniendo el mismo *Master* un trozo para participar también él en el cálculo de la matriz resultado. Esto se realiza mediante la función *MPI_Scatter*.
- El *Master* envía a cada *Worker* una copia de la matriz B completa (y se queda él mismo con una copia). Esto se realiza mediante la función *MPI_Broadcast*.
- Todos los procesos computan su trozo correspondiente de la matriz C, y, cuando finalizan, envían la parte que han calculado al proceso *Master*. Esto se realiza mediante la función *MPI_Gather*.
- El *Master* construye la matriz C a partir de lo que los *Workers* le han enviado y lo que él mismo ha calculado.

Para la validación, se adaptó la aplicación para integrarla con la funcionalidad de la metodología SMCV, replicando cada uno de sus procesos en un *thread* de la forma descripta en la sección 6.3.2. Para esto se requiere modificar el código fuente de la aplicación y la posterior recompilación.

El experimento consistió en inyectar fallos, de manera controlada, en varios puntos de la aplicación, por medio de una herramienta de depuración o *debugging* (GDB, el depurador de Linux). Para hacer esto, se inserta un *breakpoint* en algún punto de la ejecución de uno de los procesos de la aplicación, se lee el valor de una variable, se modifica y se retoma la ejecución con el valor alterado, de forma de poder analizar las consecuencias de esta modificación al final de la ejecución. De esta manera se simula un fallo transitorio simple en un registro interno del procesador, debido a que la

corrupción de un dato se manifiesta si puede observarse como una diferencia entre los estados de las réplicas en memoria.

Pese a que un fallo transitorio puede ocurrir aleatoriamente en cualquier lugar y momento durante la ejecución, para la inyección simulada controlada se seleccionaron puntos significativos del procesamiento, tanto en el cómputo realizado por el *Master* como el de los *Workers*.

Para los experimentos se utilizó la aplicación de multiplicación de matrices descripta, en la que el tamaño de las matrices cuadradas es de 10 x 10 (con todos sus elementos inicializados con el valor 1). Por lo tanto, cada uno de los 5 *Workers* debe procesar 2 filas (20 elementos) de la matriz A, y producir como resultado 2 filas (20 elementos) de la matriz C.

En la figura 6.5 se ve una ejecución normal de la aplicación, sin inyectar ningún fallo. En este caso la salida es correcta. Se puede observar un conteo inicial, que es el lapso de tiempo que se utiliza en los experimentos de inyección para adjuntar el depurador a alguno de los procesos de la aplicación, de manera de simular la ocurrencia de un fallo que afecta un dato utilizado por dicho proceso. En tanto, en la figura 6.6 se muestra la forma en la que se adjunta el depurador GDB para realizar los experimentos de inyección de fallos.

```
diego@Lidi137:~/Dropbox/diego/Para trabajo de Especialización/Experimentos$ mpirun -np 5 mm-SMCV 10
PID 4583 on 0 ready for attach
PID 4586 on 3 ready for attach
PID 4584 on 1 ready for attach
PID 4587 on 4 ready for attach
Restan 10 segundos...
PID 4585 on 2 ready for attach
Restan 9 segundos...
Restan 8 segundos...
Restan 7 segundos...
Restan 6 segundos...
Restan 5 segundos...
Restan 4 segundos...
Restan 3 segundos...
Restan 2 segundos...
Restan 1 segundos...
MM-SMCV;5;10;11.065258;11.049720;0.015538
diego@Lidi137:~/Dropbox/diego/Para trabajo de Especialización/Experimentos$
```

Fig. 6.5: salida de una ejecución sin fallos. Se muestra el tiempo reservado para adjuntar el depurador

En la figura 6.7(a) se muestra el procedimiento realizado para inyectar un fallo. En este ejemplo, el fallo se inyecta durante la operación del proceso *Master* (Rank = 0), en uno de los primeros 20 elementos de la matriz A (que son los que el *Master* conserva para realizar su cómputo local), después de la ejecución de la función *MPI_Scatter*, pero antes de la multiplicación. Esta situación simula la ocurrencia un fallo que corrompe un registro en el cual hay un dato que interviene en el cómputo del resultado, pero que nunca es transmitido a otro proceso de la aplicación, produciendo FSC. En la figura 6.7(b) se ve la salida de la aplicación, con la detección del error y la parada segura.

En la figura 6.8(a) se muestra la inyección de un fallo que produce TDC. Para esto, el fallo se inyecta durante la operación de uno de los procesos *Workers* (Rank ≠ 0), en un elemento cualquiera de la matriz B, después de la ejecución de la función *MPI_Broadcast*, pero antes de la multiplicación. De esta forma, se simula un fallo que corrompe un registro en el cual hay un dato que interviene los cálculos que realiza ese *Worker* (que fue recibido correctamente con anterioridad en el *MPI_Broadcast*). Los resultados de estos cálculos (las dos filas de C que le corresponden a ese *Worker*) se transmiten en el *MPI_Gather* posterior, por lo que un resultado incorrecto (calculado a partir del valor alterado) es detectado como TDC. En la figura 6.8(b) nuevamente se muestra la salida de la aplicación, con detección del error y

parada segura. Como en este caso el fallo inyectado ha causado TDC, el mensaje de error que se muestra a la salida es diferente del caso anterior.

Fig. 6.6: ejemplo de cómo adjuntar el depurador a la ejecución de la aplicación para poder inyectar fallos

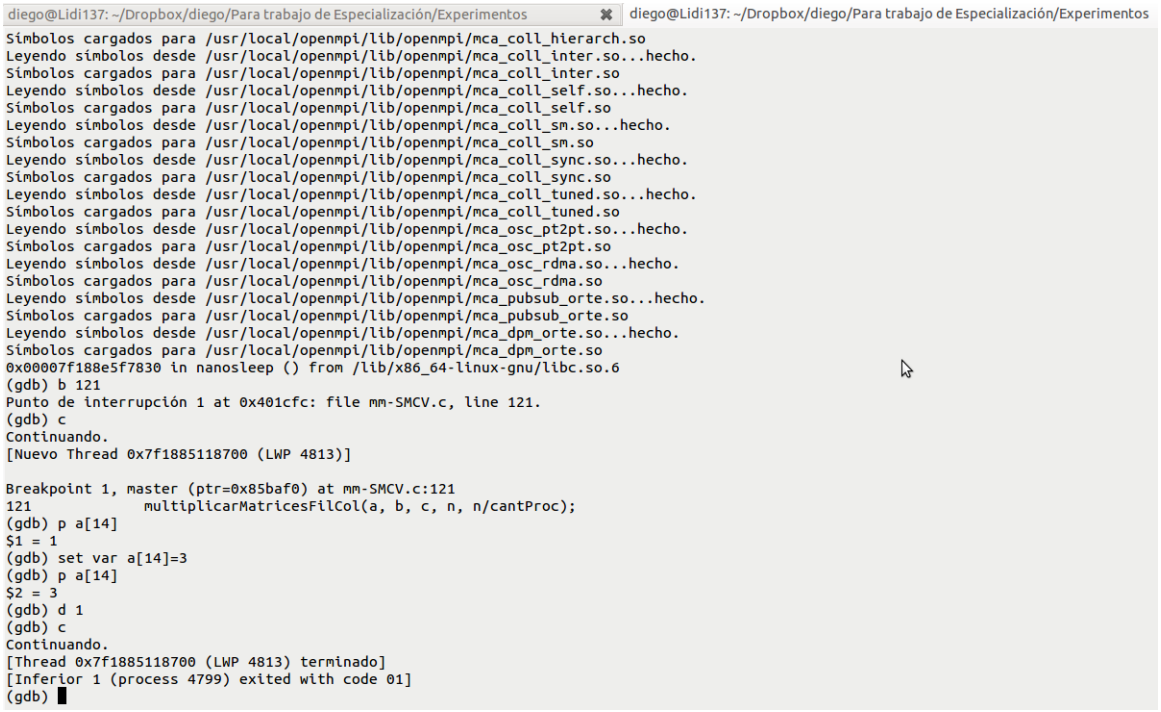

Fig. 6.7(a): ejemplo de inyección de un fallo que causa FSC

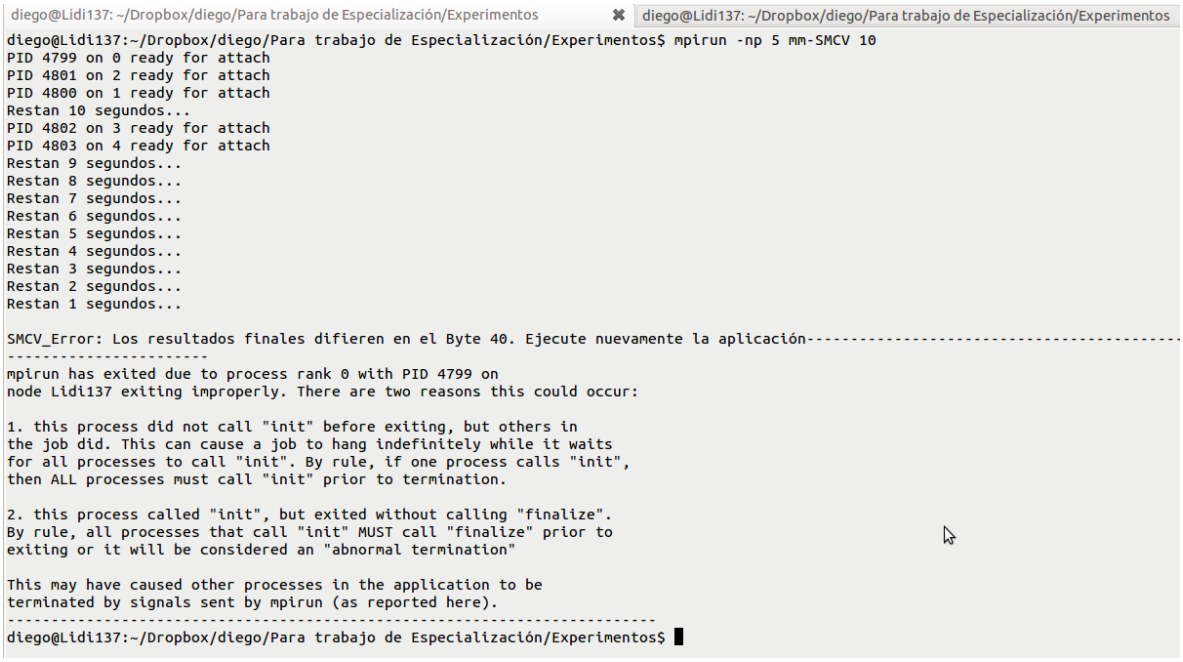

Fig. 6.7(b): salida de una ejecución en la que se inyectó un fallo que causa FSC

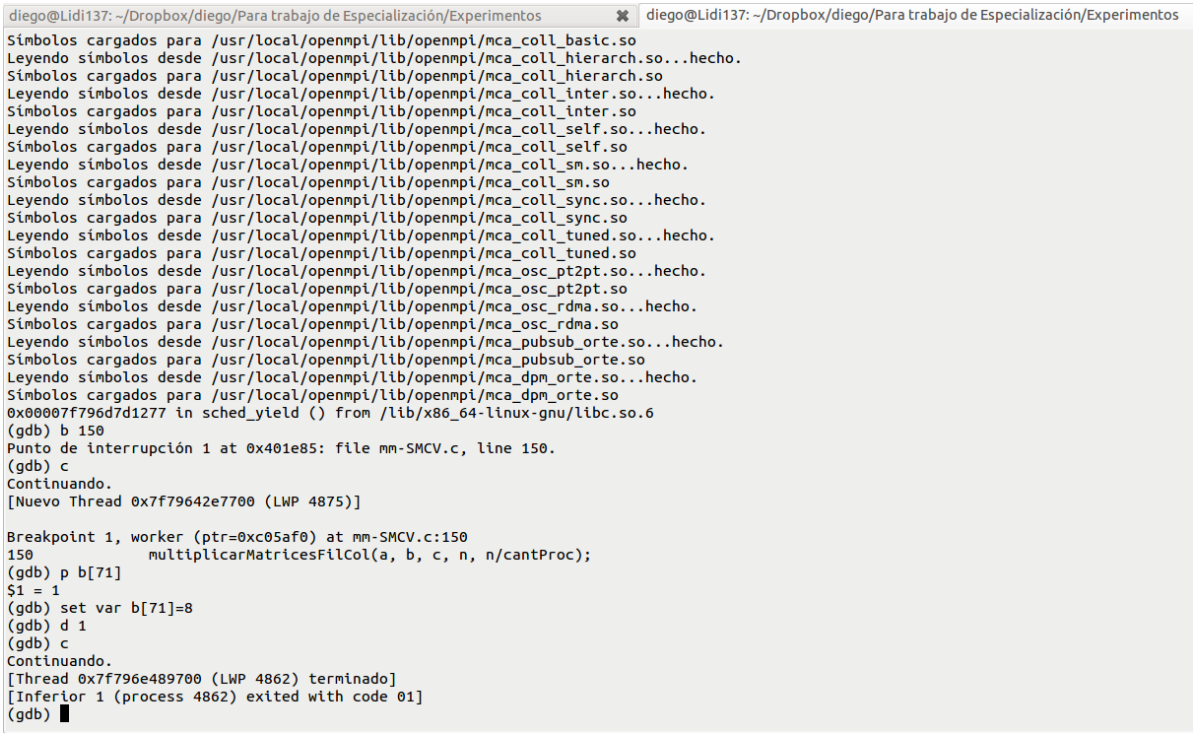

Fig. 6.8(a): ejemplo de inyección de un fallo que causa TDC

En la figura 6.9(b) se muestra la salida de la aplicación para el caso en el que se ha inyectado un fallo en un elemento cualquiera de la matriz C, antes de la multiplicación, en uno de los procesos *Workers*. En este caso, la multiplicación posterior sobreescribe el valor alterado, por lo que el fallo produce un error latente (LE). En consecuencia, como puede observarse en la figura, la salida de la aplicación es normal y correcta. La inyección se muestra en la figura 6.9(a). Debe aclararse que el fallo inyectado en este ejemplo es un fallo de memoria y no simula un fallo de registro,

porque el valor modificado no interviene en ningún cálculo. En este caso, el propósito de la inyección es causar un LE, el cual, justamente se caracteriza por afectar un dato que no es utilizado en el cómputo posterior.

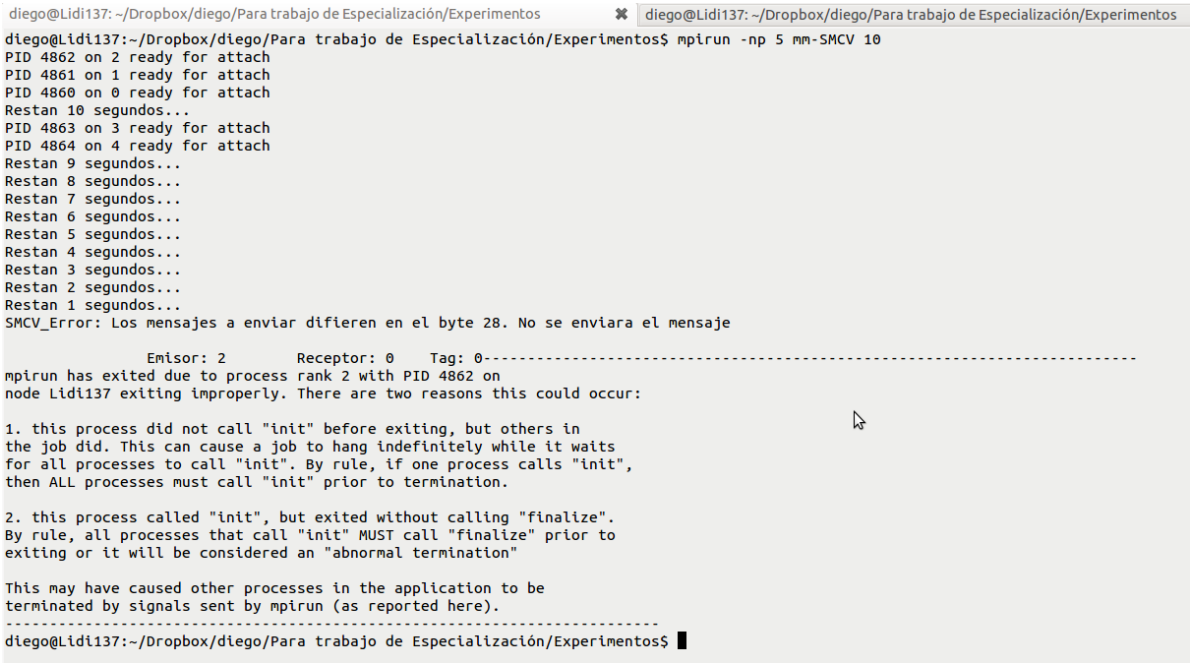

Fig. 6.8(b): salida de una ejecución en la que se inyectó un fallo que causa FSC

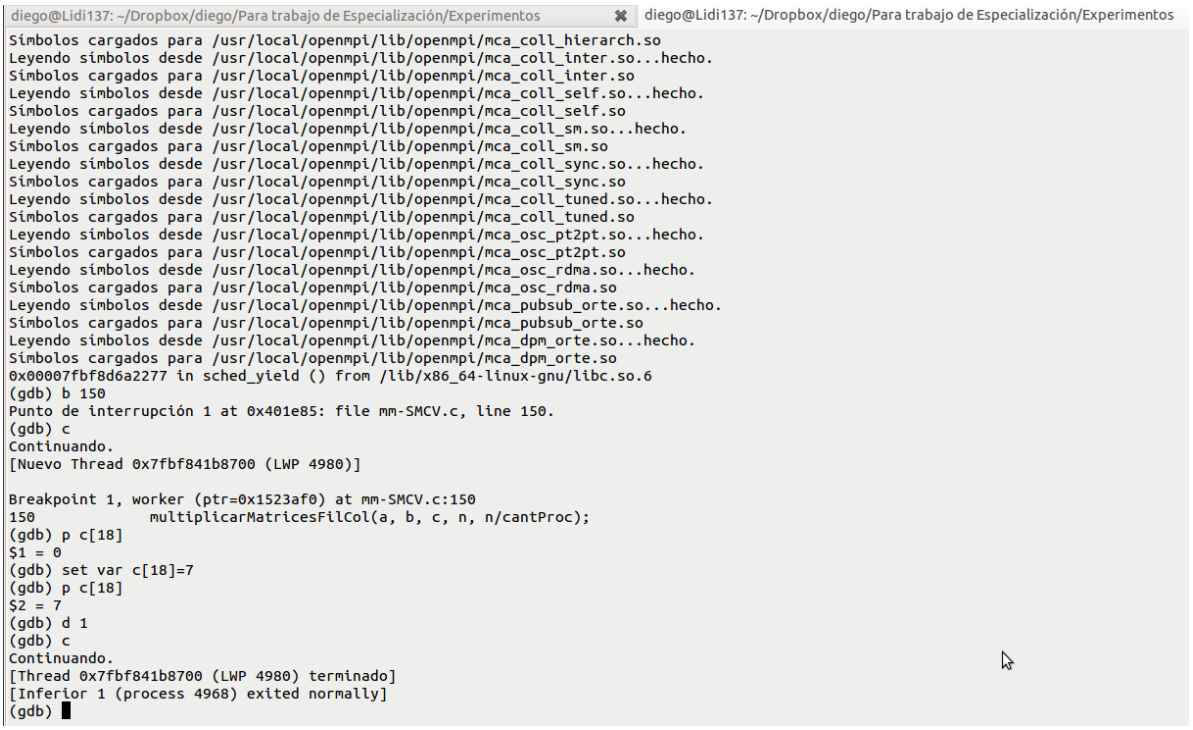

Fig. 6.9(a): ejemplo de inyección de un fallo que causa LE

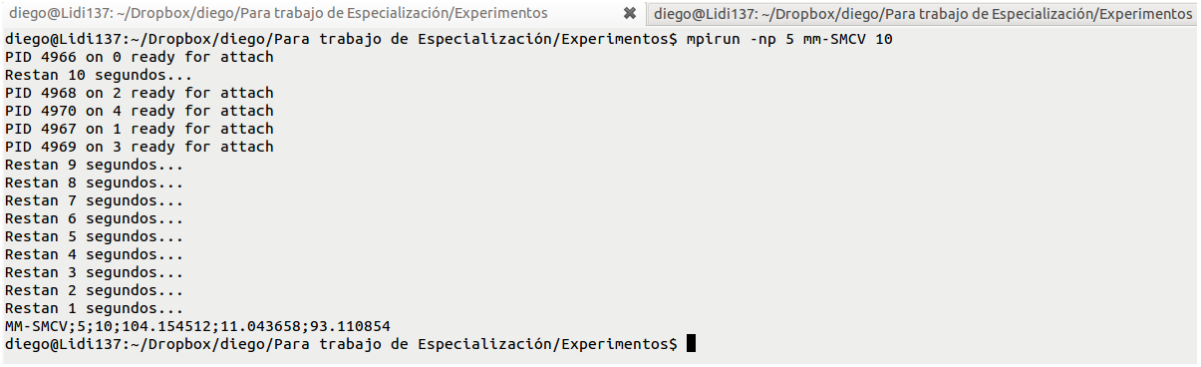

Fig. 6.9(b): salida de una ejecución en la que se inyectó un fallo que causa LE

Finalmente, en la figura 6.10 se puede ver la salida de la aplicación cuando ha ocurrido un fallo que produce *Time-Out* (TO). Nuevamente, se observa la detección del error y la parada segura. En este caso, el fallo se inyecta en una variable que actúa como índice, lo cual produce que, cuando una de las réplicas de un *Worker* ha realizado la mitad de su tarea, vuelva a comenzar el cómputo desde el principio. Esto ocasiona un retardo entre ambos hilos redundantes, que es detectado como un error por TO. El caso ideal de inyección de un fallo que produzca TO es aquel en que el proceso entra en un lazo infinito, pero en la aplicación de prueba seleccionada no es posible provocar este comportamiento mediante un fallo simple.

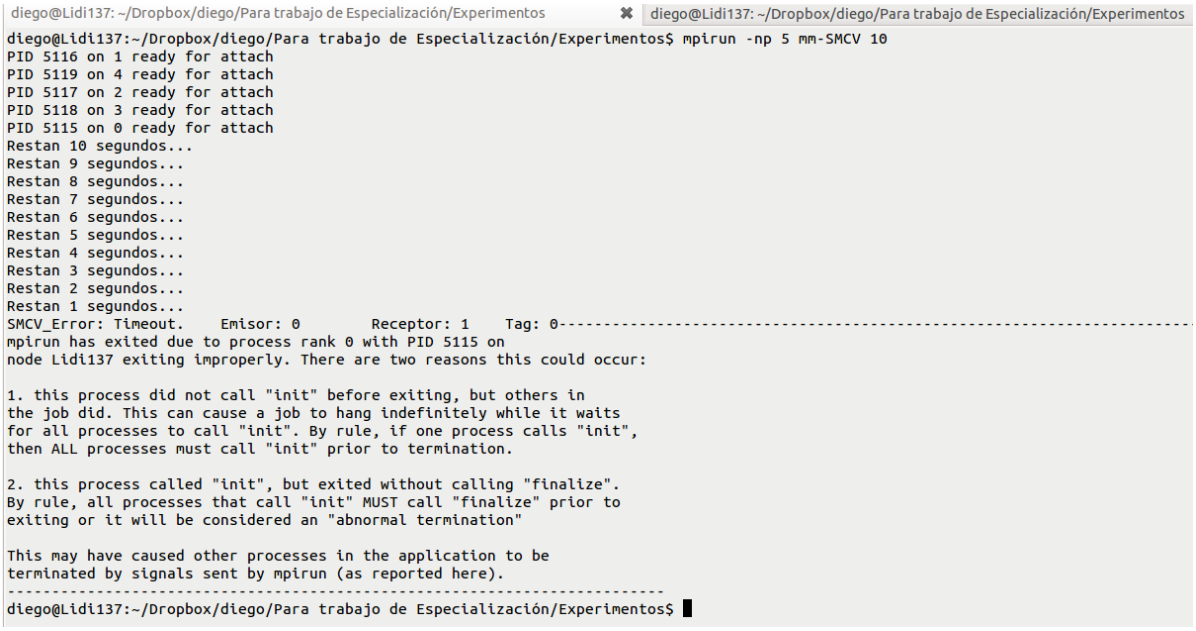

Fig. 6.10: salida de una ejecución en la que se inyectó un fallo que causa TO

Un aspecto importante a destacar respecto de la detección por *time-out* es que el lapso de tiempo pasado el cual se detecta un fallo es configurable. No existe un valor óptimo para esta configuración, sino que éste depende de la aplicación particular. Para clarificar esto, debe tomarse en cuenta que la idea de detección por *time-out* parte de la premisa de que, en una aplicación paralela que se ejecuta sobre un multicore dedicado (donde todos los cores son de las mismas características), los tiempos de ejecución de dos réplicas que realizan el mismo cómputo deberían ser similares. Por lo tanto, una asimetría notoria en los tiempos de procesamiento da lugar a pensar que ambas réplicas han separado sus

flujos de ejecución, posiblemente a causa de un fallo silencioso. Por lo tanto, la selección del lapso a configurar debe ser cuidadosa, tomando en cuenta lo que normalmente esperable de la aplicación. Si el valor de tiempo es muy alto, causará que transcurra mucho tiempo antes de la detección. En tanto, si es demasiado bajo, una pequeña asimetría en los tiempos de cómputo resultaría en la detección de un error. En el caso descripto de la multiplicación de matrices, el fallo inyectado que da lugar a la salida de la figura 6.10 en realidad no provoca que la aplicación concluya incorrectamente, sino sólo una demora anormal en la sincronización. En este caso, se configuró adrede un *time-out* breve para mostrar que el mecanismo es capaz de reaccionar frente a esta eventualidad. Sin embargo, si uno de los procesos ingresara en un bucle infinito, SMCV podría detectar esta situación, efectivamente, como un error.

A partir de las pruebas realizadas, se concluye que la incorporación de la estrategia SMCV es capaz de detectar los fallos que afectan contenidos de mensajes (TDC) de acuerdo a lo esperado, notificando al usuario y produciendo parada segura de la aplicación, de forma que la corrupción no pueda propagarse. De esta forma, todo el cómputo que realizan los *Workers* se encuentra protegido. Por otra parte, los fallos que afectan a datos que el *Master* mantiene para su cómputo local, y los que se producen luego de que el *Master* recoge los resultados parciales de todos los *Workers* en la última etapa de la ejecución (correspondientes a la fracción FSC) son detectados durante la comparación de resultados finales. En tanto, los fallos que producen asimetrías considerables en los tiempos de cómputo de las réplicas son detectadas por medio del mecanismo de *time-out*.

6.4.3 Mediciones de *overhead*

La métrica del *overhead* se analizó para evaluar la incidencia de la herramienta de detección SMCV sobre el rendimiento de las aplicaciones cuando escala el tamaño del problema y/o la arquitectura, en ausencia de fallos durante la ejecución. El *overhead* puede determinarse como el tiempo de ejecución agregado por la incorporación de la estrategia a la aplicación original, sobre la arquitectura de prueba. El tiempo extra adicionado por SMCV es consecuencia de la duplicación de cada proceso, la sincronización entre las réplicas, la comparación de contenidos realizada antes del envío de cada mensaje, la copia de los mensajes recibidos y la verificación de los resultados finales. Además, la duplicación de procesos incrementa la competencia por los recursos del sistema.

La métrica de *overhead* evaluada se calcula a partir de la ecuación 12, donde *APP_ET* es el tiempo de ejecución de la aplicación original y *SMCV_APP_ET* es el tiempo de ejecución de la aplicación con adaptada para funcionar con SMCV.

$$\boldsymbol{Overhead} = \frac{(SMCV\_APP\_ET \; - APP\_ET)}{APP\_ET} \times 100$$

Ecuación 12: *overhead* en tiempo de ejecución introducido por SMCV

Para estimar el impacto de la utilización de SMCV sobre la performance de una aplicación paralela científica de paso de mensajes, con el objetivo de evaluar la conveniencia de su utilización, se realizaron una serie de experimentos usando tres *benchmarks* paralelos. En las subsecciones siguientes se mencionan las aplicaciones de prueba y se describen los experimentos llevados a cabo y sus resultados.

6.4.3.1 *Benchmarks* utilizados

Los *benchmarks* paralelos seleccionados para los experimentos fueron tres: la multiplicación de matrices [25], la solución a la ecuación de Laplace [91] y el alineamiento de secuencias de ADN [92]. Estas aplicaciones de prueba fueron elegidas por tres motivos principales: en primer lugar, son bien conocidas y representativas de las aplicaciones científicas; en segundo lugar, son intensivas en cómputo; y por último, tienen tres patrones de comunicaciones diferentes: *Master-Worker*, SPMD (*Single-Program-Multiple-Data*) y *Pipeline* respectivamente.

6.4.3.2 Pruebas realizadas

Las pruebas fueron llevadas a cabo utilizando versiones MPI y SMCV de los tres *benchmarks* seleccionados. Se realizaron los pasos descriptos en la sección 6.3.2 para incorporar las características de SMCV al código fuente de las aplicaciones originales. Debido a que SMCV está diseñado para ser utilizado en el contexto de HPC, se compiló con el nivel −O de optimización.

Las pruebas con los *benchmarks* fueron realizadas utilizando diferentes cantidades de procesos: $P = \{4, 8, 16\}$, y para cada una de las aplicaciones se utilizaron distintos tamaños de problema: $N = \{2048, 4096, 8192, 16384\}$ para la multiplicación de matrices; $N = \{4096, 8192, 16384\}$ para la solución a la ecuación de Laplace y $N = \{65536, 131072, 262144, 524288\}$ para el alineamiento de secuencias de ADN. En cada nodo se mapearon como máximo cuatro procesos, lo que implica que en la ejecución de las aplicaciones originales sólo se pueden utilizar 4 cores del nodo. En el caso de las aplicaciones con la incorporación de SMCV, se utilizaron todos los cores del nodo (las réplicas se ejecutan en los cores disponibles). Cada experimento se ejecutó cinco veces y los resultados se promediaron para mejorar la estabilidad.

6.4.3.3 Resultados

Las figuras 6.11, 6.12 y 6.13 muestran los valores de *overhead* obtenidos con la integración de SMCV a las aplicaciones paralelas (multiplicación de matrices, solución a la ecuación de Laplace y alineamiento de secuencias de ADN, respectivamente), para varios tamaños de problema y utilizando diferentes cantidades de procesos.

Los gráficos muestran que los tres *benchmarks* presentan comportamientos similares. Se puede observar que el *overhead* disminuye cuando el tamaño del problema crece. Esto se debe a que, con tamaños de problema mayores, las aplicaciones pasan mayor cantidad de tiempo realizando cómputo que comunicando y, por lo tanto, el tiempo requerido para sincronizar réplicas y duplicar y validar contenidos de mensajes tiene un peso menor y queda oculto detrás del tiempo de procesamiento (en el caso de la multiplicación de matrices, la duplicación de datos produce *swapping* al disco para $N = 16384$ y $P = \{8, 16\}$, por lo que la tendencia de reducción del *overhead* no se mantiene). Con tamaños de problema más pequeños, el tiempo de cómputo es menor, por lo que las actividades de detección, relacionadas con las comunicaciones se vuelven más relevantes. Por otra parte, la cantidad de mensajes a enviar se incrementa cuando aumenta la cantidad de procesos. Esto produce un crecimiento del *overhead*, debido a que el tiempo requerido para sincronizar *threads* y para duplicar y comparar contenidos de mensajes se vuelve mayor.

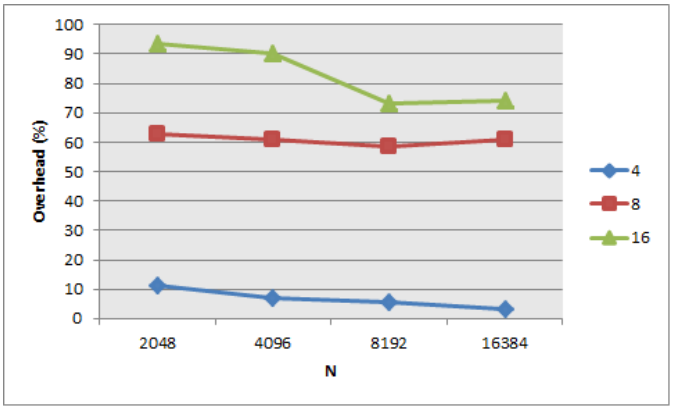

Fig. 6.11: *Overheads* obtenidos para la multiplicación de matrices con SMCV, para varios tamaños de problema con diferentes cantidades de procesos

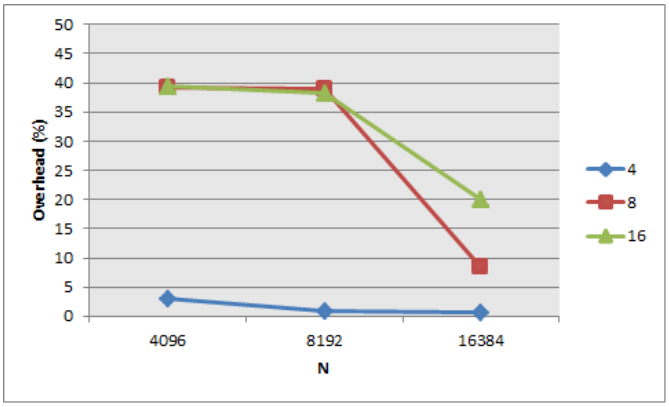

Fig. 6.12: *Overheads* obtenidos para la solución de la ecuación de Laplace con SMCV, para varios tamaños de problema con diferentes cantidades de procesos

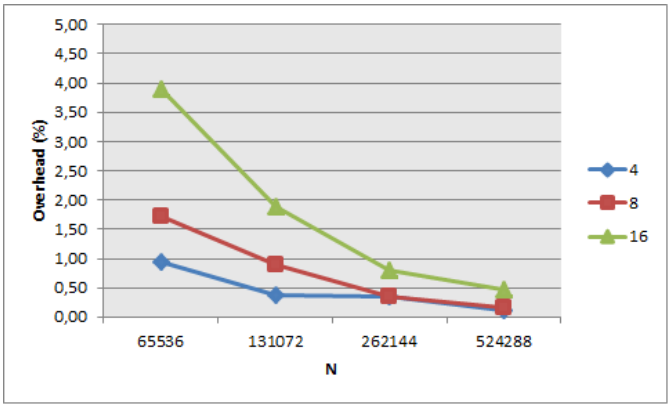

Fig. 6.13: *Overheads* obtenidos para el alineamiento de secuencias de ADN con SMCV, para varios tamaños de problema con diferentes cantidades de procesos

En la tabla 6.2 se muestra el detalle de los valores de *overhead* obtenidos en cada experimento, consistente en medir el tiempo de ejecución de cada aplicación de prueba con la incorporación de SMCV, para cada tamaño de problema listado y cada cantidad de procesos, y compararlo con el tiempo de ejecución de la aplicación original en la misma situación.

Como se mencionó, el comportamiento del *overhead* es similar para los tres casos, pero no ocurre lo mismo con los valores. La multiplicación de matrices es la aplicación que presenta mayores valores de *overhead*. Esto se debe a los tamaños de los mensajes que envían los procesos (el tamaño de las matrices va desde 16MB a 1GB dependiendo de *N*), agravado por el hecho de que se utilizan comunicaciones colectivas. A diferencia de OpenMPI, la biblioteca SMCV no optimiza este tipo de operaciones de comunicación [93]. Por último, el resultado final de esta aplicación es una matriz, por lo que el tiempo de validación requerido es significativo.

Los valores de *overhead* para la solución de la ecuación de Laplace son menores que los correspondientes a la multiplicación de matrices. Pese a que los procesos intercambian mensajes repetidamente (lo que aumenta la cantidad de operaciones de sincronización), el tiempo requerido para las verificaciones disminuye debido a que los mensajes son más pequeños (desde 16KB hasta 64KB dependiendo de *N*). Otro factor que influye es que el resultado final es un único número, por lo que el tiempo requerido para validarlo es despreciable.

El alineamiento de secuencias de ADN presenta valores de *overhead* aún menores que los de solución de la ecuación de Laplace. Todos los procesos envían y reciben mensajes repetidamente (excepto el primero y el último del *pipeline*). Debido a que estos mensajes son de tamaño fijo y muy pequeños (136B), el tiempo necesario para validarlos no es considerable. Como en el caso anterior, la verificación del resultado final demanda un tiempo insignificante.

Con el conjunto de experimentos realizados con varios tamaños de problemas y distintas cantidades de procesos, se verificó que la herramienta SMCV provee detección de fallos con un *overhead* máximo de 93.7 % y un promedio de 24.3 %. Esto implica una ventaja respecto de la ejecución original. Si en el sistema no se emplea ninguna estrategia de detección de fallos, se requieren al menos dos ejecuciones completas (y la comparación de los resultados finales de ambas ejecuciones) para garantizar que la salida es correcta y que no ha ocurrido un fallo que causa SDC. Más aún, si ocurre SDC, se requiere una tercera re-ejecución (y una nueva comparación) para recolectar los resultados que componen una mayoría como correctos. De esta manera se muestra que la herramienta SMCV es eficiente al incurrir en un *overhead* reducido para proveer detección de fallos transitorios.

Podría esperarse que las versiones de código que incorporan la detección de fallos ejecutaran a la mitad de velocidad que las originales, debido a que la cantidad de instrucciones está esencialmente duplicada. Sin embargo, la eficiencia en la planificación de la ejecución de las instrucciones y de la asignación de los recursos produce que el *overhead* sea menor que el esperado. Esto es congruente con resultados de experimentos, previamente publicados, sobre rendimiento de herramientas de fiabilidad basadas puramente en software, que muestran que la degradación de *performance* debida al código redundante es menor al doble [4,7].

Benchmark application	N	P			Promedio
		4	8	16	
Matrix multiplication	2048	11,39	62,94	93,74	57%
	4096	7,02	60,68	90,16	
	8192	5,54	58,42	73,10	
	16384	3,01	60,69	74,15	
Solution to Laplace's equation	4096	2,96	39,11	39,52	21%
	8192	0,82	38,89	38,16	
	16384	0,57	8,42	20,01	
DNA sequence alignment	65536	0,94	1,73	3,90	1%
	131072	0,37	0,90	1,88	
	262144	0,34	0,36	0,79	
	524288	0,12	0,16	0,46	
					Promedio = 24,3 %

Tabla 6.2: *Overheads* detallados para todos los experimentos

6.5 Resumen de las características de la metodología

A modo de sumario, se listan las principales características que proporciona la metodología SMCV:

- Cada proceso y su réplica son validados localmente, por lo que el mecanismo está totalmente distribuido entre todos los procesos de la aplicación. SMCV es estrategia descentralizada.
- Evita la propagación de errores entre los procesos de la aplicación. Además, detecta los errores en la fracción serie mediante la verificación de los resultados finales.
- Introduce un bajo *overhead* respecto del tiempo de ejecución, debido a que sólo se agrega una operación de comparación para cada byte de cada comunicación saliente y del resultado final (el costo de una operación de comparación es menor que el de una comunicación).
- Comparada con una estrategia de detección conservativa (diseñada para programas secuenciales), que verifica el valor de cada operación de escritura antes de realizarla para preservar la salida del programa [7], SMCV introduce una sobrecarga de operación reducida. En este sentido, puede afirmarse que es una técnica liviana.
- Cuando se detecta un fallo, se detiene la aplicación, permitiendo relanzar su ejecución. No es necesario esperar a que la aplicación finalice con resultado incorrecto para re-ejecutar, por lo que SMCV reduce la latencia del error. Esto, además de mejorar la fiabilidad, conlleva una ganancia en términos de tiempo, lo cual se vuelve particularmente significativo cuando se trata de aplicaciones científicas que pueden ejecutarse durante varios días.
- SMCV incrementa la fiabilidad del sistema, entendida como la cantidad de veces que la aplicación finaliza correctamente, debido a su capacidad de detectar todos los fallos que causan TDC.
- Logra un compromiso entre latencia de detección, la sobrecarga de operación y los recursos involucrados. SMCV permite una latencia en la detección, debido a que no se realiza ninguna verificación en el instante en que el valor corrompido se utiliza por primera vez. Esto pospone la detección hasta el momento en que el dato alterado forma parte del contenido de un mensaje que está a punto de enviarse. Sin embargo, esto conlleva una carga de trabajo menor que la de validar todas las operaciones de escritura (implica una baja latencia y un alta sobrecarga) y hace una mejor utilización de los recursos que una única verificación al final (es decir, duplicar todo el cómputo para detectar sólo al final, lo que implica una alta latencia y una baja sobrecarga). Cuanto más frecuente es la comunicación entre los procesos, la latencia es menor y la sobrecarga es mayor.

Capítulo 7

Conclusiones y trabajos futuros

Los fallos transitorios se están volviendo más frecuentes, especialmente en grandes sistemas de cómputo, y su impacto es cada vez más relevante en el caso de aplicaciones paralelas de larga duración. En este trabajo, se describe SMCV, que es una metodología de detección de fallos transitorios implementada únicamente en software y diseñada específicamente para aplicaciones paralelas científicas de paso de mensajes que se ejecutan en clusters de multicores. Bajo la premisa de que, en este tipo de aplicaciones, todos los datos que son relevantes para el resultado final son transmitidos entre los procesos que las componen, la estrategia SMCV se basa en validar los contenidos de los mensajes que se van a enviar y comparar los resultados finales, obteniendo un compromiso entre un alto nivel de cobertura frente a fallos y la introducción de un bajo *overhead* temporal, consecuencia de que no realiza trabajo para detectar fallos que normalmente no afectan a los resultados. Además, agrega una reducida sobrecarga de operación respecto de estrategias más conservativas que comprueban todas las escrituras a memoria, similares a las que se utilizan en aplicaciones secuenciales.

La implementación de la metodología de detección SMCV da lugar a una herramienta que busca ayudar a los programadores y a los usuarios de aplicaciones paralelas científicas a lograr ejecuciones fiables, obteniendo resultados finales correctos o, al menos, reportando la ocurrencia de fallos silenciosos dentro de un lapso acotado y conduciendo la ejecución a una parada segura.

Los experimentos realizados muestran la eficacia de SMCV para detectar los fallos transitorios que ocurren durante la ejecución de una aplicación paralela y que causan TDC, FSC y TO. En otra serie de pruebas, realizadas con tres *benchmarks* paralelos sobre un cluster de multicores, utilizando diferentes tamaños de problema y cantidades de procesos, SMCV mostró ser eficiente para lograr la detección de fallos introduciendo un *overhead* promedio de 24.3% y un *overhead* máximo de 93.7% en tiempo de ejecución.

Respecto de las líneas futuras, SMCV es una parte de una propuesta más extensa cuyo objetivo es proveer tolerancia a fallos transitorios para sistemas conformados por aplicaciones científicas paralelas de paso de mensajes que se ejecutan sobre arquitecturas de cluster de multicores.

La tolerancia a fallos incluye las fases de detección, protección y recuperación. En el contexto de los fallos permanentes, las técnicas existentes más utilizadas son las de *checkpointing* y *log* de eventos para la protección, y recuperación hacia atrás (*rollback-recovery*) [3]. La propuesta actual consiste en integrar la metodología de detección de fallos transitorios con las estrategias de protección y recuperación disponibles para fallos permanentes, con el propósito de proporcionar tolerancia completa a fallos transitorios. Esto implica que no hay necesidad de utilizar redundancia modular triple (TMR, [94]) con mecanismo de votación para detectar y recuperar un fallo transitorio. Además, como los fallos transitorios no requieren reconfiguración del sistema, la recuperación puede realizarse mediante re-ejecución en el mismo core que alojaba al proceso en cuyo contexto ocurrió el fallo.

En el camino hacia lograr este objetivo, quedan abiertas las siguientes líneas de trabajo:

1. Mejorar la estrategia de detección. Respecto de esto, el trabajo futuro debe enfocarse en los siguientes aspectos:

    a. Extender la implementación de la biblioteca SMCV para que proporcione soporte completo a las aplicaciones MPI. Actualmente, sólo provee operaciones de comunicación punto a punto bloqueantes y algunas comunicaciones colectivas.
    b. Optimizar la implementación de las comunicaciones colectivas para aprovechar las características de MPI, de modo de disminuir los valores de *overhead*.
    c. Automatizar el procedimiento para adaptar el código fuente de la aplicación paralela original para utilizar más fácilmente la herramienta SMCV.
    d. Emular funciones no-determinísticas, de forma de extender la metodología SMCV de forma de que pueda detectar fallos transitorios para aplicaciones científicas MPI no determinísticas.
    e. Optimizar la metodología en busca de mejorar la relación entre robustez frente a fallos, *overhead* introducido, sobrecarga de operación, latencia de detección y utilización de recursos. Una caracterización detallada permitirá proponer nuevas formas de mejora, considerando la posibilidad de configurar el nivel de cobertura tomando en cuenta las necesidades de fiabilidad de las aplicaciones y el máximo *overhead* permitido [42,44,48].

Se debe aclarar que, debido a que el procedimiento de duplicación que utiliza SMCV está basado en *threads*, se requieren algunos cambios menores en el código fuente de la aplicación (y la posterior recompilación), como se explica en la sección 6.3.2. Esto no es completamente transparente para la aplicación; lograr transparencia total requiere de implementar la replicación a nivel de procesos en lugar de a nivel de *threads*.

2. Proveer tolerancia completa a los fallos que provocan SDC y TO, restaurando el sistema a un estado consistente previo a la ocurrencia del fallo.

    En una etapa posterior, la estrategia distribuida de detección (ya optimizada) deberá ser integrada con una arquitectura de tolerancia a fallos orientada a fallos permanentes. Como se mencionó anteriormente, el objetivo es lograr un sistema capaz de tolerar tanto fallos permanentes como transitorios. en este sentido, se intentará la integración con RADIC [95]; RADIC es una arquitectura transparente, escalable, flexible y totalmente distribuida capaz de proporcionar tolerancia a fallos utilizando componentes no fiables y de recuperar al sistema de un fallo permanente de un nodo. El propósito de esto es tratar de aprovechar la metodología provista por RADIC para fallos permanentes (la recuperación por *rollback*, con el mecanismo de *checkpoints* no coordinados y el *log* de mensajes) y agregarle tolerancia a fallos transitorios. El sistema resultante deberá ser probado para determinar la confiabilidad obtenida, la transparencia para la aplicación, el *overhead* en ausencia de fallos y la degradación cuando opera con fallos.

Bibliografía

1. Montezanti, D., Frati, F.E., Rexachs, D., Luque, E., Naiouf, M.R., De Giusti, A.: SMCV: a Methodology for Detecting Transient Faults in Multicore Clusters. CLEI Electronic Journal 15(3), pp. 1-11 (2012)
2. Montezanti, D., Rucci, E., Rexachs, D., Luque, E., Naiouf, M.R., De Giusti, A.: A tool for detecting transient faults in execution of parallel scientific applications on multicore clusters. Journal of Computer Science & Technology (JCS&T), Vol. 14, No. 1, pp. 32 - 38 (2014)
3. Rexachs, D., Luque, E.: High Availiability for Parallel Computers. JCS&T Vol. 10 No. 3, pp. 110 – 116 (2010)
4. Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., August, D. I.: SWIFT: Software Implemented Fault Tolerance. In: Proceedings of the International Symposium on Code generation and optimization, pp. 243–254. IEEE Press, Washington DC (2005)
5. Gramacho, J., Rexachs del Rosario, D., Luque, E.: A Methodology to Calculate a Program ´s Robustness against Transient Faults. PDPTA 2011, 645 – 651 (2011)
6. Jalote, P. Fault Tolerance in Distributed Systems. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1994
7. Perry, F., Mackey, L., Reis G. A., Ligatti, J., August, D. I., Walker, D.: Fault-tolerant typed assembly language. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pp. 42-53. ACM Press, San Diego (2007)
8. Baumann, R. C.: Soft errors in commercial semiconductor technology: Overview and scaling trends. In: IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals, pp. 121 01.1 - 121 01.14
9. Michalak, S. E., Harris, K. W., Hengartner, N. W., Takala, B. E., Wender , S. A.: Predicting the number of fatal soft errors in Los Alamos National Labratory's ASC Q computer; IEEE Transactions on Device and Materials Reliability. 5(3), pp. 329 - 335 (2005)
10. Mukherjee, S. S.: Architecture Design for Soft Errors. Morgan Kaufmann (2008)
11. 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.
12. Li, A., & Hong, B. Software implemented transient fault detection in space computer. Aerospace science and technology, 11(2), pp. 245-252. (2007)
13. Shye, A., Blomstedt, J., Moseley, T., Reddi, V. J., Connors, D. A.: PLR: A software approach to transient fault tolerance for multicore architectures; IEEE Transactions on Dependable and Secure Computing. 6(2), pp. 135 - 148 (2009)
14. Mukherjee, S.; Weaver, C.; Emer, J.; Reinhardt, S., Austin, T.: A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 29 - 40. IEEE Press, San Diego (2003)
15. Yeh, Y.: Triple-triple redundant 777 primary flight computer. In Proceedings of the 1996 IEEE Aerospace Applications Conference, volume 1, pp. 293 - 307 (1996)
16. Yeh, Y.: Design considerations in Boeing 777 fly-by-wire computers. In Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium, pp. 64 - 72 (1998.)
17. Borkar, S.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. In IEEE Micro, volume 25, pp. 10 – 16 (2005)
18. Bronevetsky, G., Supinski, B.: Soft error vulnerability of iterative linear algebra methods. ICS '08: Proceedings of the 22nd annual international conference on Supercomputing, pp. 155-164. New York (2008)
19. Schroeder, B., Gibson, G. A.: A large-scale study of failures in high-performance computing systems. Dependable and Secure Computing, IEEE Transactions on, 7(4), pp. 337-350 (2010)
20. www.top500.org/statistics/list (2013)
21. Ibtesham, D., Arnold, D., Bridges, P. G., Ferreira, K. B., Brightwell, R.: On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance. In Parallel Processing (ICPP), 2012 41st International Conference on, pp. 148-157, IEEE (2012)
22. Lesiak, A., Gawkowski, P., Sosnowski, J.: Error Recovery Problems. Dependability of Computer Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on, 270 - 277 (2007)
23. Shivakumar, P., Kistler, M., Keckler, S. W., Burger, D., Alvisi, L.: Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks, 389 - 398 (2002)
24. Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference (Vol. 2). (1999)

25. Leibovich, F., Gallo, S., De Giusti, A., De Giusti, L., Chichizola, F., Naiouf, M.: Comparación de paradigmas de programación paralela en cluster de multicores: pasaje de mensajes e híbrido. In: Anales del XVII Congreso Argentino de Ciencias de la Computación. pp. 241 – 250, La Plata (2011)
26. Grama, A., Karypis, G., Kumar, V., Gupta, A.: An Introduction to Parallel Computing. Design and Analysis of Algorithms., 2nd ed. Addison Wesley (2003)
27. Wilkinson, B., Allen, M.: Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers, 2nd ed. Prentice Hall (2005)
28. Ben-Ari, M.: Principles of Concurrent and Distributed Programming. Second edition. Addison Wesley (2006)
29. Naiouf, M.:Procesamiento Paralelo. Balance de Carga Dinámico en Algoritmos de Sorting. La Plata: Tesis Doctoral. Universidad Nacional de La Plata. http://hdl.handle.net/10915/2264 (2004)
30. Rucci, E.: Computación eficiente del alineamiento de secuencias de ADN sobre cluster de multicores. Trabajo Final de Especialización en Cómputo de Altas Prestaciones. Universidad Nacional de La Plata. http://hdl.handle.net/10915/27737 (2013)
31. Oh, N., Shirvani, P. P., McCluskey, E. J.: Error detection by duplicated instructions in super-scalar processors; IEEE Transactions on Reliability. 51(1), pp. 63 - 75 (2002)
32. Reis, G. A., Chang, J., August, D. I.: Automatic instruction level software-only recovery methods. In IEEE Micro Top Picks, volume 27 (2007)
33. Mahmood, A., McCluskey, E. J.: Concurrent error detection using watchdog processors-a survey. IEEE Transactions on Computers, 37(2), pp. 160 - 174 (1988)
34. Gomaa M., Scarbrough C., Vijaykumar T. N., Pomeranz, I.: Transient-Fault Recovery for chip Multiprocessors. In: Proceedings of the 30th Annual International Symposium on Computer Architecture, pp. 98--109. IEEE Press, San Diego (2003)
35. Kontz M., Reinhardt S. K., Mukherjee S. S.: Detailed Design and Evaluation of Redundant Multithreading Alternatives. In: Proceedings of the 29th Annual International Symposium on Computer Architecture, pp. 99 - 110 (2002)
36. Venkatasubramanian, R., Hayes, J. P., Murray, B. T.: Low-cost on-line fault detection using control flow assertions. In Proceedings of the 9th IEEE International On-Line Testing Symposium, pp. 137 - 143 (2003)
37. Oh, N., Shirvani, P. P., McCluskey, E. J.: Control-flow checking by software signatures. IEEE Transactions on Reliability, 51(1), pp. 111 - 122 (2002)
38. Reinhardt, S. K., Mukherjee S. S.: Transient Fault Detection via Simultaneous Multithreading. In: Proceedings of the 27th annual International Symposium on Computer Architecture, pp. 25 - 36 (2000)
39. Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., August, D. I., Mukherjee, S. S.: Design and evaluation of hybrid fault-detection systems. In Proceedings of the 32th Annual International Symposium on Computer Architecture, pp. 148 – 159 (2005)
40. Batchu, R., Dandass, Y., Skjellum, A., Beddhu, M.: MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware; Cluster Computing. 7 (4), pp. 303 - 315 (2004)
41. Vijaykumar T. N., Pomeranz, I. Cheng, K.: Transient-Fault Recovery using Simultaneous Multithreading. Proceedings of the 29th Annual International Symposium on Computer Architecture, Anchorage, Alaska. Session 3: Safety and Reliability, pp. 87 - 98 (2002)
42. Golander A., Weiss S., Ronen R.: Synchronizing Redundant Cores in a Dynamic DMR Multicore Architecture. IEEE Transactions on Circuits and Systems II: Express Briefs Volume 56, Issue 6, pp. 474 - 478 (2009)
43. Barr A. H., Pomaranski K. G., Shidla D. J.: United States Patent Application Publication US 2005/0102565 A1: Fault Tolerant Multicore Microprocessing (2005)
44. Sundaramoorthy K., Purser Z., Rotenberg E.: Slipstream Processor: Improving both Performance and Fault-tolerance. ACM SIGPLAN Notices Volume 35, Issue 11, 257 - 268 (2000)
45. Rotenberg E.: AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. Proceedings of the 29th Annual International Symposium on Fault- Tolerant Computing, 84 – 91 (1999)
46. Saxena, N., McCluskey, E. J.: Dependable adaptive computing systems – the ROAR project. In International Conference on Systems, Man, and Cybernetics, pp. 2172 – 2177 (1998)
47. Ray, J., Hoe, J. C., Falsafi, B.: Dual use of superscalar datapath for transient-fault detection and recovery. In Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture, pp. 214 - 224 (2001)
48. Wells, P. M., Chacraborty K. Sohi G. S.: Mixed-Mode Multicore Reliability. ASPLOS 2009. SESSION: Reliable systems II, pp. 169 - 180 (2009)
49. Shirvani, P. P., Saxena, N., McCluskey, E. J.: Software implemented EDAC protection against SEUs. volume 49, pp. 273 - 284 (2000)
50. Oh, N., McCluskey, E. J.: Low energy error detection technique using procedure call duplication (2001)
51. Rebaudengo, M., Reorda, M. S., Violante, M., Torchiano, M.: A source-to-source compiler for generating dependable software, pp. 33 - 42 (2001)

52. Ohlsson, J., Rimen, M.: Implicit signature checking. In International Conference on Fault-Tolerant Computing, (1995)
53. Ferreira, K., Stearley, J., Laros III, J. H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R.,Bridges, P.G., Arnold, D.: Evaluating the viability of process replication reliability for exascale systems. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 44 (2011)
54. Bressoud, T.C.: TFT: A Software System for Application-Transparent Fault-Tolerance. Proc. Int'l Conf. Fault-Tolerant Computing (1998)
55. Schlichting, R.D., Schneider, F.B.: Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. ACM Trans. Computing Systems, vol. 1, no. 3, pp. 222 - 238 (1983)
56. Gomaa, M., Vijaykumar, T.N.: Opportunistic Transient-Fault Detection. Proc. 32nd Int'l Symp. Computer Architecture (2005)
57. Cloud Computing and Distributed Systems (CLOUDS) Laboratory. Department of Computer Science and Software Engineering. The University of Melbourne, Australia. Cluster and grid computing. Available: http://ww2.cs.mu.oz.au/678/ (2012)
58. Juhasz, Z., Kacsuk, P., Kranzlmuller, D. (Eds.): Distributed and Parallel Systems: Cluster and Grid Computing. New York: Springer-Verlag (2004)
59. Andrews, G. R.: Foundations of Multithreaded, Parallel, and Distributed Programming. EEUU: Addison Wesley Longman (2000)
60. Al-Jaroodi, J., Mohamed, N., Jiang, H., & Swanson, D.: Modeling Parallel Applications Performance on Heterogeneous System. Parallel and Distributed Processing Symposium, 2003. Proceedings. International (p. 7). Francia: IEEE Computer Society (2003)
61. Olukotun, K., Olukotun, O. A., Hammond, L., Laudon, J. P.: Chip Multiprocessors Architecture: Techniques to Improve Throughput and Latency , M. D. Hill, Ed. Morgan & Claypool (2007)
62. Rauber, T., Rünger, G.: Parallel Programming for Multicore and Cluster Systems. Springer-Verlag Berlin Heidelberg (2010)
63. Advanced Micro Devices, Inc.: AMD Multicore White Paper. Available: http://multicore. amd.com/es-ES/AMD-Multi-Core/resources/Technology-Evolution (2009)
64. McCool, M.: Scalable programming models for massively multicore processors, in Proceeding of the IEEE, vol. 96, no. 5, pp. 816 - 831 (2007)
65. Chai, L., Gago, Q., Panda, D. K.: Understanding the impact of multi-core architecture in cluster computing: A case study with Intel dual-core system. In Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (2007)
66. Zhang, C., Yuan, X., Srinivasan, A.: Processor Affinity and MPI Performance on SMP-CMP Clusters. In IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW) (2010)
67. Dongarra, J., Foster, I., Fox, G. C., Gropp, W., Kennedy, K., Torczon, L., White, A.: The Sourcebook of Parallel Computing. EEUU: Morgan Kaufmann Publishers (2003)
68. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. Cambridge: The MIT Press (1996)
69. http://www.mpi-forum.org (2014)
70. Message Passing Interface Forum, http://www.mpi-forum.org/ (2013)
71. http://www.open-mpi.org (2013)
72. MPICH: Available: www-unix.mcs.anl.gov/mpi/mpich2 (2012)
73. LAM/MPI: Available: http://www.lam-mpi.org/ (2012)
74. OpenMPI: Available: http://www.open-mpi.org/ (2012)
75. Fagg, G.E., Gabriel, E., Chen, Z., Angskun, T., Bosilca, G., Pjesivac-Grbovic, J., Dongarra, J.J.: Process Fault-Tolerance: Semantics, Design and Applications for High Performance Computing; International Journal of High Performance Applications. 19(4), pp. 465 - 478 (2005)
76. Ahn, J.: 2-step algorithm for enhancing effectiveness of sender-based message logging. In SpringSim '07: Proceedings of the 2007 spring simulation multiconference, pp. 429 - 434 (2007)
77. Elnozahy, E., Plank, J.: Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. Dependable and Secure Computing, IEEE Transactions on 1, 2, pp. 97 - 108 (2004)
78. Oldfield, R. A., Arunagiri, S., Teller, P. J., Seelam, S., Varela, M. R., Riesen, R., Roth, P. C.: Modeling the impact of checkpoints on next-generation systems. In 24th IEEE Conference on Mass Storage Systems and Technologies, pp. 30 - 46 (2007)
79. Huang, K. H., Abraham, J. A.: Algorithm-Based Fault Tolerance for Matrix Operations. IEEE Trans. on Computers, vol. 33, pp. 518 - 528 (1984)
80. Wang, S. J., Jha, N. K.: Algorithm-Based Fault Tolerance for FFT Networks. IEEE Trans. on Computers, vol. 43, no 7, pp. 849 - 854 (1994)

81. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems (TOCS) 20, 4, pp. 398 - 461 (2002)
82. Schroeder, B., Gibson, G. A.: Understanding failures in petascale computers. Journal of Physics: Conference Series 78, 1, 012022 (2007)
83. Chakravorty, S., Kalé, L. V.: A fault tolerant protocol for massively parallel systems. In Proceedings of the International Parallel and Distributed Processing Symposium, IEEE Computer Society Press (2004)
84. Jiang, Q., Manivannan, D.: An optimistic checkpointing and selective approach for consistent global checkpoint collection in distributed systems. In Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium (2007)
85. Moody, A., Bronevetsky, G., Mohror, K., Supinski, B. R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, IEEE Computer Society, pp. 1 - 11 (2010)
86. Engelmann, C., Ong, H. H., Scott, S. L.: The case for modular redundancy in large-scale high performance computing systems. In Proceedings of the 8th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN), ACTA Press, Calgary, AB, Canada, pp. 189 - 194 (2009)
87. Zheng, Z., Lan, Z.: Reliability-aware scalability models for high performance computing. In Cluster'09: Proceedings of the IEEE conference on Cluster Computing (2009)
88. Schneider, F.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys 22, 4, 299 - 319 (1990)
89. Brightwell, R., Ferreira, K. B., Riesen, R.: Transparent redundant computing with MPI. In EuroMPI (2010), R. Keller, E. Gabriel, M. M. Resch, and J. Dongarra, Eds., vol. 6305 of Lecture Notes in Computer Science, Springer, pp. 208 - 218 (2010)
90. Ferreira, K., Riesen, R., Oldfield, R., Stearley, J., III, J. H. L., Pedretti, K., Brightwell, R.: rMPI: Increasing fault resiliency in a message-passing environment. Technical Report . SAND2011-2488, Sandia National Laboratories (2011)
91. Andrews, G.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley Longman, EEUU (2000)
92. Rucci, E., Chichizola, F., Naiouf, M., De Giusti, A.: Parallel Pipelines for DNA Sequence Alignment on Cluster of Multicores. A comparison of communication models. Journal of Communication and Computer. 9(12), pp. 516 - 522 (2012)
93. Graham, R., Shipman, G.: MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. In: Proceedings of the 15th European PVM/MPIUsers' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. pp. 130 - 140 (2008)
94. Mathur, F., Avizienis, A.: Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair. AFIPS '70 (Spring) Proceedings of the May 5-7, 1970, Spring Joint Computer Conference (1970)
95. Santos, G., Duarte, A., Rexachs del Rosario, D., Luque, E.: Providing Non-stop Service for Message-Passing Based Parallel Applications with RADIC. Euro-Par 2008, pp. 58 – 67 (2008)