Paralelización especulativa y sus alternativas

Arturo González-Escribano, Diego R. Llanos
Dpto. de Informática
Universidad de Valladolid
47011 Valladolid, España
{arturo,diego}@infor.uva.es

Resumen

La paralelización especulativa es una técnica que permite extraer paralelismo de bucles no analizables en tiempo de compilación. Esta técnica se basa en ejecutar de forma optimista el bucle en paralelo, mientras un sistema hardware o software monitoriza la ejecución y corrige eventuales violaciones de dependencia. En este artículo se describe los fundamentos de la paralelización especulativa basada en software y se describen algunos de nuestros resultados más relevantes. Se discute también el alcance de esta técnica, describiendo los problemas que quedan aún por resolver, sus limitaciones y anticipando algunas alternativas a los mecanismos de paralelización automática.

1. Introducción

Gracias a las nuevas arquitecturas multiprocesador en un único chip (CMP), los sistemas paralelos son cada vez más comunes hoy en día, e incluso los ordenadores de sobremesa y portátiles ofrecen capacidad de cómputo en paralelo. Las nuevas arquitecturas hacen que el sistema funcione más rápido al posibilitar la ejecución de diferentes procesos al mismo tiempo. Sin embargo, los compiladores no siempre son capaces de extraer el paralelismo que puede estar oculto en un código secuencial, por lo que no es posible aprovechar de forma automática la capacidad de cómputo en paralelo de los nuevos sistemas. Para paliar este inconveniente, en las dos últimas décadas se han desarrollado numerosos lenguajes de programación paralela, extensiones paralelas para lenguajes secuenciales y bibliotecas de funciones. Sin embargo, la escritura de programas paralelos es aún una tarea difícil y propensa a errores. Para programar en paralelo es necesario un conocimiento profundo del problema que el algoritmo secuencial pretende resolver, el modelo de programación a utilizar y la arquitectura subyacente del sistema. Todo esto, unido a la gran cantidad de código secuencial desarrollado hasta la fecha, hace que sea muy atractiva la idea de poder paralelizar código de manera automática.

En este trabajo describimos la técnica de paralelización especulativa basada en software, mostrando algunos de nuestros resultados hasta la fecha, señalando posibles direcciones de investigación y analizando limitaciones de la técnica. El trabajo se estructura como sigue. La sección 2 describe el problema general de la paralelización automática. La sección 3 muestra el funcionamiento de los mecanismos de paralelización especulativa basados en software. La sección 4 describe nuestras aportaciones en este campo, así como algunas líneas de trabajo que están aún por explorar. En la sección 5 se discuten alternativas a los mecanismos de paralelización automática. Finalmente, la sección 6 resume nuestras conclusiones.

2. El problema de la paralelización automática

La paralelización automática engloba diferentes técnicas utilizadas en tiempo de compilación para transformar un algoritmo escrito en forma secuencial en una forma paralela, donde se exploten las capacidades inherentes del algoritmo paralelo original.

Para poder transformar un algoritmo escrito de forma secuencial en una forma paralela, el primer problema que tiene que resolver un compilador es determinar qué partes del código son susceptibles de poder ejecutarse en

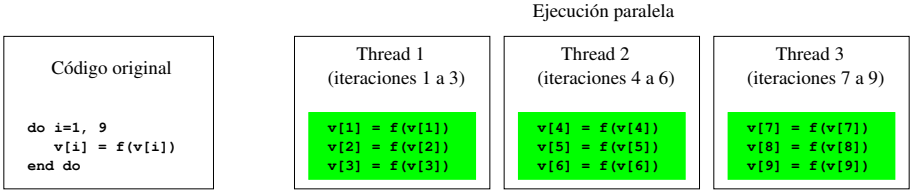

Figura 1: Ejecución paralela de un bucle sin dependencias utilizando tres threads. Como todas las iteraciones son independientes, el compilador puede ordenar su ejecución en paralelo.

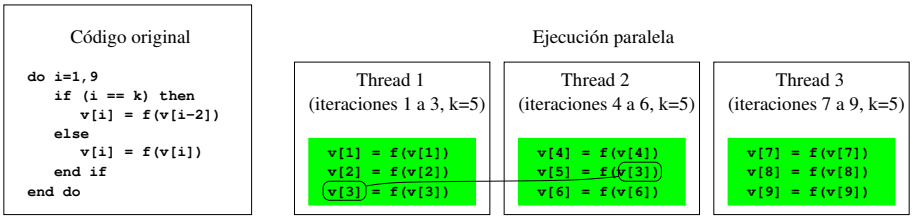

Figura 2: En este bucle, el valor `k` no se conoce en tiempo de compilación, lo que inhibe al compilador de paralelizar el bucle, ya que, si `k=5`, el thread 1 puede que no haya calculado el valor de `v[3]` a tiempo para que el thread 2 lo utilice. Esto no sería un problema si las dos iteraciones pertenecieran al mismo bucle: por ejemplo, con `k=6` el bucle podría paralelizarse.

paralelo. La mayor parte de los compiladores con capacidades de paralelización se centran en el paralelismo a nivel de bucle, estudiando cómo ejecutar diferentes iteraciones al mismo tiempo. Para que dos iteraciones puedan ejecutarse en paralelo es necesario que no estén relacionadas por ninguna *dependencia de datos*, es decir, que ninguna iteración utilice valores calculados por otra iteración previa, ya que podría utilizar valores antiguos que en la versión secuencial se calcularían antes de llegar a la iteración en curso. Como puede verse en la figura 1, si el compilador no detecta ninguna dependencia entre ellas, las dos iteraciones pueden ejecutarse en paralelo.

Lamentablemente, en muchos casos el compilador no es capaz de determinar si un par de iteraciones concretas son independientes entre sí. Por ejemplo, el código de la figura 2 presenta una dependencia entre dos iteraciones que depende del valor de la variable `k`. El problema es que el valor de esta variable puede que no se conozca en tiempo de compilación. En estos casos, el compilador no puede saber a ciencia cierta si la ejecución simultánea de las iteraciones es segura. Por lo tanto, el compilador evita paralelizar el bucle, pese a que, como puede verse en la figura 2, algunos de los posibles valores de `k` no producirían dependencias entre diferentes threads. Los problemas se agravan cuando el código secuencial realiza accesos indexados a matrices o cuando se utiliza aritmética de punteros.

Pese a estas dificultades, la paralelización automática sigue siendo un tema de gran actualidad, debido fundamentalmente a la gran cantidad de código secuencial existente y a su facilidad de desarrollo en comparación con el coste asociado a la programación paralela.

3. Paralelización especulativa

La técnica más prometedora para extraer el paralelismo de bucles no analizables en tiempo de compilación se denomina *paralelización especulativa* [4, 20, 21]. Esta técnica consiste en ejecutar el bucle directamente en paralelo, suponiendo (de manera optimista) que no se producirán violaciones de dependencia. Para ello, un mecanismo hardware o software divide el bucle en bloques de iteraciones, asignando cada una de ellas a un procesador, y ordenando y supervisando su ejecución. Si el sistema

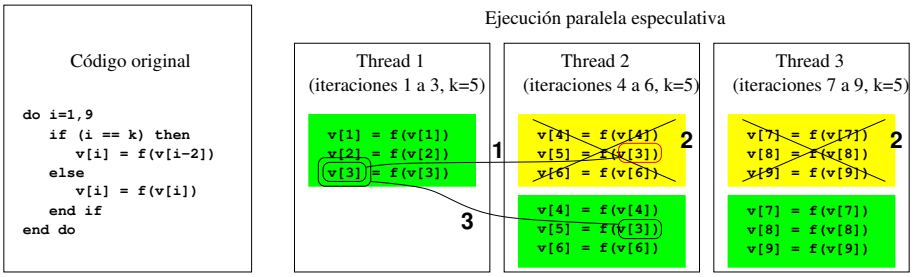

Figura 3: La paralelización especulativa inicia la ejecución del bucle en paralelo, mientras un sistema de control monitoriza la ejecución para detectar violaciones de dependencia entre distintos threads. Si se produce una violación, el sistema (1) detiene el thread que ha consumido el valor incorrecto y todos sus sucesores, (2) descarta los resultados generados por éstos, y (3) reinicia los threads de modo que consuman los valores correctos.

detecta una violación de dependencia deberá ocuparse de descartar las iteraciones calculadas incorrectamente y de reiniciar los cálculos en orden. Como puede suponerse, este proceso de parada y re-ejecución consume tiempo, por lo que la eficiencia de la paralelización especulativa mejorará cuando se produzcan menos dependencias en tiempo de ejecución entre las iteraciones del bucle.

Para comprender mejor el funcionamiento de la paralelización especulativa, es necesario distinguir entre variables privadas y compartidas. En este contexto, podemos decir de manera informal que una variable *privada* es una variable que siempre se modifica en el contexto de una iteración antes de utilizarse en esa misma iteración. Por el contrario, los valores que se almacenan en variables *compartidas* pueden utilizarse en diferentes iteraciones en cualquier momento de su ejecución. Si todas las variables presentes en un bucle son privadas, no pueden surgir violaciones de dependencia y el bucle puede ejecutarse en paralelo. Las variables compartidas no siempre provocan violaciones de dependencia. Esto sólo sucede cuando su valor se modifica en una iteración determinada y un sucesor ha consumido *con anterioridad* un valor antiguo de esa variable. La figura 2 muestra esta situación, denominada "dependencia RAW" *(Read-after-Write)*. En este caso, el sistema de control de la ejecución especulativa tiene que detener y volver a ejecutar el bloque de iteraciones que ha consumido el valor antiguo de la variable, de modo que se consuma el valor actualizado y se preserve de esta manera la semántica secuencial del programa. La figura 3 muestra ese proceso. Como puede verse, no sólo se descarta el bloque de iteraciones que ha consumido el dato incorrecto, sino también todos sus sucesores. Esto en la práctica puede que no sea estrictamente necesario, ya que bastaría con recordar qué threads han consumido datos provenientes del thread eliminado y eliminarlos a su vez, pero eliminarlos a todos simplifica la gestión.

Para simplificar el descarte de valores incorrectos, los threads que ejecutan cada bloque mantienen una copia local de los valores compartidos. Al final de cada ejecución exitosa de un thread, una operación de "consolidación" *(commit)* se encarga de actualizar las variables compartidas con los nuevos valores generados. Si el thread falla debido a una violación de dependencia, los valores calculados por éste simplemente se descartan.

El control de la ejecución especulativa puede encargarse a un sistema hardware o software. Los sistemas hardware [7, 14, 23] son más eficientes pero requieren cambios significativos en la jerarquía de memoria para poder detectar las violaciones de dependencia. Los sistemas software [4, 20, 21] no requieren modificaciones hardware, sino que le añaden a la aplicación original un conjunto de llamadas a funciones que se encargan de comprobar que no se produzcan violaciones de dependencia. Este enfoque permite utilizar sistemas convencionales de memoria compartida para la eje-

cución especulativa de aplicaciones, aunque se produce una cierta pérdida de rendimiento debido a la sobrecarga generada por la ejecución de estas instrucciones extras.

El añadido de llamadas a funciones encargadas de la ejecución especulativa puede hacerse de forma automática en tiempo de compilación. A continuación aparece una breve descripción de las funciones que deben añadirse.

- *Distribución de bloques de iteraciones.* Las iteraciones tienen que distribuirse entre los threads que participarán en la ejecución especulativa. Para ello pueden utilizarse varias estrategias: distribuir bloques de iteraciones de tamaño constante, adaptarlos a las características de la aplicación, o decidir dinámicamente el tamaño del siguiente bloque a lanzar.

- *Lecturas y escrituras especulativas.* Como cada thread mantiene copias locales de las variables compartidas, todas las lecturas y escrituras sobre éstas deben reemplazarse por funciones que, además de realizar la operación solicitada, comprueben que no se produzcan violaciones de dependencia.

- *Consolidación de resultados.* Tras la finalización exitosa de la ejecución de un bloque de iteraciones, deberá invocarse a una función que consolide los resultados producidos y solicite un nuevo bloque de iteraciones para su ejecución.

4. Nuestras aportaciones

Nuestro grupo de investigación lleva varios años trabajando en el estudio de los mecanismos de paralelización especulativa. En esta sección se describen algunos de los resultados más interesantes.

4.1. Desarrollo de un motor de ejecución especulativa

Hemos desarrollado un "motor de ejecución especulativa" basado en software, consistente en una librería de funciones que permiten ejecutar especulativamente un bucle [4, 5]. Se trata de un sistema experimental en el que, de momento, es el programador el encargado de editar el bucle e incorporar en el código original las llamadas a funciones correspondientes. Estas tareas podrían automatizarse en todos lo casos, siendo realizadas directamente por el compilador. En tiempo de ejecución se crean los threads necesarios para ejecutar el bucle, se lanzan los bloques de iteraciones y se ejecuta el bucle en paralelo. Al mismo tiempo, se monitoriza la aparición de violaciones de dependencia, deteniendo y rehaciendo los cálculos ejecutados incorrectamente.

Este motor de ejecución especulativa se ha utilizado con éxito en la paralelización especulativa de diferentes aplicaciones científicas. Estas aplicaciones, enumeradas en el cuadro 1, son `TREE` [1]; `MDG`, del PERFECT Club benchmark [2]; `WUPWISE`, del SPECfp2000 [24]; `2D-Hull`, que permite calcular el cierre convexo de una nube de puntos bidimensional, un problema de gran interés en el campo de la geometría computacional [6]; y `2D-SEC`, otro problema geométrico consistente en el cálculo del menor círculo contenedor de una nube de puntos en dos dimensiones [28]. Todas las aplicaciones presentan bucles no analizables en tiempo de compilación. En el caso de las tres primeras los bucles no presentan dependencias. `2D-Hull` es un algoritmo incremental aleatorizado cuyo número esperado de dependencias está relacionado con la forma del conjunto de datos de entrada, de ahí que hayamos considerado dos conjunto de entradas distintos, distribuyendo los puntos en un cuadrado o en un círculo; finalmente `2D-SEC` es una aplicación especialmente difícil de paralelizar, dado que presenta pocas dependencias pero el coste de resolverlas es comparable al coste del algoritmo en sí, por lo que casi no se obtienen aceleraciones.

La última fila del cuadro 1 resume los *speedups* máximos que obtienen los mecanismos de ejecución especulativa con estas aplicaciones. Como puede verse, los resultados de aceleración no son lineales. En general, la aceleración obtenida mediante las técnicas de ejecución especulativa son menores que si se paralelizara a mano la aplicación. La ventaja de la técnica es que no es necesario rediseñar o reprogramar el código secuencial ya existente. En [12] hemos estudiado en detalle el rendimiento de la pa-

Aplicación	TREE	MDG	WUPWISE	2D-Hull	2D-Hull	2D-SEC
Descripción del *input set*	Off-axis parabollic collision	Reference input set	Reference input set	Square-shaped, 10M points input set	Disc-shaped, 10M points input set	Disc-shaped, 10M points input set
Bucle paralelizado	accel_10	interf_1000'	muldeo_200', muldoe_200'	Bucle principal	Bucle principal	Bucle principal
Peso del bucle	94 %	86 %	41 %	99 %	99 %	99 %
Espacio para copias locales	$< 0{,}1$ Kb	13 Kb	12 000 Kb	13 Kb	86 Kb	$< 0{,}1$ Kb
Total de iteraciones	4 096	343	8 000	9 999 997	9 999 997	10 000 000
Porcentaje de violaciones de dependencia	0	0	0	2	15,48	$< 0,01$
Máximos *speedups*	10,03 (24 proc.)	5,11 (12 proc)	6,11 (12 proc.)	4,82 (28 proc.)	2,16 (16 proc.)	1,07 (4 proc.)

Cuadro 1: Características de los algoritmos utilizados como *benchmarks* del sistema de paralelización especulativa.

ralelización especulativa para el problema del cierre convexo en relación con el rendimiento de diferentes estrategias de paralelización manual.

4.2. Distribución de iteraciones entre procesadores

El funcionamiento de los mecanismos de ejecución especulativa basados en software plantean nuevos e interesantes desafíos. Uno de los más relevantes consiste en la determinación del tamaño óptimo del bloque de iteraciones enviado a cada procesador. Lanzar muchos bloques pequeños tiene el inconveniente de aumentar la sobrecarga del mecanismo de planificación *(scheduling)*, mientras que enviar bloques muy grandes juega en contra del balanceo de la carga que es deseable en todo sistema de cálculo paralelo. El problema se complica en el caso de la ejecución especulativa, porque la repentina aparición de una violación de dependencia puede dar lugar a un aumento inesperado de la carga de trabajo de un procesador concreto. El efecto se agrava cuando el tamaño de los bloques crece, ya que la cantidad de código a reejecutar es mayor en el caso de un fallo, y la probabilidad de encontrar una violación de dependencia en un bloque más grande en mayor.

Hemos desarrollado algunas técnicas para mitigar este problema. La más efectiva hasta la fecha es Meseta [17], un mecanismo que divide la ejecución especulativa en tres partes. Al principio se envían bloques de tamaño pequeño y creciente, ya que en las primeras iteraciones es más probable que aparezcan violaciones de dependencias. En la segunda etapa se estabiliza el tamaño, mientras que en la tercera, ya al final de la ejecución especulativa del bucle, se envían bloques de tamaño decreciente para balancear la carga. La forma de la función de tamaño de bloque respecto del tiempo (creciente-estable-decreciente) da nombre al mecanismo desarrollado.

4.3. Próximos objetivos

Aunque se han conseguido algunos resultados en la aplicación de la ejecución especulativa basada en software, esta técnica dista mucho aún de ser la respuesta definitiva a la creciente demanda de mecanismos de paralelización automática. A continuación enumeraremos brevemente algunas cuestiones para las que no se dispone aún de una solución satisfactoria y en las que continuamos trabajando.

- *Mecanismos de squash selectivo.* Al producirse una violación de dependencia, se descartan los cálculos realizados por el thread que ha consumido el dato incorrecto *y los de todos sus sucesores.* Esta solución es la más sencilla, ya que no requiere analizar qué sucesores han consumido datos incorrectos. Sin embargo, los estudios preliminares apuntan a que mantener la lista que relaciona a los productores y consumidores de los datos permitiría reducir el

número de threads afectados por una violación de dependencia, incidiendo positivamente en su rendimiento.

- *Manejo especulativo de memoria dinámica.* Todas las aplicaciones analizadas hasta ahora utilizan variables estáticas como variables compartidas. Se pretende extender los mecanismos de ejecución especulativa a estructuras dinámicas de datos.

- *Mecanismos avanzados de scheduling.* Sería deseable disponer de un mecanismo automático que determinara el tamaño de los bloques de iteraciones a enviar a cada procesador. El tamaño debe determinarse en función de parámetros como el número total de iteraciones y el número de violaciones de dependencia producidas hasta el momento, en lugar de enviar bloques de un tamaño predeterminado.

- *Estudio de la "aplicabilidad" de la ejecución especulativa.* El mecanismo desarrollado está especialmente diseñado para paralelizar bucles "grandes", que representen un porcentaje apreciable de la ejecución del programa. Queda por determinar: (a) cómo decidir de forma automática si es rentable utilizar este mecanismo, y (b) cuántas aplicaciones y de qué tipos podrían beneficiarse de la ejecución especulativa.

5. Alternativas a la paralelización automática

Como hemos visto, la paralelización automática basada en software puede obtener paralelismo en situaciones donde un análisis en tiempo de compilación no es posible. Sin embargo, las técnicas de paralelización automática tienen una limitación intrínseca: todas las técnicas utilizan exclusivamente la información proporcionada por el algoritmo secuencial, y en algunos casos por ejecuciones "de prueba" *(dry runs)*. Sin embargo, los lenguajes secuenciales tienen una semántica limitada que impide la representación de ciertas propiedades inherentes a las soluciones paralelas de un problema. La única forma de explotarlas es utiliza el conocimiento del programador para representar la estructura paralela del algoritmo original.

Para diseñar y expresar un algoritmo en paralelo se han propuesto múltiples mecanismos y lenguajes en las últimas décadas (ver por ejemplo [22, 15]). Lo ideal sería ofrecer al programador un lenguaje sencillo, que le permitiera expresar un diseño paralelo de una forma portable, sin preocuparse de los detalles de implementación.

Sin embargo, los mecanismos de programación paralela más populares hoy en día se centran en la sincronización y comunicación de procesos. En general esto obliga a un desarrollo *bottom-up*, en el que se mezclan detalles de diseño e implementación. La carencia de estados globales y la presencia de no-determinismo implican problemas de análisis complejos [16] y convierten al programa en un objeto de difícil análisis, complicando innecesariamente el proceso de depuración [8].

Una alternativa muy atractiva a estos mecanismos son las propuestas para programación paralela basadas en modelos de sincronización estructurados (ver clasificaciones en [10, 22]). Dentro de este campo, se encuentran los lenguajes basados en paralelismo-anidado o sincronización SP (Series-Parallel) [19, 26]. El modelo SP tiene una sólida base teórica, incluyendo una teoría de autómatas extendida [18], y una reducción inherente en la complejidad de muchos problemas combinatorios asociados al análisis de los programas [25]. En estos modelos se pueden utilizar técnicas de diseño de tipo *top-down*, basadas en refinamientos progresivos.La semántica del modelo implica la presencia de estados globales. El lenguaje puede diseñarse sin necesidad de constructores que impliquen no-determinismo. Las estructuras SP puras no permiten generar situaciones de deadlock u otros problemas similares asociados al uso de los mecanismos de sincronización de bajo nivel. Los programas se comportan de formas fácilmente analizables y se pueden utilizar técnicas de depuración clásicas basadas en trazas y puntos de ruptura.

En nuestro grupo se estudian lenguajes de alto nivel basados en paralelismo-anidado (estructuras SP), que permitan al programador

implementar sus diseños sin preocuparse de los detalles de implementación, como distribución de los procesos o datos, comunicaciones, etc. [11]. Para estos lenguajes es posible la generación automática del código de sincronización y comunicación necesario. Estas decisiones automáticas de implementación pueden estar guiadas por estimaciones de coste simbólicas [9] para sistematizar el balance de carga. En los casos más dinámicos existen sistemas en tiempo de ejecución muy eficientes, específicos para paralelismo anidado (ver por ejemplo [3]). Al igual que el código máquina generado a partir de un programa estructurado de alto nivel puede utilizar saltos incondicionales, los programas SP se pueden optimizar para aprovechar a bajo nivel la estructura no-SP más eficiente para una máquina concreta. Nuestra investigación en este campo se centra en la detección e implementación automática de estas estructuras de comunicación eficientes a partir de los códigos de alto nivel. El ámbito de aplicación incluye desde programas de supercomputación [13], hasta sistemas paralelos embebidos para electrónica de consumo [27]. Estos lenguajes presentan una interesante alternativa para la introducción de técnicas de ingeniería de software en la construcción y desarrollo productivo de software paralelo en todo tipo de entornos.

6. Conclusiones

La paralelización especulativa es una técnica de paralelización en tiempo de ejecución que permite extraer el paralelismo de códigos secuenciales no analizables en tiempo de compilación. Los resultados obtenidos hasta la fecha muestran que se trata de una técnica válida, aunque su alcance se ve reducido por varias cuestiones que permanecen sin resolver. Estas limitaciones, unidas al hecho de que la paralelización automática no puede aprovechar todo el conocimiento del programador sobre las soluciones inherentemente paralelas, hace que el diseñar e implementar programas directamente paralelos sea imprescindible para aprovechar al máximo las características de estos sistemas. De entre las herramientas de programación paralela destacan las basadas en modelos de sincronización estructurados, como los lenguajes de paralelismo-anidado. En estos modelos es posible ocultar al programador los detalles de implementación, aumentando el grado de determinismo del programa y permitiendo el uso de técnicas clásicas de depuración, lo que facilita en gran medida el desarrollo. Estas características hacen de los lenguajes de paralelismo anidado una opción muy interesante para el desarrollo de código paralelo.

Agradecimientos

Los autores de este trabajo agradecen a la Junta de Castilla y León su apoyo a través del Proyecto de Investigación VA031B06 y a la Comisión Europea a través del contrato RII3-CT-2003-506079.

Referencias

[1] Barnes, J. E. Institute for Astronomy, University of Hawaii, ftp://ftp.ifa.hawaii.edu/pub/barnes/treecode/.

[2] Berry, M. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications 3*, 3 (1989), 5–40.

[3] Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., and Zhou, Y. Cilk: An efficient multithreaded runtime system. In *Proc. of 5th PPoPP* (1995), ACM, pp. 207–216.

[4] Cintra, M., and Llanos, D. R. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (June 2003).

[5] Cintra, M., and Llanos, D. R. Design space exploration of a software speculative parallelization scheme. *IEEE Trans. on Paral. and Distr. Systems 16*, 6 (June 2005), 562–576.

[6] Cintra, M., Llanos, D. R., and Palop, B. Speculative parallelization of a randomized incremental convex hull algorithm. In *Proc. of the Comput. Geom. and Applications (CGA), LNCS 3045* (May 2004), pp. 188–197.

[7] Cintra, M., Martínez, J. F., and Torrellas, J. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proc. of the 27th intl. symp. on Computer architecture (ISCA)* (June 2000), pp. 256–264.

[8] Gatlin, K. Trials and tribulations of debugging concurrency. *ACM Queue 2*, 7 (Oct 2004), 67–73.

[9] Gemund, A. v. Symbolic performance modeling of parallel systems. *IEEE Transactions on Parallel and Distributed Systems 14*, 2 (Feb 2003), 154–165.

[10] González-Escribano, A. *Synchronization Architecture in Parallel Programming Models*. Phd thesis, Dpto. Informática, University of Valladolid, Jul 2003.

[11] González-Escribano, A., Gemund, A. v., Cardeñoso-Payo, V., Portales-Fernández, R., and Caminero-Granja, J. A preliminary nested-parallel framework to efficiently implement scientific applications. In *VECPAR 2004* (Apr 2005), M. D. et al., Ed., no. 3402 in LNCS, Springer-Verlag, pp. 541–555.

[12] González-Escribano, A., Llanos, D. R., Orden, D., and Palop, B. Parallelization alternatives and their performance for the convex hull problem. *Applied Mathematical Modelling, special issue on Parallel and Vector Processing in Science and Engineering 30*, 7 (July 2006), 563–577. ISSN 0307-904X.

[13] González-Escribano, A., van Gemund, A., and Cardeñoso-Payo, V. SPC-XML: A structured representation for nested-parallel programming languages. In *Euro-Par 2005, Parallel Processing* (2005), P. M. J.C. Cunha, Ed., vol. 3648 of *LNCS*, ACM, IEEE, Springer-Verlag, pp. 782–792.

[14] Hammond, L., Willey, M., and Olukotun, K. Data speculation support for a chip multiprocessor. In *Proc. of the 8th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1998), pp. 58–69.

[15] Hasselbring, W. Programming languages and systems for prototyping concurrent applications. *ACM Computing Surveys 32*, 11 (2000), 43–79.

[16] Lee, E. The problem with threads. *Computer 39*, 5 (May 2006), 33–41.

[17] Llanos, D. R., Orden, D., and Palop, B. New scheduling strategies for randomized incremental algorithms in the context of speculative parallelization. *IEEE Transactions on Computers 56*, 6 (2007), 839–852.

[18] Lodaya, K., and Weil, P. Series-parallel posets: Algebra, automata, and languages. In *Proc. STACS'98* (Paris, France, 1998), vol. 1373 of *LNCS*, Springer-Verlag, pp. 555–565.

[19] Lodaya, K., and Weil, P. Series-parallel languages and the bounded-width property. *Theoretical Comp. Science 237* (2000), 347–380.

[20] M. Gupta and R. Nim. Techniques for run-time parallelization of loops. *Supercomputing* (November 1998).

[21] Rauchwerger, L., and Padua, D. A. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems 10*, 2 (1999), 160–180.

[22] Skillicorn, D., and Talia, D. Models and languages for parallel computation. *ACM Computing Surveys 30*, 2 (Jun 1998), 123–169.

[23] Sohi, G., Breach, S., and Vijaykumar, T. Multiscalar processors. In *Proc. of the 22nd Intl. Symp. on Computer Architecture (ISCA)* (June 1995), pp. 414–425.

[24] Standard Performance Evaluation Corporation. http://www.spec.org/.

[25] Takamizawa, K., Nishizeki, T., and Saito, N. Linear-time computability of combinatorial problems on series-parallel graphs. *Journal of the ACM 29*, 3 (1982), 623–641.

[26] Valdés, J., Tarjan, R., and Lawler, E. The recognition of series parallel digraphs. *SIAM Journal of Computing 11*, 2 (May 1982), 298–313.

[27] Varbanescu, A., Nijhuis, M., González-Escribano, A., Sips, H., Bos, H., and Bal, H. SP@CE - An SP-based programming model for consumer electronics streaming applications. In *LCPC'06* (Nov 2006).

[28] Welzl, E. Smallest enclosing disks (balls and ellipsoids). In *New Results and new Trends in Computer Science*, H. Maurer, Ed., no. 555 in Lecture Notes in Computer Science. Springer, 1991, pp. 359–370.