Universidad de Valladolid

Escuela Técnica Superior de Ingeniería Informática

Departamento de Informática

Tesis Doctoral:

Design and evaluation of a Thread-Level Speculation runtime library

Presentada por **D. Álvaro Estébanez López** para optar al grado de doctor por la Universidad de Valladolid

Dirigida por:

Dr. Diego R. Llanos Ferraris

Valladolid, Octubre 2015

Resumen

En los próximos años es más que probable que máquinas con cientos o, incluso, miles de procesadores sean algo habitual. Para aprovechar estas máquinas, y debido a la dificultad de programar de forma paralela, sería deseable disponer de sistemas de compilación o ejecución que extraigan todo el paralelismo posible de las aplicaciones existentes. Para ello, durante los últimos tiempos, se han propuesto multitud de técnicas paralelas. Sin embargo, la mayoría de ellas se centran en códigos simples, es decir, sin dependencias entre sus instrucciones. La paralelización especulativa surge como una solución para estos códigos complejos, posibilitando la ejecución de cualquier tipo de códigos, con o sin dependencias. Esta técnica asume de forma optimista que la ejecución paralela de cualquier tipo de código no de lugar a errores y, por lo tanto, necesitan de un mecanismo que detecte cualquier tipo de colisión. Para ello, constan de un monitor responsable que comprueba constantemente que la ejecución no sea errónea, asegurando que los resultados obtenidos de forma paralela sean similares a los de cualquier ejecución secuencial. En caso de que la ejecución fuese errónea los *threads* se detendrían y reiniciarían su ejecución para asegurar que la ejecución sigue la semántica secuencial.

Nuestra contribución en este campo incluye (1) una librería de ejecución especulativa nueva y fácil de utilizar; (2) nuevas propuestas que permiten reducir de forma significativa el número de accesos requeridos en las operaciones especulativas, así como consejos para reducir la memoria a utilizar; (3) propuestas para mejorar los métodos de *scheduling* centradas en la gestión dinámica de los bloques de iteraciones utilizados en las ejecuciones especulativas; (4) una solución híbrida que utiliza memoria transaccional para implementar las secciones críticas de una librería de paralelización especulativa; y (5) un análisis de las técnicas especulativas en uno de los dispositivos más vanguardistass del momento, los coprocesadores Intel Xeon Phi.

Como hemos podido comprobar, la paralelización especulativa es un campo de investigación activo. Nuestros resultados demuestran que esta técnica permite obtener mejoras de rendimiento en un gran número de aplicaciones. Así, esperamos que este trabajo contribuya al acercamineto del uso de soluciones especulativas a los compiladores comerciales y/o los modelos de programación paralela de memoria compartida.

Palabras clave

Paralelización especulativa, Especulación a nivel de Thread, Paralelización optimista, Scheduling, Rendimiento, Evaluación.

Abstract

It is very likely that, in the next years, shared-memory systems with hundreds or even thousands of computational units will become commonplace. Since parallel programming is conceptually difficult, and to take advantage of these platforms, it is desirable to have compiling and/or runtime systems that automatically extract all the available parallelism of a sequential application. Although many parallel processing approaches have been developed in the last decades, most automatic parallelization proposals are focused on codes with no hurdles. Speculative parallelization (SP) techniques arise as a more general solution, allowing the parallel execution of any code, even in the presence of dependence violations. To ensure that, SP approaches rely on a runtime monitor responsible for ensuring that the results of the parallel execution match the expected output of the original, sequential code. This technique, based on the optimistic assumption that no dependences will arise when executing the code in parallel, launches threads that execute different fragments of the sequential code at the same time. If a dependence violation is detected, the offending threads are stopped and restarted with the correct values, thus ensuring that the execution follows sequential semantics.

Our contribution in this field includes (1) a new, easy-to-use speculative runtime library; (2) new proposals which allow to decrease the number of memory accesses involved in speculative operations, as well as some advice to decrement the memory footprint; (3) research on new scheduling methods focused on the dynamic management of chunks of iterations in speculative executions; (4) an hybrid approach which implements speculative parallelism using transactional memory to handle its critical sections; and (5) an analysis of the speculative techniques in one of the most state-of-the-art devices as are Intel Xeon Phi coprocessors.

Speculative parallelization is a lively research field. Our results show that these techniques have the potential of leading to considerable improvements in the performance of many applications. We expect that this work, among others, will foster the use of SP-based solutions in commercial compilers and shared-memory parallel programming models.

Keywords

Speculative parallelization, Thread-Level Speculation, Optimistic parallelism, Scheduling, Performance, Evaluation.

A mi madre

Agradecimientos

Para llevar a cabo una tesis doctoral, hace falta mucho más que matricularse. Durante su desarrollo hay muchos momentos buenos, pero también hay otros en que se hace muy duro seguir adelante. Con estas líneas quiero agradecer a todos los que celebraron conmigo los buenos ratos, pero sobretodo a los que me ayudaron a levantarme en cada caída y me animaron cuando más lo necesitaba.

Primero, quiero acordarme de mis compañeros de fatigas, Yuri, Sergio, Javier, Héctor y Ana, por esas charlas sobre nuestro futuro, por el apoyo, las risas y sobretodo el buen ambiente creado en el grupo. También mencionar a Arturo que ha sido un excelente tutor y una gran persona.

También quiero agradecer a mi familia y amigos por estar ahí, especialmente a mi hermano Juan Cruz, y mi amigo Pablo con los que he compartido tantos, y tan buenos momentos. También quiero destacar a mis abuelos, cuya pérdida ha sido el momento más agrio de mi vida. Sólo puedo agradecerles el haber estado conmigo siempre, ayudando a mi madre en todo lo posible. Sin ellos no se si hubiera llegado hasta aquí, y desde luego les recordaré siempre.

Pero para ser justos hay tres personas que merecen unos agradecimientos mucho más concretos.

Diego R. Llanos Ferraris, quiero agradecerle que haya confiado en mí desde el principio. Ese principio que data de Febrero de 2011 cuando llamé a la puerta de su despacho para empezar un proyecto de fin de carrera que me permitió obtener la Ingeniería Técnica. Cuando después, apostó por mí de nuevo al asignarme un trabajo de fin de grado con un tema tan importante para él, y que fue el germen de esta tesis doctoral, el desarrollo de la librería especulativa. También por darme todo su apoyo durante el siempre complicado máster. Con respecto al final del camino, esta tesis, decir que ha sido un placer y un orgullo trabajar bajo su tutela, siendo comprensivo, exigente, y enseñadome perfectamente cómo ser un buen investigador. Si me permites, te animo a que sigas investigando y formando doctores puesto que la ciencia y la Universidad de Valladolid lo agradecerán. Además sería injusto no recordar todo el esfuerzo económico que ha hecho por mí. Y es que, junto con Arturo, han intentado que realice la tesis con todos sus medios, me habéis apoyado cuando me quedaba a las puertas de una beca, e incluso cuando me equivocaba cumplimentando los papeles. Deciros que siempre he intentado reconocer vuestros esfuerzos, que me han motivado, y me han llevado a finalizar esta tesis doctoral, nunca lo olvidaré. Gracias a Diego me voy de la Universidad

con una tesis doctoral, unas cuantas publicaciones, y lo más importante un amigo para toda la vida.

Pilar López Alonso, a ella le quiero dar la gracias por todo lo que ha hecho por mí. Y es que aparte de darme la vida, me ha enseñado todos los valores importantes de la misma. A través de esfuerzo y constancia ha conseguido sacar a sus hijos adelante, unos hijos que le debemos todo lo que somos. Quiero darle las gracias por todo su apoyo incondicional en los buenos y malos momentos. El mundo sería perfecto si lo formase más gente como tú, espero que algún día pueda merecerme que alguien esté tan orgulloso de mí, como lo estoy yo de tí.

Tania Alonso Sambade, porque ha sido mucho más que una novia para mí. Desde que comencé mi andadura por la universidad, ella ha sido mi novia, mi mejor amiga y se podría decir que mi psicóloga. Me has apoyado, escuchado, aconsejado en los momentos difíciles, y celebrado conmigo las buenas noticias. Gracias por ser como eres y estar conmigo siempre, esta tesis no hubiera sido posible si tú no hubieras estado a mi lado. Las cosas pueden cambiar, pero tu y yo seguimos siempre juntos.

Álvaro Estébanez López

Contents

List of Figures

List of Tables

List of Listings

Resumen de la tesis

Desde la aparición de los circuitos integrados, el rendimiento de los ordenadores ha crecido exponencialmente. Sin embargo, esta progresión terminó por toparse con ciertos límites impuestos por leyes físicas, tales como la disipación de calor, que obligaron a repensar el modelo de avance. Así, en lugar de seguir aumentando la frecuencia de los procesadores cada vez más, se optó por utilizar varias unidades computacionales (actuando como una sola) en el mismo chip, comenzando la era de los sistemas multiprocesador de memoria compartida. Sin embargo, para sacar rendimiento de una misma aplicación, el código secuencial debe ser o bien rescrito como un conjunto de tareas paralelas, o transformado automáticamente en código paralelo.

En este capítulo resumiremos la motivación de esta tesis doctoral, la metodología seguida, y los principales objetivos conseguidos. Además se enumerarán las conclusiones obtenidas.

R.1 Motivación

Desde la aparición del primer microchip en 1971, el 4004 de Intel, con unos 3200 procesadores y una frecuencia de 104KHz [16, 122], los ordenadores han experimentado una evolución constante y vertiginosa. Así, el número de transistores por dispositivo ha crecido exponencialmente, con el consiguiente aumento en el rendimiento de los mismos. Sin embargo, a comienzos del siglo XXI, esta tasa de mejora se vió afectada por límites físicos, tales como la imposibilidad de extraer todo el calor producido por un chip de unos cuantos milímetros con un consumo superior a 100W.

Por estos motivos, los ingenieros tuvieron que repensar el modelo de progreso, comenzando a integrar varios procesadores en el mismo dispositivo. Esta solución se beneficia del incremento de transistores que pueden empaquetarse juntos y de la distribución del calor en puntos diferentes del chip. Por otro lado, también se logró disminuir el pico de frecuencia utilizado, por lo que los dispositivos que requieren un menor consumo de energía (como los que utilizan baterías) pudieron beneficiarse de este nuevo enfoque.

Sin embargo, por primera vez en la historia, un cambio de arquitectura no produjo automáticamente un aumento en el rendimiento de las aplicaciones subyacentes. Para aprovechar estas nuevas mejoras, los sistemas de memoria compartida deben ejecutar varias aplicaciones al mismo tiempo o utilizar versiones paralelas de aplicaciones secuenciales. Esta tesis doctoral se centra en el segundo problema, estudiando cómo una sola tarea puede ser ejecutada en paralelo por diferentes procesadores.

R.1.1 Sistemas multiprocesador

Como se ha expuesto, las máquinas multinúcleo son capaces de ejecutar tanto varios programas a la vez, como solamente uno dividido en tareas más pequeñas. Para alcanzar mejoras en la ejecución de un programa secuencial con un ordenador multinúcleo, es necesario que los programas se puedan dividir en tareas independientes. Si las tareas tienen dependencias entre ellas y se ejecutan en un orden incorrecto, es más que probable que la ejecución produzca resultados erróneos.

Localizar aquellas partes de un código que podrían ejecutarse de forma independiente es una tarea tediosa y propensa a provocar errores porque se deben tener en cuenta factores como la sincronización para evitar resultados indeseados. Actualmente existen lenguajes específicos, así como extensiones de los lenguajes secuenciales y librerías de funciones centradas en facilitar el proceso de paralelización. Sin embargo, para paralelizar un programa secuencial el programador debe (a) conocer las características del hardware inherente, (b) entender el problema resuelto en el código, y (c) conocer el modelo de programación a utilizar. Además, desarrollar software paralelo para un arquitectura específica provocaría que éste no fuese portable a otras máquinas. Estos hechos hacen de la paralelización automática una idea muy atractiva.

Actualmente, existen algunos compiladores capaces de paralelizar fragmentos de código (principalmente los bucles). Sin embargo, los compiladores paralelos rehusan paralelizar un bucle si tienen la más mínima sospecha de que haya una violación de dependencia entre las instrucciones del bucle. Determinar o predecir qué instrucciones dependen de otras es una tarea compleja debido a la explosión combinatoria que provoca la existencia de múltiples flujos de control. Además, los valores de ciertas variables pueden no ser conocidos en tiempo de compilación, previniendo la paralelización. Estos hechos son los que más limitan la paralización en tiempo de compilación.

Cabe mencionar que existen ciertas herramientas para controlar estas situaciones, como las barreras, pero su uso limita de forma significativa el rendimiento, llegando a provocar que el código paralelo resultante sea incluso más lento que el código secuencial original.

R.1.2 Violaciones de dependencia

Para entender mejor en qué consisten las violaciones de dependencia, debemos mencionar que éstas se clasifican de acuerdo a su uso. Así los modelos de programación paralela compartida clasifican las variables como *compartidas* o *privadas*.

Las variables privadas son aquellas que son escritas siempre antes de ser leídas en una misma iteración. Como se puede deducir, el ámbito de este tipo de variables es una misma iteración, y no más allá de ella. Por otro lado, el ámbito de las variables compartidas se extiende a varias iteraciones, e incluso a todas. Por lo tanto, si un *thread* utiliza una variable compartida, y más tarde otro *thread* de una iteración *previa* escribe un valor nuevo en ella, se producirá una violación de dependencia.

Como puede deducirse, un bucle que sólo utiliza variables privadas y variables compartidas de sólo lectura es paralelizable. Por el contrario, aquellos bucles cuyas instrucciones realizan operaciones tanto de lectura, como de escritura sobre variables compartidas probablemente darán lugar a violaciones de dependencia en ejecuciones paralelas.

Tipos de violaciones de dependencia

Hay tres tipos de dependencias de datos entre iteraciones:

- `Escritura-tras-escritura:` Este tipo de violaciones de dependencia se producen cuando una variable se escribe en dos iteraciones diferentes, sin lecturas entre las escrituras. Así, una ejecución desordenada de las iteraciones producirá una violación de dependencia, ya que el valor final de la variable podría no ser correcto.

- `Escritura-tras-lectura:` Este tipo de dependencia de datos aparecen cuando una variable local que ha sido leída previamente en una iteración previa se escribe en una iteración posterior. De nuevo, una ejecución paralela desordenada puede provocar una violación de dependencia.

- `Lectura-tras-escritura:` Este tipo de dependencia (la más peligrosa de todas) se produce cuando una variable se escribe con un valor que va a ser leído más tarde en iteraciones sucesivas. Si ambas operaciones se llevan a cabo en un orden incorrecto, el valor leído puede ser incorrecto.

R.1.3 Paralelización de códigos con dependencias

Aunque el problema de paralelizar bucles con dependencias puede ser algo arduo de resolver, existen algunas técnicas capaces de paralelizar tales códigos. Mientras las soluciones clásicas rechazan paralelizar cualquier código que pueda provocar violaciones de dependencia, otras propuestas como el *Inspector-Ejecutor* o la *Paralelización Especulativa* lidian con estas situaciones. Estas técnicas confían en que la ejecución seguirá un orden secuencial, detectando cualquier eventual violación de dependencia y llevando a cabo las operaciones requeridas para conseguir los resultados correctos. A continuación, describiremos ambas técnicas.

Inspector-Ejecutor

Esta técnica [165] trata de paralelizar bucles que no pueden ser paralelizados por un compilador. Su funcionamiento se basa en encontrar dependencias entre las iteraciones de un bucle mediante el uso de un bucle inspector extraído del bucle original. El bucle inspector trata de asignar todas las iteraciones dependientes entre sí al mismo procesador para que la ejecución preserve un orden correcto. Más tarde, un bucle ejecutor lanza las iteraciones en paralelo. Aunque esta técnica puede ser aplicada a cualquier tipo de bucle, su uso sólo se aconseja si el tiempo de procesamiento del bucle inspector es bastante menor que el tiempo de ejecución del bucle original. Desafortunadamente, ésta no suele ser la tónica general debido al coste de inspeccionar los bucles que utilizan aritmética de punteros, siguen flujos de control complejos, o dependen de datos de entrada.

R.1.4 Paralelización especulativa

La paralelización especulativa [49, 73, 96, 144, 273] trata de extraer paralelismo de bucles que no pueden paralelizarse en tiempo de ejecución. En otras palabras, esta técnica trata de paralelizar códigos con dependencias. Ésta técnica asume de forma optimista que no se producirán violaciones de dependencia y ejecuta los bucles en paralelo. Mientras tanto, un monitor software o hardware se encarga de controlar que la ejecución sea correcta, realizando las correcciones oportunas de ser necesario.

Aunque los mecanismos basados en hardware no añaden ni cambios en el código, ni sobrecargas en la ejecución especulativa, requieren cambios en el procesador y/o los subsistemas caché. Por otro lado, los sistemas basados en software requieren cambios en el código original de los bucles, incluyendo la adición de algunas instrucciones que controlan la ejecución y vigilan la aparición de violaciones de dependencia. A pesar de la sobrecarga de rendimiento inherente a estas instrucciones, las soluciones basadas en software pueden

implementarse en los sistemas actuales de memoria compartida sin ningún cambio en el hardware subyacente. Esta tesis doctoral se centra en la rama software de este campo.

Modelo de ejecución

Recordemos que, en caso de detectarse una violación de dependencia, los resultados calculados hasta el momento por el *thread* que utilizó el valor incorrecto de la variable y sus sucesores ya no son válidos y deben descartarse, siendo estos *threads* re-ejecutados en orden. Tras resolver este problema la paralelización optimista puede continuar. Obviamente, se este proceso de parada y re-ejecución requiere una gran cantidad de tiempo adicional, por tanto, cuantas menos violaciones de dependencia se produzcan, mejores resultados obtendremos en términos de rendimiento.

La solución más recurrente utilizada por las soluciones software para evitar tantas violaciones de dependencia como sea posible se denomina *forwarding*. Si un *thread* tiene que leer un valor de una variable compartida, debe leer el valor más reciente almacenado en la misma, para lo cual copia el valor más reciente de los disponibles en los *threads* predecesores al *thread* consumidor. En caso de que se necesite escribir sobre una variable compartida, tras la escritura, el *thread* debe comprobar si un *thread* sucesor ha utilizado un valor desfasado, para detectar tan pronto como sea posible las violaciones de dependencia que puedan surgir.

Otra solución típica adoptada por las soluciones software es el uso de una versión propia de los datos compartidos por cada *thread* en ejecución. Así, los *threads* sólo modifican sus propias copias de las variables compartidas, en lugar de la versión global. Cuando un *thread* finaliza la ejecución de su bloque de iteraciones, si no se ha producido ninguna violación de dependencia, los resultados se guardan en la variable global compartida por todos los *threads* (esta operación se conoce como consolidación). Por otro lado, si se produjese una violación de dependencia, todas las versiones locales con valores erróneos deben ser descartadas (esta operación se denomina *squash*). Después tanto el *thread* en cuestión, como sus sucesores deben comenzar de nuevo la ejecución ya que podrían haber utilizado un valor contaminado del *thread* que sufrió una violación de dependencia. La ventaja de utilizar copias locales es que la operación más frecuente (lecturas sobre variables compartidas) puede llevarse a cabo bastante rápido, siempre que exista una copia local. Nos gustaría destacar que este proceso de parada y re-ejecución conlleva un tiempo, por tanto, cuantas más violaciones de dependencia aparezcan, peores resultados se obtendrán con esta técnica.

Operaciones principales de ejecuciones especulativas

Los sistemas software de paralelización especulativa necesitan modificar el código fuente original de la aplicación en tiempo de compilación para realizar las siguientes tareas:

- *Operaciones especulativas de lectura y escritura:* Cada *thread* tiene su propia versión de las variables compartidas, por lo que todas las operaciones de lectura y escritura en variables compartidas deben sustituirse por una función que también comprueba que no aparezca ninguna violación de dependencia.

- *Consolidación de resultados:* Al final de la ejecución correcta de un bloque de iteraciones, se debe llamar a una función para consolidar los resultados producidos y solicitar un nuevo bloque de iteraciones consecutivas.

- *Reparto de bloques de iteraciones consecutivas:* Las iteraciones deben distribuirse a lo largo de todos los *threads* disponibles que intervienen en la ejecución especulativa. Esta tarea se puede realizar siguiendo diferentes estrategias: distribuyendo bloques de iteraciones de tamaño constante; adaptando el tamaño de los bloques a las características de cada aplicación; o decidir dinámicamente el tamaño del siguiente bloque de iteraciones.

R.2 Objetivos de esta tesis

El diseño de un sistema de paralelización especulativa competitivo requiere abordar diferentes problemas. Como hemos mencionado anteriormente, las operaciones involucradas en la paralelización especulativa influyen directamente en el rendimiento de una solución especulativa. Por lo tanto, mejorar todo lo posible los mecanismos de acceso a las estructuras de datos de una biblioteca de paralelización especulativa, influirá directamente en su rendimiento. Así, parece más que interesante, dedicar esfuerzos a acelerar los accesos a la estructura de datos.

Asimismo, hay muchos códigos con violaciones de dependencias que presentan aritmética de punteros, imposibilitando la aplicación de las técnicas clásicas de paralelización especulativa. Por ello, sería útil mejorar una biblioteca de paralelización especulativa para que soporte códigos con este tipo de variables.

Por otro lado, distribuir correctamente las iteraciones entre los *threads* de una ejecución especulativa, influye de forma muy significativa. Por ello, sería aconsejable trabajar en el desarrollo de nuevas técnicas de reparto dinámico de iteraciones basadas en la historia reciente de violaciones de dependencia, esperando reducir los tiempos de ejecución.

Además, aunque utilizan diferentes enfoques para resolver distintos problemas, hay bastantes similitudes entre la memoria transaccional y la paralelización especulativa. Por tanto, sería de utilidad combinar las dos soluciones de forma que las secciones críticas utilizadas en la paralelización especulativa se beneficien de la memoria transaccional.

Finalmente, con la llegada de las máquinas 'many-core', han surgido nuevas posibilidades respecto a las técnicas de paralelización. Respecto al estado del arte en el campo de la paralelización especulativa, no se habían realizado estudios previos relativos a los coprocesadores Intel Xeon Phi. Por tanto, sería deseable evaluar el comportamiento de una librería especulativa en un coprocesador Intel Xeon Phi.

R.2.1 Pregunta de investigación

De acuerdo con los problemas identificados, podemos proponer las siguientes preguntas de investigación a resolver en esta tesis doctoral:

> *¿Es posible desarrollar un sistema de paralelización especulativa en tiempo de ejecución capaz de utilizar eficientemente estructuras de datos complejas, utilizar aritmética de punteros y tener en cuenta la tendencia de violaciones de dependencia producida hasta ahora para estimar el mejor tamaño de bloque a repartir? ¿Podría implementarse, dando lugar a buenos resultados experimentales, utilizando memoria transaccional y en una arquitectura con un gran número de núcleos como los coprocesadores Intel Xeon Phi?*

Para contestar estas preguntas de investigación, necesitamos llevar a cabo algunos objetivos intermedios, más específicos:

Desarrollar un estudio en profundidad del estado del arte en paralelización especulativa

> *Establecer una base sólida de los problemas resueltos, los desafíos existentes y proponer una clasificación de las soluciones existentes en el campo especulativo.*

Dado que durante la última década se han realizado multitud de trabajos relacionados con la paralelización especulativa, consideramos que realizar un análisis tanto de las soluciones conocidas, como de las menos conocidas será un buen punto de partida. Además ya que no existe un artículo que sintetice todo este conocimiento, hemos desarrollado un compendio de paralelización especulativa como parte de nuestra investigación.

Combinar un librería especulativa con un compilador

> *Proponer e implementar una librería de paralelización especulativa que puede ser combinada fácilmente con un compilador para obtener una herramienta general de paralelización especulativa.*

Hasta donde tenemos entendido, los compiladores actuales no soportan ejecución especulativas automáticamente, en otras palabras, no pueden transformar un bucle directamente para paralelizarlo especulativamente. Por lo tanto, una de los objetivos de esta tesis doctoral es alcanzar este objetivo. Cabe mencionar que este propósito ha sido llevado a cabo con la inestimable ayuda del Dr. Sergio Aldea. Mientras su trabajo se centraba en el desarrollo de la parte del compilador, el nuestro se ha focalizado en la implementación de la librería especulativa que se acopla con su parte.

Mejorar el rendimiento de las operaciones involucradas en una librería especulativa

Encontrar un manera de reducir el tiempo requerida por las operaciones más costosas relacionadas con la librería especulativa.

Aunque el desarrollo de una librería especulativa en sí misma ya es una tarea bastante compleja, si no se logra obtener resultados mejores que los secuenciales, los esfuerzos realizados serán inútiles. Por tanto, mejorar tanto como sea posible el rendimiento de nuestra solución es algo más que recomendable. Así hemos descubierto como recorrer eficientemente las estructuras de datos involucradas en las principales operaciones de la librería, logrando una mejora de los procedimientos más pesados en términos de rendimiento. Para ello, hemos desarrollado algunas estructuras de datos que reducen tanto el espacio, como el tiempo requeridos por las soluciones clásicas.

Por otro lado, no hemos encontrado demasiados artículos que estudien el reparto de iteraciones en contextos de paralelización especulativa. Consecuentemente, buscar una solución eficiente, así como desarrollarla nos permitirá probar si el rendimiento se ve influido por este aspecto.

Combinar nuestro sistema especulativo con otras técnicas paralelas

Sugerir, implementar y probar soluciones híbridas basadas en nuestra librería especulativa y otras herramientas actuales.

Las técnicas paralelas están en constante evolución, proponiendo nuevos métodos y/o tecnologías. Por tanto, esta tesis doctoral no estaría completa sin combinar una librería de paralelización especulativa con otras técnicas paralelas para probar si combinándolas, se obtendrían mejoras. Por ello hemos combinado algunas de las técnicas más conocidas de memoria transaccional, con nuestra librería especulativa. Además, hemos adaptado el código de nuestro sistema a la interfaz de los coprocesadores Intel Xeon Phi.

R.3 Metodología de investigación

Para el desarrollo de esta tesis doctoral se ha utilizado la metodología de investigación definida por Adrion [1]. Este enfoque, denominado método de investigación para ingeniería del software, adapta las fases clásicas del método científico hipotético-deductivo a la Informática. Así consta de cuatro fases diferentes que pueden repetirse cíclicamente hasta la consecución de los objetivos.

1. *Observar las soluciones existentes.*

Una vez elegido el campo de investigación donde se va a trabajar, es requisito obligatorio conocer el estado del mismo. De otra forma, se podrían desarrollar soluciones existentes, o algo que carezca de utilidad. Por ello, estudiaremos el campo de la especulación en profundidad.

2. *Proponer soluciones mejores.*

   Tras analizar toda la información, probablemente se localice un nuevo enfoque que mejore los existentes. Así, hemos propuesto una librería de paralelización especulativa capaz de ejecutar códigos complejos, que puede combinarse fácilmente con un compilador para facilitar enormemente su uso. Además, hemos desarrollado una estructura de datos capaz de reducir los accesos requeridos por las operaciones especulativas más costosas. También hemos detectado que los métodos de reparto de iteraciones no han sido suficientemente investigados en el contexto de la paralelización especulativa. Por ello, hemos desarrollado un algoritmo capaz de adaptarse dinámicamente al número de violaciones de dependencia que aparecen para asignar de forma más optimista (o pesimista) las iteraciones. Por último, sería útil combinar dos técnicas de paralelización optimista.

3. *Construir o desarrollar la solución.*

   Durante esta etapa se deben implementar las soluciones propuestas en la etapa anterior. Por tanto, hemos desarrollado la librería especulativa, así como el nuevo método de reparto de iteraciones. Además hemos combinado nuestra solución con otra técnica de paralelización optimista como es la memoria transaccional.

4. *Medir y analizar la nueva solución.*

   Por último, todo prototipo debe probarse para medir si lo realizado es útil o no. En nuestro caso hemos comparado las versiones paralelas contra las secuenciales. Para ello hemos utilizado tanto aplicaciones sintéticas, como aplicaciones de reales. Cabe mencionar que hemos utilizado varios tipos de máquinas, con una gran diversidad de procesadores, desde sistemas con memoria compartida, sistemas con memoria transaccional, hasta coprocesadores Intel Xeon Phi.

R.4 Resumen de contribuciones

A continuación exponemos las contribuciones de esta tesis doctoral ordenadas por objetivos, así como las publicaciones obtenidas con las mismas.

R.4.1 Desarrollar un estudio en profundidad del estado del arte en paralelización especulativa

Hemos revisado la mayoría de las soluciones existentes en el campo de paralelización especulativa. Así hemos propuesto una clasificación para enmarcar todas ellas. Creemos que éste es un trabajo que nadie había realizado previamente, y que puede servir tanto para investigadores nóveles en este campo, como para veteranos. El trabajo realizado ha sido aceptado para su publicación en la siguiente revista:

1. Alvaro Estebanez, Diego R. Llanos y Arturo Gonzalez-Escribano. 'A survey on Thread-Level Speculation Techniques'. En: *ACM Computing Surveys (CSUR)*. Accepted for publication

R.4.2 Combinar un librería especulativa con un compilador

Hasta la fecha, no había ninguna solución centrada en combinar la paralelización especulativa con un compilador, hasta que Aldea et *al.* propusieron la suya [7]. Para ello utilizaron una librería de paralelización especulativa capaz tanto de ejecutar aplicaciones con operaciones complejas, como de manejar aritmética de punteros. Por lo tanto hemos implementado una librería capaz de combinarse con un compilador de manera transparente y ejecutar todo tipo de aplicaciones. Esta contribución ha sido publicada en los siguientes artículos:

2. Alvaro Estebanez, Diego R. Llanos y Arturo Gonzalez-Escribano. 'Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros'. En: *Proceedings of the XXIII Jornadas de Paralelismo*. Elche, Alicante, Spain: Servicio de Publicaciones de la Universidad Miguel Hernández, sep. de 2012. ISBN: 978-84-695-4471-6

3. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos y Arturo Gonzalez-Escribano. 'A New GCC Plugin-Based Compiler Pass to Add Support for Thread-Level Speculation into OpenMP'. English. En: *Euro-Par 2014 Parallel Processing*. Ed. por Fernando Silva, Inês Dutra y Vítor Santos Costa. Vol. 8632. Lecture Notes in Computer Science. Springer International Publishing, 2014, págs. 234-245. ISBN: 978-3-319-09872-2. DOI: 10.1007/978-3-319-09873-9_20. URL: http://dx.doi.org/10.1007/978-3-319-09873-9_20

4. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos y Arturo Gonzalez-Escribano. 'Una extensión para OpenMP que soporta paralelización especulativa'. En: *Proceedings of the XXV Jornadas de Paralelismo*. Valladolid, Spain, sep. de 2014. ISBN: 978-84-697-0329-3

5. S. Aldea, A. Estebanez, D.R. Llanos y A. Gonzalez-Escribano. 'An OpenMP Extension that Supports Thread-Level Speculation'. En: *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, n.º 99, 2015, págs. 1-14. 2015. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2393870

R.4.3 Mejorar el rendimiento de las operaciones involucradas en una librería especulativa

Como resultado de nuestro objetivo anterior, y para incrementar el posible impacto de nuestra herramienta, hemos tratado de localizar y aliviar los principales cuellos de botella de la librería especulativa. Para ello, hemos propuesto una estructura de datos basada en un *hash* que permite decrementar el número de accesos involucrados en las operaciones especulativas. Esto que nos permitió mejorar considerablemente el rendimiento de nuestra librería.

Además, aunque el reparto de iteraciones es un campo ampliamente estudiado, los trabajos no se han centrado específicamente en la paralelización especulativa. Por tanto, tras revisar algunos de los algoritmos existentes, nos hemos dado cuenta de que no había soluciones focalizadas en obtener dinámicamente el mejor tamaño de bloque de iteraciones basándose en parámetros de ejecución. Así hemos desarrollado una técnica que utiliza las violaciones de dependencia acontecidas como factor determinante en la asignación del siguiente bloque de iteraciones. Estos logros han producido las siguientes publicaciones:

6. Alvaro Estebanez, Diego R. Llanos y Arturo Gonzalez-Escribano. 'Improving the Perfomance of a Pointer-Based, Speculative Parallelization Scheme'. En: *Proceedings of the 1st First Congress on Multicore and GPU Programming*. PPGM'14. Granada, Spain, feb. de 2014. Also published in *Annals of Multicore and GPU Programming*, vol. 1, no. 1, 2014. 2014. issn: 2341-3158.

7. Alvaro Estebanez, Diego R. Llanos y Arturo Gonzalez-Escribano. 'New Data Structures to Handle Speculative Parallelization at Runtime'. En: *Proceedings of the 7th International Symposium on High-level Parallel Programming and Applications*. HLPP '14. Amsterdam, Netherlands: ACM, 2014, págs. 239-258. Also published in *International Journal of Parallel Programming*, 2015, pp. 1–20. Springer US, 2015. issn: 0885-7458. doi: 10.1007/s10766-014-0347-0. url: http://dx.doi.org/10.1007/s10766-014-0347-0.

8. Alvaro Estebanez, Diego R. Llanos, David Orden y Belen Palop. 'Moody Scheduling for Speculative Parallelization'. English. En: *Euro-Par 2015: Parallel Processing*. Ed. por Jesper Larsson Träff, Sascha Hunold y Francesco Versaci. Vol. 9233. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015, págs. 135-146. *ISBN*: 978-3-662-48095-3. *DOI*: 10.1007/978-3-662-48096-0_11. *URL*: http://dx.doi.org/10.1007/978-3-662-48096-0_11

R.4.4 Combinar nuestro sistema especulativo con otras técnicas paralelas

Para completar nuestro trabajo, hemos probado nuestra librería especulativa en otros contextos distintos a las máquinas de memoria compartida convencionales. Así hemos combinado la memoria transaccional con nuestra librería especulativa para comparar como influye el

uso de secciones críticas de forma convencional, o el uso de las transacciones software o hardware. Además hemos examinado el comportamiento de nuestro software con uno de los dispositivos más vanguardistas, un coprocesador Intel Xeon Phi. Este trabajo nos ha permitido publicar los siguientes artículos:

9. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos y Arturo Gonzalez-Escribano. 'Study and Evaluation of Transactional Memory approaches with a Software Thread-Level Speculation Framework'. En: *IEEE Transactions on Parallel and Distributed Systems*. To be submitted

10. Alvaro Estebanez, Diego R. Llanos y Arturo Gonzalez-Escribano. 'Evaluating the capabilities of the Xeon Phi platform in the context of software-only, thread-level speculation'. En: *Proceedings of the 8th International Symposium on High-level Parallel Programming and Applications*. HLPP '15. Pisa, Italy: ACM, 2015. To be also published in *International Journal of Parallel Programming*, Springer US.

R.5 Respuesta a la pregunta de investigación y conclusiones

> *¿Es posible desarrollar un sistema de paralelización especulativa en tiempo de ejecución capaz de utilizar eficientemente estructuras de datos complejas, utilizar aritmética de punteros, y tener en cuenta la tendencia de violaciones de dependencia producida hasta ahora para estimar el mejor tamaño de bloque a repartir? ¿Podría implementarse, dando lugar a buenos resultados experimentales, utilizando memoria transaccional, y en una arquitectura con un gran número de núcleos como los coprocesadores Intel Xeon Phi?*

Como resultado de nuestra investigación, podemos afirmar que la pregunta de investigación ha sido contestada claramente. Por un lado, hemos desarrollado un software de paralelización especulativa capaz de (a) controlar estructuras de datos complejas, y (b) utilizar aritmética de punteros. Además, hemos propuesto (c) una estrategia de reparto de iteraciones que tiene en cuenta no solo parámetros de tiempo de ejecución como las violaciones de dependencia, si no también el optimismo que el usuario quiera implementar en cuanto a la asignación de bloques de iteración más grandes o más pequeños.

También hemos verificado que (d) la paralelización especulativa y la memoria transaccional se pueden combinar con resultados similares a los de las técnicas clásicas, y (e) que la paralelización especulativa puede utilizarse en los coprocesadores Intel Xeon Phi.

R.6 Agradecimientos

Esta Tesis está parcialmente financiado por el Ministerio de Industria (CENIT OCEAN-LIDER), Ministerio de Ciencia e Innovación y el Fondo Europeo de Desarrollo Regional (MOGECOPP project TIN2011-25639, CAPAP-H3 network TIN2010-12011-E, CAPAP-H4 network TIN2011-15734-E, CAPAP-H5 network TIN2014-53522-REDT).

Chapter 1

Introduction

Since the invention of integrated circuit, computer performance has been improving exponentially. However, this progression quickly reached different physical limits at the beginning of the 21st century, being power dissipation the most remarkable one. Instead of augmenting clock speed even more, computer architects decided to pack several computational units in the same chip, starting the era of the affordable, shared-memory multiprocessor. However, to take advantage of these architectures in the execution of a single application, sequential code should be either rewritten as a set of parallel tasks, or automatically transformed into parallel code. In this chapter we will explain why the latter is a non-trivial problem, how speculative parallelization techniques may help, and which are the goals of this Ph.D. Thesis.

1.1 Motivation

Since the advent in 1971 of the first microchip, the Intel 4004, with about 2300 transistors and a frequency of 104 KHz [16, 122], computing machines suffered a dramatic evolution, with exponential increments on the number of transistors per surface unit and a corresponding increment in computational performance. This astounding improvement rate quickly reaches physical limits at the beginning of the 21st century, such as the difficulties associated to extract all the heat produced by a chip of just a few square millimeters that consumes more than $100W$.

To keep growing the rate of performance, computer architects started to pack several, simpler computational units in the same chip. This solution took advantage of the increment in the number of transistors that can be packed together[1], while distributing "hot spots" on different areas of the chip, thus simplifying cooling. This solution also allowed to decrease the peak frequency used, consequently making this solution affordable for its use on battery-powered systems.

Nonetheless, for the very first time in computing history, an architectural improvement does not automatically lead to better performance of sequential applications. To take advantage of these new capabilities, these shared-memory parallel systems should either be used to run several applications at once, or run a parallel version of a sequential application. This Ph.D. Thesis is focused on the last problem, studying how a single task can be automatically executed by different processors in parallel.

1.1.1 Multiprocessor computers

As stated before, multicore machines are capable of executing not only several programs simultaneously, but also the same program divided on smaller tasks. To achieve improvements in the execution of a sequential program with a multicore computer, it is necessary that programs can be decomposed on independent tasks. If tasks have *dependences* among them, and we execute them in the wrong order, the execution will likely lead to incorrect results.

To isolate the parts of a code that may be executed independently is a tedious, error prone task because factors such as synchronization have to be taken into account to avoid undesirable results. There already exist some specific languages, as well as extensions to sequential languages, and function libraries centered on ease users this parallelization process. However, in order to successfully parallelize a sequential program, the programmer should (a) know the characteristics of the underlying hardware, (b) understand the problem to be solved by the code, and (c) know the parallel programming model being used. In addition, parallel software that is tailored for a specific architecture may not be portable to other systems. These facts make automatic parallelization a very appealing idea.

[1] This is another example of an improvement that might soon reach its own physical limit.

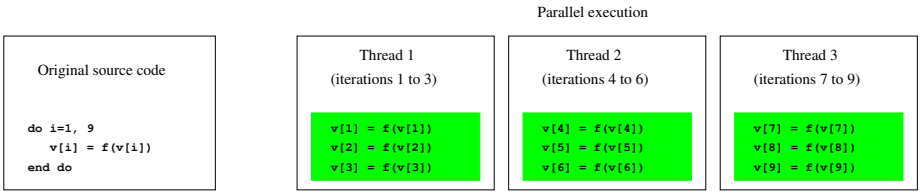


Figure 1.1: Loop without dependences between iterations.

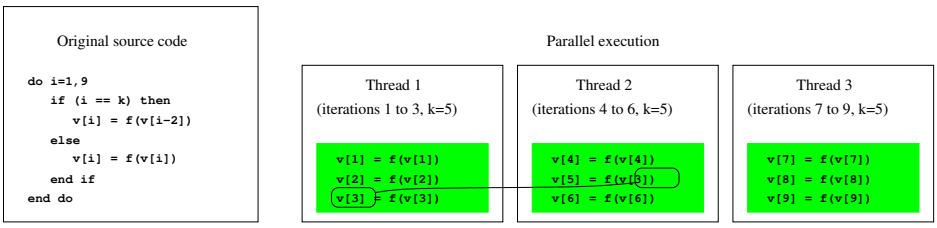


Figure 1.2: Loop with a dependence between two iterations.

Nowadays, there are some compilers capable of parallelizing fragments of a code (mainly loops). However, parallelizing compilers conservatively refuses to parallelize a loop if there is the slightest possibility of a dependence violation among its iterations. Determining or predicting which instructions depend on others is a rather difficult task, because of the combinatorial explosion due to the existence of multiple control paths. Moreover, values of certain variables may not be known at compile time, thus preventing parallelization. We will further explore this point with an example.

Figure 1.1 shows a loop without dependence violations: All the instructions are independent, so the code may be safely parallelized at compile time. On the other hand, Figure 1.2 depicts a loop that may produce some dependence violations. Supposing that the value of k is unknown at compile time, and assuming that it will be $k = 5$ at runtime, if the parallel execution of the loop executes iteration i (with $i = 5$ in our example) before iteration $i - 2$ ($i - 2 = 3$ in our example), the value of $v[i - 2]$ (that is, $v[3]$) will return an outdated value, breaking sequential semantics. This issue, called *dependence violation*, appears when a given thread produces a datum that has already been consumed by one of its successors, with respect to the original, sequential order. So as to guarantee a right behavior, it is necessary to serialize the execution of iterations $i - 2$ and i, a difficult task in the general case[2]. This one is the most severe limitation to compile-time parallelization techniques.

[2]Note that if the dependence did not cross thread boundaries (for example, with $k = 6$), the compiler could have parallelized the loop safely.

```
for (i=0; i<100; i++)
{
  localVar = sharedVarA+sharedVarB;
  sharedVarA = i*localVar;
}
```

Listing 1.1: Example of a code with private and shared variables.

There exist some parallel constructs to handle these situations, such as barriers, but their use severely affects performance, sometimes making the resulting parallel code even slower than the original, sequential one.

1.1.2 Dependence violations

In order to understand when a dependence violation may take place, we will first show how variables can be classified according to their use. When using a shared-memory programming model, variables can be classified as either *shared* or *private*.

Informally speaking, private variables are those whose values are always written before being read at a given iteration. Their intended scope is the iteration, and not beyond it. On the other hand, shared variables contain values whose scope spans several iterations. Therefore, if a thread consumes a shared value, and later a thread running an *earlier* iteration writes a new value onto it, a dependence violation will take place.

An example of private and shared variables is shown at Listing 1.1. In this example, localVar is private since it is always written locally before being read in the context of each iteration. On the other hand, sharedVarA and sharedVarB are shared variables. In the case of sharedVarB, it is a read-only variable, so it will not trigger any dependence violation. Regarding shareVarA, it is read before being written, so reads and writes that do not follow sequential semantics are likely to trigger a dependence violation.

A loop that only uses private and read-only shared variables is parallelizable. On the contrary, loops that performs reads and writes on shared variables are expected to suffer dependence violations when executed in parallel.

Types of data dependences

There are three types of data dependences among iterations:

- `Write-after-write (WAW):` This kind of data dependence is produced when a variable is written in two different iterations, with no reads between the writes. An incorrect ordering in the execution of both iterations leads to a dependence violation.

```
for (i=0;i<4;i++)
{
  if (i==1)
    localVar = 4;
  if (i==3)
    localVar = 7;
}
```

Listing 1.2: Example of Write-after-write (WAW) dependence violation.

```
// Suppose that at the beginning
// localVarA == 3
for (i=0;i<4;i++)
{
  if (i==1)
    localVarB = localVarA;
  if (i==3)
    localVarA = 5;
}
```

Listing 1.3: Example of Write-after-read (WAR) dependence violation.

Listing 1.2 shows an example. In this code, the `localVar` variable is written twice, first at iteration 2, and again at iteration 4. In a sequential code, the value obtained at the end of the loop in `localVar` would be 7. However, if we suppose a parallel execution with two processors, we cannot guarantee that iteration 2 is executed before iteration 4, and therefore, it cannot be ensured that the value of `localVar` at the end of the loop is 7 (the value might be erroneously 4).

- `Write-after-read (WAR):` This type of data dependence appears when a local variable that has been previously read in a previous iteration is written with a new value in a subsequent iteration. Again, an incorrect ordering of both iterations may trigger a dependence violation.
  Listing 1.3 shows an example of this data dependence. Variable `localVarA` is read at iteration 2, and later written at iteration 4. In a sequential execution, the value of `localVarA` at the end of the loop would be 5, and the value of `localVarB` would be 3. Now suppose a parallel execution with two processors, where iterations 1 and 2 are assigned to the first one, and iterations 3 and 4 are assigned to the other processor. In this case, it cannot be ensured that iteration 2 is executed before iteration 4, so the value of `localVarB` at the end of the loop may not be 3.

```
// Suppose that at the beginning...
// sv = 0;
// LocalVar = 1;
for (i=0;i<3;i++)
{
  localVar = sv;
  sv = 20;
  if (i==1)
  {
    sv = 10;
  }
}
```

Listing 1.4: Example of Read-after-write (RAW) dependence violation.

- `Read-after-write (RAW):` This data dependence is the most difficult to track. RAW dependence violations occur when a shared variable is written with a value that will be read by subsequent iterations. If both operations are carried out in the wrong order, the value read may be wrong.
  Listing 1.4 shows an example. A sequential execution of that code will produce the values `localVar = 10` and `sv = 20`. If we run this code in parallel using three processors, each one executing a single iteration, a parallel execution is likely to lead to wrong values. The exact values being produced depend on the particular ordering of the instructions being executed. For example, let us assume that at instant `t1` the first instruction of threads 1 and 2 are executed, so `localVar` versions of both threads are `0`. After that, at instant `t2`, the following instruction of thread 1 is executed, setting `sv = 20`. Once completed, at instant `t3` the first instruction of thread 3 is executed, triggering a dependence violation, since `localVar = 20` is consumed, instead of the value `10` that would follow sequential semantics. But problems do not stop here. If we suppose that at instant `t4` the next instruction of thread 3 is executed, the value of `sv` will be `20`. And consequently, if at instant `t5` the last instruction of thread 2 is executed, `sv` will be `10`. At the end of the execution, the values of variables would be either `localVar = 20` or `localVar = 0`, and `sv = 10`, and the parallel executions would have failed. All those operations are summarized in the Table 1.1.

1.1.3 Parallelization of codes with dependences

Although the problem of parallelizing loops with dependences may seem hard to tackle, there are some techniques which allow the parallelization of such complex codes. Whereas classical solutions simply refuse to parallelize code whenever a single dependence might take

	Thread 1		Thread 2		Thread 3	
Instant	localVar	sv	localVar	sv	localVar	sv
t1	0	0	0	0	1	0
t2	0	20	0	20	1	20
t3	0	20	0	20	20	20
t4	0	20	0	20	20	20
t5	0	10	0	10	20	10

Table 1.1: Timing of the loop with the values of each variable at any time in the supposed execution.

place, other proposals, such as *Inspector-Executor* or *Thread-Level Speculation* handles these situations. These techniques optimistically suppose that the whole execution will follow sequential order. If any dependence violation occurs, they will detect this situation and perform the required operations so as to achieve a correct result. A brief description of both techniques follows.

Inspector-Executor

This technique [165] aims to parallelize loops that cannot be parallelized by a compiler. It is mainly based on finding data dependences among the iterations of a loop through the use of an inspector loop extracted from the original loop. The inspector loop tries to assign every single iteration that depends on previous ones to the same processor, so that the execution is ensured to be done in the correct order. A second, executor loop, carries out the execution of each chunk of iterations in parallel. Although this technique can be applied to any loop, its use is only advisable if the processing time of the inspector loop is quite lesser than the execution time of the original loop. Unfortunately, this does not happen in the general case, due to the cost of inspecting loops that use pointer arithmetic, complex control flows or depend on input data. This proposal was later enhanced with different improvements, that will be discussed on Section 2.3.

1.1.4 Speculative Parallelization

Thread-Level Speculation (TLS) [49, 215, 218, 241], also called Speculative Parallelization (SP)[3] [73, 96, 144, 273] or Optimistic Parallelism [157, 158, 159, 160, 161, 162] tries to extract parallelism of loops that cannot be ensured to be fully parallel at compile time. In other words, this technique aims to parallelize codes with dependences. TLS optimistically assumes that dependence violations will not occur, launching the parallel execution of the

[3] From now on, TLS and SP concepts will be used throughout the text interchangeably.

loop. Meanwhile, a hardware or software monitor ensures the correctness of that assumption, taking corrective actions when needed.

The monitoring system may be implemented either in hardware or software. Although hardware mechanisms do not need changes in the code nor add overheads to speculative execution, they require changes in the processors and/or the cache subsystems (see e.g. [51, 113]). On the other hand, software-based systems require changes in the original source code of loops, including the addition of some instructions which manage the execution and take care of possible dependence violations that may take place. Despite the performance overhead introduced by these instructions, software-based speculative parallelization approaches can be implemented in current shared-memory systems without any hardware changes. Since this Ph.D. thesis is centered on the software-based branch, hardware solutions are not going to be detailed, and affirmations exposed are generally referred to solutions made in software.

TLS systems are usually applied to `for` loops. Other loops can be considered as well, but their number of iterations cannot be so easily predicted, and therefore, the applicability of TLS solutions is limited by scheduling problems (see Chapter 5).

Working model

Recall that, when a dependence violation is detected, results calculated so far by the thread that used an incorrect value of the variable and its successors (globally called offending threads) are not longer valid and should be discarded, and these threads should be restarted in order. After solving this issue, the optimistic parallel execution is allowed to proceed. Obviously, a great deal of time is lost stopping and restarting threads, so, the less dependence violations, the better results are obtained in terms of performance.

The most recurrent solution managed by software-based speculative approaches to avoid as much dependence violations as possible is called *forwarding*. If a thread has to read the value of a shared variable, it should read the most recently stored value of it, so the most recent copy of this value is forwarded from a predecessor to the consumer thread. In the case of needing to write over a shared variable, after the write the thread should check whether a successor has used an outdated value, in order to detect as soon as possible dependence violations which may arise. Chapter 2 discusses with more detail the possible implementations of these operations.

Another typical solution adopted by software-based TLS approaches is to provide each thread with its own shared data version. Thus, threads modify only its own copies of the shared variable, instead of the global shared variable. When a thread finishes the execution of its chunk of iterations, if no dependence violations have arisen, results would be saved in the corresponding global, shared variable (this operation is known as *commitment*). On the other hand, in the case of dependence violations, all the local versions of shared data with 'wrong' values should be discarded, in a so-called *squash operation*. Then the thread whose values has been rejected should start the work again, as well as its successors since they might have been forwarded a polluted value from him. The advantage of using local copies is that the most frequent operation (read accesses to shared variables) can be performed

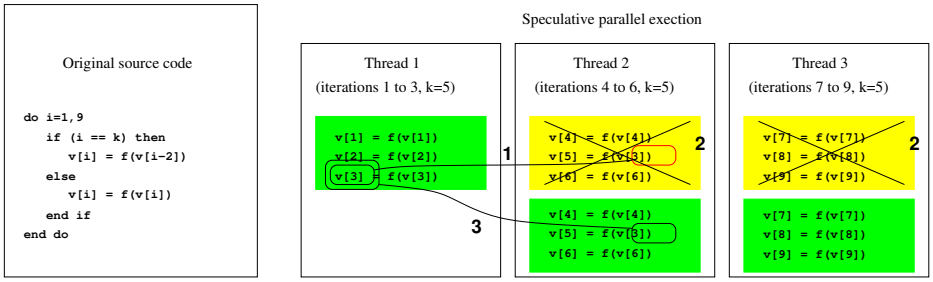


Figure 1.3: Speculative parallelization starts the parallel execution of the loop, while a control system tracks the execution to detect cross-thread dependence violations. If such a violation occurs, (1) speculative parallelization stops the consumer thread and all threads that execute subsequent blocks, (2) discards its partial results, and (3) restarts the threads to consume the correct values.

really quickly, as long as a local copy exists. An example of these situations can be found at Figure 1.3. Let us remark that this stop-and-restart process spends some time, so the more dependence violations appear, the worse results will be obtained with this technique in terms of performance.

Main operations of speculative executions

Software speculative parallelization systems need to modify the original source code of the application at compile time to perform the following tasks:

- *Speculative load and store operations:* Each thread has its own version of the shared variables, so all read and write operations on shared variables should be replaced with a function that should also check that no dependence violations appear.

- *Results commitment:* At the end of a successful execution of a chunk of iterations, a function should be called to commit the results produced and to request a new chunk of consecutive iterations.

- *Scheduling of chunks of consecutive iterations:* Iterations should be distributed throughout all available threads that take part over speculative execution. It can be done with different strategies: Distributing iteration chunks of a constant size; adapting their size to the characteristics of each application; or dynamically deciding the size of the next chunk to launch (see Chapter 5).

1.2 Objectives of this dissertation

The design of a TLS system to be used at production level implies to address different problems. As stated previously, operations involved in speculative parallelization have a direct influence in the performance of a TLS solution. Therefore, improving as much as possible the access mechanisms to data structures of a TLS library will also affect its performance. Thus, efforts devoted to speed up accesses to data structure are worthwhile.

Second, there are many codes with dependence violations which present pointer arithmetic, representing a hurdle to classic speculative parallelization techniques. So, it would be useful to enhance a TLS library so that it supports codes with these special variables.

Third, achieving a correct distribution of iterations among threads of TLS has the potential of improving speedups achieved considerably. Hence, it would be advisable to work in the development of new dynamic scheduling techniques based on the recent history of dependence violations, hoping to reduce execution times.

Fourth, although they use different approaches to solve different problems, there are some similarities between Transactional Memory and TLS. Therefore, it would be helpful to combine both solutions in a way that TLS critical sections can be benefited of Transactional Memory.

Finally, with the advent of many-cores machines, new possibilities have emerged concerning parallel techniques. Regarding the state of the art in the TLS field, no previous studies had been done related to Intel Xeon Phi coprocessors. So, it would be desired to test a TLS library within an Intel Xeon Phi coprocessor.

1.2.1 Research question

According to the identified problems detailed in the previous section, we can state the research questions to be solved in this Ph.D Thesis:

> *Is it possible to develop a runtime system for thread-level speculation able to efficiently handle complex data structures, use pointer arithmetic, and take into account the tendency of dependence violations produced so far to estimate the best chunk size to be scheduled? Could it be implemented and lead to good execution times using Transactional Memory, and in new manycores architectures such as the Intel Xeon Phi coprocessors?*

1.2.2 Milestones

In order to answer these research questions, we need to accomplish some intermediate, more specific objectives:

Goal 1: To perform an In-depth study of the state-of-the-art in TLS

To gain a solid knowledge of problems solved so far and challenges in the field, and to propose a classification of the existent approaches.

Many works have been carried out in the field of speculative parallelization during the las decade. Therefore, we consider that performing an in-depth analysis of both well-known, and not as well recognized solutions, is a good starting point. As long as there does not exist a survey paper that summarizes the status of the developments so far, we developed a survey on TLS as part of our research.

Goal 2: To combine a TLS library with a compiler

To propose and implement a new TLS library which can be coupled with a compiler in order to develop a speculative transformation framework.

To the best of our knowledge current compilers do not support speculative executions automatically, in other words, they cannot directly transform a loop so as to parallelize it speculatively. Because of this, one of the purposes of this Ph.D. thesis is to achieve this goal. This work was carried out with the help of Dr. Sergio Aldea. Whilst he was centered on the development of the compiler side, our main concern was implementing the runtime library which deals with its part. The development of the compile-time transformations are described in Dr. Aldea's Ph.D. thesis, *Compile-Time Support for Thread-Level Speculation* [7].

Goal 3: To improve the performance of operations involved in a TLS runtime library

To find ways of reducing the time needed by the costly operations related to TLS.

Although developing a runtime library which supports speculative parallelization is a hard issue, if no speedup is accomplished, efforts done will be useless. Thus, improving as much as possible the performance of our approach became mandatory. As a result, we found out how to efficiently traverse the data structures involved in the main operations done by the library, leading to an improvement of the most time-consuming procedures. Hence, we developed some data structures which reduce both space and time required by classic approaches.

On the other hand, there have not been many papers centered on scheduling strategies related to TLS. Consequently, searching for existent solutions as well as develop a new strategy has proven to be a good way of checking whether it is an important issue in terms of performance.

Goal 4: To combine our TLS runtime library with other parallel techniques

To suggest, implement, and try out hybrid solutions based on our TLS library and other, state-of-the-art approaches.

Parallel techniques are constantly evolving through both new methods and/or technology. Hence, this Ph.D. thesis will not be fully completed without combining a TLS library with other parallel techniques so as to test if joining them, we will achieve improvements. An implementation which supports vanguard technology will be useful to test new capabilities with TLS approaches as well. Therefore, we combined some Transactional Memory approaches with the TLS runtime library mentioned. Also, we adapted the TLS runtime library to the interface of Intel Xeon Phi coprocessors.

1.3 Research methodology

According to [70, 95], Computer Science cannot be directly classified as a science, but rather as an interdisciplinary science, transversal to very different domains. As a result, it is not clearly defined a standard research methodology which includes the whole areas within this discipline. Consequently, plenty of them have been proposed so far, such as the theoretical, experimental and simulation methods suggested by Freitas [95] and Dodig-Crnkovic [70], or the formal, experimental, build and process methodologies recommended by Amaral [14]. Nonetheless, the main research methodology followed throughout this Ph.D. thesis is the software engineering method described by Adrion [1]. It adapts the specific stages of the classical scientific *hypothetico-deductive method* to Computer Science. Thus, the software engineering method is composed of four stages that may be repeated or not depending on the results achieved. (1) Observe existing solutions, (2) propose better solutions, (3) build or develop them, and (4) measure and analyze the results. Let us review how these different stages will be applied throughout this Ph.D. thesis.

1. *Observe existing solutions.*

   Once the research field is chosen it is almost mandatory put into perspective the existing work. Otherwise, authors might develop something which already exists, or something which is not useful. Hence, we will study literature related to Thread-Level Speculation in-depth.

2. *Propose better solutions.*

   After analyzing all the information, authors will probably find out a new approach which improves the existing ones, at least in a certain manner. We have proposed a new TLS runtime library able to execute complex codes speculatively, which can be easily combined with a compiler in order to be used automatically. In addition,

we have developed a data new structure capable of reducing the accesses required by the most time-consuming speculative operations. Also, we detected that scheduling methods regarding speculative parallelism were not fully studied. So it could be helpful a new approach which takes advantage of the runtime parameters, and takes into account some optimistic or pessimistic parameters defined by users. Furthermore, we thought that combining two optimistic parallel techniques might lead to performance improvements.

3. *Build or develop the solution.*

   During this stage should be implemented all the solutions proposed in the previous one. Consequently, we have developed the new TLS runtime library as well as the novel scheduling method. In addition we have mixed our solution with other parallel technique as Transactional Memory.

4. *Measure and analyze the new solution.*

   At last, every prototype made has to be deeply examined so as to measure the proposed solution. In our case, we will perform some experiments with the software developed. Specifically we will compare sequential versions of applications against those speculatively parallelized with our approaches. To do so, we will use both real-world and synthetic benchmarks. In order to conduct the experiments we needed several machines, from shared-memory systems with a big number of processors, to systems with a Xeon Phi coprocessor or Transactional Memory extensions.

We would also like to note that we have followed some of the guidelines proposed by Berndtsson et *al.* [22] to develop a thesis project.

1.4 Document structure

This document is organized as follows. Chapter 2 details the state of the art of the field of Thread-Level Speculation. It describes in depth the fundamentals of TLS, referencing the most important works of this domain, and also classifying into broad categories the research published so far.

Chapter 3 describes the new TLS runtime library developed. It briefly describes the behavior of the classic solution due to Cintra and Llanos, highlighting its limitations. Then the new solution is described in detail, along with different improvements proposed to reduce TLS bottlenecks.

Chapter 4 gives details about the ATLaS framework, the system which allows to compile a given code with an extended version of the OpenMP programming model that includes a *speculative* clause. This clause eases the process of parallelizing programs that may present

dependence violations at runtime. The ATLaS framework uses our runtime library to speculatively parallelize loops. This chapter describes the compiler phase, giving some experimental results obtained with the help of our library.

Chapter 5 explores current scheduling strategies related to TLS. Not much research has been done regarding the sizes of chunks scheduled in TLS approaches so far, thus this chapter summarizes some of the work carried out papers, also describes a new proposed solution which uses some runtime parameters about dependence violations in order to adapt its behavior either optimistically or pessimistically.

Chapter 6 and Chapter 7 evaluate the adaptability of the TLS runtime library in relation to emerging programming models. Specifically, Chapter 6 contains descriptions of how to combine TLS with Transactional Memory. It reviews some Transactional Memory characteristics, to highlight the differences between this parallel approach and TLS, and also explains how to manage the critical sections of our TLS library through Transactional Memory. Meanwhile, Chapter 7 introduces the use of TLS with the Intel Xeon Phi coprocessor. It describes current state of Xeon Phi coprocessor with respect to our field as well as the steps needed to use a TLS software library within these devices.

Finally, Chapter 8 concludes this Ph.D. thesis, enumerating its contributions, the resulting publications, and some possible ways of continuing this work.

We have also included some appendices to complete the understanding of this thesis. Specifically, Appendix A details the benchmarks used to conduct the experiments performed. Finally, we also believed that an example of how to use the TLS runtime library manually could be helpful, therefore Appendix B explains how to parallelize an application speculatively using the software described in Chapter 3.

Chapter 2

State of the art

Thread-Level Speculation (TLS) is a promising technique that allows the parallel execution of sequential code without relying on a prior, compile-time dependence analysis. Since first approaches, in which loops were optimistically executed and if any dependence violations arise they were sequentially re-executed, there have been several advances. For example, current techniques apply a wide range of improvements such as value prediction, or specific data structures. In addition, takes advantage of most available devices like GPUs, or Intel Xeon Phi coprocessors. Firstly, this chapter introduces the technique and presents a taxonomy of TLS solutions. After that, it summarizes and puts into perspective the most relevant advances in this field regarding the software based solutions.

2.1 Introduction

Thread-Level Speculation (TLS), also called Speculative Parallelization (SP), or even Optimistic Parallelization, is a runtime technique that executes in parallel fragments of code that were originally intended to run sequentially. Instead of relying on compile-time analysis to identify independent parts of sequential code that can be run concurrently, TLS techniques optimistically assume that these parts can be executed in parallel by different threads. To ensure correctness, speculative threads should detect whether they have consumed a datum that was subsequently updated by a predecessor thread, that is, a thread executing an earlier part of the code, according to sequential semantics. Such situations, called *dependence violations*, should be detected and rectified by hardware or software mechanisms, or a combination of both, to keep sequential semantics. If a dependence violation is detected, a corrective action will take place, typically discarding the results calculated by the thread that has consumed the incorrect value, and restarting it to be fed with the updated datum.

Although implementations can be based on either hardware or software, in the following lines we will just describe in-depth the software branch since our work are fully based on software. Precursors and hardware-based approaches are just introduced.

In this chapter we review the literature related to Thread-Level Speculation techniques, presenting a taxonomy that helps to better understand each proposed solution in its context. The chapter is organized as follows. Section 2.2 presents a global view of the problem, including a description of sources of speculation in the code, together with the main design choices that may arise while designing a TLS solution. Section 2.3 examines the first solutions that served as a base for the development of TLS systems. Section 2.4 briefly gives some notions about hardware-based approaches, where additional hardware is added to support speculation. Section 2.5 shows software-based proposals, which do not require additional hardware to monitor the parallel execution, at the cost of a certain performance loss. Section 2.6 describes other works that take advantage of TLS capabilities for different purposes. Section 2.7 cites some studies that have pointed out the theoretical and practical limits of the TLS paradigm. Finally, Section 2.8 concludes this chapter.

2.2 Sources of TLS and design choices

In [244], an accurate summary of Thread-Level Speculation techniques is given by Torrellas, including a detailed description of the two main issues that any TLS system should solve: How to buffer and manage speculative states, and how to detect and handle dependence violations. His analysis makes any effort to reproduce a summary of TLS characteristics here meaningless. Instead, we will briefly discuss where are the main sources of speculation, and which are the most important design choices that have to be faced to set up a TLS system.

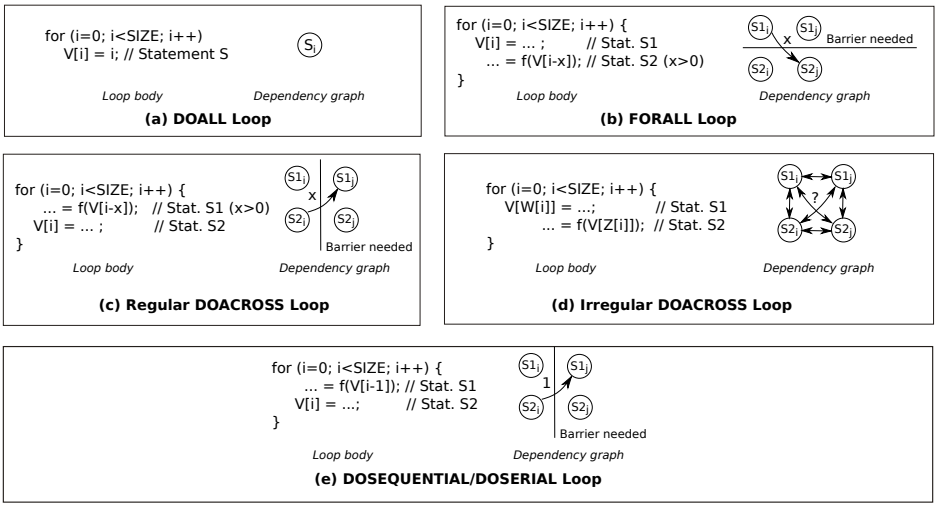


Figure 2.1: Different types of loops according to the presence of data dependences. The label in each edge represents the dependence distance.

2.2.1 Loops as a source of speculation

Due to how easy it is to distribute work among threads, loops are the most important source for TLS. The synthesis of loop-based speculation written by Rauchwerger [217], who was also a pioneer in the field, accurately reflects the importance of loops as a source of speculation. We will first briefly describe how data processed in one iteration may interact with calculations in different iterations, a situation known as data dependence.

There are three basic types of data dependences among two fragments of code, namely *true*, *anti*, and *output* dependences. In the following examples, let S_i and S_j be two statements, where S_i should be executed earlier than S_j according to sequential semantics.

- *True dependence:* Statement S_i writes into a location that is later read by S_j.
- *Anti dependence:* Statement S_i reads a location that is later written by S_j.
- *Output dependence:* Both statements S_i and S_j writes into the same location.

These definitions can be used to create a taxonomy of loops, according to the presence of data dependences among their iterations. One of the first taxonomies was proposed by Polychronopoulos and Kuck [205]. They proposed a mechanism for scheduling nested loops of parallel programs taking into account communication between processors and reducing shared memory accesses. Also, this work classified loops into three different types: *doall*, *forall*, and *doacross*.

- *Doall loops:* Loops that do not present any dependence among their iterations. Therefore, all iterations can be processed in parallel with no further checking [238]. Figure 2.1(a) shows an example of this loop. Most of current compilers can parallelize this kind of loops automatically.

- *Forall loops:* Loop whose iterations may present true dependences: Values produced by one iteration may be used in a subsequent iteration. An example is depicted in Fig. 2.1(b). All iterations of a *forall* loop can be executed simultaneously if and only if all the statements that produce the value (S1 in the figure) have finished *before* the execution of any statement that consumes the value (S2 in the figure). If this behavior cannot be guaranteed, a synchronization mechanism is needed.

- *Doacross loops:* Loops whose iterations may present *anti dependences*: Values consumed may be overwritten later. [154] divides *doacross* loops into three categories:

  - *Regular doacross loops:* Loops whose anti dependences among iterations are dominated by a constant value x. Figure 2.1(c) shows an example. These loops can be parallelized by ensuring that the execution of the iterations involved in the dependence follows sequential semantics. If the value of x is known at compile time, compilers are usually able to produce a parallel version of the loop.

  - *Irregular doacross loops:* Loops whose dependences among iterations are not known at compile time. Figure 2.1(d) shows an example. These loops are commonly called "irregular loops", and in general they cannot be parallelized safely at compile time.

  - *Dosequential or doserial loops:* A special type of *regular doacross* whose iterations depend on the previous one (that is, loops that have a dependence distance of one). Figure 2.1(e) shows an example. These loops have no parallelism at the iteration level.

Compile-time techniques can be used to generate parallel versions of *doall*, *forall* and, when the dependence distance is known at compile time, *regular doacross* loops. Since TLS is a runtime technique, it can use the available information in all of the described loops, including *irregular doacross* loops. However, TLS will likely be slower than a compile-time parallelization, if the latter can be applied. With respect to *dosequential* loops, a TLS system will also guarantees that the parallel execution will be correct, at the cost of squashing and restarting iterations continuously to follow sequential semantics, thus degrading performance. The main application of TLS is in the parallel execution of *irregular doacross* loops when the total number of dependences that appear at runtime is low.

2.2.2 Drawbacks of TLS

Although TLS can extract parallelism even from *irregular doacross loops*, it will likely be slower than a compile-time parallelization, if the latter can be applied. Sources of overhead in TLS include the cost associated to thread squash and restart due to data dependence violations, speculative buffer overflows, load imbalance due to data locality issues, thread dispatch and commit, and inter-thread communications [73].

TLS overheads may not only lead to lower performance in terms of execution time, but also to a greater energy consumption. This issue appears in software solutions, due to the energy cost associated to the execution of additional instructions to guarantee that sequential semantics are followed, and to the wasted work carried out by squashed threads. Energy inefficiencies also appears in hardware approaches, due to the need of additional hardware structures in the cache hierarchy for data versioning, dependence checking, and its associated bus traffic [220]. We will return to this problem in Section 2.6.3.

2.2.3 A first classification of TLS techniques

According to [143, 177], there are three types of speculation techniques: (1) control speculation; (2) data dependence speculation; and (3) data values speculation. These types are not disjoint, and their basis can be combined to achieve better results.

Control speculation

Control speculation applies speculation to loops that include conditional sentences. Execution paths of each iteration are detected, mapping them to different threads. [4, 130, 253] combined control speculation with branch prediction. Puiggali et *al.* [211] tried to predict the outcome of conditional branches without the need to know all the variables implied in the condition.

Data dependence speculation

Data dependence speculation is a technique suitable for the parallel execution of loops that may lead to inter-thread memory dependences. Load operations from speculative variables (that is, variables whose use may lead to a dependence violation) usually return the most recent value for that variable, while speculative store operations search for the use of outdated values in those threads, executing subsequent iterations according to sequential semantics. Many researchers have contributed to this solution, including [29, 49, 94, 177, 218, 241].

Data value speculation

Data value speculation techniques, also known as *value prediction techniques*, predict at runtime the result of instructions before their execution. This approach is based on the idea that an accurate prediction may avoid a squash. For example, the work by Raman et *al.* [215]

describes a prediction-based TLS software that predicted values of the following iterations without specifying the iteration where a value would be taken from. The main disadvantage of these proposals is that, in general, for loops with irregular memory accesses and complex control flow, this solution does not obtain good predictions. Other works that use predictors are [4, 52, 55, 84, 96, 167, 208, 233, 235, 239].

2.2.4 Design choices overview

To be speculatively executed, the original code should be instrumented at compile or runtime to handle different operations, such as loading and storing of speculative data, performing commit operations if the speculative execution succeeds, and discarding incorrect work if it does not.

The main design choices that should be faced in a TLS system are described by Yiapanis et *al.* in [267]. To implement a TLS system, a number of decisions should be taken:

Metadata management

TLS approaches should manage some information in order to detect whether conflicts have occurred. Thus, each thread should know both what memory addresses have been used, what operations have been done, and which thread has done each operation. All this information is collectively known as *metadata* [267], and its management has two goals: Preserving the information related to variables at risk of suffering violations, such as which thread has loaded, stored, or is locking a certain variable; and maintaining references about operations done by each thread, specifically, recording the variables loaded or written.

Version Management

When executing several consecutive fragments of sequential code in parallel, each thread usually maintains a version copy of the data structure that is accessed speculatively. This solution allows changes to this data to be performed locally, only storing these changes to a permanent place if the speculative execution of this thread proves successful. To do so, TLS systems require some additional storage to maintain the intermediate copies of each thread. There are two ways of managing these data:

- *Lazy Version Management*. In this case, a local copy of the exposed data is individually stored and managed. Therefore, when a load or store operation is performed, only the local version is changed. When a conflict is detected, only local versions of threads in conflict have to be discarded, instead of modifying the reference version in memory. Most approaches are based on lazy versions.

- The other approach, *Eager Version Management*, requires fewer resources, because the reference version in memory is modified. An additional buffer (called *undo log* in

the literature) records old values and is used to restore original data in the case of a dependence violation.

Regarding version management, Garzaran et *al.* [100, 101] proposed a taxonomy to classify speculative systems according to the way of buffering the speculative versions of variables. They took into account the isolation of speculative task states in each processor, and how the state produced by tasks is merged with the main memory.

Conflict Detection

Dependence violations can be checked with either a lazy or an eager approach: *Lazy Conflict Detection* avoids the need to check for conflicts on every access, by delaying this task to a later stage before the commit operation. A more strict approach, called *Eager Conflict Detection*, looks for conflicts on every access. This design avoids performance losses produced by later checks. However, the time devoted to checking is much higher.

Scheduling of iterations

To speculatively parallelize a loop, it should be partitioned into chunks of iterations to be assigned to different threads. The simplest solution is to use chunks of fixed size [155]. The particular size chosen is an important design decision. The use of smaller chunks will reduce squashing costs, at the cost of a higher scheduling overhead. On the other hand, bigger chunks will lead to higher speedups if no dependences arise, but they will increase the cost of thread squashing and may lead to load imbalance.

To mitigate these problems, variable chunk size strategies originally designed to achieve load balancing in parallel computations, such as [120, 205], can also be used in speculative execution. Regarding the particular context of TLS, Llanos et *al.* [174] proposed a variable chunk size for the speculative execution of randomized incremental algorithms, an important class of problems where the probability of a dependence violation decreases as execution proceeds. Their work uses smaller chunks for the first iterations, where randomized incremental algorithms present more dependence violations, then gradually increases the chunk size to reduce scheduling overheads, and finally reduces the size of the chunks again to achieve a better load balancing.

The use of chunk sizes that follows a predefined distribution, however, may not be the best solution. Speculative parallelization poses a more complex scheduling challenge than traditional parallelization, because, for irregular applications, both the number and the particular distribution of dependence violations are unknown before the loop is executed. Therefore, the idea of changing the chunk size at runtime if the number of squashes exceeds a certain threshold, or to augment it if no dependences arise, makes sense [84, 134, 172]. Recently, Estebanez et *al.* [83] proposed a method, called Moody Scheduling, that makes use of both the number of re-executions of the last chunks of iterations and their tendency (increasing, decreasing, stable) to figure out an appropriate chunk size for the following chunk to be scheduled.

Chapter 5 contains more details about scheduling policies in TLS, and an in-depth description of [83].

Squashing alternatives

If a dependence violation is produced, the offending thread should be discarded. The mechanism chosen to do so is a design decision that severely affects performance. Some approaches just discard the threads that have consumed the wrong value, and others discard the offending thread and all its successors. This leads to the following solution space, as described by Garcia et *al.* [97]:

- *Stops parallel execution*: First solutions, such as [218], simply discard the speculation when a dependence violation was produced, and then restart the loop sequentially. These solutions only benefit loops that were indeed parallel.

- *Inclusive squashing*: This approach stops and restarts the first thread that manages the wrong value, together with all its successors. Due to its simplicity of implementation, this is the most used solution (see [38, 49, 52, 208]), although it may discard potentially useful work carried out by a successor that has not consumed polluted data.

- *Exclusive squashing*: Only offending threads and those successors that have consumed *any* value generated by them are discarded and restarted. Li et *al.* [167] tried to implement this ideas in hardware. Colohan et *al.* [57] also used this kind of squashing mechanism in the context of databases (where restarting a thread leads to big performance losses), and used sub-threads to check for squashed threads. Tian et *al.* [240] also proposed a solution that does not discard all the produced values, only a small part of them. Also, Garcia et *al.* [97, 98] developed a software-only version of this idea, with the help of a list that stores which threads have consumed a value for a particular predecessor.

- *Perfect squashing*: Discards offending threads and those successors that have consumed *the outdated value*. This is the approach that leads to fewer squashes. However, to keep track of the definition and use of each particular datum, an in-depth analysis should be performed. This operation seems to be too costly. For example, [4] proposed a specific table to store dependences, while [225] used a table that saved all intermediate values. Nevertheless, [243] addressed this problem and concluded that this squash mechanism is not profitable.

The following section describes the ideas that led to modern TLS techniques.

2.3 Precursors

One of the first approaches centered on the parallelization of loops that may present dependence violations was the one proposed by Knight [149]. With the functional languages in mind, specifically the Multi-Lisp approach, Halstead [111] introduced a hardware approach that allowed speculation through the use of two different caches, one dedicated to storing those values loaded from memory, and the other used to hold those values produced by the processor whose accuracy was not confirmed yet. Midkiff and Padua [182] described a solution to synchronize the concurrent execution of singly-nested loops, while Zhu and Yew [274] described an algorithm to handle all types of loops. Aiken and Nicolau [3] described another scheduling algorithm, that analyzed loops and obtained the optimal, dependence-free distribution. In those years, Saltz et *al.* [21] performed research to extract some parallelism of *Doconsider* loops (a kind of regular *Doacross* loops where iterations could be rearranged), in order to preserve dependence semantics, and parallelize as many iterations as possible. They developed a compiler plugin that divided iterations into subsets of iterations that depend on each other, so as to execute several independent subsets at the same time. Although this paper was focused on programs whose dependences are known at compile time, it also mentioned codes not schedulable at *start-time* [183, 184], which are codes whose dependences could only be extracted during their execution. Krothapalli and Sadayappan [153] explored a solution to remove anti and output dependences. For that purpose, they performed a reference analysis, storing multiple copies of suspicious variables used in the loop. Later, [154] proposed a dynamic scheduler based on synchronism, that allowed *doacross* loops to be addressed with complex inter-iteration dependences. Afterwards, Wolf and Lam [260] used matrices to transform and parallelize loops in a general way, including nested loops.

The idea of the use of a dynamic *inspector-executor* model appeared at that time. With this approach, an inspector loop checks for dependences in a preliminary phase, and if no dependences arise, a second phase executes the loop in parallel. Saltz et *al.* [228] introduced this method in order to parallelize loops, showing that this technique allowed a significant performance improvement in loops with a big number of operations, where inspector phase time was not significant compared to the executor phase. Leung and Zahorjan [165] also worked in this solution. However, none of these approaches parallelize loops with output dependences. Chen et *al.* [43] developed a software solution that reduced delays between processor communications and allowed the parallelization of loops with output dependences. They reused some results during the execution, allowing the overlap of dependence iterations and the sharing of some information between inspector and executor phases.

2.4 Hardware-based approaches

Several hardware implementations have been developed to support TLS, mainly through the addition of auxiliary registers to manage speculation. Since the work performed in this Ph.D. thesis is fully made by software, this section will only cite a brief part of the existent solutions.

There are mainly two ways to implement TLS on hardware (HTLS): Developing a chip from scratch, or customizing an existing chip. On the one hand the most relevant solutions from scratch according to the author's criteria are mainly the Multiscalar architecture [94, 105, 233], the Trace processor [224, 225], Oplinger et *al.*' architecture [194, 195], and STAMPede [234, 236].

On the other hand, the customization of an existing chip led to the Simultaneous multithreading paradigm [247, 248], mainly supported by the Speculative multithreaded processor [177, 246] and I-ACOMA [151, 152], and also to the solutions based on Chip Multiprocessor. The main hardware speculative solutions included in the latter are Hydra chip multiprocessor [112, 113, 192] and Atlas Chip-Multiprocessor [55, 56].

2.5 Software-based approaches

Software-based TLS systems implement techniques to guarantee the coherence of the optimistic parallel execution on conventional processors, without the need for dedicated functional units. Research in this field has been centered on reducing, whenever possible, the overheads in execution times due to the need to ensure consistency by software.

As we will see, first proposals usually executed the loop in parallel, and if a dependence violation was produced, the work already carried out was discarded, and the loop was re-executed sequentially. More recent approaches perform partial commits, in order to take advantage of the work carried out before a dependency violation appears, and thus try to minimize the number of squashed threads to those that have actually consumed a polluted value.

Again, we will follow a historical perspective to describe the research in this field. We will first center our attention on those solutions where programmers should explicitly invoke runtime library functions and/or compiler support to manage speculative execution. Then, we will move to solutions that are based on higher-level programming abstractions. We will finish this discussion with some proposals related to TLS behavior, and a brief review of some works that mixed TLS with other techniques.

2.5.1 Solutions relying on compile-time and runtime support

First approaches required programmers to use different methods to explicitly invoke TLS mechanisms. The most representative ones are described below.

LRPD test

We can place the origins of Software TLS (STLS) in the work carried out by Rauchwerger and Padua [216, 218], with their research in the parallelism of *doall* loops. They proposed the use of a test called LRPD to support the speculative parallelization of loops with some backtracking capabilities. This proposal re-executed the loop serially if the runtime test failed. The proposal worked as follows: The target loop was firstly transformed through privatization and reduction parallelization, and then it was speculatively executed as a *doall* loop. After that, a fully-parallel data dependence test was applied to ensure that the loop had no cross-iteration dependence. If the test failed, the loop was sequentially re-executed. Otherwise, the parallel execution of the loop was considered successful. Dang et *al.* [61] developed a technique to extract the maximum available parallelism for loops that were known to present some dependences. This solution presented an evolution of the LRPD test, called Recursive LRPD (R-LRPD). The basic idea was to transform a partially-parallel loop into a sequence of fully-parallel loops. At each stage, this proposal speculatively executed all remaining iterations in parallel and the RLRPD test was applied to detect the potential dependences.

Based on static analysis

Gupta and Nim in [108] proposed a set of new runtime tests for speculative parallelization of loops that defied parallelization methods based solely on static analysis. They presented a more efficient method for speculative array privatization that did not require the computation to be rolled back when a particular variable was not found to be privatizable. They also presented a technique that allowed the early detection of loop-carried dependences, and another that detected parallelization hazards immediately after they were produced.

Software versions of hardware solutions

Rundberg and Stenström [227] applied many of the ideas of hardware-based speculative architectures in software. First, name dependences were solved by dynamically renaming data at run time. Second, the overhead of restoring the system state after a misspeculation was greatly reduced by reducing the amount of states to commit, and by supporting parallel implementations of the commit phase. Third, some anti data dependence violations were avoided by supporting lazy forwarding without the need to enforce synchronizations between a pair of conflicting threads. Fourth, true data dependence violations were detected when they happened, which reduces the cost of misspeculations. To do so, each instruction on

speculative data was augmented with a checking code that detects data dependence violations dynamically. Finally, it committed data following sequential semantics.

Cintra and Llanos [48, 49] developed a different scheme based on an aggressive sliding window. It checks for data dependence violations on every speculative stores, while avoiding synchronization whenever possible. The sliding window used consisted of an array of slots which store the status of each running thread, and pointers to their own version of the speculative data. Commits were carried out in order from the non-speculative thread. Each time a commit operation was finished, the sliding window advanced one position, allowing a new, most-speculative thread to start. More recently, [81] improved this solution with a different implementation that supported the speculative access to dynamic data structures and support for the use of pointer arithmetic (see Chapter 3). This solution used hash tables to reduce the time needed to find the most up-to-date version of a datum, a problem also described in [240].

Based on master/slave paradigm

Zilles and Sohi [275] introduced the Master/Slave speculative parallelism, a new kind of speculation whose basics were the use of a master thread and some slaves that performed the task assigned by their master. The main idea of this technique was to divide the program into tasks that would be carried out by the slaves, while the master thread predicted the values that would be produced by each task and continued with the execution of the code without waiting for their results. This approximation needed to check all the values produced by slaves after the execution of a task with respect to the values predicted by the master. If both were equal, the master's prediction had been successful, on the other hand, a misspeculation had been detected. In this case, the work incorrectly carried out by the master and all slaves since the last checkpoint needed to be discarded and re-executed.

Automatic thread extraction

[197] proposed an automatic approach for thread extraction. The system, called DSWP, exploited the fine-grained pipeline parallelism of many applications to extract long-running, concurrently executing threads. Their results showed significant improvements when executing these applications on a dual-core CMP.

Complementing compile-time techniques for auto-parallelization

[245] proposed the use of profile-driven parallelism detection to augment the number of loops that may considered safe to parallelize, relying on the user for final approval. This work also uses machine-learning techniques to take better mapping decisions for different target architectures.

Other solutions: SpLIP, MiniTLS, and Later

Oancea et *al.* [190] developed *SpLIP*, a speculative tool centered on decreasing overheads of speculative operations of previous approaches, implementing non-locking operations where was possible, and used a hash function to improve the location of version copies.

Yiapanis et *al.* [267] introduced a new structure that reduced memory overheads of classical approaches based on the idea of mapping every user-accessed address into an array of integers using a hash function. The authors implemented this compact data structure in two approaches, namely *MiniTLS* and *Later*. The main characteristic of *MiniTLS* was that threads updated memory locations in-place, and also that all operations followed fast and optimistic design patterns. This approach required rollback mechanisms because speculative threads modified values directly, possibly producing errors that needed to be handled. This solution is similar to SpLIP, so both were compared in this work. *Later* followed a different design, implementing a lazy version management of values, together with pessimistic design patterns in its operations. The structure used was a bit different, but it was based on the same operations and patterns. This approach also introduced a combination of inspector-executor techniques (described in Sect. 2.3) and the LRPD test (described in Sect. 2.5.1), implementing the new solution upon them.

TLS compiler and runtime for distributed systems

[146] present an automatic speculative DOALL parallelization system, composed of a parallelizing compiler and a speculative runtime for clusters that minimizes the overheads due to validations. Other STLS runtime solutions for distributed enviroments are covered in Section 2.5.5.

TLS for web applications

Martinsen, Grahn and Isberg [179] used a speculative mechanism in the context of web browsing. To do so, they implemented their software by means of the Squirrelfish JavaScript environment, that enabled the parallel execution of Javascript functions. They modified Squirrelfish interpreter to enable each instance of the interpreter to be executed as a thread, while executing as many instances as functions. The used variables were maintained in a special vector that showed modified values to detect dependence violations. The use of TLS in this context allowed these authors to achieve noticeable speedups.

Apollo

Jimborean et *al.* [133, 134, 135] introduced a TLS framework specially designed to speculatively execute nested loops. To do so, the authors used features of the polyhedral model to dynamically transform code into a more optimized version that led to higher speedups. First, a compiler [136] generated skeletons that were the basis of executions, due to their ability to produce different code versions that could be selected at runtime. Then, a dynamic part

was responsible for (a) building interpolating functions, (b) performing dynamic dependence analysis and transformation selection, (c) instantiating the parallel skeleton code, and (d) guiding the execution. The execution was based on profiling the code several times during the execution in order to choose the polyhedral transformations that could better speed up the execution. The detection of errors was done at three levels: Basic scalars, memory accesses, and loop bounds. This framework led the authors to parallelize some benchmarks that had not been parallelized before due to dependence management hurdles.

2.5.2 Solutions relying on programming abstractions

Our second set of software-based solutions eased the use of TLS by offering new, higher-level abstraction layers.

FastTrack

Kelsey et *al.* [144] developed a system called *FastTrack*, that proposed performing TLS with the help of unsafe optimizations of sequential code. Specifically, their programming interface allowed users to suggest faster implementations based on partial knowledge of a program and its usage. They divided code into two branches, the fast track and the normal track, and programmers could change between both tracks when needed. Their implementation included both compile-time and runtime support. A compiler insert function calls to ensure that the fast track produced the same result as the sequential execution. To do so, the authors limited the use of global and heap data, and the insertion of extra variables to support stack data. The runtime support checked program correctness through the comparison of states at the end of the tracks. If both results were similar, results were supposed to be correct. Otherwise, the fast track results were discarded. In this system, one processor was reserved to run the fast track, and the rest to the execution of normal tracks.

The Copy-or-Discard model

Tian et *al.* proposed the Copy-or-Discard (CorD) execution model [241, 242], in which the execution of parallel threads were separately managed by a non-speculative one. Speculative threads read values of the non-speculative thread and performed their computation. After that, speculative threads were committed in order. Then, results were checked by a non-speculative thread so as to preserve the semantics of the sequential order. The commit operation was performed by the non-speculative thread through the CorD mechanism, which checked whether results were correct. In this case, results were copied to the non-speculative data. Otherwise, they were discarded at no additional cost. The speculative execution was performed in three sections: Prologue, speculative section, and epilogue. The prologue contained those instructions that could not be speculatively executed, i.e., input statements and loop index update statements. The speculative section contained the parts that were not expected to suffer a dependence violation. Finally, the epilogue was intended to manage

output data. A buffer was used to maintain a copy of the output just in case the code contained any output instruction in the speculative part, to be finally processed in the epilogue.

CorD and dynamic memory The CorD approach did not give support to those applications whose speculative variables were dynamically allocated, so [240] enhanced CorD to be used with programs that had such dynamic data structures. The main problem of this approach was data traversing, because a dynamic structure could change their size during the execution. Pointers imposed another problem, since a speculative copy of a dynamic structure might have a pointer with an address to a non-speculative copy. In order to solve these problems, they proposed using a mapping table that translated addresses among speculative and non-speculative threads. They also included optimizations in the treatment of linked structures. Finally, [239] used a value predictor to improve the parallelization of programs with frequent and predictable cross-iteration dependences.

Reducing misspeculations [243] later tried to further reduce misspeculations. They proposed an approach intended to reuse almost all the correct calculations performed by a thread whose iterations had suffered dependence violations, instead of discarding all this information, as most approaches did. To do so, they used a partial speculative space in addition to the primary speculative space of each thread. This new space maintained the first read values of a speculative variable. If a misspeculation was found, only the successor spaces of the offending space were discarded. This approach led to better performance and to a reduction in the number of dependence violations, due to lower recovery times.

TLS based on the use of compile-time directives

Bhowmik and Franklin [26] described a compiler framework for TLS that allowed the parallelization of all instructions of a code, instead of only those that compose a loop. This feature specially benefited non-numerical applications with complex instructions. Codes were initially analyzed and profiled to produce a control flow graph. It ws then used to produce partitions that could be executed by multiple threads. Chen et *al.* [45] also developed a compiler that focused on providing a quantitative analysis of codes with complex dependences. Their aim was to give probabilities about the possible flows of the code, and detect if a squash was likely to be produced.

Mitosis It is a compiler framework developed by Quinones et *al.* [212] which had several thread units capable of independently executing different instructions and of storing speculative values. TLS execution started when the non-speculative thread found what they called the *spawning point*, where a new speculative thread was launched. Once a new speculative thread had been launched, it predicted the possible values, and began its execution at the end of the spawning point. Meanwhile, the non-speculative thread continued its execution. If no errors were produced, speculative threads were committed, otherwise, they were discarded.

The choice of these *spawning points* was a key part of the work. To do so, marks were chosen with the use of a synthetic trace. It selected the most suitable parts of codes to be speculatively executed regarding some requirements, such as the amount of workload of routines with respect to the total, or possible misspeculations.

Spice C SpiceC was an approach proposed by Feng et *al.* [91]. SpiceC implemented a number of directives that, when added to sequential code, eased parallel programming. Programmers did not need to be particularly careful about communications or dependences, because this model supported *doall*, *doacross*, pipelining and speculative parallelism. This solution also supported dynamic structures and pointer addresses. SpiceC threads had their own private space for data. A shared global space was used to store shared data. Threads' first accesses were referred to shared space and loaded to each local space, where following accesses were redirected to. When threads ended their executions, they checked for misspeculations, and committed their data to the shared space if they were correct. Directives were similar to OpenMP's [59], so sequential programs only needed a few additional directives: A directive to suggest what kind of parallelism would be used, and another to mark where commit operations had to take place.

[92] extended SpiceC with some additional directives to support I/O operations within parallel loops. To the best of our knowledge, this was the first approach that addressed the parallelization of this kind of codes through TLS. The main idea behind this research was to break the cross-iteration dependences caused by I/O operations modifying the original code. To parallelize input operations, this approach calculated file pointers before entering the loop to be used in each iteration. File pointer copies were created on demand by the iterations that used them. Regarding output operations, they required the use of some additional buffers, in order to store intermediate outputs produced by each thread. Each output value was stored in the corresponding thread buffer and flushed at the end of each iteration following sequential semantics. [93] also augmented SpiceC directives to parallelize loops with dynamically-linked data structures. This work tried to manage different data partitions of loops using the same code, addressing the problem of codes where multiple threads managed several data partitions.

ATLaS Aldea et *al.* [5, 8] developed a GCC plugin so as to add loop-based TLS support to OpenMP. This solution consisted of the development of a new OpenMP clause to be used in `for` loops called *speculative*, which allowed programmers to declare all variables whose reads or writes may lead to dependence violations (see Chapter 4 for more details). The internals of the runtime library that managed dependence violations were described in [81] (see Chapter 3 for more information).

The Galois model

Kulkarni et *al.* [161, 162] introduced *Galois*, a system that supported complex pointer-based sets of elements in optimistic parallelism. They were centered on the benchmarks that should get a subset of points from a big set, in order to obtain a solution to the problem. To do so, they defined two Java iterators which traversed a set of elements, thus allowing the concurrent execution of many iterations in a transparent way (note that in this system iterations could be understood as the search of an element within the set). There were different iterators for ordered and non-ordered sets. The consistency of data was implemented using locks. Moreover, to allow recovery from misspeculations, all operations had their corresponding inverse methods. With this purpose, an undo log was defined for each iteration. In order to manage all iterations, this solution defined a commit pool that contained data such as the state of iterations, or the position of the log. It controlled the entire execution, deciding how iterations were assigned and committed, conflicts were solved, etc.

Efficiency improvement through data partitioning After that, [160] introduced some improvements that increased the efficiency of Galois. This work implemented a method to perform data partitioning in data in such a way that all elements of a set were first mapped to an abstract domain, and then transformed again to physical cores. The abstract domain could be composed by data partitions (note that their number had to be higher than the number of existing cores, in order to assign more than one data partition to each core). This method allowed the authors to implement a way to "steal" work from an overloaded core by an idle core, achieving more efficiency in the execution. It also allowed cores that support multithreading to execute more than one partition at the same time.

Scheduling [158] addressed the problem of scheduling, developing an additional framework to Galois. Although iterations could be executed in any order within their baseline scheduling policy, this work showed the inefficiencies associated to this behavior, and proposed an improvement based on clustering (select a cluster of iterations), labelling (assign the selected clusters to cores), and ordering (order of the clusters to be executed) of iterations. Scheduling strategies for irregular applications in TLS were also addressed by [138] with Galois. Their strategies went from "stealing" the work of overloaded processors by idle processors in the static assignment, to using a centralized place as a warehouse for the extracted partitions.

A profiler: ParaMeter [157] developed a tool to extract parallelism profiles from irregular applications. This method took abstract measures of the inherent parallelism of the different points of a code, showing instructions that could be executed concurrently. Although this tool has been used in the context of Galois, the authors affirmed that it is framework-independent. [156] also introduced a suite of benchmarks to test irregular applications with the use of TLS libraries, including those used in the mentioned papers.

Optimizing irregular applications [181] described three manual techniques to optimize irregular applications in order to improve their parallel execution. The first one was based on the idea of modifying codes in such a way that all read operations were done before any write operation. The second one, called "one-shot", was based on the detection of dependences before the execution. If none were detected, checks for them could be disabled, and code could be parallelized without locks. Finally, for those algorithms whose bottlenecks were located in the accesses to data sets (appropriate for the benchmarks tested by them, described in [156]), they developed the "iteration coalescing" optimization. This was based on removing the correspondence between iterations and activities. So, if an iteration produced a value, it was processed before its publication in the working set. Later, Prountzos et *al.* [210] completed this work by automatizing the manual techniques described. They addressed again the overhead problems that emerged from optimistic parallelization, specifically, those related to conflict checking and undo actions. The center of this research was to reduce locks and rollbacks of the shared objects, using some inferred properties. In 2011, [159] analyzed whether the order used to launch methods affected execution times.

SEED

An adaptive approach for speculative loop execution that handled nested loops was recently proposed by Gao et *al.* [96]. They developed and implemented SEED, a tool that consisted of an adaptive dynamic scheduling of iterations based on a cost-benefit analysis, and a selection of the most suitable loops to be benefited by the use of TLS. This tool was composed by two phases, one related to compilation time, and the other related to runtime. In the compiler phase, loops were selected, threads were exposed in order to be later created, and the resulting code was optimized using precomputation and software value prediction to reduce misspeculations. At runtime, the basic TLS operations, such as thread spawning, dependence violations detection, and squashes, were carried out, together with the use of adaptive scheduling techniques.

2.5.3 Other proposals

Finite-State Machine in TLS

Zhao et *al.* [272, 273] introduced the use of probabilistic analysis into the design of speculation schemes. In particular, they focused on applications that were based on Finite-State Machines. The authors affirmed that this type of applications had the most prevalent dependences of all the programs. They developed a probabilistic model to formulate the relationship between speculative executions and the properties of the target computation and inputs. Based on that formulation, they proposed two model-based speculation schemes that automatically customized themselves with the best configurations for a given Finite-State Machine and its inputs. [271] presented a set of techniques to remove the need of offline training to collect

probabilistic properties that help to reduce the probability of misspeculations. Instead, their techniques allowed probabilistic analysis to be performed on-the-fly.

HVD-TLS

Fan et *al.* in [84] developed a software-based speculative framework that improved classical TLS mechanisms by the development of new techniques to improve value prediction, value checking, dynamic task partition, and scheduling. Predictions performed were done using several predictors based on the original value of the variables in conflict. Such predictions used a predictor table that also maintained the number of correct predictions. Values were checked by the main thread to prevent committing unmodified values, a situation repeated many times according to the authors. Also, this system allowed different levels of granularity to be assigned at runtime, following a linear scheme or a heuristic scheme, where the system monitored the execution and changed the granularity accordingly.

MUTLS

Cao and Verbrugge [37] introduced a mixed model to fork threads in both in-order and out-of-order ways in a TLS library. Their work was based on the use of the LLVM compiler framework [164] that allowed multiple source languages and target architectures through the use of an intermediate representation. MUTLS allowed threads to fork and join in different parts of the code, and also implemented barriers to avoid some rollbacks. Threads were managed by four modules: one dedicated to maintaining the status of speculative threads, two dedicated to manage local and global variables of speculative threads, and the last one used to managing other modules and interact with the LLVM speculator pass. Speculation was mainly centered on functions whose speculative versions were available. Other functions interacted with the library and implemented synchronism.

TLS to decompress: SDM

Jang, Kim and Lee in [131] described a TLS scheme specially designed to be applied to decompression algorithms. Their approach was centered on the application of prediction techniques based on partial decompression and pattern matching, to quickly identify block chunks that can be independently decompressed. The tool decompressed in parallel all the blocks identified.

2.5.4 TLS mixed with other techniques

Helper threads, run-ahead and multi path execution

Xekalakis, Ioannou and Cintra in [264] proposed a model that combined different techniques such as TLS, helper threads, and run-ahead execution, in order to dynamically choose at

runtime the most appropriate combination. The Helper threads technique is based on the extraction of small threads (also called slices) from the main thread to improve its efficiency, for example, by resolving highly-unpredictable branches and cache misses. Runahead execution was based on executing instructions in advance when a long latency operation was expected. The main difference of this technique with respect to helper threads was that the former did not require additional threads. The main idea behind this research was to start a TLS execution with some prefetched threads in a runahead mode (since these threads would be faster than the others), and predict cache misses to help TLS threads. In other words, these prefetched threads were essentially helper threads acting in a runahead mode to help the execution of main threads. Xekalakis and Cintra [263] later combined TLS with MultiPath execution, a technique consisting in executing the two branches of hard-to-predict branches. The main idea behind this approach was to enhance processors with multiple-context execution to enable a fast way to discard erroneous data of wrong branches. The execution had normal TLS and MultiPath modes, depending on the number of occurrences of hard-to-predict branches. The previous combinations were more detailed, extended and mixed in [265], where the authors described a system that applied TLS to loops. Other techniques were also proposed, such as the use of prefetched threads when delays were detected.

Continuous speculation

Zhang et *al.* [269] described continuous speculation, a technique whose main objective was to achieve full-occupancy of processors. For that purpose, they used speculation techniques to achieve the parallelization of large sequential codes. Their solution used a sliding window and a group classification to ensure the correct order of the tasks. To get information about the possibly parallel regions of a sequential code, they used BOP [68], a tool that analyzed the program behavior to parallelize it.

Software-based lock elision

Locks could also be used to guarantee sequential semantics of parallel programs. In this way, [213] proposed speculative lock elision, to execute those sections without conflicts in parallel. To do so, they automatically replaced locks by optimistic hardware transactions, checking that no errors were produced. If transactions failed, the system used the original lock. Some studies were conducted to evaluate Transactional Lock Elision (TLE) supports in different architectures, including [2, 34, 66].

Roy et *al.* [226] proposed a software version of the speculative lock elision proposed by Rajwar and Goodman [213] that was fully implemented in software. If a misspeculation was produced, the system executed the original lock. Synchronization and privatization were implemented through special instrumentation for objects and through signals between threads implemented inside the Linux kernel.

2.5.5 STLS on distributed-memory systems

There have been some efforts on applying TLS techniques on clusters of commodity servers. [148] present a runtime monitor called Distributed Software Multi-threaded Transactional Memory (DSMTX) that allows the application of pipeline parallelism, multi-threaded transactions and TLS on distributed-memory environments. [150] described dyDSM, a distributed-shared memory abstraction to process large dynamic graphs that provides support for exploiting speculative parallelism. The balance between communication and computation in graph-based applications is studied in [41], proposing a new runtime, called ABC^2, that dynamically modified the configuration of the underlying DSM.

2.5.6 STLS using GPUs

Nowadays, parallelism applied to GPUs is one of the most profitable research fields due to their large number of computer units. This characteristic makes them desirable to find ways to use TLS with these architectures. Liu et *al.* [168] discussed how TLS could be correctly used in the context of GPU computation. Meanwhile, Diamos and Yalamanchili [64] extended *Harmony*, a runtime for heterogeneous, many-core systems, to support speculation in GPUs. Samadi et *al.* [229] introduced *Paragon*, a solution that combined CPU and GPU executions to achieve the best performance. Feng et *al.* [89, 90] proposed a framework to run loops speculatively in GPUs. The main idea of their solution was to divide the tasks that should be carried out by a speculative runtime framework into five categories, and to assign some of them to CPUs and the others to GPUs. Scheduling, results committing and misspeculation recoveries were assigned to CPUs, while computation and misspeculation checks were carried out in GPUs. In a more recent approach, Zhang et *al.* [270] introduced a new library based on sliding windows that support TLS in GPUs. Classical solutions that were expected to have a better behavior with GPUs, such as hybrid dependence checking, and the use of a parallel commit scheme, were adapted by these authors to their software.

2.6 Other studies related to TLS

There are several works that used TLS for other purposes, such as improving manual parallelization, or performing module-level speculation. Other studies include how the energy consumed by TLS proposals could be reduced. In this section we will review some of them.

2.6.1 TLS as a help to manual parallelization

In order to avoid making speculative codes that might be slower than the original sequential ones, some researchers have proposed techniques to predict overheads of speculative parallelization. For example, the work developed by Dou and Cintra in [72, 73] contained a compiler

pass that can be used to estimate the overheads and the expected resulting performance gains, if any. Ding et *al.* [68] proposed a software-based TLS system to help in the manual parallelization of applications. The system required the programmer to mark "possibly parallel regions" (PPR) in the application to be parallelized. The system relied on a so-called "tournament" model, with different threads cooperating to execute the region speculatively, while an additional thread ran the same code sequentially. If a single dependence arose, speculation failed entirely and the sequential execution results were used instead. Ke et *al.* [139] improved that work with a system that relied on dependence hints provided by the programmer. This allowed explicit data communication between threads, thus reducing runtime dependence violations. Ioannou and Cintra [127] studied the problem of taking advantage of future many-core architectures by complementing parallel programming at a coarse-grain level with hardware TLS support to launch fine-grain implicit speculative threads. Other authors have focused on providing assistance to those programmers that extract TLS from the applications. For example, [6, 262] developed tools that made a static and/or dynamic profile of the codes, returning information that allowed a decision to be made about which loop would be the best candidate to be speculatively parallelized. There are also libraries that perform speculative parallelization directly, such as in [209], where Prabhu et *al.* developed some directives and operations to facilitate programmers to make their own speculative programs. Chen et *al.* [46] designed a dependence profiler to extract information from a code. Bhattacharyya [24] also developed a similar tool that studied the profitability of TLS with the use of profiling. More recently, [25] used polyhedral analysis to detect dependences of loops, stating that this analysis overcame the previous one.

2.6.2 Module-level speculation

Module-level speculation is the application of speculation in a module-based layer. Chen and Olukotun in [44] applied this technique to object-oriented Java programs. Warg and Stenstrom [257] compared the use of object-oriented and imperative languages in the context of Module-level parallelism, concluding that there were not significant differences between both approaches. Their experimental results showed that the use of this method in modules with a low computational load would adversely affect the performance of applications. [256] described a predictor that allowed the detection of whether a module had a low overhead. In this case, no threads were created to help him. Later, [258] developed another prediction technique to detect when misspeculations would occur, in order to avoid executions that were expected to be squashed. Their prediction algorithm was based on the analysis of the execution history. Predictions were managed using two strategies: Assigning this operation to the closest fork, in other words, the nearest thread created, or assigning it to the common predecessor, which was the thread that was speculating a previous module. Pickett and Verbrugge [203] addressed the requirements that Java language imposed to implement this approach, also analyzing their costs. They developed a design based on the application of Module-level speculation to support TLS at the level of Java bytecode. To do so, they

introduced two new bytecodes, to fork and join speculative threads, in order to ensure their correctness. They also gave an implementation of the mentioned design, called SableSpMT analysis framework (described in more depth in [201, 202]).

2.6.3 Energy consumption

Since their origins, TLS was claimed to be energy inefficient. [54] analyzed the energy consumption of some approaches based on the Trace processors [224, 225]. Renau et *al.* [219, 220] searched for the main sources of energy consumption in TLS, giving some advice for energy saving. With the same goal, Xekalakis et *al.* [266] proposed a power allocation scheme for TLS systems based on Dynamic Voltage and Frequency Scaling (DVFS) that took power from non-profitable threads that would need to be discarded and used it to speed up more useful ones. Li and Guo [166] proposed two algorithms also based on DVFS. They proposed both static and dynamic assignment algorithms, achieving significant reductions in energy consumption. However, in the work carried out by these authors, energy savings came at the cost of lower performances. This topic was also addressed by Luo et *al.* [176], which developed a system that analyzed speculative execution, and managed resources in a way that decreased the energy needed.

2.6.4 Benchmarks for TLS

Many of the aforementioned studies shared the same benchmarks to give experimental results. We could highlight *Standard Performance Evaluation Corporation (SPEC)* benchmarks [69, 116, 117, 200], a benchmark suite to measure computing performance. Other benchmark suites frequently used are *Olden* [223], a set of relatively small programs that perform a monolithic task with minimal user feedback; and *MiBench* [109], a set of programs to test embedded systems. *STAMP* [35] is a benchmark suite designed for transactional memory applications, that was also used by different TLS approaches. The *LLVM* compiler infrastructure [164] also offered some benchmarks. *Princeton Application Repository for Shared-Memory Computers (PARSEC)* [27] is another benchmark suite composed of multithreaded programs, and [156] described *Lonestar*, a suite of benchmarks of TLS specially developed to test the Galois system, described in Sect. 2.5.2.

2.7 Limits to TLS

A number of papers are mainly centered on the analysis of TLS performance and its limitations. Although Prabhu and Olukotun [207] affirmed that significant parallelism could be extracted using TLS in SPEC2000 applications, Kejariwal et *al.* [143] affirmed that it is very difficult to achieve a high level of performance through TLS with this benchmark. [142] performed an analysis of TLS using SPEC CPU2006 benchmarks and affirmed that the use

of TLS with these benchmarks did not lead to significant benefits, with just a 1% improvement. However, as [126] and [198] noted, the former study only considered parallelism at the innermost loop level, while the parallelization of outer loops would lead to speedups. Later, [141] briefly analyzed the performance of TLS at module-level (also called graph-level). They studied factors such as recursion, or I/O, which limits TLS applicability at this level. Also, [140] proposed an analytical model based on conditional probability to gauge the suitability of nested TLS.

[128, 129] analyzed loop-level parallelism in embedded applications with and without TLS, concluding that TLS is useful for extracting the most possible parallelism from this kind of programs. Finally, [23] studied the influence of input sets in dependences of some TLS benchmarks. To do so, he had proven 57 benchmarks of SPEC2006, PolyBench/C, BioBenchmark and NAS in the IBMs BlueGene/Q supercomputer. The author concluded that the input set did not noticeably change the dependence behavior in the loops of the benchmarks studied.

2.8 Conclusions

Thread-level speculation is an active field, thanks in part to the appealing idea that it may be possible to automatically extract the loop-level parallelism of sequential applications without a prior and costly dependence analysis. Most studies described in this work have shown that TLS techniques effectively lead to a speedup when used under certain conditions. However, this technique is highly sensitive to the actual number of dependence violations that appear at runtime. A second drawback of TLS techniques is their comparatively high costs in terms of energy consumption.

While more general TLS solutions are developed, speculative-based techniques will likely coexist with other solutions in the execution of irregular codes that are not analyzable by other means. Current trends are focused on: avoiding, as much as possible, dependences with better predictors; developing advanced squashing techniques; and the use of TLS in manycore systems such as GPUs or Intel Xeon Phi.

The work described in this chapter was presented in the following publication:

- Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'A survey on Thread-Level Speculation Techniques'. In: *ACM Computing Surveys (CSUR)*. Accepted for publication

Chapter 3

The ATLaS runtime system

We have developed a new TLS runtime library that supports the speculative execution of for loops. The library architecture follows the same design principles of the speculative parallelization library developed by Cintra and Llanos. This original library had the following limitations: (1) Loops to be parallelized should have a number of iterations known before its execution. (2) Loops to be parallelized must not work with pointer arithmetic. (3) Inside a target loop we cannot use dynamic memory. Our new thread-level speculative runtime library removes all these limitations. It allows to speculatively access variables of any data type, either by name or address. In addition manages the space needed for version copies on demand. In this chapter we will detail the general architecture of the TLS runtime library developed as well as some improvements implemented such as a hash-based structure which severely reduce the accesses involved in the main speculative operations.

3.1 Problem description

Software speculative schemes should allocate some additional memory in order to hold the information related to speculative executions. The use of this data is mandatory to enable recovery operations that could arise in an optimistic execution. In this context, memory needed could be allocated either dynamically, or statically, and the use of an approach instead of the other is a critical decision that directly influences in the overall memory used in a program.

One of the biggest challenges in software-based TLS is how to reduce the time needed (a) to get the most up-to-date value when reading speculative data, and (b) to search for a possible dependence violation when a thread writes on a speculative variable. The most common solution to maintain speculative data is allowing each thread to keep a version copy of all the speculative variables that have been locally accessed. Once a thread finishes the execution of its chunk of iterations, all changes in the speculative data are committed to main memory. Note that both operations described above imply traversing all the version copies maintained by other threads. In the first case, the search for an up-to-date value implies to traverse all the data being kept by all threads executing earlier chunks of iterations, namely predecessor threads. In the second case, the search implies to traverse all the speculative data being maintained by all successors.

Access to predecessor and successor copies of the data are in the critical path of any TLS system. The problem is even more difficult to solve if the TLS library allows to speculate over dynamic structures and/or pointer-based references.

In this chapter we address the problem of how to traverse speculative data efficiently in a software-based TLS library. To do so, we first review the internals of the speculative parallelization library of Cintra and Llanos [48, 49, 170, 171], highlighting some of its limitations. The new TLS runtime library proposed [75, 76, 78] supports the speculative execution of for loops with both dynamic and pointer-referenced speculative variables, handling dynamic memory and managing on demand the space needed for speculative variables in each thread. This TLS runtime library allows the parallelization of loops with variables of any data type, referencing these variables either by name or by address.

However, as we will see throughout the chapter, although this library effectively removes many constrains of Cintra and Llanos' solution, the strict adherence to the original architecture leads to unacceptable costs for speculative reads and writes.

We have improved the performance of the new library by proposing some enhancements specially designed to be used in TLS approaches. In particular, we describe how to dramatically decrease the number of memory accesses when searching for predecessor and successor versions of speculative data, while keeping the cost of local data storage in $O(1)$. Our experimental results with well-known benchmarks on a real system show that these optimizations lead to significant reductions in the number of accesses needed (by a factor of three orders of magnitude) comparing with a competitive baseline implementation that lacks this feature [76]. In addition, we have proposed additional solutions to further reduce

```
1 for (i=1; i<5; i++)
2 {
3   LocalVar1 = SV[x];
4   SV[x] = LocalVar2;
5 }
```

Listing 3.1: Example where appear a RAW dependence. This type of loop may cause dependence violations in parallel executions.

the memory allocation calls, needed to dynamically add new variables to the speculative structures that should be managed at runtime. The combined effect of all these improvements is an impressive increment in the speedups obtained.

The rest of this chapter is structured as follows: Section 3.2 describes the original TLS runtime library developed by Cintra and Llanos. Section 3.3 summarizes the limitations of this solution, in order to establish the requirements that the new library should solve. Section 3.4 details the data structures used by the TLS runtime library. Section 3.5 describes how the main speculative operations work. Section 3.6 introduces the bottlenecks of the TLS runtime library, and addresses some proposals in order to improve its performance. Mainly, we can highlight a hash-based data structure which can reduce the number of accesses required by speculative operations. Section 3.7 shows some experimental results in terms of performance measured in a real system. Finally, Section 3.8 concludes this chapter.

3.2 Cintra and Llanos' original solution

Our research framework is based on a new, pointer-based version of a software-based TLS library originally developed by Cintra and Llanos [48, 49, 170, 171]. This library was based on a set of functions that allows the speculative execution of a loop. It required programmers to add additional code so as to parallelize. In this section we will briefly describe the original approach by Cintra and Llanos, in order to understand its limitations. An in-depth explanation of this approach can be found in [48].

To better understand Cintra and Llanos' solution, let's see an example of the speculative execution of a loop with the mentioned tool. Our example is based on loop of the Listing 3.1 with four iterations, and the possible execution trace depicted in Figure 3.1 which supposes that our system has enough processors to execute a single iteration in each one. As can be seen, all operations follow sequential semantic until instant **t10**. Then, thread three modifies the value of the shared vector $SV[X]$ used at **t7** by thread four. Hence, results calculated so far by thread four are discarded.

Remark that, as commonly accepted, threads of the lesser iterations are called non-speculative thread, and threads of the highest iteration are called most-speculative (these

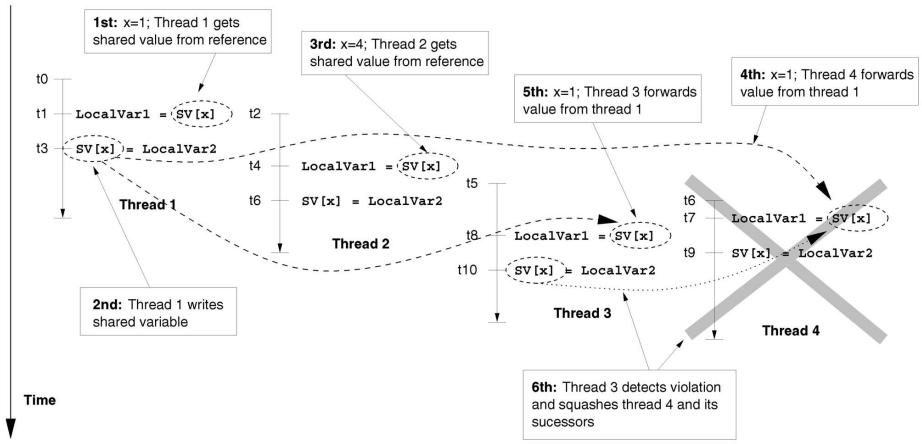


Figure 3.1: Speculative load operations search for the most updated version of $SV[X]$, consulting previous threads. Speculative store operations assign a value to the variable as well as check if any of the following threads have used a wrong value. If so, they are also responsible of discarding outdated threads.

concepts would be deeper exposed throughout this chapter). Thus, in the example drawn in Figure 3.1 the thread which executes iteration 1 is the non-speculative, whereas the thread which executes iteration 4 is the most-speculative.

Each thread managed its own version of shared data, so, if all threads commit their results with no order, then possible incoherences may appear. Therefore, each thread should commit its data following an order: from the least speculative thread to the most one.

3.2.1 Modifications in original source codes

All operations mentioned (loads, stores and commitments) required some modifications in original source codes:

- **Load operations** over speculative variables are replaced with a function that recovers the most recent value, in other words, the most updated value of the variable.
- **Store operations** over speculative variables are replaced with a function that stores the value and detects sporadic dependence violations.
- Each thread executes a **commitment of calculated data** at the end of the execution of its chunk of iterations (this operation is also responsible for assigning a new chunk of iterations to be executed).

Figure 3.2 gives an example of these operations.

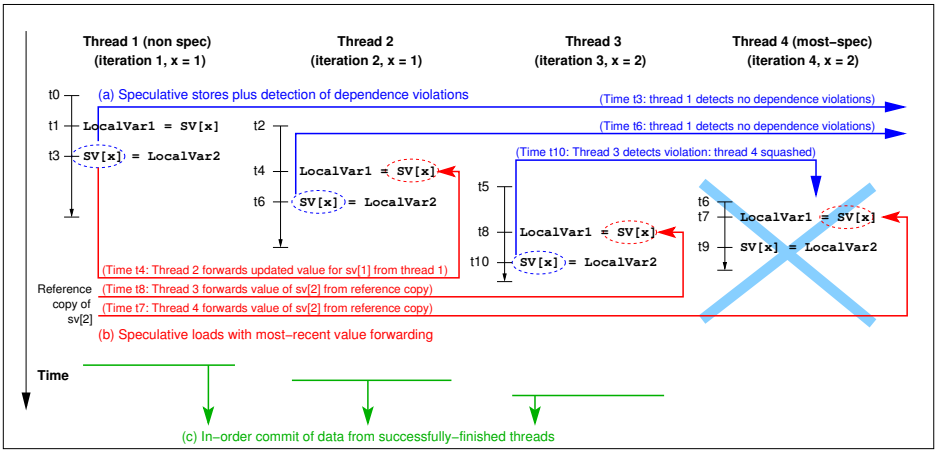


Figure 3.2: Example of speculative execution of a loop and summary of operations carried out by a runtime TLS library.

3.2.2 Classification of variables

In order to speculatively parallelize a loop, it is necessary to classify all variables (supposing a manual parallelization). Users should label variables, following OpenMP syntax, as private or shared. Variables in risk of suffering a dependence violation should be classified as shared. However, it is mandatory to modify all accesses to speculative variables in the target code with the corresponding functions.

3.2.3 Distribution of iterations

An additional change is required to parallelize a loop speculatively with this system. Instead of executing a loop with its original syntax, it is replaced by another loop with P iterations (being P the number of processors). In other words, if a loop has N iterations, it will be replaced by another with P. Then, the N iterations are divided into the processors. Note that the number of iterations given to each chunk has to be specified previously (static scheduling, see Chapter 5). Once no more chunks remain to be assigned, and all threads end their execution, the speculative execution will have ended.

3.2.4 Thread management

One of the main novelties of the TLS library made by Cintra and Llanos is the use of a sliding window mechanism in the management of threads. Thus, when the non-speculative thread finishes its execution, and a partial commit takes place, the thread executing the following chunk becomes the new, non-speculative thread. Then the sliding window advances, allowing

the execution of new chunks of iterations by other non/most-speculative threads. Further details of this original TLS library can be found in [48, 49, 76, 82, 170, 171]

The work by Cintra and Llanos was one of the first solutions in parallelize general applications speculatively. However, its practical use was constrained by several limitations. In the following section we will enumerate these limitations, in order to solve them.

3.3 Main limitations of Cintra and Llanos' solution

Although the TLS library developed by Cintra and Llanos has got several advantages, it requires to comply with the following premises:

- Their library requires that **all speculative variables were packed in a single, one-dimensional vector** before executing the speculative loop. Therefore, to use this TLS library, programmers should modify the original source code of applications introducing some extra lines so as to define the mentioned structure which stores all speculative variables. The use of this solution required threads to allocate memory for the entire vector, even if most of the positions of it were not used during the execution of the assigned chunk of iterations. So, in the case that M was the number of speculative variables used in a problem executed with T threads, and each variable need a byte, variables require $M \times (T + 1)$ bytes to be stored (because an additional space is required to save the persistent copy). Hence, this library can be only used with vectors or matrices (casted to vectors), and does not support more complex data structures.

- As a result of the previous requirement, all **speculative variables should share the same data type**. In fact, sharing a single, one-dimensional vector to preserve all variables implies that all of them should be from a single data type: *char*, *int*, *double*, etc.

- **Speculative variables can only be accessed by name** inside the loop (no references by addresses or pointers were allowed).

To remove these limitations is critical not only to extend the scope of applications that may be benefited from TLS techniques, but also to allow a compiler which automatically performs the transformations required. The following sections will describe our new TLS runtime, while the following chapter will deal with the compiler support. The entire package is called ATLaS framework.

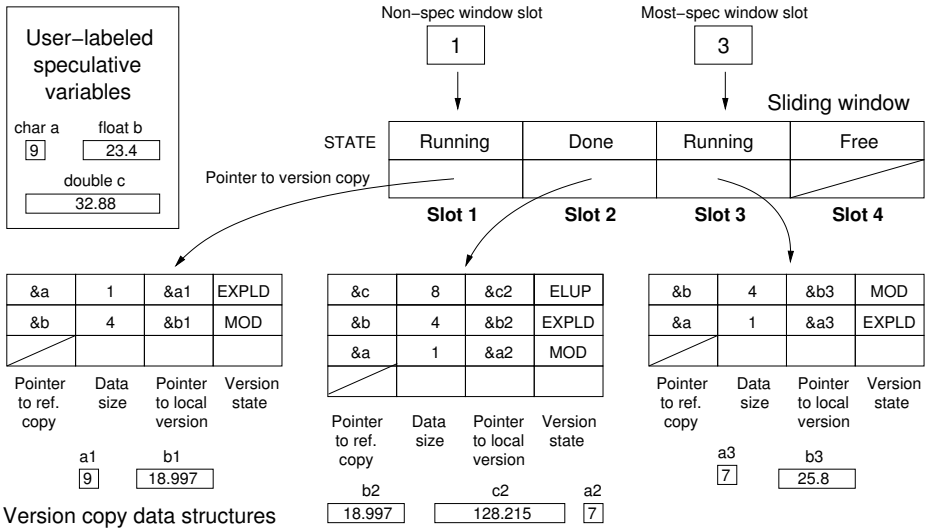


Figure 3.3: Data structures of our new speculative library.

3.4 Our new TLS library

The TLS runtime library developed overcame the issues of the Cintra and Llanos' proposal, by supporting both dynamic memory and pointer arithmetic. We will first describe the new data structures needed for this task.

3.4.1 Data structures

The data structures needed by the speculative library are depicted in Fig. 3.3. A matrix with W window slots (four in the figure) implements a sliding window that manages the runtime of the library. Each slot is responsible to manage the speculative execution of a particular set of iterations. Slots assigned to the non-speculative and the most-speculative threads are indicated by two variables, non-spec and most-spec. Each slot is composed of two fields, STATE with the state of the execution being carried out in each slot; and a pointer to maintain the position of speculative variables used by each slot in the execution.

Fig. 3.3 also depicts an example of the execution of a loop. The loop has been divided into three chunks of iterations, and it will be executed in parallel using three threads. It is really important to understand that there is not a fixed association between threads and slots. Whenever a thread is assigned a new chunk of iterations, the corresponding slot to work in, is specified as well. This allows to maintain an order relationship among the chunks being executed.

In the depicted example, thread working in slot 1 is executing the non-speculative chunk of iterations (as indicated by its RUNNING state). The following chunk has been already executed and its data has been left there to be committed after the non-spec chunk finishes (since it is in DONE state), while the last one, the most-speculative chunk launched so far, is also RUNNING. In other words, the thread in charge of the second chunk has already finished, whilst the non-spec and most-spec threads are working. If more chunks were pending, the following chunk will be assigned to the freed thread, starting its execution in slot 4. Slot 2 cannot be re-used yet, because the execution of the chunk 2 left changes to speculative variables that are yet to be committed. As we will see in Section 3.5.3, when the non-speculative thread working in slot 1 finishes, it will commit its results and the results stored in all subsequent DONE slots, since commits should be carried out in order. After that, in our example, the non-spec pointer will be advanced to slot 3 to reflect the new situation.

In addition to its STATE, each slot points out to a data structure that holds the version copies of the data being speculatively accessed. Figure 3.3 represents a loop with three speculative variables. At the given time, the thread executing the non-speculative chunk has speculatively accessed variables a and b. Each row of the version copy data structure keeps the information needed in order to manage the accesses to different speculative variables. The first column indicates the address of the original variable, known as the *reference copy*. The second one points out its data size. The third one shows the address of the local copy of this variable associated to this window slot. Finally, the fourth column indicates the state associated to this local copy. Once accessed by a thread, the version copies of the speculative data can be in three different states: *Exposed Loaded*, if the thread has forwarded its value from a predecessor or from the main copy; *Modified*, indicating that the thread has written to that variable without having consumed its original value; and *Exposed Loaded and Updated*, where a thread has first forwarded the value for a variable and has later modified it. The transition diagram for these states is shown in Fig. 3.4.

Fig. 3.3 represents a situation where the thread working in slot 1 has performed a speculative load from variable a (obtaining its value from the reference copy) and a speculative store to variable b. Regarding a, the figure shows that the thread working in slot 3 has forwarded its value. With respect to variable b, the information in the figure shows that b has been overwritten by threads working in both slots 1 and 3.

3.5 New speculative operations

3.5.1 Speculative reads

The use of the data structures described above requires the definition of new interfaces for speculative operations.

The new interface of `specload()` is as follows:

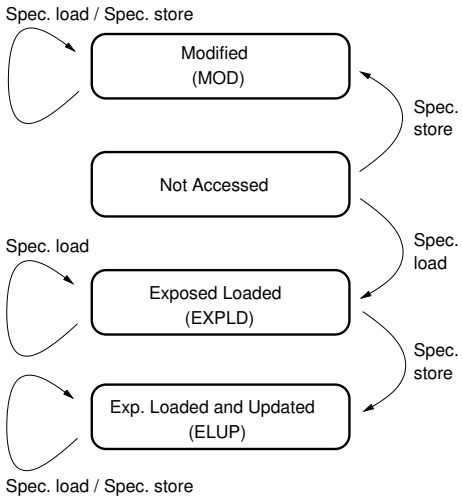


Figure 3.4: State transition diagram for speculative data.

specload(VOID addr, UINT size, UINT chunk_number, VOID* value)*

Arguments have the following meaning:

1. *addr* is the address of the speculative variable.

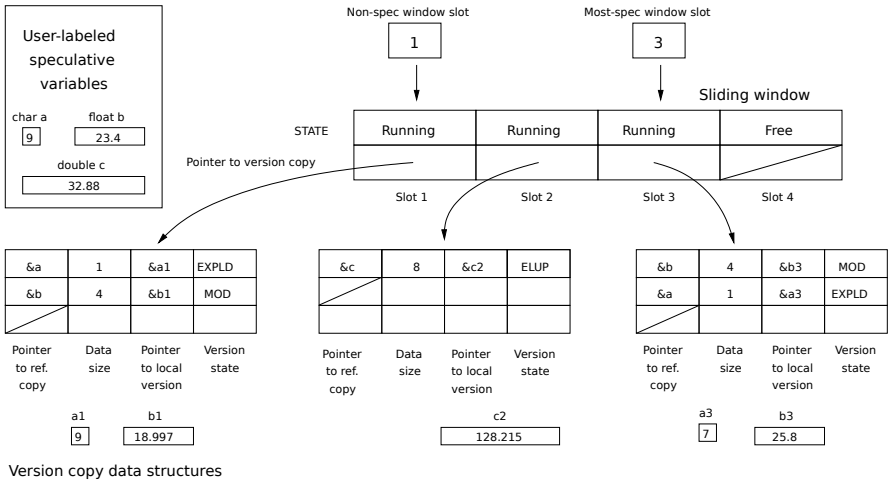
2. *size* is the size of the variable.

3. *chunk_number* is the number of the chunk being executed (needed to infer the slot being used).

4. *value* is a pointer to a place in which the datum requested will be stored.

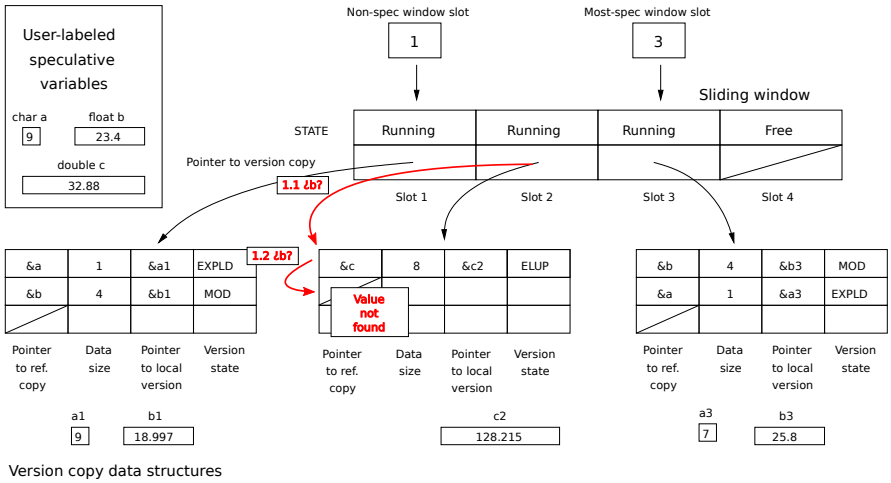
`specload()` should return the most up-to-date value available for the speculative variable. Figures 3.5 and 3.6 show how the speculative load works. Suppose that the thread working in slot 2 has only accessed to variable `c` so far (as is described in Figure 3.5(a)), and then it calls `specload(&b, sizeof(b), 2, &value)` to obtain a value for `b`. The sequence of events is the following:

1. Thread working in slot 2 scans its version copy data structure to check whether a value for `b` has been already stored there. As long as the only speculative variable accessed so far is `c`, this search produces no results (see Figure 3.5(b)).



Version copy data structures

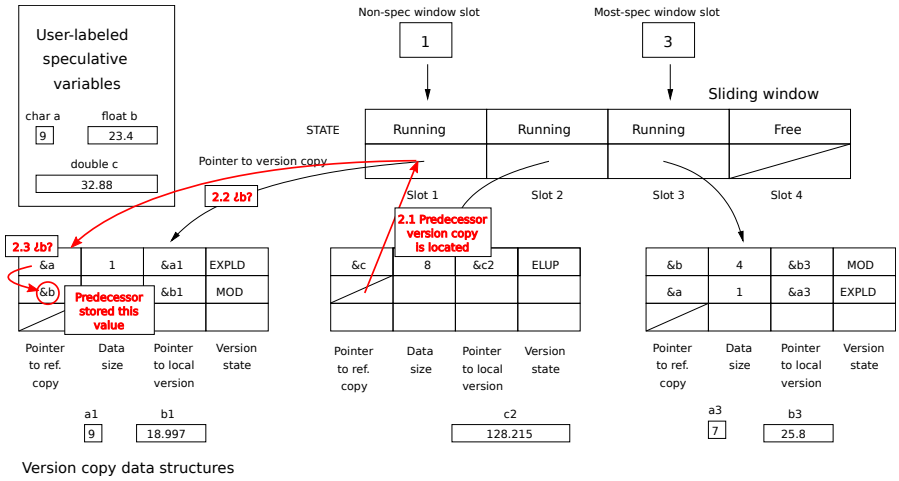
(a)



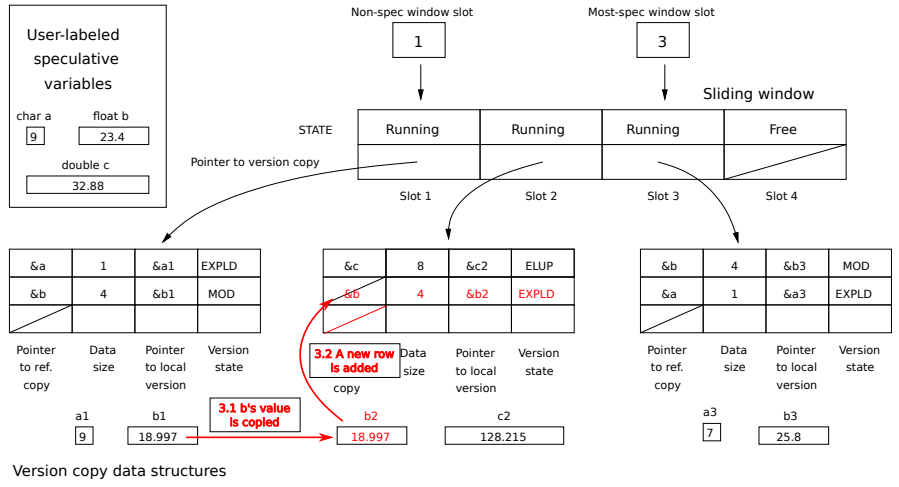
Version copy data structures

(b)

Figure 3.5: Speculative load example (1/2). (a) Initial values of the example. (b) The thread working in slot 2 scans its version copy to find the value.



(c)



(d)

Figure 3.6: Speculative load example (2/2). (c) The thread working in slot 2 goes to its predecessor version copy data structure and scans it in order to find a value for b. (d) After storing a copy of b's value, the thread working in slot 2 adds a new row to its version copy data structure.

2. Our thread goes to its predecessor version copy data structure and scans it in order to find a value for `b`. Its predecessor has stored a value for it, so our thread copies its value to a new location (see Figure 3.6(c)). Note that, if no value for `b` had been found there, our thread would have gone to its predecessor, until the non-speculative thread were found. If no predecessor had used the value, our thread would have gotten the value from the reference copy.

3. After storing a copy of `b`'s value, the thread working in slot 2 adds a new row to its version copy data structure, storing the address of `b`, its data size, the address of the version copy of `b` being managed by the thread, and the new state for this version copy, `EXPLD` (see Figure 3.6(d)).

The call to `specload()` finishes returning the value 18.997 in the address indicated by its fourth parameter.

Early Squashing

Load operations executed by this library will be optimized to improve the performance of applications. This optimization is called *Early Squashing* and consist on performing a squash operation before the end of the chunk of iterations. To understand this operation, let us suppose that an application performs a *specstore()* (store operation described at the following subsection). This operation checks the slots of the following threads in search for dependence violations. If a dependence violation is found, the slots will be changed to the `SQUASHED` state, but its execution continues, until the end of its iterations. On the contrary, if *earlySquash()* operation is introduced, *specload()* calls check the state of the thread involved. If it is `SQUASHED` the *specload()* call will end, avoiding unnecessary time losses since *earlySquash()* will trigger a "jump" to the part of the code where the thread ends the execution of its block of iterations.

3.5.2 Speculative stores

specstore(VOID addr, UINT size, UINT chunk_number, VOID* value)*

Arguments have the following meaning:

1. *addr* is the address of the speculative variable.

2. *size* is the size of the variable.

3. *chunk_number* is the number of the chunk being executed (needed to infer the slot being used).

4. *value* is the address of the variable which will be stored.

As it can be noticed, the interface of `specstore()` is almost the same as `specload()` except for the last parameter, which is a pointer to the value to be stored. Recall that `specstore()` should not only store the new value, but also check whether a successor has consumed an outdated value for it.

Figures 3.7, 3.8 and 3.9 show the sequence of events related to a speculative store. Suppose that the thread working in slot 2 executes `specstore(&a, sizeof(a), 2, &temp)`, where `temp` holds the value 7. The sequence of events is the following, taking into account that initial values are depicted in Figure 3.7(a):

A. The thread working in slot 2 searches for a local version copy of `a`. Then, the search produces no results since only copies of `c` and `b` are stored in its version copy data structure (see Figure 3.7(b)). If `a` had been found, this thread would have updated its status according to the state diagram of Figure 3.4, and it would have proceed to step D.

B. The thread working in slot 2 creates a local copy of `a`, storing value 7 on it (see Figure 3.8(c)).

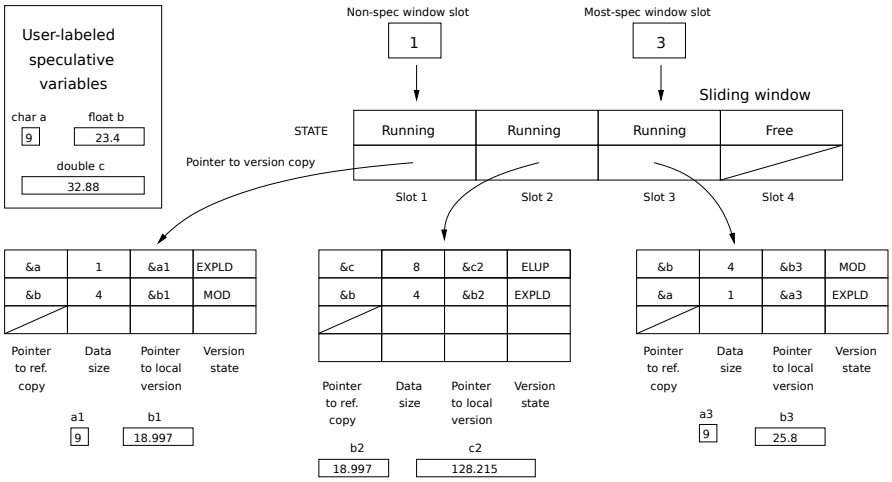
C. A new row is added to the version copy data structure, with a pointer to `a`, its size, the pointer to the local copy, and the status, that will be `MOD` in this case (see Figure 3.4). All these operations will be seen in Figure 3.8(c).

D. After storing the value locally, the thread working in slot 2 should check whether any successor has consumed an outdated value. To do so, our thread would scan (following the increasing order imposed by speculative behavior) for any successor slot that holds a copy of `a` in `EXPLD` or `ELUP` state. These states would indicate that the successor has used the value (see Figure 3.8(d)).

E. In our example, the search finds out that thread working in Slot 3 has consumed an incorrect value for `a` (see Figure 3.8(d)). If no dependence violation was detected, the call to `specstore()` would finish here.

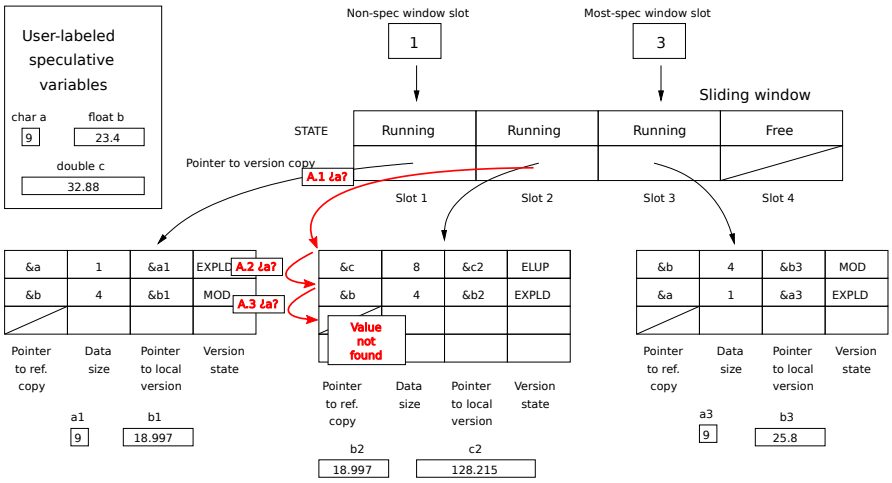
F. A dependence violation has been detected. The thread working in slot 3 should be squashed. To do so, the thread working in slot 2 changes the state of slot 3 from `RUNNING` to `SQUASHED` (see Figure 3.9(e)). Since all threads check their own state at the beginning of each `specload()` and `specstore()` call, the thread working in slot 3 will eventually detect that it has been squashed, and will execute a call to `commit_or_discard()` to be assigned a new chunk (possibly the same) and start the process again.

G. Finally, the thread working in slot 2 marks itself as the most-speculative thread, since data stored in association with slot 3 is no longer valid (see Figure 3.9(f)). The most-spec pointer will be advanced later by the thread that will receive the task of re-executing chunk 3.



Version copy data structures

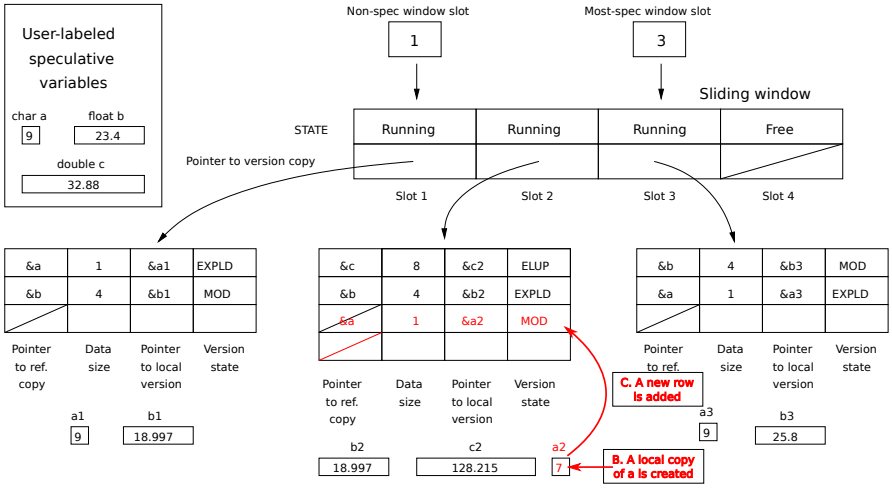
(a)



Version copy data structures

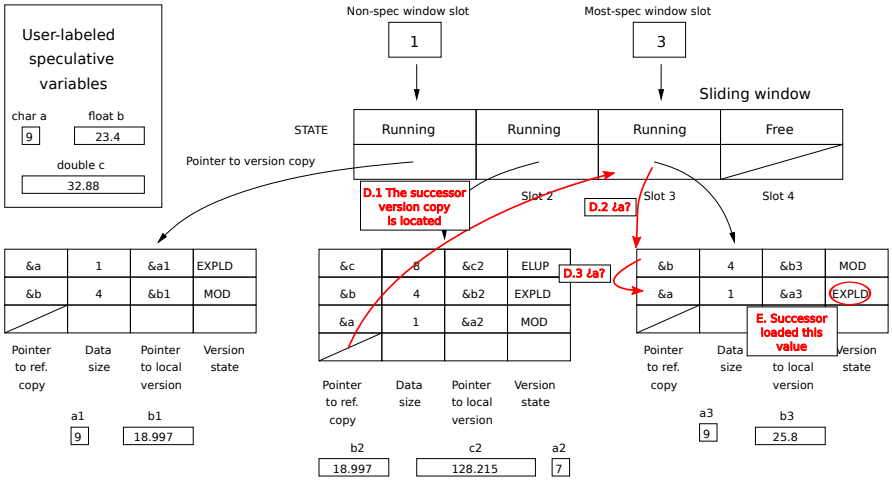
(b)

Figure 3.7: Speculative store example (1/3). (a) Initial values of the example. (b) The thread working in slot 2 scans its version copy to find the value.



Version copy data structures

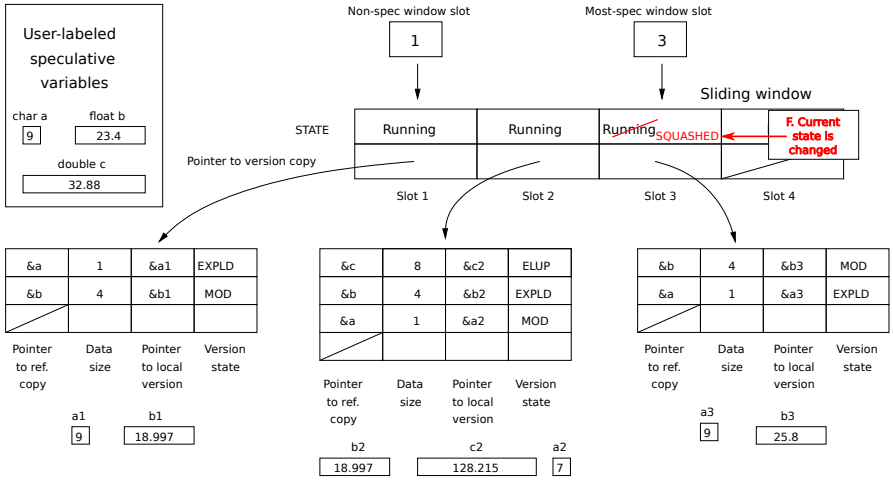
(c)



Version copy data structures

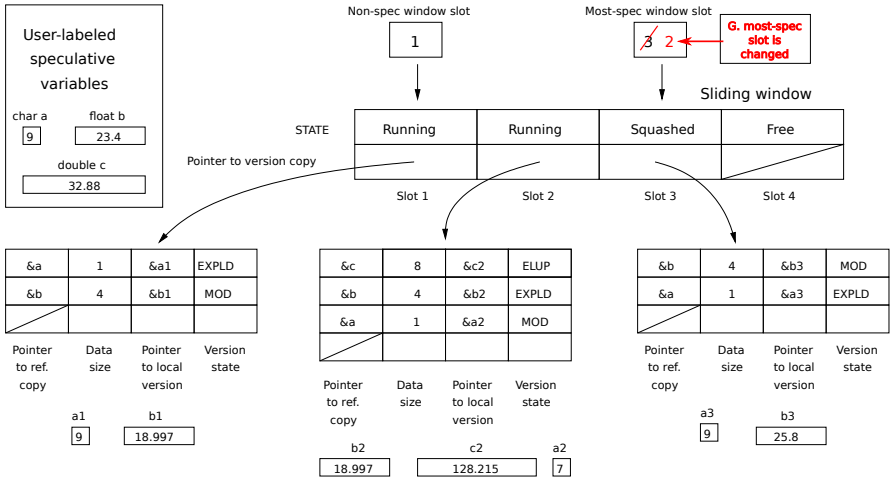
(d)

Figure 3.8: Speculative store example (2/3). (c) After creating a local copy of a, the thread working in slot 2 adds a new row to its version copy data structure. (d) The thread working in slot 2 should check whether any successor has consumed an outdated value. In our example, the search finds out that the thread working in Slot 3 has consumed an incorrect value for a.



Version copy data structures

(e)



Version copy data structures

(f)

Figure 3.9: Speculative store example (3/3). (e) The thread working in slot 3 should be squashed. To do so, the thread working in slot 2 changes the state of slot 3 from `RUNNING` to `SQUASHED`. (f) The thread working in slot 2 marks itself as the most-speculative thread, since data stored in association with slot 3 is not longer valid.

If, after these events, the thread working in slot 2 finishes its execution, while threads associated to slot 1 and 3 are still working, we will reach to the situation shown in (see Figure 3.3). Note that, at that moment, the thread working in slot 3 has already been re-started and it has forwarded the most up-to-date value for `a` (that is, 7) from slot 2.

3.5.3 Speculative commits

The partial commit operation is exclusively carried out by the non-speculative thread. Every time a thread executes `commit_or_discard()`, it first checks if it has not been squashed and if is the non-speculative. If the thread is speculative, the slot will be left to be committed by the non-spec thread.

As in the case of previous operations, let us examine an example case: Suppose that we are in the situation depicted in Figure 3.10(a), and the thread working in slot 2 finishes.

I At this time, the thread working in slot 2 should change its state in order to show other threads that its chunk of iterations have been executed, and should be committed (see Figure 3.10(b)). However, this operation could only be performed by the non-speculative thread.

II Afterwards, the non-spec thread working in slot 1 finishes, it therefore begins to commit all of its values. Operations performed to carry out this task are depicted in Figures 3.11(c) and 3.11(d). For example, `b` element should be committed, so it copies the content of `b1` into `b`.

III When no more elements are available, i.e., after committing the version copy data structure associated to slot 1, it changes its state to `FREE`. (see Figure 3.12(e)).

IV Then this thread checks if any successor thread has finished its execution. In our example the thread working in slot 2 has finished, so its elements must be committed (see Figure 3.12(f)).

V Elements of the slot 2 are committed following the order depicted in Figures 3.13(g), 3.13(h) and 3.14(i).

VI When no more elements are available, the state associated to the thread working in slot 2 is changed to `FREE`. (see Figure 3.14(j)).

VII Since the state of the thread working in the next slot is not `DONE`, commit operation could be seen as finished. But, we also have to take into account that the non-spec pointer should be advanced to this slot 3 (see Figure 3.15(k)).

After executing these operations we get to the situation depicted in Figure 3.15(l). In this way, version copies of slots committed are not entirely reseted until another chunk of iterations is assigned to them, and change their state to `RUNNING`.

It might be interesting to remark that each thread only writes on its local version copy data structure, and, as a result, no critical sections are needed to protect them. It is necessary just a single critical section to protect the sliding window data structure, in which threads can overwrite the state of another one.

3.5.4 Reduction operations

Some operations cannot be parallelized because of the own nature of their instructions. Nevertheless, there are some kind of operations which could be transformed so that we can be able to parallelize them. When one of these cases appear, it is said that the operation can be *reduced*. For example, sum and the calculation of the maximum and minimum operations can be reduced.

Sum reduction

Imagine that the loop to be parallelized contains the following sum operation.

$$matrix(i) = matrix(i) + value$$

This type of operation involves a sum operation for each iteration of the loop. So, if we wanted to parallelize it, too many dependence violations would be produced since iteration that modifies `matrix(i)` value needs the value of the previous iteration. In other words, iteration `J` needs to know the value of `matrix(i)` in the iteration `J-1` so as to update its value.

In order to solve this problem, it is possible to perform a simple sum reduction in the following way:

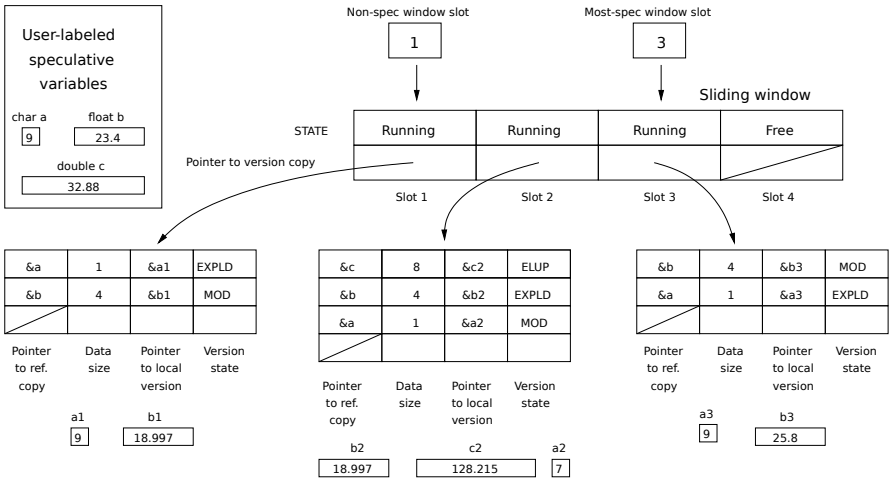
$$matrix(i) = matrix(i) + valor1 + valor2 + ... + valorK$$

In this way, instead of summing a quantity each iteration, all sums are concurrently performed so that dependence violations can be avoided. This operation is known as *specredadd()*:

$$matrix(i) = matrix(i) + value$$
$$\Downarrow$$
$$specredadd(VOID^* addr, UINT\ size, UINT\ chunk\_number, VOID^* value)$$

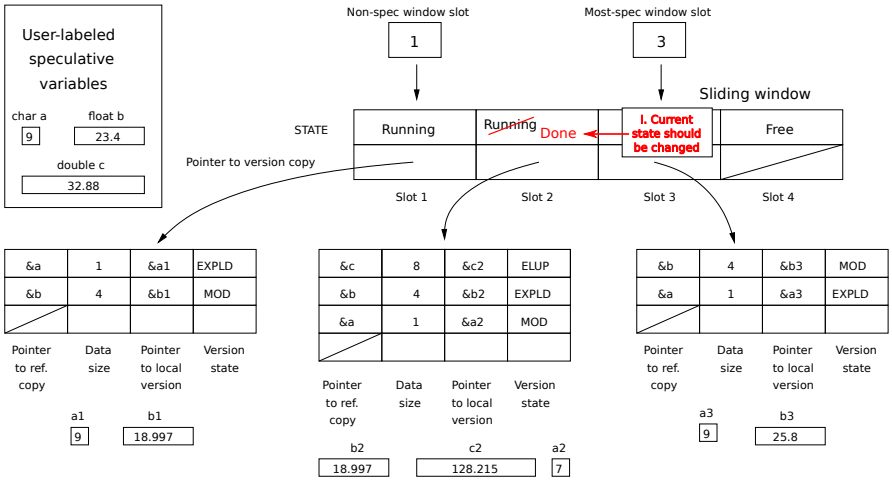
Arguments have the following meaning:

1. *addr* is the address of the speculative variable.
2. *size* is the size of the variable.
3. *chunk_number* is the number of the chunk being executed (needed to infer the slot being used).
4. *value* is the address of the variable which is going to be summed.



Version copy data structures

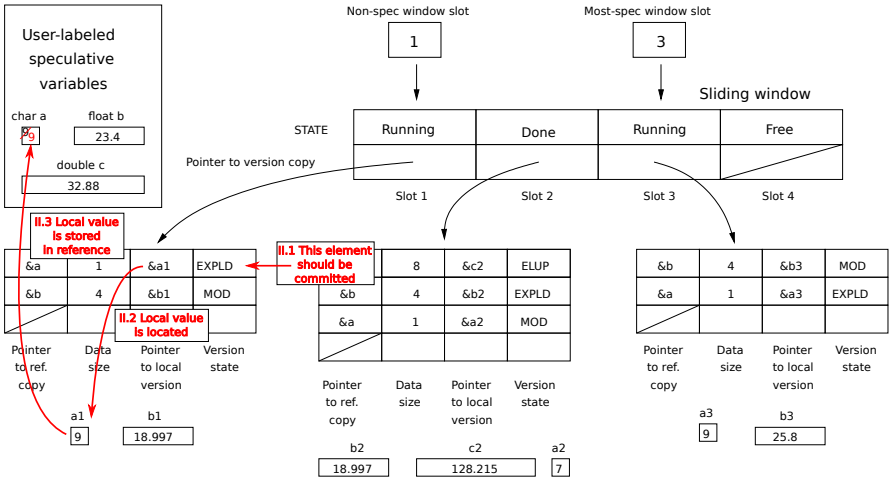
(a)



Version copy data structures

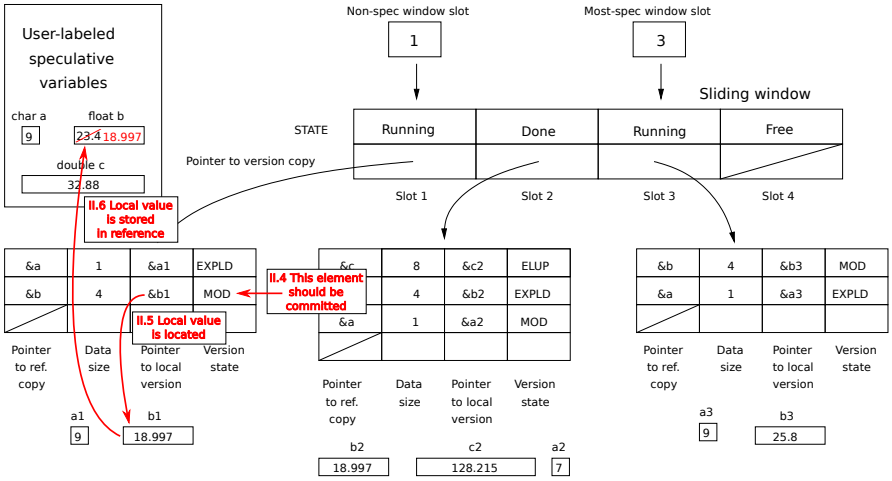
(b)

Figure 3.10: Speculative commit example (1/6). (a) Initial values of the example. (b) The thread working in slot 2 finishes its chunk of iterations.



Version copy data structures

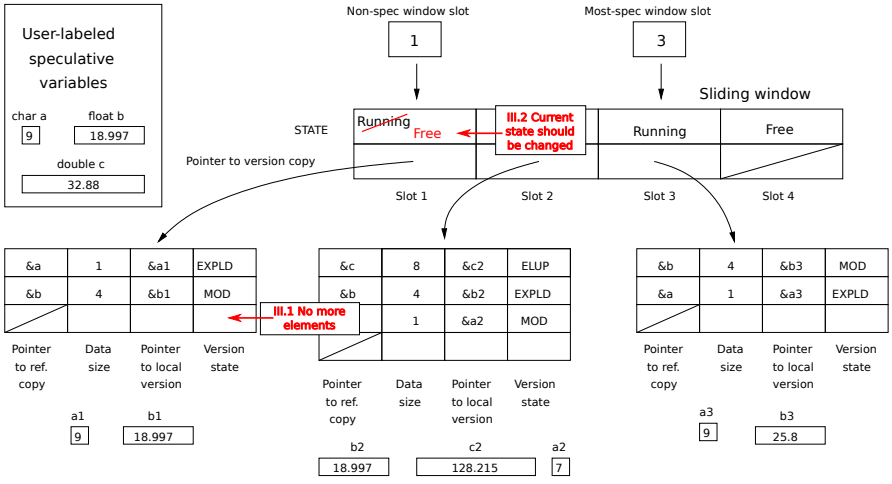
(c)



Version copy data structures

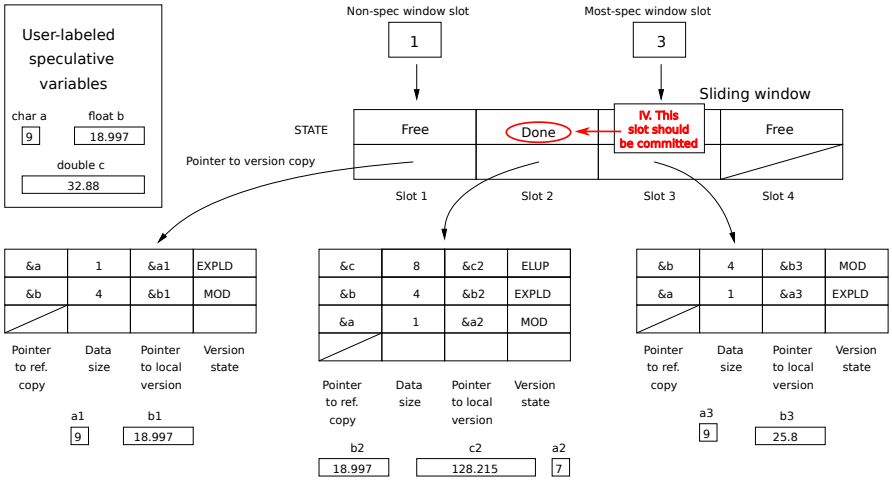
(d)

Figure 3.11: Speculative commit example (2/6). (c) The non-speculative thread finishes its execution so its elements start to be committed. (d) The next value of the non-speculative thread is committed.



Version copy data structures

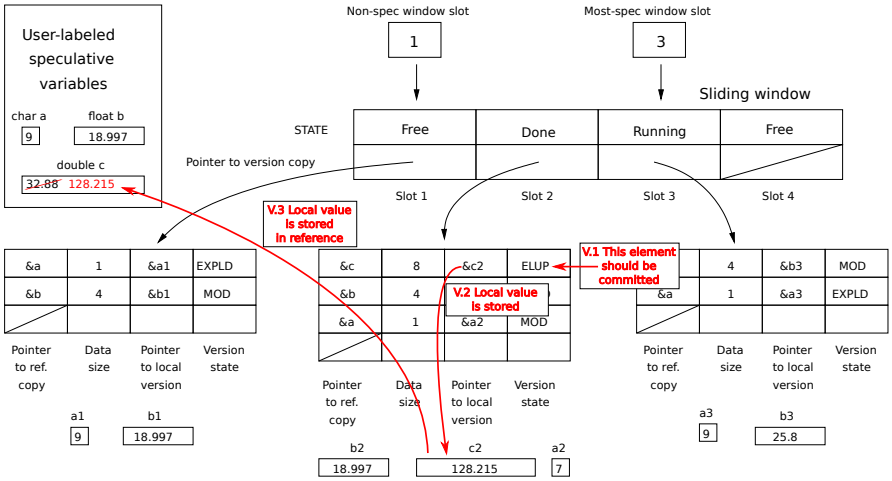
(e)



Version copy data structures

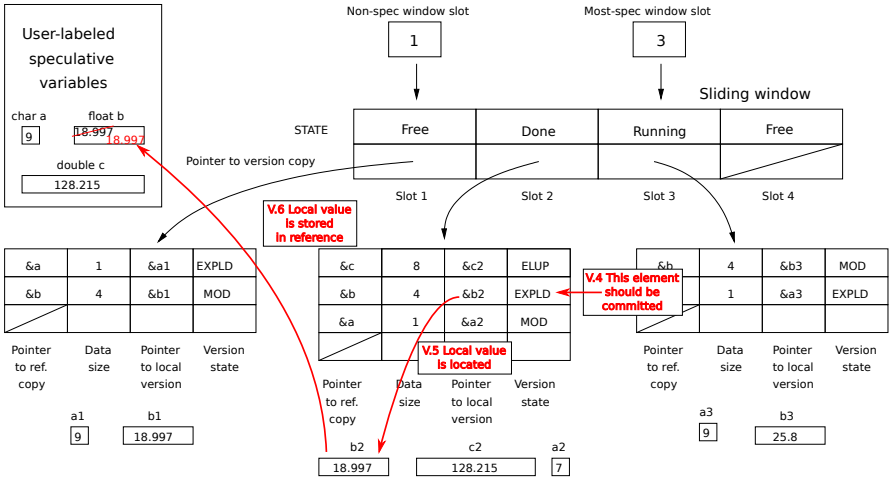
(f)

Figure 3.12: Speculative commit example (3/6). (e) No more elements are located in the version copy of the non-speculative slot, so the thread working in slot 1 changes its state from `RUNNING` to `FREE`. (f) After changing its own state, the thread working in slot non-speculative, checks if any slot of its successors could be committed, in other words, if their states are `DONE`.



Version copy data structures

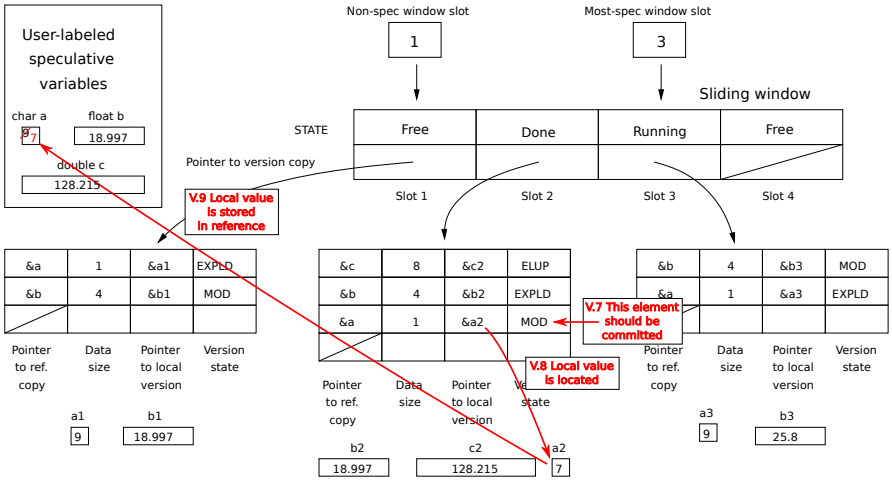
(g)



Version copy data structures

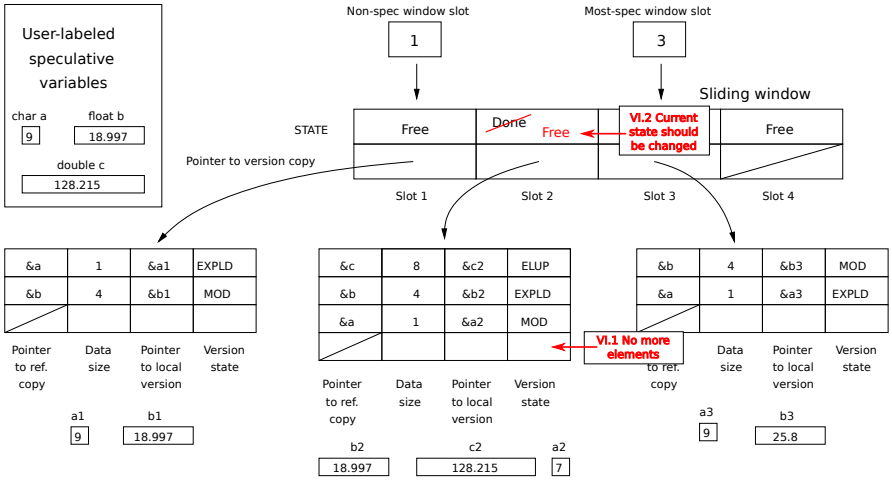
(h)

Figure 3.13: Speculative commit example (4/6). (g) The thread working in slot non-speculative starts to commit elements from slot 2 because its state was DONE. (h) The next element from slot 2 is committed.



Version copy data structures

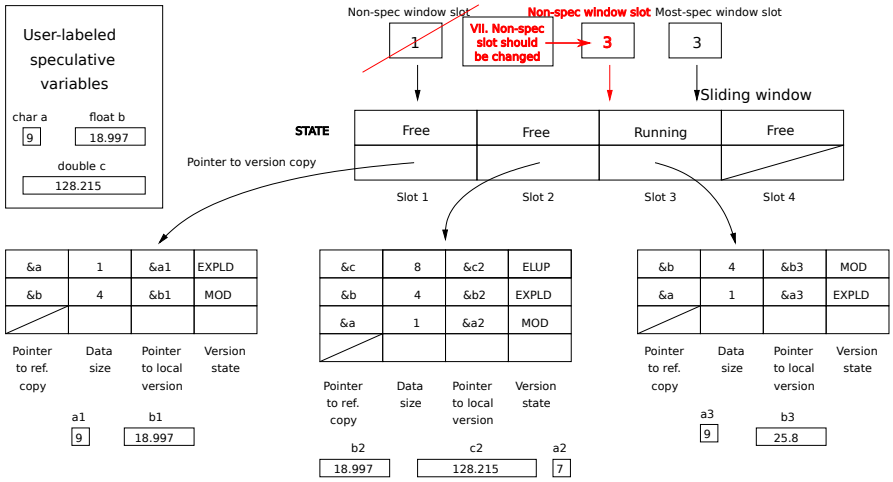
(i)



Version copy data structures

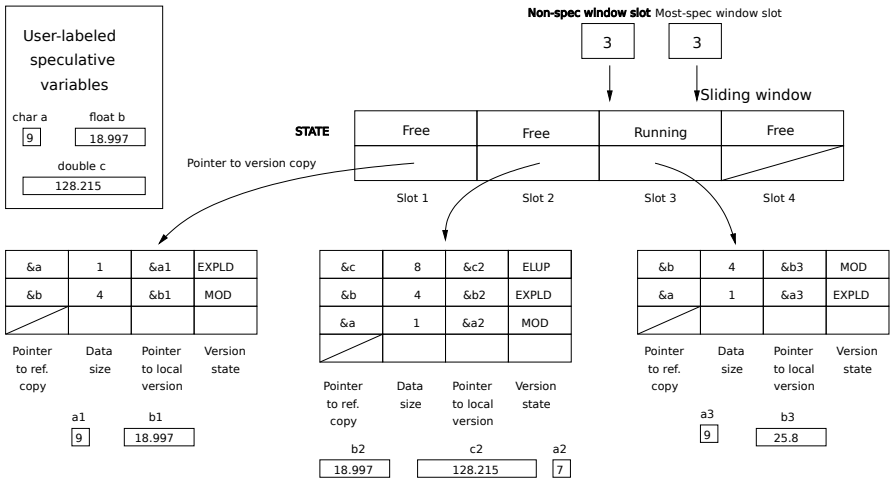
(j)

Figure 3.14: Speculative commit example (5/6). (i) The last element from slot 2 is committed. (j) There are not more elements to commit, so, the state of this slot is changed from DONE to FREE.



Version copy data structures

(k)



Version copy data structures

(l)

Figure 3.15: Speculative commit example (6/6). (k) No more slots could be committed, therefore, pointer to non-speculative slot is updated. (l) Final situation.

This operation requires changes over the commit operation. Thus, each thread accumulates in its local structure the amount added each iteration. Hence, if a thread should execute `N` iterations, when all of them finish its execution, the value will be '$N \times value$'. So, regarding the commit operation, instead of directly modifying the reference value, it performs a sum of '$N \times value$' each time that is called. Consequently, results are correct at the end of the operation.

Maximum reduction

Let us suppose that the loop to be parallelized contains an operation which calculates the maximum in the following way:

$$\texttt{IF } matrix(i) < value \texttt{ THEN } matrix(i) = value$$

Or with a function like:

$$matrix(i) = MAX(matrix(i), value)$$

This kind of operations cannot be parallelized because, as in the sum reduction, dependence violations would appear continuously. Nevertheless, this operation can be also reduced with something similar to:

$$matrix(i) = MAX(matrix(i), value1, value2, ..., valueK)$$

At the beginning the maximum was obtained comparing just two values, but now it requires to compare every single value. The function which implements this operation is the following.

$$matrix(i) = MAX(matrix(i), value)$$
$$\Downarrow$$
$$specredmax(VOID^* addr, UINT\ size, UINT\ chunk\_number, VOID^* value))$$

Where arguments have the following meaning:

1. *addr* is the address of the speculative variable in which the maximum value will be stored.
2. *size* is the size of the variable.
3. *chunk_number* is the number of the chunk being executed (needed to infer the slot being used).
4. *value* is the address of the variable which will be compared with the speculative variable.

This function works as follows. Each thread obtains the maximum of the values of its corresponding chunk of iterations. When a thread finishes its execution, committing its data, its *maximum* value is compared with the reference variable. If the reference value is lower than the value obtained by the thread, it will be modified, otherwise, no operation will be done.

Minimum reduction

Following the same approach than in the maximum reduction, sometimes might be required to obtain the minimum value of a set. This operation is also implemented in the speculative library, but since its behavior is similar to the *maximum reduction* described previously, it is not going to be detailed. The function which implements this operation is the following.

matrix(i) = MIN(matrix(i),value)

$$\Downarrow$$

specredmin(VOID addr, UINT size, UINT chunk_number, VOID* value))*

The Appendix B details the steps needed to parallelize a given application manually with the speculative library described.

3.6 Performance improvements

Throughout this section we will describe the methodology to find out what were the main bottlenecks of our new TLS runtime library. In addition, it contains the proposed solutions to mitigate the bottlenecks found.

3.6.1 Locating bottlenecks in the new TLS runtime library

In order to find the bottlenecks that our speculative engine have, we examined the source code in detail to extract some ideas of the tasks which require more time.

The main problem we have located is related with one of the new functionalities implemented. One of the main advantages of our new speculative parallelization library is that each thread only allocates the memory needed to store local copies of the data being speculatively accessed. This design decision comes at the cost of longer times to find the most-up-to-date value in speculative loads, and longer times to detect dependence violations in speculative stores, because both operations should traverse all the values accessed by all the predecessors and successors, respectively. Being T the number of threads, in [49], this operation was in $T \times O(1) = O(T)$, since all the memory needed to any data that might be accessed was allocated in advance. In our scheme, being N the number of data elements stored locally, the search is done in $T \times O(N) = O(TN)$.

In order to search for some results that endorse our theories, we implemented some auxiliary structures to store time measurements of all the functions involved in the execution. We can take a view in the Listing 3.2. The vectors shown save the time spent in the main speculative operations carried out by each thread. With their help, we can collect information about the average time spent by each function.

```
// time vectors
double time_specload[threads];
double time_specstore[threads];
double time_commit[threads];
```

Listing 3.2: Vectors to measure time spend by each function.

```
// At the beginning of the function...
#ifdef TIME_SPECLOAD
  double time_ini, time_end;
  int id = omp_get_thread_num();
  time_ini = get_time();
#endif

// Function code

// At the end of the function...
#ifdef TIME_SPECLOAD
  time_end = get_time();
  time_specload[id] += (time_end - time_ini);
#endif
```

Listing 3.3: Additional code to measure the time spent by each function.

In order to use the vectors described, we had to include some code in the functions to measure. In the Listing 3.3, we can see the additional code implemented in the *specload()* function. Where *get_time()* is a function that returns current time in microseconds. As we can see, we have used an `#IFDEF` clause, hence, we can measure just a single function, all of them, or none in a given execution. These parameters allow us to get only the interesting values of each experiment.

Experimental results show us that the main bottlenecks were in the *specload()* and *specstore()* functions, thus reinforcing our initial guess in which we supposed that traversing all the values of predecessors, or successors, respectively, was the main bottleneck. We decided to perform more comprehensive measures since time counters can be affected due to multitasking issues. Therefore, we perform a deeper analysis using *ICC* (*Intel C compiler*). This compiler allows to profile functions' execution times just modifying the compilation flags of the application. Specifically, we had to add the flag `-profile-functions` However, one of the disadvantages that presents this software is that the use of OpenMP with profiling functions is not allowed so far. Therefore we could use this measurement strategy only in

```
#ifdef MEASURE_COUNTERS
  long long int totalAccessesSpecload[threads];
  long long int totalCallsSpecload[threads];
#endif
```

Listing 3.4: Vectors to measure calls and accesses done by specload().

experiments with a single thread. Nevertheless, this requirement was acceptable because it allowed us to extract a measure of the most time-consuming functions of the software.

To be able to understand the acquired results (some XML files produced after executing the application with the mentioned compilation flag), we needed to install the *VTune Amplifier XE 2011*. Once installed, and executed, we could see that results were similar than those obtained with the calls to the *get_time()* function.

We also decided to directly measure the number of accesses required by each call to the most time-consuming functions. Listing 3.4 shows the vectors defined to measure the calls and accesses done in a specload() call, while Listing 3.5 contains the additional code used to obtain these parameters. As a result, we obtained the Tables 3.1 and 3.2 which show the total calls to *specload()* and *specstore()* operations for the benchmark tested. In these tables, TC (total calls) is the total number of calls to each function; TS (total searches) is the total number of accesses in order to complete the corresponding call (getting the most up-to-date value in *specload()*, and searching for potential dependence violations in *specstore()*). These numbers are average values obtained in three real executions in our target system.

Table 3.1 shows the values obtained when using the TLS library to execute each benchmark (described in Appendix A), but using just one thread. This means that all accesses counted in TS were to the local version data. As it can be seen, this number of accesses in speculative loads is high on average, from 33.18 to 102.86. The situation is much worse on speculative stores, with up to 737 accesses on average in order to detect a potential dependence violation. Compare these numbers with the single access needed by both reads and writes in a non-speculative execution of the same algorithm.

Table 3.2 shows the same values when we speculatively execute the benchmarks with eight threads. Costs of both speculative loads and stores are roughly doubled. This situation becomes even worse when the number of cooperative threads was increased.

These figures strongly endorse the affirmation that the main bottleneck and the most severe scalability limitation comes from the sequential traversing of version copies during speculative loads and stores operations. One way to speed up these searches is to switch to a different data structure to hold local version copies of data. Instead of using a single table per thread as version copy data structure, we have developed a simple but extremely powerful alternative, using a hash function and H tables.

```
// each time specload() is called...
#ifdef MEASURE_COUNTERS
  totalCallsSpecload[thread]++;
#endif

// each time a new variable is accessed...
#ifdef MEASURE_COUNTERS
  totalAccessesSpecload[thread]++;
#endif
```

Listing 3.5: Additional code to measure calls and accesses done by specload().

		Speculative load			Speculative store		
Application	**Input set**	**TC**	**TS**	**TS/TC**	**TC**	**TS**	**TS/TC**
2D-Hull	Disc	2.63×10^{8}	2.23×10^{10}	87.39	4.39×10^{4}	2.63×10^{7}	598.10
	Square	2.97×10^{8}	3.05×10^{10}	102.86	6.65×10^{3}	4.90×10^{6}	737.35
	Kuzmin	2.29×10^{8}	1.13×10^{10}	49.35	1.65×10^{3}	5.30×10^{5}	320.64
Delaunay	100K	1.14×10^{7}	3.77×10^{8}	33.18	5.06×10^{6}	1.69×10^{8}	33.44
	1M	1.47×10^{8}	8.17×10^{9}	55.42	5.28×10^{7}	3.29×10^{9}	62.32

Table 3.1: Profile of main functions with a single thread in the baseline TLS library.

		Speculative load			Speculative store		
Application	**Input set**	**TC**	**TS**	**TS/TC**	**TC**	**TS**	**TS/TC**
2D-Hull	Disc	2.74×10^{8}	4.00×10^{10}	145.85	7.48×10^{4}	1.14×10^{8}	1526.99
	Square	3.17×10^{8}	3.55×10^{10}	111.86	2.40×10^{4}	3.30×10^{7}	1374.33
	Kuzmin	2.35×10^{8}	1.18×10^{10}	50.18	5.78×10^{3}	4.22×10^{6}	729.73
Delaunay	100K	1.14×10^{7}	1.42×10^{9}	124.83	5.07×10^{6}	7.51×10^{8}	148.09
	1M	1.47×10^{8}	3.24×10^{10}	219.89	5.28×10^{7}	1.31×10^{10}	248.69

Table 3.2: Profile of main functions with eight threads in the baseline TLS library.

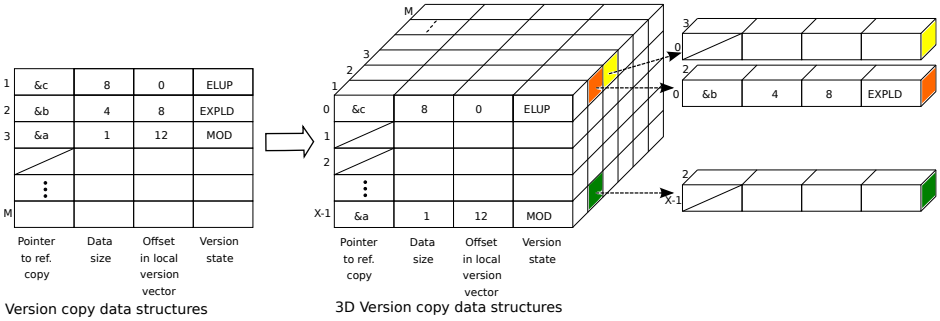


Figure 3.16: Hash-based version copy data structures with three dimensions.

3.6.2 Keeping version copies: A hash-based solution

We have devised an extremely simple solution that is capable of reducing the number of accesses needed by a factor of one to three orders of magnitude, while keeping the storage cost of local versions in $O(1)$. In addition, the developed technique can be seen as a general solution that can be applied not only to our baseline implementation, but also to most TLS systems. The solution works at follows: At the beginning of each *specload()* and *specstore()* call, we perform an AND operation of the address of the datum to be processed with a mask composed by H 1s. Since many addresses are multiple of 4 or 8, the address is first shifted three positions to its right, to avoid biased hash values. The resulting value will be used as a hash value. Considering an address A, the operation is:

$$\text{hash} = (A >> 3) \wedge 00...0\overbrace{11...1}^{H}$$

Instead of having just one table to keep local values, each thread maintains H tables. The obtained hash is used to look into the H-th table of all predecessors and successors, effectively speeding up the search by an average factor of H without increasing the time needed to add a new row to the corresponding table, leading to $O(T \times \frac{M}{H})$ search times. Note that, while T is a relatively small number (typically up to 64 for current shared memory architectures), H can be set as big as needed to compensate for higher values of M.

Figure 3.16 shows the new version copy structure in three dimensions with an example. Suppose that the previous version copy of a thread has the values depicted on the left of the figure. With the new structure in three dimensions, this version copy is transformed into the structure shown on the right of the picture, supposing that the hash of `c` and `b` is `0`, and the hash of `a` is `X-1`. Note that the first value of each hash row that has not been previously used is marked as void. The same occurs with the third position (depth) of the `0` hash position, and the second position (depth) of the `X-1` hash position.

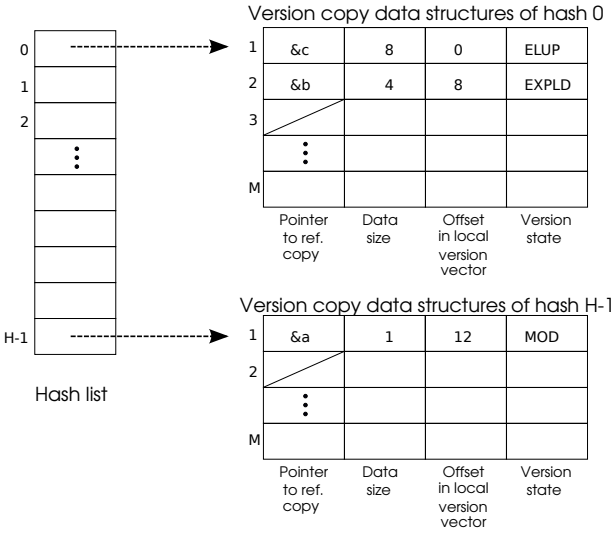


Figure 3.17: Hash-based version copy data structures.

In practice, these ideas have been implemented with the third dimensional structure mentioned but, in order to have a better understanding of this optimization, let us introduce another point of view. Imagine the structure with three dimensions as a vector with H positions where each one of them has H pointers to H version copy structures. So, instead of using a version copy for each slot, H version copies for each slot will be used. This idea, conceptually similar to the one explained in the previous paragraph, is depicted in Figure 3.17.

Tables 3.3 and 3.4 show the total calls to *specload()* and *specstore()* operations, with the total accesses that each one of them needed to find the desired value, and the average accesses per call, in the new hash-based solution with one and eight processors respectively for each benchmark used. Compare the values in both tables for TS/TC columns (up to 6.29 accesses) with the much higher values (up to 1500) of the sequential searches shown in Tables 3.1 and 3.2. Moreover, this solution comes at no cost, since the hash function is extremely easy to implement.

Memory consumption

New structures could seem to use more space than the previous ones, however, the memory used are similar in both schemes: Let us suppose that in the first approach each thread managed N rows to store intermediate values. If there were T threads, the cost to store them was $T \times O(N)$. On the new scheme, with H hash rows that contain M positions to store values, considering that $H \times M = N$ the cost to store them is $T \times O(H \times M)$. In the best case, if values were uniformly dispersed throughout the hash rows, $H \times M \approx N$. In the

		Speculative load			Speculative store		
Application	**Input set**	**TC**	**TS**	**TS/TC**	**TC**	**TS**	**TS/TC**
2D-Hull	Disc	2.62×10^8	3.46×10^8	1.32	4.39×10^4	1.47×10^5	3.34
	Square	2.97×10^8	3.94×10^8	1.33	6.65×10^3	2.53×10^4	3.80
	Kuzmin	2.29×10^8	2.78×10^8	1.21	1.65×10^3	3.45×10^3	2.08
Delaunay	100K	1.14×10^7	1.74×10^7	1.53	5.06×10^6	8.18×10^6	1.62
	1M	1.47×10^8	2.35×10^8	1.59	5.28×10^7	9.02×10^7	1.71

Table 3.3: Profile of main functions with a single thread in the hash-based version of the library.

		Speculative load			Speculative store		
Application	**Input set**	**TC**	**TS**	**TS/TC**	**TC**	**TS**	**TS/TC**
2D-Hull	Disc	2.74×10^8	4.47×10^8	1.63	7.24×10^4	4.55×10^5	6.29
	Square	3.19×10^8	4.38×10^8	1.37	2.28×10^4	1.42×10^5	6.23
	Kuzmin	2.34×10^8	2.85×10^8	1.22	7.18×10^3	2.16×10^4	3.01
Delaunay	100K	1.14×10^7	6.06×10^7	5.32	5.07×10^6	8.67×10^6	1.71
	1M	1.47×10^8	7.43×10^8	5.04	5.28×10^7	1.08×10^8	2.05

Table 3.4: Profile of main functions with eight threads in the hash-based version of the library.

worst case, if a single hash stored all the N values, the cost of the new approach would be $(T-1)\times O(H\times M)+T\times O(N)\approx O(T\times N)$. So, required memory to store the new approach will be similar to the previous one.

3.6.3 Additional improvements

Use of dedicated buffers

One of the problems detected was the excessive number of calls to the *malloc()* and *free()* functions. To better understand the reasons, we will use an example. Suppose that a thread executes a *specload()* or *specstore()* call. In both functions, the first task carried out by the thread is to search in its version copy matrix to check whether this address has been accessed by this thread. Suppose that the datum has not been used yet, so it should be added to the matrix. In this process of attaching the new datum to the matrix of the thread, we had to allocate some memory so as to store the local copy, so *malloc()* should be called (see Figures 3.5 and 3.6 (specload), or Figures 3.7, 3.8 and 3.9 (specstore)).

Once the thread has finished the speculative execution of the chunk of iterations which had been assigned, it should free all its allocated memory. Therefore, *free()* should be re-

peatedly called to deallocate each single version copy of speculative data, in order to reuse the remaining data structures to handle the execution of a new chunk of iterations.

It is easy to see that these operations are costly because they are called quite frequently. Furthermore, calls to *malloc()* are in the critical path of the speculative execution. A solution to avoid these calls is to implement a container for all the data used by each thread. Hence, a new, *Local Version Data* dynamic vector was added to each thread. These vectors need an initial call to the *malloc()* function to allocate them, and a single *free()* call to deallocate them once the parallel loop has been executed entirely. If the vector is full, an additional call to *realloc()* may be needed. This solution greatly improves the performance observed.

This new structure, however, leads to changes in the basic structures of the architecture. Initially we had an structure with four entries, where one of them was a pointer to the local copy of the datum. Instead, the new solution manages an offset for each datum. In this way, each datum will be stored in the Local Version Data vector at the position pointed by its offset, in other words, it will be stored from the position pointed out by its offset to the same position plus the size of the datum, since each position of the vector will store a byte. Note that, once stored, the datum will not be deleted until the entire vector containing speculative data has been committed, so fragmentation will not occur.

As a result, each slot of the sliding window requires an additional pointer to the first free position of its vector, to allow fast insertions. Hence, the sliding window has been augmented with this additional datum. This ideas will be better understood with the aid of a graphical example.

Figure 3.18(a) shows the new structures that have been implemented and the initial values used in the example. In this way, we supposed a situation in which just a single thread is in execution (to avoid an unnecessarily complex situation) and is managing two variables, `c` and `b`, with a size of `8` and `4` respectively. Consequently, taking into account that this library is implemented in `C` language (vectors begin at 0 position), actual offset of this thread will be `12`.

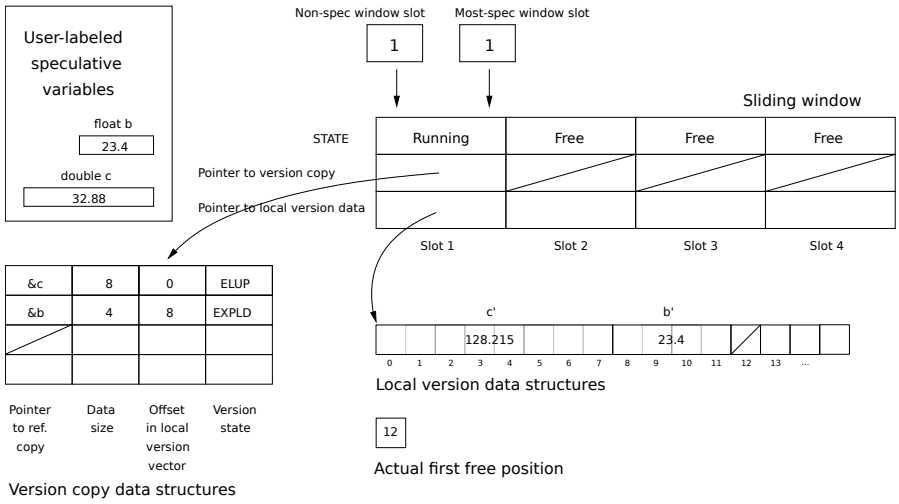
Figure 3.18(b), and Figures 3.19(c) and (d) show the process of adding new data to the version copy of a slot. It starts when a new data is modified in the context of the thread, specifically `a`, with a size of `1` byte. Figure 3.18(b) shows that this datum has not been used yet, and therefore it should be added.

First of all, current value of `a` is stored in the new `Local Version Data` vector. After that, thread working in slot 1 adds a new row to its version copy data structure, storing the address of `a`, its data size, the offset where is stored the version copy of `a` being managed by the thread, and the new state for this version copy, `MOD` (see Figure 3.19(c)).

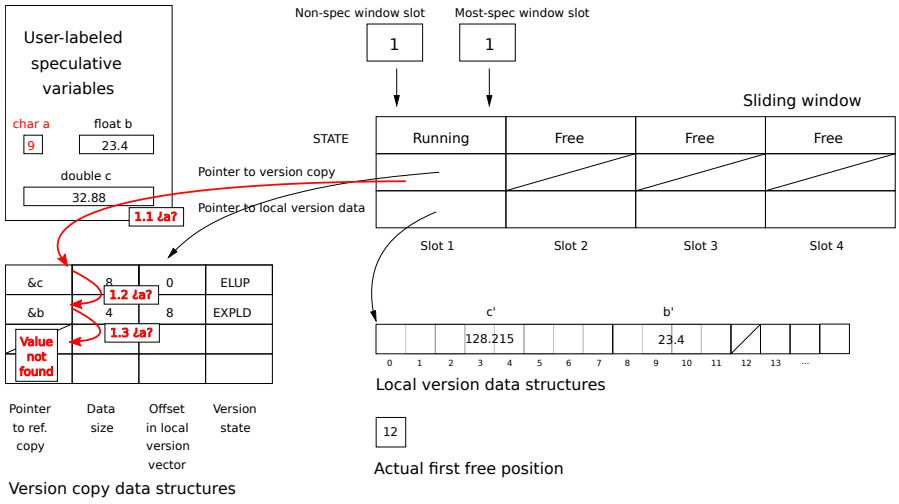
Finally, we should update the value of the first free position in the `Version Data` vector, in this case, we should augment this value by one: $12 + 1 = 13$ (see Figure 3.19(d)).

Note that, after executing a chunk of iterations, it is no longer needed to *free()* this buffer. It is enough resetting the index that points to the beginning of the free space on it.

Being T the number of threads, and M the number of data elements stored locally, the original solution came at a cost that was in $T \times O(M) \subset O(T \times M)$. With the new

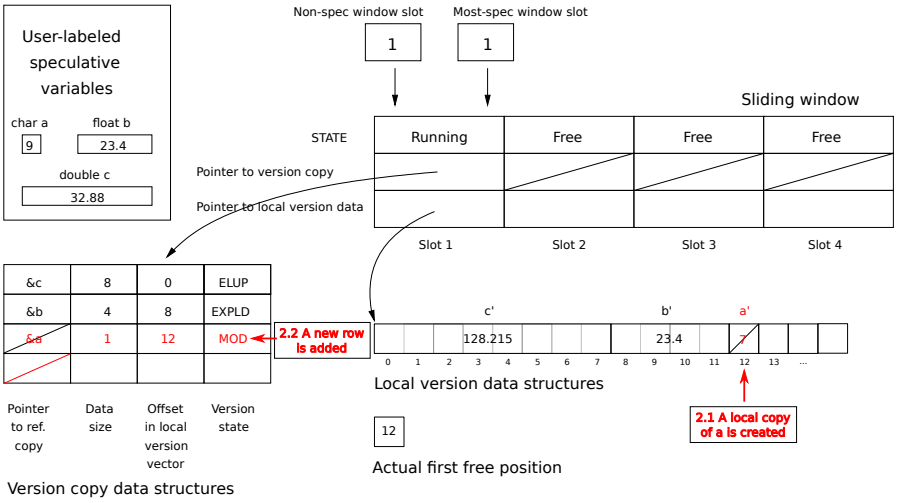


(a)

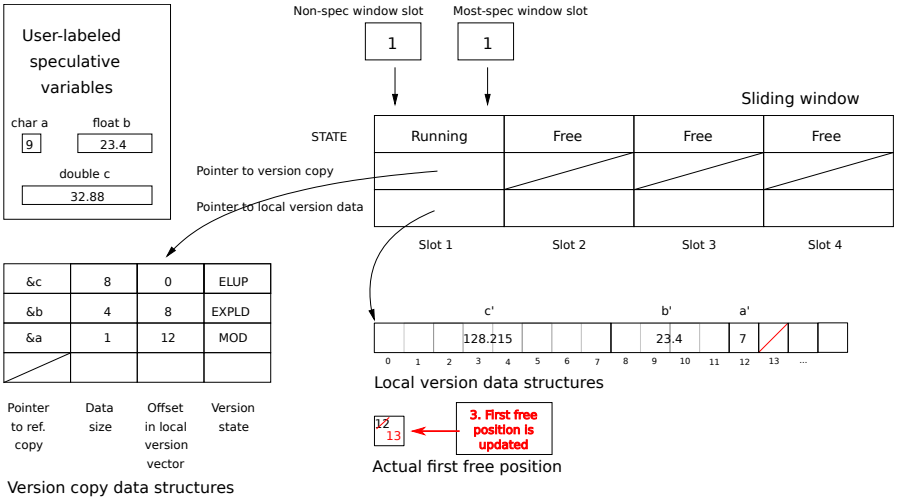


(b)

Figure 3.18: Reducing operating system calls example (1/2). (a) Initial values of the example with the new data structures of this optimization. (b) The thread working in slot 1 scans its version copy to find the value.



(c)



(d)

Figure 3.19: Reducing operating system calls example (2/2). (c) After creating a local copy of a in its vector of local copies, the thread working in slot 1 adds a new row to its version copy data structure. (d) After storing a copy of a's value, the thread working in slot 1 augment the indicator to the first free position in the vector of local copies.

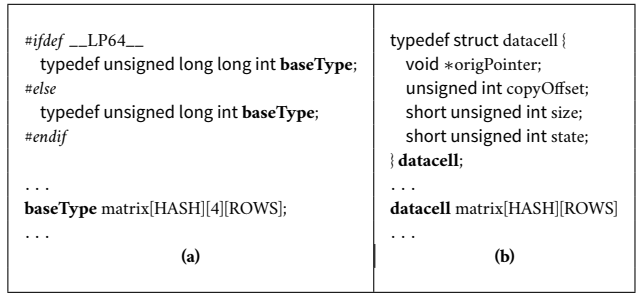


Figure 3.20: Implementation of a static example of the data structure in (a) the baseline solution, and (b) the improved version.

scheme, the space allocation is done in $T \times O(1) \subset O(T)$, asymptotically improving the performance of the library.

As the reader may have deducted, Figures 3.16 and 3.17 already reflect this improvement, so it do not require further changes in the implementation.

Structures instead of buffers

We have also modified the implementation of version copy data structures in order to improve data alignment [30] and the space needed by each version copy data structure. The baseline representation is shown in Fig. 3.20(a). Although the different elements in a row have different sizes, the declaration should allocate enough space to store the biggest one, in our case the pointer to the original data. This implementation requires 8 bytes for each value, that is, a total of 32 bytes for the representation of each row. Our new representation, shown in Fig. 3.20(b), states variables as a struct. With this representation we achieve two goals, first, we reduced the memory needed by half, since the memory used to store this new structure is 16 bytes (a void pointer needs 8 bytes, an unsigned int needs 4 bytes, and each unsigned short int needs 2 bytes). Second, this structure also exploits the memory representation of C structs because these types are usually stored with the following patterns[30]: Structures between 1 and 4 bytes of data are usually padded so that the total structure is 4 bytes; Structures between 5 and 8 bytes of data are padded so that the total structure is 8 bytes; structures between 9 and 16 bytes of data are padded so that the total structure is 16 bytes; and structures greater than 16 bytes are padded to 16 byte boundary. A different order of the variables would add paddings because the compiler may decide to store them in 4-bytes places. Therefore, our new structure is optimized to minimize the memory space needed, thus reducing the number of cache misses.

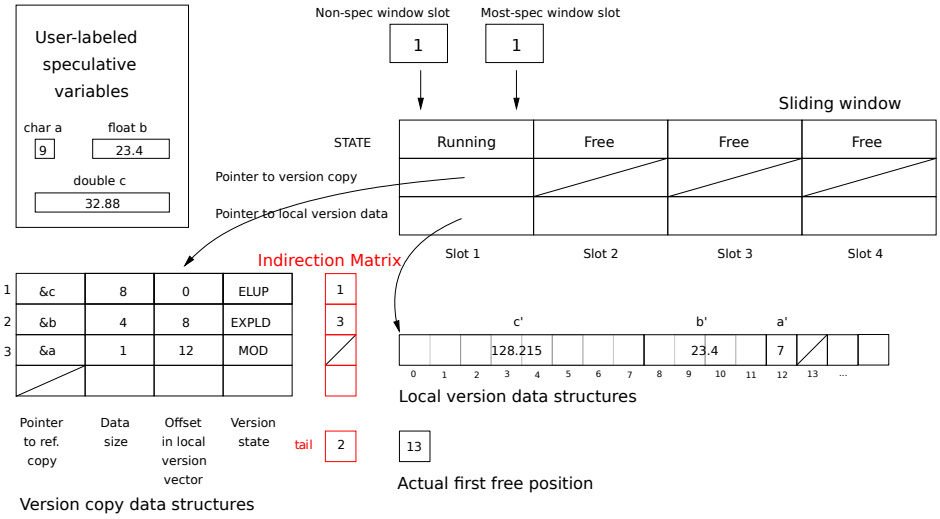


Figure 3.21: Structures with the Indirection Matrix.

Commit optimization

Initially, commit operations required to check all local elements accessed by a thread, both modified and unmodified items. In addition to the other modifications, we performed an optimization inspired in the work described in [48, 49], an *Indirection Matrix*. This matrix will optimize commit operations since it will store the list with updated elements, allowing to just commit them.

With this solution each thread will have its own vector of modified items, whose positions will point to positions of updated items. Additionally, a *tail* pointer which points out the position of the last modified item of the *Indirection Matrix* will be added so that ease even more the commit process, and the addition of new elements to the vector. Now, to perform the *commit* operation it is just needed to copy the elements of this list.

Figure 3.21 shows this new structures in the general architecture, including the optimization of the vector with the local data. In the situation depicted in the example, the thread has just finished the execution of its chunk of iterations. Since is the non-speculative one, it will begin to commit its elements. It therefore scans its *Indirection Matrix* to locate those variables in `ELUP` or `MOD` state. In the example, `c` has been loaded and updated, `a` has just been modified while `b` has been read. So, the thread copies the content of `a'` into `a`, and the content of `c'` into `c`. In this way, the attempt of committing the content of `b` is avoided, unlike what had happened in the original solution (see Figures 3.10, 3.11, 3.12, 3.13, 3.14 and 3.15).

Unlike other improvements, the *Indirection Matrix* cannot be directly applied to the hash-based main optimization. Since hash-data structure requires third dimensions to be implemented, in order to combine both approaches, the *Indirection Matrix* also needs an additional dimension to its structure. Otherwise, each position of the *Indirection Matrix* will point to a single `version copy` instead of taking into account existing `H version copies`. Just including the hash position `H` instead of the position of the datum in the version copy will not be desirable because a hash position will point to several data, and then updating just one of them, its position would be added, thus including those data from the row that have not been modified, and reducing the effectiveness of the *Indirection Matrix*.

Moreover, if the implementation was not modified the same row will be attached several times in the same column of the mentioned matrix. This case will be better understood with the use of an example. Suppose that a thread updates a datum from the *hash* position `30`, that points to the address `5 000` of the memory. In order to follow the semantics of the *specstore()* operation, this thread will add the datum to its matrix in the `30`*th hash* position. Finally, this *hash* position is added to the *Indirection Matrix*, and the pointer to the last data of this matrix is augmented. Then, suppose that, afterwards, the same thread updates another datum, with the same *hash*, `30`, but with other address, for example, `6 000`. Then the datum will be added to the matrix of the thread, and then, the *hash* position of this datum (`30`) is erroneously attached again to the *Indirection Matrix*. Hence, attaching a new dimension to the mentioned *Indirection Matrix* has become mandatory.

This implementation will allow to only commit the elements of the *hash* position that have been used, instead of all of them. In this way, each version copy used will have its own *Indirection Matrix*. Now, when a thread tries to add a new datum to its *Indirection Matrix*, it also needs a hash `H` to obtain the third dimension of this new *Indirection Cube*.

Consequently, it is also required to add a new dimension to the tail position of the indirection matrix because each `H` hash position of the structure has its own number of modified items.

3.7 Experimental evaluation

3.7.1 Experimental setup

Experiments were carried out on an Intel S7000FC4URE server, equipped with four quad-core Intel Xeon MPE7310 processors at 1.6 GHz and 32 GB of RAM. The system runs Ubuntu Linux operating system. All threads had exclusive access to the processors during the execution of the experiments, and we used wall-clock times in our measurements. We have used gcc 4.6.2 for all applications. Times shown in the following sections represent the time spent in the execution of the parallelized loop for each application. The time needed to read the input set and the time needed to output the results have not been taken into account.

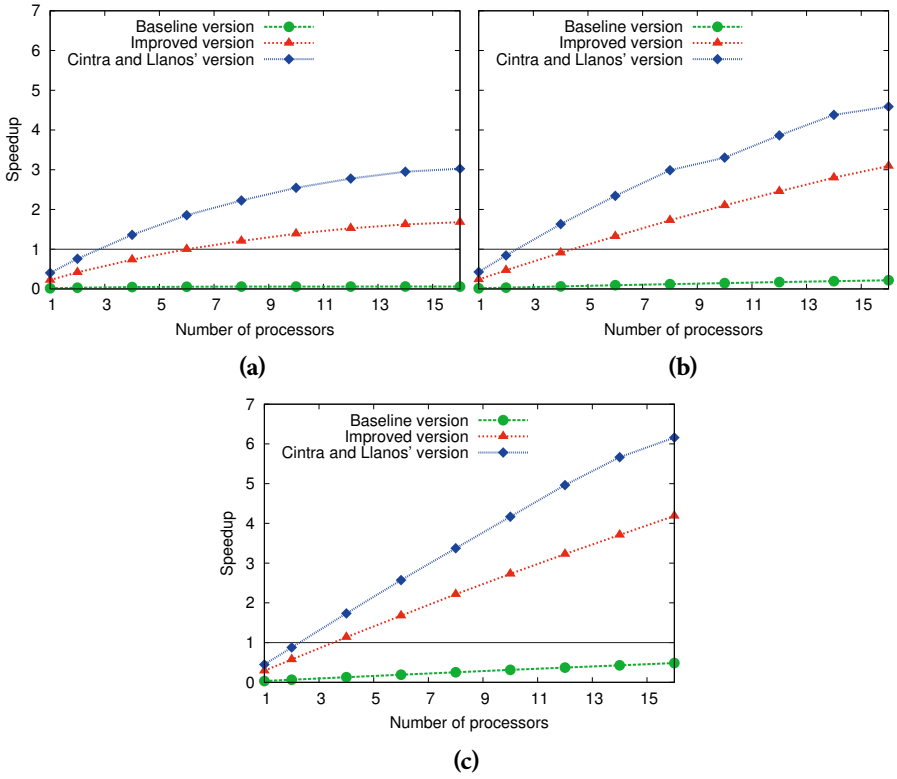


Figure 3.22: Performance comparison for 2D-Hull benchmark with three different input sets: (a) Disc, (b) Square, and (c) Kuzmin.

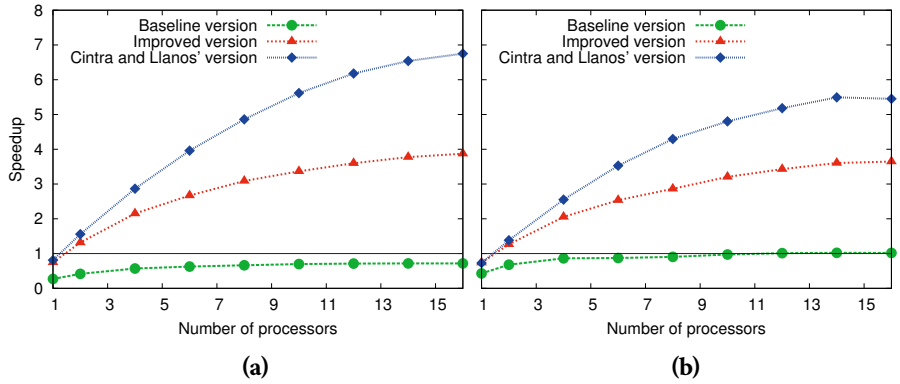


Figure 3.23: Performance comparison for Delaunay benchmark with an input set of (a) 100K points, and (b) 1M points.

3.7.2 Experimental results

To test the improvements implemented, we have used two applications the 2-dimensional Convex Hull problem (2D-Hull) and the Delaunay Triangulation problem. The 2D-Hull problem solves the computation of the convex hull (smallest enclosing polygon) of a set of points in the plane. The Delaunay triangulation applied to a two-dimensional set of points affirms that a network of triangles is a Delaunay triangulation if all the circumcircles of all the triangles of the network are empty. These benchmarks are detailed in more depth in Appendix A.

Figure 3.22 compares the performance results of the speculative parallelization of the 2D-Hull with different input sets. Both the baseline version of the library and the version with all improvements implemented are compared. While the baseline system is not able to obtain speedups in any case, the new solution leads to a maximum speedup of 1.681× for the Disc input set (representing a 28× performance increment with respect to the baseline TLS library), 3.094× for the Square input set (14.19× performance increment) and 4.188× for Kuzmin (8.63× performance increment). To put these results into perspective, we also show the best speedups obtained by Cintra and Llanos with their speculative runtime system. Recall that their solution, while effective, is constrained by many limitations, and it is not generalizable to any applications. Results show that our solution allows to deliver a good percentage of the maximum speedup attainable (up to 68%), while offering a speculative solution applicable in many more cases.

Figure 3.23 compares the performance results of the speculative parallelization of the Delaunay triangulation with two input sets. The new solution is again clearly better, with a maximum speedup of 3.646× for the 1M-points input set (representing a 3.58× performance increment with respect to the baseline TLS library), and 3.873× for the 100K-points input

set (5.40× performance increment). Again, our solution is compared to the tailored library of Cintra and Llanos, obtaining, on average, a 75% and a 68% of their maximum speedup in the 1M-points and the 100K-points input sets, respectively.

3.8 Conclusions

In this chapter, we have described a TLS runtime library which can be easily combined with a compiler in order to ease to parallelize applications speculatively. Thus we have detailed its structures as well as its operations. Furthermore we have shown a solution to a problem that is common to any software-based TLS library: How to reduce search times when accessing to remote versions of speculative data. To mitigate this problem, we have implemented some optimizations such as the use of an extremely-simple hash function to avoid the need of traversing all version data, the reduction in the number of memory management system calls, and the development of new data structures to reduce memory consumption and cache misses. Our experimental evaluation with non-synthetic benchmarks on a real, shared-memory multiprocessor clearly shows that these improvements have a dramatic impact on performance: All applications tested had far better execution times than those obtained in the baseline version, and the performance results are a significant fraction of those obtained with a system specifically designed to handle these benchmarks.

Figure 3.24 summarizes the final scheme.

The work described in this chapter has generated the following publications:

- Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros'. In: *Proceedings of the XXIII Jornadas de Paralelismo*. Elche, Alicante, Spain: Servicio de Publicaciones de la Universidad Miguel Hernández, Sept. 2012. ISBN: 978-84-695-4471-6

- Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Improving the Perfomance of a Pointer-Based, Speculative Parallelization Scheme'. In: *Proceedings of the 1st First Congress on Multicore and GPU Programming*. PPGM'14. Granada, Spain, Feb. 2014. Also published in *Annals of Multicore and GPU Programming*, vol. 1, no. 1, 2014. 2014. issn: 2341-3158.

- Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'New Data Structures to Handle Speculative Parallelization at Runtime'. In: *Proceedings of the 7th International Symposium on High-level Parallel Programming and Applications*. HLPP '14. Amsterdam, Netherlands: ACM, 2014, pp. 239–258. Also published in *International Journal of Parallel Programming*, 2015, pp. 1–20. Springer US, 2015. issn: 0885-7458. doi: 10.1007/s10766-014-0347-0. url: http://dx.doi.org/10.1007/s10766-014-0347-0.

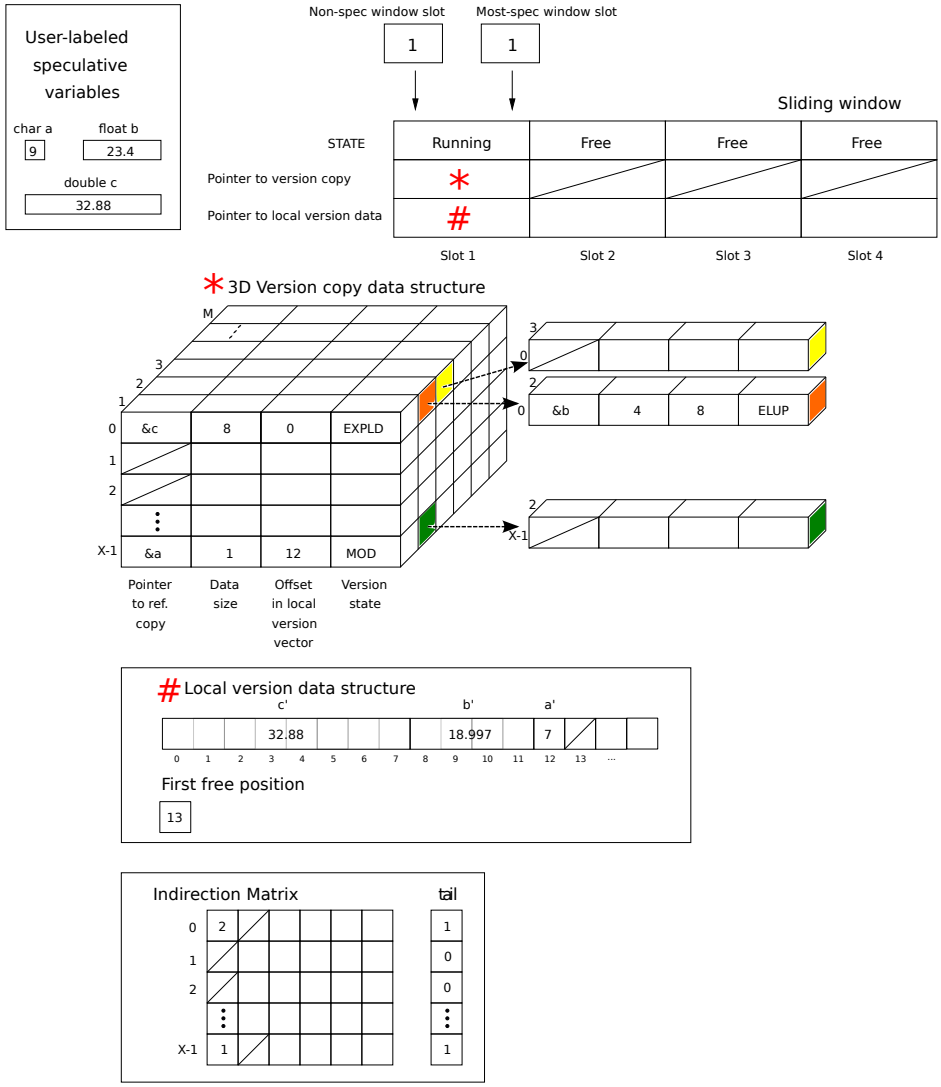


Figure 3.24: Current appearance of the TLS runtime library.

Chapter 4

The ATLaS framework

OpenMP directives are the *de-facto* standard for shared-memory parallel programming. However, OpenMP does not guarantee the correctness of the parallel execution of a given loop if runtime data dependences arise. Consequently, many highly-parallel regions cannot be safely parallelized with OpenMP due to the possibility of a dependence violation. In this chapter, we detail the ATLaS C Compiler framework, which takes advantage of TLS techniques to expand OpenMP functionalities, and guarantees the sequential semantics of any parallelized loop. Specifically, the ATLaS framework is able to handle *speculative* variables, classified in the same way as OpenMP does, through a new clause. It makes the compilation process, and the management of variables completely transparent to users, considerably reducing the required effort.

```
for (i=0; i<MAX; i++) {
   b = func(i);
   v[i] = b * a[i];
}
```
(a)

```
#pragma omp parallel for \
 private (i,b) shared (a,v)
for (i=0; i<MAX; i++) {
   b = func(i);
   v[i] = b * a[i];
}
```
(b)

Figure 4.1: Example of loop parallelization with OpenMP.

4.1 Problem description

The advent of multicore technologies in the new century made parallel processing ubiquitous. Many parallel languages and parallel extensions to sequential languages have been proposed to exploit the capabilities of modern multicore systems. The most successful proposal is OpenMP [40], a directive-based parallel extension to sequential languages (such as C, Fortran or C++) that allows parallel execution of user-defined code regions.

Figure 4.1 shows an example of (a) a sequential C loop, and (b) its parallelization with OpenMP directives. As can be seen, all variables inside the loop body should be classified as private or shared. Informally speaking, variables whose values are always set in a given iteration before their use should be labeled as private, while variables that have values visible by all threads executing the loop in parallel should be classified as shared (Remark that Chapter 1 explained with more detail how to classify variables). In our example, `a[]` is a read-only shared vector, while `v[]` is a shared vector that is modified by each iteration.

As OpenMP is a simple and powerful mechanism for code parallelization, its use has several limitations. First, the classification of all variables inside the critical region, according to their use, is a time-consuming, error-prone task. Second, OpenMP does not ensure the parallel execution of the code according to sequential semantics, as the programmer is responsible for such a task. In the example shown in Fig. 4.1, the programmer is responsible for ensuring that each thread modifies a different element of `v[]`. Third, in many cases, potentially-parallel regions cannot be safely parallelized because their control flow depends on runtime data. Consider the code depicted in Fig. 4.2. Suppose that the value of `k` is not known at compile time. Assuming `b>0` for a given `i`, if the parallel execution of the loop calculates iteration `i` *before* iteration `i-b`, access to `v[i-b]` may return an outdated value, breaking sequential semantics. The only way to guarantee a correct behavior would be to serialize the execution of iterations $i - b$ and i, a difficult task in the general case.

Safely parallelizing loops that may present runtime dependence violations can have a significant impact in terms of performance. Aldea et *al.* have previously measured the amount of loop-level parallelism that could be extracted from the SPEC CPU 2006 benchmark, with different techniques [12]. Their results show that, while around 48% of the loops present in the applications analyzed (representing around 13% of their aggregate execution time) are potentially parallelizable with existent parallel programming models such as OpenMP, an

```
for (i=0; i<MAX; i++) {
   b = func(i);
   if (b==k)
      v[i] = v[i-b];
   else
      v[i] = b * a[i];
}
```

(a)

```
#pragma omp parallel for \
 private (i,b) shared (a,k) \
 speculative(v)
for (i=0; i<MAX; i++) {
   b = func(i);
   if (b==k)
      v[i] = v[i-b];
   else
      v[i] = b * a[i];
}
```

(b)

Figure 4.2: A loop that cannot be safely parallelized with current OpenMP clauses (a), and its parallelization with our new speculative clause (b).

additional 38% of loops (representing around 20% of the execution time) could be run in parallel with the help of runtime speculative parallelization techniques.

The ATLaS framework consists in augmenting OpenMP with software-based, Thread-Level Speculation (TLS) techniques to ensure that definitions and uses of shared variables are carried out according to sequential semantics. This solution allows the OpenMP programming model to be used even when dependence violations may arise at runtime. To do so, we define a new speculative clause. Variables labeled as speculative will be accessed following two simple rules:

- All reads of a speculative variable will return the most up-to-date value for this variable. This value can either be generated previously by this thread or by any of its *predecessors*, defined as threads that execute earlier iterations according to sequential semantics. This is called a *forwarding* operation.
- All writes to a speculative variable will store the value in a local copy, and will check whether a *successor* thread (that is, threads that are executing "future" iterations) has consumed an outdated value of this variable. In this case, the offending thread (and possibly some of its successors) will be stopped and re-started, in order to force them to consume the updated value of the variable. This is called a *squash* operation.

As long as a dependence violation forces the values of speculative variables to be discarded, all threads maintain version copies of the speculative variables being accessed. When a *non-speculative* thread (that is, a thread with no alive predecessors) successfully finishes the execution of its block of consecutive iterations, all changes are committed to the main copy of all speculative variables. After this commit operation, the thread will become the *most speculative* one, since it will execute the following block of iterations that remains unassigned.

The three main contributions of this chapter, i.e., of ATLaS framework, are the following:

1. It extends OpenMP specifications adding a clause to support speculative accesses to data in omp parallel for constructs. This clause follows the guidelines proposed by Aldea et *al.* [13].

2. It is internally managed by the brand-new TLS runtime library described in the previous chapter. This runtime library not only manages accesses to speculative data, but also handles the scheduling of iterations among threads and ensures correctness in the parallel execution of the loop.

3. It defines a new plugin-based compiler pass to the GCC OpenMP implementation to support the speculative clause. This pass transforms the loop to be parallelized, inserting the runtime TLS calls needed to (a) distribute blocks of iterations among processors, (b) perform speculative loads and stores of speculative variables, and (c) perform partial commits of the correct results calculated so far.

The result is ATLaS, a complete framework that allows OpenMP to execute loops in parallel without the need of a prior dependence analysis. Our performance evaluation, using both synthetic and real-world applications on a real multicore system, shows that this approach leads to performance speedups.

The rest of the chapter is organized as follows. Section 4.2 briefly describes Aldea et *al.*'s proposal of a new OpenMP speculative clause, and how they added support to handle this clause in the GCC OpenMP compiler. Section 4.3 presents the experimental evaluation. Finally, Section 4.4 summarizes our conclusions.

4.2 Compilation phase description

First of all, remark that this phase was fully developed by Aldea et *al.*. Thus, we are not going to give a detailed description of this phase. Inside each subsection are referenced the documents in which find out more information.

4.2.1 Semantics of Aldea et *al.*'s *speculative* clause

The problem of adding speculative parallelization support to OpenMP can be handled using two approaches. The first one requires the addition of a new directive, such as `pragma omp speculative for`. However, there are many OpenMP related components that should be modified in order to add a new directive. A simpler solution is to add a new OpenMP clause to the list of available parallel constructs, which allows the programmer to enumerate which variables should be handled speculatively. The syntax of this clause is:

```
speculative(variable[, var_list])
```

In this way, if the programmer is unsure about the use of a certain data structure, he can simply label it as speculative. In this case, a tailored OpenMP implementation should replace all definitions and uses of this data structure with the corresponding specload() and specstore() function calls. An additional commit_or_discard() function will be automatically inserted once each thread has finished its chunk of iterations, to either commit the results, or to restart the execution if the thread has been squashed due to a runtime dependence violation.

Our new TLS runtime library, described in-depth in the following chapter, was indeed developed using standard OpenMP clauses. In order to integrate our library into an experimental OpenMP framework that includes a new speculative clause, two particularities of our TLS library should be taken into account. First, since our TLS runtime library has also been developed using OpenMP, some private and shared control variables should be added to the target loop in order to use it. Therefore, if a speculative clause is found by the compiler, this occurrence, which implies the use of our speculative library, should trigger the inclusion of several private and shared variables to the existing lists. Fortunately OpenMP allows the repetition of clauses, so the compile time support for this new speculative clause can add additional private and shared clauses that will later be expanded by the compiler.

Second, the standard scheduling methods implemented by OpenMP are not enough to handle speculative parallelization. These methods assume that the execution of a chunk of iterations will never fail, so they do not consider the possibility of restarting a chunk that has failed due to a dependence violation. Therefore, it is necessary to use a speculative scheduling method (Scheduling methods for TLS are detailed in Chapter 5).

For further information about the development of this clause see [7, 13].

4.2.2 Compiler support for the *speculative* clause

The compiler phase of ATLaS is implemented on the GCC C compiler [103], extending its functionality through a plugin. Before describing the implementation of the plugin, it is necessary to introduce the GCC architecture.

GCC architecture in a nutshell

Figure 4.3 shows the scheme of the GCC architecture [102, 187]. In basic terms, GCC is a big pipeline that converts one program representation into another, in different stages. Each stage generates a lower-level representation, until the assembly code is generated at the last stage. GCC architecture has three clearly-defined blocks: *Front End*, *Middle End* and *Back End*. There is one front end for each programming language. The parser of each language converts source files into a unified tree form, called GENERIC, which is a high-level tree representation. When it finishes, the Front End emits a GENERIC intermediate representation (IR) of the code, which serves as the interface between the front end and the rest of the compiler.

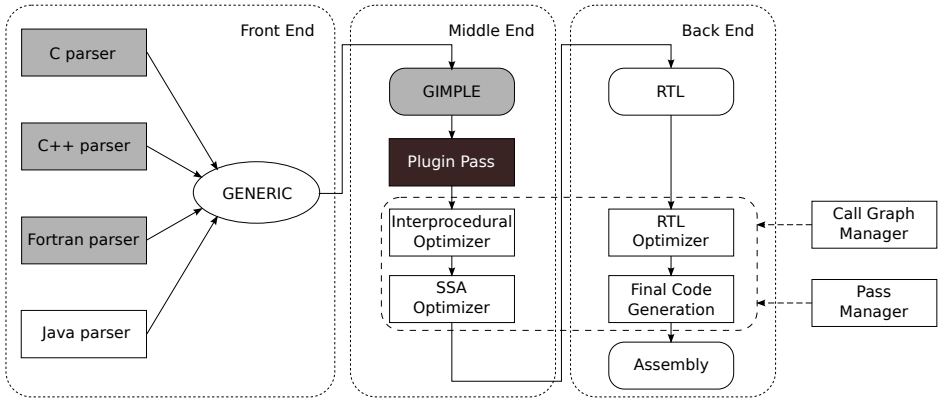


Figure 4.3: GCC Compiler Architecture. The main OpenMP related components, highlighted in grey, are the C, C++ and Fortran parsers, and the GIMPLE IR level. Highlighted in black is the location of the plugin pass which implements ATLaS.

The Middle End works on GIMPLE, which is a 3-address language with no high-level control flow structures. In GIMPLE, each statement does not contain more than three operands (except function calls); control flow structures are combinations of conditional statements and goto operators; and there is a single scope for variables. This kind of representation is convenient to optimize the source code. Once the source code is in GIMPLE form, an *interprocedural optimizer* is called, where *inlining* operations, *constant propagation*, or *static variable analysis* are performed. We have inserted our plugin at this point.

The following step is the transformation from GIMPLE into SSA (Static Single Assignment) representation. In SSA form, each variable is assigned or written only once, creating new versions for each assignment of the same variable, which can be read many times. When different versions of the same variable are written into both branches of a conditional expression, a ϕ-function is added just after the conditional block, allowing the selection of the correct version of the variable, depending on the branch executed. SSA representation is used for several optimizations, such as forward expression substitution, loop interchange, vectorization or parallelization, among others. These optimizations are performed in around 100 passes.

After these optimizations, the SSA representation is converted back to the GIMPLE form, which is transformed into a *register-transfer language* (RTL) form, in which the Back End works on. RTL was the original primary intermediate representation used by GCC. It is a hardware-based representation which corresponds to an abstract machine with an infinite number of registers. GCC also uses this form to perform several optimizations, such as branch prediction or register renaming, in around 70 passes.

Finally, the *Final Code Generation* step of the Back End creates the assembly code for the target architecture (x86, mips, etc.) from the RTL representation.

Transactions between the different phases are sequenced by the *Call Graph* and the *Pass Manager*. The Call Graph Manager generates a call graph for the compilation unit, decides in which order the functions are optimized, and drives the interprocedural analysis. The Pass Manager sequences individual transformations and handles pre- and post-cleanup actions as needed by each pass.

Parsing the new clause

In order to parse the new speculative clause, we have extended the GNU OpenMP (GOMP) compiler, the OpenMP implementation for GCC. The main parts of the GCC architecture related with OpenMP are highlighted in grey in Figure 4.3. GOMP has four main components [188]: parser; intermediate representation; code generation; and the runtime library called `libGOMP`. We have focused on modifying the GOMP parsing phase. The generation of new code to support TLS is located in the plugin developed, and mainly consists of inserting calls to the TLS library functions described in the previous sections.

The parser identifies OpenMP directives and clauses, and emits the corresponding GENERIC representation. We have modified the C parser and the IR to add support for the new speculative clause. First, we have created the GENERIC representation of the new clause as other standard clauses. Then, the compiler has been modified to recognize and parse that clause as part of the parallel loop construct. When the new clause has been parsed, and the IR is generated, our plugin detects the clause and triggers all the transformations needed by the code.

GCC speculative plugin description

GCC plugins provide extra features to the compiler –although they cannot extend the parsed language–, allowing passes to be added, replaced, monitored, or even removed from the GCC compiler without touching the GCC source code. Hence, plugins ease the programming of modifications and contributions to the GCC community. Using this mechanism, our system adds a new pass in the GCC pipeline. This new pass performs all the transformations needed in the code when the programmer marks a variable as speculative.

The new pass is added before the compiler optimization passes, and just before GCC does the first pass in relation with OpenMP: *omplower*. At this point, we have the code in a GIMPLE representation, and the for loop marked with the parallel loop directive preserves all the clauses introduced by the programmer. Therefore, we have the information about which variables are speculative. After this pass, GCC manages speculative variables as shared, while their handling as speculative is carried out by the TLS runtime library.

Figure 4.4 shows a brief example of the transformations made by the plugin. The parser detects the new speculative clause, and the new compiler pass performs automatically all the transformations needed to speculatively parallelize the loop. With the list of variables and data structures that should be speculatively updated, the plugin replaces each read of one of these variables or data elements with a specload() function call. Similarly, all write

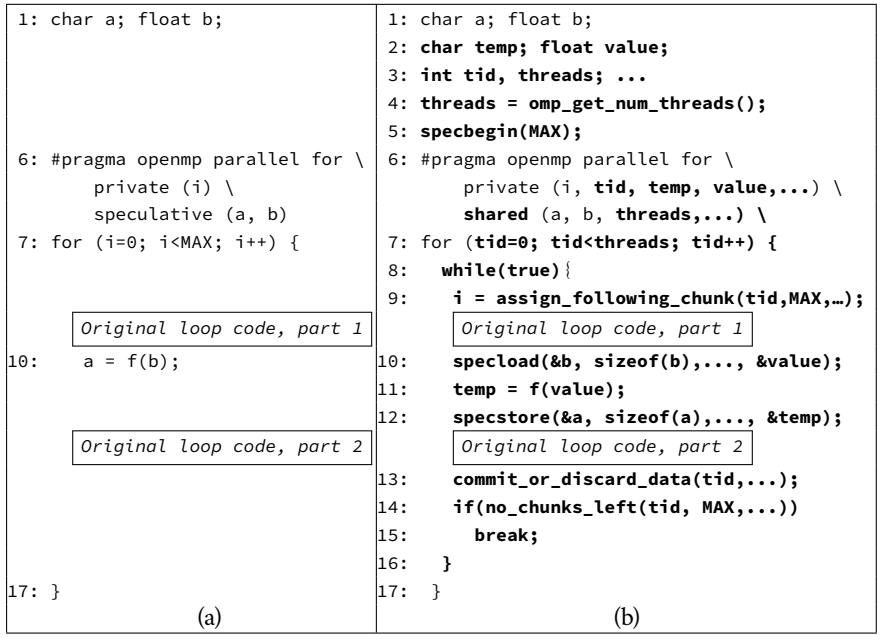


Figure 4.4: Loop transformation to allow its speculative execution: Original (a) and transformed (b) code.

operations to speculative variables are replaced with a specstore() function call. Loads or stores involving other variables do not require additional changes in the code, since all flavors of private and shared variables keep their respective semantics in the context of a speculative execution. The plugin also adds all the structures and functions needed to speculatively parallelize the code. This process is completely transparent to the programmer, who does not need to know anything about the speculative parallelization model. The programmer should only label the variables involved in the target loop as private or shared, as with any other OpenMP program, and mark as speculative those variables that might lead to any dependence violation.

The scheme of the process followed by the plugin can be summarized in the following steps:

1. The plugin traverses each function of the original program looking for an OpenMP parallel loop directive with a speculative clause on it. If the plugin does not find the speculative clause on the pragma, the semantic of the loop remains identical to any other standard OpenMP loop.

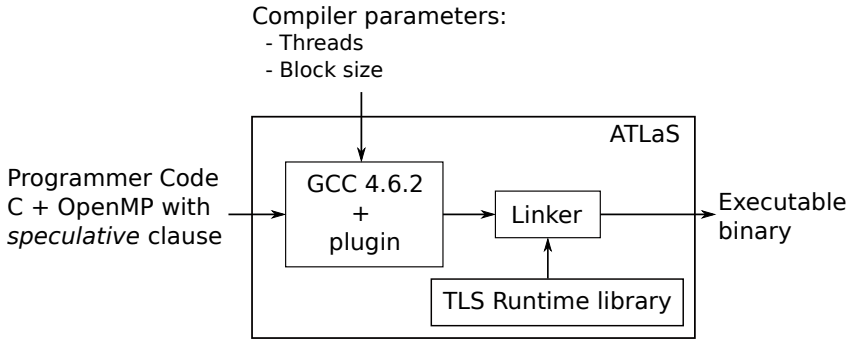


Figure 4.5: Overview of the code generation process for the speculative clause.

2. If the plugin finds the speculative clause, it extracts the speculative variables pointed by the clause, and two functions are added before the loop: omp_set_num_threads(T), where T is the number of threads indicated in the compilation command; and specbegin(N), where N is the number of iterations of the loop.

3. The plugin adds, as private or shared variables, those variables needed by the runtime system. The code generated by the plugin also includes the creation of other new variables, which are also added as private or shared.

4. The plugin adds all the code needed to run the TLS system, including the replacement of the original loop by a new loop that drives the speculative execution.

5. The plugin traverses the GIMPLE nodes of the loop, searching for readings from and writings into the speculative variables. Each read and write are replaced by a specload() and specstore() function, respectively.

Once the plugin has transformed the loop, GCC operation continues with the next passes. When the compilation ends, the resulting binary file is prepared to run speculatively.

Use of the ATLaS framework

To speculatively parallelize a source code with our system, programmers should add the OpenMP directive in the target loop, and classify its variables, according to their usage, into private (and its variants), shared, or speculative. To compile the program, the programmer should also indicate the size of the block of iterations that will be issued for speculative execution, among other minor parameters. With these simple modifications, a programmer can speculatively parallelize a code, while the rest of the transformations needed are transparently performed by the plugin and the compiler. Figure 4.5 summarizes the code generation process performed by the plugin, and the link to the TLS runtime system, which is transparent to the user.

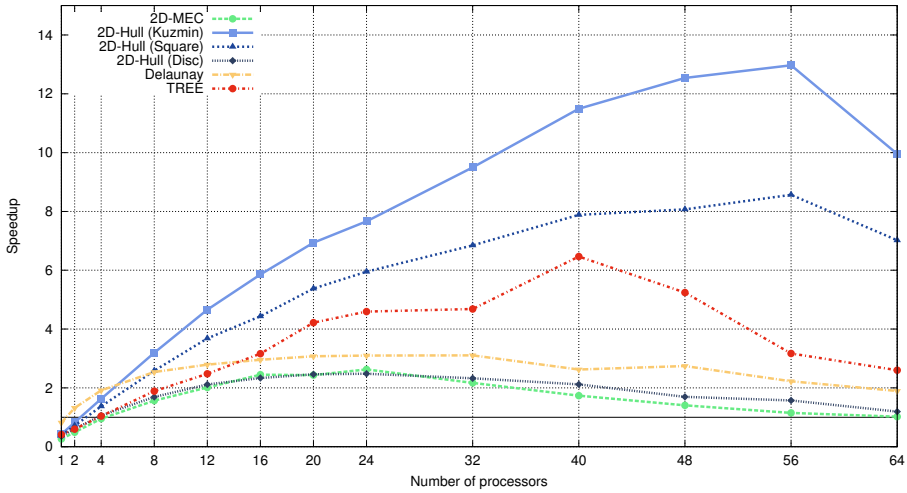


Figure 4.6: Performance achieved by the parallelizable loop of the benchmarks considered.

For further information about the development of this compiler support for the speculative clause see [5, 6, 7, 8, 9, 11].

4.3 Experimental evaluation

Experiments were carried out on a 64-processor server, equipped with four 16-core AMD Opteron 6376 processors at 2.3GHz and 256GB of RAM, which runs Ubuntu 12.04.3 LTS. All threads had exclusive access to the processors during the execution of the experiments, and we used wall-clock times in our measurements. We have used the ATLaS plugin together with gcc for all applications.

4.3.1 Benchmark evaluation

To test the ATLaS framework, we have used both real-world and synthetic benchmarks. The real-world applications include the 2-dimensional Minimum Enclosing Circle (2D-MEC) problem, the 2-dimensional Convex Hull problem (2D-Hull), the Delaunay Triangulation problem, and a C implementation of the TREE benchmark. The synthetic benchmarks are Complete, Tough, and FAST. Benchmarks used are fully described in Appendix A.

Figure 4.6 shows the speedups achieved using the proposed OpenMP speculative clause with the mentioned applications. For the 2D-MEC benchmark, our solution achieves a peak

speedup of $2.6\times$. Although these are not big figures, these results are achieved by simply declaring as speculative the variables that hold the solution found so far.

In the case of 2D-Hull, as described above, results depend on the input set. Performance varies from a $2.4\times$ speedup with the *Disc* input set, which causes a huge number of dependence violations, to a $13\times$ speedup with the *Kuzmin* input set, which leads to fewer violations.

Delaunay's execution produces a high number of dependence violations, which affects the speedup. Delaunay achieves a peak performance of $3.1\times$ speedup.

Finally, TREE obtains a peak of $6.5\times$ speedup. This benchmark is characterized by the presence of reductions over sum and maximum operations that involve speculative variables.

Regarding synthetic benchmarks, let us start with the so-called Complete. While executing this loop in parallel, all the iterations lead to dependence violations. Consequently, the speedup obtained is very poor ($0.03\times$ with 64 processors), but the parallel execution finishes successfully.

Concerning Tough synthetic benchmark, all of its iterations perform a load and a store on the same speculative data structure, with almost no computational load on private variables. This situation adversely affects performance (speedups obtained ranged from $0.1\times$ with 2 processors to $0.002\times$ with 64 processors), even when the number of dependence violations during parallel execution is relatively small (4.46%). Despite of this, the parallel execution is also successful.

With reference to FAST synthetic benchmark, only two of the 180 000 iterations (0.001%) lead to a dependence violation. Note that these two dependences are enough to prevent the compile-time parallelization of this loop. The speculative execution of this benchmark in our shared-memory parallel system returns a maximum speedup of $44.5\times$ with 64 processors. When using 32 processors, the speedup is $29.3\times$, representing an efficiency of more than 90%. These results indicate that the overhead due to the runtime speculative library is negligible.

4.3.2 Effectiveness of the ATLaS runtime library

Table 4.1 summarizes the percentage of time consumed by the target loops of each benchmark, together with an estimation of the maximum speedup obtained (using Amhdahl's Law), and the performance results obtained by our runtime library for the entire application, both in terms of speedup and as a percentage of the maximum speedup attainable. The last two columns indicate the percentage of iterations that lead to runtime dependence violations, and the number of speculative variables. Since all benchmarks but TREE present dependences among some iterations, the value given by Amhdahl's law is just an upper bound of the available parallelism.

The percentage of the speedup effectively exploited depends on a number of factors. The first one is the occurrence of runtime dependence violations. In general, the more dependencies there are, the less speedup there will be. This fact can be observed in the results for the execution of 2D-Hull with different input sets that lead to a different number of

Application	% target loop	Max. speedup P = 64 (Amhdahl)	Maximum speedup obtained	% of exploited speedup	% of iterations of that present dep. violations	# of potentially speculative scalar variables	Size of chunks issued
2D-MEC	43.75	1.76	1.37	**77.84%**	0.009%	10	1 800
FAST	100	64	44.49	**69.52%**	0.001%	2	25
TREE	95.17	15.84	5.12	**32.32%**	0%	259	100
2D-Hull, Kuzmin	100	64	12.92	**20.18%**	0.0008%	1 206	11 000
2D-Hull, Square	100	64	8.47	**13.23%**	0.0032%	3 906	3 000
Delaunay	97.60	25.47	2.96	**11.62%**	0.5%	12 030 060	2
2D-Hull, Disc	100	64	2.48	**3.88%**	0.0219%	26 406	1 250

Table 4.1: Percentages of parallelism effectively exploited by ATLaS for the benchmarks considered, together with some benchmarks' characteristics. I/O time consumed by the benchmarks were not taken into account.

runtime dependencies. The second factor is load imbalance, since not all iterations present the same amount of workload. As long as the scheduling mechanism implemented in ATLaS issues chunks of iterations of fixed size (with the best sizes obtained by experimentation), runtime load imbalance is not being mitigated in any way. The third factor that affects parallel performance is the efficiency of the ATLaS runtime library itself. The performance results obtained with the FAST benchmark, show that the library presents a very low runtime overhead. Finally, the fourth factor is the number of speculative variables. As can be seen, the more speculative variables there are, the less percentage of exploited speedup there will be. This is due to the cost of the commit operation, which should be done sequentially for each variable by the non-speculative thread.

A more detailed analysis of the TLS operations carried out by the library, together with an execution breakdown for speculative loads and stores, can be found in the following chapter. Regarding the influence of the number of squashes in performance, please see [97]. The ATLaS framework incorporates tools to measure these and other values.

To find the most recent version of the ATLaS framework, and the user manual [9] we recommend visiting `http://atlas.infor.uva.es/`.

4.4 Conclusions

In this chapter we have described ATLaS, a framework which allow to automatize the process of parallelizing applications speculatively. Commonly processes of parallelization are hard task, and usually are error-prone. So, we have introduced why might be useful to develop a tool like the developed.

Afterwards we have explained some brief notions about the compiler plugin which uses ATLaS.

The work described in this chapter has generated following publications:

- Sergio Aldea, Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'A New GCC Plugin-Based Compiler Pass to Add Support for Thread-Level Speculation into OpenMP'. English. In: *Euro-Par 2014 Parallel Processing*. Ed. by Fernando Silva, Inês Dutra and Vítor Santos Costa. Vol. 8632. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 234–245. ISBN: 978-3-319-09872-2. DOI: 10.1007/978-3-319-09873-9_20. URL: http://dx.doi.org/10.1007/978-3-319-09873-9_20

- Sergio Aldea, Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Una extensión para OpenMP que soporta paralelización especulativa'. In: *Proceedings of the XXV Jornadas de Paralelismo*. Valladolid, Spain, Sept. 2014. ISBN: 978-84-697-0329-3

- S. Aldea, A. Estebanez, D.R. Llanos and A. Gonzalez-Escribano. 'An OpenMP Extension that Supports Thread-Level Speculation'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, 2015, pp. 1–14. 2015. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2393870

Chapter 5

Scheduling strategies for Thread-Level Speculation

Scheduling is one of the factors that most directly affect performance in Thread-Level Speculation (TLS). Since loops may present dependences that cannot be predicted before runtime, finding a good chunk size is not a simple task. The most used mechanism, Fixed-Size Chunking (FSC), requires many "dry-runs" to set the optimal chunk size. If the loop does not present dependence violations at runtime, scheduling only needs to deal with load balancing issues. For loops where the general pattern of dependences is known, as is the case with Randomized Incremental Algorithms, specialized mechanisms have been designed to maximize performance. To make TLS available to a wider community, a general scheduling algorithm that does not require a-priori knowledge of the expected pattern of dependences nor previous dry-runs to adjust any parameter is needed. In this chapter, we present an algorithm that estimates at runtime the best size of the next chunk to be scheduled. The result is a method with a solid mathematical basis that, using information of the execution of the previous chunks, decides the size of the next chunk to be scheduled. This algorithm simply offers two parameters that need to be adjusted: how optimistically and how pessimistically we increase or decrease the chunk-size, depending on the runtime information of the previous chunks executed so far. Our experimental results show that the use of the proposed scheduling function compares or even increases the performance that can be obtained by FSC, greatly reducing the need of a costly and careful search for the best fixed chunk size.

5.1 Problem description

Thead-Level Speculation (TLS) [49, 190, 218] is the most promising technique for automatic extraction of parallelism of irregular loops. With TLS, loops that cannot be analyzed at compile time are optimistically executed in parallel. A hardware or software mechanism ensures that all threads access to shared data according to sequential semantics. A *dependence violation* appears when one thread incorrectly consumes a datum that has not been generated by a predecessor yet. In the presence of such a violation, earlier software-only speculative solutions (see, e.g. [108, 218]) interrupt the speculative execution and re-execute the loop serially. Subsequent approaches [48, 61, 227] squash only the offending thread and its successors, re-starting them with the correct data values. More sophisticate solutions [97, 167, 239] squash only the offending thread and subsequent threads that have actually consumed any value from it.

It is easy to see that frequent squashes adversely affect the performance of a TLS framework. One way to reduce the cost of a squash is to assign smaller subsets (called *chunks*) of iterations to each thread, reducing both the amount of work being discarded in the case of a squash, and the probability of occurrence of a dependence violation. However, smaller chunks also imply more frequent commit operations and a higher scheduling overhead. Therefore, a correct choice of the chunk sizes is critical for speculation performance. Most scheduling methods proposed so far in the literature deal with independent blocks of iterations, and were not designed to take into account the cost of re-executing threads in the context of a speculative execution.

The rest of the chapter is organized as follows. Section 5.2 reviews some of the classical scheduling alternatives developed for executing loops in parallel. In this way, Section 5.3 completes the previous information focusing on scheduling techniques concerning TLS. Section 5.4 introduces the main aspects of our proposal. Section 5.5 describes the function from a mathematical point of view. Section 5.6 explores two different uses for our Moody Scheduling. Section 5.7 gives some experimental results, comparing the new algorithm with FSC, while Section 5.8 concludes this chapter.

5.2 Classical scheduling alternatives for parallel loops

As described in [174], the problem of scheduling iterations of irregular loops in order to assign them to different processors has been extensively studied in the literature. All existing proposals assume that there are no dependences among iterations, and therefore all the iterations can be executed in parallel in any order. We review in this section some of the solutions that have been proposed in the last years to this problem.

We will describe first the three most well-known techniques to distribute iterations among processors. Let N be the total number of iterations, and P the total number of threads

(equal to the number of processors in the system). The first one, called *static scheduling*, divides the iteration space statically into N/P chunks of equal size. This system does not allow to balance dynamically the workload during the execution of the loops. Hence, the processors may finish at very different times, leading to a poor load balance. On the other hand, *self scheduling* [237] and *dynamic scheduling* change the amount of iterations to execute given to processors from one place to others. These approaches minimize load imbalance, but at the cost of an increase of the scheduling overhead. The main difference between *self* and *dynamic* scheduling is that *self scheduling* arranges iterations before being executed whereas *dynamic scheduling* manage iterations at runtime frequently using parameters of current execution.

5.2.1 Self scheduling

Within self scheduling different alternatives have been proposed. A brief description follows.

Fixed-size chunking (FSC). In this approach, proposed by Kruskal and Weiss [155], the iteration space is statically divided into chunks of fixed size. Each free thread executes the following chunk. This solution reduces synchronization overhead in comparison with self scheduling, with a better load balancing than the static scheduling. The efficiency of this scheme depends on the choice of an appropriate value for the chunk size, K, a difficult task for both programmers and compilers.

Guided self-scheduling (GSS) This technique, proposed by Polychronopoulos and Kuck [204], addresses the problem of uneven start times for each processor. Instead of using a fixed chunk size, they propose decreasing chunk sizes, calculated as a decreasing function of the current iteration number i being executed. As execution proceeds, smaller chunks improve the balance of the workload toward the end of the loop.

In order to avoid having many small chunks by the end of the loop, an additional function GSS(K) is proposed to bound the chunk size from below by K, specified either by the compiler or the programmer.

Factoring. This mechanism, proposed by Hummel et *al.* [119], is similar in concept to guided self-scheduling, but the allocation of iterations to processors proceeds in phases. In each phase, a part of the remaining iterations is divided in batches of P equal-size chunks.

Factoring can be viewed as a generalization of GSS and Fixed-Size Chunking: GSS is factoring where each batch contains a single chunk, while Fixed-Size Chunking is factoring with a single batch.

Trapezoidal scheduling (TSS). This technique, proposed by Tzen and Ni [249], uses chunks that decrease in size linearly. This approach is simpler to implement than GSS and specially GSS(K), thus reducing scheduling overhead. Moreover, according to their authors, a big value of K in GSS(K) leads to a high unbalance, while small values lead to too much scheduling overhead. Consequently, an optimum value of K for GSS is difficult to obtain, particularly in unbalanced loops. By decreasing the chunk size linearly, TSS reduces the number of chunks, and hence the overhead, and simplifies the calculation of the next chunk size, allowing its computation with atomic *Fetch-&-Increment* operations.

Wang et *al.*'s solution. More recently, Wang et *al.* [254] developed a self-scheduling method able to support hardware faults on shared memory systems.

The total number of iterations being scheduled is, at least, N for all scheduling alternatives described. Only Self Scheduling always leads to exact correspondence. Consequently, the scheduler should always check whether the upper limit will be exceeded, and order the execution of only the remaining iterations.

5.2.2 Dynamic scheduling

Finally, dynamic proposals determine the optimum chunk size at runtime, based on the total available parallelism, the optimal grain size and the statistical variance of execution times for individual tasks. Let us reference some of them in order to put into perspective our proposal framed in this group.

The Tapering algorithm. Lucco [175] proposed one of the first approaches which followed a dynamic scheduling of iterations. He was centred on the improvement of GSS methods taking into account some parameters extracted at execution time.

Affinity scheduling. Markatos and LeBlanc [178] proposed the affinity scheduling in which iterations were assigned mainly taking into account processor affinity. Iterations were divided into the available processors up to load imbalance occurred, when idle processors 'stole' some iterations of others. Later, Jin et *al.* [137] improved this algorithm by allowing to change the number of iterations to execute in each chunk. This was one of the first approaches of pure dynamic scheduling. Thus, authors used the number of iterations executed so far in order to increase, or decrease sizes of the following chunks. In the initial phase of their scheduling policy, iterations were divided constantly throughout the available processors. Afterwards, during execution, some of the remaining iterations of each processor are downloaded to a queue allowing other processors to execute them when finishing their work. The number of iterations downloaded changed according to the workload of the processor.

Chen and Guo' solution. Recently, Chen and Guo [42] proposed an enhancement of the static scheduling of OpenMP (based on FSC) which changed static chunk sizes in order to better exploit load balance of loops. Thus, they adapted FSC to a dynamic approach.

Scheduling regarding energy. Scheduling has not only been studied in order to improve performances, but also regarding energy costs. Hence Dong et *al.* [71] adapted scheduling algorithms so as to save some energy.

5.3 Scheduling iterations under TLS

The scheduling method used with speculative parallelization is different from classic scheduling methods, e.g. [110, 155, 249]. Under TLS, the execution of an iteration or chunk of iterations can be discarded, so the scheduling method should be able to re-assign the squashed

iteration to the same or a different thread. The loop structure should be changed to allow re-execution of iterations. Thus, the main concern of scheduling in TLS is avoiding dependence violations.

Some research done regarding scheduling of TLS solutions has just address thread scheduling, e.g. [73, 96, 255], or instruction scheduling, e.g. [268], more than sizes of chunks to be executed. However, since the size of the chunk assigned to each processor directly affects performance in TLS, numerous algorithms have been proposed so as to give a solution to this problem.

The simplest and, according to [192], the most used one is the FSC [155] described previously. Nevertheless, finding the right constant needs several dry-runs on each particular input set for each parallelized loop. When no dependence violations arise at runtime, this technique is perfectly adequate. The only remaining concern is to achieve a good load balance when the last iterations are being scheduled. Some examples of mechanisms that uses this technique can be found in [208] and [129].

Due to the complexity of TLS approaches, most of the TLS papers developed so far are not focused on the scheduling of iterations, and therefore, details of the policy used are brief, e.g. [90], or even unavailable. Thus, we suppose that probably those works used FSC or similar approaches, many of them giving chunks of a single iteration. Following lines contain a brief description of the TLS papers in which some information about the way iterations were scheduled was available or could be deducted. For example, Raman et *al.* [215] gave some notes about a load balancing algorithm which dynamically assigned iterations, but it is not detailed, nor described. Gupta and Nim [108] affirmed that its solution could be easily enhanced with a dynamic scheduler of iterations, but again it is the only information shown.

There are solutions based on compile-time dependence analysis [73, 196, 268]. In these approaches, scheduling decisions are taken by reviewing the possible dependence pattern that can arise, so an in-depth analysis of the loop is needed.

Other approaches rely on the *expected* dependence pattern of the loop to be parallelized. In particular, for Randomized Incremental Algorithms, where dependences tend to accumulate in the first iterations of the loop, two methods have been shown to improve performance. The first one, called Meseta [173], divides the execution in three stages. In the first one, chunks of increasing sizes are scheduled, aiming to compensate for possible dependence violations, until a lower bound of the probability of finding a dependence is reached. From then on, a second stage applies FSC to execute most of the remaining iterations. A third stage gradually decreases the chunk size, aiming to achieve a better load balancing.

The second mechanism is called Just-In-Time (JIT) Scheduling [172]. This method also focuses on randomized incremental algorithms, where dependences are more likely to appear during the execution of the first chunks. JIT Scheduling defines different logarithmic-based functions that issue chunks of increasing size, and relies on runtime information to modulate these functions according to the number of dependence violations that effectively appear.

Kulkarni et *al.* [158, 162] also discussed the importance of scheduling strategies in TLS. These authors defined a schedule through three steps, i.e., three design choices that

specify the behavior of a schedule, namely *clustering*, *labeling* and *ordering*. They tested several strategies for each defined module, using their Galois framework. Their results show that each application analyzed was closely linked to a different scheduling strategy.

Some approaches [214, 251] used Decoupled Software Pipelining to enhance scheduling of iterations. This technique, instead of executing full iterations by the same thread, is based on dividing each iteration into smaller parts and assigning them to the available threads. Thus, threads, ordered forming a pipeline, executed parts of all iterations.

Feng et *al.* [92] in their approach centred on adapt I/O operations to TLS approaches, used the GSS algorithm described in the previous section. Tian et *al.* [241], in their Copy-or-Discard approach, addressed scheduling by unrolling loops to reduce dependences. Oancea et *al.* [190] tried to schedule iterations whose instructions had dependences among them to the same processor, i.e., avoiding the sequential order. To did so, they needed to perform a dependences analysis before parallel executions.

Both [84] and [134] proposals are the most similar scheduling techniques to the developed in this chapter called Moody [83]. Specifically, they increased or decreased the number of iterations executed regarding the runtime parameters that reflected dependence violations produced. However, they were not based on a mathematical basis like ours, and therefore their approaches are different.

In summary, we can conclude that proposed solutions so far either depend on the expected dependency pattern of the loop to be speculatively executed, or require a big number of training experiments to be tuned, as in the case of FSC. In this chapter we present a new mechanism that issues chunks of different sizes, by taking into account the actual occurrence of dependence violations, without using any prior knowledge about their distribution.

5.4 Moody Scheduling: Design guidelines

Our main purpose is to design a scheduling function that is able to predict the best size for the following chunk to be issued at runtime, without the need of a knowledge of the underlying problem. In order to decide the size of the next chunk to be scheduled, we will use the number of times that the last h chunks have been squashed and re-executed due to dependence violations. As an example, Fig 5.1(a) shows, for each scheduled chunk (x-axis), the number of times it has been executed so far (y-axis).

The design guidelines of the Moody Scheduling strategy were developed by Diego Llanos, Belen Palop and David Orden between 2009 and 2014. We will first present these guidelines, to later discuss the implementation issues and the experimental evaluation we carried out.

Given the number of executions of the last h chunks (regardless whether they were already committed or not), we will consider two parameters. The first one is the average number of executions of the last h chunks, which we call meanH and whose value is, at least, 1. The second one is the *tendency* of these re-executions. This value, which we call d, lies in the

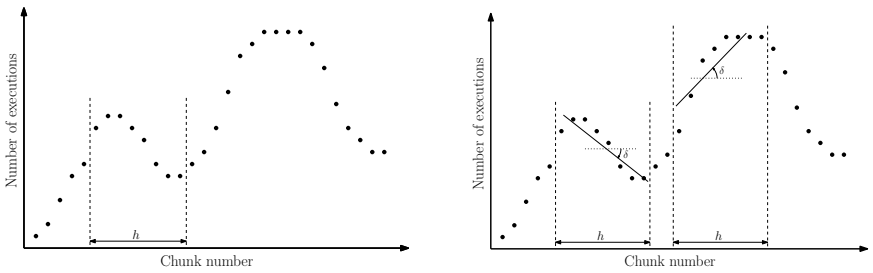


Figure 5.1: (a) A possible execution profile for a given loop, and (b) an example of the use of linear regression to measure the tendency of the last h chunks. Recall that the y-axis does not represent the chunk size, but the number of re-executions for each chunk.

interval $(-1, 1)$ and determines if the number of executions is decreasing ($d < 0$), increasing ($d > 0$), or remaining unchanged ($d = 0$). As we will see, d depends on the angle δ between the linear regression line for the last h chunks and the horizontal axis (see Fig. 5.1(b)).

The size of the following chunk to be scheduled will depend on these two parameters. We will first present an informal description of the idea. The following section shows the mathematical background and the implementation details.

1. If the tendency of re-executions is decreasing (d close to -1):
   (a) If meanH is very low (close to 1), we will (optimistically) set the chunk size to the maximum size suitable for this problem. We will call this maximum value maxChunkSize.
   (b) If meanH is between the minimum value (1) and an *acceptable* value (that we call accMeanH), we will (optimistically) increase the chunk size.
   (c) If meanH is between accMeanH and an upper limit (that we call maxMeanH), we will keep the same chunk size, with the aim that its execution will help to further reduce meanH.
   (d) If meanH is higher than maxMeanH, we set the size of the following chunk to 1.
2. If the tendency of re-executions is stable (d close to 0):
   (a) If meanH is very low (close to 1), then we will (optimistically) issue a larger chunk size.
   (b) If meanH is acceptable (close to accMeanH), then we will keep the same chunk size.

	meanH ≈ 1	meanH $\approx$ accMeanH	meanH $\approx$ maxMeanH	meanH $>$ maxMeanH
$d \to -1$	$\uparrow$	$\nearrow$	$=$	1
$d \approx 0$	$\nearrow$	$=$	$\searrow$	1
$d \to 1$	$=$	$\searrow$	1	1

Table 5.1: Changes on the following chunk sized according to d and meanH parameters.

(c) If meanH is between accMeanH and maxMeanH, then we will (pessimistically) decrease the chunk size.

(d) If meanH is higher than maxMeanH, we set the size of the following chunk to 1.

3. If the tendency of re-executions is increasing (d close to 1):

(a) If meanH is very low (close to 1), then we propose to keep the same chunk size, waiting for the next data to confirm if meanH really gets larger.

(b) If meanH is acceptable (close to accMeanH), then we decrease the chunk size, intending to reduce the number of executions.

(c) If meanH is close to (or higher than) maxMeanH, then we propose a chunk of size 1 intending to minimize the number of re-executions.

The last question is what size we should use to issue the first chunk, where there is no past history to rely on. As we will see in Sect. 5.7, setting this inital value to 1 leads to a good performance in all the applications considered.

Table 5.1 summarizes the behavior of our scheduling mechanism. Using this approach, given the current lastChunkSize and a pair of values (d, meanH) our function will use the guidelines described above to propose a value for nextChunkSize. The following section discusses the implementation details.

5.5 Moody Scheduling function definition

After the informal description presented above, the following step is to define a function that determines the value for nextChunkSize using the current value of lastChunkSize, together with d and meanH. In order to obtain the value of δ, we compute the regression line defined by the last h points in our execution window (see Fig. 5.1(b)).

The main problem with the intuitive behavior described above is that its straightforward implementation (with nested if…then constructs) leads to a discontinuous function. This is not a desirable situation, since the behavior of the scheduling function would drastically change for very similar situations.

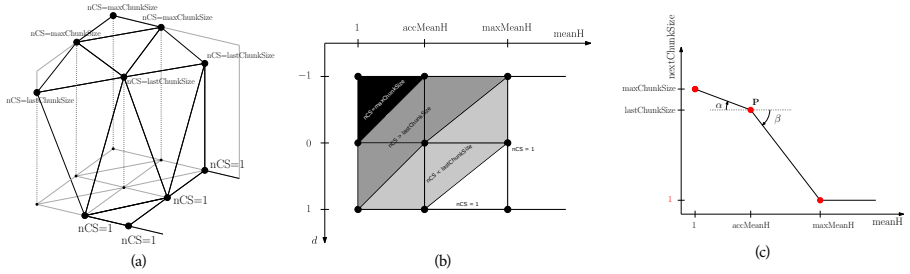


Figure 5.2: (a) 3D representation of the Moody Scheduling function, that returns a value for nextChunkSize (nCS) provided the current lastChunkSize and depending on d and meanH; (b) 2D representation that connects our function with the intuitive behavior described in Sect. 5.4; (c) Intersection of the graphic of nextChunkSize(d, meanH) with $d = 0$.

Instead, we define a bidimensional function that, for a given value of meanH and d, returns the size of the next chunk to be scheduled. Figure 5.2(a) shows a 3D representation of the Moody Scheduling function proposed. Figure 5.2(b) shows its projection onto a horizontal plane, using the same grey scale as in Table 5.1.

To properly define this scheduling function, several parameters should be set. The value of d is calculated by measuring the angle δ of the tendency with respect to the horizontal axis. This angle lies in $(-\pi/2, \pi/2)$. Our growth tendency $d \in (-1, 1)$ will be given by $d = \frac{\delta}{\pi/2}$.

The following parameter to be defined is accMeanH, that is, the highest value of meanH considered to be acceptable. We initially set accMeanH $= 2$, considering that, on average, we will accept that chunks have to be reexecuted at most once.

There are two remaining parameters: maxChunkSize and maxMeanH, whose values depend on the slopes of the graphic of the bidimensional scheduling function as follows. If we fix $d = 0$ in the scheduling function, we obtain the plot depicted in Fig. 5.2(c). In this case, we can define two angles, α and β (see figure). The angle α represents how optimistically the chunk size is going to be increased. The higher the value for α, the most optimistic the scheduling function will be. Analogously, β represents how pessimistically the chunk size is going to be decreased. If we fix the value for these two angles, the value of maxChunkSize is determined by the intersection between the segment from P with angle α, and the vertical line defined by meanH $= 1$. Analogously, the value for maxMeanH is determined by the intersection between the segment from P with angle β, and the horizontal line defined by nextChunkSize $= 1$. In the case that lastChunkSize $= 1$, β will be 0. On the other hand, $\alpha \neq 0$ as long as accMeanH will never be set to 1.

The nine particular points defined by meanH $\in \{1, \text{accMeanH}, \text{maxMeanH}\}$ and $d \in \{-1, 0, 1\}$ are defined by the values described above. Given that the call to the function nextChunkSize(d, meanH) will return maxChunkSize for the three points $(-1, 1)$, $(-1, \text{accMeanH})$, and $(0, 1)$, the function will also return maxChunkSize to all points inside

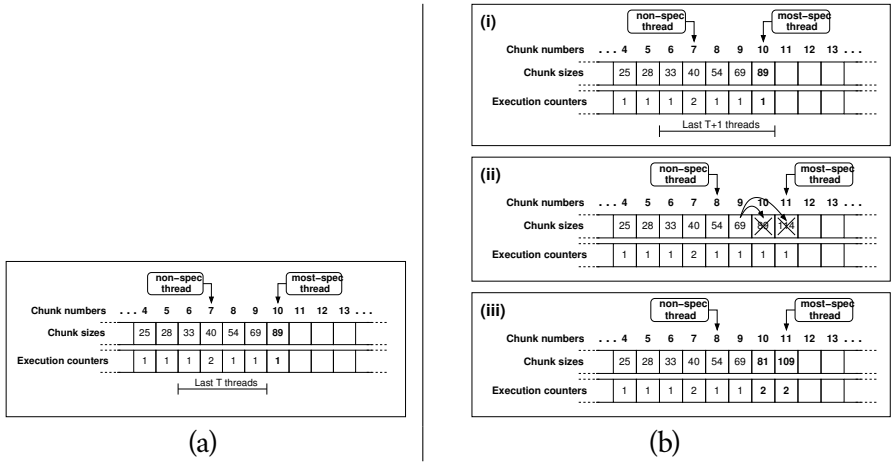


Figure 5.3: (a) Dynamic Moody Scheduling. The size for the following chunk to be executed (#10) is calculated once (89 iterations). Its size will be preserved regardless of the number of re-executions of this chunk. (b) Adaptive Moody Scheduling. (i) Size of chunk #10 is calculated with the Moody Scheduling function (89 iterations). (ii) Chunk #9 issues a squash operation. (iii) Squashed threads recalculate in program order the new sizes of the chunks to be executed, using the new values of the execution counters.

this triangle. Analogously, for all points inside the triangle with vertices $(1, \text{accMeanH})$, $(1, \text{maxMeanH})$, and $(0, \text{maxMeanH})$, the function will return 1. Notice that points on the diagonals $(1, 1)$ to $(0, \text{accMeanH})$, and from there to $(-1, \text{maxMeanH})$ will return lastChunkSize. These three facts provide a natural triangulation for the space in Figure 5.2(b).

5.6 Dynamic and Adaptive Implementations

If no dependences arose during the parallel execution, the size of the following chunk would be calculated only once, that is, just before issuing its execution. Otherwise, if the execution of the chunk fails, it gives the runtime system an opportunity to adjust its calculation by calling the scheduling function with updated runtime information. As it happens in [172], this leads to two different ways to use the scheduling function:

- To calculate the size of the following chunk only the first time this particular chunk will be issued. Subsequent re-executions will keep the same size. See Fig.5.3(a).
- To re-calculate the size of the following chunk each time the chunk is scheduled. This solution is called *adaptive scheduling* in [172]. See Fig.5.3(b).

Application	Input set	Loop to parallelize	Loop time as % of total time	Iterations per call	% of dependence violations	FSC chunk size used (iterations)
TREE	Off-axis parab. collision	accel_10	94	4 096	0	100
2D-Hull	Kuzmin, 10M points	Main loop	99	9 999 997	0.0008	11 000
2D-Hull	Square, 10M points	Main loop	99	9 999 997	0.0032	3 000
2D-Hull	Disc, 10M points	Main loop	99	9 999 997	0.021	1 250
2D-MEC	Disc, 10M points	Inner loop	99	Changes dynamically	0.009	1 800
Delaunay	100K points	Main loop	99	95 000	0.5	2

Table 5.2: Characteristics of the algorithms and input sizes used.

The advantage of adaptive over dynamic scheduling is that the first calculation of the chunk size may rely on incomplete information, since some or all of the previous chunks are still being executed, and therefore they may suffer additional squashes. Adaptive scheduling will always reconsider the situation using updated data. Naturally, this comes at the cost of additional calls to the scheduling function.

5.7 Experimental evaluation

We have used the ATLaS framework (described in Chapter 4) to execute in parallel four different benchmarks (described in Appendix A), specifically, the TREE benchmark; the 2-Dimensional Convex Hull (2D-Hull), with their corresponding three different input sets namely Disc, Square, and Kuzmin; the 2-Dimensional Minimum Enclosing Circle (2D-MEC); and the Delaunay triangulation using the input set of 100K points. Table 5.2 summarizes the characteristics of each application considered.

5.7.1 Environment setup

Experiments were carried out on a 64-processor server, equipped with four 16-core AMD Opteron 6376 processors at 2.3GHz and 256GB of RAM, which runs Ubuntu 12.04.3 LTS. All threads had exclusive access to the processors during the execution of the experiments, and we used wall-clock times in our measurements. Applications were compiled with gcc. Times shown below represent the time spent in the execution of the main loop of the application. The time needed to read the input set and the time needed to output the results have not been taken into account.

5.7.2 Experimental results

Figure 5.4 shows the relative performance of the mentioned applications when executed with the ATLaS speculative parallelization framework [5] and three different scheduling mechanisms: Adaptive Moody Scheduling, Dynamic Moody Scheduling and Fixed-Size Chunking (FSC).

The plots show the performance obtained when an optimum chunk size is used for FSC (a choice that required more than 20 experiments per application) and for Moody Scheduling, whose choice of parameters required less than five experiments in all cases. In the case of Moody Scheduling, we have used a value of 2 for accMeanH, $\beta = \frac{\pi}{4}$, and a value for h (the size of the window to be considered) equal to twice the number of processors for all applications. Regarding α, we have used values $\in (\frac{\pi}{20}, \frac{\pi}{6})$, depending on whether the application is known to produce dependence violations at runtime.

Furthermore, Moody Scheduling turns out to be competitive even without any tuning: If we set to 1 the initial chunk size, its performance reaches 88.3% of the best FSC on geometric average. Meanwhile, the performance of FSC with chunk size 1 drops almost to zero (except for Delaunay, when the best chunk size for FSC is 2).

Regarding 2D-Hull (Figs. 5.4(a), 5.4(b), and 5.4(c)), the results for the Disc and Square input sets show that our scheduling method leads to a better performance than FSC. For the Disc input set, the highest speedup ($2.17\times$) is achieved with 32 processors and the Dynamic version. For the Square input set, the biggest speedup ($6.81\times$) is achieved with the Dynamic version and 40 processors. Finally, the performance figures when processing the Kuzmin input set are similar for all the scheduling alternatives. The best performance ($11.11\times$) is achieved with 56 processors and the Adaptive version. The two remaining applications lead to similar performance results with all the scheduling mechanisms considered. The 2D Smallest Enclosing Circle (Fig. 5.4(d)) achieved a speedup of $2.18\times$ with 24 processors and the Dynamic version. The Delaunay triangulation (Fig. 5.4(e)) achieved a speedup of $2.58\times$ using the Adaptive version. Finally, Fig. 5.4(f) shows the speedup of TREE. This benchmark gained with the use of our scheduling method: With 40 processors, FSC approach achieved its speedup peak, while the Adaptive version continued improving its performance even with the maximum of available processors. It is interesting to note that, for TREE, both the Dynamic and Adaptive mechanisms are equivalent: As long as no squashes are issued, the size for each new chunk is calculated only once. The best performance in this benchmark ($7.96\times$) is obtained with the Adaptive version and 64 processors.

Concerning the relative performance of FSC and Moody Scheduling, both strategies lead to similar performance figures for all applications, with the exception of the TREE benchmark, where Moody Scheduling is clearly better. The main difference between them is that the choice of the optimum block size in FSC required a prior, extensive testing (more than 20 runs per benchmark), while the Moody Scheduling self-tuning mechanism leads to competitive results right from the beginning. Moreover, the results obtained for the TREE application show that, contrary to intuition, our self-tuning mechanism leads to better results than FSC even without dependence violations, despite the higher computing cost added by

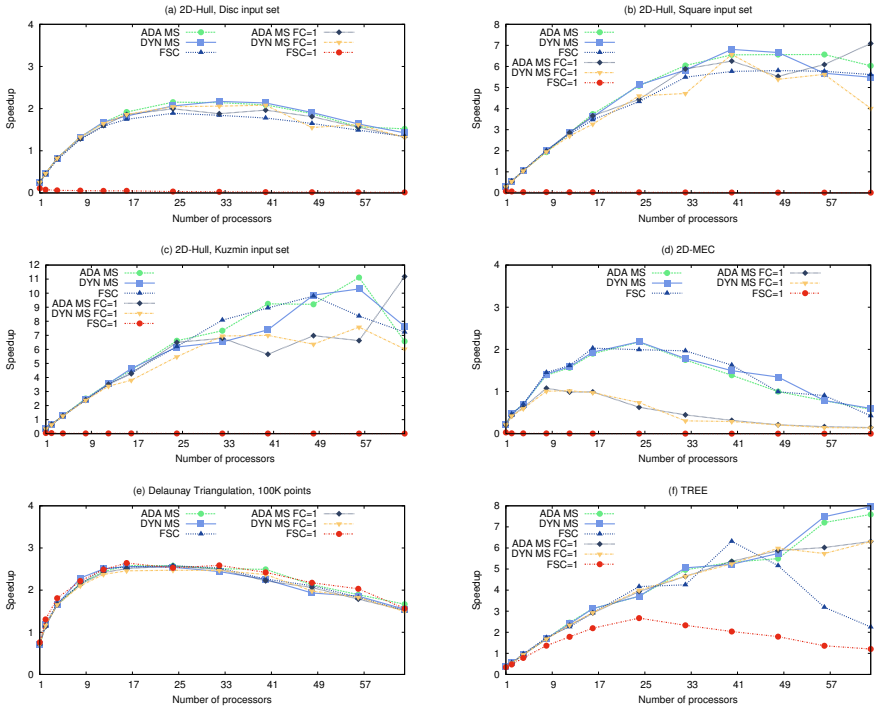


Figure 5.4: Performance comparison for 2D-Hull with Disc, Square, and Kuzmin input sets, and 2D-MEC, Delaunay, and TREE benchmarks. Note the extremely poor performance of FSC when the chunk size is set to 1.

the runtime calls to the Moody Scheduling function. Regarding which approach is better, Dynamic or Adaptive, it seems to depend on the application. Therefore, we will keep both of them in the ATLaS framework.

5.8 Conclusions

This work addresses an important problem for speculative parallelism: How to compute the size of the following chunk of iterations to be scheduled. We have found that most of the existent solutions are highly dependent on the particular application to parallelize, and they require many executions of the problem to obtain the scheduling parameters. Our new method, Moody Scheduling, automatically calculates an adequate size for the next chunk of iterations to be scheduled, and can be tuned further by making slight changes to its parameters, namely α, β, h, and accMeanH. Our scheduling method can be used as a general approach that avoids most of the 'dry-runs' required to arrive to scheduling parameters in other methods. Results show that execution times are similar (or better) to those obtained with a carefully-tuned FSC execution. Moody Scheduling just needs from the user to decide how optimistic, and pessimistic, the TLS system will be when it schedules the following

The work described in this chapter has not only been developed by the main authors of this thesis, but also by David Orden and Belen Palop. With their invaluable collaboration our approach has generated the following publication:

- Alvaro Estebanez, Diego R. Llanos, David Orden and Belen Palop. 'Moody Scheduling for Speculative Parallelization'. English. In: *Euro-Par 2015: Parallel Processing*. Ed. by Jesper Larsson Träff, Sascha Hunold and Francesco Versaci. Vol. 9233. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015, pp. 135–146. *ISBN*: 978-3-662-48095-3. *DOI*: 10.1007/978-3-662-48096-0_11. *URL*: http://dx.doi.org/10.1007/978-3-662-48096-0_11

Chapter 6

ATLaS in the context of emerging architectures: TLS and Transactional Memory

Transactional Memory (TM) is a technique that aims to mitigate the performance losses that are inherent to the serialization of accesses in critical sections. Some studies have shown that the use of TM may lead to performance improvements, despite the existence of management overheads. However, the relative performance of TM, with respect to classical critical sections management depends greatly on the actual percentage of times in which the same data are handled simultaneously by two transactions.

In this chapter, we compare the relative performance of the critical sections provided by OpenMP with respect to not only two Software Transactional Memory (STM) implementations, namely TinySTM and GCC-STM, but also a Hardware Transactional Memory device, i.e., Intel's Haswell chips. These methods are used to manage concurrent data accesses in ATLaS, the software-based, Thread-Level Speculation (TLS) system described throughout this Ph.D. thesis. The complexity of this application makes it extremely difficult to predict whether two transactions may conflict or not, and how many times the transactions will be executed. Our experimental results show that TM solutions produce similar performances with respect to OpenMP critical sections.

6.1 Problem description

Current multicore processors offer an opportunity to speed up the computation of sequential applications. To exploit these parallel technologies, the software needs to be parallelized, that is, transformed in order to correctly distribute the work among different threads. This process usually involves synchronizing accesses to certain memory areas that are shared by the concurrent threads, with the aim of avoiding potential data races. This synchronization is usually performed by using critical sections which protect shared memory structures.

To simplify this process, parallel programming models such as OpenMP [40] offer compiler directives to parallelize the code as well as to synchronize accesses and define and manage critical sections. Despite their simplicity, these solutions present a problem: Critical sections introduce performance losses, not only because they serialize the code, but also because of the cost associated to locking management.

Transactional Memory (TM) [232] arises as a possible solution to the first problem, allowing programmers to transform critical sections in transactions that are concurrently and atomically executed. This is based on the optimistic assumption that the code inside a transaction will access to different locations of the shared memory being protected. In these cases, accesses are carried out concurrently. If this is not the case, conflictive transactions should be rolled back and executed one at a time.

This chapter compares the OpenMP critical sections approach with a Software TM (STM) library as well as a Hardware TM (HTM) architecture. To do so, we use them to implement the critical sections which handle the runtime library of the ATLaS framework detailed in the Chapter 3. Our goal is to study the relative performance of both approaches when managing concurrent accesses in such a complex piece of code.

The rest of this chapter is structured as follows: Section 6.2 describes the fundamentals of TM, describing the most important software- and hardware-based solutions, and also lists some of the existent approaches which connect TM with TLS. Section 6.3 compares both TM and TLS approaches. Section 6.4 details how critical data structures are protected in the ATLaS runtime library to ensure correctness, and enumerates the critical sections used in it. Section 6.6 describes how this protection can be ensured using OpenMP and TM, and details the time spent inside critical sections for each approach in the TLS library. Section 6.7 shows the performance results obtained in a system with 8 cores and HTM extensions. Finally, Section 6.8 concludes this chapter.

6.2 Background

Since TLS has been described well enough throughout this Ph.D. thesis, we are just going to describe TM in this section.

6.2.1 Transactional Memory in a Nutshell

Transactional Memory (TM) [114, 115, 163] is another method of optimistic execution. The main difference between TLS and TM is that TM does not have to preserve any order among execution threads whereas TLS does (in the following subsection these differences are explained deeply). TM main concepts come from database systems where transactions handles all the operations guaranteeing that concurrent accesses follow ACID properties [106]. According to memory semantics, TM does not have to keep durability, however, it partly[1] meets atomicity, concurrency, and isolation standards to coordinate concurrent threads of shared memory systems.

When a conflict appears among certain transactions, TM systems are responsible to abort all but one of the transactions. Then, the discarded transactions are restarted while the other is committed, avoiding inconsistent states. As occurs with TLS, TM approaches can be also implemented using either hardware or software.

In order to improve its procedure, let us regard the example seen at figure 3.2 with a TM perspective. Let us assume that the code depicted is enclosed in a transaction. Remark that now non-speculative or most-speculative labels have no sense since TM approaches imply no order among threads. At the beginning Thread 1 started its transaction at t_1, finishing it at t_3 with no other thread using shared resources. Afterwards, Thread 3 did the same operation with similar results. Then, the first conflict appeared at t_8 when Thread 3 started its transaction using the shared resource `SV`. Before Thread 3 finished, Thread 4 had started its own transaction working with `SV` at t_7. A TM system should deal with such a situation discarding the results of Thread 3, and when Thread 4 release the resources, Thread 3 should be retried. As can be seen, a TM execution would not produce the same results of TLS. While a TLS system would have discarded results of Thread 4, a TM one would have discarded results of Thread 3.

6.2.2 Brief review of software TM libraries

STM simulates transactions using software libraries. As described in [114] the main purpose of these libraries is to provide separate per-thread views of the heap as transactions execute, and a mechanism for detecting and resolving conflicts between transactions.

Works such as [261] have shown that STM can outperform OpenMP critical sections, despite the relatively high overheads of STM. However, the relative performance of STM versus OpenMP critical sections is highly dependent on the running profile of each particular application. Different patterns of accesses to the same critical section may lead to different performance figures.

There exist different software TM (STM) proposals, although some of them have a limited scope of applicability. A brief review of STM libraries follow.

[1] According to [163] ACI model is just partially applicable to TM because TM models have to interact with codes outside of transactions, whilst database systems execute every single operation as a transaction.

- **IBM XL C/C++ Compiler [121].** Although the compiler is available for different operating systems, the transactional built-in memory functions are only valid for Power8 architecture and Blue Gene/Q.
- **Intel C++ STM Compiler [123].** It is not currently possible to be used, because the project was retired in 2012, and licenses are not longer available.
- **GCC-TM C/C++ Compiler [222].** Experimental feature available since version 4.7. Part of the work done in this branch of GCC belongs to the VELOX project.
- **TinySTM [86].** This STM library is compatible with GCC-TM and DTMC (Dresden TM Compiler) [47] from the VELOX project [88]. This runtime library can also be used independently of a compiler, gaining control in the parallelization process, but increasing the required effort.
- **SUN TL2 STM [67].** It is available to be used with the STAMP benchmark suite [36], and it was originally designed by the "Scalable Synchronization" SUN's project. The parallelization process is similar to TinySTM's; in fact, it is possible to replace both libraries in the STAMP benchmark by simply changing a link in the corresponding Makefile.

6.2.3 Transactional Synchronization Extensions

The simplest hardware TM (HTM) solutions only require to modify the cache consistency protocols and complementing the instruction set architecture with a number of new instructions[114]. These modifications allow to manage speculative states.

Intel provides the Transactional Synchronization Extensions (TSX), an extension to the x86 instruction set that adds hardware transactional memory support to some of their microprocessors. TSX extension allows programmers to potentially improve the performance of lock-protected critical sections by enabling concurrent threads to execute the same critical section. Intel TSX detects conflicts at L1 cache granularity, hence concurrent threads can update shared resource locations without leading to transactional aborts as long as these threads do not update the same L1 cache line. However, the progress of a transaction is not guaranteed. HTM are based on a "Best-effort" model , which means that a transaction may always abort. This is why it is necessary that either the source code, or the underlying library enabling TSX, implements a fallback execution path. The simplest fallback technique consists in acquiring a lock when the transaction does not succeed, whilst in the transaction mode the lock is elided. This technique is called "lock elision". Therefore, TSX-enabled locks are not fully acquired, but elided: the lock is only read and watched, but not written to.

The Intel TSX feature can be implemented by using different alternatives. On the one hand, Intel defines two software interfaces: Hardware Lock Elision (HLE), and Restricted Transactional Memory (RTM). HLE targets legacy hardware, being the code paths for both transactional and non-transactional execution the same. In transactional mode, the program

will try first to complete the transaction, atomically committing on memory operations if it successes. If the transaction aborts, all the memory operations would be rolled back in the buffer, and the program will execute the code path in non-transactional mode, acquiring the corresponding lock. This operation is completely different in RTM: the code paths are different for the transactional and non-transactional modes, the so-called fallback path. Compilers such as Intel and GCC provides HLE and RTM intrinsics to enable Intel TSX.

On the other hand, the OpenMP library shipped with the Intel C++ compiler provide a useful functionality: Traditional critical sections protected by OpenMP explicit locks[2] can be enabled with an Intel TSX code path. Therefore, using these OpenMP explicit locks, any application can be run in transactional mode, only setting the environment variable KMP_LOCK_KIND=adaptive.

6.2.4 TLS-TM hybrid approaches

So far, transactions have been combined with TLS in two ways, (1) using transactions to guarantee sequential semantics of loops, and (2) speculating over large transactions to be executed in parallel. Regarding the first approach, [107] proposed *LogSPoTM*, a hardware-based solution that enhanced a TM system called *LogTM* [185]), to ensure sequential semantics and give support to TLS. *LogSPoTM* was mainly based on the integration of timestamps and arbitration policies to impose an order among threads and preserve semantics. [62] improved this solution with the help of a hardware value predictor. [60] also used LogSPoTM in their work. They tried to reduce squashes applying some of the ideas of network theories to speculation. This work considered that thread data were packets, and assigned higher priority to those belonging to predecessor threads. This solution helped to reduce both timeouts and the number of squashes.

Ceze et *al.* tried to improve the way data dependences are managed in the speculative context of TLS and TM threads with *Bulk* [38]. To do so, they used signatures to encode the needed information related to the variables involved in speculative executions and each corresponding operations.

Bulk and *LogSPoTM* are both fully based on hardware, so their operations are driven by the hardware signals of a specific multiprocessor.

Other approaches are focused on decreasing transactions' contention with the use of TLS. *STMLite* [180] is a software TM mechanism which uses TLS so as to reduce the overheads of accesses to logs of the variables used in transactions. Some approaches [19, 147, 214, 250] based their ideas on Multi-threaded Transactions (MTXs) in which threads were speculatively executed for a single transaction, thus speculating inside each transaction. These works are mainly based on changes to the cache coherence protocol. Note that [250] is based on hardware, while *SMTX*[214], *DSMTX*[147], and *TLSTM*[19] are software-based. This approach was also followed in [206] whose authors also used TLS to improve the performance of their hardware-based TM architecture.

[2] omp_set_lock() and omp_unset_lock()

6.3 Comparison of TM and TLS

Speculative parallelization techniques are runtime-based solutions that aim to solve many of the problems and limitations of the compile-time parallelization techniques. Speculative solutions include Transactional Memory (TM) approaches [115, 118, 163], and Thread-Level Speculation (TLS) [217, 244]. Although both TM and TLS are speculative parallelization techniques, they have some differences and cannot be directly applied for the same purposes.

TM is suitable in those situations that do not require to maintain the sequential semantics of the operations performed by the involved threads during the parallel execution. Typical applications are codes that can be divided into independent fragments that can be executed in any order, such as independent transactions. For this kind of applications, TM reduces the costs of the required locks to avoid race conditions. To do so, some TM approaches propose language extensions and new constructs to declare a *transactional block* which comprises statements that must be executed atomically. In order to parse these new constructs, it is necessary to implement or modify an existing compiler. To declare a transactional block, TM libraries rely on different alternatives, such as new constructs (e.g. GCC-TM's `__transaction_atomic{}`, Intel's `__tm_atomic{}`, or the more generic `__transaction{}`), new compiler directives (such as IBM's `#pragma tm_atomic{}`) or even new OpenMP pragmas, such as `#pragma omp transaction{}`, defined by OpenTM [17]. The latter is the closest approach to ATLaS [5] from the syntactic point of view, defining a new OpenMP clause to handle dependence violations.

It is important to highlight that the goals of TLS and TM are different. While TLS is intended to automatically parallelize sequential code, TM libraries aim to improve the performance of already parallelized codes, typically using Pthreads [31]. As long as the use of TM does not guarantee the order in which threads make their commits, they cannot be used directly to mimic the behavior of loop-based speculative parallelization whenever sequential semantics should be preserved.

6.4 Critical sections in ATLaS

One of the key advantages of the ATLaS runtime library over previous designs is that ours is almost free of critical sections. The only critical section needed is the one that manages the data structure which maintains the assignment of chunks of iterations to each thread. ATLaS handles the parallel execution of each chunk of iterations through a sliding window mechanism, which is implemented by a matrix with W columns representing W window slots. Figure 6.1 depicts a simplified version of the sliding window implementation. The figure represents a sliding window with four slots, hosting the execution of three parallel speculative threads.

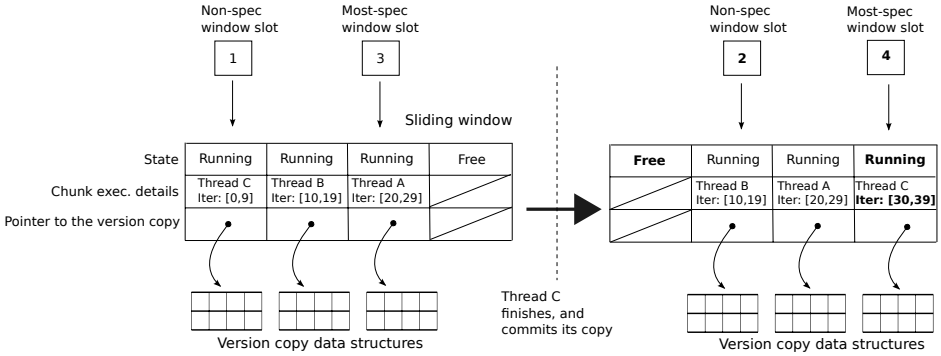


Figure 6.1: Updating the sliding window that handles the parallel, speculative execution. At a given moment (left), the thread C working in slot 1 is running. When Thread C finishes, it frees its slot and gets a new one, updating non-spec and most-spec pointers (right).

The thread executing the earliest chunk of iterations (Thread C in our example) is called *non-speculative*, since it has no predecessors that may squash it. Conversely, the thread executing the latest chunk is called the *most-speculative* thread. As can be seen in Figure 6.1, two pointers indicate the slots where the non-speculative and most-speculative threads are being executed. The part of the window being used is always the one from the non-spec pointer to the right, up to the most-spec pointer.

The only critical section in the ATLaS runtime library is the one that protects this sliding window. If two or more threads finish at the same time, they could be assigned to the same `Free` slot, resulting in an incorrect execution. Therefore, in order to ensure the correct operation of the ATLaS runtime library, it is necessary to protect the accesses to these shared structures, including the matrix that implements the sliding window mechanism, and the variables that point to the non- and most-speculative slots.

Figure 6.1 shows what happens when a non-speculative thread successfully finishes its execution. Suppose that Thread C, the one executing the non-speculative thread, finishes its execution and commits its data (the commit operation is not shown in the figure). After this, it enters the critical section to perform several actions. It marks slot 1 as Free; it advances the non-speculative pointer to slot 2; after checking that the slot past the most-speculative one is Free, it assigns it to itself, setting the most-speculative pointer to 4 and changing its state to Running; and finally, after getting the following chunk of iterations to be executed (iterations 30 to 39 in our example), it exits the critical section. Note that the implementation of the sliding window works in a circular way: When Thread B eventually finishes, it will assign itself the slot that follows the one used by Thread C, in our case the leftmost slot.

6.4.1 Location

The sliding window is modified in three different locations within the ATLaS runtime library. Therefore, the same lock is used in three different parts of the code to protect the access to these data structures. As will be seen, the place from where the access is performed has a noticeable impact in the performance of the protecting system being used. These places are the following:

- **(A) Each time a dependence violation is detected.** In the case of a write to a speculative variable, the thread in charge should update its version copy, and check whether a successor has consumed an outdated value of this variable. If this is the case, a *dependence violation* has happened, so the offending thread should be restarted in order to consume an updated version of the variable. This is done in several steps. First, the thread that has detected the situation should enter the critical section to change the state of the offending thread, from `Running` to `Squashed`, and the most-speculative pointer should be moved backwards to the last Running thread. After these changes, the thread exits the critical section and resumes its normal operation. The offending squashed thread will eventually discover its new state and will enter the critical section (see below).

- **(B) Each time a thread finishes its work,** either because the chunk has been successfully executed or because the thread discovers that it has been squashed. In both cases, the thread enters the critical section to change its own state from Running (resp. Squashed) to Free. After this operation, if the slot following the most-speculative one is Free, the thread assigns it to itself, and advances the most-speculative pointer by one. Otherwise, it means that the following slot is occupied either by a Running thread (this means that the window is full) or by another Squashed thread. In both cases our thread should exit the critical section and attempt to re-enter again, in order to give the thread that is using the slot the opportunity to free it[3] (see below).

- **(C) Each time a thread should wait for a free slot.** If a thread is not able to get a free window slot to work, because the following slot is not Free yet, it should get out and try to gain access again to the critical section to assign itself the following slot and to advance the most-speculative pointer.

6.5 Benchmarks used

To perform the experiments, we used both real-world and synthetic benchmarks. The real-world applications include the 2-dimensional Minimum Enclosing Circle (2D-MEC),

[3] Our thread cannot simply wait inside the critical section, because it should get out in order to let the thread using that slot to get in and change its own state.

the 2-dimensional Convex Hull problem (2D-Hull), the Delaunay Triangulation problem, and a C implementation of the TREE benchmark. We have also used a synthetic benchmark called Fast, which presents almost no dependences between iterations, and which was designed to test the overheads of the ATLaS runtime library. All these benchmarks are detailed in Appendix A, so here they are just going to be briefly introduced.

The Fast benchmark was designed to test the efficiency of the speculative scheduling mechanism, with few iterations leading to a dependence violation, although they are enough to prevent a compiler from parallelizing the loop. This benchmark has very few dependence violations, so the critical section is primarily accessed to get the following chunk of iterations to be executed (access of type B in our library).

Unlike the rest of the benchmarks, TREE does not suffer from dependence violations, but it is still not parallelizable at compile time because the compiler is not able to ensure that there are no data dependencies. Since it does not present dependence violations, the code that accesses the critical section is primarily B.

The 2D-MEC benchmark is a tricky code which has only 10 speculative variables that are frequently accessed. This benchmark calls the speculative loop many times with a very different number of iterations each time, making threads access the sliding window system frequently to get the following chunk. As long as it presents some dependence violations, the critical sections are accessed by codes A, but mostly B and C.

The overheads are even more noticeable for the Delaunay problem, the benchmark with the highest number of dependence violations (0.5%). Critical sections are accessed quite frequently, because of the detection of dependence violations (code A), the need to schedule the execution of many chunks (code B), and some degree of load imbalance which frequently makes the window to be full, leading to contention (code C). In order to ease these factors, we set the smallest optimum chunk size (just two iterations), making the need to schedule the execution of new chunks the main issue, instead of the others. However, the stress on the exclusive access management is so high that, in fact, the TinySTM library is not able to properly handle the accesses to the protected sliding window when running this problem with two or more threads. Hence, we cannot present performance figures for this library in this case.

A similar issue occurs with the execution of the 2D-Hull problem with different input sets. We have found that runs of the 2D-Hull problem, with datasets whose execution involves a larger amount of conflicts and dependence violations (as happen when the Square and Disc input sets are used), do not finish when using TinySTM, regardless of the number of parallel threads. TinySTM also fails when using 2 threads and the Kuzmin dataset, producing different, unexpected outcomes on each execution. This benchmark leads to accesses of the three types, but as occurs with the others, the most expensive critical section is the B.

because of the detection of dependence violations (code A), the need to schedule the execution of many chunks (code B), and some degree of load imbalance which frequently makes the window to be full, leading to contention (code C).

	A		B		C	
Application	**# accesses**	**%**	**# accesses**	**%**	**# accesses**	**%**
FAST	2	0.03	7 204	99.61	26	0.36
TREE	0	0.00	31 529	99.73	86	0.27
2D-MEC	2 236	5.81	28 864	75.02	7 375	19.17
2D-Hull, Kuzmin	74	5.20	1 091	76.67	258	18.13
2D-Hull, Square	329	5.61	4 285	73.01	1 255	21.38
2D-Hull, Disc	1 557	6.63	14 436	61.44	7 504	31.94
Delaunay	276	0.06	498 693	99.55	1 986	0.40

Table 6.1: Number of accesses to each protected zone (results obtained from the average values of three executions). Each execution used the 8 threads of a 8-core machine, and was compile using GCC.

Application	**% target loop**	**Max. speedup P = 64 (Amhdahl)**	**% of iterations that present dep. violations**	**# of potentially speculative scalar variables**	**Size of chunks issued**	**Critical Sections accessed**
FAST	100	64	0.001%	2	25	A, **B**
TREE	95.17	15.84	0%	259	100	**B**
2D-MEC	43.75	1.76	0.009%	10	1 800	A, **B**, C
2D-Hull, Kuzmin	100	64	0.0008%	1 206	11 000	A, **B**, C
2D-Hull, Square	100	64	0.0032%	3 906	3 000	A, **B**, C
2D-Hull, Disc	100	64	0.0219%	26 406	1 250	A ,**B**, C
Delaunay	97.60	25.47	0.5%	12 030 060	2	**B**

Table 6.2: Percentages of potentially parallelism for the benchmarks and loops considered, together with some benchmarks' characteristics.

Table 6.1 shows the number of accesses obtained to each critical section. As can be seen, generally the zone A (due to the detection of dependence violations) takes few accesses, achieving a 6.63% of the total accesses in the highest case (2D-Hull, Disc). The zone C (some degree of load imbalance which frequently makes the window to be full, leading to contention) requires from some more accesses, needing a 31.94% of the total accesses in the highest case (2D-Hull, Disc). Finally, the zone B (due to the need to schedule the execution of many chunks) gets most of the exclusive accesses used in the execution of all applications tested, achieving a 99.73% of the total accesses in the highest case (TREE). So we can affirm that the need to schedule the execution of many chunks is the main bottleneck regarding critical sections (note that Delaunay required almost 500 000 accesses on average).

Table 6.2 summarizes the characteristics of each benchmark, including the percentage of execution time consumed by each target loop, an estimation of the maximum speedup attainable (applying Amhdahl's Law), the percentage of iterations of the target loop that lead to runtime dependence violations, the number of speculative variables within the loop, and the size of the chunk of consecutive iterations speculatively executed. I/O time consumed by the benchmarks were not taken into account. We also give an indication of which accesses to the sliding window protected by the critical section are more frequent in the benchmark (bold letters indicate that the corresponding call is more frequent).

6.6 Protecting data accesses: OpenMP *critical* vs. TM

The original TLS runtime library uses the OpenMP critical directive to guarantee exclusive access of the threads to the three parts of the code mentioned above. Because the same data structures are accessed from three different places, the same lock is used to protect them in all cases. Recall that a block of code marked with an OpenMP critical directive is only executed by one thread at a time, whilst the rest of the threads that have reached the same point in the code have to wait. This procedure ensures that the sliding window is always in a consistent state, thus avoiding multiple threads concurrently updating this structure with the potential loss of consistency.

It is easy to see that the serialization of operations described above should imply a noticeable overhead in the performance of the speculative runtime library. A possible way to reduce this performance penalty would be to replace the strict, OpenMP critical construct with the more optimistic constructs that offer the Transactional Memory paradigm. The goal of TM is precisely to help in explicit parallel programming by reducing the costs of the locks required to avoid race conditions in critical sections [19, 39]. While OpenMP critical constructs only allows one single thread at a time inside the critical section, a transactional-based implementation allows several threads inside it, permitting their concurrent execution as long as consistency is not compromised.

However, the optimism of TM and TLS, comes at the cost of some overheads, because of the extra instrumentation needed to handle the transactions, as well as the cost associated to the extra runs of particular transactions when a conflict appears. As can be seen, both OpenMP and TM approaches to protect data integrity have identified overheads. It is extremely difficult to predict which approach will be better for a particular problem, since it depends on the application, its running profile, and how often the benchmark accesses the potentially conflictive shared variables, among other factors.

Regarding the programmability, OpenMP has been designed to simplify, to a great extent, the process of parallelization, while the direct use of TM approaches involves a non-trivial instrumentation of the source code, from the definition of the transactional region to monitoring each access to speculative variables. This effort is mitigated by the

existence of TM solutions that rely on the compiler to replace TM constructs with calls to the TM library or directly to the subjacent hardware. Some STM approaches propose language extensions or new constructs to declare transactional code regions that comprises statements that must be executed atomically. Then, either an ad-hoc compiler, or an existing compiler modified for this purpose, parses these new constructs, and generates all the instrumentation, in the same way as compilers process OpenMP constructs.

As we said above, OpenMP allows the user to delimit the critical sections with the construct `omp critical`. To declare a transactional region, STM libraries rely on different alternatives, such as new constructs (e.g. GCC-TM's `transaction_atomic{}` [222], the Intel's `tm_atomic{}` [123], or the more generic `transaction{}`), new compiler directives (such as IBM's [121] `tm_atomic{}`), or even new OpenMP pragmas, such as `omp transaction`, defined by OpenTM [17]. Unfortunately, Intel STM compiler and OpenTM are not currently available, while the IBM compiler's transactional built-in memory functions are only valid for Power8 architecture and Blue Gene/Q.

In this work, we have used the Intel OpenMP implementation, the Intel HTM extensions, the GCC OpenMP implementation, the GCC-TM, and the TinySTM libraries [86, 87] to protect the accesses to the sliding window described previously. These five approaches simplify the parallelization process with the mentioned constructs and directives. Moreover, GCC-TM defines a specification for transactional language constructs that other STM libraries can leverage, and hence, changing the underlying STM library is just a process of proper linking. In fact, TinySTM is compatible with GCC-TM, allowing programmers to use the same interface and save some programming effort. In addition, it is important to say in order to understand the experimental results that if the GCC-TM detects a hardware transactional support on the machine, it will use it instead of executing software transactions. Hence, in this work, we have used just the hardware approach.

Handling the critical sections with OpenMP is straightforward: The programmer should simply delimit the region by using the defined `omp critical` directive. This process is similar when using the GCC-TM specification. However, to ensure that the transaction is atomically executed, there may be certain functions inside the transaction that must *not* be executed. Since the compiler is not able to detect this issue for the functions called within a transaction, it is also necessary to annotate their declaration and specify whether they are safe to be called, with the `transaction_safe` attribute.

Table 6.3 details the time used by critical sections using GCC HTM and GCC omp critical. A similar comparison is done in table 6.4 where Intel HTM and ICC omp critical are exposed. Similar results with the rest of approaches will be found at tables 6.5, 6.6, 6.7, and 6.8.

Furthermore tables 6.9 and 6.10 show the average time in seconds spent by a thread within critical sections. They show a four threads execution, and another with eight threads respectively. These tables are supplemented with the figure 6.2. This figure has got depicted the accumulated time used inside critical regions by every thread, both with four and eight threads.

	4 threads			8 threads		
Application	**(A) GCCHTM**	**(B) GCCOMP**	**A÷B**	**(A) GCCHTM**	**(B) GCCOMP**	**A÷B**
FAST	0.0011	0.0009	1.2222	0.0010	0.0006	1.6667
TREE	0.0262	0.0152	1.7237	0.0209	0.0092	2.2717
2D-MEC	0.0062	0.0033	1.8788	0.0100	0.0033	3.0303
2D-Hull, Kuzmin	0.0035	0.0017	2.0588	0.0037	0.0014	2.6429
2D-Hull, Square	0.0061	0.0033	1.8485	0.0077	0.0029	2.6552
2D-Hull, Disc	0.0108	0.0053	2.0377	0.0244	0.0072	3.3889
Delaunay	0.4993	0.2662	1.8757	0.3635	0.1648	2.2057

Table 6.3: Comparison of the time in seconds, on average, spent by each thread within critical sections using GCC HTM and GCC omp critical.

	4 threads			8 threads		
Application	**(A) IntelHTM**	**(B) ICCOMP**	**A÷B**	**(A) IntelHTM**	**(B) ICCOMP**	**A÷B**
FAST	0.0008	0.0012	0.6667	0.0005	0.0007	0.7143
TREE	0.0130	0.0150	0.8667	0.0078	0.0086	0.9070
2D-MEC	0.0030	0.0044	0.6818	0.0039	0.0048	0.8125
2D-Hull, Kuzmin	0.0018	0.0019	0.9474	0.0019	0.0034	0.5588
2D-Hull, Square	0.0033	0.0035	0.9429	0.0032	0.0043	0.7442
2D-Hull, Disc	0.0052	0.0058	0.8966	0.0078	0.0083	0.9398
Delaunay	0.2264	0.2170	1.0433	0.1365	0.1502	0.9088

Table 6.4: Comparison of the time in seconds, on average, spent by each thread within critical sections using Intel HTM and ICC omp critical.

	4 threads			8 threads		
Application	**(A) IntelHTM**	**(B) TinySTM**	**A÷B**	**(A) IntelHTM**	**(B) TinySTM**	**A÷B**
FAST	0.0008	0.0037	0.2162	0.0005	0.0027	0.1852
TREE	0.0130	0.0708	0.1836	0.0078	0.0513	0.1520
2D-MEC	0.0030	0.0112	0.2679	0.0039	0.0121	0.3223
2D-Hull, Kuzmin	0.0018			0.0019		
2D-Hull, Square	0.0033			0.0032		
2D-Hull, Disc	0.0052			0.0078		
Delaunay	0.2264			0.1365		

Table 6.5: Comparison of the time in seconds, on average, spent by each thread within critical sections using Intel HTM and Tiny STM.

	4 threads			8 threads		
Application	**(A) GCCHTM**	**(B) TinySTM**	**A÷B**	**(A) GCCHTM**	**(B) TinySTM**	**A÷B**
FAST	0.0011	0.0037	0.2973	0.0010	0.0027	0.3704
TREE	0.0262	0.0708	0.3701	0.0209	0.0513	0.4074
2D-MEC	0.0062	0.0112	0.5536	0.0100	0.0121	0.8264
2D-Hull, Kuzmin	0.0035			0.0037		
2D-Hull, Square	0.0061			0.0077		
2D-Hull, Disc	0.0108			0.0244		
Delaunay	0.4993			0.3635		

Table 6.6: Comparison of the time in seconds, on average, spent by each thread within critical sections using GCC HTM and Tiny STM.

	4 threads			8 threads		
Application	**(A) GCCOMP**	**(B) TinySTM**	**A÷B**	**(A) GCCOMP**	**(B) TinySTM**	**A÷B**
FAST	0.0009	0.0037	0.2432	0.0006	0.0027	0.2222
TREE	0.0152	0.0708	0.2147	0.0092	0.0513	0.1793
2D-MEC	0.0033	0.0112	0.2946	0.0033	0.0121	0.2727
2D-Hull, Kuzmin	0.0017			0.0014		
2D-Hull, Square	0.0033			0.0029		
2D-Hull, Disc	0.0053			0.0072		
Delaunay	0.2662			0.1648		

Table 6.7: Comparison of the time in seconds, on average, spent by each thread within critical sections using GCC OMP and Tiny STM.

	4 threads			8 threads		
Application	**(A) IntelOMP**	**(B) TinySTM**	**A÷B**	**(A) IntelOMP**	**(B) TinySTM**	**A÷B**
FAST	0.0012	0.0037	0.3243	0.0007	0.0027	0.2593
TREE	0.0150	0.0708	0.2119	0.0086	0.0513	0.1676
2D-MEC	0.0044	0.0112	0.3929	0.0048	0.0121	0.3967
2D-Hull, Kuzmin	0.0019			0.0034		
2D-Hull, Square	0.0035			0.0043		
2D-Hull, Disc	0.0058			0.0083		
Delaunay	0.2170			0.1502		

Table 6.8: Comparison of the time in seconds, on average, spent by each thread within critical sections using Intel OMP critical and Tiny STM.

	4 threads				
Application	**(A) GCCHTM**	**(B) GCCOMP**	**(C) IntelHTM**	**(D) IntelOMP**	**(E) TinySTM**
FAST	0.0011	0.0009	0.0008	0.0012	0.0037
TREE	0.0262	0.0152	0.0130	0.0150	0.0708
2D-MEC	0.0062	0.0033	0.0030	0.0044	0.0112
2D-Hull, Kuzmin	0.0035	0.0017	0.0018	0.0019	
2D-Hull, Square	0.0061	0.0033	0.0033	0.0035	
2D-Hull, Disc	0.0108	0.0053	0.0052	0.0058	
Delaunay	0.4993	0.2662	0.2264	0.2170	

Table 6.9: Comparison of the time in seconds, on average, spent by each thread within critical sections with an execution of four threads.

	8 threads				
Application	**(A) GCCHTM**	**(B) GCCOMP**	**(C) IntelHTM**	**(D) IntelOMP**	**(E) TinySTM**
FAST	0.0010	0.0006	0.0005	0.0007	0.0027
TREE	0.0209	0.0092	0.0078	0.0086	0.0513
2D-MEC	0.0100	0.0033	0.0039	0.0048	0.0121
2D-Hull, Kuzmin	0.0037	0.0014	0.0019	0.0034	
2D-Hull, Square	0.0077	0.0029	0.0032	0.0043	
2D-Hull, Disc	0.0244	0.0072	0.0078	0.0083	
Delaunay	0.3635	0.1648	0.1365	0.1502	

Table 6.10: Comparison of the time in seconds, on average, spent by each thread within critical sections with an execution of eight threads.

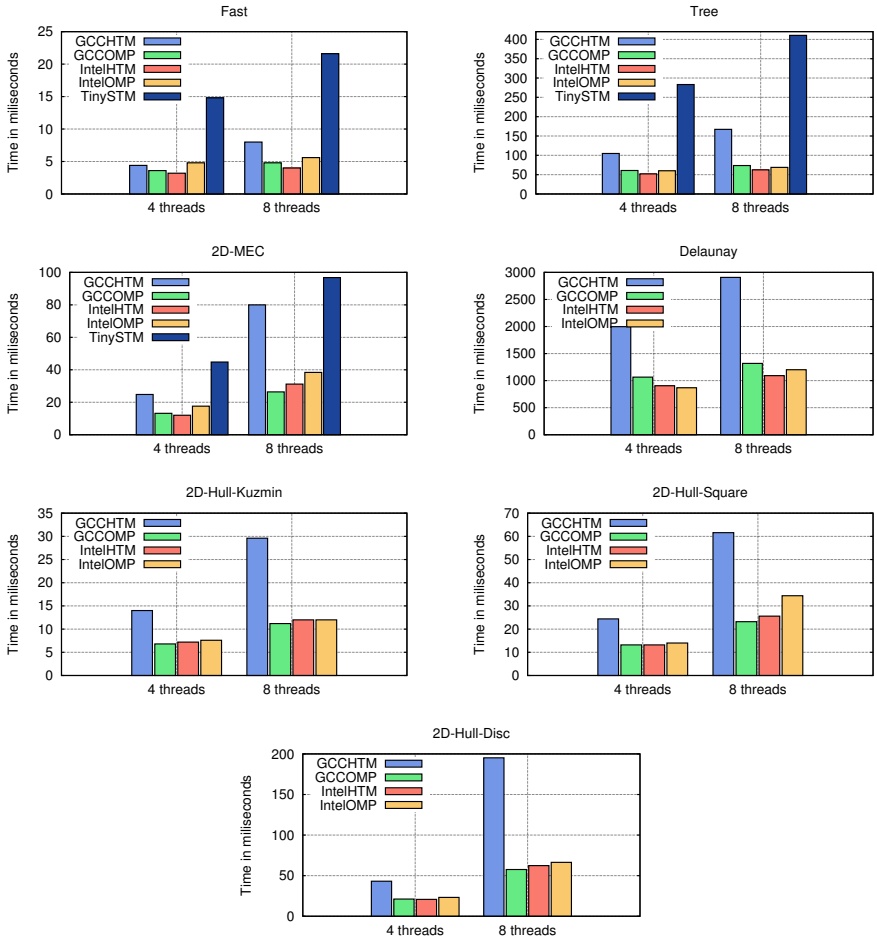


Figure 6.2: Accumulated time in miliseconds required by critical sections regarding the different approaches tested.

As can be seen, generally, TinySTM is the approach which spends more time within critical sections regarding all the benchmarks tested. In addition, results show that GCC using TM also requires, on average, more time using critical sections than the others. This noticeable result may be due to the TM extension of GCC is more general than the TM extension of ICC. Thus, whereas ICC probably adjust its HTM extension to the subjacent architecture (Intel), GCC is not as customized to this hardware. In this sense, Intel extension to HTM even needed the least time with several benchmarks in some executions, e.g., Fast (four and eight threads), Tree (four and eight threads), 2D-MEC (four threads) and 2D-Hull Kuzmin (eight threads). However, we can observe that, overall, ICC HTM extension, GCC OpenMP and ICC OpenMP spent similar times within critical sections. Thus, in the benchmarks examined, we can conclude that the use of HTM does not produce any improvement with respect to OpenMP critical regarding the time spent inside critical sections of the TLS library. Furthermore, this affirmation may be extended to all the approaches examined because results depicted in the plots are expressed in miliseconds, and therefore, differences between times can be neglected concerning the global execution times of the benchmarks. Hence, they will not influence on the speedups (as will be seen in the following sections).

6.7 Experimental results

This section describes the performance results obtained by ATLaS when using GCC-HTM, Intel HTM, GCC OpenMP critical sections, Intel OpenMP critical sections, and TinySTM to execute the benchmarks exposed.

6.7.1 Experimental setup

HTM experiments were performed in a 8-processor server, equipped with eight Intel® Core™ i7-4770 processors at 3.40GHz and 16GB of RAM, which runs Ubuntu 14.04 LTS. We used wall-clock times in our measurements. We have used the OpenMP implementation from GNU Compiler Collection (GCC) 4.8.2 and Intel C/C++ Compiler (ICC) 15.0.1, the transactional libraries from GCC-TM 4.8.2[4] and TinySTM 1.0.5, and the RTM of Intel TSX provided by the OpenMP library of ICC. In order to confirm the use of hardware transactions, and their effect in the execution, we used the Linux kernel tool *perf stat -T*.

We would like to thank to Dr. Luján, of the University of Manchester, for allowing us to use their machine with the mentioned capabilities.

[4] Recall that, in spite of the theoretical simulation of transactions through software performed by GCC-TM, our experiments lead us to affirm that this extension of GCC is able to detect the subjacent architecture, mapping software transactions to hardware extensions. Hence, we have also concluded that GCC-TM uses hardware extensions as long as possible.

6.7.2 Results for OpenMP, STM and HTM

Figure 6.3 illustrates the experimental results obtained. Remind that in this case GCC-TM uses hardware transactions, and note that sequential versions were compiled with GCC.

Experimental results for the Fast bechmark show that there are no differences related to the use of OpenMP critical sections, STM, or HTM. However, there exist significant contrast concerning the use of GCC or ICC. Whilst every GCC versions had the same behavior, ICC versions led to worse results. Actually, GCC versions achieved almost 3.5× with 5 processors, whereas ICC versions gained almost 2× with 8 processors.

Regarding TREE benchmark, again, OpenMP, STM, or HTM solutions did not influence in the performance. However, depending on the compiler used, different results were produced. Whereas every GCC versions had a similar performance, with a peak speedup of 1.35× when running with 8 processors, ICC had different, and faster results, leading to worse times than GCC. ICC versions achieved a speedup of almost 1.25× with 8 processors.

For the 2D-MEC benchmark the use of OpenMP critical sections, STM, and HTM led to rather similar performance. OpenMP critical sections gets a peak speedup of 1.0× with 6 processors, while HTM a 1.0× with 7 processors, and STM a 0.97× with 6 processors. In this case, compiling with GCC or ICC lead to similar results.

Concerning Delaunay benchmark, again, there are no diferences related to the use of OpenMP critical sections, STM, or HTM. However, ICC led to rather lower speedups compared with the use of GCC compiler. GCC versions got a peak of speedup of more than 2×, whilst ICC versions got a 1.8× both with 8 processors.

Finally, 2D-Hull benchmarks behaved similarly for OpenMP's and TM's versions for each dataset.

From the results described above we can make the following observation:

1. In general, HTM approaches do not improve OpenMP critical sections, or STM solutions. Actually, in some cases even obtain worse results. However, it is highly possible that, due to the limited number of processors available, results cannot be extrapolated to every approaches. A possible future work could be repeat the experiments in a machine with many more cores.

2. In some cases, similar compilation parameters lead to better results using ICC or GCC. Specifically, it seems that in benchmarks where dependences hardly appear, or with a pattern of appearances throughout the whole runtime, GCC produces higher speedups. This issue can also be related to the optimization flag used `-O0`.

6.8 Conclusions

The aims of this study were to test whether the use of TM might lead to an improvement in the performance of our software-based, thread-level speculation system. Our experimental

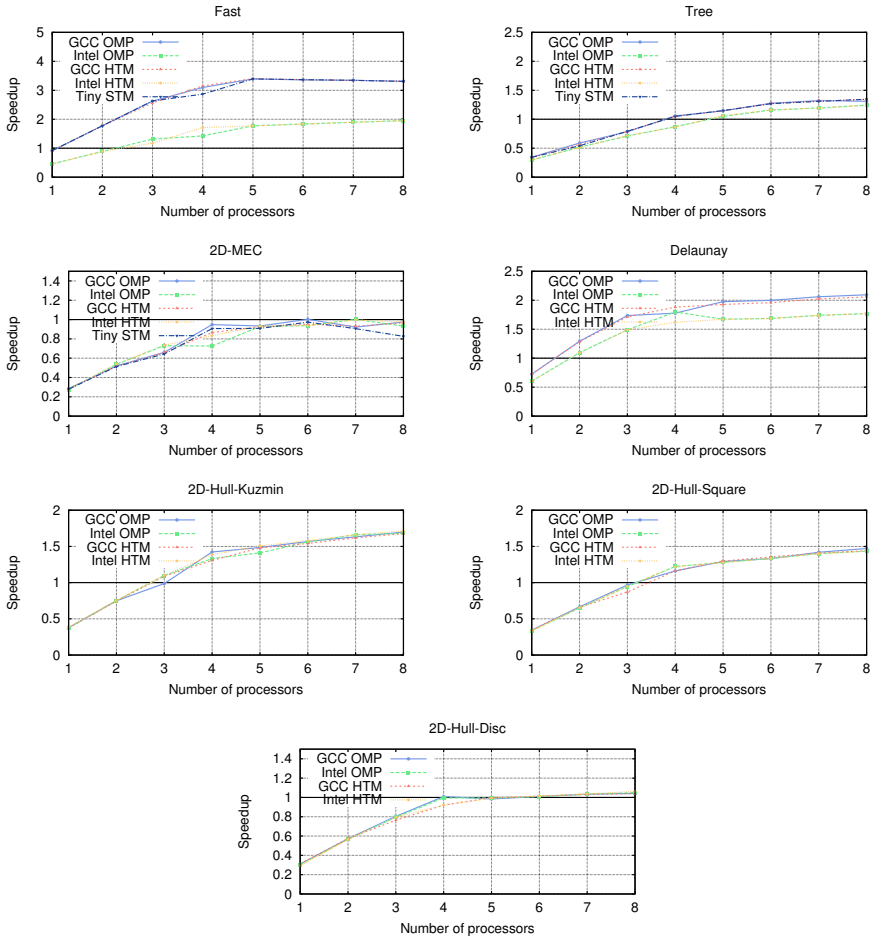


Figure 6.3: Speedups by number of processors for each benchmark tested, comparing the performance obtained by using OpenMP critical sections, compiled with both GCC and Intel compilers, GCC-TM, Intel Hardware Transactional Memory, and TinySTM.

results show that, in general, TM solutions deliver similar performances than the use of the classical OpenMP critical sections for our problem.

The work described in this chapter has produced the following publication:

- Sergio Aldea, Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Study and Evaluation of Transactional Memory approaches with a Software Thread-Level Speculation Framework'. In: *IEEE Transactions on Parallel and Distributed Systems*. To be submitted

Chapter 7

ATLaS in the context of emerging architectures: TLS and Xeon Phi coprocessors

Intel Xeon Phi accelerators are one of the newest devices used in the field of parallel computing. However, there are comparatively few studies concerning their performance when using most of the existing parallelization techniques. One of them is thread-level speculation, a technique that optimistically tries to extract parallelism of loops without the need of a compile-time analysis that guarantees that the loop can be executed in parallel.

In this chapter we evaluate the performance delivered by an Intel Xeon Phi coprocessor when using a software, state-of-the-art thread-level speculative parallelization library in the execution of well-known benchmarks. Our results show that, although the Xeon Phi delivers a relatively good speedup in comparison with a shared-memory architecture in terms of scalability, the low computing power of its computational units when specific vectorization and SIMD instructions are not exploited, indicates that further development of new specific techniques for this platform is needed to make it competitive for the application of speculative parallelization comparing with high-end processors or conventional shared-memory systems.

7.1 Problem description

Currently, physical limitations of single core chips are inducing a quick development of multicore architectures. One of the most recent approaches is the Intel® Xeon Phi™ [58, 124, 132], a coprocessor with more than 60 cores able to execute either offloaded and native codes. Nonetheless, due to its own novelty, this coprocessor has not yet been extensively tested with non-regular parallel codes. The dissemination of experimental results under these conditions would be really useful to test the behavior and capabilities of this computing resource.

In this chapter we use a Xeon Phi coprocessor to run irregular applications that were speculatively parallelized, with the help of a software-only, speculative parallelization library. TLS is useful when executing codes that present scarce dependence violations at runtime. Otherwise, costs associated to check for correctness, stop and retry executions, and commitments, make this technique inefficient.

The contribution of this chapter is to test the performance of a state-of-the-art TLS runtime library using an Intel Xeon Phi. This coprocessor has a big number of parallel threads, therefore, it is interesting to measure its behavior with a shared-memory technique such as TLS, when data is permanently shared among threads. Our experimental results show that the benchmarks considered scale well when running them with a Xeon Phi coprocessor. However, our results also show that, due to the irregular nature of the target applications for TLS techniques, and the modest computing capabilities of each individual core when vectorized and SIMD instructions are not exploited, execution times are much higher than those gauged in conventional shared-memory systems.

The rest of this chapter is structured as follows: Section 7.2 describes the main characteristics of the Xeon Phi coprocessor. Section 7.3 describes both the experimental environment and the benchmarks used. Section 7.4 shows some experimental results in terms of performance measured in a shared-memory system without coprocessor, and in a Xeon Phi coprocessor. Section 7.5 summarizes some works that helps to put into perspective our contribution. Finally, Sect. 7.6 concludes this chapter.

7.2 Intel Xeon Phi in a nutshell

Intel Xeon Phi [58, 124, 132] is a coprocessor launched by Intel in 2012. It is called coprocessor because, although it can run a Linux operating system by itself, it should be placed aside another processor to work properly. Although first impressions might suggest a number of similarities, it is not an accelerator such as GPUs. Whereas the Intel Xeon Phi cores are more similar to classical complete CPUs, the GPUs thread scheduling hardware is different.

Furthermore, Intel Xeon Phi coprocessors do not use the grid, and groups of threads concept[1] in the same way, and also the memory latency hiding mechanisms are different. This issue hinders easy code migrations to the latter kind of accelerators, and requires and in-depth understanding of special programming models as CUDA [189], or OpenCL [145]. On the other hand, the Xeon Phi coprocessor is able to use all standard parallel programming models such as OpenMP [59], POSIX threads, MPI [252], or even OpenCL. Thus, using this new coprocessor only requires a minimum learning curve, assuming that the programmer knows at least one of these common parallel programming models.

7.2.1 Internal details

Intel Xeon Phi coprocessors have up to 61 cores at 1090 MHz, interconnected by a high-speed bidirectional ring. Each core is enhanced with four hardware threads (up to 244 threads per coprocessor), and with a 512-KB L2 cache. L2 cache levels are shared by all cores. Furthermore, in addition to 64-bit x86 instructions, cores offer 512-bit wide SIMD vectors, making vectorization the most powerful way to gain performance. The coprocessor is generally connected to the host system via the PCI Express bus, and supports up to 8 GB GDDR5 memory. Figure 7.1 briefly describes the architecture of the Intel Xeon Phi.

7.2.2 Use of the Xeon Phi

There are mainly two ways of executing an OpenMP program into a Xeon Phi coprocessor:

Native Execution: The Intel Xeon Phi coprocessor is capable of running a Linux operating system. It is possible to log in to the Xeon Phi from the host processor using SSH, through a `mic0` network interface, added to the kernel by a module provided by Intel, and use it natively. Thus, it allows the execution of the typical Linux-based commands as well as our own programs.

Offload Extensions from the host: Intel defined a set of pragmas and keywords to be used in parallel codes in order to execute them in coprocessors. A programmer only needs to declare the region which should be executed in a coprocessor. Inside this region, any kind of function can be used. For example, in the case of OpenMP, a single pragma defined as `#pragma offload target{mic}` should be used, where `mic` represents the identifier of the target Xeon Phi coprocessor. In addition, we should point out the variables that will be used in the coprocessor, declaring their use with the clauses `in()`, `out()`, or `inout()`. The use of variables with dynamic size requires to explicitly

[1] As the reader may know, GPUs have a hierarchical hardware architecture, so they should be programmed with a hierarchical thread structure in mind [33], that uses the concept of threads, blocks, and grids. A thread is the simplest unit of execution, intended to process a specific code. A block is defined as a group of threads, where threads can be executed concurrently or sequentially with no order. At this level, a block allow the coordination of its threads with the use of barriers. A grid is a group of blocks without any possible synchronization among them.

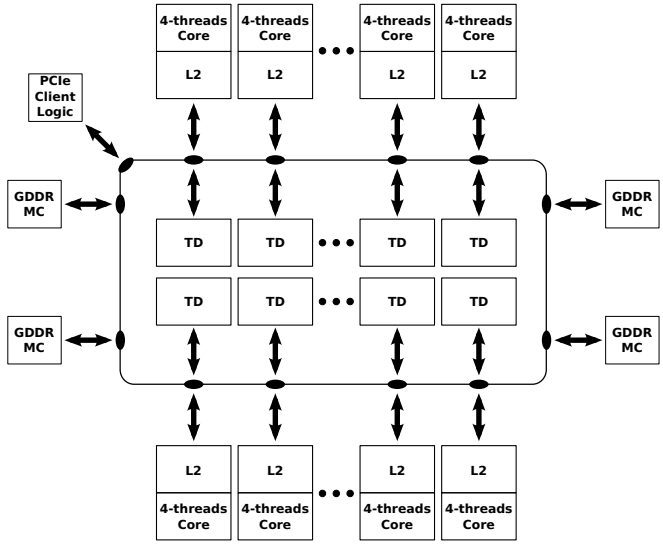


Figure 7.1: Overview of the microarchitecture of an Intel Xeon Phi coprocessor.

declare the size, e.g. in(a:length(n)). These variables will be copied from the host to the device, and/or vice versa, depending on their usage.

As can be seen, the Xeon Phi programming methodology is really convenient in order to gain speedup with a relatively low programming effort.

7.3 Experimental evaluation

The goal of this work is to test the Xeon Phi coprocessor in off-loading mode to speculatively execute in parallel different, well-known benchmarks. In this way, the ATLaS runtime library was adjusted to offload the execution of the parallel loop to the Xeon Phi coprocessor, without further optimizations such as vectorization, one of the most important features of the Xeon Phi. In any case, this feature is not very useful for our benchmarks, mainly composed of irregular code.

To test the performance of the ATLaS TLS runtime, we have used three different real-world benchmarks, together with a synthetic one. The real-world applications include the 2-dimensional Convex Hull problem (2D-Hull) [53], the Delaunay Triangulation problem [63, 186], and a C implementation of the TREE benchmark [18]. The synthetic benchmark is the FAST. All of them are described in the Appendix A.

7.3.1 Environmental setup

We have used two different platforms to compare the scalability of the speculative execution of our benchmarks. The first one is Heracles, a 64-processor server, equipped with four 16-core AMD Opteron 6376 processors at 2.3GHz and 256GB of RAM, which runs CentOS 7 Linux. The second one is Chimera, a server equipped with two Intel Xeon E5-2620 V2 processors with six cores each, 32 Gb of RAM, and a Xeon Phi 3120A coprocessor with 6 Gb of RAM running at 1.1 GHz. The system also runs CentOS 7 Linux.

All threads had exclusive access to the processors during the execution of the experiments, and we used wall-clock times in our measurements without taking into account times spent in data transfer. We have used icc (ICC) 15.0.2 for all applications in both platforms (Xeon Phi offloaded codes can only be compiled with ICC). Times shown in the following sections represent the time spent in the execution of the parallelized loop for each application. To better assess the scalability offered by the Xeon Phi, the time required for data offloading has not been taken into account in the measurements.

7.4 Experimental results

7.4.1 Scalability

Figure 7.2 compares the speedup obtained with the same parameters in both the shared-memory processor, and the Xeon Phi coprocessor. Results show that, regarding the speedup, the Xeon Phi coprocessor delivers a better scalability than a conventional, shared-memory system. This scalability improvement is related to the Xeon Phi memory architecture. All TLS runtime libraries require many accesses to shared data, so the faster and higher bandwidth, the better performance. In our case, while the AMD Opteron 6376 achieves up to 51.2 GB/s memory bandwidth per socket, [15], the Intel Xeon Phi coprocessor achieves a peak of 240 GB/s [125]. In our experiments, the benchmark with the highest number of variables involved in the speculative execution is the Delaunay triangulation, with more than 12 million, different scalar variables, while the one with the smallest shared data set is FAST, with just two variables. Whilst in the latter benchmark the speedup is similar in both architectures, in the Delaunay triangulation the speedup achieved by the Intel Xeon Phi is up to $2.38\times$ higher with respect to the AMD Opteron 6376.

7.4.2 Oversubscription

Figure 7.3 shows the experimental results produced with the execution of the benchmarks using the whole threads of the Xeon Phi coprocessor. The particular nature of the threads per core in this platform, being not independent of each other, severely limits the scalability when more than 60 or 70 threads are launched, depending on the application. In some cases,

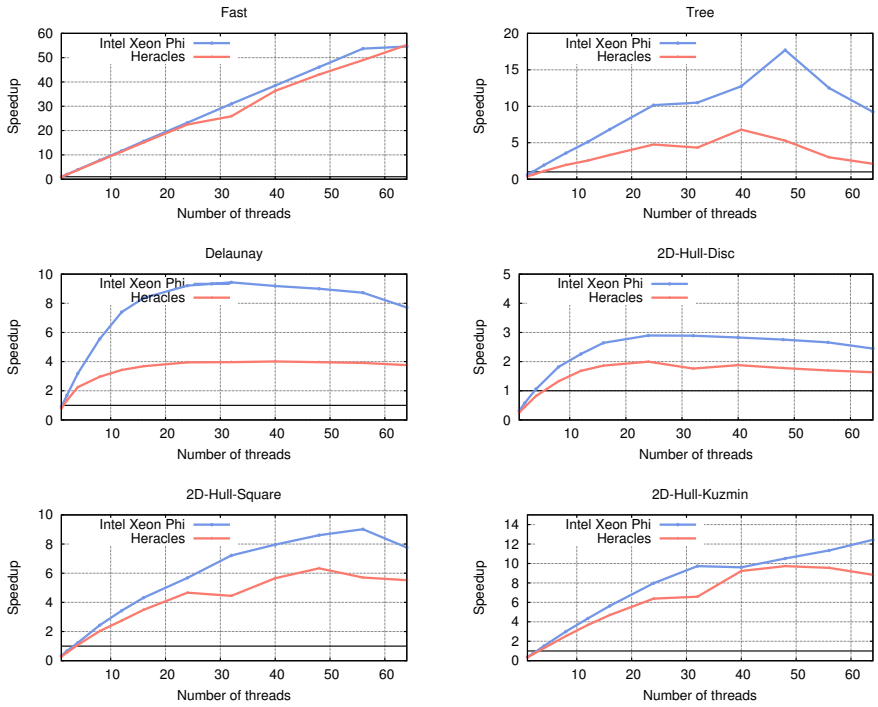


Figure 7.2: Speedups by number of processors for each benchmark tested, comparing the performance obtained by using Intel Xeon Phi coprocessor, and a conventional shared-memory system.

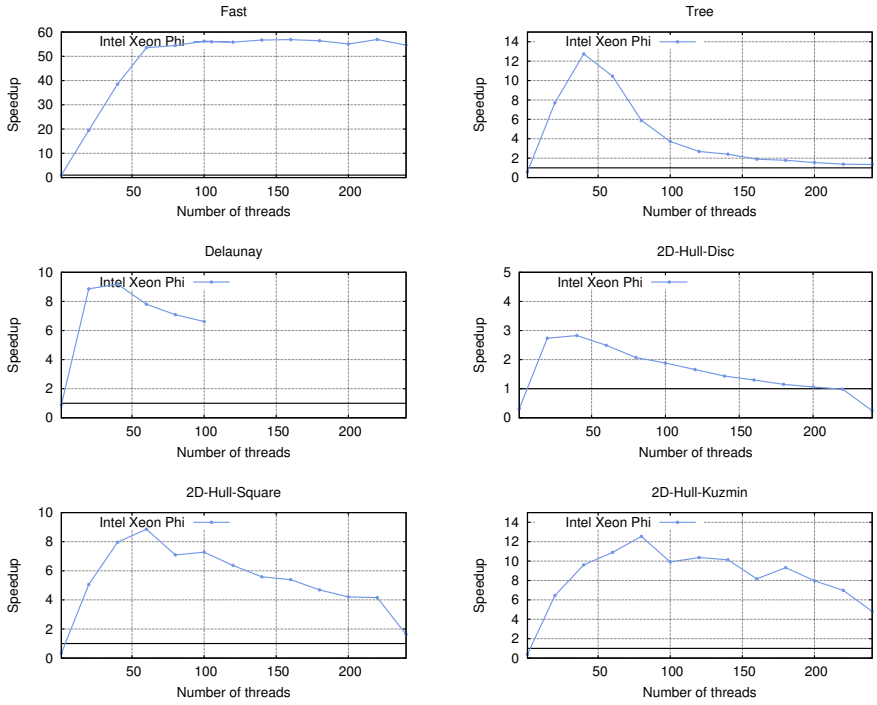


Figure 7.3: Speedups by number of processors for each benchmark tested on the Intel Xeon Phi coprocessor.

performing such an oversubscription with respect to the number of cores leads to slightly better results, but the performance decays when we tried to use more cores. We attribute this fact mainly to memory issues. As we have exposed in Sect. 7.2.1, Xeon Phi coprocessors can manage up to 244 threads. However, due to the fact that threads of each core are not independent, from 61 threads on (there are in total 61 cores) most of them are idle when executing a speculative operation.

In conclusion, we have found that the particular architecture of the Xeon Phi, with threads working synchronously in each core, is not particularly suitable for software-based speculative execution.

7.4.3 Absolute performance

Although the Xeon Phi presents a better scalability when comparing with a conventional, shared-memory system, when considering absolute times, the picture is very different. Fig-

	32 processors			64 processors		
Application	**(A) Xeon Phi**	**(B) Heracles**	**A÷B**	**(A) Xeon Phi**	**(B) Heracles**	**A÷B**
FAST	154.45	19.28	8.01	87.68	9.03	9.71
2D-Hull, Disc	11.71	2.22	5.27	13.81	2.39	5.77
2D-Hull, Square	4.93	0.99	4.98	4.58	0.80	5.75
2D-Hull, Kuzmin	3.01	0.54	5.62	2.36	0.40	5.90
Delaunay	114.08	22.04	5.18	139.50	23.24	6.00
TREE	87.30	23.18	3.77	99.33	47.49	2.09

Table 7.1: Comparison of the time in seconds required to execute the benchmarks tested in both the Heracles, the shared memory system, and in the Xeon Phi coprocessor of Chimera.

ure 7.4 shows the absolute times required to run each benchmark in Heracles and in the Xeon Phi installed in Chimera. The analysis of this figure leads to two conclusions. First, the use of the Xeon Phi to execute these benchmarks in parallel reduces the execution time needed by a single, high-end processor for the Fast, Tree, and 2D-Hull benchmarks, using the Square and the Kuzmin input sets. On the contrary, Delaunay and 2D-Hull using the Circle input set does not benefit at all from this architecture.

The second conclusion is that when comparing the absolute times obtained with the same number of threads in both architectures, we can see that the shared-memory architecture of Heracles allows to obtain execution times that are roughly an order of magnitude better than those produced by the Xeon Phi. The reasons are the more advanced architecture of AMD processors, with out-of-order execution, and their higher clock speed, among other factors. These reasons compensates the performance losses derived from the memory organization in the shared-memory system with respect with the one offered by the Xeon Phi, that leads to a better scalability as we saw in previous sections.

Table 7.1 summarizes the execution times for 32 and 64 threads in both architectures, with the corresponding relationship. As can be seen, relative speedups obtained by Heracles range from 2.09× for TREE with 64 processors, to 9.71× for FAST with 64 processors.

Despite the poor performance delivered, we consider that the Xeon Phi coprocessor may still help in the speculative execution of loops thanks to their comparatively big number of threads. Our future work include the combination of software-based TLS techniques with other solutions, such as value prediction, or the use of helper threads.

7.5 Related work: TLS and the Xeon Phi coprocessor

To the best of our knowledge, this is the first research that tests TLS with Xeon Phi coprocessors. We have deeply reviewed some related TLS approaches in the Chapter 2, so in the

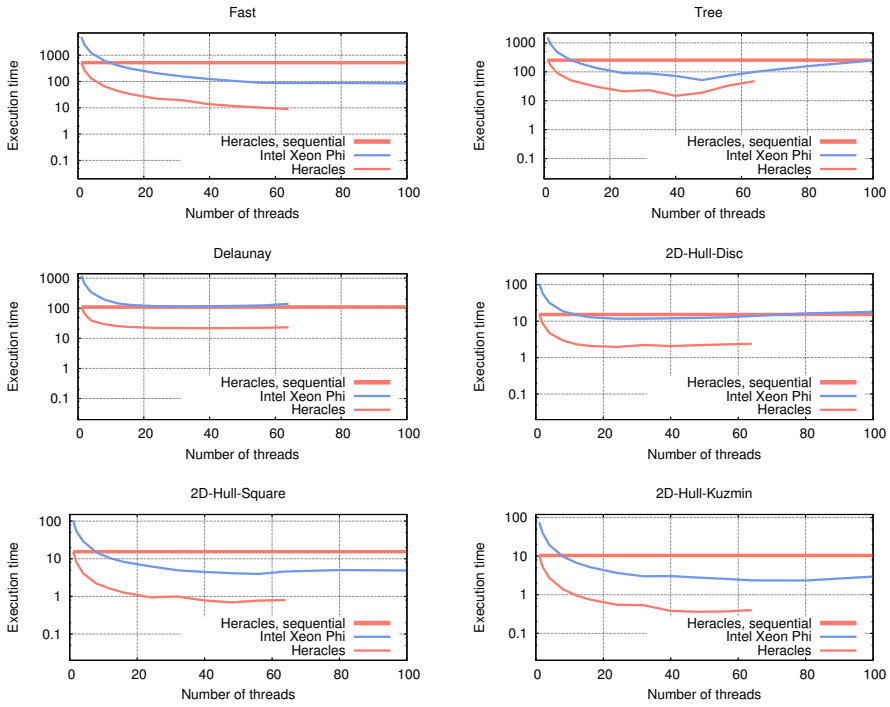


Figure 7.4: Execution time in seconds with respect to the number of threads for each benchmark. The sequential time obtained with a single Xeon processor in Heracles is also shown.

following related work we are going to center our efforts just on research about Xeon Phi capabilities. To do so, we will propose a number of possible hardware additions which might improve the performance of TLS on the Intel Xeon Phi coprocessors. Afterwards, we will describe some of the papers related to Xeon Phi coprocessors.

7.5.1 Hardware improvements to benefit software TLS

We will now explore some enhancements which might possibly improve the performance of TLS on Intel Xeon Phi coprocessors. We will center our discussion on applying ideas belonging to the classical hardware approaches to manage speculation in multicore processors. Therefore, the implementation of these ideas would need changes in the Xeon Phi architecture.

Sohi et *al.* [233] developed the Multiscalar processor, where cores were interconnected through a ring, an approach also followed in the Speculative multithreaded processor [177]. For these systems, hardware modules developed to store intermediate versions of variables were also proposed, such as ARB [94] or SVC [105]. As long as the ring interconnection mechanism is also present in the Xeon Phi coprocessors', the application of their mechanisms to handle dependences in hardware might decrease software overheads.

Another possible improvement might be the addition of a new cache, based on the Trace cache [224]. This proposal stored traces (dynamic sequences of instructions stored in the hardware) at runtime, and instructions were executed in parallel, while dependences were speculated with the use of predictors.

A different approach like the used in the I-ACOMA architecture [151] may work as well. They used a binary annotator that added some notes into executable files to detect possible dependences, that were managed at runtime with a special module called Memory Disambiguation Table. Another source of ideas for improvements is the Threaded Multi-Path Execution [253] approach, that was focused on prediction techniques. This proposal executed all possible branches of a loop, whilst there were enough resources, a situation that is likely to occur in Xeon Phi coprocessors.

7.5.2 Studies related to the Xeon Phi coprocessor

The Xeon Phi coprocessor is being extensively studied. Some papers have developed extensions to offloaded regions. For example, COSMIC [32] is a middleware integrated in the subjacent software that tried to ease and improve the performance of multiprocessing in Xeon Phi coprocessors. This work aimed to reduce imbalance and overheads through the management of resources. It handled offload regions and takes care of the request of coprocessors, cores and memory. Snapify [221] tried to reduce failure rates of Xeon Phi coprocessors. The underlying idea was taking snapshots during execution (saving the state of applications) and if an error was produced, the execution was restored to a correct, saved state, instead of being restarted.

Some essays are focused in the implementation of existing algorithms into coprocessors. For example, [199] developed a multi-node 1D FFT implementation on coprocessors; [169] implemented a sparse matrix-vector multiplication; and [191] developed a SQL engine that benefited from the inherent parallelism related to Xeon Phi coprocessors.

Furthermore, as it is the case, there are many other papers centered on the measurement of the performance obtained from a Xeon Phi. [230] was one of the first papers that used Intel Xeon Phi coprocessor (that was called Intel Knights Ferry) to evaluate the performance of scientific applications. Later, Cramer et *al.* [58] evaluated the behavior of some OpenMP benchmarks in a Xeon Phi coprocessor. They affirmed that common OpenMP codes could be easily migrated to Intel Xeon Phi, gaining more parallel performance without adding overheads. This study was enhanced in [231]. [85] also tested the Xeon Phi through the development of some microbenchmarks.

7.6 Conclusions

In this work we have evaluated the behavior of the Xeon Phi coprocessor in the context of software-only, thread-level speculation (TLS), a parallel technique that optimistically executes in parallel sequential codes without a prior dependence analysis. Intel Xeon Phi coprocessors are one of the state-of-the-art architecture that aims to execute parallel codes. Our experimental results show that the particular memory architecture of the Xeon Phi leads to better scalability with regards to speculative execution, with better relative speedups than those obtained using a conventional, shared-memory architecture. However, the relative low computing power of its computational units when specific vectorization and SIMD instructions are not exploited, indicates that further development of new specific techniques for this platform is needed to make it competitive for the application of speculative parallelization comparing with high-end processors or conventional shared-memory systems.

Although the use of a Xeon Phi coprocessor to execute software-based, TLS codes is not competitive, the Xeon Phi architecture might be useful when combining TLS solutions with other existing techniques such as value prediction or helper threads. In this way, some of the available threads could be used to help TLS execution, reducing dependence violations and thus improving performance.

The work described in this chapter has generated the following publications:

- Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Evaluating the capabilities of the Xeon Phi platform in the context of software-only, thread-level speculation'. In: *Proceedings of the 8th International Symposium on High-level Parallel Programming and Applications*. HLPP '15. Pisa, Italy: ACM, 2015. To be also published in *International Journal of Parallel Programming*, Springer US.

Chapter 8

Conclusions

Speculative parallelism is an optimistic parallel technique whose importance has increased in the last two decades due to advances in parallel computing. Nonetheless, the fact that it requires irregular codes with dependences, or irregularities which does not allow standard parallel techniques parallelize them, as well as the need that these codes do not have too many dependencies (otherwise results are even poorer), hinders speculative parallelism to be useful in real-life applications. It, therefore, continues being a research field with both supporters and detractors. We might be included in the former group. Thus, throughout this Ph.D. thesis we have developed tools and strategies in order to contribute to this promising method of parallelism. Among them we can highlight the runtime library of the ATLaS framework described in chapters Chapter 3 and Chapter 4 respectively, the hash-based data structure which severely reduces accesses to data structures involved in speculative operations, the moody scheduling approach which dynamically ease the achievement of the best size of chunks of iterations executed, and the combination of our speculative library with some Transactional Memory implementations in order to test if these approaches are better than the critical sections used. Furthermore, we give some experimental results to endorse our research not only in conventional shared-memory machines, but also in one of the most novel devices, the Intel Xeon Phi coprocessor. This chapter summarizes the work carried out throughout this Ph.D. thesis emphasizing on the contributions, the answer to the research question formulated at the beginning of this document, and proposes a number of ways so that this work will be continued in the future.

8.1 Summary of results and contributions

The contributions of this Ph.D. thesis ordered by goals, as well as the publications achieved with them are the following.

8.1.1 Goal 1: Deep study of the state-of-the-art in TLS

We have reviewed most of the existent solutions regarding speculative parallelism. Thus, we have proposed a classification in which every approach may be framed. To the best of our knowledge no previous work had been done of this kind, at least, as wide as ours. The work done has been accepted to be published in the following journey:

1. Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'A survey on Thread-Level Speculation Techniques'. In: *ACM Computing Surveys (CSUR)*. Accepted for publication

8.1.2 Goal 2: Combine a TLS library with a compiler

Concerning compilers, there were no approaches centered on giving support to TLS until Aldea et *al.* proposed one [7]. To did so their framework required of a TLS runtime library able to execute applications with complex instructions such as pointer arithmetic or complicated 'structs'. We therefore implemented an easy-to-use TLS runtime library capable of both communicating with a compiler effortlessly and executing all kind of applications. This contribution have been published in the following papers:

2. Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros'. In: *Proceedings of the XXIII Jornadas de Paralelismo*. Elche, Alicante, Spain: Servicio de Publicaciones de la Universidad Miguel Hernández, Sept. 2012. *ISBN*: 978-84-695-4471-6

3. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'A New GCC Plugin-Based Compiler Pass to Add Support for Thread-Level Speculation into OpenMP'. English. In: *Euro-Par 2014 Parallel Processing*. Ed. by Fernando Silva, Inês Dutra and Vítor Santos Costa. Vol. 8632. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 234–245. *ISBN*: 978-3-319-09872-2. *DOI*: 10.1007/978-3-319-09873-9_20. *URL*: http://dx.doi.org/10.1007/978-3-319-09873-9_20

4. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Una extensión para OpenMP que soporta paralelización especulativa'. In: *Proceedings of the XXV Jornadas de Paralelismo*. Valladolid, Spain, Sept. 2014. *ISBN*: 978-84-697-0329-3

5. S. Aldea, A. Estebanez, D.R. Llanos and A. Gonzalez-Escribano. 'An OpenMP Extension that Supports Thread-Level Speculation'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, 2015, pp. 1–14. 2015. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2393870

8.1.3 Goal 3: Improve operations involved in a TLS runtime library

As a result of the previous goal, and in order to increase the possible impact which ATLaS may achieve, we tried to locate and ease the main bottlenecks included in the TLS library. To do so, the main proposal has been a hash-based data structure able to severely decrease accesses to data structures involved in main speculative operations. Hence, it lead us to quite noticeable improvements in the performance of the library.

Additionally, scheduling of iterations have been a research field deeply studied. Nonetheless, regarding scheduling of iterations under TLS there are not much work done so far. Thus, after reviewing some of the existent algorithms, we realized that there were no solution focused on achieve the best chunk size dynamically based on both runtime and user-defined parameters. We devised *'Moody scheduling'* so as to speculatively execute loops following the dependence pattern as well as the optimism desired to set sizes of chunks of iterations under TLS. These achievements led to the following publications:

6. Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Improving the Perfomance of a Pointer-Based, Speculative Parallelization Scheme'. In: *Proceedings of the 1st First Congress on Multicore and GPU Programming*. PPGM'14. Granada, Spain, Feb. 2014. Also published in *Annals of Multicore and GPU Programming*, vol. 1, no. 1, 2014. 2014. issn: 2341-3158.

7. Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'New Data Structures to Handle Speculative Parallelization at Runtime'. In: *Proceedings of the 7th International Symposium on High-level Parallel Programming and Applications*. HLPP '14. Amsterdam, Netherlands: ACM, 2014, pp. 239–258. Also published in *International Journal of Parallel Programming*, 2015, pp. 1–20. Springer US, 2015. issn: 0885-7458. doi: 10.1007/s10766-014-0347-0. url: http://dx.doi.org/10.1007/s10766-014-0347-0.

8. Alvaro Estebanez, Diego R. Llanos, David Orden and Belen Palop. 'Moody Scheduling for Speculative Parallelization'. English. In: *Euro-Par 2015: Parallel Processing*. Ed. by Jesper Larsson Träff, Sascha Hunold and Francesco Versaci. Vol. 9233. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015, pp. 135–146. ISBN: 978-3-662-48095-3. DOI: 10.1007/978-3-662-48096-0_11. URL: http://dx.doi.org/10.1007/978-3-662-48096-0_11

8.1.4 Goal 4: Test a TLS runtime library with other parallel techniques

So as to complete our work, we have tested our TLS runtime library in other contexts different from the conventional shared-memory machines. In this way, we have combined the transactional memory approach with our library. Hence, we compared the original critical sections implemented in the software, against a new approach which uses a number of hardware- and software-based transactional memory solutions. Furthermore, we have examined the behavior of our software in one of the most state-of-the-art devices of the moment, a Intel Xeon Phi coprocessor. This work allow us to publish the following paper:

9. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Study and Evaluation of Transactional Memory approaches with a Software Thread-Level Speculation Framework'. In: *IEEE Transactions on Parallel and Distributed Systems*. To be submitted

10. Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Evaluating the capabilities of the Xeon Phi platform in the context of software-only, thread-level speculation'. In: *Proceedings of the 8th International Symposium on High-level Parallel Programming and Applications*. HLPP '15. Pisa, Italy: ACM, 2015. To be also published in *International Journal of Parallel Programming*, Springer US.

8.2 Answer to the research question

> *Is it possible to develop a runtime system for thread-level speculation able to efficiently handle complex data structures, use pointer arithmetic, and take into account the tendency of dependence violations produced so far to estimate the best chunk size to be scheduled? Could it be implemented and lead to good execution times using Transactional Memory, and in new manycores architectures such as the Intel Xeon Phi coprocessors?*

As a result of the research done we can affirm that our research question has been clearly answered. On the one hand, we developed a TLS software capable of (a) handling complex data structures, as well as, (b) using pointer arithmetic. Moreover, we proposed (c) a scheduling strategy which takes into account not only runtime parameters such as dependence violations, but also the optimism decided by the user to assign higher or smaller chunks of iterations.

In addition, we verified that (d) TLS can be combined with transactional memory producing similar results than the classical approach, and (e) can be used in a Intel Xeon Phi coprocessor.

8.3 Future work

There are still a wide range of work to do in the field. In this sense, this Ph.D. thesis might be extended in future research projects following one of this proposals.

- *Helper threads.* Currently devices have more and more processors, consequently there are some algorithms which do not improve their performance even using a bigger number of processors. Thus, some of these free processors might be used as a helper thread. In other words, a thread whose aim is ease the operations to others. We guess that devices such as Xeon Phi coprocessors would be likely to gain from this idea.

- *GPUs.* Nowadays, one of the most successful approaches to parallelize codes are GPUs since they may consists of more than a thousand of processing units. There are not many speculative solutions using these devices so far, therefore, it could be a good idea either extend the capabilities of our software runtime library, or develop a new branch to take advance of the high computational capacity of GPUs.

- *Use predictors.* As stated throughout this thesis, speculative operations spend a big amount of time looking for the most recent values. Thus the use of a predictor responsible of forecasting the possible values of variables with a high and accurate success rate would be likely a good way of improving the performance (as some approaches has already proven).

- Polyhedral model. One of the most relevant approaches to transform loops in order to reduce dependences is the polyhedral model [20]. To the best of our knowledge only one approach takes advantage of such an interesting technique as is [135]. Therefore it could be a good idea combine these two approaches and compare with the mentioned solution. Hence both research fields will likely be benefited of the possible conclusions which may arise.

Appendix A

Benchmarks description

The use of some well-known benchmarks to test the work done is mandatory in order to put a development into perspective. In this chapter are described the benchmarks used throughout this Ph.D. thesis to test every solution made. They include some randomizes incremental algorithms as well as other applications widely used in the 'real life'. In addition, we have implemented three synthetic benchmarks which able to check the correctness of our developments.

To test the speculative library, we have used both real-world and synthetic benchmarks. In order to obtain a better understanding of the experimental results, both of them are going to be explained.

A.1 Randomized incremental algorithms

First of all, we will briefly review a kind of algorithms called randomized incremental algorithms, because some of the benchmarks tested in our experiments are included in this category.

Randomized incremental algorithms have been deeply studied in the context of Computational Geometry and Optimization. Their use have allowed the development of simple, easy-of-implement and efficient algorithms that solve several problems. For example, line segment intersection, Voronoi diagrams, triangulation of simple polygons, linear programming and many others.

In its most general formulation, the input set of a randomized incremental algorithm is a set of elements (they can be points or not) subjected to some operations in order to obtain certain output. Generally, a loop iterates over every element of the set, trying to find those which fulfill a range of requirements. The simultaneous execution of two iterations in two different processors requires that no dependence exists between results calculated in the first iteration and values needed by the second iteration. These kind of algorithms present a common dependence pattern among iterations of loops independently of the problem to solve. Informally, it could be said that at the beginning of the execution most of the elements inserted modify the solution being calculated iteration by iteration. However, as the execution progresses, less dependencies appear, i.e., less elements modify the solution. Regarding complexity analysis, it is expected that would normally be much lower than the complexity found in the worst case. Speculative parallelization is the most effective technique to execute in parallel this dependencies distribution.

Some of the problems addressed to obtain experimental results are:

- Welzl algorithm to calculate the "2-dimensional Minimum Enclosing Circle" problem (2D-Mec).
- Clarkson et *al.* algorithm to calculate the "2-dimensional Convex Hull" problem (2D-Hull).
- *Jump-and-Walk* strategy to calculate the "2-dimensional Delaunay Triangulation" problem (2D-DT).

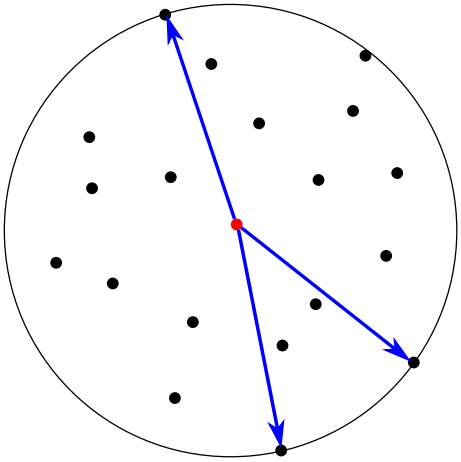


Figure A.1: Minimum enclosing circle defined by three points.

A.1.1 Minimum enclosing circle

The 2D-MEC problem consists in finding the smallest circle that encloses a set of points. We have parallelized the randomized incremental approach due to Welzl [259], which solves the problem in linear time. This algorithm starts with a circle of radius equal to zero located in the center of the search space. If a point lies outside the current solution, the algorithm defines a new circle that uses this point as one of its frontiers. It is interesting to note that points inside the old solution may lie outside the new one. Therefore, all points should be processed again to check if the new circle encloses them. The solution can be defined by two or three points, and the algorithm is composed of three nested loops. We have used a random, ten-million point, uniformly distributed input set. We have speculatively parallelized the innermost loop, which consumes 43.75% of the total execution time (see Tab. 4.1). In [82] can be found experimental results of the parallelization of the other loops. Figure A.1 shows an example of the minimum enclosing circle of a given input set.

A.1.2 Convex hull

The 2D-Hull problem solves the computation of the convex hull (smallest enclosing polygon) of a set of points in the plane. We have parallelized Clarkson et *al.*'s [53] implementation [50, 65, 104]. The algorithm starts with the triangle composed by the first three points and adds points in an incremental way. If the point lies inside the current solution, it will be discarded. Otherwise, the new convex hull is computed. Note that any change to the solution found so far generates a dependence violation, because other successor threads may have used the old enclosing polygon to process the points assigned to them. The probability of a dependence

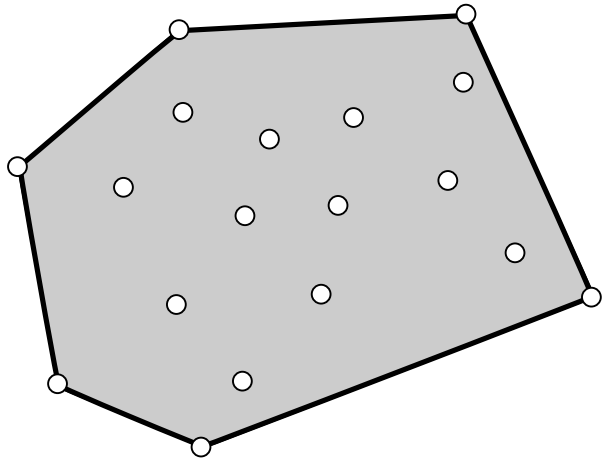


Figure A.2: Convex hull of a set of points.

violation in the 2D-Hull algorithm depends on the shape of the input set. Therefore, we have used three different, ten-million-point input sets to run this benchmark. The *Kuzmin* input set [28] follows a Gauss-Kuzmin distribution, with a higher density of points around the center of the distribution space, which leads to very few dependence violations, since points far from the center are very scarce. The two other input sets, *Square* and *Disc*, cause more dependence violations than *Kuzmin*, with their points uniformly distributed inside a square and a disc, respectively. The *Square* input set leads to an enclosing polygon with fewer edges than the *Disc* input set, thus generating fewer dependence violations. Figure A.2 shows an example of the convex hull of a given input set.

A.1.3 Delaunay triangulation

A triangulation is a subdivision of an area or plane polygon into a set of triangles, taking into account that each side of the triangle is shared by two adjacent triangles. Analogously a triangulation of a two-dimensional set of points is defined as a convex hull partition into triangles. The structure is a maximal family of disjoint interior triangles whose vertices are points of the set. Of course, there are not points located inside the triangles. Figure A.3 shows that a single data set could generate different triangulation.

Delaunay triangulation [99] applied to a two-dimensional set of points affirms that a network of triangles is a Delaunay triangulation if all the circumcircles of all the triangles of the network are empty, i.e., the circumcircle of each triangle of the network contains no other vertices that those three that define the triangle. This condition ensures that the interior angles of the triangles are as large as possible and the length of the sides of the triangles is minimal. See Figure A.4.

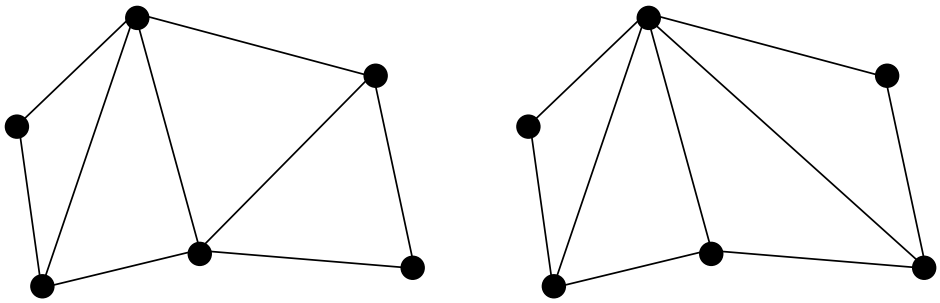


Figure A.3: Delaunay: Two different triangulations with the same set of points.

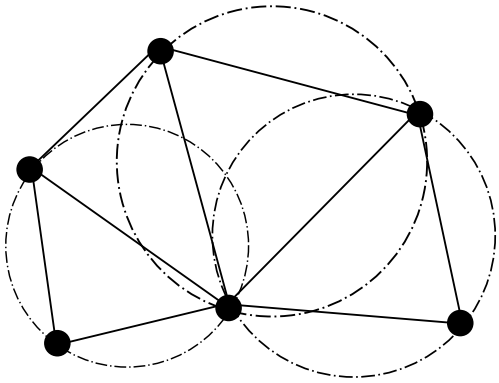


Figure A.4: Delaunay triangulation of a set of points: Circumcircles of triangles shown do not contain any point inside them.

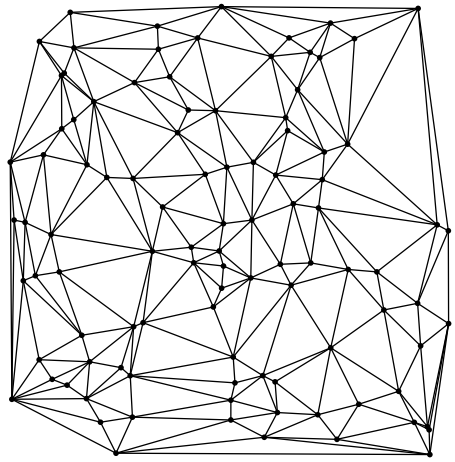


Figure A.5: Delaunay triangulation of a set of 100 points.

In this case, the randomized incremental construction of the Delaunay triangulation will be addressed by using *Jump-and-Walk* strategy, introduced by Mücke, Zhu et *al.* [63, 186]. This incremental strategy starts with a number of points, called anchors, whose containing triangles are known. The algorithm finds the closest anchor to the point to be inserted (the jump phase), and then traverses the current triangulation until the triangle which contains the point to be inserted is found (the walk phase). The goal of the algorithm is to find the network of triangles in which all the circumcircles of all triangles in the network are empty, i.e., the circumcircle of each triangle contains no other vertices than those three that define the triangle. We have used an input set of 5000 anchors, and one million points to be inserted, or another of 5000 anchors, and one hundred thousand points to be inserted. Figure A.5 shows an example of the Delaunay triangulation of a given input set that contains 100 points.

A.2 TREE benchmark

The TREE problem [18], unlike the previous three applications, does not suffer from dependence violations, but it is still not parallelizable at compile time because the compiler is not able to ensure that there are no data dependencies. This application spends a large fraction of its sequential execution time on a loop that can not be automatically parallelized by state-of-the-art compilers because it has dependence structures that are either too complicated to be analyzed at compile time or dependent on the input data. Specifically, we have been focused on the loop that iterates over the bodies and computes the forces on them, which has more than 150 code lines. Compilers also find hurdles in several sum and maximum

reductions contained in the code, which ATLaS detects and handles properly. We have run this benchmark with a 4096-point input set.

A.3 Synthetic benchmarks

Listings A.1, A.2 and A.3 show the code of three synthetic benchmarks: Complete, Tough and Fast. The purpose of these benchmarks is to test (a) the correctness of our solution when using different speculative data sizes (the Complete benchmark); (b) the robustness of the solution when few speculative variables are under heavy use (the Tough benchmark); and (c) the overheads due to the scheduling of the speculative code (the Fast benchmark). However, these synthetic benchmarks were not designed to exhaustively study the sensitivity of the ATLaS framework in different situations, such as the number of speculative variables or the chunk sizes.

A.3.1 Complete

The Complete benchmark, shown in Listing A.1, aims to concurrently test the most useful features of our solution, including (1) speculative accesses to scalar data with different sizes, and (2) speculative accesses to elements that are part of more complex data structures. While executing this loop in parallel, all the iterations lead to dependence violations.

A.3.2 Tough

The Tough benchmark, depicted in Listing A.2, was designed to heavily test the robustness of our solution and of the underlying consistency protocol used. All of its iterations perform a load and a store on the same speculative data structure, with almost no computational load on private variables.

A.3.3 Fast

The Fast benchmark, shown in Listing A.3, has been designed to test the efficiency of the speculative scheduling mechanism. In this benchmark, only two of the 180 000 iterations (0.001%) lead to a dependence violation. Note that these two dependences are enough to prevent the compile-time parallelization of this loop.

```
#define NITER 6000
int array[MAX], array2[MAX];
struct card{ int field; };
struct card p1 = {3}, p2 = {99999}, p3 = {11111};
char aux_char = ''a;
double aux_double = 3.435;
int i, j;

/*...*/

#pragma omp parallel for default(none) \
  private(i,j) shared(array1,p2) \
speculative(p1,p3,aux_char,aux_double,array2)
for ( i = 0 ; i < NITER ; i++ ) {
  for ( j = 0 ; j < NITER ; j++ ) {

    if (i <= 1000)
      p1.field = array[i%4] + j;
    else
      array2[i%4] = p1.field;

    if (i > 2000)
      aux_char = i%20 + 48 + aux_char%48;
    else
      aux_char = i%20 + array[i%4]%10 + 48;

    if (i > 1500)
      aux_double = array[i%4]/(i+1) + aux_double;
    else
      array2[i%4] = (int) (aux_double / i*j) +
                          (array2[(i+j)%4] +i*j)%1234545;
    if (i*j > 10000)
      p1 = p2;
    else
      p3 = p1;
  }
}
```

Listing A.1: Code of the ‘Complete’ synthetic benchmark.

```
#define NITER 1000000
#define MAX 100
int array[MAX];

/*...*/

#pragma omp parallel default(none) \
  private(P) \
  speculative(array)
for ( P = 0 ; P < NITER ; P++ ) {
  Q = P % (MAX) + 1;
  aux = array[Q-1];
  Q = (4 * aux) % (MAX) + 1;
  array[Q-1] = aux;
}
```

Listing A.2: Code of the 'Tough' synthetic benchmark.

```
#define NITER 180000
#define MAX 4

int array[MAX];
int i,j,k;
int spec1=0, spec2=0;
int iter1, iter2;

/*...*/

#pragma omp parallel default(none) \
  private(i,k) shared(array,iter1,iter2) \
speculative(spec1,spec2)
for ( i = 0 ; i < NITER ; i++ ) {
  if (i == iter1)
    j = spec1;
  if (i == iter2)
    j = spec2;
  for (k = 0; k<array[i%MAX]+j; k++) {

    if (k >= 179900)

      spec1 = k + array[(i+k)%MAX];

    if (k <= 1200)
      spec2 = array[i%MAX];
  }
  if (i == NITER-1)
    spec1 = spec2;
}
```

Listing A.3: Code of the ‘Fast’ synthetic benchmark.

Appendix B

Example of manual use of the TLS runtime library

This chapter completes the description of the TLS runtime library seen in the Chapter 3, detailing the functions and variables needed to parallelize a given code manually. In addition, it provides an example of use of the speculative library which eases readers' learning process.

B.1 Initialization of the engine

specbegin(UINT maxIterations)

The speculative library developed in this work uses some data structures (described in Chapter 3), which should be initialized before executing any speculative code. To do so, we have defined a function called `specbegin()`. It only requires a single argument which is the number of iterations of the target loop. Let us enumerate the responsibilities of this function.

- Initialize the number of iterations to execute in the loop.
- Clean the number of attempts of re-executions for this execution of the block (this number is required for scheduling techniques).
- Initialize each position of the structure used to manage threads. If it is the first time that the function is called:
  - Allocate the memory needed by the dynamic data structures used by the speculative library.
- Initialize the structures of the Indirection matrices.
- Set the state of the first threads to RUNNING, and consequently, establish the non-speculative and most-speculative pointers.
- Schedule iterations (depending on the dynamic or static approach followed, it schedules the whole of them, or only a few for starting the execution).
- Initialize counters for measurements.

B.2 Use of the engine and variable settings

There are a number of variables whose definition is mandatory before executing any code. They are defined through constants in the source files of the library. Some of them are directly declared in header files because they are not frequently modified. The most dependent on the target code, on the contrary, are set adding parameters to the compilation tool `make`. Let us enumerate the most important parameters.

- *wsize*: window size, that is, the number of slots of the sliding window. This parameter is not set as a parameter but as a constant since it is not normally changed.
- *threads*: indicate the number of threads to execute the problem.

- *blocksize*: specifies the size of each block of iterations. Either dynamic or static approaches can be used (see Chapter 5 for more information). Details of how to use this parameter can be found in [9].
- *maxiter*: Maximum number of iterations of the loop to be parallelized.
- *maxpointer*: specifies the maximum number of elements which are speculative.
- *mask*: specifies the size of the mask used (see Section 3.6).

A template for a correct execution of ATLaS could be the following:

```
$ make threads=4 maxiter=1000000 blocksize=100 maxpointer=101 mask=127
```

Once implemented the values of these constants some changes should be applied in the original code. We should include OpenMP library, and the library of the TLS library called '`specEngine.h`'. After that, it is also necessary to call the initializing function *specstart(iterations)*), and set the number of threads to be used. To do so we should call the function *omp_set_num_threads(threads)*. Then we should classify variables into two types: `private` or `shared` (this is one of the requirements of OpenMP [40, 59, 193]).

Some variables used inside speculative library are always classified in the same way:

- *Private*: current, tid, retflag.
- *Shared*: wheel_ns, wheel_ms, wheel, upper_limit, varblock.

Note that speculative variables should be labeled as shared.

Once classified every variable, all accesses to speculative ones should be modified with the corresponding functions, specifically, *specload()* when the variable is read, or *specstore()* when the variable is written. These functions are described in-depth in 3.5.

Finally, in order to support the partial commit operation, we should declare the function *threadend()* at the end of the loop. This function is responsible to manage the sliding window, and commit it following the sequential order. Furthermore, it checks if there are any more blocks to be executed, in which case it assigns them.

Let us see the explained process with the aid of an example.

B.3 An example of use

In order to know how to manually use the TLS runtime library, we are going to show an example. The application of the example[1] (developed using C) will only have a single speculative load, and a single speculative store to simplify the process.

[1] Note that the example application provided is the Tough synthetic benchmark seen detailed at section A.3.2

B.3.1 Sequential application

Listing B.1 shows the sequential version of the application.

First of all let us describe the input set used. We developed a specific application which produces random numbers in order to produce an input set file with a million of random numbers. In the code shown in the Listing B.1 from line 19 to 30 the file is read, obtaining random numbers. In spite of the fact that numbers are random, input file will always be the same, therefore, similar experimental results will be produced, achieving a deterministic version of this application. Note that numbers of the file must be higher than 0 because they are used as the indexes of a vector.

Once the file is read and their data are stored at the variable `vector`, the main loop start its `NITER` iterations (from line 32 to 40). Then, at line 37, an element of the vector is read, and also, at line 39, another one is written. Note that being `Q` the variable which stores the index of the element to be load or stored:

- In the case of load operation, the value of `Q` is equal to the remainder between current iteration and the maximum size of the vector plus one.
- In the case of store operation the value of `Q` is equal to the remainder between the value load, multiplied by four and the maximum size of the vector plus one.

The value of `Q` is unknown at compile time because it depends on the value read from the input file. Therefore, compilers cannot guarantee concurrent execution of some iterations without errors, so, the loop of the line 19 is not parallelized. In these kind of codes we can take advantage of speculative parallelization, and use our TLS runtime library.

Finally, from line 42 to 45 data calculated are checked by summing `vector` elements.

It may be interesting to describe the meaning of the variables used in the application before starting the explanation of the process followed to speculatively parallelize this code.

- *NITER*: The number of iterations of the loop.
- *MAX*: Vector size.
- *aux*: Used to store the value of an element of the vector.
- *Q*: Used to save indexes of the vector.
- *P*: It indicates current iteration of the loop.
- *sum*: It stores the sum of the elements of the vector at the end of the execution. It is used only to check the results.

```c
// Synthetic application written in C //
// Requirements: input values should be higher than 0 //
#include <stdio.h>
#include <stdlib.h>
// Vector size
#define MAX 100
int vector[MAX];
// Number of iterations
#define NITER 1000000

int main() {
  // Local variables: P = current iteration, Q = vector index
  int P, Q, aux, i, sum=0;
  FILE *file;
  if ((file = fopen("rand1000000.in", "r")) == NULL) {
    printf ( "Error opening the file \n " );
    exit(0);
  } else {
    fscanf(file, "%d", &aux);
    for ( i = 0 ; i < MAX ; i++) {
      vector[i] = aux;
      fscanf(file, "%d", &aux);
    }
    fclose (file);
  }

  // Loop can not be parallelized because the index of the elements
  // of the vector written depends on the input values of the file
  for ( P = 1 ; P <= NITER ; P++ ) {
    Q = P % (MAX+1);
    aux = vector[Q-1];
    Q = (4*aux)%(MAX+1);
    vector[Q-1] = aux;
  }  // END for

  printf(" Vector results \n");
  for ( i = 0; i < MAX; i++)
    sum = sum + vector[i];
  printf("%d\n",sum);
}
```

Listing B.1: Example of manually speculative parallelization: Sequential application.

B.3.2 Speculative Parallelization of the sequential application

In order to speculatively parallelize a sequential code there exist some questions that have to be answered.

- *What lines should be parallelized?* Firstly, it is necessary to know what are the lines of the code that can be parallelized. It is a simple step in this example because there is a single loop. If there had been several loops, all of them could have been speculatively parallelized. Note that it is recommended that loops to be parallelized consist of a high number of iterations in order to extract better performances. Obtaining good performance is not the objective of the example shown, but to show how an application could be parallelized.
  In the case of nested loops only one of the loops can be parallelized, the outermost, the innermost, or any of the intermediate loops.

- *Which are the speculative variables?* This question refers to those shared variables of the loop that can induce dependence violations during a parallel execution. In the example seen, four variables are used inside the loop, namely, `P`, `Q`, `aux` and `vector`. The first three variables modify their values at the beginning of the loop, before their usage. Therefore, they do not induce dependence violations and can be considered private variables. On the other hand, the fourth variable is shared by all iterations of the loop. If, while a given thread writes the position `k` of the vector, another one reads the same position `k`, a dependence violation arises. So, the speculative variable is `vector`, and its accesses should be protected.

Once answered these questions, the modification of the sequential application can be started. First of all, we have to link the TLS runtime library files to the application path.

- *specEngine.h*. This is the header file in which the main functions and structures of the speculative library are defined.

- *speccode.c*. It contains the implementation of the main functions used by the speculative library.

- *user_parameters.h*. This file is used to check that all the required parameters are defined correctly.

Once configured the parameters of the library, the original code can be modified. First, we should declare the following two headers.

```
// speccode: OpenMP header file included
#include <omp.h>
```

```
// speccode: Main header file with all common variables
#include "specEngine.h"
```

The next step will be initialize the structures used by the library, and explicitly set the number of threads.

```
// speccode: OMP threading directive
omp_set_num_threads(threads);
// Initialize data structures needed to speculative parallelism
specbegin(NITER);
```

Just before the beginning of the loop the following lines, related to the use of OpenMP, should be written.

```
#pragma omp parallel default(none)      \
  private(Q, P, aux, iteration, ini, \
      current, tid, retflag)    \
  shared(array, wheel_ns, wheel_ms,    \
      wheel, upper_limit, varblock)
  {
#pragma omp for schedule(static)
```

Note that variables called *iteration* and *ini* are auxiliary and might be omitted.

Line 87 contains the OpenMP directive used to mark the start of the `for` loop which will be parallelized. From line 87 to 91 the clauses of directive `parallel for` are found. The `default(none)` clause indicates to OpenMP library what are the classification of variables that will be performed manually, specifically, that no variable should be classified automatically. Finally, line 93 indicates that the size of the chunks of iterations used will be static (it does not mean that we cannot use dynamic scheduling in the speculative execution, it is just an indicator to the OpenMP scheduler).

The loop statement:

```
for (P=1; P<=NITER; P++)
```

is replaced by:

```
//for (P = 1; P <= NITER; P++)
for (tid = 0; tid <= threads - 1; tid++) {

  ini = 1;
  current = tid;
  iteration = varblock[0][current]+ini;

  // If there is no work for this thread, go out.
  if (iteration > upper_limit - 1)
    goto labelSquash;
```

Hence, it is scheduled a single iteration of the transformed loop to the threads launched. Note that *ini* is an auxiliary variable which marks the initial value to iterate, *current* is the variable used to know the identity of a thread (also is optional), *iteration* is the auxiliary variable used to manage the iteration of the loop to execute (since *varblock* is the data structure which manage the scheduling), and *upper_limit* is the variable which contains the limit of the loop, in this case `NITER`.

The following step will be search all lines where a speculative variable is used. In this way, `vector` has been used in lines 37 and 39 of the sequential code. The first one is a load operation, therefore, it is replaced by the *specload()* function (detailed in section 3.5.1).

```
//************************************************************************
// speccode: speculative load. Original line:
//    aux = vector[Q-1];
    if (specload((void *) &(vector[Q-1]), sizeof (vector[Q-1]), current,
                 (void *) &aux) == -1)
        earlySquash();
//************************************************************************
```

The *earlySquash()* function is also described in 3.5.1.

The second operation related to speculative variables is a store in the line 39 of the sequential application. So it is modified (*specstore()* function is detailed in section 3.5.2).

```
//************************************************************************
// speccode: speculative store. Original line:
//  vector[Q-1] = aux;
    specstore((void *) &(vector[Q-1]), sizeof (vector[Q-1]), current,
              (void *) &aux);
//************************************************************************
```

Finally, we have to check if we have more iterations left to be executed, or otherwise call the function which manages commitments, that is, *threadend()*. This operations are performed with the following lines.

```
  if ((iteration != varblock[1][current] + ini) && (iteration < NITER - 1)) {
    iteration = iteration + 1;
    goto labelStartIteration;
  }

  /* Thread done, perform the commit */
labelSquash:
  retflag = threadend(&current);

  /* End if job done */
  if (retflag == JOBDONE) goto labelEndLoop;
    if (retflag == JOBTODO) {
    /*  Set loop variable */
    iteration = varblock[0][current] + ini;
    goto labelStartIteration;
  }
```

Recall that *threadend()* is the function responsible of managing the sliding window, committing it following the sequential order (call the *commit_or_discard()* function detailed in section 3.5.3), and, if there are more iterations, it schedules some of them. Also note that *threadend()* requires an argument, the identity of the thread that calls the function (used to manage speculative order), and return a value (*retflag* in this case) to show whether the thread may execute more iterations or not.

B.3.3 Summary

Steps performed to speculatively parallelize an application can be resumed in:

1. Identify the loop to be parallelized and its number of iterations.
2. Identify speculative variables.
3. Link the files of the speculative library to the path of the application.
4. Add the headers of OpenMP and those related to the speculative library.
5. Initialize the structures of the engine.
6. Classify as private or shared the variables of the loop with the use of the OpenMP directives.
7. Change the original declaration of the loop.
8. Replace load and store operations of the speculative variables with *specload()* and *specstore()* functions respectively.
9. Change the end of the loop with the lines that checks its limits, and with the *threadend()* function which manages commitments.

The source code of this library is included in the package of the ATLaS framework which can be found at `http://atlas.infor.uva.es/`.

Bibliography

[1] W. Richards Adrion. 'Research Methodology in Software Engineering'. In: *ACM SIGSOFT Software Engineering Notes. Summary of the Dagstuhl Workshop on Future Directions in Software Engineering*, vol. 18, no. 1, 1993, pp. 36–37. ACM, 1993. DOI: 10.1145/157397.157399.

[2] Yehuda Afek, Amir Levy and Adam Morrison. 'Software-improved hardware lock elision'. In: *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. ACM. 2014, pp. 212–221.

[3] A. Aiken and A. Nicolau. 'Optimal loop parallelization'. In: *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. PLDI '88. Atlanta, Georgia, USA: ACM, 1988, pp. 308–317. ISBN: 0-89791-269-1. DOI: 10.1145/53990.54021. URL: http://doi.acm.org/10.1145/53990.54021.

[4] Haitham Akkary and Michael A. Driscoll. 'A Dynamic Multithreading Processor'. In: *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 31. Dallas, Texas, USA: IEEE Computer Society Press, 1998, pp. 226–236. ISBN: 1-58113-016-3. URL: http://dl.acm.org/citation.cfm?id=290940.290988.

[5] S. Aldea, A. Estebanez, D.R. Llanos and A. Gonzalez-Escribano. 'An OpenMP Extension that Supports Thread-Level Speculation'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, 2015, pp. 1–14. 2015. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2393870.

[6] S. Aldea, D.R. Llanos and A. González-Escribano. 'Towards a Compiler Framework for Thread-Level Speculation'. In: *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*. Ayia Napa, Cyprus: IEEE Computer Society, Feb. 2011, pp. 267–271. DOI: 10.1109/PDP.2011.14.

[7] Sergio Aldea. 'Compile-Time Support for Thread-Level Speculation'. PhD thesis. Valladolid, Spain: University of Valladolid, July 2014.

[8] Sergio Aldea, Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'A New GCC Plugin-Based Compiler Pass to Add Support for Thread-Level Speculation into OpenMP'. English. In: *Euro-Par 2014 Parallel Processing*. Ed. by Fernando Silva, Inês Dutra and Vítor Santos Costa. Vol. 8632. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 234–245. ISBN: 978-3-319-09872-2. DOI: 10.1007/978-3-319-09873-9_20. URL: http://dx.doi.org/10.1007/978-3-319-09873-9_20.

[9] Sergio Aldea, Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. *ATLaS: Applied Thread-Level Speculation - User Manual*. 2015.

[10] Sergio Aldea, Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Study and Evaluation of Transactional Memory approaches with a Software Thread-Level Speculation Framework'. In: *IEEE Transactions on Parallel and Distributed Systems*. To be submitted.

[11] Sergio Aldea, Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Una extensión para OpenMP que soporta paralelización especulativa'. In: *Proceedings of the XXV Jornadas de Paralelismo*. Valladolid, Spain, Sept. 2014. ISBN: 978-84-697-0329-3.

[12] Sergio Aldea, Diego R. Llanos and Arturo Gonzalez-Escribano. 'The Bonafide C Analyzer: Automatic Loop-level Characterization and Coverage Measuremen'. English. In: *The Journal of Supercomputing*, vol. 68, no. 3, 2014, pp. 1378–1401. Springer US, 2014. ISSN: 0920-8542. DOI: 10.1007/s11227-014-1091-3. URL: http://dx.doi.org/10.1007/s11227-014-1091-3.

[13] Sergio Aldea, Diego R. Llanos and Arturo González-Escribano. 'Support for Thread-Level Speculation into OpenMP'. English. In: *OpenMP in a Heterogeneous World*. Ed. by BarbaraM. Chapman, Federico Massaioli, MatthiasS. Müller and Marco Rorro. Vol. 7312. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 275–278. ISBN: 978-3-642-30960-1. DOI: 10.1007/978-3-642-30961-8_25. URL: http://dx.doi.org/10.1007/978-3-642-30961-8_25.

[14] José Nelson Amaral, Renee Elio, Jim Hoover, Ioanis Nikolaidis, Mohammad Salavatipour, Lorna Stewart and Ken Wong. *About Computing Science Research Methodology*. 2007. URL: webdocs.cs.ualberta.ca/~c603/readings/research-methods.pdf.

[15] *AMD Opteron 6300 Series processor - Quick Reference Guide*. [Last visit: June 2015]. URL: https://www.amd.com/Documents/Opteron_6300_QRG.pdf.

[16] William Aspray. 'The Intel 4004 microprocessor: what constituted invention?' In: *Annals of the History of Computing, IEEE*, vol. 19, no. 3, July 1997, pp. 4–15. July 1997. ISSN: 1058-6180. DOI: 10.1109/85.601727.

[17] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis and Kunle Olukotun. 'The OpenTM Transactional Application Programming Interface'. In: *16th ISCA Proceedings*. IEEE Computer Society, 2007, pp. 376–387.

[18] Joshua E. Barnes. *TREE*. Institute for Astronomy, University of Hawaii. http://www.ifa.hawaii.edu/~barnes/ftp/treecode/. Jan. 1997.

[19] João Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe and Rachid Guerraoui. 'Unifying thread-level speculation and transactional memory'. In: *Proceedings of the 13th International Middleware Conference*. Middleware '12. ontreal, Quebec, Canada: Springer-Verlag New York, Inc., 2012, pp. 187–207. ISBN: 978-3-642-35169-3. URL: http://dl.acm.org/citation.cfm?id=2442626.2442639.

[20] C. Bastoul. 'Code generation in the polyhedral model is easier than you think'. In: *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on*. Sept. 2004, pp. 7–16. DOI: 10.1109/PACT.2004.1342537.

[21] D. Baxter, R. Mirchandaney and J. H. Saltz. 'Run-time Parallelization and Scheduling of Loops'. In: *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '89. Santa Fe, New Mexico, USA: ACM, 1989, pp. 303–312. ISBN: 0-89791-323-X. DOI: 10.1145/72935.72967. URL: http://doi.acm.org/10.1145/72935.72967.

[22] Mikael Berndtsson, Jörgen Hansson, Björn Olsson and Björn Lundell. *Thesis Projects, A Guide for Students in Computer Science and Information Systems*. 2nd. ISBN 978-1848000087. Springer, Oct. 2007.

[23] Arnamoy Bhattacharyya. 'Do Inputs Matter?: Using Data-dependence Profiling to Evaluate Thread Level Speculation in BG/Q'. In: *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*. PACT '13. Edinburgh, Scotland, UK: IEEE Press, 2013, pp. 401–402. *ISBN*: 978-1-4799-1021-2. *URL*: http://dl.acm.org/citation.cfm?id=2523721.2523775.

[24] Arnamoy Bhattacharyya. 'Using Combined Profiling to Decide when Thread Level Speculation is Profitable'. In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12. Minneapolis, Minnesota, USA: ACM, 2012, pp. 483–484. *ISBN*: 978-1-4503-1182-3. *DOI*: 10.1145/2370816.2370908. *URL*: http://doi.acm.org/10.1145/2370816.2370908.

[25] Arnamoy Bhattacharyya and José Nelson Amaral. 'Automatic Speculative Parallelization of Loops Using Polyhedral Dependence Analysis'. In: *Proceedings of the First International Workshop on Code OptimiSation for MultI and Many Cores*. COSMIC '13. Shenzhen, China: ACM, 2013, 1:1–1:9. *ISBN*: 978-1-4503-1971-3. *DOI*: 10.1145/2446920.2446921. *URL*: http://doi.acm.org/10.1145/2446920.2446921.

[26] Anasua Bhowmik and Manoj Franklin. 'A General Compiler Framework for Speculative Multithreading'. In: *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '02. Winnipeg, Manitoba, Canada: ACM, 2002, pp. 99–108. *ISBN*: 1-58113-529-7. *DOI*: 10.1145/564870.564885. *URL*: http://doi.acm.org/10.1145/564870.564885.

[27] Christian Bienia. 'Benchmarking Modern Multiprocessors'. PhD thesis. Princeton University, Jan. 2011.

[28] G. E. Blelloch, G. L. Miller, J. C. Hardwick and D. Talmor. 'Design and Implementation of a Practical Parallel Delaunay Algorithm'. In: *Algorithmica*, vol. 24, no. 3, July 1999, pp. 243–269. July 1999.

[29] Scott Elliott Breach. 'Design and evaluation of a multiscalar processor'. AAI9910432. PhD thesis. The University of Wisconsin - Madison, 1998. *ISBN*: 0-599-31966-6.

[30] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 2nd. USA: Addison-Wesley Publishing Company, 2010. *ISBN*: 0136108040, 9780136108047.

[31] Dick Buttlar and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. "O'Reilly Media, Inc.", 1996.

[32] Srihari Cadambi, Giuseppe Coviello, Cheng-Hong Li, Rajat Phull, Kunal Rao, Murugan Sankaradass and Srimat Chakradhar. 'COSMIC: Middleware for High Performance and Reliable Multiprocessing on Xeon Phi Coprocessors'. In: *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*. HPDC '13. New York, New York, USA: ACM, 2013, pp. 215–226. *ISBN*: 978-1-4503-1910-2. *DOI*: 10.1145/2462902.2462921. *URL*: http://doi.acm.org/10.1145/2462902.2462921.

[33] Panpan Cai, Yiyu Cai, Indhumathi Chandrasekaran and Jianmin Zheng. 'A GPU-Enabled Parallel Genetic Algorithm for Path Planning of Robotic Operators'. English. In: *GPU Computing and Applications*. Ed. by Yiyu Cai and Simon See. Springer Singapore, 2015, pp. 1–13. ISBN: 978-981-287-133-6. DOI: 10.1007/978-981-287-134-3_1. URL: http://dx.doi.org/10.1007/978-981-287-134-3_1.

[34] Harold W Cain, Maged M Michael, Brad Frey, Cathy May, Derek Williams and Hung Le. 'Robust architectural support for transactional memory in the power architecture'. In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM. 2013, pp. 225–236.

[35] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis and Kunle Olukotun. 'STAMP: Stanford Transactional Applications for Multi-Processing'. In: *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. Seattle, WA, USA: IEEE Computer Society, Sept. 2008, pp. 35–46. DOI: 10.1109/IISWC.2008.4636089.

[36] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis and Kunle Olukotun. 'STAMP: Stanford Transactional Applications for Multi-Processing'. In: *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*. Sept. 2008.

[37] Zhen Cao and C. Verbrugge. 'Mixed Model Universal Software Thread-Level Speculation'. In: *Parallel Processing (ICPP), 2013 42nd International Conference on*. Lyon,France: IEEE Computer Society, Oct. 2013, pp. 651–660. DOI: 10.1109/ICPP.2013.79.

[38] Luis Ceze, James Tuck, Josep Torrellas and Calin Cascaval. 'Bulk Disambiguation of Speculative Threads in Multiprocessors'. In: *Proceedings of the 33rd annual international symposium on Computer Architecture*. ISCA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 227–238. ISBN: 0-7695-2608-X. DOI: 10.1109/ISCA.2006.13. URL: http://dx.doi.org/10.1109/ISCA.2006.13.

[39] Luis Ceze, James Tuck, Josep Torrellas and Calin Cascaval. 'Bulk disambiguation of speculative threads in multiprocessors'. In: *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, 2006, pp. 227–238. 2006.

[40] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan and Jeff McDonald. *Parallel Programming in OpenMP*. 1st ed. San Francisco, California, USA: Morgan Kaufmann, Oct. 2000. ISBN: 1558606718.

[41] Sai Charan Koduru, Keval Vora and Rajesh Gupta. 'ABC2: Adaptively Balancing Computation and Communication in a DSM Cluster of Multicores for Irregular Applications'. In: *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE. 2014, pp. 391–400.

[42] Benbin Chen and Donghui Guo. 'A static scheduling scheme of multicore compiler for loop load imbalance in OpenMP'. In: *Anti-counterfeiting, Security, and Identification (ASID), 2014 International Conference on*. Dec. 2014, pp. 1–4. DOI: 10.1109/ICASID.2014.7064954.

[43] Ding Kai Chen, Josep Torrellas and Pen Chung Yew. 'An Efficient Algorithm for the Run-time Parallelization of DOACROSS Loops'. In: *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. Supercomputing '94. Washington, D.C.: IEEE Computer Society Press, 1994, pp. 518–527. ISBN: 0-8186-6605-6. URL: http://dl.acm.org/citation.cfm?id=602770.602857.

[44] M.K. Chen and K. Olukotun. 'Exploiting method-level parallelism in single-threaded Java programs'. In: *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. PACT '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 176–184. DOI: 10.1109/PACT.1998.727190.

[45] Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju and Jenq Kuen Lee. 'Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis'. In: *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '03. San Diego, California, USA: ACM, 2003, pp. 25–36. ISBN: 1-58113-588-2. DOI: 10.1145/781498.781502. URL: http://doi.acm.org/10.1145/781498.781502.

[46] Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu and Pen-Chung Yew. 'Data Dependence Profiling for Speculative Optimizations'. In: *Compiler Construction*. Ed. by Evelyn Duesterwald. Vol. 2985. Lecture Notes in Computer Science. Berlin Heidelberg: Springer, 2004, pp. 57–72. ISBN: 978-3-540-21297-3. DOI: 10.1007/978-3-540-24723-4_5. URL: http://dx.doi.org/10.1007/978-3-540-24723-4_5.

[47] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier and Etienne Rivière. 'Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack'. In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France, 2010, pp. 27–40. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755918.

[48] Marcelo Cintra and Diego R. Llanos. 'Design Space Exploration of a Software Speculative Parallelization Scheme'. In: *IEEE Trans. on Paral. and Distr. Systems*, vol. 16, no. 6, June 2005, pp. 562–576. June 2005.

[49] Marcelo Cintra and Diego R. Llanos. 'Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors'. In: *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '03. San Diego, California, USA: ACM, 2003, pp. 13–24. ISBN: 1-58113-588-2. DOI: 10.1145/781498.781501. URL: http://doi.acm.org/10.1145/781498.781501.

[50] Marcelo Cintra, Diego R. Llanos and Belén Palop. 'Speculative Parallelization of a Randomized Incremental Convex Hull Algorithm.' In: *ICCSA 2004: Proc. Intl. Conf. on Computer Science and its Applications*. LNCS 3045, ISSN 0302-9743. Perugia, Italy: Springer-Verlag, May 2004, pp. 188–197.

[51] Marcelo Cintra, José F. Martínez and Josep Torrellas. 'Architectural Support for Scalable Speculative Parallelization in Shared-memory Multiprocessors'. In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ISCA '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 13–24. ISBN: 1-58113-232-8. DOI: 10.1145/339647.363382. URL: http://doi.acm.org/10.1145/339647.363382.

[52] Marcelo Cintra and Josep Torrellas. 'Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors'. In: *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. HPCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 43–. URL: http://dl.acm.org/citation.cfm?id=874076.876479.

[53] K. L. Clarkson, K. Mehlhorn and R. Seidel. 'Four results on randomized incremental constructions'. In: *Comput. Geom. Theory Appl.* Vol. 3, no. 4, 1993, pp. 185–212. 1993.

[54] Michele Co, Dee A. B. Weikle and Kevin Skadron. 'Evaluating Trace Cache Energy Efficiency'. In: *ACM Trans. Archit. Code Optim.* Vol. 3, no. 4, Dec. 2006, pp. 450–476. New York, NY, USA: ACM, Dec. 2006. *ISSN*: 1544-3566. *DOI*: 10.1145/1187976.1187980. *URL*: http://doi.acm.org/10.1145/1187976.1187980.

[55] L. Codrescu and D.S. Wills. 'Architecture of the Atlas chip-multiprocessor: dynamically parallelizing irregular applications'. In: *Proceedings of the 1999 IEEE International Conference on Computer Design*. ICCD '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 428–435. *ISBN*: 0-7695-0406-X. *DOI*: 10.1109/ICCD.1999.808577.

[56] Lucian Codrescu, D. Scott Wills and James Meindl. 'Architecture of the Atlas Chip Multiprocessor: Dynamically Parallelizing Irregular Applications'. In: *IEEE Transactions on Computers*, vol. 50, no. 1, Jan. 2001, pp. 67–82. Washington, DC, USA: IEEE Computer Society, Jan. 2001. *ISSN*: 0018-9340. *DOI*: 10.1109/12.902753. *URL*: http://dx.doi.org/10.1109/12.902753.

[57] Christopher B. Colohan, Anastassia Ailamaki, J. Gregory Steffan and Todd C. Mowry. 'Tolerating Dependences Between Large Speculative Threads Via Sub-Threads'. In: *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. ISCA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 216–226. *ISBN*: 0-7695-2608-X. *DOI*: 10.1109/ISCA.2006.43. *URL*: http://dx.doi.org/10.1109/ISCA.2006.43.

[58] Tim Cramer, Dirk Schmidl, Michael Klemm and Dieter an Mey. 'OpenMP Programming on Intel R Xeon Phi TM Coprocessors: An Early Performance Comparison'. In: 2012. 2012.

[59] Leonardo Dagum and Ramesh Menon. 'OpenMP: An Industry-Standard API for Shared-Memory Programming'. In: *IEEE Comput. Sci. Eng.* Vol. 5, no. 1, Jan. 1998, pp. 46–55. Los Alamitos, CA, USA: IEEE Computer Society Press, Jan. 1998. *ISSN*: 1070-9924. *DOI*: 10.1109/99.660313. *URL*: http://dx.doi.org/10.1109/99.660313.

[60] Wenbo Dai, Hong An, Qi Li, Gongming Li, Bobin Deng, Shilei Wu, Xiaomei Li and Yu Liu. 'A Priority-Aware NoC to Reduce Squashes in Thread Level Speculation for Chip Multiprocessors'. In: *Proceedings of the 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*. ISPA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 87–92. *ISBN*: 978-0-7695-4428-1. *DOI*: 10.1109/ISPA.2011.21. *URL*: http://dx.doi.org/10.1109/ISPA.2011.21.

[61] Francis Dang, Hoo Yu and Lawrence Rauchwerger. 'The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops'. In: *Proceedings of the 16th International Parallel and Distributed Processing Symposium*. IPDPS '02. Washington, DC, USA: IEEE Computer Society, Apr. 2002, pp. 318–327. *ISBN*: 0-7695-1573-8. *DOI*: 10.1109/IPDPS.2002.1015493. *URL*: http://dl.acm.org/citation.cfm?id=645610.662024.

[62] Bobin Deng, Hong An, Qi Li, Gongming Li and Mengjie Mao. 'Value Predicted LogSPoTM: Improve the Parallesim of Thread Level System by Using a Value Predictor'. In: *Proceedings of the 2012 IEEE/ACIS 11th International Conference on Computer and Information Science*. ICIS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 130–135. *ISBN*: 978-0-7695-4694-0. *DOI*: 10.1109/ICIS.2012.116. *URL*: http://dx.doi.org/10.1109/ICIS.2012.116.

[63] L. Devroye, E. P. Mücke and Binhai Zhu. 'A Note on Point Location in Delaunay Triangulations of Random Points'. English. In: *Algorithmica*, vol. 22, no. 4, 1998, pp. 477–482. Springer-Verlag, 1998. *ISSN*: 0178-4617. *DOI*: 10.1007/PL00009234. *URL*: http://dx.doi.org/10.1007/PL00009234.

[64] G. Diamos and S. Yalamanchili. 'Speculative execution on multi-GPU systems'. In: *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. Atlanta, GA, USA: IEEE Computer Society, Apr. 2010, pp. 1–12. *DOI*: 10.1109/IPDPS.2010.5470427.

[65] Pedro Díaz, Diego R. Llanos and Belén Palop. 'Parallelizing 2D-Convex Hulls on clusters: Sorting matters'. In: *Proc. XV Jornadas de Paralelismo*. ISBN 84-8240-714-7. Almería, Spain, Sept. 2004, pp. 247–252.

[66] Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum and Marek Olszewski. *Early experience with a commercial hardware transactional memory implementation*. Tech. rep. 2009.

[67] Dave Dice, Ori Shalev and Nir Shavit. 'Transactional Locking II'. In: *Proceedings of the 20th International Conference on Distributed Computing*. DISC'06. Stockholm, Sweden, 2006, pp. 194–208. *ISBN*: 3-540-44624-9, 978-3-540-44624-8.

[68] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang and Chengliang Zhang. 'Software behavior oriented parallelization'. In: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '07. San Diego, California, USA: ACM, 2007, pp. 223–234. *ISBN*: 978-1-59593-633-2. *DOI*: 10.1145/1250734.1250760. *URL*: http://doi.acm.org/10.1145/1250734.1250760.

[69] Kaivalya M Dixit. *Overview of the SPEC Benchmarks.* 1993.

[70] Gordana Dodig-Crnkovic. 'Scientific methods in computer science'. In: *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*. 2002, pp. 126–130.

[71] Yong Dong, Juan Chen, Xuejun Yang, Lin Deng and Xuemeng Zhang. 'Energy-Oriented OpenMP Parallel Loop Scheduling'. In: *Parallel and Distributed Processing with Applications, 2008. ISPA '08. International Symposium on*. Dec. 2008, pp. 162–169. *DOI*: 10.1109/ISPA.2008.68.

[72] Jialin Dou and Marcelo Cintra. 'A compiler cost model for speculative parallelization'. In: *ACM Trans. Archit. Code Optim.* Vol. 4, no. 2, June 2007, pp. 71–81. New York, NY, USA: ACM, June 2007. *ISSN*: 1544-3566. *DOI*: 10.1145/1250727.1250732. *URL*: http://doi.acm.org/10.1145/1250727.1250732.

[73] Jialin Dou and Marcelo Cintra. 'Compiler Estimation of Load Imbalance Overhead in Speculative Parallelization'. In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 203–214. *ISBN*: 0-7695-2229-7. *DOI*: 10.1109/PACT.2004.12. *URL*: http://dx.doi.org/10.1109/PACT.2004.12.

[74] Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'A survey on Thread-Level Speculation Techniques'. In: *ACM Computing Surveys (CSUR)*. Accepted for publication.

[75] Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros'. In: *Proceedings of the XXIII Jornadas de Paralelismo*. Elche, Alicante, Spain: Servicio de Publicaciones de la Universidad Miguel Hernández, Sept. 2012. *ISBN*: 978-84-695-4471-6.

[76] Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros'. Trabajo de Fin de Grado. Universidad de Valladolid, July 2012.

[77] Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Evaluating the capabilities of the Xeon Phi platform in the context of software-only, thread-level speculation'. In: *Proceedings of the 8th International Symposium on High-level Parallel Programming and Applications*. HLPP '15. Pisa, Italy: ACM, 2015.

[78] Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Improving the Perfomance of a Pointer-Based, Speculative Parallelization Scheme'. MA thesis. Spain: University of Valladolid, July 2013.

[79] Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Improving the Perfomance of a Pointer-Based, Speculative Parallelization Scheme'. In: *Proceedings of the 1st First Congress on Multicore and GPU Programming*. PPGM'14. Granada, Spain, Feb. 2014.

[80] Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'New Data Structures to Handle Speculative Parallelization at Runtime'. In: *Proceedings of the 7th International Symposium on High-level Parallel Programming and Applications*. HLPP '14. Amsterdam, Netherlands: ACM, 2014, pp. 239–258.

[81] Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'New Data Structures to Handle Speculative Parallelization at Runtime'. English. In: *International Journal of Parallel Programming*, 2015, pp. 1–20. Springer US, 2015. *ISSN*: 0885-7458. *DOI*: 10.1007/s10766-014-0347-0. *URL*: http://dx.doi.org/10.1007/s10766-014-0347-0.

[82] Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Paralelización especulativa de un algoritmo para el menor círculo contenedor'. Proyecto de Fin de Carrera. Universidad de Valladolid, Sept. 2011.

[83] Alvaro Estebanez, Diego R. Llanos, David Orden and Belen Palop. 'Moody Scheduling for Speculative Parallelization'. English. In: *Euro-Par 2015: Parallel Processing*. Ed. by Jesper Larsson Träff, Sascha Hunold and Francesco Versaci. Vol. 9233. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015, pp. 135–146. *ISBN*: 978-3-662-48095-3. *DOI*: 10.1007/978-3-662-48096-0_11. *URL*: http://dx.doi.org/10.1007/978-3-662-48096-0_11.

[84] Xu Fan, Shen Li and Wang Zhiying. 'HVD-TLS: A Novel Framework of Thread Level Speculation'. In: *Proceedings of the 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. TRUSTCOM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1912–1917. *ISBN*: 978-0-7695-4745-9. *DOI*: 10.1109/TrustCom.2012.176. *URL*: http://dx.doi.org/10.1109/TrustCom.2012.176.

[85] Jianbin Fang, Henk Sips, LiLun Zhang, Chuanfu Xu, Yonggang Che and Ana Lucia Varbanescu. 'Test-driving Intel Xeon Phi'. In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ICPE '14. Dublin, Ireland: ACM, 2014, pp. 137–148. *ISBN*: 978-1-4503-2733-6. *DOI*: 10.1145/2568088.2576799. *URL*: http://doi.acm.org/10.1145/2568088.2576799.

[86] Pascal Felber, Christof Fetzer, Patrick Marlier and Torvald Riegel. 'Time-based Software Transactional Memory'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, Dec. 2010, pp. 1793–1807. Dec. 2010. *ISSN*: 1045-9219. *DOI*: 10.1109/TPDS.2010.49.

[87] Pascal Felber, Christof Fetzer and Torvald Riegel. 'Dynamic Performance Tuning of Word-Based Software Transactional Memory'. In: *PPoPP '08 Proceedings*. 2008, pp. 237–246.

[88] Pascal Felber, Etienne Riviere, Walther Maldonado Moreira, Derin Harmanci, Patrick Marlier, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Adrian Cristal, Ibrahim Hur et al. 'The velox transactional memory stack'. In: *Micro, IEEE*, vol. 30, no. 5, 2010, pp. 76–87. 2010.

[89] Min Feng, Rajiv Gupta and Laxmi N Bhuyan. 'Optimistic Parallelism on GPUs'. In: *Languages and Compilers for Parallel Computing*. Springer, 2014, pp. 3–18.

[90] Min Feng, Rajiv Gupta and Laxmi N. Bhuyan. 'Speculative Parallelization on GPGPUs'. In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New Orleans, Louisiana, USA: ACM, 2012, pp. 293–294. *ISBN*: 978-1-4503-1160-1. *DOI*: 10.1145/2145816.2145860. *URL*: http://doi.acm.org/10.1145/2145816.2145860.

[91] Min Feng, Rajiv Gupta and Yi Hu. 'SpiceC: Scalable Parallelism via Implicit Copying and Explicit Commit'. In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. PPoPP '11. San Antonio, TX, USA: ACM, 2011, pp. 69–80. *ISBN*: 978-1-4503-0119-0. *DOI*: 10.1145/1941553.1941564. *URL*: http://doi.acm.org/10.1145/1941553.1941564.

[92] Min Feng, Rajiv Gupta and Iulian Neamtiu. 'Effective parallelization of loops in the presence of I/O operations'. In: *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: ACM, 2012, pp. 487–498. *ISBN*: 978-1-4503-1205-9. *DOI*: 10.1145/2254064.2254122. *URL*: http://doi.acm.org/10.1145/2254064.2254122.

[93] Min Feng, Changhui Lin and Rajiv Gupta. 'PLDS: Partitioning Linked Data Structures for Parallelism'. In: *ACM Trans. Archit. Code Optim.* Vol. 8, no. 4, Jan. 2012, 38:1–38:21. New York, NY, USA: ACM, Jan. 2012. *ISSN*: 1544-3566. *DOI*: 10.1145/2086696.2086717. *URL*: http://doi.acm.org/10.1145/2086696.2086717.

[94] Manoj Franklin and Gurindar S. Sohi. 'ARB: A Hardware Mechanism for Dynamic Reordering of Memory References'. In: *IEEE Trans. Comput.* Vol. 45, no. 5, May 1996, pp. 552–571. Washington, DC, USA: IEEE Computer Society, May 1996. *ISSN*: 0018-9340. *DOI*: 10.1109/12.509907. *URL*: http://dx.doi.org/10.1109/12.509907.

[95] Ricardo Freitas. 'Scientific Research Methods and Computer Science'. In: *Proceedings of the MAP-I Seminars Workshop*. Porto, Portugal, 2009.

[96] Lin Gao, Lian Li, Jingling Xue and Pen-Chung Yew. 'SEED: A Statically-Greedy and Dynamically-Adaptive Approach for Speculative Loop Execution'. In: *IEEE Transactions on Computers*, vol. 62, no. 5, 2013, pp. 1004–1016. Los Alamitos, CA, USA: IEEE Computer Society, 2013. *ISSN*: 0018-9340. *DOI*: http://doi.ieeecomputersociety.org/10.1109/TC.2012.41.

[97] Alvaro Garcia-Yaguez, Diego R. Llanos and Arturo Gonzalez-Escribano. 'Squashing Alternatives for Software-based Speculative Parallelization'. In: *IEEE Transactions on Computers*, vol. 63, no. 7, 2014, pp. 1826–1839. Los Alamitos, CA, USA: IEEE Computer Society, 2014. *ISSN*: 0018-9340.

[98] Álvaro García-Yágüez, Diego R. Llanos and Arturo González-Escribano. 'Exclusive squashing for thread-level speculation'. In: *Proceedings of the 20th international symposium on High performance distributed computing*. HPDC '11. San Jose, California, USA: ACM, 2011, pp. 275–276. ISBN: 978-1-4503-0552-5. DOI: 10.1145/1996130.1996172. URL: http://doi.acm.org/10.1145/1996130.1996172.

[99] Alvaro Garcia-Yaguez, Diego R. Llanos, David Orden and Belen Palop. 'Paralelización especulativa de la triangulación de Delaunay'. In: *Proc. XX Jornadas de Paralelismo*. A Coruña, Spain, Sept. 2009.

[100] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger and Josep Torrellas. 'Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors'. In: *ACM Trans. Archit. Code Optim.* Vol. 2, no. 3, Sept. 2005, pp. 247–279. New York, NY, USA: ACM, Sept. 2005. ISSN: 1544-3566. DOI: 10.1145/1089008.1089010. URL: http://0-doi.acm.org.almena.uva.es/10.1145/1089008.1089010.

[101] María Jesús Garzarán, Milos Prvulovic, Víctor Viñals, José María Llabería, Lawrence Rauchwerger and Josep Torrellas. 'Using Software Logging to Support Multi-Version Buffering in Thread-Level Speculation'. In: *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. PACT '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 170–. ISBN: 0-7695-2021-9. URL: http://dl.acm.org/citation.cfm?id=942806.943847.

[102] GNU Project. *GCC Internals*. http://gcc.gnu.org/onlinedocs/gccint/. [Last visit: June 2015].

[103] GNU Project. *GCC, the GNU Compiler Collection*. http://gcc.gnu.org/. [Last visit: June 2015].

[104] Arturo González-Escribano, Diego R. Llanos, David Orden and Belén Palop. 'Parallelization alternatives and their performance for the convex hull problem'. In: *Applied Mathematical Modelling, special issue on Parallel and Vector Processing in Science and Engineering*, vol. 30, no. 7, July 2006. ISSN 0307-904X, pp. 563–577. July 2006.

[105] S. Gopal, T. N. Vijaykumar, J.E. Smith and G.S. Sohi. 'Speculative versioning cache'. In: *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. HPCA '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 195–205. ISBN: 0-8186-8323-6. DOI: 10.1109/HPCA.1998.650559. URL: http://dl.acm.org/citation.cfm?id=822079.822729.

[106] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601902.

[107] Rui Guo, Hong An, Ruiling Dou, Ming Cong, Yaobin Wang and Qi Li. 'LogSPoTM: a scalable thread level speculation model based on transactional memory'. In: *Computer Systems Architecture Conference, 2008. ACSAC 2008. 13th Asia-Pacific*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 1–8. DOI: 10.1109/APCSAC.2008.4625443.

[108] Manish Gupta and Rahul Nim. 'Techniques for Speculative Run-time Parallelization of Loops'. In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. SC '98. San Jose, CA: IEEE Computer Society, 1998, pp. 1–12. ISBN: 0-89791-984-X. URL: http://dl.acm.org/citation.cfm?id=509058.509070.

[109] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown. 'MiBench: A free, commercially representative embedded benchmark suite'. In: *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. Washington, DC, USA: IEEE Computer Society, Dec. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[110] Torben Hagerup. 'Allocating Independent Tasks to Parallel Processors: An Experimental Study.' In: *J. Parallel Distrib. Comput.* Vol. 47, no. 2, 1997, pp. 185–197. 1997.

[111] Robert H. Halstead Jr. 'MULTILISP: a language for concurrent symbolic computation'. In: *ACM Trans. Program. Lang. Syst.* Vol. 7, no. 4, Oct. 1985, pp. 501–538. New York, NY, USA: ACM, Oct. 1985. ISSN: 0164-0925. DOI: 10.1145/4472.4478. URL: http://doi.acm.org/10.1145/4472.4478.

[112] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen and Kunle Olukotun. 'The Stanford Hydra CMP'. In: *IEEE Micro*, vol. 20, no. 2, Mar. 2000, pp. 71–84. Los Alamitos, CA, USA: IEEE Computer Society Press, Mar. 2000. ISSN: 0272-1732. DOI: 10.1109/40.848474. URL: http://dx.doi.org/10.1109/40.848474.

[113] Lance Hammond, Mark Willey and Kunle Olukotun. 'Data Speculation Support for a Chip Multiprocessor'. In: *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VIII. San Jose, California, USA: ACM, 1998, pp. 58–69. ISBN: 1-58113-107-0. DOI: 10.1145/291069.291020. URL: http://doi.acm.org/10.1145/291069.291020.

[114] T. Harris, A. Cristal, O.S. Unsal, E. Ayguade, F. Gagliardi, B. Smith and M. Valero. 'Transactional Memory: An Overview'. In: *Micro, IEEE*, vol. 27, no. 3, May 2007, pp. 8–29. May 2007. ISSN: 0272-1732. DOI: 10.1109/MM.2007.63.

[115] Tim Harris, James Larus and Ravi Rajwar. *Transactional Memory, 2nd Edition*. 2nd. San Rafael, CA, USA: Morgan and Claypool Publishers, 2010. ISBN: 1608452352, 9781608452354.

[116] John L. Henning. 'SPEC CPU Suite Growth: An Historical Perspective'. In: *SIGARCH Comput. Archit. News*, vol. 35, no. 1, Mar. 2007, pp. 65–68. New York, NY, USA: ACM, Mar. 2007. ISSN: 0163-5964. DOI: 10.1145/1241601.1241615. URL: http://doi.acm.org/10.1145/1241601.1241615.

[117] John L. Henning. 'SPEC CPU2006 Benchmark Descriptions'. In: *SIGARCH Comput. Archit. News*, vol. 34, no. 4, Sept. 2006, pp. 1–17. New York, NY, USA: ACM, Sept. 2006. ISSN: 0163-5964. DOI: 10.1145/1186736.1186737. URL: http://doi.acm.org/10.1145/1186736.1186737.

[118] Maurice Herlihy and J. Eliot B. Moss. 'Transactional memory: architectural support for lock-free data structures'. In: *20th ISCA Proceedings*. ISCA '93. ACM, 1993, pp. 289–300. ISBN: 0-8186-3810-9. DOI: 10.1145/165123.165164. (Visited on 03/03/2013).

[119] Susan Flynn Hummel, Edith Schonberg and Lawrence E. Flynn. 'Factoring: A Method for Scheduling Parallel Loops'. In: *Communications of the ACM*, vol. 35, no. 2, Aug. 1992, pp. 90–100. Aug. 1992.

[120] Susan Flynn Hummel, Edith Schonberg and Lawrence E Flynn. 'Factoring: A method for scheduling parallel loops'. In: *Communications of the ACM*, vol. 35, no. 8, 1992, pp. 90–101. ACM, 1992.

[121] IBM. *Thread-Level Speculative Execution for C/C++. IBM XL C/C++ for Blue Gene*. Tech. report. Last access: April 2014. 2012.

[122] Intel. *The Story of the Intel® 4004*. http://www.intel.co.uk/content/www/uk/en/history/museum-story-of-intel-4004.html. [Last visit: June 2015].

[123] *Intel C++ STM Compiler, Prototype Edition*. 2012. URL: https://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition.

[124] *Intel® Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*. URL: https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf.

[125] *Intel®Xeon Phi Product Family: Product Brief*. [Last visit: June 2015]. URL: https://www-ssl.intel.com/content/dam/www/public/us/en/documents/product-briefs/high-performance-xeon-phi-coprocessor-brief.pdf.

[126] N. Ioannou, J. Singer, S. Khan, P. Xekalakis, P. Yiapanis, A. Pocock, G. Brown, M. Luján, I. Watson and M. Cintra. 'Toward a more accurate understanding of the limits of the TLS execution paradigm'. In: *Workload Characterization (IISWC), 2010 IEEE International Symposium on*. Washington, DC, USA: IEEE Computer Society, Dec. 2010, pp. 1–12. DOI: 10.1109/IISWC.2010.5649169.

[127] Nikolas Ioannou and Marcelo Cintra. 'Complementing user-level coarse-grain parallelism with implicit speculative parallelism'. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2011, pp. 284–295.

[128] M.M. Islam, A. Busck, M. Engbom, S. Lee, Michel Dubois and P. Stenstrom. 'Limits on Thread-Level Speculative Parallelism in Embedded Applications'. In: *Eleventh Workshop on Interaction between Compilers and Computer Architectures (INTERACT-11)*. Phoenix, USA, Feb. 2007.

[129] M.M. Islam, A. Busck, M. Engbom, S. Lee, Michel Dubois and P. Stenstrom. 'Loop-level Speculative Parallelism in Embedded Applications'. In: *Parallel Processing, 2007. ICPP 2007. International Conference on*. Washington, DC, USA: IEEE Computer Society, Sept. 2007, pp. 3–13. DOI: 10.1109/ICPP.2007.53.

[130] Quinn Jacobson, Eric Rotenberg and James E. Smith. 'Path-based Next Trace Prediction'. In: *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 30. Research Triangle Park, North Carolina, USA: IEEE Computer Society, 1997, pp. 14–23. ISBN: 0-8186-7977-8. URL: http://dl.acm.org/citation.cfm?id=266800.266802.

[131] Hakbeom Jang, Channoh Kim and Jae W. Lee. 'Practical speculative parallelization of variable-length decompression algorithms'. In: *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*. LCTES '13. Seattle, Washington, USA: ACM, 2013, pp. 55–64. ISBN: 978-1-4503-2085-6. DOI: 10.1145/2465554.2465557. URL: http://doi.acm.org/10.1145/2465554.2465557.

[132] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[133] Alexandra Jimborean. 'Adapting the polytope model dor dynamic and speculative parallelization'. PhD thesis. Strasbourg,France: University of Strasbourg, 2012.

[134] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner and JuanManuel Martinez Caamaño. 'Dynamic and Speculative Polyhedral Parallelization Using Compiler-Generated Skeletons'. English. In: *International Journal of Parallel Programming*, vol. 42, no. 4, 2013, pp. 529–545. USA: Springer US, 2013. *ISSN*: 0885-7458. *DOI*: 10.1007/s10766-013-0259-4. *URL*: http://dx.doi.org/10.1007/s10766-013-0259-4.

[135] Alexandra Jimborean, Philippe Clauss, Benoît Pradelle, Luis Mastrangelo and Vincent Loechner. 'Adapting the Polyhedral Model As a Framework for Efficient Speculative Parallelization'. In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New Orleans, Louisiana, USA: ACM, 2012, pp. 295–296. *ISBN*: 978-1-4503-1160-1. *DOI*: 10.1145/2145816.2145861. *URL*: http://doi.acm.org/10.1145/2145816.2145861.

[136] Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner and Philippe Clauss. 'VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework'. In: *Proceedings of the 21st International Conference on Compiler Construction*. CC'12. Tallinn, Estonia: Springer-Verlag, 2012, pp. 220–239. *ISBN*: 978-3-642-28651-3. *DOI*: 10.1007/978-3-642-28652-0_12. *URL*: http://dx.doi.org/10.1007/978-3-642-28652-0_12.

[137] Canming Jin, Yong Yan and Xiaodong Zhang. 'An adaptive loop scheduling algorithm on shared-memory systems'. In: *Parallel and Distributed Processing, 1996., Eighth IEEE Symposium on*. Oct. 1996, pp. 250–257. *DOI*: 10.1109/SPDP.1996.570341.

[138] Youngjoon Jo and Milind Kulkarni. 'Brief Announcement: Locality-aware Load Balancing for Speculatively-parallelized Irregular Applications'. In: *Proceedings of the 22Nd ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '10. Thira, Santorini, Greece: ACM, 2010, pp. 183–185. *ISBN*: 978-1-4503-0079-7. *DOI*: 10.1145/1810479.1810516. *URL*: http://doi.acm.org/10.1145/1810479.1810516.

[139] Chuanle Ke, Lei Liu, Chao Zhang, Tongxin Bai, Brian Jacobs and Chen Ding. 'Safe Parallel Programming using Dynamic Dependence Hints'. In: *OOPSLA'11 Proceedings*. Portland, Oregon, USA: ACM, 2011, pp. 243–258.

[140] Arun Kejariwal, Milind Girkar, Xinmin Tian, Hideki Saito, Alexandru Nicolau, Alexander V. Veidenbaum, Utpal Banerjee and Constantine D. Polychronopoulos. 'Exploitation of Nested Thread-level Speculative Parallelism on Multi-core Systems'. In: *Proceedings of the 7th ACM International Conference on Computing Frontiers*. CF '10. Bertinoro, Italy: ACM, 2010, pp. 99–100. *ISBN*: 978-1-4503-0044-5. *DOI*: 10.1145/1787275.1787302. *URL*: http://doi.acm.org/10.1145/1787275.1787302.

[141] Arun Kejariwal, Milind Girkar, Xinmin Tian, Hideki Saito, Alexandru Nicolau, Alexander V. Veidenbaum, Utpal Banerjee and Constantine D. Ppoluchronopoulos. 'On the Efficacy of Call Graph-level Thread-level Speculation'. In: *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*. WOSP/SIPEW '10. San Jose, California, USA: ACM, 2010, pp. 247–248. *ISBN*: 978-1-60558-563-5. *DOI*: 10.1145/1712605.1712645. *URL*: http://doi.acm.org/10.1145/1712605.1712645.

[142] Arun Kejariwal, Xinmin Tian, Milind Girkar, Wei Li, Sergey Kozhukhov, Utpal Banerjee, Alexander Nicolau, Alexander V. Veidenbaum and Constantine D. Polychronopoulos. 'Tight analysis of the performance potential of thread speculation using spec CPU 2006'. In: *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '07. San Jose, California, USA: ACM, 2007, pp. 215–225. *ISBN*: 978-1-59593-602-8. *DOI*: 10.1145/1229428.1229475. *URL*: http://0-doi.acm.org.almena.uva.es/10.1145/1229428.1229475.

[143] Arun Kejariwal, Xinmin Tian, Wei Li, Milind Girkar, Sergey Kozhukhov, Hideki Saito, Utpal Banerjee, Alexandru Nicolau, Alexander V. Veidenbaum and Constantine D. Polychronopoulos. 'On the performance potential of different types of speculative thread-level parallelism: The DL version of this paper includes corrections that were not made available in the printed proceedings'. In: *Proceedings of the 20th annual international conference on Supercomputing*. ICS '06. Cairns, Queensland, Australia: ACM, 2006, pp. 24–. *ISBN*: 1-59593-282-8. *DOI*: 10.1145/1183401.1183407. *URL*: http://0-doi.acm.org.almena.uva.es/10.1145/1183401.1183407.

[144] Kirk Kelsey, Tongxin Bai, Chen Ding and Chengliang Zhang. 'Fast Track: A Software System for Speculative Program Optimization'. In: *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '09. Washington, DC, USA: IEEE Computer Society, Mar. 2009, pp. 157–168. *ISBN*: 978-0-7695-3576-0. *DOI*: 10.1109/CGO.2009.18. *URL*: http://dx.doi.org/10.1109/CGO.2009.18.

[145] Khronos. *Open Computing Language (OpenCL)*. http://www.khronos.org/opencl/, Last visit: December 2, 2013. 2010.

[146] Hanjun Kim, Nick P Johnson, Jae W Lee, Scott A Mahlke and David I August. 'Automatic speculative DOALL for clusters'. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM. 2012, pp. 94–103.

[147] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee and David I. August. 'Scalable Speculative Parallelization on Commodity Clusters'. In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 3–14. *ISBN*: 978-0-7695-4299-7. *DOI*: 10.1109/MICRO.2010.19. *URL*: http://dx.doi.org/10.1109/MICRO.2010.19.

[148] Hanjun Kim, Arun Raman, Feng Liu, Jae W Lee and David I August. 'Scalable speculative parallelization on commodity clusters'. In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2010, pp. 3–14.

[149] Tom Knight. 'An architecture for mostly functional languages'. In: *Proceedings of the 1986 ACM conference on LISP and functional programming*. LFP '86. Cambridge, Massachusetts, USA: ACM, 1986, pp. 105–112. *ISBN*: 0-89791-200-4. *DOI*: 10.1145/319838.319854. *URL*: http://doi.acm.org/10.1145/319838.319854.

[150] Sai Charan Koduru, Min Feng and Rajiv Gupta. 'Programming large dynamic data structures on a DSM cluster of multicores'. In: *7th International Conference on PGAS Programming Models*. 2013, p. 126.

[151] V. Krishnan and J. Torrellas. 'A chip-multiprocessor architecture with speculative multithreading'. In: *IEEE Transactions on Computers*, vol. 48, no. 9, 1999, pp. 866–880. 1999. ISSN: 0018-9340. DOI: 10.1109/12.795218.

[152] Venkata Krishnan and Josep Torrellas. 'Executing Sequential Binaries on a Clustered Multithreaded Architecture with Speculation Support'. In: *Proceedings of the 1998 Fourth International Symposium on High-Performance Computer Architecture*. HPCA '98. Washington, DC, USA: IEEE Computer Society, 1998.

[153] V. P. Krothapalli and P. Sadayappan. 'An approach to synchronization for parallel computing'. In: *Proceedings of the 2nd international conference on Supercomputing*. ICS '88. St. Malo, France: ACM, 1988, pp. 573–581. ISBN: 0-89791-272-1. DOI: 10.1145/55364.55420. URL: http://doi.acm.org/10.1145/55364.55420.

[154] V. P. Krothapalli and P. Sadayappan. 'Dynamic scheduling of DOACROSS loops for multiprocessors'. In: *Databases, Parallel Architectures and Their Applications,. PARBASE-90, International Conference on*. Washington, DC, USA: IEEE Computer Society, 1990, pp. 66–75. DOI: 10.1109/PARBSE.1990.77118.

[155] C.P. Kruskal and A. Weiss. 'Allocating Independent Subtasks on Parallel Processors'. In: *Software Engineering, IEEE Transactions on*, vol. SE-11, no. 10, 1985, pp. 1001–1016. 1985. ISSN: 0098-5589.

[156] M. Kulkarni, M. Burtscher, C. Cascaval and K. Pingali. 'Lonestar: A suite of parallel irregular programs'. In: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. Washington, DC, USA: IEEE Computer Society, Apr. 2009, pp. 65–76. DOI: 10.1109/ISPASS.2009.4919639.

[157] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali and Calin Casçaval. 'How Much Parallelism is There in Irregular Applications?' In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '09. Raleigh, NC, USA: ACM, 2009, pp. 3–14. ISBN: 978-1-60558-397-6. DOI: 10.1145/1504176.1504181. URL: http://doi.acm.org/10.1145/1504176.1504181.

[158] Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala and L. Paul Chew. 'Scheduling Strategies for Optimistic Parallel Execution of Irregular Programs'. In: *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '08. Munich, Germany: ACM, 2008, pp. 217–228. ISBN: 978-1-59593-973-9. DOI: 10.1145/1378533.1378575. URL: http://doi.acm.org/10.1145/1378533.1378575.

[159] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui and Keshav Pingali. 'Exploiting the Commutativity Lattice'. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 542–555. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993562. URL: http://doi.acm.org/10.1145/1993498.1993562.

[160] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala and L. Paul Chew. 'Optimistic Parallelism Benefits from Data Partitioning'. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: ACM, 2008, pp. 233–243. *ISBN*: 978-1-59593-958-6. *DOI*: 10.1145/1346281.1346311. *URL*: http://doi.acm.org/10.1145/1346281.1346311.

[161] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala and L. Paul Chew. 'Optimistic Parallelism Requires Abstractions'. In: *Commun. ACM*, vol. 52, no. 9, Sept. 2009, pp. 89–97. New York, NY, USA: ACM, Sept. 2009. *ISSN*: 0001-0782. *DOI*: 10.1145/1562164.1562188. *URL*: http://doi.acm.org/10.1145/1562164.1562188.

[162] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala and L. Paul Chew. 'Optimistic parallelism requires abstractions'. In: *PLDI 2007 Proceedings*. New York, NY, USA: ACM, 2007, pp. 211–222.

[163] James Larus and Christos Kozyrakis. 'Transactional memory'. In: *Commun. ACM*, vol. 51, no. 7, July 2008, pp. 80–88. July 2008. *ISSN*: 0001-0782. *DOI*: 10.1145/1364782.1364800. (Visited on 03/03/2013).

[164] Chris Lattner and Vikram Adve. 'LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation'. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. *ISBN*: 0-7695-2102-9. *URL*: http://dl.acm.org/citation.cfm?id=977395.977673.

[165] Shun-Tak Leung and John Zahorjan. 'Improving the performance of runtime parallelization'. In: *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPOPP '93. San Diego, California, USA: ACM, 1993, pp. 83–91. *ISBN*: 0-89791-589-5. *DOI*: 10.1145/155332.155341. *URL*: http://doi.acm.org/10.1145/155332.155341.

[166] Peng Li and Song Guo. 'Energy Minimization on Thread-Level Speculation in Multicore Systems'. In: *Proceedings of the 2010 Ninth International Symposium on Parallel and Distributed Computing*. ISPDC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 125–132. *ISBN*: 978-0-7695-4120-4. *DOI*: 10.1109/ISPDC.2010.17. *URL*: http://dx.doi.org/10.1109/ISPDC.2010.17.

[167] Xiao-Feng Li, ZhaoHui Du, Chen Yang, Chu-Cheow Lim and Tin-Fook Ngai. 'Speculative Parallel Threading Architecture and Compilation'. In: *Proceedings of the 2005 International Conference on Parallel Processing Workshops*. ICPPW '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 285–294. *ISBN*: 0-7695-2381-1. *DOI*: 10.1109/ICPPW.2005.81. *URL*: http://dx.doi.org/10.1109/ICPPW.2005.81.

[168] Shaoshan Liu, Christine Eisenbeis and Jean-Luc Gaudiot. 'Speculative Execution on GPU: An Exploratory Study'. In: *Proceedings of the 2010 39th International Conference on Parallel Processing*. ICPP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 453–461. *ISBN*: 978-0-7695-4156-3. *DOI*: 10.1109/ICPP.2010.53. *URL*: http://dx.doi.org/10.1109/ICPP.2010.53.

[169] Xing Liu, Mikhail Smelyanskiy, Edmond Chow and Pradeep Dubey. 'Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors'. In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. Eugene, Oregon, USA: ACM, 2013, pp. 273–282. ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465013. URL: http://doi.acm.org/10.1145/2464996.2465013.

[170] Diego R. Llanos. 'Introducción a las técnicas de ejecución especulativa'. In: *Proceedings of speculative parallelization at running time (UVa)*. Oct. 2008.

[171] Diego R. Llanos. 'Un modelo software de ejecución especulativa'. In: *Proceedings of speculative parallelization at running time (UVa)*. Oct. 2008.

[172] Diego R. Llanos, David Orden and Bel? Palop. 'Just-In-Time Scheduling for Loop-based Speculative Parallelization'. In: *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 2008, pp. 334–342. Los Alamitos, CA, USA: IEEE Computer Society, 2008. ISSN: 1066-6192. DOI: http://doi.ieeecomputersociety.org/10.1109/PDP.2008.13.

[173] Diego R. Llanos, David Orden and Belén Palop. 'Meseta: A new scheduling strategy for speculative parallelization of randomized incremental algorithms.' In: *Proc. 2005 ICPP Workshops (HPSEC-05)*. Oslo, Norway, June 2005, pp. 121–128. ISBN: 0-7695-2381-1.

[174] Diego R. Llanos, David Orden and Belén Palop. 'New Scheduling Strategies for Randomized Incremental Algorithms in the Context of Speculative Parallelization'. In: *IEEE Transactions on Computers*, vol. 56, no. 6, 2007, pp. 839–852. Los Alamitos, CA, USA: IEEE Computer Society, 2007. ISSN: 0016-9340. DOI: http://doi.ieeecomputersociety.org/10.1109/TC.2007.1030.

[175] Steven Lucco. 'A dynamic scheduling method for irregular parallel programs'. In: *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. San Francisco, California, United States: ACM Press, 1992, pp. 200–211. ISBN: 0-89791-475-9. DOI: http://doi.acm.org/10.1145/143095.143134.

[176] Yangchun Luo, Wei-Chung Hsu and Antonia Zhai. 'The Design and Implementation of Heterogeneous Multicore Systems for Energy-efficient Speculative Thread Execution'. In: *ACM Trans. Archit. Code Optim.* Vol. 10, no. 4, Dec. 2013, 26:1–26:29. New York, NY, USA: ACM, Dec. 2013. ISSN: 1544-3566. DOI: 10.1145/2541228.2541233. URL: http://doi.acm.org/10.1145/2541228.2541233.

[177] Pedro Marcuello, Antonio Gonzalez and Jordi Tubella. 'Speculative multithreaded processors'. In: *Proceedings of the 12th international conference on Supercomputing*. ICS '98. Melbourne, Australia: ACM, 1998, pp. 77–84. ISBN: 0-89791-998-X. DOI: 10.1145/277830.277850. URL: http://doi.acm.org/10.1145/277830.277850.

[178] E.P. Markatos and T.J. LeBlanc. 'Using processor affinity in loop scheduling on shared-memory multiprocessors'. In: *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 4, Apr. 1994, pp. 379–400. Apr. 1994. ISSN: 1045-9219. DOI: 10.1109/71.273046.

[179] Jan Martinsen, Hakan Grahn and Anders Isberg. 'Using Speculation to Enhance JavaScript Performance in Web Applications'. In: *IEEE Internet Computing*, vol. 17, no. 2, Mar. 2013, pp. 10–19. Piscataway, NJ, USA: IEEE Educational Activities Department, Mar. 2013. ISSN: 1089-7801. DOI: 10.1109/MIC.2012.146. URL: http://dx.doi.org/10.1109/MIC.2012.146.

[180] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu and Scott Mahlke. 'Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory'. In: *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 166–176. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542495. URL: http://doi.acm.org/10.1145/1542476.1542495.

[181] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher and Keshav Pingali. 'Structure-driven Optimizations for Amorphous Data-parallel Programs'. In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '10. Bangalore, India: ACM, 2010, pp. 3–14. ISBN: 978-1-60558-877-3. DOI: 10.1145/1693453.1693457. URL: http://doi.acm.org/10.1145/1693453.1693457.

[182] Samuel P Midkiff and David A Padua. 'Compiler algorithms for synchronization'. In: *IEEE Transactions on Computers*, vol. 100, no. 12, 1987, pp. 1485–1495. IEEE, 1987.

[183] R. Mirchandaney and J. H. Saltz. *Dodynamic: A construct for on-the-fly parallelization of loops*. Tech. rep. 650. in preparation. 1988.

[184] J. H. Saltz and R. Mirchandaney. *How to schedule complex loops in parallel*. Tech. rep. 657. 1988.

[185] K. Moore, J. Bobba, M.J. Moravan, M.D. Hill and D.A. Wood. 'LogTM: log-based transactional memory'. In: *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*. Feb. 2006, pp. 254–265. DOI: 10.1109/HPCA.2006.1598134.

[186] E. P. Mücke, I. Saias and B. Zhu. 'Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations'. In: *Proceedings of the 12th ACM Symposium on Computational Geometry*. 1996, pp. 274–283.

[187] Diego Novillo. 'GCC an architectural overview, current status, and future directions'. In: *Proceedings of the Linux Symposium*. Tokyo, Japan, Sept. 2006, pp. 185–200.

[188] Diego Novillo. 'OpenMP and automatic parallelization in GCC'. In: *Proceedings of the 2006 GCC Developers' Summit*. Ottawa, Canada, 2006, pp. 135–144.

[189] NVIDIA. *NVIDIA CUDA Architecture Introduction and Overview Version 1.1*. 2009.

[190] Cosmin E. Oancea, Alan Mycroft and Tim Harris. 'A lightweight in-place implementation for software thread-level speculation'. In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. SPAA '09. Calgary, AB, Canada: ACM, 2009, pp. 223–232. ISBN: 978-1-60558-606-9. DOI: 10.1145/1583991.1584050. URL: http://doi.acm.org/10.1145/1583991.1584050.

[191] Stuart Olsen, Brian Romoser and Ziliang Zong. 'SQLPhi: A SQL-Based Database Engine for Intel Xeon Phi Coprocessors'. In: *Proceedings of the 2014 International Conference on Big Data Science and Computing*. BigDataScience '14. Beijing, China: ACM, 2014, 17:1–17:6. ISBN: 978-1-4503-2891-3. DOI: 10.1145/2640087.2644172. URL: http://doi.acm.org/10.1145/2640087.2644172.

[192] Kunle Olukotun, Lance Hammond and Mark Willey. 'Improving the performance of speculatively parallel applications on the Hydra CMP'. In: *Proceedings of the 13th international conference on Supercomputing*. ICS '99. Rhodes, Greece: ACM, 1999, pp. 21–30. ISBN: 1-58113-164-X. DOI: 10.1145/305138.305155. URL: http://doi.acm.org/10.1145/305138.305155.

[193] *OpenMP Specification, version 4.0*. http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf. [Last visit: June 2015].

[194] Jeffrey T. Oplinger, David L. Heine and Monica S. Lam. 'In Search of Speculative Thread-Level Parallelism'. In: *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*. PACT '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 303–. ISBN: 0-7695-0425-6. URL: http://dl.acm.org/citation.cfm?id=520793.825732.

[195] Jeffrey Oplinger, David Heine, Shih Liao, Basem A. Nayfeh, Monica S. Lam and Kunle Olukotun. *Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor*. Tech. rep. Stanford, CA, USA, 1997.

[196] Guilherme Ottoni and David August. 'Global Multi-Threaded Instruction Scheduling'. In: *MICRO 40 Proceedings*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 56–68. ISBN: 0-7695-3047-8.

[197] Guilherme Ottoni, Ram Rangan, Adam Stoler and David I August. 'Automatic thread extraction with decoupled software pipelining'. In: *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2005, pp. 105–118.

[198] V. Packirisamy, A. Zhai, Wei-Chung Hsu, Pen-Chung Yew and Tin-Fook Ngai. 'Exploring speculative parallelism in SPEC2006'. In: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. Washington, DC, USA: IEEE Computer Society, Apr. 2009, pp. 77–88. DOI: 10.1109/ISPASS.2009.4919640.

[199] Jongsoo Park, Ganesh Bikshandi, Karthikeyan Vaidyanathan, Ping Tak Peter Tang, Pradeep Dubey and Daehyun Kim. 'Tera-scale 1D FFT with Low-communication Algorithm and Intel&Reg; Xeon Phi&Trade; Coprocessors'. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM, 2013, 34:1–34:12. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503242. URL: http://doi.acm.org/10.1145/2503210.2503242.

[200] A. Phansalkar, A. Joshi, L. Eeckhout and L. K. John. 'Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites'. In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*. ISPASS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 10–20. ISBN: 0-7803-8965-4. DOI: 10.1109/ISPASS.2005.1430555. URL: http://dx.doi.org/10.1109/ISPASS.2005.1430555.

[201] Christopher J. F. Pickett. 'Software Speculative Multithreading for Java'. In: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007, pp. 929–930. ISBN: 978-1-59593-865-7. DOI: 10.1145/1297846.1297950. URL: http://doi.acm.org/10.1145/1297846.1297950.

[202] Christopher J. F. Pickett and Clark Verbrugge. 'SableSpMT: A Software Framework for Analysing Speculative Multithreading in Java'. In: *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '05. Lisbon, Portugal: ACM, 2005, pp. 59–66. ISBN: 1-59593-239-9. DOI: 10.1145/1108792.1108809. URL: http://doi.acm.org/10.1145/1108792.1108809.

[203] Christopher J. F. Pickett and Clark Verbrugge. 'Software Thread Level Speculation for the Java Language and Virtual Machine Environment'. In: *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*. LCPC'05. Hawthorne, NY: Springer-Verlag, 2006, pp. 304–318. ISBN: 3-540-69329-7, 978-3-540-69329-1. DOI: 10.1007/978-3-540-69330-7_21. URL: http://dx.doi.org/10.1007/978-3-540-69330-7_21.

[204] C. D. Polychronopoulos and D. J. Kuck. 'Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers'. In: *IEEE Transactions on Computers*, vol. C-36, no. 12, Dec. 1987, pp. 1425–1439. Dec. 1987.

[205] C. D. Polychronopoulos and D. J. Kuck. 'Guided self-scheduling: A practical scheduling scheme for parallel supercomputers'. In: *IEEE Trans. Comput.* Vol. 36, no. 12, Dec. 1987, pp. 1425–1439. Washington, DC, USA: IEEE Computer Society, Dec. 1987. ISSN: 0018-9340. DOI: 10.1109/TC.1987.5009495. URL: http://dx.doi.org/10.1109/TC.1987.5009495.

[206] Leo Porter, Bumyong Choi and Dean M. Tullsen. 'Mapping Out a Path from Hardware Transactional Memory to Speculative Multithreading'. In: *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. PACT '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 313–324. ISBN: 978-0-7695-3771-9. DOI: 10.1109/PACT.2009.37. URL: http://dx.doi.org/10.1109/PACT.2009.37.

[207] Manohar K. Prabhu and Kunle Olukotun. 'Exposing speculative thread parallelism in SPEC2000'. In: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '05. Chicago, IL, USA: ACM, 2005, pp. 142–152. ISBN: 1-59593-080-9. DOI: 10.1145/1065944.1065964. URL: http://doi.acm.org/10.1145/1065944.1065964.

[208] Manohar K. Prabhu and Kunle Olukotun. 'Using Thread-level Speculation to Simplify Manual Parallelization'. In: *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '03. San Diego, California, USA: ACM, 2003, pp. 1–12. ISBN: 1-58113-588-2. DOI: 10.1145/781498.781500. URL: http://doi.acm.org/10.1145/781498.781500.

[209] Prakash Prabhu, Ganesan Ramalingam and Kapil Vaswani. 'Safe programmable speculative parallelism'. In: *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '10. Toronto, Ontario, Canada: ACM, 2010, pp. 50–61. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806603. URL: http://doi.acm.org/10.1145/1806596.1806603.

[210] Dimitrios Prountzos, Roman Manevich, Keshav Pingali and Kathryn S. McKinley. 'A Shape Analysis for Optimizing Parallel Graph Programs'. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 159–172. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926405. URL: http://doi.acm.org/10.1145/1926385.1926405.

[211] Joan Puiggali, Boleslaw K Szymanski, Teo Jové and Jose L Marzo. 'Dynamic branch speculation in a speculative parallelization architecture for computer clusters'. In: *Concurrency and Computation: Practice and Experience*, vol. 25, no. 7, 2012. Wiley Online Library, 2012.

[212] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González and Dean M. Tullsen. 'Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices'. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 269–279. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065043. URL: http://doi.acm.org/10.1145/1065010.1065043.

[213] Ravi Rajwar and James R. Goodman. 'Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution'. In: *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 34. Austin, Texas: IEEE Computer Society, 2001, pp. 294–305. ISBN: 0-7695-1369-7. URL: http://dl.acm.org/citation.cfm?id=563998.564036.

[214] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin and David I. August. 'Speculative parallelization using software multi-threaded transactions'. In: *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. ASPLOS XV. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 65–76. ISBN: 978-1-60558-839-1. DOI: 10.1145/1736020.1736030. URL: http://doi.acm.org/10.1145/1736020.1736030.

[215] Easwaran Raman, Neil Vahharajani, Ram Rangan and David I. August. 'Spice: speculative parallel iteration chunk execution'. In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. CGO '08. Boston, MA, USA: ACM, 2008, pp. 175–184. ISBN: 978-1-59593-978-4. DOI: 10.1145/1356058.1356082. URL: http://0-doi.acm.org.almena.uva.es/10.1145/1356058.1356082.

[216] L. Rauchwerger and D. A. Padua. 'The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 2, 1999, pp. 160–180. 1999.

[217] Lawrence Rauchwerger. 'Speculative Parallelization of Loops'. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. USA: Springer US, 2011, pp. 1901–1912. ISBN: 978-0-387-09765-7. DOI: 10.1007/978-0-387-09766-4_35. URL: http://dx.doi.org/10.1007/978-0-387-09766-4_35.

[218] Lawrence Rauchwerger and David Padua. 'The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization'. In: *SIGPLAN Not.* Vol. 30, no. 6, June 1995, pp. 218–232. New York, NY, USA: ACM, June 1995. ISSN: 0362-1340. DOI: 10.1145/223428.207148. URL: http://doi.acm.org/10.1145/223428.207148.

[219] J. Renau, K. Strauss, L. Ceze, Wei Liu, S.R. Sarangi, J. Tuck and J. Torrellas. 'Energy-Efficient Thread-Level Speculation'. In: *IEEE Micro*, vol. 26, no. 1, 2006, pp. 80–91. 2006. ISSN: 0272-1732. DOI: 10.1109/MM.2006.11.

[220] Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti Sarangi, James Tuck and Josep Torrellas. 'Thread-Level Speculation on a CMP can be energy efficient'. In: *Proceedings of the 19th annual international conference on Supercomputing*. ICS '05. Cambridge, Massachusetts: ACM, 2005, pp. 219–228. ISBN: 1-59593-167-8. DOI: 10.1145/1088149.1088178. URL: http://doi.acm.org/10.1145/1088149.1088178.

[221] Arash Rezaei, Giuseppe Coviello, Cheng-Hong Li, Srimat Chakradhar and Frank Mueller. 'Snapify: Capturing Snapshots of Offload Applications on Xeon Phi Manycore Processors'. In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. HPDC '14. Vancouver, BC, Canada: ACM, 2014, pp. 1–12. ISBN: 978-1-4503-2749-7. DOI: 10.1145/2600212.2600215. URL: http://doi.acm.org/10.1145/2600212.2600215.

[222] Torvald Riegel. *Transactional Memory in GCC*. 2012. URL: https://gcc.gnu.org/wiki/TransactionalMemory.

[223] Anne Rogers, Martin C. Carlisle, John H. Reppy and Laurie J. Hendren. 'Supporting Dynamic Data Structures on Distributed-memory Machines'. In: *ACM Trans. Program. Lang. Syst.* Vol. 17, no. 2, Mar. 1995, pp. 233–263. New York, NY, USA: ACM, Mar. 1995. ISSN: 0164-0925. DOI: 10.1145/201059.201065. URL: http://doi.acm.org/10.1145/201059.201065.

[224] Eric Rotenberg, Steve Bennett and James E. Smith. 'Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching'. In: *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 29. Paris, France: IEEE Computer Society, 1996, pp. 24–35. ISBN: 0-8186-7641-8. URL: http://dl.acm.org/citation.cfm?id=243846.243854.

[225] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides and Jim Smith. 'Trace Processors'. In: *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 30. Research Triangle Park, North Carolina, USA: IEEE Computer Society, 1997, pp. 138–148. ISBN: 0-8186-7977-8. URL: http://dl.acm.org/citation.cfm?id=266800.266814.

[226] Amitabha Roy, Steven Hand and Tim Harris. 'A Runtime System for Software Lock Elision'. In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 261–274. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519094. URL: http://doi.acm.org/10.1145/1519065.1519094.

[227] Peter Rundberg and Per Stenström. 'Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors'. In: *Workshop on Scalable Shared Memory Multiprocessors*. June 2000.

[228] Joel H Saltz, Ravi Mirchandaney and Kay Crowley. 'Run-time parallelization and scheduling of loops'. In: *IEEE Transactions on Computers*, vol. 40, no. 5, 1991, pp. 603–612. IEEE, 1991.

[229] Mehrzad Samadi, Amir Hormati, Janghaeng Lee and Scott Mahlke. 'Paragon: Collaborative Speculative Loop Execution on GPU and CPU'. In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. GPGPU-5. London, United Kingdom: ACM, 2012, pp. 64–73. ISBN: 978-1-4503-1233-2. DOI: 10.1145/2159430.2159438. URL: http://doi.acm.org/10.1145/2159430.2159438.

[230] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar and Pradeep Dubey. 'Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?' In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12. Portland, Oregon: IEEE Computer Society, 2012, pp. 440–451. ISBN: 978-1-4503-1642-2. URL: http://dl.acm.org/citation.cfm?id=2337159.2337210.

[231] Dirk Schmidl, Tim Cramer, Sandra Wienke, Christian Terboven and MatthiasS. Müller. 'Assessing the Performance of OpenMP Programs on the Intel Xeon Phi'. English. In: *Euro-Par 2013 Parallel Processing*. Ed. by Felix Wolf, Bernd Mohr and Dieter an Mey. Vol. 8097. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 547–558. *ISBN*: 978-3-642-40046-9. *DOI*: 10.1007/978-3-642-40047-6_56. *URL*: http://dx.doi.org/10.1007/978-3-642-40047-6_56.

[232] Nir Shavit and Dan Touitou. 'Software Transactional Memory'. In: *Distributed Computing*, vol. 10, 1997, pp. 99–116. 1997.

[233] Gurindar S. Sohi, Scott E. Breach and T. N. Vijaykumar. 'Multiscalar processors'. In: *Proceedings of the 22nd annual international symposium on Computer architecture*. ISCA '95. S. Margherita Ligure, Italy: ACM, 1995, pp. 414–425. *ISBN*: 0-89791-698-0. *DOI*: 10.1145/223982.224451. *URL*: http://doi.acm.org/10.1145/223982.224451.

[234] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai and Todd C. Mowry. 'A scalable approach to thread-level speculation'. In: *Proceedings of the 27th annual international symposium on Computer architecture*. ISCA '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 1–12. *ISBN*: 1-58113-232-8. *DOI*: 10.1145/339647.339650. *URL*: http://doi.acm.org/10.1145/339647.339650.

[235] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai and Todd C. Mowry. 'Improving Value Communication for Thread-Level Speculation'. In: *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. HPCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 65–. *URL*: http://dl.acm.org/citation.cfm?id=874076.876480.

[236] J. Steffan and T Mowry. 'The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization'. In: *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. HPCA '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 2–. *ISBN*: 0-8186-8323-6. *URL*: http://dl.acm.org/citation.cfm?id=822079.822712.

[237] Peiyi Tang and Pen-Chung Yew. 'Processor Self-Scheduling for Multiple Nested Parallel Loops'. In: *IEEE Intl. Conf. on Parallel Processing*. Aug. 1986, pp. 528–535.

[238] Peiyi Tang and Pen-Chung Yew. 'Processor Self-Scheduling for Multiple-Nested Parallel Loops.' In: *ICPP*. Vol. 86. USA: CRC Press, 1986, pp. 528–535.

[239] Chen Tian, Min Feng and Rajiv Gupta. 'Speculative Parallelization Using State Separation and Multiple Value Prediction'. In: *Proceedings of the 2010 International Symposium on Memory Management*. ISMM '10. Toronto, Ontario, Canada: ACM, 2010, pp. 63–72. *ISBN*: 978-1-4503-0054-4. *DOI*: 10.1145/1806651.1806663. *URL*: http://doi.acm.org/10.1145/1806651.1806663.

[240] Chen Tian, Min Feng and Rajiv Gupta. 'Supporting speculative parallelization in the presence of dynamic data structures'. In: *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '10. Toronto, Ontario, Canada: ACM, 2010. *ISBN*: 978-1-4503-0019-3. *DOI*: 10.1145/1806596.1806604.

[241] Chen Tian, Min Feng, Vijay Nagarajan and Rajiv Gupta. 'Copy or Discard execution model for speculative parallelization on multicores'. In: *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 330–341. ISBN: 978-1-4244-2836-6. DOI: 10.1109/MICRO.2008.4771802. URL: http://dx.doi.org/10.1109/MICRO.2008.4771802.

[242] Chen Tian, Min Feng, Vijay Nagarajan and Rajiv Gupta. 'Speculative Parallelization of Sequential Loops on Multicores'. In: *Int. J. Parallel Program.* Vol. 37, no. 5, Oct. 2009, pp. 508–535. Norwell, MA, USA: Kluwer Academic Publishers, Oct. 2009. ISSN: 0885-7458. DOI: 10.1007/s10766-009-0111-z. URL: http://dx.doi.org/10.1007/s10766-009-0111-z.

[243] Chen Tian, Changhui Lin, Min Feng and Rajiv Gupta. 'Enhanced speculative parallelization via incremental recovery'. In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. PPoPP '11. San Antonio, TX, USA: ACM, 2011, pp. 189–200. ISBN: 978-1-4503-0119-0. DOI: 10.1145/1941553.1941580. URL: http://doi.acm.org/10.1145/1941553.1941580.

[244] Josep Torrellas. 'Speculation, Thread-Level'. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. USA: Springer US, 2011, pp. 1894–1900. ISBN: 978-0-387-09765-7. DOI: 10.1007/978-0-387-09766-4_170. URL: http://dx.doi.org/10.1007/978-0-387-09766-4_170.

[245] Georgios Tournavitis, Zheng Wang, Björn Franke and Michael FP O'Boyle. 'Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping'. In: *ACM Sigplan Notices*. Vol. 44. 6. ACM. 2009, pp. 177–187.

[246] Jordi Tubella and Antonio Gonzalez. 'Control speculation in multithreaded processors through dynamic loop detection'. In: *Proceedings of the 1998 Fourth International Symposium on High-Performance Computer Architecture*. HPCA '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 14–23. DOI: 10.1109/HPCA.1998.650542.

[247] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo and Rebecca L. Stamm. 'Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor'. In: *Proceedings of the 23rd Annual International Symposium on Computer Architecture*. ISCA '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 191–202. ISBN: 0-89791-786-3. DOI: 10.1145/232973.232993. URL: http://doi.acm.org/10.1145/232973.232993.

[248] Dean M. Tullsen, Susan J. Eggers and Henry M. Levy. 'Simultaneous Multithreading: Maximizing On-chip Parallelism'. In: *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. ISCA '98. Barcelona, Spain: ACM, 1998, pp. 533–544. ISBN: 1-58113-058-9. DOI: 10.1145/285930.286011. URL: http://doi.acm.org/10.1145/285930.286011.

[249] Ten H. Tzen and Lionel M. Ni. 'Trapezoid Self-Scheduling: A Pratical Scheduling Scheme for Parallel Compilers'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, 1993, pp. 87–98. 1993.

[250] Neil Amar Vachharajani. 'Intelligent speculation for pipelined multithreading'. AAI3338698. PhD thesis. Princeton, NJ, USA: Princeton University, 2008. ISBN: 978-0-549-93358-8.

[251] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni and David I. August. 'Speculative Decoupled Software Pipelining'. In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. PACT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 49–59. *ISBN*: 0-7695-2944-5. *DOI*: 10.1109/PACT.2007.66. *URL*: http://dx.doi.org/10.1109/PACT.2007.66.

[252] David W. Walker. 'The design of a standard message passing interface for distributed memory concurrent computers'. In: *Parallel Comput.* Vol. 20, no. 4, 1994, pp. 657–673. 1994. *URL*: http://portal.acm.org/citation.cfm?id=180103.

[253] Steven Wallace, Brad Calder and Dean M. Tullsen. 'Threaded Multiple Path Execution'. In: *Proceedings of the 25th Annual International Symposium on Computer Architecture*. ISCA '98. Barcelona, Spain: IEEE Computer Society, 1998, pp. 238–249. *ISBN*: 0-8186-8491-7. *DOI*: 10.1145/279358.279392. *URL*: http://dx.doi.org/10.1145/279358.279392.

[254] Yizhuo Wang, A. Nicolau, R. Cammarota and A.V. Veidenbaum. 'A fault tolerant self-scheduling scheme for parallel loops on shared memory systems'. In: *High Performance Computing (HiPC), 2012 19th International Conference on*. Dec. 2012, pp. 1–10. *DOI*: 10.1109/HiPC.2012.6507476.

[255] F. Warg and P. Stenstrom. 'Dual-Thread Speculation: Two Threads in the Machine are Worth Eight in the Bush'. In: *Computer Architecture and High Performance Computing, 2006. SBAC-PAD '06. 18TH International Symposium on*. Oct. 2006, pp. 91–98. *DOI*: 10.1109/SBAC-PAD.2006.17.

[256] Fredrik Warg and Per Stenström. 'Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction'. In: *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IPDPS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 12.2–. *ISBN*: 0-7695-1926-1. *URL*: http://dl.acm.org/citation.cfm?id=838237.838521.

[257] Fredrik Warg and Per Stenström. 'Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms'. In: *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*. PACT '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 221–230. *ISBN*: 0-7695-1363-8. *URL*: http://dl.acm.org/citation.cfm?id=645988.674160.

[258] Fredrik Warg and Per Stenström. 'Reducing misspeculation overhead for module-level speculative execution'. In: *Proceedings of the 2nd conference on Computing frontiers*. CF '05. Ischia, Italy: ACM, 2005, pp. 289–298. *ISBN*: 1-59593-019-1. *DOI*: 10.1145/1062261.1062310. *URL*: http://doi.acm.org/10.1145/1062261.1062310.

[259] Emo Welzl. 'Smallest enclosing disks (balls and ellipsoids)'. In: *New results and new trends in computer science*. Vol. 555. Lecture notes in computer science. Springer-Verlag, 1991, pp. 359–370.

[260] Michael E Wolf and Monica S Lam. 'A loop transformation theory and an algorithm to maximize parallelism'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, 1991, pp. 452–471. IEEE, 1991.

[261] Michael Wong, Barna L. Bihari, Bronis R. de Supinski, Peng Wu, Maged Michael, Yan Liu and Wang Chen. 'A case for including transactions in OpenMP'. In: *IWOMP'10 Proceedings*. 2010, pp. 149–160. *ISBN*: 3-642-13216-2, 978-3-642-13216-2. *DOI*: 10.1007/978-3-642-13217-9_12. (Visited on 02/03/2013).

[262] Peng Wu, Arun Kejariwal and Călin Caşcaval. 'Compiler-Driven Dependence Profiling to Guide Program Parallelization'. In: *Languages and Compilers for Parallel Computing*. Ed. by José Nelson Amaral. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 232–248. *ISBN*: 978-3-540-89739-2. *DOI*: 10.1007/978-3-540-89740-8_16. *URL*: http://dx.doi.org/10.1007/978-3-540-89740-8_16.

[263] P. Xekalakis and M. Cintra. 'Handling branches in TLS systems with Multi-Path Execution'. In: *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. Washington, DC, USA: IEEE Computer Society, Jan. 2010, pp. 1–12. *DOI*: 10.1109/HPCA.2010.5416632.

[264] Polychronis Xekalakis, Nikolas Ioannou and Marcelo Cintra. 'Combining Thread Level Speculation, Helper Threads and Runahead Execution'. In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS '09. Yorktown Heights, NY, USA: ACM, 2009, pp. 410–420. *ISBN*: 978-1-60558-498-0. *DOI*: 10.1145/1542275.1542333. *URL*: http://doi.acm.org/10.1145/1542275.1542333.

[265] Polychronis Xekalakis, Nikolas Ioannou and Marcelo Cintra. 'Mixed Speculative Multithreaded Execution Models'. In: *ACM Trans. Archit. Code Optim.* Vol. 9, no. 3, Oct. 2012, 18:1–18:26. New York, NY, USA: ACM, Oct. 2012. *ISSN*: 1544-3566. *DOI*: 10.1145/2355585.2355591. *URL*: http://doi.acm.org/10.1145/2355585.2355591.

[266] Polychronis Xekalakis, Nikolas Ioannou, Salman Khan and Marcelo Cintra. 'Profitability-based power allocation for speculative multithreaded systems'. In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–11.

[267] Paraskevas Yiapanis, Demian Rosas-Ham, Gavin Brown and Mikel Luján. 'Optimizing software runtime systems for speculative parallelization'. In: *ACM Trans. Archit. Code Optim.* Vol. 9, no. 4, Jan. 2013, 39:1–39:27. New York, NY, USA: ACM, Jan. 2013. *ISSN*: 1544-3566. *DOI*: 10.1145/2400682.2400698. *URL*: http://doi.acm.org/10.1145/2400682.2400698.

[268] Antonia Zhai, J. Gregory Steffan, Christopher B. Colohan and Todd C. Mowry. 'Compiler and Hardware Support for Reducing the Synchronization of Speculative Threads'. In: *ACM Trans. Archit. Code Optim.* Vol. 5, no. 1, May 2008, 3:1–3:33. New York, NY, USA: ACM, May 2008. *ISSN*: 1544-3566. *DOI*: 10.1145/1369396.1369399.

[269] Chao Zhang, Chen Ding, Xiaoming Gu, Kirk Kelsey, Tongxin Bai and Xiaobing Feng. 'Continuous speculative program parallelization in software'. In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '10. Bangalore, India: ACM, 2010, pp. 335–336. *ISBN*: 978-1-60558-877-3. *DOI*: 10.1145/1693453.1693501. *URL*: http://doi.acm.org/10.1145/1693453.1693501.

[270] Chenggang Zhang, Guodong Han and Cho-Li Wang. 'GPU-TLS: An Efficient Runtime for Speculative Loop Parallelization on GPUs'. In: *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 120–127. *DOI*: 10.1109/CCGrid.2013.34.

[271] Zhijia Zhao and Xipeng Shen. 'On-the-Fly Principled Speculation for FSM Parallelization'. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2015, pp. 619–630.

[272] Zhijia Zhao, Bo Wu and Xipeng Shen. 'Challenging the embarrassingly sequential: parallelizing finite state machine-based computations through principled speculation'. In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 1. ACM. 2014, pp. 543–558.

[273] Zhijia Zhao, Bo Wu and Xipeng Shen. 'Speculative parallelization needs rigor: probabilistic analysis for optimal speculation of finite-state machine applications'. In: *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. PACT '12. Minneapolis, Minnesota, USA: ACM, 2012, pp. 433–434. *ISBN*: 978-1-4503-1182-3. *DOI*: 10.1145/2370816.2370882. *URL*: http://doi.acm.org/10.1145/2370816.2370882.

[274] Chuan-Qi Zhu and Pen-Chung Yew. 'A scheme to enforce data dependence on large multiprocessor systems'. In: *IEEE Transactions on Software Engineering*, vol. SE-13, no. 6, June 1987, pp. 726–739. IEEE, June 1987. *ISSN*: 0098-5589. *DOI*: 10.1109/TSE.1987.233477.

[275] Craig Zilles and Gurindar Sohi. 'Master/slave speculative parallelization'. In: *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. MICRO 35. Istanbul, Turkey: IEEE Computer Society Press, 2002, pp. 85–96. *ISBN*: 0-7695-1859-1. *URL*: http://dl.acm.org/citation.cfm?id=774861.774871.