Fernando Brunetti

Guillermo Benítez

TAI 2

Indice

Speculative Multithreaded Processors

El objetivo de este trabajo es presentar una de la tendencias del diseño de microprocesadores. El modelo presentado en este trabajo todavía se encuentra en etapa de investigación pero ya están siendo adoptados algunos conceptos en las arquitecturas comerciales de última generación y en procesadores superescalares.

De acuerdo a los objetivos de la Materia este trabajo valoriza los campos de investigación y futuras aplicaciones. Se adjuntan artículos de implementaciones basadas en estos nuevos diseños.

Speculative Multithreaded Processors

La tecnología en semiconductores sumada a as innovaciones de las arquitecturas de computadoras, han proveído las herramientas necesarias para generar un avance fenomenal en los procesadores durante la década pasada, culminando en los últimos años con cientos de millones de transistores utilizados para crear dispositivos on-chip más veloces. Las innovaciones hechas el micro

arquitectura de procesadores y sus respectivos compiladores nos han permitido optimizar el uso de estos nuevos recursos para lograr un mayor desempeño de los sistemas.

Típicamente, nosotros decidimos como usar los recursos de la tecnología en semiconductores en dos pasos. Primero, elegimos la funcionalidad deseada y utilizamos las técnicas de diseño para lograr dicho propósito. En la fase de implementación traducimos esas técnicas en estructuras, señales que de ver ser hechas y verificadas. Aunque a veces describimos estas fases por separado en la práctica están muy fuertemente unidas.

Durante los años '90, la nueva funcionalidad jugó un papel preponderante en el diseño de procesadores. Dado un razonable límite en el tamaño total - por ejemplo menos que 10 millones de transistores – podríamos optimizar nuestro presupuesto de transistores usando métricas de muy alto rendimiento. Haciendo esto la verificación relativamente más sencilla, y los diseños no tienen que explicitar cantidades para el retardo en los cables, que son insignificantes al lado de los retardo introducidos por compuertas lógicas.

En el futuro, el asunto de la implementación va dominar básicamente cada funcionalidad. Se está comprobando que usar técnicas escalares convencionales en el diseño de procesadores superescalares incrementan la complejidad en el diseño y su elevado costo tampoco es una garantía de que tal diseño obtendrá buenos resultados. Diseños con tecnología monolítica que utilizan cientos de millones de transistores serán muy difíciles de diseñar, optimizar y verificar más los retardos de los buses hará de la comunicación intrachip y el clock muy costosos. Por consiguiente, algunas arquitecturas de computadoras se avocan a pasar del alto rendimiento alto flujo de datos (throughput) procesados., utilizando componentes distribuidos que dividan la complejidad del diseño y utilicen las ventajas de la intercomunicación local para evitar los retardos de los buses.

Con esta tendencia surge el interés en las arquitecturas Multihilos. Tales arquitecturas pueden sacar paralelismo de programas secuenciales utilizando algoritmos especulativos multihilos de tercer nivel – sean ellos mediante el control o por datos- dándoles flexibilidad para operar en ambos múltiples programas, mayor flujo de datos procesados para un simple programa y por lo tanto un alto rendimiento.

1. Razones para los Multihilos especulativos.

Afortunadamente, Las ventajas de incrementar el rendimiento de un programa único y decrementar la dificultad en la implementación no entran en conflicto necesariamente. La motivación par utilizar Multihilos especulativos viene de dos partes: somos testigos del potencial disminuido de la técnicas corrientes para extraer paralelismo de un programa secuencial único. Y por otra parte la tendencia en la tecnología sugiere la necesidad de procesadores capaces de manejar múltiples hilos de programas independientes. De esta manera estamos casi obligados a encontrar innovaciones que permitirán a los procesadores Multihilo soportar la ejecución paralela de un programa único.

Un procesador multihilo especulativo consiste en una repetición de elementos lógicos que cooperadamente ejecutan paralelamente un programa único secuencial convencional. – también llamado hilo del programa – dividido en partes llamados hilos especulativos. Especulación es la clave: sin especulación solo podemos dividir programas conservadoramente en hilos no especulativos los cuales garantiza la mutua independencia de los hilos entre sí. La especulación permite divisiones mucho más agresivas que explotan a toda su capacidad los hilos sin garantizar que la ejecución sea independiente pero sí paralela, y con mucha probabilidades de ser ejecutadas.

1.1 Extracción de paralelismo. Método común y sus limitaciones.

Hoy, los modelos superescalares mantienen la posición inalterable de alcanzar un rendimiento alto en la ejecución de un programa único. Programas imperativos –escritos en lenguaje como

Fortran, C, and Java- están definidos por un control de flujo estático. Como la figura 1 muestra, en tiempo de ejecución, el procesador realiza el control estático de flujo para producir una flujo dinámico de instrucciones. Un procesador superescalar opera en el grupo de instrucciones de las primeras instrucciones de la cola de algoritmo de despacho. El procesador revisa repetidamente esta ventana de l cola dinámica de instrucciones todavía no ejecutadas, las instrucciones independientes son ejecutadas en paralelo. Para lograr alto rendimiento cada ventana debe contener bastantes instrucciones independientes para aprovechar el paralelismo basado en este método (instruccion-level parallelism eficaz (ILP))

Desgraciadamente, las convenciones de la programación imperativa no utilizan alto ILP constantemente. Los programadores tienden a agrupar declaraciones dependientes en sus estructuras estáticas –logrando con ésto códigos más sencillo pero desaprovechando la capacidad de paralelismo proveídas por el procesador limitando el trabajo independiente disponible en cualquier ventana dinámica de instrucciones. Los compiladores modernos tratan de mejorar el uso de las ventanas ILP reordenando las instrucciones transparentemente a fin de conseguir instrucciones independientes de zonas cercanas del programa, pero incluso el más sofisticado de los compiladores está fundamentalmente limitado por la incapacidad de determinar la intención del programador y su necesidad de conservar la estructura de alto nivel del programa y su respectiva semántica.

Dada la cantidad de trabajo paralelo que se hace, nosotros podríamos construir un procesador superescalar con una ventana de instrucciones bastante grande como para contener código de distintas partes de programa simultáneamente, funciones diferentes o iteraciones. Sin embargo, por encima de los muchos obstáculos de la ingeniería, manteniendo una ventana grande, inmediata llena de instrucciones útiles propone un problema fundamental. Específicamente, la exactitud decreciente de una serie de saltos condicionales que se predicen conducen a una probabilidad que decrece exponencialmente de que las instrucciones en la cola de la ventana sean útiles.

Para superar este problema se requiere un modelo que permita paralelismo de regiones diferentes del programa suponiendo su independencia para explotar en un término razonablemente –esto implica instrucciones no continuas ni relacionadad en forma serial-. El modelo del "multithreading" especulativo considera que cada región del programa para es un hilo especulativo o un programa pequeño. Ejecutando hilos especulativos múltiples en grados paralelos, altos de concurrencia puede lograrse, sobre todo si cada hilo es principalmente secuencial. El modelo une los hilos seguidamente para recrear el programa original. El multithreading especulativo nos permite formar una ventana de instrucciones grande, fuera de un conjunto de ventanas más pequeñas de instrucciones , facilitando la implementación. Además, la división del hilo apropiada puede aislar saltos en un hilo de aquellos en otros[2], resolviendo el problema fundamental de la utilidad de la instrucción ejecutada.

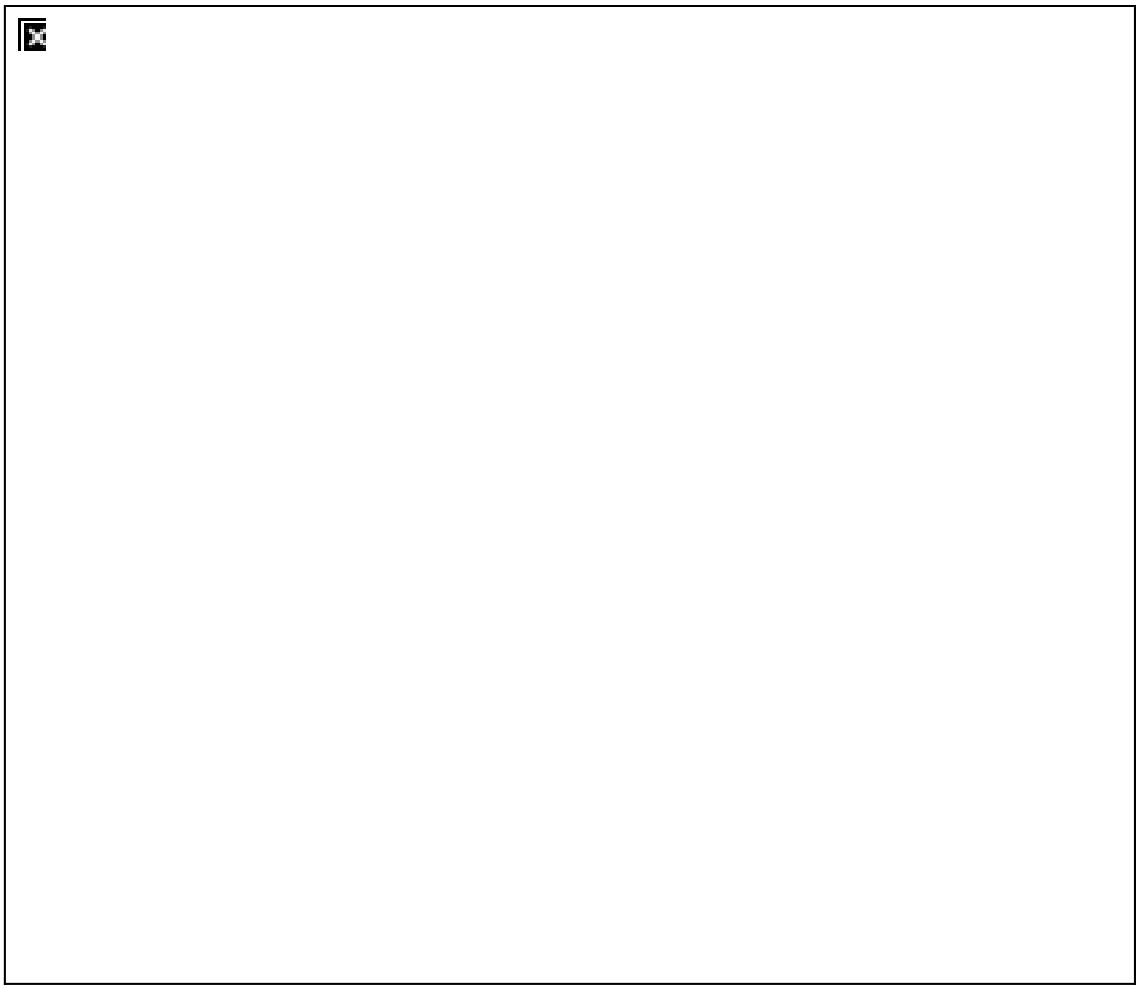

Figura 1. Un flujo dinámico de instrucciones. Un procesador superescalar opera gradualmente en el grupo de instrucciones al frente de la cola dinámica de instrucciones a medida que el procesador sirve a la cola.

1.2 Arquitecturas de Multithreaded

Procesadores de Multithreaded que apoyan ejecución concurrente de hilos múltiples en un "single chip" dominarán la próxima década; existiendo dos modelos que se exploran actualmente. Multithreading simultáneo (Simultaneous Multithreading, SMT) usa un diseño monolítico con la mayoría de los recursos compartidos entre los hilos[1,3] "Chip multiprocessing" (CMP) propone un diseño distribuido que usa una colección de elementos del proceso independientes con menos recursos compartidos[4].

SMT se esfuerza por proporcionar soporte multithreading económico por encima de los procesadores superescalares ILP convencionales, en tanto que CMP ofrece simplicidad en el diseño y argumentos similares. Ambos modelos apuntan a los hilos independientes y usan multithreading para mejorar el "throughput". Si estos hilos se derivan de un solo programa, el aumento del throughput también se ve en la ejecución de un programa único . El "Piranha: single chip explota su característica Multiprocessing"(ver Apéndice A) el artículo muestra una aplicación de CMP, mientras el artículo "Cray MTA: Multithreading para Tolerancia de Latencia" (ver Apéndice B) describe una aplicación de procesador de multithreaded.

Como adelanto de la tecnología, la distinción entre el SMT y la micro arquitectura del CMP desaparecerá probablemente. Sin tener en cuenta la aplicación específica, los procesadores del multithreaded parecerán lógicamente ser colecciones de elementos de proceso. Si podemos aprovechar esta organización para mejorar el throughput y " tiempo de la ejecución de un solo programa permanece incierto. Con especulación a nivel de hilo, los procesadores lógicos pueden ejecutar hilos paralelos convencionales así como los solos programas divididos en hilos especulativos.

2. Dividiendo los programas en múltiple hilos

Hay varias maneras de dividir programas en hilos. Nosotros categorizamos estas divisiones como manejado por control o manejado por los datos dependiendo adelante si los hilos se dividen principalmente a lo largo de flujo de control o por los límites del flujo de datos.. Nosotros podemos llevar más allá subcategorizando estas divisiones como no especulativo o especulativo. Desde el punto de vista del procesador, los hilos no especulativos son completamente independientes, siendo cualquier dependencia explícitamente forzada utilizando estructuras de la sincronización basadas en la arquitectura. Los hilos especulativos, por otro lado, no necesitan ser absolutamente independiente o sincronizado, con diseñadores que dejan al hardware descubrir y recuperarse de las violaciones de las asunciones de independencia.

Se esperan que los hilos obtenidos de la división del programa sean ejecutados en unidades del proceso lógicas diferentes. Para lograr la concurrencia, hilos próximos –hilos que coexistirán simultáneamente en el procesador- deben ser muy independientes en cuanto a los datos. Si nosotros logramos tal independencia de los datos, la concurrencia, y por lo tanto rendimiento puede mejorar casi linealmente con el número de hilos inclusive para los tamaños de ventana pequeños por hilo de ejecución.. La eficacia permanece constante como el ancho de banda y, por consiguiente, aumenta el rendimiento. Nosotros creemos que la especulación puede permitir lograr el criterio de la independencia de datos más fácilmente y las soluciones especulativas posee ventajas a diferencia de las no especulativas.

2.1 Hilos manejados por control

Multithreading busca dividir los programas en los hilos paralelos de datos independientes. Para los programas imperativos, la división más natural ocurre a lo largo de los límites de control de flujo debido a los mismos límites en sí. En la semántica de la arquitectura de hilos manejada por el control : Las instrucciones son totalmente ordenadas, y el estado de la arquitectura se define precisamente sólo en los límites de la instrucción, con lo cual se logra un control de flujo explícito y un flujo de datos implícito. El multithreading manejado por control divide la cola dinámica de instrucciones en segmentos inmediatos que el sistema puede seguidamente "coser" de extremo-a-extremo para reconstruir la ejecución secuencial. El multithreading manejado por control presenta el desafío hallazgo de los puntos de división del programa que minimizan la dependencias de datos entre los hilos.

El multithreading manejado por control difiere de la programación paralela. Considerando que los programas paralelos ejecutan múltiples hilos coexistentes manejados por control, estos hilos raramente intercambian datos de manera arbitraria y su semántica raramente coincide con la semántica de los hilos individuales corrida en series. En contraste, el multithreading manejado por control nos permite imponer ejecución paralela en lo que es esencialmente un programa secuencial. Los datos fluyen secuencialmente entre los hilos manejados por control en una sola dirección de los hilos más viejos a los más jóvenes.

2.1.1 Hilos no especulativos manejados por control.

Sin soporte para descubrir y recuperarse de las violaciones de dependencia de datos o para abortar hilos innecesarios y desechar sus efectos, el Multithreading no especulativo manejado por control requiere garantías estrictas sobre la certeza de la ejecución de hilo y la integridad de los datos. Porque la ejecución del hilo no puede deshacerse, los requisitos de la certeza de la ejecución dicta que solo podemos tratar el hilo no especulativo manejado por control si sabemos que su ejecución realmente se necesita. Para aumentar al máximo la concurrencia, normalmente se logra certeza en la ejecución bifurcando un hilo en un punto previo de control equivalente -por ejemplo bifurcando un loop al inicio de la iteración previa-.

La integridad de los datos requiere que el acceso a los datos compartidos por los hilos ocurra o parezca ocurrir en orden secuencial. Cuando hablamos de integridad de los datos, nos referimos principalmente a integridad de memoria. El soporte para la comunicación directa entre los registros de los hilos generalmente no está disponible, pero, cuando lo está, asumimos que la sincronización apropiada se proporciona. En contraste, la comunicación de memoria de interthread está naturalmente disponible lo que implica que el acceso a cualquier sitio de memoria potencialmente compartido con otros hilos debe sincronizarse explícitamente. El compartimiento de los datos y la sincronización deben guardarse sobre un mínimo para permitir concurrencia adecuada entre los hilos.

Con tales requisitos estrictos de seguridad s, dividir un programa en hilos no especulativos tradicionalmente han sido obviados por el programador y compilador. El programador tiene el conocimiento más profundo de las dimensiones del paralelismo del algoritmo, así como el potencial de la capacidad de diferentes divisiones de compartir los datos. Sin embargo, lo tedioso de la división del hilo en forma manual lleva a menudo a los errores. Debido a que los compiladores de alta calidad no cometen errores, los informáticos se han esforzado considerablemente para tener compiladores que automáticamente consideren el multithreading en el programa y el paralelismo, pero sólo ha tenido éxito en campos muy limitados.

2..1.2 Hilos especulativos manejados por control.

Multithreading no especulativo manejado por control padece dos grandes problemas. Primero, los requisitos de certeza de la ejecución limitan la división en hilos a puntos del programa de control independientes que no pueden satisfacer el criterio primario de la independencia de datos. Segundo, incluso cuando los hilos próximos son independientes en cuanto a sus datos, si no podemos demostrar esta independencia, la sincronización conservadora debe usarse para salvaguardar contra el improbable pero remotamente posible caso de accesos concurrentes a mismo sitio del dato. Dondequiera que diseñadores aplican sincronización inútilmente, ellos pierden concurrencia y rendimiento innecesariamente.

La especulación puede aliviar estos problemas. En el multithreading especulativo, manejado por control, la memoria no debe ser necesariamente sincronizada en absoluto. El orden total correcto de las operaciones de memoria puede reconstruirse del orden explícito o implícito de los hilos. Esta clasificación puede usarse como la base para apoyo del hardware para descubrir y potencialmente recuperarse del interthread y de las violaciones del orden de la memoria[5,6] . Con tal soporte, acceder a los datos compartidos por los hilos puede procederse con optimismo, con penalizaciones incurridas sólo en esos casos donde los hilos próximos realmente comparten datos y los accesos ocurren en orden no secuencial. Yendo todavía más, desde que los patrones de las violaciones son típicamente predecibles, ligeras modificaciones al mecanismo básico le permiten aprender a reconocer estos patrones tempranamente y artificialmente sincronizar los pares ofensivos de load-store[6] .

El problema de la certeza en la ejecución puede alivianarse usando mecanismos similares. Podemos recuperar del interthread las violaciones en cuanto al orden de memoria usando hardware

que almacene o deshaga los cambios al estado del hilo. Tal soporte del hardware permite iniciar los hilos en puntos donde su utilidad final no puede garantizarse en forma absoluta, pero donde ellos muestran una alta probabilidad a de utilidad y las características de la independencia de los datos son más favorables.

El multithreading especulativo manejado por control ha sido un tema de investigación académica desde los años noventa y ha lentamente va encontrando su camino en los productos comerciales. El Sun's MAJC architecture[7] soporta tales hilos vía su modelo de computación de tiempo espacial. Un chip prototipo de NEC –Merlot- usa el multithreading especulativo manejado por control para paralelizar la ejecución de código que no puede ser paralelizado a través de los otros medios conocidos[8]. Esperamos que más procesadores hicieran uso del Speculative control-driven threads en la próxima década, cuando esta tecnología pase de la fase de la investigación a las aplicaciones comerciales.

2.2 Hilos manejados por datos

Mientras que el multithreading manejado por control divide los programas delante de las demarcaciones del control de flujo, el multithreading manejado por datos utiliza límites del flujo de datos como el criterio de la división mayor. Semejante división logra la deseada independencia de los datos interthread y el paralelismo resulta naturalmente[9]. Hilos manejados por datos proporcionan un rendimiento casi ideal y optimiza la eficacia. En su forma pura, el multithreading manejado por datos ocurre al desmenuzar una sola instrucción[9]. La obtención de la instrucción es activada por la disponibilidad de uno de su operandos de entrada. Las instrucciones entran en la máquina en cuanto ellos puedan ejecutarse, pero no más pronto. Este arreglo aumenta al máximo la cantidad de trabajo que puede solapar instrucciones de latencia largas, mientras no desperdician recursos en instrucciones que todavía no puede usarlos.

Opciones además del nivel de instrucción, secuencias manejadas por datos existen.. También pueden usarse secuencias manejadas por datos en un hilo desmenuzado convencionalmente, siendo al secuencia manejada por control en el nivel de instrucción[10]. En esta organización, se condensan instrucciones de uno o varios cómputos relacionados en hilos totalmente ordenados que implícitamente especifican relaciones del flujo de datos. Hilos individuales, asignados a elementos de procesadores, secuencia y ejecutan de la manera manejada por control.

Sin embargo, se representan las relaciones del flujo de datos entre los hilos explícitamente, y la creación del hilo se activa en una manera manejada por datos – por la disponibilidad de sus entradas de los datos del resultado de un hilo anterior. Los hilos manejados por datos que esperamos ver en procesadores futuros tomarán esta forma probablemente.

2.2.1 Hilos no especulativos manejado por datos.

Los Leguajes imperativos no pueden llevar a cabo multithreading no especulativo manejado por datos fácilmente. La principal barrera surge de la incapacidad de un programa imperativo para especificar un flujo de datos explícito a priori que represente el programa porque la información del flujo de datos sólo se aplica dentro de unos límites bien definidos. Un error tanto en la remisión de datos cambia el significado del programa. La conversión automática de código imperativo a la forma de flujo de datos explícita ha sido el tema de algunas investigaciones, pero representaciones del programa manejado por datos generalmente pueden construirse sólo para código escrito en idiomas funcionales, manejados por datos.

2.2.2 Hilos especulativos manejados por datos.

Multithreading No especulativo manejado por datos padece dos problemas mayores. Primero, los programas generalmente no pueden ser divididos en los hilos manejados por datos. Segundo, incluso en casos donde una división es posible, la representación resultante rompe la semántica secuencial creada por el programador y destruye la correlación entre el programa que se ejecuta y la fuente codificada del que fue derivado. La semántica secuencial, o por lo menos su apariencia, es muy importante para el desarrollo del programa , debugging, y la interacción con componentes del sistema no manejados por datos y otras tareas. La pérdida de semántica secuencial causa consecuencias más serias que la perturbación al programador simplemente.

De nuevo, la especulación podría resolver estos problemas si cambiamos nuestro intento de alcanzar el multithreading. Empezamos con la idea que una semántica secuencial requiere la presencia de un hilo principal o una arquitectura que ejecute el programa por completo. Tales hilos principales significan que los hilos manejados por datos pueden realizar sólo el trabajo auxiliar y que su presencia exigirá inevitablemente algún recargo en el sistema total. Sin embargo, el concepto del hilo principal significa también que dividiendo un programa en hilos disjuntos no son estrictamente necesarios y ese hilos especulativos manejados por datos pueden concentrarse en reforzar el rendimiento en tareas sin las obligaciones de exactitud. Es probable que los hilos manejados por datos jugarán el papel de hilos del auxiliador que corren delante de un hilo principal y absorben latencias de la micro arquitectura en su nombre.

Nosotros creemos que los programas inherentemente contienen varios niveles de ILP, disimulados por eventos de micro arquitectura de latencia larga como la “cache misses” y los“branch mispredictions” de la rama. El rendimiento alto de un solo hilo puede lograrse si pueden quitarse estas latencias que se pondrán relativamente más largas de algún modo probablemente o puedan esconderse del hilo principal. Esta tarea puede ser lograda aumentando el programa con hilos del auxiliador manejado por datos que pre-ejecutan los cómputos de instrucciones del problema antes de que ellos causen paros en el hilo del programa principal.

El modelo del auxiliador copia seleccionados cómputos del programa en los hilos manejados por datos[11,12]. Mientras el programa todavía se ejecuta como un solo hilo manejado por control, los hilos manejados por datos se anidan en ciertos puntos en el programa principal que computa alguna instrucción futura. Cuando el hilo del programa principal se pone al día con el hilo manejado por datos, tiene la opción de usar el resultado o repetir el cómputo, aunque con una latencia reducida.[12]

La especulación se entrelaza con el estado del reducido auxiliador de hilos manejados por datos. La última responsabilidad del hilo principal para la interface de arquitectura releva hilos manejados por datos de cualquier obligación de exactitud. Sin estas obligaciones, pueden construirse hilos manejados por datos usando información del flujo de datos disponible. Además, los hilos manejados por datos no necesitan comprender una parte completa del programa –su uso puede ser reservado solamente para esas situaciones que la más necesitan características reforzadas de paralelismo.

3. Arquitectura del Sistema

El soporte futuro para los hilos especulativos depende del descubrimiento de soluciones aceptables a varios problemas y va desde la aplicación del hilo a bajo nivel hasta estrategias del uso de los hilos a alto nivel.

La decisión más importante radica en la división de labor entre el programador, compilador, sistema operativo, y procesador. ¿Ejecutará el procesador los hilos, pero qué entidad debe ser la responsable para otras tareas relacionadas con los hilos como selección del hilo, creación de un

nuevo, fijación, asignación del recurso, y comunicación? Poniendo toda la responsabilidad en el procesador presenta una opción atractiva. Dado que los procesadores del futuro tendrán casi mil millones transistores, unos millón podría dedicarse fácilmente a las tareas de dirección multithreading-específicas.

Una aplicación de solo procesador no tiene ningún problemas de compatibilidad n hacia atrás ni hacia adlante; conserva la interface del sistema actual mientras refuerrza el rendimiento de software legado. Esta implementación tiene inconvenientes, sin embargo, incluyendo la complejidad del diseño agregada y la rigidez asignada y simplicidad de selección de hilo y algoritmos de dirección se pueden obtener resultados bastantes auspiciosos..

Porque la selección del hilo presenta un problema importante y delicado, nosotros podríamos asignar esta función al software o incluso el programador, produciendo por lo tanto mucho mejores divisiones de hilos. Sin embargo, cualquier camino en el que la información del multithreading fluye de o a través del software al hardware requiere un cambio en la interface del software-hardware. Cosas así se encuentran con resistencia, sobre todo si ellos requieren un cambio en la semántica de la arquitectura que debe llevarse a cabo. Los intentos exitosos probablemente serán aquellos que sólo perturben ligeramente una arquitectura existente, o preferentemente nada.

Esperamos que esa selección del hilo se lleve a cabo en software y se llevada al hardware, pero de una forma consultada. Las instrucciones del prefetch encontradas en recientes arquitecturas proporcionan un ejemplo de información consultada. En este modelo, el hardware puede actuar totalmente sobre la información, parcialmente, o selectivamente, o incluso la ignora, todos sin impactar en la exactitud. El procesador mantiene la opción de reforzar o refinar dinámicamente también esta información. Restringiendo la información del hilo especulativo a un papel asesor releva a la arquitectura de muchas garantías de funcionalidad que estorbarían aplicaciones de la o-generación futura.

Conclusión

Debemos desarrollar varias tecnologías antes del multithreading especulativo se ponga común en los procesadores corrientes. Estas tecnologías incluyen medios para llevar información del hilo del software al hardware, algoritmos para la selección del hilo y dirección, y el hardware y software para soportarr la ejecución simultánea de una colección de hilos especulativos y hilos no especulativos. Se está investigando varios aspectos de procesadores del multithreaded especulativos. Este trabajo incluye identificar patrones del problema que podrían beneficiar al multithreading, identificando y seleccionando hilos apropiados, determinando métodos para llevar la información del hilo del software al hardware, e investigando los aspectos de la micro arquiitectura para ejecutar un conjunto de hilos especulativos y no especulativos simultáneamente de una manera aprovechable. El desafío principal que abarca todos los aspectos de este trabajo involucra encontrar hilos apropiados para las situaciones de los problemas pertinentes para que pueda lograrse un beneficio adecuado cuando ellos se ejecutan en un micro arquitectura subyacente.

Referencias

1. J. Emer, "Multithreading Simultáneo: Multiplicando la Actuación de Alfa," la presentación, Foro del Microprocesador, Oct. 1999.
2. G.S. Sohi, S. Breach, y T.N. Vijaykumar, "los Procesadores de Multiscalar," Proc. 22 Int'l Symp. Arquitectura de la computadora, ACM Press, Nueva York, 1991, pp,. 414-425.

3. D.M. Tullsen et al., "Aprovechándose de Opción: La instrucción Saca y Emite en un Implementable el Procesador de Multithreading Simultáneo," Proc. 23 Int'l Symp. Arquitectura de la computadora, ACM Press, Nueva York, 1996, pp,. 191-202.
4. L. Hammond, B.A. Nayfeh, y K. Olukotun, "UN Solo-astilla Multiprocessor," la Computadora, Sept. 1997, pp. 79-85.
5. M. Franklin y G.S. Sohi, "ARB: Un Mecanismo del Hardware por el Pedir de nuevo Dinámico de Referencias de Memoria," IEEE Trans. Computadoras, 1996 de mayo, pp. 552-571.
6. A. el Moshovos et al., "la Especulación Dinámica y Sincronización de Dependencia de los Datos," Proc. 24 Int'l Symp. Arquitectura de la computadora, ACM Press, Nueva York, 1997, pp,. 181-193.
7. M. el Tremblay et al., "La Arquitectura de MAJC: Una Síntesis de Paralelismo y Scalability," IEEE Micro, Nov. /Dec. 2000, pp. 12-25.
8. N. el Nishi et al., "UN 1-GIPS 1-W Solo-astilla Acopló Cuatro-manera Herméticamente Multiprocessor con Apoyo de la Arquitectura por la Ejecución de Mando-flujo Múltiple," Proc. 47 Int'l IEEE los Circuitos Transistorizados Conf., IEEE Press, Piscataway, N.J., 2000, pp. 418-475.
9. Arvind y R.S. Nikhil, "Ejecutando un Programa en el Etiquetar-ficha de MIT la Arquitectura de Dataflow," IEEE Trans. Computadoras, Mar. 1990, pp. 300-318.
10. R.A. Iannucci, "Hacia un Dataflow/von Neumann la Arquitectura Híbrida," Proc. 15 Int'l Symp. Arquitectura de la computadora, ACM Press, Nueva York, 1988, pp,. 131 -140.
11. R.S. Chappell et al., "Microthreading Subordinado Simultáneo (SSMT)," Proc. 26 Int'l Symp. Arquitectura de la computadora, ACM Press, Nueva York, 1999, pp,. 186-195.

Apéndice A

Piranha: Exploiting Single-Chip Multiprocessing

Luiz André Barroso*Compaq Computer Corp.*
Kourosh Gharachorloo*Compaq Computer Corp.*
Tom Heynemann*Compaq Computer Corp.*
Dan Joyce*Compaq Computer Corp.*
David Lowell*Compaq Computer Corp.*
Harland Maxwell*Compaq Computer Corp.*
Joel McCormack*Compaq Computer Corp.*
Ravishankar Mosur*Compaq Computer Corp.*
Jeff Sprouse*Compaq Computer Corp.*
Robert Stets*Compaq Computer Corp.*
Scott Smith*Compaq Computer Corp.*

Today's microprocessor industry struggles with escalating development and design costs, which arise from exceedingly complex processors that push the limits of instruction-level parallelism. Meanwhile, such designs yield diminishing returns and are ill-suited for commercial applications such as database and Web workloads, which constitute the high-performance servers' most important market. These server applications typically suffer from large memory stall times, exhibit little instruction-level parallelism, and have no use for high-performance floating-point or multimedia functionality.

Fortunately, increasing chip densities provide architects with many options for tackling design complexities while addressing commercial applications' needs. Integrating all system-level components onto the processor die—as the upcoming Alpha 21364 does—enables a more efficient memory hierarchy without further increasing design complexity. Beyond that, exploiting the abundance of thread-level parallelism in commercial workloads through simultaneous multithreading or chip multiprocessing seems promising. The chip multiprocessing approach is particularly useful for addressing design complexity because it enables using simpler cores.

The Piranha project's[1] primary goals are to

- build a system that achieves superior performance on commercial workloads, and
- effectively address design cost and complexity issues.

Our research prototype aggressively exploits chip multiprocessing by integrating eight simple Alpha processor cores along with a two-level cache hierarchy, memory controllers, coherence protocol engines, and an interconnect router onto a single chip. Combining simple, single-issue in-order processor cores with an industry-standard ASIC design methodology should let us complete our design with a shorter schedule and smaller team and budget than a commercial microprocessor requires.

Although each Piranha processor core is substantially slower than a conventional next-generation processor because of its simpler design and the constraints of an ASIC process, integrating eight cores onto a single chip provides Piranha with a twofold to threefold performance margin on important commercial workloads. This advantage can approach a factor of five using full-custom instead of ASIC logic.

Most of Piranha's architectural innovations lie in the memory and interconnect subsystems, including a shared eight-way banked, eight-way associative noninclusive L2 cache; highly specialized microprogrammed coherence protocol engines; and an aggressive two-level coherence protocol. Piranha is particularly efficient for applications with little instruction-level parallelism because it can exploit thread-level parallelism to issue multiple independent misses and better utilize its aggressive memory. In addition, significant constructive cache interference among threads for applications such as databases lets a relatively small, 1-Mbyte L2 cache effectively handle the eight processor cores.

While the exceedingly complex general-purpose architecture of most current processors is not optimal for any given application domain, Piranha's focused design targets an important market segment at the possible expense of other workloads, resulting in superior performance and improved time to market.

Reference
1. L.A. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 2000), ACM Press, New York, June 2000, pp. 282-293.
The authors are affiliated with Compaq Computer Corp. Contact **Luiz André Barroso** at Luiz.Barroso@compaq.com.

Apéndice B

Cray MTA: Multithreading for Latency Tolerance

Burton Smith*Cray*

In the uniform shared-memory programming model, computer system performance does not depend significantly on data placement in memory. The Cray MTA implements this model using a very high bandwidth interconnection network and the parallelism generated by fine-grained multithreading to tolerate memory latency.
The MTA system accommodates up to 256 custom multithreaded processors. Instead of data caches, each processor switches context every cycle among as many as 128 instruction streams, or hardware threads, choosing on each cycle only from those streams ready to execute their next instruction. The processors thereby tolerate up to 128 cycles of memory latency while achieving high utilization. Further, each stream can issue as many as eight memory references without waiting for earlier ones to finish, augmenting the processors' memory latency tolerance to a maximum of 1,024 cycles.
The instructions are 64 bits wide and can contain a memory reference operation, an arithmetic or logical operation, and a branch or simple arithmetic or logical operation. By issuing an instruction on

nearly every cycle, each processor achieves a sustainable two operations per cycle, irrespective of data placement in memory.

The MTA's thread-based programming model permits mixing implicit and explicit parallelism. The virtual machine has an unbounded number of processors with uniform access to all memory locations. The programmer can specify an unbounded number of threads that interact via shared data structures. The runtime system acquires physical resources from the operating system and uses these resources to implement the virtual machine.

The runtime environment normally multiplexes hardware streams among an unbounded number of threads. If the runtime environment encounters insufficient parallelism to fully occupy its processor resources, it surrenders some protection domains to the operating system for allocation to other tasks. Conversely, the runtime demands additional processor resources as parallelism increases.

In addition to providing a high level of single-thread optimization, MTA compilers perform automatic parallelization of Fortran, C, and C++ source code. The compilers restructure loop nests to enhance parallelism. They perform whole-program analysis and optimization, including in-line expansion and parallelization of loops containing function calls and input-output operations. Linear recurrences are automatically parallelized, as are reductions even in the presence of unknown dependences—as in histogramming, for example.

The programmer can insert pragmas and directives to help the compiler perform automatic parallelization. The parallelism that the compiler discovers and exploits supplements whatever the user explicitly generates. Inner-loop parallelism is easy for the compilers to discover and manage, whereas outer-loop parallelism may be evident only to the programmer.

The tremendous software support that the MTA provides for parallel programming depends crucially on its uniform shared-memory architecture, which multithreading alone makes possible.

Burton Smith is chief scientist at Cray in Seattle. Contact him at burton@cray.com.