Tesis Doctoral

Nuevos mecanismos hardware-software para explotar paralelismo en sistemas multinúcleo masivos

González Márquez, David Alejandro

2017-10-02

UNIVERSIDAD DE BUENOS AIRES
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Nuevos mecanismos hardware-software para explotar paralelismo en sistemas multinúcleo masivos

Tesis presentada para optar al título de Doctor de la
Universidad de Buenos Aires en el Área Ciencias de la Computación

Lic. David Alejandro González Márquez

Director de tesis: Dr. Esteban Mocskos
Director Asistente: Dr. Adrián Cristal
Consejero de estudios: Dr. Diego Fernández Slezak

Jurados:

Dr. Augusto J. Vega (IBM Research)
Dr. Elías Todorovich (UNICEN)
Dr. Pablo M. Ezzatti (UdelaR)

Fecha y lugar de defensa
2 de Octubre de 2017
Buenos Aires, Argentina

Firma

Nuevos mecanismos hardware-software para explotar paralelismo en sistemas multinúcleo masivos

Abstract

Uno de los componentes principales de cualquier computadora de propósito general es el *procesador*. Éste se puede encontrar en sistemas tan diversos como servidores o sistemas de cómputo de alto rendimiento (HPC) hasta plataformas portátiles como tablets, inclusive en *smartcards*. Su principal tarea es ejecutar programas tan rápido como sea posible considerando limitaciones como el costo del propio sistema y su consumo de energía. Los procesadores *multicore* se pueden encontrar en todos los segmentos del mercado, desde procesadores embebidos hasta procesadores para HPC, pero pocas aplicaciones pueden hacer uso eficiente de estos recursos computacionales. Los actuales *frameworks* tienen como objetivo soportar paralelismo a nivel de thread en aplicaciones o permitir sincronizar el cómputo por medio de pasaje explícito de mensajes, pero el *overhead* que se agrega al utilizar estas técnicas impide su uso para instancias pequeñas de problemas.

En esta tesis se presenta *Micro-threads* (*Mth*), una propuesta co-diseñada de hardware-software enfocada en la administración de threads y a permitir que las aplicaciones puedan acceder a recursos paralelos con un overhead reducido. Se busca, entonces, ejecutar eficientemente fragmentos de código pequeños o instancias de problemas que, de otra manera, resultaría impracticable debido al overhead. El mecanismo propuesto se basa en delegar el control de los recursos a las aplicaciones y a mejorar la forma en que se almacena y carga un contexto de ejecución, junto con un eficiente sistema de sincronización de threads.

Los experimentos realizados sobre la propuesta se enfocan sobre dos aspectos, por un lado, un conjunto de algoritmos paradigmáticos bien definidos que cubren un amplio espectro de formas de sincronización y patrones de cómputo: filtro de conversión a HSL (trivialmente paralelizable), FFT Radix2 (algoritmo recursivo), descomposición LU (barrera de sincronización en cada ciclo) y algoritmo de camino mínimo de Dantzig (basado en grafos y operaciones sobre matrices). Los resultados muestran un notable speedup para todos los casos, incluso para instancias de tamaño reducido, donde su tamaño no supera los cientos de bytes. Además, el estudio muestra que la inicialización y sincronización no impacta de forma significativa sobre *Mth* a diferencia de otras tecnologías estudiadas.

Por otro lado, se analizó el comportamiento de *Mth* sobre una aplicación de interés real y amplio uso. Los motores de base de datos (DBMS) constituyen no solo una parte fundamental de cualquier negocio, sino que también son utilizados diariamente en sistemas embebidos en aplicaciones móviles.

Se propuso implementar modificaciones a un DBMS embebido utilizando *Mth* de manera que pudiera soportar la resolución de planificaciones paralelas de consultas. La propuesta incluye dar soporte al paralelismo utilizando *Mth* y un esquema para coordinar los recursos resolviendo planificaciones en múltiples máquinas virtuales. La combinación de ambos enfoques produjo un uso eficiente de los recursos computacionales, incluso para consultas de muy bajo volumen de datos, permitiendo la ejecución en paralelo de consultas que no pueden ser resueltas de esta forma por los DBMS actuales. Los resultados muestran una notable ganancia en términos de reducción del tiempo de ejecución y consumo de energía utilizando *Mth* aplicado a una DBMS embebida (SQLite).

En términos generales, los resultados obtenidos alientan la utilización del esquema propuesto en *Mth*, que aparece como una solución gradual al uso de múltiples cores en aplicaciones que actualmente no pueden aprovechar el incremento en la cantidad de recursos paralelos disponibles.

New hardware-software mechanisms to exploit parallelism on masive multicore systems

Abstract

One of the main components of any general-purpose computer is the *microprocessor*, which can be found in the heart of every computer: from standard servers and high performance computing nodes to portable mobile platforms. Its main task is to correctly execute programs as fast as it can, having the production cost and the energy consumption as design bounds. Multi-core processors are ubiquitous in all market segments from embedded to high performance computing, but only few applications can efficiently utilize them. Existing parallel frameworks aim to support thread-level parallelism in applications, but the imposed overhead prevents their usage for small problem instances.

This work presents *Micro-threads* (*Mth*) a hardware-software co-designed proposal focused on a shared thread management model enabling the use of parallel resources in applications that have small chunks of parallel code or small problem inputs by a combination of software and hardware: delegation of the resource control to the application, an improved mechanism to store and fill processor's context, and an efficient synchronization system.

In this work, we firstly demonstrate the factibility of our proposal using a set of algorithms: HSL filter (trivially parallel), FFT Radix2 (recursive algorithm), LU decomposition (barrier every cycle) and Dantzig algorithm (graph based, matrix manipulation). Remarkable speedups and efficiency are obtained in all cases, even when dealing with instance size of order of hundreds of bytes. Moreover, initialization and synchronization do not impact in its behavior showing remarkable speed-up in the wide range of analyzed cases.

To complete the evaluation of *Mth*, we modified an embedded relational database engine (DBMS) to support parallel operation. Database servers constitute not only the backbone of almost every enterprise, but also are used daily as embedded DBMS by millions of users in mobile applications, to store, manage and retrieve data without the need of special administration or configuration. We propose the use of *Mth* processor architecture (hardware side) with a parallel resources coordination schema (software side) in an embedded DBMS modified to support parallel query solving. The combination of both approaches produces an efficient use of multicore processors even when the workload is small, enabling the parallel execution of queries that are out of reach of the parallel support in current DBMS. We show that parallel query execution of very small queries can be treated with remarkable gains in terms of execution time and energy consumption using *Mth* which is applied to a broadly used embedded relational DBMS (i.e. SQLite).

The results encourage the use of *Mth* and could smooth the use of multiple cores for applications that currently can not take advantage of the proliferation of the available parallel resources in each chip.

Dedicado a Silvana, mi familia y amigos.

Índice general

Capítulo 1

Introducción

Uno de los componentes principales de cualquier computadora de propósito general es el *procesador*. Éste se puede encontrar en sistemas de cualquier tipo, desde servidores o sistemas de cómputo de alto rendimiento hasta plataformas portables como tablets, hasta incluso en *smartcards*. Su principal tarea es ejecutar programas tan rápido como sea posible considerando como limitantes, el costo del propio sistema y su consumo energético.

La investigación en arquitectura de procesadores, centra sus esfuerzos en optimizar sus diseños de acuerdo al uso específico que éstos tendrán y a las características de rendimiento esperadas. Para esto consideran las tecnologías tanto actuales como futuras. La optimización de los diseños en procesadores se basa en diferentes parámetros: rendimiento, consumo, superficie y costo de producción.

En esta tesis, el término *rendimiento* hace referencia a la cantidad trabajo que puede ser resuelta por unidad de tiempo, ya sea trabajo medido como instrucciones u operaciones básicas y tiempo medido en ciclos o segundos. Por otro lado, el *consumo* se medirá como la energía que utiliza un sistema bajo una determinada carga. El consumo depende de factores tales como la tecnología de fabricación o la complejidad del sistema. El factor *superficie* en procesadores refiere al tamaño del sustrato de silicio utilizado para su fabricación, esto depende del diseño y de la combinación de todas las variables anteriores.

Actualmente, las limitaciones tecnológicas en la carrera por mejorar el rendimiento, llevaron a que los procesadores de altas prestaciones sean multicore y multithreads, derivando en una fuerte actividad de investigación en este campo. Esto último produjo que se puedan encontrar procesadores multicore en prácticamente cualquier dispositivo. El paralelismo permite acelerar aplicaciones ejecutando múltiples operaciones independientes de forma concurrente [17]. El paralelismo en procesadores se puede encontrar a tres niveles: instruction-level parallelism (ILP), thread-level parallelism (TLP) y data-level parallelism (DLP).

La figura 1.1 ilustra los distintos tipos de paralelismo. ILP considera la ejecución simultánea de instrucciones lograda por técnicas en la arquitectura del procesador. El hardware dinámicamente detecta cuando dos instrucciones, dentro de una ventana, pueden ser ejecutadas de forma simultánea en distintas unidades funcionales. Estas técnicas están limitadas a una ventana de instrucciones sobre las que analizan las dependencias entre instrucciones. Existen otras formas de lograr ILP por parte del software, en este caso, las instrucciones deben estar ordenadas de forma compatible con las restricciones de la arquitectura para que éstas sean ejecutadas simultáneamente, ya sea con un orden implícito o explícito. TLP es un tipo de paralelismo dado por la ejecución de múltiples threads de forma simultánea en diferentes cores lógicos. Los múltiples threads son ejecutados concurrentemente ordenados según el sistema operativo o el framework utilizado. Una forma común de TLP es el patrón de diseño tipo *pipeline*, que consiste en resolver conjuntos de datos a lo largo de una serie de tareas donde cada una es ejecutada independientemente de las otras. Por otro lado, DLP consiste en distribuir datos en varios cores, que ejecutan en paralelo las mismas instrucciones, en general de forma acoplada.

Técnicas para obtener y soportar ILP

Las principales técnicas utilizadas para obtener paralelismo a nivel de instrucción son:

- **Segmentación**: La ejecución de las instrucciones se divide en etapas, el conjunto de estas etapas se denomina *pipeline*. Cada instrucción pasa por todas las etapas una a una a medida que se resuelve. Esto permite que, a medida que se libera una etapa, una nueva instrucción pueda tomar su lugar. De esta forma se podría tener una instrucción por etapa. La segmentación permite que cada etapa tenga menos trabajo que hacer y que demore menos tiempo. Se gana principalmente debido a que se reduce el tiempo de ciclo de cada instrucción superponiendo parte del proceso de resolver una instrucción junto con otras. La segmentación, por otro lado, trae aparejado problemas denominados *hazards*. Un ejemplo son las dependencias de datos, donde una instrucción necesita del resultado de la instrucción anterior para poder ser ejecutada.
- **Predicción de saltos y ejecución especulativa**: Cuando en un *pipeline* se encuentra una instrucción de control sobre el flujo del programa, no se puede conocer cuál será la próxima instrucción a ejecutar hasta que no finalice la instrucción de control. Esto implica que la instrucción deba recorrer todo el *pipeline* dejando desiertas las primeras etapas del mismo, lo que se conoce como un *stall*. Para mitigar esta situación y aprovechar los ciclos de ejecución, se recurre a dos técnicas:

  i) *Predicción de saltos*: consiste en predecir cuál será la próxima instrucción a ejecutar.
  ii) *Ejecución especulativa*: consiste en ejecutar las instrucciones según se predijo, teniendo la capacidad para desecharlas en caso de que la predicción fuera incorrecta.

- **Ejecución fuera de orden**: Permite tener un conjunto de instrucciones en vuelo simultáneamente, el procesador analiza las dependencias dentro de un subconjunto de instrucciones y las ejecuta sin seguir el orden original del programa. Luego hace visibles los cambios producidos por las instrucciones en la memoria en el orden correcto del programa. Esta técnica resulta muy útil para soportar latencias de los fallos de las cachés de primer y segundo nivel.
- **Caching and Prefetching**: La relación entre la velocidad de operación de la memoria y los procesadores, típicamente difiere en dos órdenes de magnitud a favor del procesador [21]. La latencia en el acceso a memoria es uno de los principales problemas que enfrentan los procesadores. Para mitigar el desacople entre procesador y memoria, se utilizan dos técnicas:

  – *Memoria caché*: la utilización de memoria caché consiste en uno o más niveles de memoria cuyo acceso es mucho más rápido que la memoria principal del sistema. El tiempo de acceso permite soportar la ejecución fuera de orden o el uso de múltiples hilos.
  – *Prefetching*: las técnicas de pre-búsqueda intentan predecir cuál será el patrón de acceso a la memoria para accederla antes. De esta forma, cuando el programa necesite los datos, éstos ya estén en la memoria caché y se evite tener que esperar a ser servidos desde la memoria principal.

- **Simultaneous Multithreading (SMT)**: Esta técnica se basa en el hecho de que los recursos de los procesadores no se aprovechan totalmente. Por ejemplo, cuando hay un fallo en la caché y el procesador tiene que acceder a memoria por el dato. En este caso, el procesador estaría inactivo durante un tiempo considerablemente alto. Otro caso es cuando no hay paralelismo a nivel de instrucción o la tasa de fallos del predictor de saltos es alta, en todos estos casos los recursos del procesador no están siendo bien aprovechados. La idea que hay detrás del SMT no es aumentar el rendimiento de un programa en particular sino aumentar el rendimiento del procesador permitiendo que más de un programa comparta los recursos del procesador. En este caso la ejecución de cada programa individualmente podría tomar más tiempo, pero menos que si fueran ejecutados secuencialmente. Por lo tanto, los recursos del procesador son mejor utilizados.

Estas técnicas son utilizadas en la mayor parte de los procesadores actuales mejorando significativamente su rendimiento. No obstante, las técnicas mencionadas tienen un alcance muy limitado sobre una ventana pequeña de instrucciones. Es decir, eventos que sucedan fuera de esta ventana no son visibles por técnicas de paralelismo a nivel de instrucción y, por ende, no pueden ser objeto de su consideración para lograr una ejecución más eficiente.

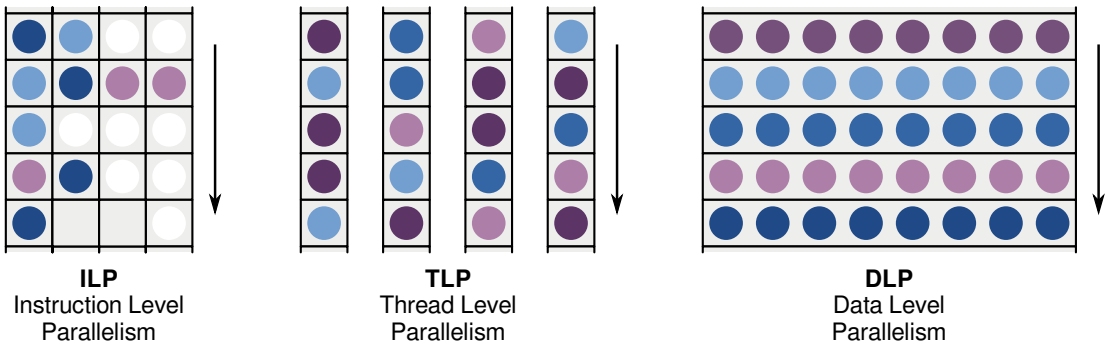


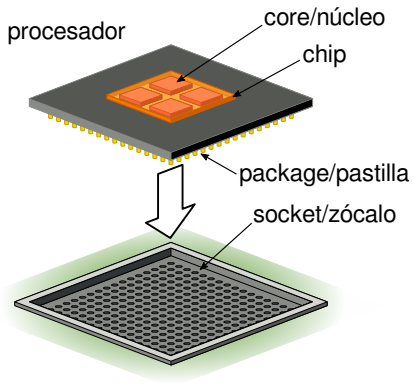
Figura 1.1: Diferentes niveles en los cuales se puede aplicar técnicas de paralelismo: ILP utiliza múltiples unidades funcionales en un procesador, TLP saca provecho de varios cores y DLP hace una asignación basada en los datos a procesar.

Para mejorar ILP, existen técnicas que pueden sacar provecho de las múltiples unidades de ejecución presentes en los procesadores, ejecución fuera de orden, ejecución especulativa y predicción de saltos que, implementadas a altas frecuencias de reloj, llevan a un incremento significativo de la complejidad de los diseños, baja utilización del área de silicio y un aumento extremo en disipación de energía [44].

El principal camino que, históricamente, se siguió en la optimización del rendimiento de procesadores fue la mejora del paralelismo de instrucciones (ILP) y el aumento en la frecuencia de reloj [18]. Otro factor sobre el que se enfoca la mejora en el rendimiento de procesadores es la tecnología de integración. La mejora en este aspecto limita el consumo y las características de interconexionado.

Las futuras arquitecturas deberán considerar paralelismo explícito para permitir continuar incrementando el paralelismo a nivel de instrucciones y minimizar el interconexionado, haciendo uso de la localidad de los datos y permitiendo arquitecturas heterogéneas para mejorar el rendimiento energético.

Nomenclatura



Llamamos *core* o núcleo a una de las unidades de procesamiento dentro de un procesador. Un procesador, por su parte, está formado por cores, junto con memorias caché y controladores. Tanto procesador como microprocesador se utilizarán como sinónimos. La palabra CPU, del inglés (*Central Processing Unit*) tendrá el mismo significado que procesador, a excepción que se utilice como "modelo de CPU" donde se referirá a la forma de implementar un core por parte del simulador. Desde el punto de vista físico, un procesador está compuesto por un circuito (*chip*) en un encapsulado (pastilla). El lugar donde un procesador es colocado se denomina zócalo o *socket*.

Típicamente, un *proceso* consiste en un estado de registros del CPU, una pila de kernel, un directorio de trabajo, una lista de descriptores de archivos, una tabla de señales, un usuario y grupo de trabajo, un mapa de memoria, entre otros datos dependiendo del sistema. El estado de la CPU incluye un *stack pointer* (SP), que apunta al *stack frame* actual, el *program counter* (PC), que indica cual es la próxima instrucción y los registros restantes que almacenan información tanto local como global de la aplicación.

Intercambiar procesos que se ejecutan concurrentemente introduce overhead. El estado de la CPU junto con cualquier otra información del proceso es almacenada, el Translation Lookaside Buffer (TLB) es, en general, limpiado (*flushed*) y el estado del nuevo proceso es cargado en la CPU, junto con toda la información necesaria para poder ejecutarlo. Comunicar información entre procesos puede ser un procedimiento igual de complejo.

Un *thread* es un proceso liviano desarrollado para superar estos problemas. Múltiples threads pueden coexistir dentro de un proceso, compartiendo el mapa de memoria, los descriptores de archivo, el código y información global. El estado de cada thread está solamente compuesto por el contador de programa, la pila, el puntero al tope de la pila, los registros y una pequeña cantidad de información adicional para administrarlo. Permitir la creación de threads administrados por el usuario es una forma de evitar la excesiva interacción entre el kernel y el espacio de usuario y, por ende, del overhead que se generaría. Minimizar esta interacción permitiría obtener un mejor rendimiento en la paralelización de trabajos muy pequeños. Crear y ejecutar un solo thread dentro de una aplicación introduce un overhead dado por la creación del thread en sí, la asignación de recursos y otros costos asociados al scheduler que puede generar a la hora de ejecutar el thread [35].

El paralelismo obtenido por utilizar múltiples threads introduce overhead dentro del propio programa por tres razones: (1) Costos de creación, scheduling y finalización de threads. (2) Cuando varios threads acceden a los mismos datos, se requiere de código adicional para prevenir condiciones de carrera o resultados inesperados. (3) Los threads puede ser ejecutados en cualquier orden, esto puede implicar que no se aplicasen algunas de las optimizaciones de código para ciclos.

Actualmente, los procesadores high-end de venta masiva pueden soportar 16 o más threads simultáneos por zócalo (CPU socket). La mayoría de los nodos de cómputo y estaciones de trabajo pueden contener múltiples zócalos, permitiendo tener más de una pastilla en el sistema ejecutando de forma simultánea. El contar con aplicaciones que logren utilizar de manera eficiente este hardware, podría mejorar notablemente su rendimiento. Una de las maneras más usuales de trabajo de este tipo de aplicaciones consiste en dividir el problema a resolver en múltiples partes que son, luego, distribuidas entre los threads disponibles. Éstos son ejecutados simultáneamente resolviendo cada parte del problema de forma independiente. Aquellas aplicaciones que solo utilicen un único thread o unos pocos, no podrán beneficiarse con este tipo de arquitecturas [8].

Desarrollar software que no tenga en cuenta el uso de tecnologías que involucren múltiples procesadores, llevará a construir aplicaciones que utilicen un solo thread, sin obtener beneficio alguno de los recursos computacionales disponibles en las plataformas de cómputo actuales y en las que aparecerán en el futuro próximo. En este sentido, `OpenMP` es un framework centrado en el cómputo científico que tiene como objetivo dar soporte al desarrollo de aplicaciones multi-thread. Fürlinger et. al. analizaron la escalabilidad y el overhead de aplicaciones `OpenMP` [13]. Definen cuatro categorías como fuentes de overhead, *synchronization*, *load imbalance*, *limited parallelism* y *thread management*, éstas pueden ser cuantificadas y analizadas. Estudiando el código de ciertas aplicaciones, logran identificar las características clave que limitan la escalabilidad. Mostrando entonces que analizar y entender la escalabilidad de aplicaciones es un paso importante en el proceso de desarrollo de software paralelo. La diferencia entre la paralelización óptima y la obtenida, puede resultar insignificante cuando la cantidad de procesadores es baja o la cantidad de trabajo es muy alta, pero se torna en un elemento clave que termina limitando la capacidad de escalar de una aplicación.

La tarea de dividir los programas en threads para ser ejecutados en paralelo es bastante sencilla para aplicaciones con un patrón de cómputo regular, como suele ser el caso en aquellas utilizadas para simulación numérica. A pesar del overhead introducido por ejecutar programas en paralelo y, siempre que el trabajo a realizar sea suficiente, los compiladores actuales junto al framework de paralelización, logran administrar eficientemente threads. Sin embargo, para programas de propósito general (i.e. no numéricos), los compiladores no logran sacar provecho del potencial paralelismo a nivel de threads eficientemente [29].

El origen de los beneficios obtenidos por la propuesta se basa en reducir las fuentes de overhead mencionadas anteriormente que, para casos de tamaño reducido, juegan un rol fundamental.

A pesar de los avances en la tecnología de los procesadores y su poder de cómputo, los sistemas operativos (especialmente UNIX y sus derivados) mantienen la misma forma tradicional de administrar los

recursos. Es claro, entonces, que las políticas de *scheduling* no lograron alcanzar el mismo nivel de desarrollo en comparación con la tecnología de procesadores. Además, solamente un subconjunto de aplicaciones puede hacer uso efectivo de los múltiples recursos computacionales disponibles [5].

Tradicionalmente, el sistema operativo es el responsable de gestionar los recursos compartidos de hardware: procesadores, memoria y Entrada/Salida. Esto funciona correctamente en sistemas en los cuales los cores actúan como entidades independientes. Sin embargo, en sistemas multicore que ejecutan threads de forma concurrente, éstos compiten por los recursos que provee la microarquitectura del procesador. Más aún, las políticas de los sistemas operativos convencionales no tienen el control fino sobre los recursos de hardware, entrando en conflicto con el diseño de la microarquitectura [33].

Además, la evolución de las aplicaciones paralelas, mezclando diferentes modelos de programación, promete proveer en un futuro cercano el poder de cómputo equivalente al de un cluster actual en un solo nodo. En este escenario, los sistemas operativos deberán evolucionar, pasando de scheduling de procesos (enfoque clásico) a scheduling de aplicaciones. Todos los procesos y threads dentro de una aplicación se deberían considerar como una sola entidad en lugar de tomarlos como un conjunto de threads. El sistema operativo, entonces, podría encontrar la mejor asignación para el scheduler, minimizando el overhead y la migración de procesos. De esta manera se podrá evitar un impacto negativo en el rendimiento y escalabilidad global [14].

Existen herramientas para hacer uso de múltiples threads, éstas permiten a las aplicaciones soportar TLP. `OpenMP`, como ya fue mencionado, es un framework orientado al cálculo científico. `Cilk` es una extensión que permite soportar paralelismo de tareas y datos. `Pthreads` surge como una interfaz del lenguaje `C` para soportar threads, fue especificada en el estándar `IEEE POSIX 1003.1c`. `Pthreads` presenta una interfaz simple, que permite al programador tener control completo de los threads de la aplicación. El programador es el encargado de crear y sincronizar threads manualmente, especialmente teniendo en consideración las áreas de memoria compartida. `Pthreads` proporciona un control detallado, pero aún así, la política de *scheduling* es administrada por el sistema operativo. Éste, usualmente, agrega overhead y, por lo tanto, si las tareas no tienen suficiente trabajo asignado, resulta ineficiente su utilización. La flexibilidad ofrecida por `Pthreads` permite al programador exponer más paralelismo en sus aplicaciones, pero requiere un mayor esfuerzo de desarrollo.

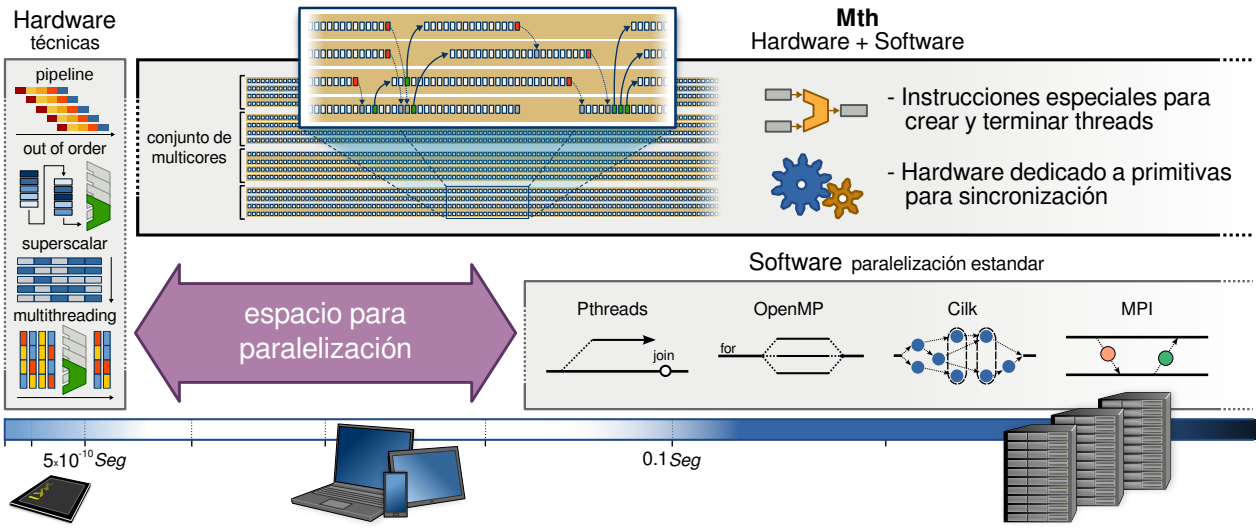


Figura 1.2: Las técnicas actuales de paralelización pueden exponer paralelismo a nivel de instrucciones o threads, pero el espacio entre estos dos tipos de mecanismos no es cubierto por ningún framework o modelo de programación. Nuestra propuesta busca ayudar a dar soporte para exponer paralelismo en este rango.

Estas técnicas mencionadas son representadas por el lado derecho de la figura 1.2. Son utilizadas para paralelizar aplicaciones en entornos de memoria compartida. El lado izquierdo, en cambio, ilustra las técnicas complejas que implementan los procesadores para mejorar el ILP, mencionadas anteriormente. Éstas limitan su accionar a una ventana relativamente pequeña de instrucciones, que suele ser del orden de cientos de instrucciones [32].

> En este trabajo se presenta *Micro-threads* (*Mth*) una propuesta de hardware y software enfocada en la administración de threads. El objetivo es permitir ejecutar eficientemente aplicaciones que utilicen paralelismo de grano fino (*fine-grain*), principalmente buscando paralelizar pequeñas secciones de código que las técnicas disponibles actualmente no pueden ejecutar de manera eficiente. La propuesta no requiere intervención de un *runtime* o del sistema operativo, la ejecución simultánea de múltiples threads es controlada directamente por la aplicación por medio de soporte de hardware. Esta propuesta busca la adopción del uso de múltiples procesadores, en aplicaciones que actualmente no pueden tener ninguna ventaja de la proliferación de los recursos paralelos disponibles en una misma pastilla.

La sección central de la figura 1.2 muestra el espacio de trabajo donde la propuesta busca mejorar el rendimiento. Sus límites son, por un lado, código donde no se pueden aplicar las técnicas de paralelismo a nivel de instrucciones, y por el otro, donde el volumen de trabajo a resolver es muy pequeño para que plataformas estándar lo pueda tratar eficientemente en paralelo.

En este universo se pueden resolver problemas en paralelo del orden de entre 10000 y 100000 ciclos muy eficientemente. En este tipo de problemas, consecutivas fallas de caché o errores en la predicción de saltos pueden tener costos altos en el rendimiento final obtenido.

1.1 Motivación

La métrica de *speedup* mide en terminos de latencia la mejora de un sistema con respecto a un sistema referencia. Ambos sistemas resolviendo el mismo problema. Un speedup de 1, significa que ambos sistemas demoran el mismo tiempo en resolver el problema. Un speedup de 2, implica que el sistema estudiado resuelve el problema en la mitad de tiempo que el sistema referencia. Mientras que un speedup menor que 1 muestra que el sistema demora más tiempo que el sistema referencia.

El speedup de un programa paralelo que utiliza múltiples procesadores está limitado por la cantidad de tiempo necesario para resolver la fracción secuencial del mismo. La Ley de Amdahl [3], establece que la mejora en rendimiento de un sistema, obtenida por modificar uno de sus componentes, está limitada a la fracción de tiempo que utilice dicho componente. La siguiente formula indica la mejora en tiempo, con respecto a la mejora de un componente.

$$T_m = T_o \cdot \left((1 - F_m) + \frac{F_m}{A_m} \right) \tag{1.1}$$

- F_m = fracción de tiempo que el sistema utiliza en el componente mejorado
- A_m = factor de mejora sobre el componente mejorado
- T_o = tiempo de ejecución original
- T_m = tiempo de ejecución mejorado

Esta formula puede ser reescrita en terminos de speedup.

$$Speedup = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}} \tag{1.2}$$

Adicionalmente, al paralelizar una aplicación secuencial se introduce overhead, limitando aún más el potencial paralelismo que se pueda alcanzar. Si la cantidad de trabajo disponible no es suficiente para que pueda ser paralelizada, la solución paralela puede, incluso, tomar más tiempo que la solución secuencial. El mecanismo usado para construir threads agrega un overhead adicional, que incluye la creación, asignación de recursos y los costos de scheduling [35].

Por su parte, *Mth* busca minimizar el overhead relacionado con la gestión de threads (creación, administración y sincronización), permitiendo el uso de los recursos a las aplicaciones, para que éstas, a su vez, puedan ser capaces de ejecutar pequeños fragmentos de código en paralelo o solucionar problemas en paralelo para tamaños de entrada muy reducidos. Esto es realizado mediante una combinación de software y hardware que tiene las siguientes características principales:

- delegación del control de recursos a las aplicaciones
- mejorar el mecanismo de carga de un thread en un procesador
- proporcionar un sistema para sincronizar eficientemente los threads

La figura 1.3 muestra una comparación entre diferentes mecanismos de administración de threads: `Pthreads`, `OpenMP`, `Cilk` y *Mth*. El código del experimento consiste en completar un arreglo de números aleatorios obtenidos utilizando un generador lineal congruencial. Este ejemplo ayuda a dar cuenta del tamaño de la tarea a resolver para poder superar el overhead de paralelización. Los tamaños del arreglo varían desde 2^6 a 2^{23} elementos, representando desde 256 B a 32 MB de memoria. Para ejecutar este experimento fueron utilizadas dos plataformas diferentes, mostrando como resultado el speedup relativo a la ejecución secuencial para cada una de las plataformas.

i) ARM en `gem5`
ii) Intel i7-920

La figura 1.3 deja claro que el speedup se incrementa a medida que el tamaño del problema es mayor para cualquiera de los mecanismos. En el caso de *Mth*, la versión paralela usando dos cores alcanza el speedup ideal con aproximadamente 1000 elementos, mientras que se necesitan 10^4 elementos para balancear el overhead en el caso de cuatro cores. En el resto de los casos se requieren más de 10^6 elementos con dos core, mientras que llegar a cuatro cores requiere del orden de 10^7.

1.2 Trabajo relacionado

Esta sección tiene como objetivo mencionar aquellos trabajos publicados que han formado el núcleo de los avances en los que se sostiene esta tesis. Se separan en tres aspectos relevantes: i) propuestas de software/hardware para administrar threads, ii) soporte de paralelismo de grano fino en sistemas operativos, y iii) propuestas o extensiones en arquitectura de procesadores.

En 1983, Fisher [10] introduce el término `VLIW` (Very Long Instruction Word) motivado por una técnica de compilación denominada *trace scheduling*. En la búsqueda por ejecutar programas cada vez más rápido, aparece la idea de planificar varias instrucciones de forma estática dentro de una sola instrucción de máquina, obteniendo así la posibilidad de ejecutar en paralelo un flujo de operaciones fuertemente acopladas. El objetivo era mejorar el valor IPC de 2 o 3 a más de 10.

Sin embargo, las técnicas que pueden ser aplicadas a nivel de procesador presentan serias limitaciones, como muestra el trabajo de Wall [49]. Éste explora diferentes configuraciones de *branch predictor*, *jump predictor*, *register renaming* y *alias analysis*, pero incluso en condiciones perfectas, los resultados del estudio resultan decepcionantes.

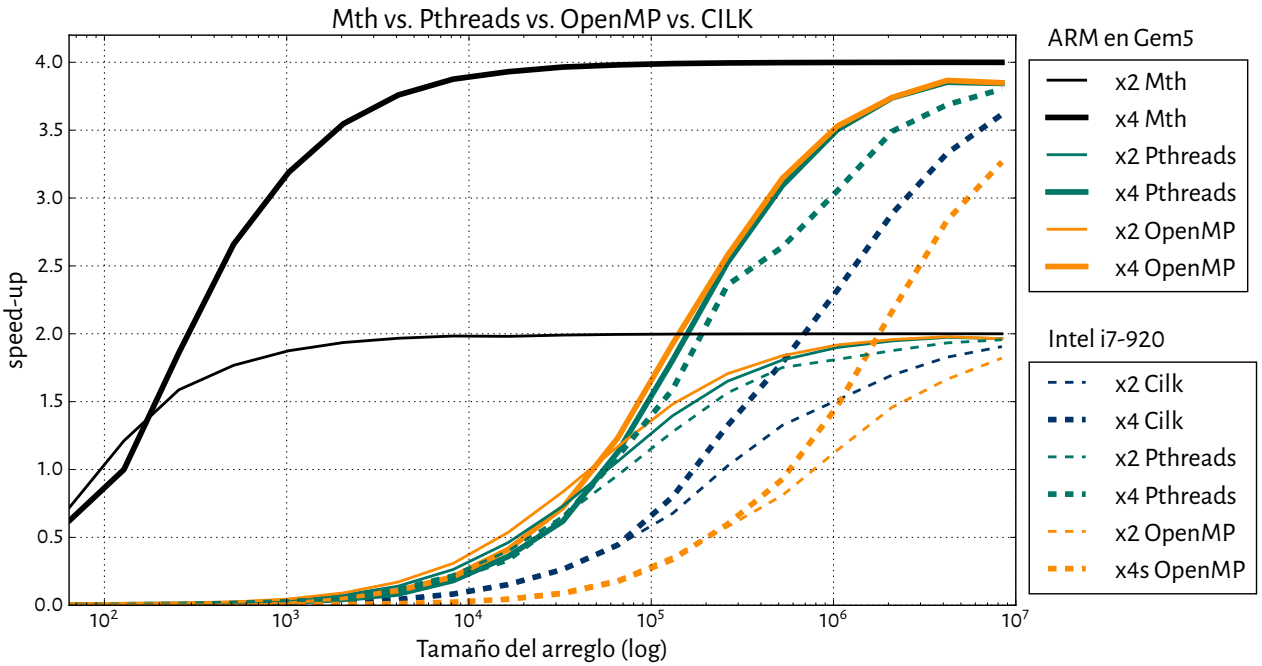


Figura 1.3: Overhead introducido al utilizar distintos frameworks de soporte al paralelismo: `Pthreads`, `Cilk`, `OpenMP` y *Mth*. Se reporta speed-up para dos y cuatro cores en diferentes plataformas en la generación de un vector de números aleatorios.

Llegado el año 1995, Sohi et al. [43] sientan las bases de los *multiscalar processors*, como un nuevo paradigma que permitiría exponer un mayor nivel de paralelismo a nivel de instrucciones. La idea fue dividir un programa en una colección de tareas por medio de una combinación de responsabilidades entre el software y el hardware. En términos actuales, tener varios threads de ejecución lógicos sobre la misma maquinaria de ejecución física.

En el mismo año se presenta una solución innovadora, que busca un objetivo similar, pero con notables cambios a nivel de microarquitectura. Fillo et al. [9] presentan M-Machine, una arquitectura de procesadores enfocada a exponer el paralelismo de grano fino para problemas de tamaño fijo. Se incluye mecanismos de comunicación entre registros en los que un thread puede directamente escribir un registro de otro thread, además se incorporan canales de comunicación directos entre procesadores. Un flujo de instrucciones es particionado por el compilador en threads horizontales (H-Threads), que son ejecutados concurrentemente en diferentes clusters de procesadores exponiendo ILP. La sincronización entre las unidades se realiza de forma explícita por medio de registros. Además, las unidades funcionales son compartidas temporalmente entre threads verticales (V-Threads) que exponen paralelismo solapando accesos a memoria y latencias de comunicación. En esta arquitectura, el espacio de direcciones es uniforme, lo que simplifica la comunicación, mientras que la información en caché, puede migrar entre threads a fin de exponer la localidad de los datos. La solución planteada en este trabajo implica un replanteo completo de la arquitectura de un procesador, además de la necesidad fundamental de soporte de compilación, limitando entonces el alcance de la propuesta. El trabajo de esta tesis se diferencia en plantear una propuesta que pueda ser aplicable a otras arquitecturas a modo de extensión. Sin necesidad de diseñar una arquitectura completa.

Además de la microarquitectura, un aspecto que comienza a jugar un rol fundamental es el consumo energético. En este sentido, el trabajo de Maro et al. [30] propone un procesador capaz de examinar el uso de los recursos y deshabilitarlos dinámicamente a fin de reducir el consumo de energía. Seleccionar cuidadosamente los periodos en los que deshabilitar recursos permitiría reducir el consumo sin impactar negativamente en el rendimiento. Sus resultados muestran una reducción promedio del consumo del $12\,\%$, degradando el rendimiento solo en $2{,}5\,\%$.

El enfoque de reducción de consumo para Kumar et al. [23] es diferente. Proponen un mecanismo para reducir el consumo de energía en arquitecturas heterogéneas, pero respetando el mismo ISA. El diseño propuesto incorpora cores de diferentes prestaciones, tanto en consumo como en rendimiento. Al ejecutar una aplicación, el sistema decide dinámicamente cuál procesador es más apropiado para reducir el consumo sin perder rendimiento. Su evaluación muestra importantes beneficios en reducción de energía gracias a la diversidad de cores. Este tipo de arquitecturas pueden adaptarse a workloads de formas que un procesador homogéneo no puede.

Existen múltiples trabajos específicamente en reducción del consumo de energía, enfocados tanto a mejoras en hardware como en soporte de software. En esta tesis el objetivo es lograr mejorar el paralelismo para casos en que actualmente las soluciones existentes resultan ineficientes. En este sentido, la reducción del consumo de energía juega un rol fundamental. En los trabajos mencionados, se pone de manifiesto el impacto de las innovaciones en hardware en el consumo energético.

Entrando nuevamente en el marco de las arquitecturas, Mutlu et al. [32] plantean que los procesadores pueden tolerar grandes latencias en las operaciones gracias a la ejecución fuera de orden. Pero a medida que estas latencias aumentan, el tamaño de la ventana de instrucciones debe incrementarse también. El trabajo presenta una forma de sortear el bloqueo de la ventana de instrucciones por la espera de datos de la memoria principal. Permite ejecutar las siguientes instrucciones realizando independientemente las cargas y almacenamientos sobre los primeros y segundos niveles de memoria caché. Todos estos eventos son realizados en paralelo junto con el *miss* a memoria principal por el que originalmente se esperaba. Sus experimentos muestran mejores resultados en comparación con un aumento sustancial del tamaño de la ventana de instrucciones. Este tipo de resultados deja en evidencia las limitaciones en la búsqueda de mejorar el ILP.

Otro enfoque en arquitecturas es planteado por Sankaralingam et al. [41] que describen una nueva arquitectura de procesadores que puede ser configurada para aprovechar distintos tipos de paralelismo. La arquitectura permite que un conjunto de unidades de procesamiento y memoria sean configuradas para múltiples tipos de aplicaciones. El objetivo final es alcanzar el rendimiento y eficiencia que logran sistemas de propósito específico. El desafío más grande de este tipo de sistemas es el diseño de las interfaces entre el software y el hardware configurable que permita determinar cuándo se debe modificar la configuración del sistema en sí mismo.

Siguiendo las ideas de este trabajo, Zhong et al. [51] proponen una nueva arquitectura para aprovechar diferentes tipos de paralelismo: ILP, Fine-Grain Thread Level Parallelism (Fine-Grain-TLP) and Loop-level Parallelism. Considera que las aplicaciones de propósito general no proveen muchas oportunidades para identificar threads; el uso frecuente de punteros, estructuras de datos recursivas, saltos condicionales, funciones muy pequeñas y ciclos cortos, son algunas de las fuentes del problema. La arquitectura propuesta provee a los cores dos modos de operación, modo acoplado (*coupled*) y desacoplado (*decoupled*). En el modo acoplado, los cores ejecutan un flujo de múltiples instrucciones paso a paso colectivamente como si fuera una VLIW (Very Long Instruction Word). Esto permite tener una comunicación muy rápida entre cores y exponer ILP. En cambio, en modo desacoplado, los cores ejecutan un conjunto de threads muy livianos que fueron orquestados en tiempo de compilación. Este modo ofrece rápida sincronización y la posibilidad de superponer la ejecución entre iteraciones o fallos de caché con cómputo. En el trabajo analizan diferentes benchmarks e identifican secciones de código donde se pueden aprovechar distintos tipos de paralelismo. La tarea de compilación fue realizada usando la infraestructura de compilación Trimaran (Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/). Este trabajo muestra que existen muchas oportunidades para exponer paralelismo de grano fino en aplicaciones.

Otro trabajo que plantea la reconfiguración de recursos es Ipek et al. [20]. Fundamentalmente proponen un arquitectura basada en múltiples cores que dinámicamente puede combinarse en uno más grande,

compartiendo así sus recursos. Ya sea utilizando los recursos independientes, o combinados, el modelo de ejecución se mantiene a lo largo de las distintas configuraciones posibles, sin requerir esfuerzo adicional de programación o un compilador especializado.

Un enfoque de alto nivel a esta problemática es el planteado por Nesbit et al. [33]. Consideran que los mecanismos y políticas de administración de recursos actuales no resultarán adecuados para futuros sistemas multicore. Por su parte, las políticas de administración de recursos proveen soluciones que, por flexibilidad, son implementadas en software. Mientras que los mecanismos son las primitivas utilizadas para construir estas políticas. Como las primitivas son universales, éstas pueden ser implementadas tanto en software como en hardware. Los autores, proponen las *virtual private machine* (VPM) como medio para la administración de recursos. A diferencia de las máquinas virtuales clásicas, que virtualizan las funcionalidades de un sistema, éstas se ocuparán de virtualizar las características de organización del sistema, afectando entonces el rendimiento y consumo. Las VPM proporcionan una interfaz conceptual entre las políticas y los mecanismos. Las políticas traducen la aplicación y objetivos del sistema en asignaciones de recursos. Los mecanismos VPM multiplexan, arbitran o distribuyen los recursos de hardware de forma segura para satisfacer las asignaciones. Este enfoque sobre la importancia de utilizar adecuadamente los recursos de hardware es parte de los fundamentos detrás de los objetivos de esta tesis.

Madriles et al. [28] afirman que los sistemas multi-core basados en núcleos simples pueden ser muy efectivos para exponer TLP, pero su rendimiento se ve comprometido al momento de ejecutar aplicaciones secuenciales. Por el contrario, las arquitecturas multi-core basadas en núcleos grandes suelen tener poca cantidad de núcleos, debido principalmente a restricciones de área y potencia, limitando entonces sus posibilidades de exponer TLP. Proponen un diseño para que aplicaciones desarrolladas para ejecutar en un solo thread aprovechen múltiples cores, a través de threads especulativos soportados por hardware. Esta técnica proporciona un conjunto de mecanismos de hardware para soportar la ejecución de threads generados en tiempo de compilación. Los threads provienen de una descomposición especulativa de la aplicación original y son ejecutados dentro de un sistema multi-core que soporta mecanismos para: ejecutar múltiples versiones de threads, detectar violaciones entre threads, reconstruir el orden secuencial original y, crear checkpoints para recuperar de la ejecución especulativa. El compilador es un componente clave, es el responsable de distribuir instrucciones a los cores, mientras que el hardware incluye componentes especiales para soportar este modelo de ejecución. Si bien el potencial del trabajo se encuentra en la forma de construir threads especulativos por parte del compilador, se deja en claro el potencial del uso de núcleos grandes y pequeños para soportar múltiples threads minimizando las restricciones de área y potencia.

Un enfoque diferente para que aplicaciones mayoritariamente seriales puedan aprovechar múltiples cores es el planteado por Pricopi et al. [36, 37]. En su trabajo presentan una arquitectura reconfigurable que inicialmente opera como un multiprocesador homogéneo, pero que tiene la capacidad de combinarse para formar múltiples procesadores heterogéneos bajo directivas de software. El modo combinado, el modelo de ejecución es similar a un pool de threads. Las unidades de ejecución de los cores serán bloques básicos (secuencias de instrucciones con un punto de entrada y un punto de salida). Estos bloques son creados por el compilador e identificados por el procesador por medio de instrucciones especiales.

Hasta aquí, todos los trabajos en arquitecturas de procesadores dejan claro que para maximizar el rendimiento y minimizar el consumo se deben considerar los recursos de hardware que específicamente las aplicaciones requieren. Sobre esta idea, el sistema operativo juega un rol central en la asignación y administración de recursos.

En el trabajo de Price y Lowenthal [35] se estudian diferentes paquetes para administrar threads (Cilk [6, 12], Filaments [11, 27], Lazy Threads [15] y StackThreads/MP [46, 45]), que solucionan, de diversas formas, el problema de crear y gestionar un gran número de threads.

Cada paquete es comparado en base al nivel de soporte de un modelo general de threads, como también su rendimiento en programas que usan paralelismo de grano fino (*fine-grain parallelism*). El estudio encontró que los paquetes soportan un grado medio de paralelismo eficientemente, pero que no siempre

soportan paralelismo de grano fino. Bajo determinadas condiciones, se pueden llegar a comportar correctamente, pero esto depende tanto del paquete como de la habilidad del programador. A pesar de ser un estudio realizado sobre versiones actualmente obsoletas, las conclusiones sobre el potencial de soportar eficientemente paralelismo de grano fino continúan siendo relevantes.

Entendiendo las limitaciones de los frameworks de paralelización para la administración de threads, Kumar [24] et al. plantean una solución radical. Para obtener TLP en aplicaciones, el enfoque estándar es descomponer el programa en tareas y ejecutarlas por medio de una capa de software denominada scheduler. El scheduler de tareas por software puede proporcionar un buen rendimiento paralelo, siempre y cuando, las tareas sean *grandes* en comparación con el overhead generado por el software. Carbon es una propuesta con el objetivo de obtener escalabilidad en el rendimiento por ejecutar en paralelo tareas pequeñas. Carbon consiste en hardware dedicado a acelerar dinámicamente el scheduling de tareas. En el trabajo se compara la solución de Carbon con schedulers por software, como también con una planificación ideal como control. Los resultados dan cuenta del impacto del scheduler en el rendimiento de aplicaciones paradigmáticas.

Por su parte, Sanchez et al. [40] presentan un enfoque diferente combinando hardware-software para construir schedulers de grano fino, dando la flexibilidad de los scheduler por software, mientras se logra la velocidad y escalabilidad de las soluciones por hardware. Para lograrlo proponen una extensión a la arquitectura del procesador que provee un mecanismo para enviar mensajes directos entre cores de forma asincrónica y sin necesidad de pasar a través de la jerarquía de memoria. Este mecanismo es suficiente para implementar schedulers, y coordinar eficientemente la información de tareas. Tanto en escalabilidad como rendimiento, los resultados muestran un mejor comportamiento que las soluciones de solo hardware, dejando en evidencia que fijar los algoritmos de scheduling no resulta una buena estrategia aplicable a todos los casos, sino que se requiere mayor grado de flexibilidad.

Siguiendo el objetivo de mejorar el scheduling de tareas, Gioiosa et al. [14] enuncian que el diseño de los sistemas operativos en capas independientes, provee portabilidad y transparencia pero que no necesariamente resulta una solución eficiente para sistemas de High Performance Computing (HPC). Presentan sus experiencias en el diseño de políticas de scheduling, haciendo foco en mejorar el desempeño de aplicaciones de HPC por medio de reducir el ruido de administración generado por la intervención del sistema operativo. Realizan experimentos sobre un kernel basado en Linux que reduce el overhead y la variación en el rendimiento de threads, mejorando así la escalabilidad y rendimiento general de aplicaciones HPC. En términos generales, su kernel realiza un balance de tareas de HPC cuando se involucra el llamado a *fork*, impidiendo que CFS (Completely Fair Scheduler) realice su propio balance de tareas. Los resultados obtenidos muestran el impacto de la variación de rendimiento sobre aplicaciones que ejecutan múltiples threads, dejando claro que, al momento de ejecutar múltiples threads, resulta fundamental que éstos estén sincronizados y que su rendimiento sea similar, caso contrario el rendimiento obtenido será compatible con el thread que peor rendimiento presente.

Esta revisión muestra, por un lado, el rico camino que ha tenido el tema de desarrollo de nuevas técnicas de soporte al paralelismo tanto a nivel software como en hardware. Por otro lado, también deja claro que aún hay espacio y necesidad para el desarollo de técnicas que integren ambos aspectos y permitan generar soluciones eficientes para sustentar el incremento de los recursos computacionales que se avisoran estarán presentes en los procesadores.

1.3 Investigación en arquitectura de procesadores

La arquitectura de procesadores es un área de investigación que se nutre notablemente de las herramientas de simulación. Los costos del desarrollo de sistemas reales obligan a realizar cientos de simulaciones antes de acceder a implementar un procesador real. Incluso, el desarrollo de un sistema real tiene sus limi-

taciones si éste no se realiza a gran escala, por ejemplo en términos de nivel de integración o superficie de silicio utilizada. Una herramienta de bajo costo y que resulta en un paso intermedio entre la simulación y el sistema físico final corresponde con la implementación en FPGAs. Éstos permiten hacer comprobaciones de funcionamiento e implementación, pero se ven limitados en mediciones de tiempos o energía. Las herramientas de simulación por otro lado, permiten comprobar ciertos funcionamientos esperados, además de dar la posibilidad de estimar rendimiento, tanto en términos de tiempos como de energía.

En la búsqueda de una nueva arquitectura o el desarrollo de un sistema nuevo, normalmente se utiliza una arquitectura existente y sobre ella se proponen modificaciones, ya sea como protocolo experimental o como referencia. Estas modificaciones o nuevos desarrollos, se modelan basándose en modelos existentes y buscando alcanzar el nivel de detalle adecuado para los experimentos que se propongan. El nivel de detalle del modelo de simulación es fundamental para alcanzar resultados realistas, un modelo excesivamente detallado demorará mucho tiempo en ejecutar, generando resultados muy limitados, mientras que un modelo poco detallado generará resultados deficientes. Encontrar un compromiso entre el nivel de detalle del modelo y el tiempo necesario para ejecutar experimentos es parte del trabajo a realizar en esta área.

Luego, a través de simulaciones, ya sea de un conjunto de aplicaciones o de experimentos de análisis de límites, se busca comparar el comportamiento de la arquitectura propuesta con respecto a una arquitectura de referencia. Estas comparaciones se pueden dar en varios aspectos, tanto en características internas como uso de recursos, o aprovechamiento de unidades funcionales, como también en términos generales de rendimiento.

Dentro del aspecto de energía, una de las formas de estimar el consumo, se basa en utilizar modelos genéricos de circuitos para modelar cada una de las unidades del procesador, desde memorias caché hasta etapas del pipeline. Luego, con estos modelos se genera una posible implementación a nivel de circuitos sobre la que estimar tanto consumo, como área y tiempo.

Capítulo 2

Metodos

A lo largo de esta tesis se utilizaron diferentes herramientas, tanto para desarrollo y visualización, como para simulación. Se contó con la colección de compiladores `GNU`, junto con sus herramientas de debug. Para visualización, en general, se utilizó `Matplotlib` y scripting en lenguaje `Python`. La simulación se realizó con `gem5`, mientras que la estimación de consumo con `McPAT`. En este capítulo se detallarán, principalmente, las herramientas utilizadas para la simulación de la plataforma y la estimación de consumo energético.

2.1 Simulador `gem5`

El simulador `gem5` [4] es una plataforma modular para la investigación en arquitectura de procesadores y sistemas de cómputo. Permite construir simulaciones tanto a nivel de sistema, como a nivel de microarquitectura y emular el comportamiento de un sistema completo con un alto nivel de detalle, pero sin emular la implementación a nivel de circuitos electrónicos. La figura 2.1 ilustra la relación entre las clases que implementan los diferentes módulos simulados por `gem5`. Se puede ver que la implementación de `CPU` admite diferentes interfaces para acceder a su estado y, por otro lado, la conexión de la memoria se da por medio de puertos que pueden ser conectados formando distintas jerarquías.

`Gem5` es un simulador de eventos discretos e implementa diferentes módulos con distintos niveles de detalle. La conexión entre los módulos se resuelve mediante una intefaz que permite interconectarlos. El procesamiento de eventos se realiza independientemente de la implementación de la lógica de los módulos, es decir, de los elementos de la simulación. Gracias a esto, es posible implementar diferentes tipos de pipeline con diferentes tipos de unidades funcionales bajo el mismo motor de simulación.

El procesador puede ser simulado usando diferentes modelos, en `gem5` se denomina *modelo de CPU* a cada uno de ellos. Se proveen distintos modelos de simulación: modelo simple, temporizado y detallado, este último, ya sea operando como un procesador in-order o como un procesador out-of-order. Todos los modelos de simulación utilizan una descripción común del ISA (*Instrucction Set Architecture*), lo que permite a su vez, tener la capacidad de simular distintos tipos de procesadores. Éstos pueden ser arquitecturas tipo *Alpha*, *ARM*, *SPARC*, *x86*, entre otros.

El sistema simulado por `gem5` provee la capacidad de simular accesos a memoria, incluyendo modelos de cachés, *crossbars*, *snoop filters* y un preciso modelo del funcionamiento de un controlador de DRAM. Todos estos componentes pueden ser organizados para construir cualquier tipo de jerarquía de memoria.

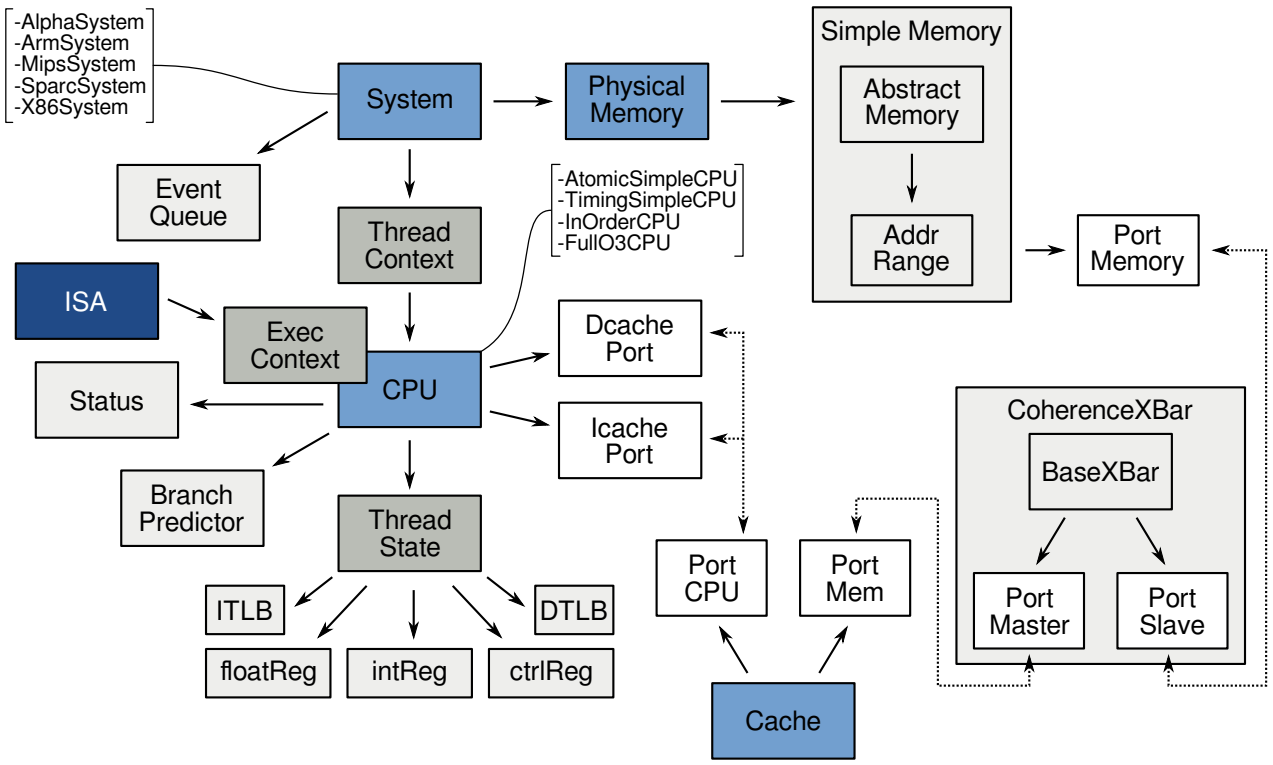


Figura 2.1: Clases y organización básica de la implementación del simulador gem5.

2.1.1 Uso

Para ejecutar un programa sobre el simulador, se deben llevar a cabo una serie de pasos que dependen del tipo de experimento que se busque construir. Gem5 es una pieza de software altamente configurable y, por esta razón, construir un entorno de ejecución resulta laborioso. La construcción de dicho entorno consiste en instanciar todas las piezas de software que hacen al funcionamiento de un sistema.

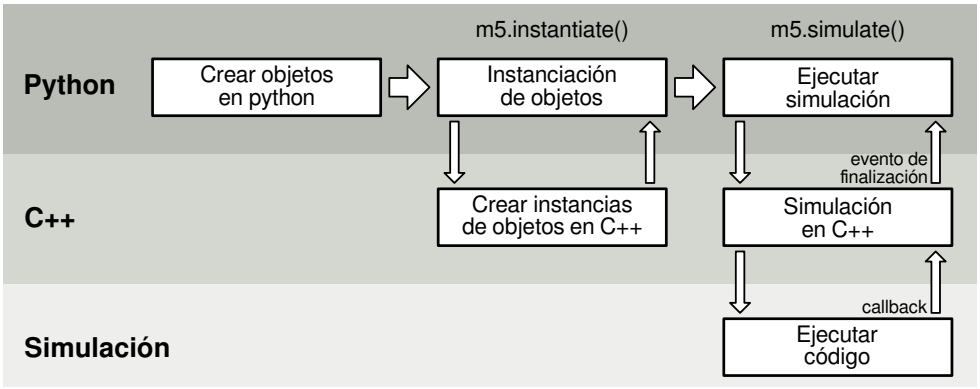


Figura 2.2: Pasos para la construcción de una simulación y las distintas herramientas que intervienen en el proceso.

Considerando esta dificultad, los desarrolladores de gem5 decidieron utilizar el lenguaje Python para la configuración del entorno de ejecución. Por medio de un *script* se construyen todas las instancias de software, así como también se configuran y conectan entre sí.

La figura 2.2 ilustra esta serie de pasos: cuando comienza la ejecución de un experimento en gem5, éste ejecuta un script en Python que construye una serie de objetos Python que representan la configuración de todo el sistema a ejecutar. Una vez creado todo el sistema y comprobado que es posible generarlo,

se instancian todas las estructuras y objetos para construir el entorno de ejecución. Esta etapa se resuelve mediante software implementado en C++. Una vez que el total del entorno es construido, se procede a comenzar la ejecución.

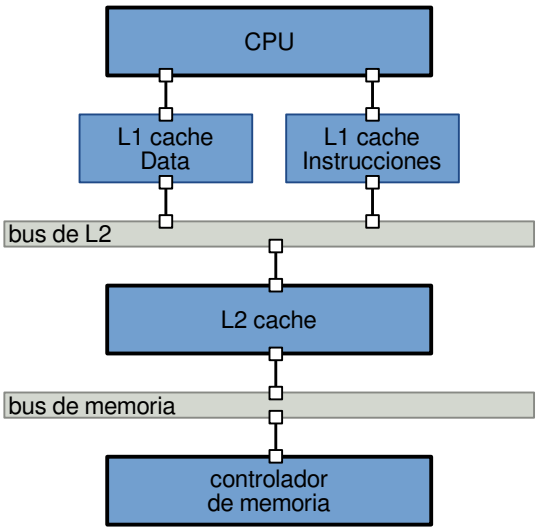


Figura 2.3: Instancias básicas de software creadas por una simulación de gem5. La conexión entre cada una se modela a través

Todo este proceso es guiado por el mismo script en Python, que es pasado como parámetro a gem5. Si bien el usuario puede crear sus propios scritps, gem5 provee scritps generales, que pueden ser configurados en función del experimento a realizar. La figura 2.3 muestra las instancias básicas de alto nivel, construidas para una simulación en gem5. Cada una se conecta mediante puertos, éstos funcionan bajo un modelo maestro-esclavo y, de esta forma, se logra la libertad de construir todo tipo de simulaciones conectando componentes.

2.1.2 Modos de operación

Gem5 provee dos modos de funcionamiento de alto nivel, FS (Full System) y SE (System Emulation). Ambos modos pueden utilizar cualquier configuración del entorno de simulación. Gem5 provee scritps que permiten configurar un sistema de forma genérica, ya sea tanto para FS como para SE.

- Full System (FS)
  Simula un sistema completo, se inicializa el sistema desde una imagen de disco que carga un sistema operativo. En este entorno es posible ejecutar cualquier tipo de aplicación, ya que el sistema operativo será el encargado de ejecutarla, como si se tratara de un sistema real. Las aplicaciones pueden hacer uso de todas las funcionalidades que el sistema operativo provea, desde cargar bibliotecas dinámicas hasta comunicarse con dispositivos externos como placas de red. En este entorno se puede medir la interacción de las aplicaciones con el sistema operativo, ya que la traza generada por gem5 incluye todas las instrucciones ejecutadas por el procesador. Sin embargo, tanto el sistema operativo, como el código que se esté estudiando, son ejecutados de la misma forma. Por esta razón, no es posible tener control de que se está ejecutando en cada momento. La aplicación de estudio está *a merced* de la lógica de scheduling del sistema operativo.

- System Emulation (SE)
  En este entorno nuestra aplicación es ejecutada aisladamente. Se tiene control completo del procesador y todos los llamados al sistema operativo son emulados. Cada vez que se requiera de un servicio del sistema operativo, éste es resuelto por código fuera de la simulación; la demora y las instrucciones ejecutadas para resolver el servicio no forman parte de la simulación. No todos los llamados al

sistema operativo son emulados, la implementación del simulador está limitada a soportar solo un subconjunto. Por esta razón, no es posible ejecutar cualquier tipo de código bajo este modo. Además, la aplicación debe estar compilada de forma estática. Todas las bibliotecas de funciones que la aplicación requiera deben formar parte de la aplicación.

2.1.3 Modelos de CPU

En gem5, el modelo de CPU refiere a la forma en que se simula la ejecución de instrucciones. Existen distintos modelos de CPU que se diferencian en el nivel de detalle con el que son ejecutadas las instrucciones y a la forma en que éstas son resueltas. Cualquiera sea el modelo de CPU, gem5 provee una API que permite a cualquier ISA utilizar cualquier modelo.

Los modelos de CPU que provee gem5 son los siguientes:

- atomic
  Es el modelo más simple de gem5, implementa accesos a memoria de forma atómica. Estima los tiempos de accesos a memoria caché a partir de la estimación de la latencia de las operaciones. El modelo implementa funciones para leer y escribir en memoria, además define las funciones de "tick" que se ejecutan en cada ciclo de CPU.

- timing
  Este modelo, a diferencia del atómico, utiliza accesos a memoria temporizados. Es decir, se considera el tiempo requerido para mover datos entre la memoria y el procesador, ya sean parámetros de instrucciones o resultados de éstas. Las instrucciones son resueltas en dos etapas, por un lado se trae la instrucción desde memoria (*fetch*) y por otro se la ejecuta(*execute*). El fetch es completado una vez que la memoria responde al pedido, en tanto ese tiempo el sistema espera a que los datos lleguen desde la memoria. El execute es ejecutado de forma atómica siempre que no se requieran parámetros desde memoria. En ese caso, la instrucción esperará a que lleguen desde memoria los datos para completar la ejecución.

- detailed
  El modelo es implementado por la clase `O3CPU`. Corresponde a un procesador out-of-order basado en la máquina de procesamiento fuera de orden del procesador `Alpha 21264`. Implementa en este sentido las etapas: Fetch, Decode, Rename, Issue/Execute/Writeback y Commit. Además, considera los siguientes recursos en el pipeline: Branch predictor, Reorder buffer, Instruction queue, Load-store queue, Functional units y Memory dependence prediction using store sets. Este modelo enfoca sus esfuerzos en construir un modelo de tiempos muy preciso, para esto las instrucciones son ejecutadas en el momento en que deben ser ejecutadas en el pipeline. Esto diferencia gem5 de otro tipo de simuladores, donde la ejecución de las instrucciones se realiza al principio del ciclo y luego las unidades se encargan de imitar la ejecución sin hacerla realmente.

- arm_detailed
  Este modelo es una derivación del modelo out-of-order (detailed), es decir, no es una nueva clase de gem5. Instancia todas las unidades de modelo utilizando parámetros compatibles con un `ARM v7a`. Esta configuración es parte de los trabajos de Endo et. al [7].

- minor
  Implementa un modelo de procesador in-order con un pipieline fijo pero configurable. El comportamiento de este modelo simula un procesador estrictamente in-order, provee una interfaz que permite configurar el modelo para simular un procesador particular. En la implementación actual de gem5, las características son las de un ARM in-order.

2.1.4 Construcción de experimentos en modo Full System

Para ejecutar experimentos en FS se requiere cargar inicialmente un sistema operativo. La carga de este sistema puede demorar mucho tiempo dependiendo del detalle del modelo de procesador instanciado. Si cada vez que se ejecuta un experimento en FS se debe cargar todo el sistema, entonces se vuelve impracticable lanzar varios experimentos.

Para solucionar este problema, gem5 cuenta con un sistema de checkpointing. Éste permite generar un checkpoint de toda la ejecución de un sistema y almacenar los datos necesarios para reiniciarla posteriormente. Entonces, es posible iniciar el sistema una sola vez y luego tener almacenado un checkpoint correspondiente a la carga inicial, evitando tener que volver a realizar esa carga cada vez que se busque ejecutar un experimento.

Simular usando un modelo detallado demora mucho tiempo, luego, iniciar un sistema operativo sobre un modelo de procesador detallado se vuelve impracticable. Por esta razón, gem5 provee una funcionalidad que permite recuperar un checkpoint sobre cualquier modelo de procesador. Es decir, cuando se levanta un checkpoint, éste puede ser ejecutado en un modelo con mayor nivel de detalle. De esta manera, se logra que el proceso de inicialización de un sistema demore poco tiempo, ya que se utilizaría un modelo de procesador con poco detalle. Luego, se cargaría un modelo más detallado al reiniciar el checkpoint, que corresponde al momento de la simulación de interés.

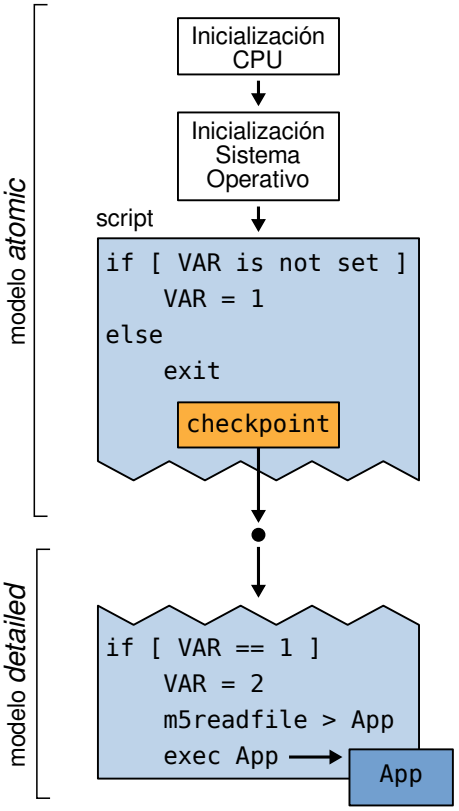


Figura 2.4: Funcionamiento del script para generar checkpoints en FS.

El otro problema a solucionar es que una vez creado el checkpoint, éste funciona como una *foto* del sistema completo. Entonces, si se estaba ejecutando un determinado programa, éste seguirá ejecutándose. En este sentido, no sería posible ejecutar distintos programas en un sistema sobre el que se generó un checkpoint. Para solucionar esto, gem5 provee una funcionalidad de bajo nivel que resuelve esta situación. Cuando un sistema es inicializado y, sobre éste se ejecuta un programa, el mismo no puede cambiar a menos que se cargue un nuevo programa. El problema reside en cómo cargar el nuevo programa desde el simulador, sin afectar el funcionamiento del sistema operativo.

Sería posible cargar el nuevo programa desde una imagen de disco o incluso desde la red, pero para esto se debe tener soporte para estas alternativas desde el sistema operativo simulado. En el caso que no se cuente con ningún tipo de soporte, sería imposible cargar un nuevo programa. La solución que plantean los desarrolladores de gem5 es utilizar una instrucción especial que permita traer información desde el exterior a la simulación.

Esta funcionalidad puede ser utilizada desde un script que provee el simulador (`/config/boot/hack_back_ckpt.rcS`). Una vez iniciado el sistema, se ejecuta el script. Su funcionamiento, ilustrado en la figura 2.4, consiste en crear una variable global para indicar si es la primera o la segunda vez que es ejecutado y luego construir un checkpoint. Al iniciar nuevamente el checkpoint, el script reconoce que está siendo ejecutado por segunda vez y utiliza un programa, que forma parte de los utilitarios (/sbin/m5 readfile) de gem5, para cargar un ejecutable externo y correrlo dentro del sistema.

2.1.5 Mediciones

Las herramientas de simulación, a diferencia de ejecutar código en un sistema nativo, permiten tener control de todo el proceso ejecución sin tener en consideración el tiempo real. Permiten, entonces, capturar marcas de tiempo o logs sin alterar el funcionamiento del sistema. Incluso, si la captura de estos logs requiere de tiempo de cómputo, este tiempo no afectaría la simulación final.

La herramienta `gem5` opera como un simulador de eventos discretos, una de las funcionalidades que provee es una interfaz para crear y construir todo tipo de logs de eventos con un amplio nivel de detalle. Los eventos se pueden clasificar en dos grupos, por un lado, los relacionados con la simulación, como por ejemplo, las etapas del pipeline del procesador, el controlador de memoria, dispositivos, etc. Sobre éstos se pueden calcular métricas del funcionamiento del sistema simulado. Por otro lado, existen eventos relacionados con la simulación en sí misma, como el control de la cola de eventos, accesos a unidades, datos de control, etc. Estos últimos permiten realizar un seguimiento de qué está sucediendo en el motor de simulación.

Además de la generación de trazas de ejecución, `gem5` construye dos archivos de datos como resultado de la simulación. Por un lado, estadísticas de la ejecución y por el otro, información de configuración. Las estadísticas almacenan información agregada de la ejecución, como pueden ser, tiempo total de ejecución, cantidad de accesos a caché de nivel 1, cantidad de instrucciones de punto flotante ejecutadas, etc.

El archivo de configuración, en cambio, almacena información de la configuración de cada unidad del sistema, desde la cantidad de entradas en la TLB, hasta detalles de la cantidad de ciclos requeridos para ejecutar una instrucción particular.

La información provista por estos dos archivos se utiliza como entrada para el modelo de energía explicado en la sección 2.2.

2.1.6 Utilitarios e instrucciones de control

`Gem5` provee un conjunto de funcionalidades para acceder desde el entorno simulado al entorno de simulación. Éstas permiten generar volcados de estadísticas, checkpoints, cargar datos al entorno simulado o incluso salir de la simulación. A continuación se listan:

- `exit [delay]`: Detiene la simulación en `delay` *nanoseg*.
- `resetstats [delay [period]]`: Resetea las estadísticas de la simulación en `delay` *nanoseg*.
- `dumpstats [delay [period]]`: Guarda las estadísticas de la simulación en un archivo en `delay` *nanoseg*.
- `dumpresetstats [delay [period]]`: Mismo comportamiento que `dumpstats` y `resetstats`.
- `checkpoint [delay [period]]`: Crea un checkpoint en `delay` *nanoseg*.
- `readfile`: Obtiene el archivo especificado por el parámetro `system.readfile`. Esta funcionalidad es utilizada por scripts para copiar archivos dentro del entorno de simulación.
- `debugbreak`: Llama a `debug_break()` dentro del simulador. Causa una `SIGTRAP` para debugging utilizando GDB.
- `switchcpu`: Genera un evento *exit* de tipo `switch cpu`, permite reiniciar la simulación cambiado el modelo de CPU.

Estas funcionalidades son implementadas en `util/m5/` dentro del código fuente de `gem5`. En FS es posible acceder a las mismas mediante el utilitario `m5`, éste forma parte de las aplicaciones que deben ser agregadas a la imagen de disco de la simulación. Por otro lado, en SE se pueden incluir como una biblioteca de funciones dentro de la aplicación a ejecutar.

2.1.7 Compilación y ensamblado

Las simulaciones pueden ser ejecutadas en una arquitectura distinta de la nativa del sistema que se esté utilizando para correr el motor de simulación. Por esta razón es muy importante construir un entorno de *cross-compiling* compatible con el sistema que se quiera estudiar. En este trabajo se utilizó la herramienta `crosstool-ng` (`http://crosstool-ng.github.io/`) que permite construir cualquier entorno de compilación, seleccionando para esto las versiones específicas de todos los aplicativos del sistema, incluyendo la versión del kernel y bibliotecas.

Parte de los experimentos realizados modifican el ISA del procesador, agregando instrucciones que no forman parte del mismo. Las herramientas utilizadas para compilar, ensamblar/desensamblar y realizar debuging, no soportan estas nuevas instrucciones. Por esta razón se debe tener especial cuidado, tanto en la forma en que las nuevas instrucciones son declaradas, como en la presentación del código por parte de las herramientas de debugging.

2.2 Simulador McPAT

La herramienta McPAT [26] (Multicore Power, Area, and Timing), es un framework de modelado para el cálculo de energía, área y tiempo, en arquitecturas de procesadores multicore, multithreaded y manycore. Permite explorar el espacio de diseño de procesadores para diferentes configuraciones, evaluando simultánea y consistentemente características de: consumo de energía, área necesaria y frecuencia de operación. Todo esto para tecnologías desde 90 nm a 22 nm e, incluso, de menor tamaño. Incluye modelos completos de componentes de procesadores, como núcleos in-order y out-of-order, networks-on-chip, shared cachés, y controladores de memoria. El modelo soporta el cálculo de consumo para cada componente independientemente siguiendo el ITRS (*International Technology Roadmap for Semiconductors*).

A continuación se resume parte del reporte técnico [25] de McPAT.

2.2.1 Descripción y funcionamiento

McPAT utiliza una interfaz basada en XML que permite especificar una gran cantidad de parámetros de bajo nivel. Esta interfaz incluye tanto parámetros de configuración de la microarquitectura estática, como estadísticas de la actividad dinámica generadas por algún simulador de arquitecturas (por ejemplo gem5).

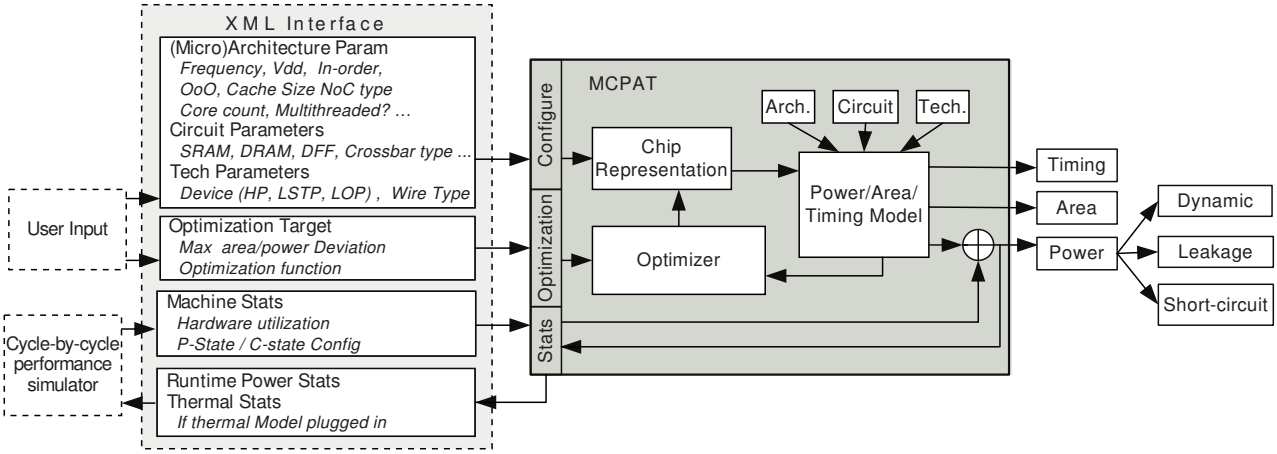


Figura 2.5: Diagrama en bloques de los componentes principales de McPAT y sus datos de entrada. Figura reproducida de [25].

McPAT puede enviar información sobre el estado térmico del sistema al simulador, permitiendo a este último actuar sobre cambios en la temperatura o energía. Dado que McPAT proporciona modelos jerárquicos completos desde la arquitectura hasta el nivel de la tecnología de integración utilizada, la interfaz XML también debe proveer parámetros como el tipo de implementación de los circuitos y los parámetros tecnológicos específicos del procesador a utilizar. Ejemplos de estos parámetros son los tipos de arrays, crossbar y de tecnología CMOS, con sus voltajes y dispositivos asociados.

La figura 2.5 muestra un diagrama de bloques de McPAT, sus componentes principales son: (1) modelos jerárquicos de consumo, área y tiempo, (2) optimizador para determinar la implementación a nivel de circuitos, y (3) representación interna de componentes, entrada del análisis de potencia, área y tiempo. La mayor parte de los parámetros de la representación interna, como tamaño de la memoria caché o la cantidad de unidades funcionales de cada core, son configurados directamente por el usuario. La organización jerárquica de McPAT permite modelar estructuras de bajo nivel, considerando las condiciones impuestas por tecnología de fabricación, sin que el usuario intervenga activamente en este modelado. Permitiendo entonces, centrar esfuerzos en la configuración arquitectónica de alto nivel, mientras que el optimizador determina que parámetros resultan óptimos para la representación interna a nivel de circuitos.

El optimizador genera una representación final del circuito, que es usada para calcular el área, la temporización y el consumo pico final. El consumo pico de cada unidad y las estadísticas de utilización (factor de actividad) son utilizadas para calcular el consumo final del experimento.

McPAT funciona en dos faces: Inicialización y Cálculo.

Durante la fase de inicialización son utilizadas las especificaciones estáticas, que corresponden a parámetros sobre los tres niveles del modelo: arquitectura, circuitos y tecnología. En el nivel de arquitectura los parámetros son similares a los utilizados para los simuladores como gem5, incluyendo la cantidad de cores y routers, los parámetros de la caché compartida, el core issue width, características para el motor de ejecución out-of-order o in-order, cantidad de hardware threads, entre otros. A nivel de circuitos, los parámetros especifican detalles de implementación, por ejemplo es posible precisar que un array esté basado en flip-flop o en celdas SRAM. Por último, a nivel de tecnología, los parámetros especifican el tipo de dispositivo en términos de consumo (high performance, low standby power, low operating power) y características de interconexión. Los parámetros estáticos también incluyen opciones de optimización, como la máxima área y consumo. Terminada la etapa de inicialización, McPAT generará un modelo de la representación interna del procesador optimizando las características solicitadas por el usuario.

La fase de cálculo, por su parte, es la encargada de generar las métricas de consumo dinámico. Utilizando las estadísticas generadas por el simulador se calcula el factor de actividad (*ActivityFactor*) para cada componente de forma individual. El *ActivityFactor* está dado por la siguiente expresión:

$$ActivityFactor = \frac{AccessCount \cdot \left(\left(\frac{\sum_{i=1}^{n} HammingDistance)}{n} \right) \right)}{n} \tag{2.1}$$

donde n corresponde a la cantidad de ciclos del periodo simulado, *AccessCount* es el número de accesos al componente durante el mismo periodo, y *HammingDistance* es la cantidad total de bits modificados entre dos accesos consecutivos. Cuando el simulador no pueda calcular *HammingDistance*, McPAT supone que todos los bits fueron modificados.

Si durante la simulación se utiliza McPAT para el cálculo de consumo por cada ciclo de reloj, se obtendrá un perfil de consumo en el tiempo, éste resulta útil cuando se busca estudiar picos en el consumo. En cambio, si se utiliza McPAT luego de una simulación completa, se obtendrá el perfil promedio de consumo.

McPAT es ejecutado independientemente del simulador utilizado. Luego,las estadísticas generadas durante la simulación son utilizadas por McPAT para construir el perfil de consumo. Esta tarea, demora poco tiempo, por lo que el impacto en el tiempo total de simulación por utilizar McPAT en cada ciclo de reloj es

mínimo. Se debe considerar que la etapa de inicialización demora un tiempo considerable, si el espacio de optimización es muy grande. Sin embargo, esta etapa solamente será ejecutada al comienzo de la simulación. Es decir, que afectará el tiempo de simulación como una constante.

Para más información acerca del modelo de consumo (*Power Modeling*), de tiempos (*Timing Modeling*) y área (*Area Modeling*) se recomienda leer al reporte técnico [25]. Éste detalla la implementación de los diferentes modelos, principalmente las decisiones de diseño detrás del modelo de arquitectura y circuitos.

2.2.2 Interacción entre gem5 y McPAT

La herramienta McPAT requiere como entrada un archivo en formato XML que define tanto las características del sistema como datos estadísticos de uso. Con esta información, la herramienta genera un modelo de consumo y calcula una estimación de la energía consumida por cada parte del sistema.

Para calcular el consumo de un determinado experimento realizado con gem5, se extrae la información estadística de la simulación y se usan estos datos como entrada de McPAT. El formato aceptado por este último difiere del provisto por gem5, por esta razón se deben transformar los datos al formato aceptado por McPAT. La interacción entre estos dos programas se muestra en la figura 2.6.

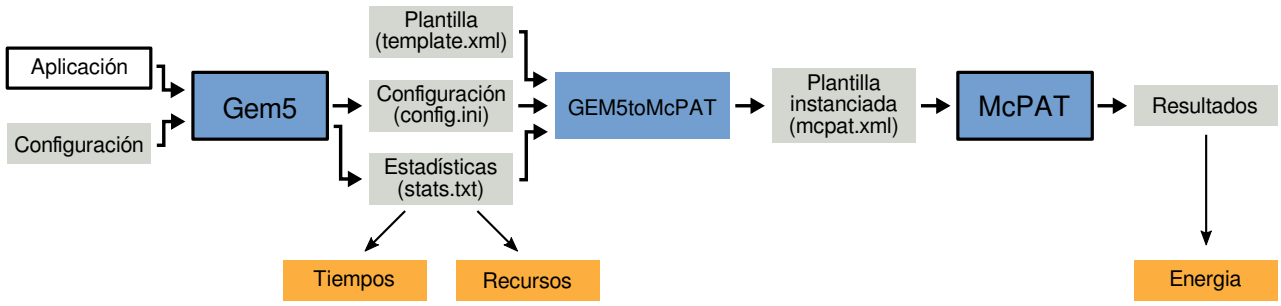


Figura 2.6: Interacción entre gem5 y McPAT

Esta transformación consiste en la utilización de un archivo `template` que respeta el formato de McPAT con todos sus campos, pero haciendo referencia a la información contenida en los archivos de configuración y estadísticas de gem5. Estos últimos corresponden a una lista de campos y datos numéricos para cada uno.

Por ejemplo, en el archivo `template` es posible encontrar las siguientes líneas:

```
...
<param name="clock_rate" value="10**6/config.system.cpu_clk_domain.clock[0]"/>
...
<stat name="total_cycles" value="stats.system.switch_cpus0.numCycles"/>
...
```

En la primer línea de ejemplo, se toma la primera posición del vector `clock` en la clase `cpu_clk_domain` de `system` dentro del archivo de configuración. Por otro lado, en la segunda línea se realiza lo equivalente sobre el archivo de estadísticas. Es posible realizar operaciones aritméticas, ya sea para cambiar el formato de los valores o para realizar operaciones entre éstos. Esta tarea es realizada por la aplicación `GEM5toMcPAT` que, simplemente, consiste en un script que reemplaza textos en el archivo template y evaluá sus operaciones.

Finalmente, los resultados obtenidos por ambas herramientas serán tiempos y recursos utilizados para el caso de gem5, y estadísticas sobre el consumo de energía para McPAT.

2.2.3 Cálculo de consumo de energía

Para estimar el consumo de energía utilizando las herramientas antes mencionadas, se evalúa la siguiente ecuación:

$$\text{Energy Consumption} = \underbrace{(\texttt{Total\_Leakage} + \texttt{Runtime\_Dynamic})}_{\text{Power Consumption}} \cdot \texttt{runtime} \tag{2.2}$$

donde `Total_Leakage` y `Runtime_Dynamic` son métricas reportadas por `McPAT` y `runtime` se toma de las estadísticas de `gem5`. La configuración del procesador utilizada está basada en el trabajo de Li et al. [26], donde los autores modelan el perfil de consumo de un `ARM A9` de 2.0 GHz. La evaluación de energía también es calculada considerando distintas frecuencias base. En estos casos, cada escenario es ejecutado en `gem5` bajo la correspondiente frecuencia y luego es estimado el consumo con `McPAT`.

Capítulo 3

Aportes en metodología experimental

Los experimentos realizados con las herramientas de simulación consisten en ejecutar código de distintas aplicaciones y obtener resultados estadísticos, junto con la captura de métricas de rendimiento y consumo.

Dependiendo de qué experimento se busque realizar, se optará entre los dos modos de operación de gem5. En SE, se debe enlazar el programa de forma estática, es fundamental entonces que todos los llamados al sistema que pueda realizar el código estén emulados por gem5. Caso contrario será necesario remplazar las bibliotecas de funciones por versiones compatibles. Se debe tener en consideración que el código que se esté estudiando no utilice llamados al sistema operativo, ya que estos no serán fielmente simulados.

Los experimentos en modo FS requieren de una planificación especial. En este modo es posible tener todo el soporte que provee un sistema operativo, resultando en el modo ideal para realizar experimentos en los cuales se busque tener en cuenta la operatoria que realiza el sistema. En este trabajo se realizaron experimentos utilizando distintos frameworks de paralelización, éstos requieren de soporte del sistema operativo para gestionar threads. Esta tarea demora tiempo y recursos que deben ser contemplados por las métricas obtenidas de la simulación.

En este capítulo, se detalla una serie de soluciones metodológicas para la puesta en funcionamiento de experimentos. Los métodos presentados no son de uso estándar, ni forman parte de las herramientas de gem5, sino que corresponden con desarrollos necesarios para llevar adelante la evaluación de la propuesta *Mth*. El objetivo detrás de este esfuerzo de desarrollo fue la simplificación de la configuración de experimentos, aceleración de su ejecución, ayuda con la captura de mediciones y evitar errores por procedimientos manuales complejos y repetitivos.

3.1 Método de detección del sistema operativo y apagado de cores

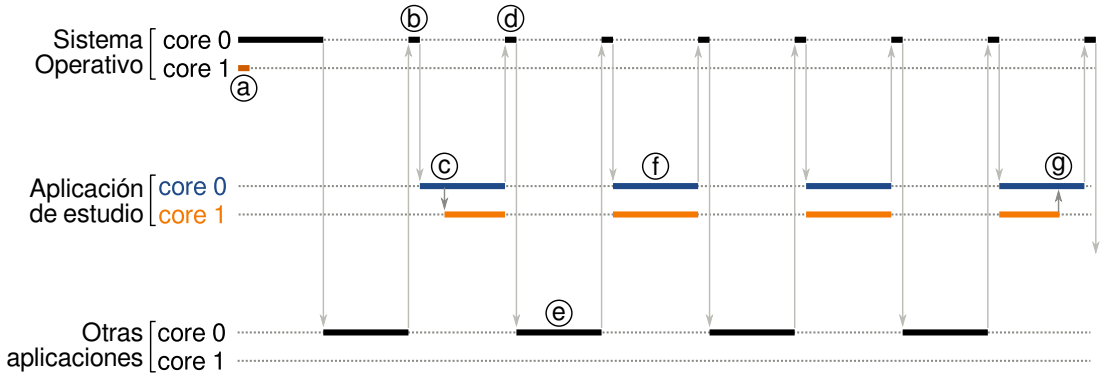
En una simulación en modo FS, los procesadores son administrados por el sistema operativo y el kernel es el encargado de asignarles tareas a los mismos. Para la simulación de la propuesta presentada en este trabajo, se debe contar con la posibilidad de administrar procesadores desde el contexto de una aplicación. Los sistemas operativos estándar no están preparados para este tipo de funcionalidades, en las que se delega el control de un core a una pieza de software de nivel de usuario. La funcionalidad requerida implica que el sistema operativo ceda el control del scheduler a la aplicación, para que ésta no sea desalojada por un periodo de tiempo dado. Por esta razón se optó por implementar el método descripto a continuación. Éste detecta cuando se ejecuta una aplicación y permite que tenga control total de todos los cores.

El primer paso consiste en modificar parte del sistema operativo para que inicie todos los cores, pero una vez en funcionamiento, asigne tareas solamente a uno de estos. En Linux, esta modificación consiste

en alterar la etapa de incialización de los cores para que el sistema detecte la presencia de un único core, denominado *core principal*. Resulta fundamental que el sistema operativo tenga en cuenta todos los cores del sistema, ya que se debe poder resolver las interrupciones que éstos generen. Un core puede generar interrupciones por muchas razones, en nuestro caso, las interrupciones podrían estar dadas por fallas de página, que deben poder ser resueltas desde cualquier core del sistema. Un caso particular de las interrupciones que se pueden generar es un fallo de TLB. En arquitecturas como `ARM`, la carga del TLB se realiza por hardware, pasando inadvertido por la aplicación y resolviéndose a bajo nivel. En arquitecturas como `Alpha`, esta resolución se realiza por software, invocando a una sección de código específico denominado *PALcode* (Privileged Architecture Library code). Esta diferencia implica que la detección de que un core está ejecutando dependerá fuertemente de la arquitectura utilizada. Luego, para implementar esta solución resulta fundamental conocer qué código es ejecutado por cada core. Las posibilidades para identificarlo están limitadas a: código del sistema operativo, código de la aplicación de estudio o código de otras aplicaciones.

El siguiente paso consiste en ejecutar el sistema operativo. Éste ejecutará las aplicaciones según dicten las reglas del scheduler. En algún momento será ejecutada la aplicación bajo estudio. Cuando sea detectada, ésta tendrá control de todos los cores, pudiendo así asignar tareas a los mismos por medio del soporte especial provisto por la propuesta de este trabajo. La tarea bajo estudio está subordinada a las interrupciones del scheduler del sistema operativo. Llegado el momento, la tarea será desalojada del core principal, para ejecutar otras tareas. Cuando esto suceda, el motor de simulación debe detectar este cambio, y detener todos los cores restantes del sistema. Nuevamente, cuando la aplicación vuelva a ser ejecutada en el core principal, el motor de simulación automáticamente iniciará los cores detenidos.

Mediante este método, es posible realizar experimentos que requieran del control de procesadores y del soporte de sistema operativo al mismo tiempo.



a. Inicialización del core 1, no vuelve a ser utilizado por ninguna tarea.
b. Eventualmente la aplicación de estudio comienza a ser ejecutada
c. La aplicación generar un thread en el core 1, sin interacción con el sistema operativo.
d. La aplicación es interrumpida y otras aplicaciones son ejecutadas. El core 1 es interrumpido también.
e. Otras aplicaciones son ejecutadas.
f. El tiempo de procesador es compartido con la aplicación de estudio, que cuando es ejecutada enciende el core 1.
g. La aplicación de estudio se finaliza el trabajo con el core 1.

Figura 3.1: Diagrama de tiempos del método de apagado de cores.

3.2 Método de encapsulado de aplicaciones en scripts

Al reanudar un checkpoint, es posible cambiar el binario que se esté ejecutando por uno nuevo desde el exterior de la simulación. El problema que presenta esta funcionalidad es que todo el experimento que se busque realizar debe estar en el nuevo binario.

Cuando el caso de experimentación consiste en ejecutar una aplicación con distintos parámetros de entrada, existen dos soluciones básicas: i) ejecuta el experimento reiniciando el checkpoint con los distintos parámetros de entrada o ii) incluir en la aplicación toda la lógica para ejecutar la aplicación con los distintos parámetros.

Para la primer opción, se debe reiniciar un checkpoint e iniciar todo el sistema por cada experimento. La demora generada por inicializar el sistema cada vez, resulta en tiempo desperdiciado. En el segundo caso, se debe modificar la aplicación, que implica modificar el propio experimento.

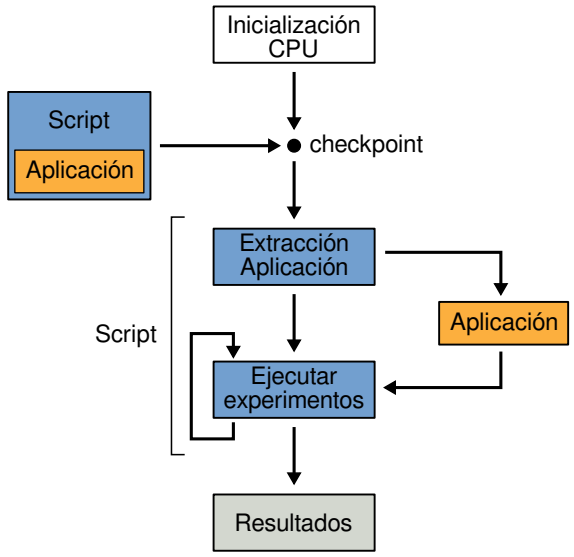


Figura 3.2: Diagrama de interacción entre la aplicación y el script para el método de encapsulado de aplicaciones.

La solución que se plantea a este problema, consiste en encapsular la aplicación dentro de un script (Fig. 3.2). Una vez que el script está siendo ejecutado por el simulador, se almacena en un archivo el código de la aplicación a ejecutar. Cuando se termine de almacenar la aplicación en un archivo, este binario es ejecutado una y otra vez con los distintos parámetros desde el script.

Adicionalmente, sería posible tener la aplicación en la imagen de disco del sistema, pero cada vez que se modifique la aplicación se deberá crear un nuevo checkpoint. Esta solución podría ser utilizada en el caso que la aplicación del experimento sea fija.

3.3 Uso de SE para simular la propuesta

A diferencia de FS, en SE no se requiere de un sistema operativo. Los cores son administrados por una capa de emulación provista por `gem5`. La propuesta de este trabajo requiere de la administración de los cores de forma interna, es decir, la lógica para administrar, iniciar y configurar un core será parte del core en sí. Por esta razón el modo SE se adapta mejor a la posibilidad de administrar manualmente los procesadores.

La implementación de la propuesta permite que, por medio de instrucciones, se ejecute un contexto de ejecución en un determinado procesador. Esta lógica no puede ser implementada en FS si no se tiene en consideración al sistema operativo, como fue mencionado en la sección 3.1.

Las nuevas instrucciones acceden al estado interno de los cores por una interfaz abstracta provista por `gem5`. Desde ésta, ejecuta funciones para acceder a registros y encender o apagar cores. Las funciones internamente consideran la lógica detrás del modelo de CPU, permitiendo acceder a una interfaz arquitectural del procesador y abstrayendo el comportamiento interno del modelo de CPU.

Tanto para detener como para encender un core se debe tener en cuenta el estado del pipeline. En particular, se debe esperar por el *commit* de todas las instrucciones en vuelo antes de ser detenido. A su vez, antes de ser encendido, éste debe estar inicializado. Esta última tarea se realiza cargando un estado válido al core.

3.4 Preparación de programas

Para que un programa genérico soporte la propuesta presentada en este trabajo, se deben tener en consideración los siguientes puntos:

- Los compiladores no soportan instrucciones fuera de las dadas por la arquitectura: Para definir instrucciones, éstas se deben codificar directamente en binario, ya que el ensamblador no conoce los nombres ni la codificación de las nuevas instrucciones. Incluso esto puede traer problemas en el uso de herramientas de debug o desensambladores.
- No existe soporte de compilador específico para la propuesta: Por esta razón es fundamental tener en cuenta la forma en que el compilador genera código. Las licencias que éste pueda tomar deben ser limitadas, a fin de generar el código esperado. Desde restringir el uso de registros específicos, hasta realizar operaciones atómicas.
- La paralelización de código se realiza manualmente: No se tienen herramientas que permitan marcar código y automáticamente generar su versión paralela. Esta tarea se realiza manualmente por medio del agregado de instrucciones específicas y modificaciones simples sobre código secuencial.

Teniendo en consideración estas limitaciones, preparar programas para soportar este tipo de propuestas de paralelización resulta laborioso. La figura 3.3 ilustra el proceso para paralelizar una aplicación.

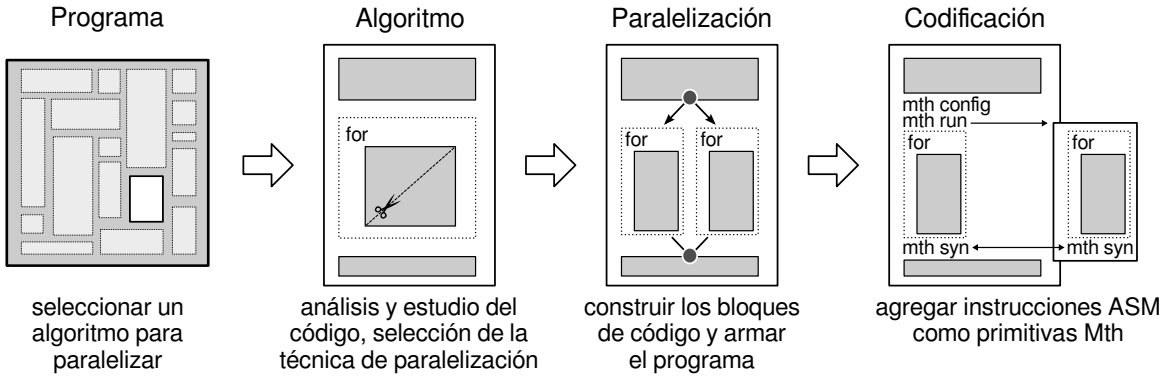


Figura 3.3: Proceso para convertir una aplicación para soportar *Mth*.

En primer lugar, se debe detectar un fragmento de código que pueda ser paralelizado dentro de una aplicación completa. Esta tarea no es imposible, pero encontrar manualmente dentro de la lógica de una aplicación, el conjunto de *spots* (lugares) donde sea aplicable este tipo de paralelización, lleva mucho tiempo y es muy susceptible a errores.

Una vez detectado el lugar donde la paralelización es factible, el próximo paso es analizar el código y determinar qué estrategia de paralelización resulta más conveniente.

El tercer paso consiste en construir el algoritmo paralelo, por ejemplo, separar el `for` para luego ser resuelto mediante dos threads diferentes. El último paso es implementar la propuesta, en el caso de *Mth*, es insertar instrucciones especiales que resolverán la creación y sincronización de threads. Ambas tareas son resueltas por el hardware de *Mth* detallado en la capítulo 5.

Realizar esta serie de pasos de forma manual es laborioso y muy susceptible a errores. Por esta razón será fundamental para el futuro contar con soporte de las herramientas de desarrollo para paralelizar aplicaciones. Además, si bien el funcionamiento de una aplicación puede ser paralelizado, las decisiones de diseño tomadas para su implementación, pueden impedir que la paralelización sea eficiente.

Capítulo 4

Modelo de threads livianos por hardware

En este capítulo se explica en detalle el modelo planteado como propuesta de esta tesis. La lectura comienza describiendo la organización del sistema, en particular, la interacción y responsabilidades de los cores (hardware) con el software. Luego, se enumeran las propuestas de extensión a la arquitectura, detallando principalmente decisiones de diseño. Por último, en un plano de más alto nivel, se explica el modelo de programación. Este punto resulta fundamental para mostrar las capacidades de la propuesta de dar soporte a todo tipo de frameworks de programación paralela.

4.1 Propuesta

El objetivo principal es lograr la utilización de todo el poder de cómputo disponible en sistemas multicore por parte de aplicaciones que, actualmente, no pueden aprovechar el creciente número de cores. La visión detrás de este objetivo considera que la mayoría de los problemas a resolver suelen ser simples, o presentarse en instancias muy pequeñas. Es decir, que el usuario promedio no requiere resolver problemas grandes, sino que al contrario, requiere resolver rápidamente muchos problemas distintos de pequeña magnitud. Esto puede significar resolver varios problemas en cores diferentes o tener la capacidad de resolver un solo problema pequeño en múltiples cores.

Ejecutar y administrar múltiples threads es una tarea compleja, que involucra a distintas piezas de software: aplicaciones, bibliotecas, frameworks y sistema operativo. Cada uno tiene sus responsabilidades y sus propias decisiones de diseño que impactan al momento de crear, ejecutar o sincronizar y, en su conjunto, generan overhead de administración. Además, en el caso general, el software opera a distintos niveles de protección, tanto como usuario como de kernel. Esto obliga al software de usuario a interactuar con el sistema operativo para la manipulación de threads, profundizando la generación de overhead. Si el objetivo es ejecutar múltiples threads con poco trabajo a resolver, entonces resulta clave la minimización de este overhead.

La propuesta consiste en extensiones de hardware que, activamente, resuelvan la administración de threads. El hardware dará soporte al software de usuario para construir contextos de ejecución y lanzarlos sin interacción con el sistema operativo. El software de usuario utilizará instrucciones especiales para crear contextos en otros cores y, eventualmente, lanzarlos cuando la lógica de la aplicación lo requiera.

Para crear un contexto y ejecutarlo rápidamente, se requiere un lugar en el cual almacenarlo e instrucciones específicas para su gestión. El objetivo de las extensiones propuestas es solucionar estos dos aspectos.

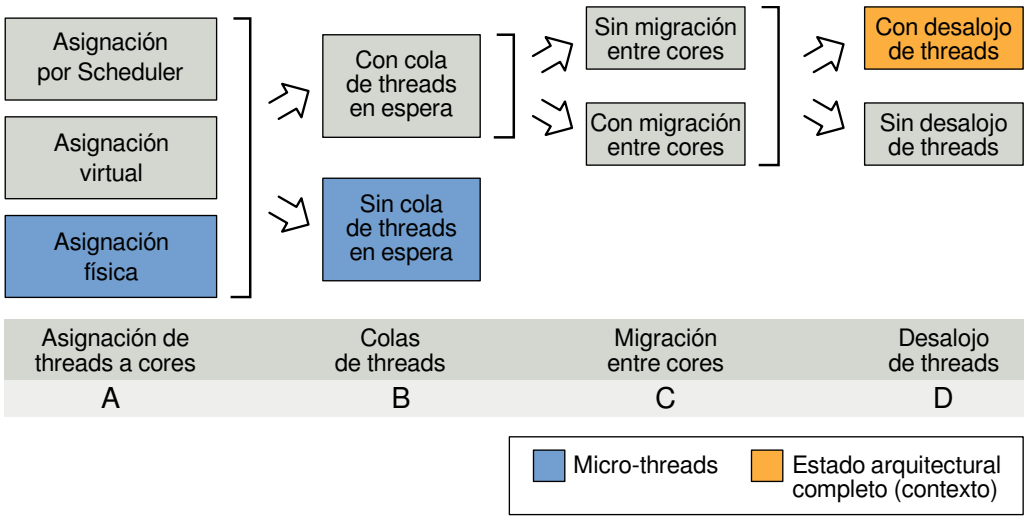


Figura 4.1: Distintas opciones de diseño para la administración de threads en cores. Se remarcan las opciones elegidas para este trabajo.

La figura 4.1 muestra un esquema de posibles decisiones de diseño de alto nivel para la gestión de threads en cores. A continuación se explican cada uno de estos puntos como opciones, para luego pasar a profundizar acerca de los detalles de su implementación.

A— Modo de asignación de threads a cores:

- *Scheduler*: Implementación de un scheduler por hardware, las instrucciones asociadas crean un contexto y lo ejecutan. El scheduler se encarga de designar el core donde va a ser ejecutado sin intervención del software.
- *Virtual*: Desde el punto de vista del usuario, la numeración de los cores se presenta mediante una tabla de renombres. Cada core físico puede estar asignado, o no, a un core o, incluso, a varios cores virtuales.
- *Físico*: El usuario referencia a cada core según su numeración física.

B— Tipo de colas de threads: Cada core puede tener una cola de contextos de ejecución (threads desde el punto de vista del usuario). Estas colas pueden ser de tamaño fijo o variable, haciendo uso de memoria principal o registros internos en el procesador.

C— Migración entre cores: Se implementa a nivel del scheduler por hardware, estudia la carga de los cores y migra los contextos de ejecución de un core a otro. El objetivo es el balanceo de la carga para el conjunto de cores.

D— Desalojo de threads: La capacidad de desalojar contextos implica el desarrollo de un scheduler completamente en hardware. Éste requiere contener el estado completo de un contexto además de información complementaria para decidir qué thread ejecutar.

La combinación de las opciones mencionadas, abren un abanico de posibles implementaciones para la gestión de threads. En la figura 4.1 se ordenan desde las más complejas a las más simples de arriba hacia abajo.

La asignación de cores por parte del scheduler requiere una visión global de todas las tareas que están siendo ejecutadas y cuáles están en espera, una implementación de este modelo se muestra en el trabajo de Kumar et al. [24]. En un modelo con estas características, la extensión de instrucciones resulta simple ya que se encargaran de encolar y desencolar tareas que el scheduler asignará a los cores.

La asignación virtual de cores, por su parte, presenta algunas dificultades. Cuando la cantidad de cores disponibles es menor que la cantidad solicitada, un core físico será asignado a más de un core virtual. En este caso, será necesario implementar un sistema de colas para ejecutar los threads a medida que son cargados.

Para implementar eficientemente colas de procesos, se pueden considerar dos variantes. Que cada core tenga su propia cola de threads y algún mecanismo de balance de carga, o una cola de threads global y asignación a los cores con balance de carga. Modificaciones sobre estas dos variantes se pueden ver como implementaciones tanto en software como en hardware.

La migración de threads, se puede implementar de formas variadas. Por un lado, es posible tener un conjunto de excepciones, que actúen bajo eventos de disparidad en el balance de carga de los cores. Las rutinas que atiendan estas excepciones ejecutarán código provisto por el usuario o el sistema para balancear las colas de threads. Otra posibilidad es tener soporte específico de hardware para esta tarea, en cuyo caso se debe tener una visión global de todas las colas de procesos de todos los cores.

Además, el modelo no está limitado a cores homogéneos. Las políticas de asignación y designación de cores, como la posibilidad de migración de threads, juegan un rol importante en este aspecto. La manera en que son elegidos los cores, de forma que el usuario tenga control de las características de los cores donde son ejecutados los threads resulta fundamental. Como así también la posibilidad de que el sistema automáticamente detecte la carga de los cores y actúe en consecuencia, tanto migrando los threads a cores más potentes, como el caso contrario.

En el caso de no contar con un sistema para almacenar colas de threads, es posible considerar migración de threads, si se busca balancear la carga de cores heterogéneos. Además, este mecanismo puede ser utilizado para balancear el consumo energético o temperatura a raíz de la disposición física de los cores en el chip. Para el caso de desalojo de contextos de ejecución no es posible realizarlo por hardware, sin tener soporte de colas. Sin embargo, es posible simplificar esta tarea si se cuenta con un mecanismo por hardware para la implementación de un scheduler en software.

Si bien todas estas opciones pueden resultar interesantes y eficientes para casos particulares, el modelo de *Mth* busca minimizar el overhead al máximo posible. El objetivo de *Mth* es poder extraer paralelismo a código que, de no ser por esta propuesta, no podría ser ejecutado en paralelo debido a que la relación entre overhead y cómputo útil lo haría inviable. Por esta razón, se optó por la solución más simple y que menos overhead implica. Es decir, no tener soporte de threads en espera, denominado a esta configuración como *modelo estático*. Se debe tener en cuenta que implementar cualquier maquinaria de gestión de threads para una cantidad reducida, resulta en un desperdicio de recursos en el caso general.

Un proceso tiene un conjunto de threads, la gestión por hardware de threads en *Mth* no tiene colas, ni soporta migración o desalojo. Ambas tareas, en el caso de ser requeridas, se deben resolver por software. Para *Mth*, los cores serán identificados por su número. El usuario tendrá a disposición un mecanismo para poder cargar en cada core una nueva tarea a ejecutar.

Un proceso, entonces, tendrá asociado un conjunto de threads (i.e. *micro-threads*). Cada micro-thread mantendrá información sobre el estado del procesador desde el punto de vista arquitectural (i.e. visión del programador). El proceso, como un todo, será administrado por el sistema operativo, pero éste no tendrá control sobre los micro-threads asociados. La administración de los recursos asignados al proceso caerá bajo la responsabilidad de cada proceso.

4.2 Organización del sistema

La premisa fundamental de diseño es proporcionar un conjunto de modificaciones que puedan ser implementadas sobre **procesadores existentes** para dar soporte *Mth*, pero manteniendo el funcionamiento

compatible con el procesador sin modificar. Es decir, las extensiones no deben alterar la arquitectura básica de los procesadores ni su lógica de funcionamiento. Desde el punto de vista del usuario, las extensiones se deben utilizar de la misma forma entre plataformas distintas, un ejemplo de esto son las extensiones multimedia en los procesadores ARM e Intel, que son implementadas por cada procesador de forma completamente diferente, pero manteniendo el mismo objetivo.

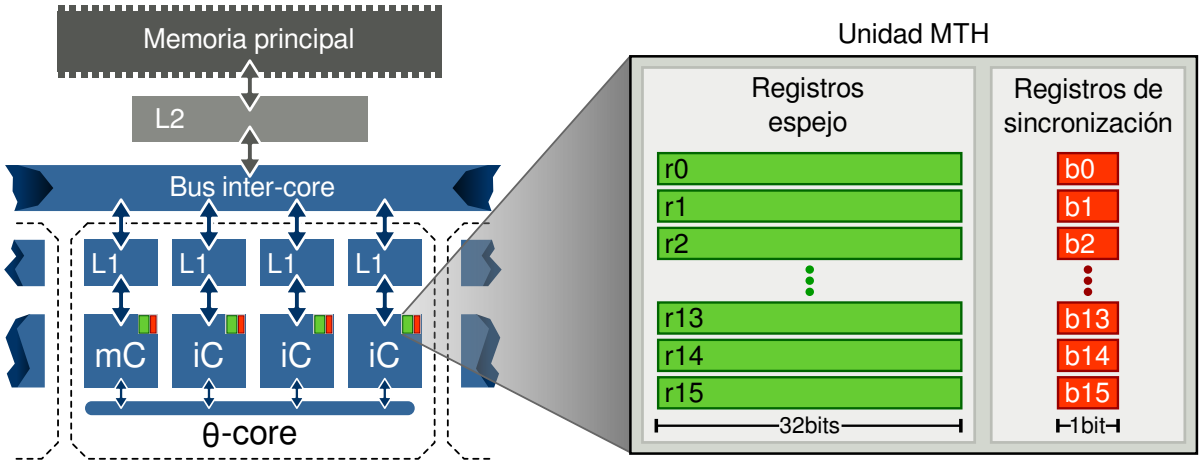


Figura 4.2: Cada core contiene su propia caché de L1, su banco de registros espejo y bits de sincronización. Un Θ-core es un grupo de cores conectados por medio de un bus, que comparten caché de L2.

Los cores dentro del procesador son agrupados respetando una jerarquía de dos niveles. En el primer nivel se encuentran los Θ-core, que agrupan una cantidad limitada de cores. La figura 4.2 ilustra la organización un Θ-core de cuatro cores, detallando en el margen izquierdo la información almacenada por la unidad *Mth* para uno de los cores. Dentro de cada Θ-core, un core es designado como principal, *main core* (mC), que controla al resto de los cores internos (*internal cores*, iC). No existe ninguna diferencia entre los cores, esta designación es arbitraria y atiende al funcionamiento del software. Esta misma decisión se puede ver en la *MultiProcessor Specification*[19] que identifica al *bootstrap processor* (BSP) y a los *application processors* (AP).

Todos los cores tiene su propia caché de L1 y están conectados a memoria principal a través de la caché de L2. La administración interna de los Θ-core es realizada por medio de un bus que interconecta a todos los cores que lo integran. En general, la organización de un Θ-core es equivalente a un multicore clásico con el agregado de los módulos *Mth*.

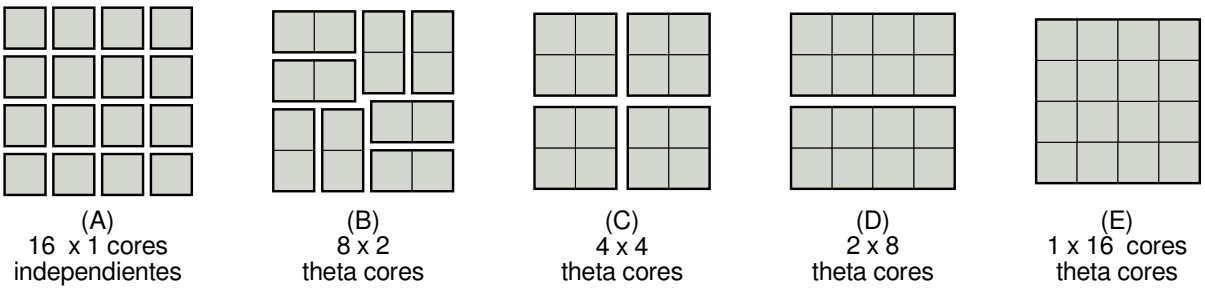


Figura 4.3: Posibles grupos de cores dentro de un procesador.

En la figura 4.2 cada Θ-core posee cuatro cores idénticos. Es de notar que esta configuración no se debe a una restricción de diseño. Tanto la cantidad como las características de los cores pueden ser variadas. En este sentido, la figura 4.3 muestra distintas organizaciones posibles. Las opciones disponibles agrupan los cores en conjuntos tanto uniformes como no. Otro aspecto de la organización se ilustra en la figura 4.4,

donde la distribución de cores en grupos puede ser: variada pero fija para cada conjunto (A), variable pero en posiciones fijas (B), o variable sobre cualquier posición (C). Cualquiera sea el caso, esto no limita la posibilidad de tener cores especializados, o de distintas características de rendimiento y consumo (D).

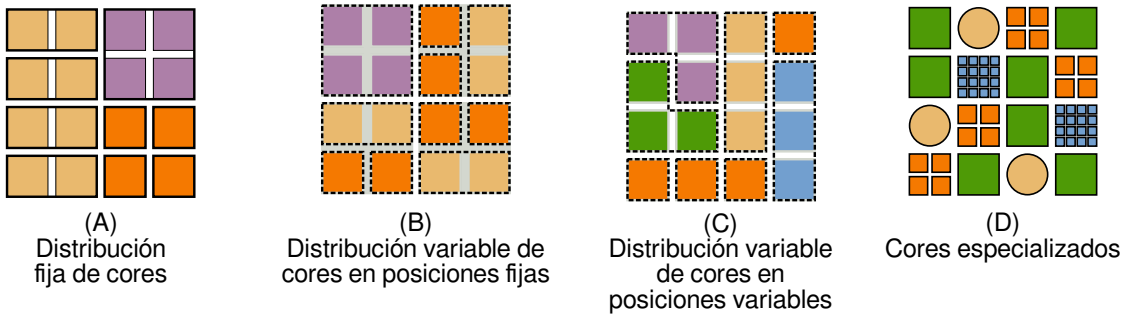


Figura 4.4: Posibles modelos de grupos de cores dentro de un procesador.

4.3 Extensiones a la arquitectura

El soporte de hardware se propone como un conjunto de extensiones a un procesador estándar. Estas extensiones operan como parte del procesador y afectan tanto su funcionamiento como organización. Un procesador multicore típico está organizado como un conjunto de procesadores independientes que comparten unidades como memorias caché, buses y controladores.

Se consideran dos extensiones a los cores:

(i) Un banco de registros espejo utilizado para guardar el contexto de un proceso y bits de sincronización.

(ii) Instrucciones específicas para el control y administración de procesadores.

El banco de registros espejo provee un espacio temporario en el cual almacenar el contexto de la próxima tarea durante la etapa de configuración. Éste no incrementa la complejidad de la arquitectura, ya que no requiere nuevos puertos para interconexiones sino, solamente, de un mecanismo simple que permita copiar todos los valores del banco de registros espejo al banco de registros de la arquitectura. Su objetivo es contener la información necesaria para crear un contexto (program counter, stack y parámetros). No todos los registros del sistema necesitan tener una copia en el banco de registros espejo, para nuestra implementación se consideraron solamente el banco de registro de enteros.

Considerar por ejemplo un procesador ARM A9 de 32 bits, con una tecnología de integración de 22 nm. En este caso, el banco de registros representa un $3{,}5\,\%$ del area del procesador, calculado por medio de la herramienta `McPAT`. En un procesador Intel Xeon a 65 nm, el banco de registros representa menos del $1\,\%$ del area total. En este trabajo se propone tener una copia solamente de los registros arquitecturales, esto representaría significativamente menos que duplicar el banco de registros por completo.

La arquitectura provee instrucciones para escribir valores en el banco de registros espejo de cualquier core. Durante la etapa de configuración el código de cualquier procesador puede escribir el banco de registros espejo de otro procesador y cargar en este un contexto. Las instrucciones utilizadas pueden ser implementadas para ser ejecutadas sin interferir con la tarea que se esté ejecutando, es decir, modificar el banco de registros espejo no altera la ejecución del core modificado.

Para la sincronización de las tareas, se cuenta con registros de 1-bit, que operan como un *full-empty bit*. Si el valor difiere del dado, entonces se cambia y continúa, caso contrario se bloquea a la espera del cambio en el bit de sincronización.

A continuación se listan las instrucciones agregadas a la arquitectura:

- `mth_run <cpu>`: Inicia un core. Copia todos los valores del banco de registros espejo, al banco de registros de la arquitectura. Los valores almacenados en el banco de registros espejo fueron configurados durante una etapa incial. Estos serán persistentes entre llamados a `mth_run`.
- `mth_mov <reg_src><reg_dst><cpu_dst>`: Mueve el valor almacenado en `reg_src` desde el core actual al registro `reg_dst` del banco de registros espejo en el core `cpu_dst` (core diferente al actual). El movimiento puede ser realizado sin interferir con la ejecución del core destino `cpu_dst`.
- `mth_end`: Detiene un core. Esta instrucción es similar a `halt` y es ejecutada desde el core que será detenido. Este core permanecerá en este estado hasta un nuevo llamado a `mth_run` para comenzar otra tarea.
- `mth_set <bit><cpu_dst><state>`: Escribe un 1 o 0 en un bit de estado. Permite inicializar los bits de sincronización.
- `mth_syn <bit_dst><cpu_dst><state><end>`: Sincroniza un core basándose en el valor de un registro de 1-bit. Si `<state>` difiere del valor de `bit_dst`, este cambia el valor y continúa. Si el valor es igual, bloquea el thread hasta que cambie. El argumento `<end>` es opcional, detiene la ejecución luego de la sincronización.

En la implementación actual sobre el simulador gem5, las instrucciones operan como un *fence*(*Instruction barrier*), previniendo cargar nuevas instrucciones hasta que éstas sean ejecutadas. Esta decisión resulta más restrictiva que la implementación óptima, creando un desfasaje de, al menos, 10 ciclos dentro del pipeline.

4.3.1 Ejemplo de funcionamiento

Con el fin de ilustrar el funcionamiento de las instrucciones, se propone el siguiente ejemplo: ejecutar en paralelo dos tareas diferentes.

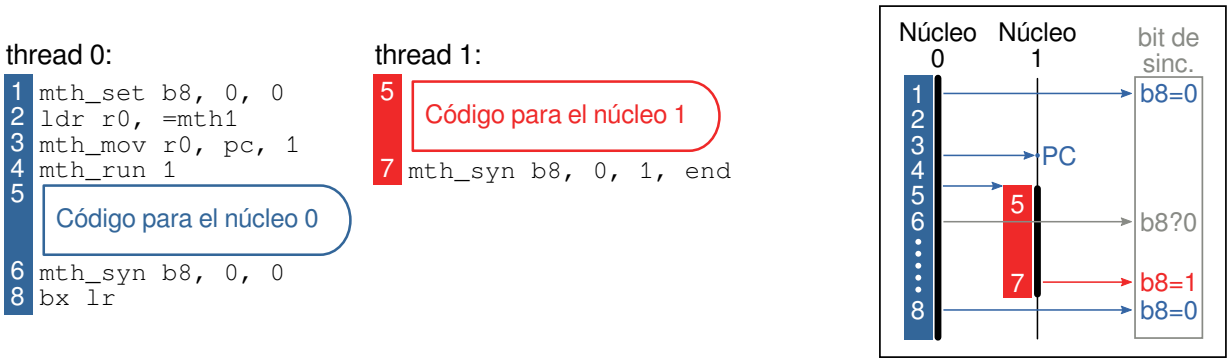


Figura 4.5: Ejemplo de uso de primitivas *Mth*. El `thread 0` es ejecutado en el core 0. Esté, configura el core 1 para ejecutar el `thread 1`. Luego, la nueva tarea es lanzada, ejecutada y sincronizada posteriormente.

La figura 4.5 muestra dos tareas ejecutando en un Θ-core utilizando instrucciones ARM. La tarea principal (`thread 0`) ejecuta en el core 0 (mC), configura los registros para la nueva tarea (`thread 1`) en el core 1 (iC). Luego, la segunda tarea es lanzada, ejecuta su código y se sincroniza.

A continuación se describe cada uno de los pasos de la figura:

(1) Configura en 0 el bit de sincronización b8 en el core principal.

(2) Carga la dirección de `thread 1` en el registro `r0`.

(3) Carga desde el registro `r0` la dirección de la tarea `thread 1` al PC del banco de registros espejo del core 1.

(4) Comienza a ejecutar el core 1, es decir, la tarea `thread 1`.

(5) Ambas tareas `thread 0` y `thread 1` son ejecutadas.

(6) Paso de sincronización: `thread 0` trata de configurar a 0 el bit `b8`. Como este bit tiene el valor 0, el core se queda a la espera hasta que `b8` tenga el valor 1.

(7) Paso de sincronización: `thread 1` trata de configurar en 1 el bit `b8`. Como este bit tiene el valor 0, se puede cambiar a 1. Esta instrucción termina la tarea `thread 1`.

(8) Luego de sincronizar los cores, las instrucción puede ser ejecutada.

4.4 Modelo de programación

El mecanismo de *threading* provee a los desarrolladores una forma de dividir sus programas en partes mayormente independientes, bajo la premisa que siempre será posible ejecutar algún thread [34]. Además, para la mayoría de los sistemas, intercambiar threads es más rápido que intercambiar procesos debido a que los threads son más livianos que los procesos. Los threads están contenidos dentro de procesos y utilizan el mismo código ejecutable. Por lo general, comparten la misma memoria y acceso a dispositivos. Para que los threads puedan ser ejecutados independientemente, éstos requieren tener tanto el contexto de ejecución, como el stack de forma independiente.

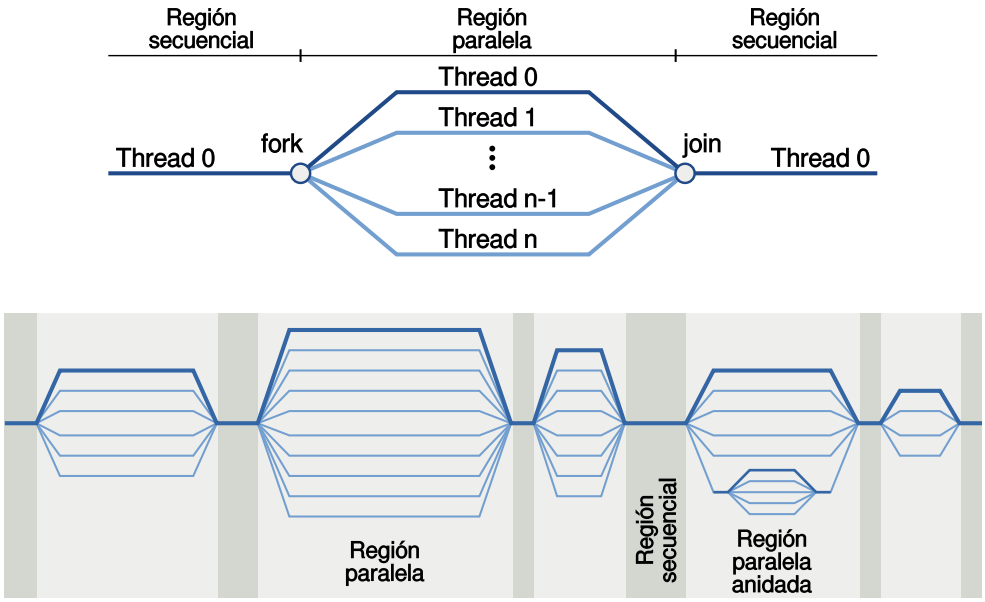


Figura 4.6: Modelo de ejecución `fork-join`. Regiones de ejecución secuencial y paralelo.

En la figura 4.6 se ilustra el proceso por el cual un thread se divide (`fork`) en varios threads dando inicio a una región de ejecución en paralelo. Cuando estos threads finalizan, realizan la operación de `join` para que, al quedar un único thread, se continúe con la ejecución secuencial. Es importante destacar que este procedimiento se puede realizar en múltiples ocaciones, incluso de forma anidada.

Una implementación de este modelo y el estándar *de facto* es `POSIX threads`, o simplemente, `Pthreads`. Éste especifica una API para programación multithreaded, operando como una biblioteca que debe ser enlazada con el código de usuario.

Desde el punto de vista abstracto, el modelo de programación de *Mth* es similar a `Pthreads` pero con tres diferencias fundamentales:

i) *Costo de creación de nuevos threads*: Reduce considerablemente el costo de creación de threads. El soporte de hardware juega un rol fundamental en disminuir el tiempo necesario para comenzar un nuevo thread. Lanzar un thread consiste en mover un conjunto de registros desde el banco de registros espejo al banco de registros de la arquitectura, evitando la intervención del sistema operativo.

ii) *Sincronización activa*: Los cores tienen la capacidad de esperar por algunos ciclos de reloj mientras las tareas son completadas. Esto permite disminuir el tiempo de espera por un evento, pero la aplicación debe estar diseñada para que los cores no deban esperar por tiempos muy prolongados.

iii) *Creación de thread limitada*: Dependiendo del modelo de micro-thread, la cantidad de threads que pueden ser ejecutados depende de la cantidad de cores disponibles o de la cantidad de lugares en la cola de contextos.

Al dejar el control de los cores en manos de la aplicación, se permite realizar un uso altamente eficiente de los mismos. Esto puede ayudar a reducir notablemente el consumo de energía si fuera combinado con un mecanismo de soporte en hardware para encender y apagar los cores, dando la posibilidad de utilizar los cores internos solo sobre demanda.

4.4.1 Rol del Sistema Operativo

Un proceso estará compuesto, no solo por un único contexto, sino por un conjunto de contextos válidos de un Θ-core, correspondientes a cada uno de los cores internos. El sistema operativo administrará el uso de los Θ-core, cargando contextos de ejecución y estados en los bancos de registros espejo. Además, el sistema operativo podrá identificar si el contexto de un core interno es válido, al igual que el estado del banco de registros espejo.

La aplicación, desde el punto de vista del programador, tendrá control de los recursos disponibles, conociendo la cantidad de cores dentro del Θ-core y sus características. La gestión por parte de la aplicación dispondrá de los cores internos (iCs), creando threads y controlando que los threads sean ejecutados en cada uno, sin intervención del sistema operativo.

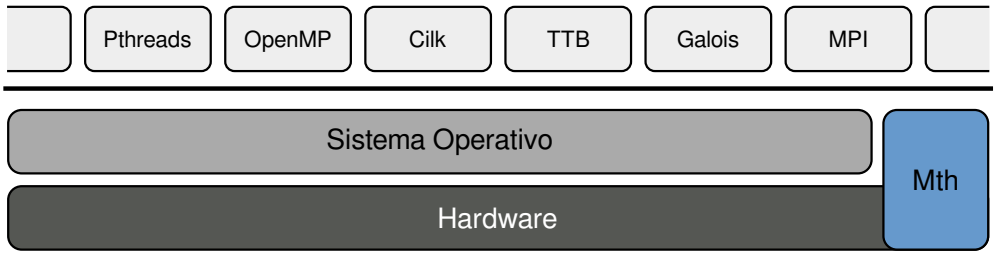


Figura 4.7: *Mth* puede ser usado como soporte de frameworks, para poder extraer paralelismo que actualmente no es posible conseguir.

Logrando este tipo de soporte por parte del sistema operativo, es posible implementar todo tipo de frameworks de paralelismo adaptados a los requerimientos específicos de cada aplicación. Incluso es posible que frameworks existentes sean implementados utilizando *Mth* como base para acceder a los recursos de cómputo paralelo (fig. 4.7), abriendo la puerta para que, sin realizar una gran alteración del código fuente de la aplicación, se pueda acceder a los beneficios de *Mth*.

Capítulo 5

Aportes en infraestructura de simulación

Durante la inicialización, gem5 requiere completar una serie de pasos de configuración e instanciación de módulos (explicados en la sección 2.1.1). Gracias a esta arquitectura modular es posible simular diversos tipos de sistemas.

El módulo *Mth*, por su parte, está diseñado para alterar esta arquitectura lo mínimo posible. Para lograrlo, su funcionamiento fue aislado de todo el código de gem5 por medio de interfaces. Éstas permiten acceder a las estructuras internas del simulador de forma consistente con el motor de simulación.

En este capítulo se describirán las instrucciones agregadas a gem5, su codificación y detalles de implementación. Además, se detallará la forma en que se describen ISAs dentro de gem5. Por último se buscará precisar la implementación del módulo *Mth* en sí mismo.

5.1 Nuevas instrucciones

Para la implementación del modelo se deben considerar más instrucciones que las planteadas en la sección 4.3, ya que se deben implementar instrucciones de inicialización y *debug*. También aparecen conjuntos de instrucciones que codifican una sola instrucción lógica, pero que, desde el formato de instrucción, se ven como diferentes instrucciones.

Las nuevas instrucciones definidas para implementar el modelo básico de *Mth*, es decir, el modelo de asignación estática de cores, son las siguientes:

- Inicialización y debug
  - **Init**: No toma parámetros, inicializa el módulo *Mth* una vez que el sistema está en ejecución. Se ejecuta una sola vez desde que se inicia el sistema.
  - **Debug**: Toma como parámetros los valores de los registros y funciona como herramienta de debug. Permite desde capturar tiempos de ejecución sin utilizar el sistema de estadísticas de gem5, hasta dar cuenta del valor de registros o posiciones absolutas en memoria.
- Instrucciones del modelo estático de ejecución
  - **Run**: Comienza a ejecutar el core indicado por parámetro.
  - **End**: Detiene la ejecución del core actual.

- Instrucciones para el banco de registros espejo
  - **MovRM** y **MovWM**: Lee o escribe un registro del banco de registros espejo de un core.
- Instrucciones de los registros de sincronización
  - **SetReg**: Dado un core y un registro de sincronización, escribe un 0 o 1 en éste.
  - **MovRC** y **MovWC**: Lee o escribe un registro de sincronización de un core (implementa la instrucción `syn`).
  - **MovRCend** y **MovWCend**: Lee o escribe un registro de sincronización de un core y lo detiene (implementa la instrucción `syn`).

5.1.1 Formato de instrucción en ARM

En etapas preliminares de este trabajo se implementó toda la maquinaria de *Mth* sobre la arquitectura *Alpha*. Esta decisión se tomó basado en el amplio soporte en `gem5` para los distintos modelos de simulación. Sin embargo, la arquitectura *Alpha* actualmente no es utilizada, incluso se han dejado de fabricar procesadores basados en este ISA. La falta de soporte de herramientas de desarrollo, como de sistema operativo, obligó a migrar el trabajo a una arquitectura más utilizada. La elección fue `ARM`, ya que es la segunda arquitectura mejor soportada por `gem5`.

La tabla 5.1 describe el formato de instrucción de `ARM`, se indica el tipo de instrucción que será utilizado para codificar las instrucciones de *Mth*.

	31 30 29 28	27 26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
A	cond	0 0	I	Opcode				S	Rn	Rd	Operand 2		
B	cond	0 0 0	0 0 0				A	S	Rd	Rn	Rs	1 0 0 1	Rm
C	cond	0 0 0	0 1	U			A	S	RdHi	RdLo	Rs	1 0 0 1	Rm
D	cond	0 0 0	1 0	B			0	0	Rn	Rd	0 0 0 0	1 0 0 1	Rm
E	cond	0 1	I	P	U	B	W	L	Rn	Rd	offset		
F	cond	1 0 0	P	U	S	W	L		Rn	register list			
G	cond	0 0 0	P	U	1	W	L		Rn	Rd	offset1	1 S H 1	offset2
H	cond	0 0 0	P	U	0	W	L		Rn	Rd	0 0 0 0	1 S H 1	Rm
I	cond	1 0 0	L	offset									
J	cond	0 0 0	1 0 0 1	0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1	Rn				
K	cond	1 1 0	P	U	N	W	L		Rn	CRd	CPNum	offset	
L	cond	1 1 1 0	Op1						CRn	CRd	CPNum	Op2	0 CRm
M	cond	1 1 1 0	Op1				L		CRn	CRd	CPNum	Op2	1 CRm
N	cond	1 1 1 1	SWI Number										

A: Data processing / PSR Transfer, **B**: Multiply, **C**: Long Multiply, **D**: Swap, **E**: Load/Store Byte/Word, **F**: Load/Store Multiple, **G**: Halfword transfer (Immediate offset), **H**: Halfword transfer (Register offset), **I**: Branch, **J**: Branch exchange, **K**: Coprocessor data transfer, **L**: Coprocessor data operation, **M**: Coprocessor register transfer, **N**: Software interrupt

Tabla 5.1: Formato de instrucción de ARM

Los nombres de los campos dependen de cada instrucción:

- `cond`: Predicado, condición sobre los *flags* que determina si la instrucción debe ser ejecutada.
- `Rd`: Registro destino.

- `Rn`, `Rs` y `Rm`: Registros operandos.
- `CPNum`: Número de coprocesador.
- `CRd`, `CRn` y `CRm`: Registros de coprocesador.
- `Opcode`, `Op1` y `Op2`: Código de operación.

5.1.2 Codificación de instrucciones en ARM

La codificación de instrucciones se realizó considerando las mismas como un coprocesador. Esta forma de implementar las instrucciones es equivalente a la utilizada por los desarrolladores de `gem5` para implementar las pseudoinstrucciones mencionadas en la sección 2.1.6.

En este caso, se utilizó el formato tipo `M` de la tabla 5.1. En la tabla 5.2 se describe en detalle cada una de las instrucciones implementadas para *Mth*, se indican además los parámetros de cada una.

	31-28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	cond	1	1	1	0	Op1			L	CRn				CRd				CPNum				Op2			1	CRm			
Init*	cond	1	1	1	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
Debug*	cond	1	1	1	0	0	1	0	1	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
Run	cond	1	1	1	0	0	1	0	1	0	0	1	1	0	0	0	0	0	0	1	0	core			1	0	0	0	0
End	cond	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
SetReg	cond	1	1	1	0	0	1	0	1	1	0	0	0	reg				0	0	1	0	core			1	type			
MovRM	cond	1	1	1	0	0	1	0	1	1	0	1	0	reg dst				0	0	1	0	core src			1	reg src			
MovWM	cond	1	1	1	0	0	1	0	1	1	0	1	1	reg dst				0	0	1	0	core dst			1	reg src			
MovRC	cond	1	1	1	0	0	1	0	1	1	1	0	0	reg dst				0	0	1	0	core src			1	reg src			
MovWC	cond	1	1	1	0	0	1	0	1	1	1	0	1	reg dst				0	0	1	0	core dst			1	reg src			
MovRCend	cond	1	1	1	0	0	1	1	0	1	1	0	0	reg dst				0	0	1	0	core src			1	reg src			
MovWCend	cond	1	1	1	0	0	1	1	0	1	1	0	1	reg dst				0	0	1	0	core dst			1	reg src			

* Instrucciones de debug e inicialización. No se requieren en la implementación real.

Tabla 5.2: Formato de las instrucción agregadas por *Mth*.

Los bits de las instrucciones indicados en gris corresponden a los parámetros de cada una. El recuadro `core` indica el procesador, mientras que `reg` corresponde al registro usado como parámetro. Tanto `dst` y `src` indican si actúan como destino o como fuente. Por último `type` puede valer `0000` o `0001`, como valor inmediato para la carga del registro de sincronización.

5.2 Filosofía de diseño de `gem5`

El diseño de `gem5` tiene como pilar la modularidad y la reutilización de código. Es por esto que agregar nuevos modelos de CPU o modelos de dispositivos resulta relativamente simple, no así modificaciones integrales que afecten varios módulos transversalmente. Todo el código está diseñado de forma genérica, cada módulo debe proveer una interfaz para *hablar* con el resto del sistema y puertos para conectar dispositivos entre sí. Un ejemplo de esto es como *el ISA* se aisla del modelo de CPU. Este último debe proveer una interfaz para que, desde el ISA, pueda resolver las instrucciones sin importar sobre qué modelo de CPU se esté simulando.

Por debajo de los módulos que integran el sistema, se encuentra la maquinaria de simulación de eventos. Ésta implementa una cola de eventos que son resueltos uno a uno dentro del ciclo principal del simulador, en la función `doSimLoop`. Para resolver un evento, se debe implementar la función `serviceOne`, que

llamará a alguna función de algún módulo. Las funciones que resuelven eventos son las encargadas de encolar los nuevos eventos en la mayor parte de los casos. De esta forma, solamente las funciones asociadas a eventos del simulador son las que interactúan con la maquinaria de eventos.

En este trabajo no se modificó la maquinaria de eventos, sino que se operó dentro del marco de los módulos de cores. Por esta razón solamente se detallará este último aspecto. En las siguientes secciones, se dará cuenta de cómo está estructurado el código de gem5 y el funcionamiento del generador de ISAs, junto con detalles sobre su lenguaje de descripción.

5.2.1 Estructura del código

El código de gem5 está organizado en diferentes carpetas, la carpeta `configs` contiene una serie de scripts desarrollados en `Python` para configurar simulaciones. Principalmente hay dos *scripts* ejemplo, uno encargado de configurar el entorno de simulación en modo *full-system* y otro para *system-emulation*. Adicionalmente contiene otro tipo de scripts, utilizados para inicializar entornos de simulación específicos, como fue explicado en la sección 2.1.4. Estos scripts no son parte del código fuente del simulador, sino que sirven como parámetros del mismo. Al inicio de un experimento en gem5, uno de los parámetros es el script de configuración, éste es ejecutado y como resultado se construye el entorno de simulación.

Por otro lado, la carpeta `src` contiene todo el código fuente del simulador dividido en varias subcarpetas. Fundamentalmente la carpeta `sim` contiene la implementación de la maquinaria de simulación. Es el punto de entrada del simulador, encargado de cargar el script `Python` para generar el entorno de simulación. Contiene las funciones para administrar la cola de eventos y el ciclo principal del simulador.

En la carpeta `arch` se definen las ISAs de cada arquitectura que gem5 soporta. Se define la forma de decodificación de cada instrucción, sus parámetros, la tarea que realiza y qué función de alto nivel la resuelve. Además, cada instrucción, según el modelo de CPU que se utilice, tiene propiedades y características que se relacionan con la forma en que esa instrucción funciona. Ejemplo de esto, es el nivel de privilegio en que puede ser ejecutada, o si soporta ser ejecutada fuera de orden. Todo el código que definen las ISAs está escrito en un lenguaje propio de gem5 para la descripción de arquitecturas. En la sección 5.2.2 se mencionan las generalidades del mismo y como este código es transformado a C++.

La implementación de los modelos de CPU, se encuentra en la carpeta `cpu`. Todos los modelos heredan de la clase `BaseCPU`. Sobre esta se implementa `BaseSimpleCPU` de la que heredan `AtomicSimpleCPU` y `TimingSimpleCPU`. Estos últimos son los modelos de CPU más simples que provee gem5. Por otro lado, los modelos más complejos son implementados por las clases `BaseO3CPU`, `InOrderCPU` y `MinorCPU` que heredan directamente de `BaseCPU`. En la sección 2.1.3 se explican las características de cada uno.

Por último, la carpeta `python` contiene todas las clases necesarias para la configuración de alto nivel de un entorno. Este código, además, contiene la interfaz para acceder desde `Python` a las clases de C++. En la sección 2.1.1 se describe el proceso de inicialización de una simulación, la implementación tanto del llamado a `instanciate` como a `simulate` es parte de este código.

El resto de los archivos se organiza siguiendo la estructura descripta a continuación:

- `configs/`: Scripts en `Python` para la configuración de simulaciones.
- `system/`: Firmware y bootloaders utilizados para entornos simulados.
- `src/`: Código fuente de gem5.
  - `src/arch/`: Implementaciones de ISAs y código de sistema operativo para SE.
  - `src/base/`: Estructuras de datos generales y algoritmos comunes a todo el proyecto.
  - `src/cpu/`: Implementación de los modelos de CPU.
  - `src/dev/`: Modelos de drivers y dispositivos.

- `src/kern/`: Código de sistema operativo, pero independiente de la arquitectura, definiciones.
- `src/mem/`: Modelos de administración de memoria y controladores.
- `src/sim/`: Código que implementa las funciones básicas y fundamentales del simulador.
- `src/python/`: Código `Python` para configuración y funciones de alto nivel.

Se debe observar que existen tres partes del código de `gem5` escritas en lenguaje `Python`. La primera corresponde a los scripts para generar el sistema (script de usuario en `/configs`), la segunda al código de alto nivel que funciona como interfaz de C++ (en `src/python`) y, la última, al código que interpreta y compila el lenguaje de descripción de ISAs (`src/arch/isa_parser.py`).

A fin de alterar mínimamente el código de `gem5`, la implemetación del módulo *Mth* se agregó como un conjunto nuevo de archivos dentro de una nueva carpeta entre los archivos fuente. Se tuvieron que realizar modificaciones al soporte de compilación para este fin.

5.2.2 Lenguaje de descripción de ISA

Es un lenguaje personalizado, diseñado especialmente para generar definiciones de clases y funciones de decodificación de instrucciones para M5 (simulador predecesor de `gem5`). A continuación se describe brevemente las características más destacadas, base necesaria para comprender parte del presente capítulo.

El parser se encarga de procesar las especificaciones y extraer las características de las instrucciones, con esta información genera tres archivos `decoder.hh`, `decoder.cc` y `execute.cc`. Éstos contienen el código necesario para decodificar instrucciones y ejecutarlas.

El archivo que describe un ISA está dividido en dos partes, la sección de declaraciones y de decodificación. Esta última especifica la estructura del decodificador y define qué instrucciones van a ser aceptadas por el decodificador. La sección de declaraciones, por su parte, define información general (clases, formatos de instrucción, templates, entre otros) que son requeridos para dar soporte del al decodificador.

Sección de decodificación

Esta sección está definida por un conjunto de bloques de decodificación anidados. Un bloque indica qué acción tomar para el conjunto de bits de una instrucción. El funcionamiento de un bloque de decodificación es similar a un *switch/case* de C. El siguiente es un ejemplo simplificado:

```
decode OPCODE {
  0: Integer::add({{ Rc = Ra + Rb; }});
  1: Integer::sub({{ Rc = Ra - Rb; }});
}
```

Un bloque de decodificación comienza con la palabra reservada `decode` seguida del nombre del campo de bits a decodificar. El resto del bloque es una lista de sentencias de la forma: *valor* `:` *formato* `::` *instrucción* `(` *código* `)`. *valor* indica el número que debe tener el campo de bits para activar la sentencia. *formato* es el formato usado para la decodificación, describe el template de código que será utilizado para resolver la instrucción. *instrucción* se refiere al nombre de la instrucción y *código* será instanciado dentro del template que resuelve la instrucción. En particular, *código*, será uno de los parámetros de la función encargada de generar el código para un formato de instrucción determinado.

El uso de doble llaves para la declaración de código, corresponde a la forma de declarar código literal. En este caso, se pueden agregar varias líneas de código sin necesidad de utilizar caracteres de escape. Si bien esta es la forma general de declarar sentencias de decodificación, no es la única. Existen formas adicionales de agrupar formatos o reutilizar código, pero en las que no se entrará en detalle en esta sección. Tanto la definición del campo de bits, como los formatos, serán definidos en la sección de declaraciones.

En una sentencia de decodificación, es posible remplazar la definición de código por otra sentencia de decodificación anidada. En este caso tanto el campo de bits interior como el exterior deben ser aceptados para que la sentencia pueda ser resuelta.

Sección de declaraciones

El propósito principal de esta sección es definir los formatos de instrucción y elementos de soporte usados por los bloques de decodificación para generar el código final del decodificador. Se detallará, a modo introductorio, solamente dos de los elementos que contiene la sección de declaraciones: las definiciones de formatos de instrucción y las definiciones de campos de bits.

Definición de formato de instrucción

Un formato de instrucción es, básicamente, una función `Python` de características especiales. Toma un conjunto de parámetros de la definición de la instrucción y genera, al menos, cuatro bloques de código `C++`. Estos bloques se distinguen por el lugar en que aparecerán en los archivos generados como salida.

- *header output*: Será parte del archivo `decoder.hh`, éste se incluirá en todos los archivos generados, tanto en el `decoder.cc` como el `execute.cc`.
- *decoder output*: Se ubica en el archivo `execute.cc`, luego de la función general de decodificación. Contiene los constructores de las clases de las instrucciones y definiciones de funciones especiales que dependen del tipo de instrucción.
- *exec output*: Contiene el código para función `execute()` de la instrucción. Esta función es la que efectivamente resuelve la tarea que realiza la instrucción.
- *decode block*: Contiene el bloque de código que ejecutará la función de decodificación para la instrucción dada. En el caso general, crea el objeto que representa la instrucción.

Una vez que la función de decodificación reconoce el patrón especificado por el bloque de decodificación, el código *decode block* es el encargado de crear una instrucción, devolver el objeto de la misma y poder ejecutarla (*exec output*).

La sintaxis para definir un formato de instrucción es el siguiente:

```
def format FormatName(arg1, arg2) {{
    [code omitted]
}};
```

En el ejemplo, el formato se denomina "`FormatName`", las instrucciones que utilicen este formato deben proveer dos argumentos (`arg1` y `arg2`). Adicionalmente, a los argumentos explícitos se proporcionan dos parámetros: `name` que es nemotécnico de la instrucción y `Name` que es el nemotécnico de la instrucción pero con la primer letra en mayúscula.

A pesar de que la descripción del ISA es independiente del modelo de CPU que se esté utilizando, parte del código debe ser especializado para cada modelo. En particular, el código que resuelve las instrucciones. Este comportamiento diferenciado es gobernado por la sustitución de símbolos específicos para cada modelo de CPU. Estos símbolos que comienzan con el texto `CPU` y son tratados especialmente por el parser. En la implementación actual solamente se utiliza el símbolo `CPU_exec_context` que se evaluá al nombre de la clase del modelo de CPU.

Si algún símbolo específico de CPU aparece dentro de los bloques de código `header_output`, `decoder_output` o `decode_block`, se duplica todo el bloque para cada uno de los modelos de CPU soportados. Para el caso de `exec_output`, siempre se replica el código para todos los modelos de CPU, sin importar si contiene o no algún símbolo.

Definición de campos de bits

Un campo de bits (*bitfield*) provee nombre a un conjunto de bits de una instrucción. Estos nombres son utilizados, en general, para especificar bits dentro de los bloques de decodificación, aunque también pueden ser utilizados por la definición de las instrucciones o el código del decodificador. El siguiente es un ejemplo de diferentes definiciones de campos de bits.

```
def bitfield OPCODE <31:26>;
def bitfield IMM <12>;
def signed bitfield MEMDISP <15:0>;
```

El rango que definen incluye siempre los dos extremos. El bit `0` corresponde al bit menos significativo de la instrucción. Para el ejemplo, `OPCODE` extrae los seis bits más significativos de una instrucción de 32 bits. En `IMM`, se extrae solamente un bit que es extendido con ceros, y para `MEMDISP`, se agregó la palabra reservada `signed`, que indica que este campo de bits será extraído extendiendo el signo. La implementación de los campos de bits se basa en macros del preprocesador de C++, que extraen el campo de bits especificado a partir de la variable implícita `machInst`. Al ser una macro, el tamaño del resultado dependerá del contexto.

5.3 Modificaciones sobre el código de gem5

La implementación del módulo *Mth* sobre gem5 implica tres cambios:

- Modificar el ISA para soportar nuevas instrucciones.
- Agregar el módulo *Mth* como parte de gem5.
- Modificar el sistema para que contenga una referencia a una instancia de *Mth*.

Para soportar nuevas instrucciones, se debió modificar la implementación del ISA. Dentro de los archivos `arch/arm/isa/decoder/arm.isa` y `arch/arm/isa/decoder/thumb.isa` se agregó un nuevo bloque de decodificación que resuelve todas las instrucciones de *Mth*. El primero de los archivos define la decodificación de todas las instrucciones en modo `ARM`, mientras que el segundo corresponde a la decodificación de instrucciones en modo *thumb*. En este último, a pesar de ser un formato de instrucciones reducido de 16 bits, soporta instrucciones de 32 bits equivalentes al modo `ARM`. Luego se agregó el archivo `arch/arm/isa/formats/mth.isa` que contiene la definición del código que identifica qué instrucción de *Mth* será decodificada y construye las instrucciones con los parámetros que correspondan. Con estos dos cambios, se define todo el código básico para decodificar instrucciones. Resta construir las definiciones de las instrucciones en sí. Para esto se agregó el archivo `arch/arm/isa/insts/mth.isa` en donde se define el código que definirá a las instrucciones. Este es el código que construye los strings *header output*, *decoder output*, *exec output* y *decode block*. Por último, se deben definir los templates para la construcción de esos strings. Para la mayoría de las instrucciones de *Mth* se utilizaron templates ya definidos, pero para otras fue necesario definir nuevos templates con parámetros especiales no contemplados en los casos generales. Además del código agregado, las instrucciones resuelven sus tareas llamando a funciones definidas

en el módulo de *Mth*, por lo que es necesario agregar las definiciones de las funciones que serán llamadas dentro del espacio de nombres creado para tal fin, en este caso `namespace mthInst`.

Completada la decodificación de las instrucciones, se crean instancias de la mismas. Al momento de ejecutar estas instrucciones, se llamará al módulo *Mth* que deberá ser parte del sistema. Sobre la clase `System` dentro de `sim(sim/system.cc)` se agregó el código necesario para que esta clase contenga una referencia al módulo *Mth* y una función para su inicialización. Esta última no es llamada desde el constructor de `system`, porque para inicializar el módulo *Mth* se debe conocer cuántos procesadores hay en el sistema y de qué tipo son, datos que al momento de crear la instancia de `system` son desconocidos. Por esta razón, la inicialización de *Mth* se realiza desde una instrucción (**Init**) una vez que el sistema se encuentra inicializado.

5.4 Funcionamiento del módulo *Mth*

Desde *Mth* es necesario administrar el contexto de ejecución de un core, en `gem5` esta tarea se puede realizar desde tres interfaces distintas dependiendo desde qué lugar de la infraestructura de `gem5` se quiera acceder. La figura 5.1 esquematiza esta relación.

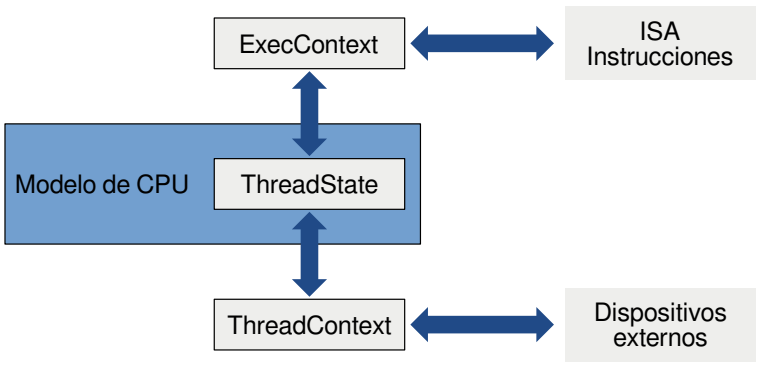


Figura 5.1: Relación entre las diferentes interfaces para acceder al estado de un core.

- `ThreadState`: Mantiene el estado general de un thread para cualquier modelo de CPU. Incluye punteros a los procesos que ejecuta el thread (en modo SE), puertos a memoria, información sobre los eventos *quiesce* (asociados a las seudoinstrucciones) y una considerable cantidad de registros de información estadística.

- `ExecContext`: Es una clase abstracta que provee la interfaz que utiliza el ISA para acceder al estado del CPU de un modelo dado. Cuando se resuelve una instrucción, independientemente del modelo de CPU, se debe conocer que índice dentro del banco de registros físicos ocupa cada uno de los operandos de la instrucción. Esta clase facilita este trabajo, ya que las funciones para leer y escribir registros utilizan como índices el número del operando de la instrucción a resolver. Además, es posible conocer el registro de la arquitectura al que se hace referencia por medio de una tabla de índices.

- `ThreadContext`: Es la interfaz externa del estado de un thread para cualquier componente fuera del core. Provee funciones para acceder al estado del thread, tanto para modificar registros de la arquitectura, como cambiar el estado de un thread o incluso obtener información estadística. A diferencia del caso anterior, esta clase provee una interfaz para acceder al thread desde el punto de vista de la arquitectura. Mientras que `ExecContext` provee una interfaz que debe ser implementada para que el ISA pueda acceder al estado que requiera, con el solo fin de resolver instrucciones.

Si bien *Mth* implementa el funcionamiento de nuevas instrucciones, la interfaz `ExecContext` está pensada para resolver operaciones entre registros, abstrayendo la complejidad del modelo de CPU y permitiendo acceder al estado del core desde el punto de vista de cómo resolver una instrucción. Por otro lado, la interfaz `ThreadContext` provee una forma de acceder al estado arquitectural, es decir, deja visible los registros que efectivamente provee la arquitectura sin considerar el modelo de CPU que se tenga por debajo. Toda la implementación de *Mth* utilizará esta última interfaz, ya que desde *Mth* es necesario poder acceder al estado del core desde la visión de un contexto de ejecución.

El módulo *Mth* contendrá la siguiente información, para su implementación sobre el simulador.

- *Puntero al Sistema*: Puntero de retorno al sistema que creó la instancia de *Mth*.
- *Arreglo de cores*: Por cada core se almacenará la siguiente información
  - *Puntero al ThreadContext*: Permite acceder al estado del core.
  - *Contexto duplicado*: Permite implementar el banco de registros espejo.
  - *Core libre*: Indica si el core está libre o no, es decir, si ejecuta un thread lanzado por *Mth*.
  - *Registros de sincronización*: Almacena el estado de sus registros de sincronización de 1bit.

Adicionalmente el módulo contendrá otro tipo de información no funcional, relevante para debug y experimentación, como por ejemplo un flag que indique si el sistema fue detenido por una interrupción o el modelo de CPU que se está utilizando.

5.4.1 Inicialización del módulo *Mth*

El proceso de inicialización de *Mth* se realiza al ejecutar una instrucción destinada a tal fin. Éste crea el módulo *Mth*, busca todos los cores que tenga el sistema y crea todas las estructuras de datos asociadas. Indica los cores como disponibles, a excepción del actual, e inicializa los contextos de ejecución del banco de registros espejo y los bits de sincronización. Además, inicializa información de debug y control.

Este proceso presenta una problemática particular, el módulo *Mth* está aislado de la arquitectura como decisión de diseño. Su información es almacenada de forma externa y utilizada emulando como si fuera parte del estado del core. Esto presenta un problema a la hora de generar un checkpoint. El módulo *Mth* no es almacenado por el checkpoint, ya que solo guarda un estado desde el punto de vista de la arquitectura (sin importar el modelo de CPU).

Cuando un checkpoint es restaurado, si éste se encontraba ejecutando instrucciones de *Mth*, las mismas generarán un error al no encontrar una instancia del módulo en cuestión. Incluso, si la instancia existiera, el estado de los registros espejo y los registros de sincronización se habría perdido.

Una solución posible, correspondería a rediseñar el módulo *Mth* como parte de la definición de la arquitectura. También es posible que el módulo *Mth* sea almacenado de forma independiente respetando parte de la arquitectura actual del simulador. Cualquiera sea el caso, no fue necesario implementar una solución, ya que en los experimentos realizados, se crearon checkpoints antes de lanzar los programas que utilizaban *Mth*.

5.4.2 Detalles de implementación de instrucciones en `gem5`

Si bien la definición de las instrucciones aparenta ser independiente del modelo de CPU utilizado, se deben definir características a las instrucciones, para que dependiendo del modelo de CPU, éste pueda suponer su comportamiento e interacción con otras instrucciones. Un ejemplo de ello son las instrucciones que una

vez ejecutadas no pueda volverse hacia atrás. Este tipo de instrucciones no pueden ejecutarse especulativamente. Un procesador que ejecute fuera de orden debe conocer esta propiedad antes de intentar ejecutar una instrucción.

Las instrucciones de *Mth* fueron implementadas agregando los siguientes *flags*:

- `Non Speculative`: No puede ser ejecutada de forma especulativa.
- `Squash After`: Resuelve todas las acciones pendientes luego de ejecutar la instrucción (no se ejecutarán nuevas instrucciones).
- `Serialize`: Serializa el pipeline, ejecuta todas las acciones pendientes antes de ejecutar la instrucción.

Esta forma de implementar las instrucciones obliga al modelo de CPU a operar de forma segura con las instrucciones, resolviendo las mismas de manera bloqueante sobre el pipeline.

Capítulo 6

Resultados sobre factibilidad de *Mth*

Este capítulo tiene como objetivo presentar los resultados de la exploración inicial acerca de la factibilidad de utilizar *Mth* en distintos tipos de algoritmos. Para ello se han realizado simulaciones utilizando la infraestructura del simulador `gem5` con la aplicación de las modificaciones propuestas y mencionadas en los capítulos anteriores. Se busca dar respuesta a la pregunta de la viabilidad técnica y el potencial de *Mth*.

6.1 Introducción

Realizar experimentos sobre un sistema que busca extender una arquitectura, requiere tener control de todos los aspectos involucrados en la planificación, medición y ejecución de estos. El principal desafío es construir programas que resulten adecuados para mostrar el potencial de *Mth*. Como fue explicado en la sección 3.4, la propuesta requiere modificaciones a lo largo de todo un programa para poder ser utilizada en su máximo potencial. Es por esta razón que se decidió utilizar algoritmos bien conocidos y con diferentes características a modo de micro-benchmarks. Las soluciones paralelas de estos problemas están bien definidas y dejan en claro el código que es ejecutado para resolver cada uno.

La experimentación realizada sobre la propuesta *Mth* fue simulada en `gem5` sobre la arquitectura `ARM`. Todo el código de los experimentos fue compilado con `arm-linux-gnueabihf` utilizando configuraciones de optimización agresivas.

El simulador fue configurado para utilizar, en todos los casos, el modelo de simulación detallado tanto simulando un procesador *Out-of-order* (`arm_detailed`, `Cortex-A9`) como un procesador *in-order* (`minor`, `Cortex-A8`) [7]. Ambos modelos han sido utilizados en todos los experimentos.

6.2 Aplicaciones

Como casos de estudio fueron utilizados los siguientes algoritmos:

- **HSL Filter**: Representa el caso del programa trivialmente paralelizable. Esta aplicación consiste en transformar una imagen de pixeles RGB al espacio de color HSL. La operatoria consiste en una serie de comparaciones sucesivas que impide el uso eficiente de instrucciones vectoriales. La estrategia de paralelización consiste en dividir los pixeles a procesar en la cantidad de procesadores disponibles.

- **FFT Radix2**: Consiste en la implementación recursiva in-place del algoritmo de Cooley-Tukey para calcular FFT (Fast Fourier Transform). El algoritmo divide en cada paso recursivo el problema en dos partes independientes. En el segundo llamado recursivo, es creada una nueva tarea que soluciona

la mitad del problema. Los subsiguientes llamados recursivos son resueltos por cada procesador de forma independiente.

- **LU decomposition**: Este algoritmo descompone una matriz en dos matrices L y U. En la figura 6.1 se describe el código secuencial utilizado para resolver la descomposición. En cada paso del ciclo (k), se resuelve una submatriz cuadrada que comienza en la fila k+1 y termina en la última identificada como n. Los dos ciclos internos se encargan de toda la operatoria, que se realiza independientemente para cada fila. Luego, otro ciclo se encarga de copiar los resultados a la matriz U. La paralelización en este caso consiste en resolver con dos threads los cálculos sobre las filas de los ciclos internos. La copia de los resultados se realiza en forma serial.

- **Dantzig**: Calcula todos los caminos mínimos sobre un grafo. En cada iteración se agrega un nuevo nodo al grafo, los dos primeros ciclos calculan todos los caminos desde el nuevo nodo a todos los nodos del grafo actual y viceversa. El tercer ciclo se ocupa de mejorar todos los caminos entre todos los nodos a partir del agregado del nuevo nodo. La estrategia de paralelización consiste en ejecutar en dos procesadores los dos primeros ciclos de forma independiente. Luego, el tercer ciclo es particionado también en dos procesadores.

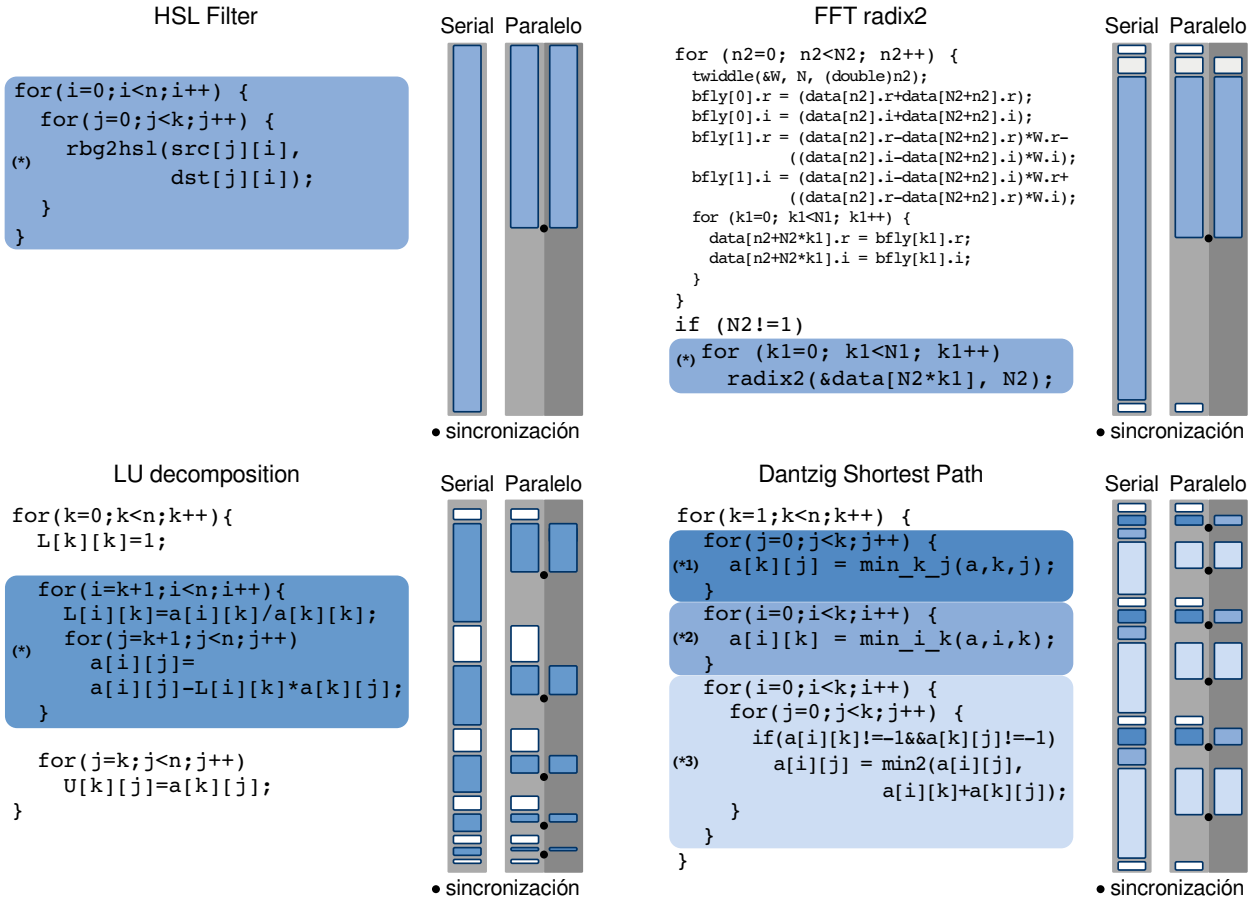


Figura 6.1: Código referencia de los algoritmos utilizados en los experimentos.

En la figura 6.1 se puede identificar el código básico de los algoritmos utilizados para los experimentos. Los códigos presentados corresponden a su versión secuencial. Se indica en color la sección afectada por la paralelización en todos los casos. A la derecha de cada código se ilustra qué parte del programa es ejecutada de forma secuencial y cuál de forma paralela. Los bloques de programa fueron alineados junto al caso serie,

para así ayudar a identificar a qué parte del código corresponden. Además, al finalizar cada bloque se marca el punto de sincronización.

Para los primeros dos ejemplos, la sincronización se realiza al final del experimento, mientras que en los restantes ejemplos la sincronización se realiza por cada iteración del ciclo principal, incluso más de una vez por ciclo.

La estrategia de paralelización no incluye ningún tipo de preparación especial para *Mth*. Las modificaciones a los códigos presentados para poder soportar *Mth* corresponden solamente a la creación de threads y sincronización. En todos los casos casos, se agregan las instrucciones especiales tanto para indicar el nuevo PC como para cargar registros de sincronización.

6.3 Resultados y discusión

A continuación se presentarán dos conjuntos de resultados, ambos estudiarán el rendimiento en términos del speedup logrado por *Mth* en comparación al caso secuencial. En el primer conjunto, se muestran los resultados obtenidos por las distintas aplicaciones estudiadas. En el segundo conjunto se presentan experimentos acerca de la relación entre la sincronización y el trabajo a realizar, se plantea un algoritmo que reparte trabajo entre threads e impone restricciones de sincronización. En ambos conjuntos se utilizó como referencia el rendimiento obtenido al utilizar otros frameworks de paralelización.

6.3.1 Rendimiento

La figura 6.2 compara el speedup de `OpenMP` y `Pthreads` con *Mth* a medida que se incrementa el tamaño del problema. Esta comparación se realiza sobre el modelo de procesador in-order (InO) y out-of-order (OOO).

Resulta fundamental prestar especial atención al volumen de datos a procesar en cada experimento, ya que se está aplicando paralelismo con entradas extremadamente pequeñas. Por ejemplo, en el caso del filtro HSL, una imagen cuadrada de 20 filas corresponde a 1200 bytes de datos. Para el caso de FFT, el tamaño indicado en la figura es cantidad de valores `double` en el vector a procesar, mientras que el caso de LU representa una matriz cuadrada también de valores `double`. El caso de camino mínimo por otro lado, el tamaño indicado corresponde a la cantidad de nodos del grafo, por ejemplo, para el caso de 20 nodos, la matriz asociada que representa el grafo tiene 400 bytes.

En todos los casos, se obtienen para *Mth* speedups cercanos al ideal, incluso para tamaños pequeños. Se logra un speedup ideal cuando este corresponde la cantidad de cores. Para el caso del filtro HSL, se obtiene un speedup ideal prácticamente para todo el rango de entrada del experimento. En el caso de utilizar cuatro núcleos, se alcanza más del 80 % de eficiencia. El algoritmo Radix2 para FFT, excede el 75 % de eficiencia solamente con 64 valores en el vector de muestras. Este algoritmo presenta un paso secuencial muy importante al principio del código, la paralización se realiza recién en el segundo llamado recursivo. Esto impide alcanzar el speedup ideal para el rango de tamaños del experimento. Sin embargo, para el caso de 256 elementos, que representa un tamaño en memoria de 2KB, el speedup alcanzado es de $1{,}7$X. La descomposición LU por otro lado, requiere un tamaño mayor para alcanzar el 75 % de eficiencia. En los casos de tamaño mayor a 100×100 (78KB de datos) se alcanza 2X speedup, tomando para esto $8{,}56 \times 10^8$ ciclos en out-of-order y $2{,}4 \times 10^9$ ciclos en in-order para el caso secuencial. El algoritmo de Dantzig presenta dependencia de datos dentro del ciclo principal, en cada iteración del mismo se deben resolver dos pasos, uno dependiente del otro. Además, las iteraciones del ciclo principal dependen entre sí. Para un tamaño de entrada mayor a 40 nodos ($1{,}56$KB de datos), se alcanza un speedup mayor a $1{,}7$X, incluso más del 80 % de eficiencia para el caso out-of-order. En el caso in-order, por otro lado, se obtiene al menos $1{,}9$X de speedup y un 95 % de eficiencia.

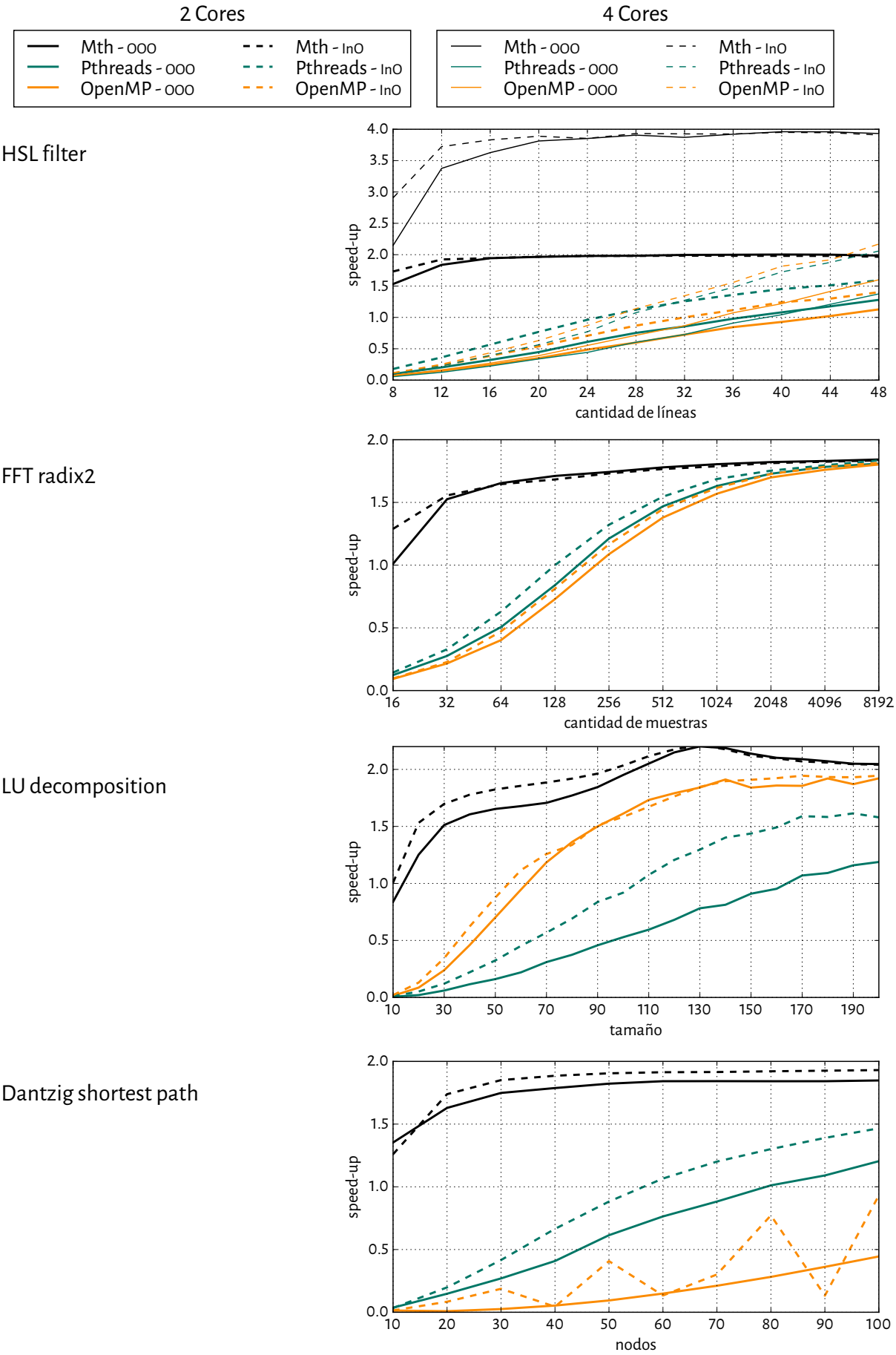


Figura 6.2: Resultados de rendimiento de *Mth*. En todos los casos se obtiene notable rendimiento, incluso para instancias pequeñas.

6.3.2 Sincronización

En la figura 6.3 se muestra la versión OpenMP de un programa que calcula el md5 de un arreglo de valores. El código divide el arreglo en dos secciones que se ejecutan de forma paralela, ambas resuelven el cálculo para $L/2$ valores. Además, este procedimiento se realiza S veces. Es decir, L regula la cantidad de trabajo que realizará cada thread, mientras que S incrementa la cantidad total de trabajo que debe ser realizado. Con valores de S grandes y L chicos, la operatoria resultante corresponderá con una gran cantidad de trabajo a realizar en paralelo, pero con muchos puntos de sincronización para realizarlo. Por otro lado, si se aumenta L y se reduce S, se realizará la misma cantidad de trabajo, pero se reducirá la cantidad de puntos de sincronización. El costo de inicialización entonces, impactará en menor medida en cuanto mayor sea S, es decir, la cantidad de trabajo a realizar.

```
for(istep=0;istep<S;istep++) {
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      for(i=0;i<L/2;i++)
        getMd5(data+n*i);
    }
    #pragma omp section
    {
      for(i=L/2;i<L;i++)
        getMd5(data+n*i);
    }
  }
}
```

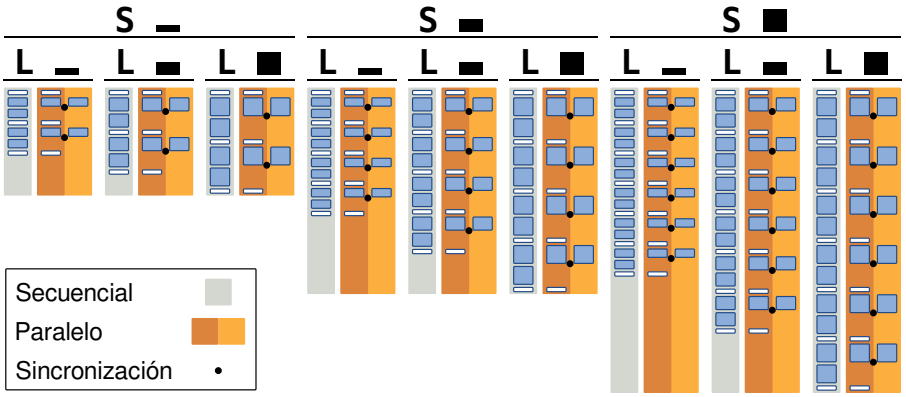


Figura 6.3: Código referencia para el experimento de sincronización. A la derecha se ilustra la cantidad de trabajo para las distintas configuraciones y los puntos de sincronización de las mismas.

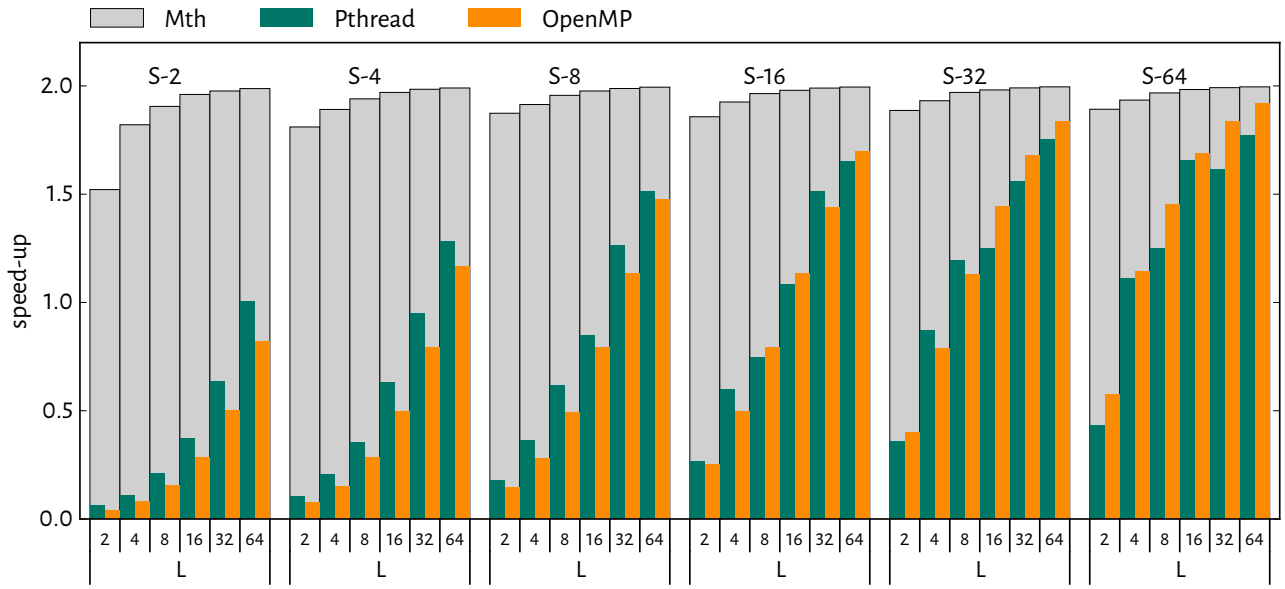


Figura 6.4: Impacto de la inicialización y sincronización usando dos cores: S grande implica más trabajo, L chico mayor sincronización.

La figura 6.4 presenta los resultados para las diferentes configuraciones. *Mth* presenta un speedup superior a $1{,}8$X en la mayoría de los casos. Tanto Pthreads como OpenMP, pueden alcanzar un speedup de $1{,}8$X solamente en casos donde S es suficientemente grande. Si bien los tiempos de sincronización y de

inicialización en *Mth* son muy reducidos, se puede ver como afectan el rendimiento para casos muy chicos. Este impacto se ve reflejado para el caso más chico de la figura de L=2 y S=2 (dos iteraciones por cada ciclo). Para el resto de los experimentos, el speedup presentado es superior a $1{,}8$X. El rendimiento presentado para OpenMP y Pthreads muestra una importante pérdida de rendimiento en la mayoría de los casos, obteniendo una leve mejora en speedup para S=64.

Capítulo 7

Resultados sobre una aplicación concreta: SQLite

Habiendo realizado una evaluación de la factibilidad técnica de *Mth* por medio de la implementación de una serie de micro-benchmars, el siguiente objetivo que naturalmente surge es la evaluación de la propuesta en un contexto de una aplicación real y de interés. Este capítulo presenta los resultados obtenidos de paralelizar la planificación de transacciones sobre el motor de bases de datos SQLite. Luego de introducir los fundamentos del experimento, en términos de limitaciones de las DBMS actuales, se menciona una serie de artículos relacionados con la optimización en bases de datos. Se detalla también la implementación del motor de bases de datos las modificaciones provistas. Por último, se plantea la metodología experimental y la discución de los resultados.

7.1 Introducción

Los motores de bases de datos (Database Management Systems - DBMS) actuales tienen fuertes restricciones para obtener un completo beneficio de la creciente cantidad de cores en el procesamiento de consultas en paralelo. La disponibilidad de cores para este tipo de sistemas permite resolver mayor cantidad de consultas, pero no resolver cada una más rápidamente. Su diseño se basa en la administración de los accesos de E/S y la ejecución de consultas se realiza secuencialmente en cores independientes [39, 42]. En este sentido, se han ido planteando diferentes propuestas de cambios de diseño para enfrentar estas limitaciones. Estos avances se encuentran tanto en productos comerciales como, más tímidamente, en productos de código abierto. Actualmente, los motores de bases de datos como `IBM DB2`, `Microsoft SQLServer` y `Oracle` incluyen, de forma limitada, soporte para la generación y ejecución de planificaciones paralelas. Por otro lado, los desarrolladores de `PostgreSQL`, una DBMS de código abierto, se encuentran implementando soporte para planificaciones paralelas, teniendo al momento soporte parcial sobre algunas operaciones. Sin embargo, mientras el diseño de las DBMS se adapta al soporte de ejecución paralela, su foco se pone sobre la resolución de consultas con una gran carga de trabajo computacional, buscando superar el overhead introducido por la paralelización.

Medir el rendimiento de un motor de bases de datos es una tarea compleja, ya que implica controlar una gran cantidad de variables. Se deben considerar mínimamente las consultas que se van a realizar, sobre qué tablas operan, qué índices tienen e, incluso, el volumen de información almacenada por la base de datos. Es posible construir benchmarks que estudien un caso particular, o una optimización específica, pero su alcance será limitado. Con el objetivo de definir y estandarizar benchmarks sobre bases de datos y procesamiento de transacciones se creó *Transaction Processing Performance Council* (TPC). TPC define una serie de benchmarks orientados a transacciones del mundo empresarial, incluyendo cuestiones como control de

inventario, reservas aeronáuticas u operaciones bancarias. Más adelante se utilizarán consultas derivadas de uno de estos benchmarks denominado TPC-H *Decision Support Benchmark*.

El soporte provisto por las DBMS es utilizado por una gran cantidad de aplicaciones en todo el mundo todos los días. Adicionalmente, una gran cantidad de aplicaciones utilizan DBMSs embebidas para gestionar la información. Estos sistemas son diseñados para ser fácilmente puestos en funcionamiento, sin configuración ni administración particular por parte del usuario final. Incluso, hay aplicaciones ejecutadas en sistemas como smartphones que también hacen un uso muy fuerte de este tipo de servicios.

Todo hace indicar que la utilización de smartphones continúe creciendo, en este sentido el rol de las bases de datos embebidas cobrará una gran importancia como componente clave de software. Reproductores de música, navegadores, sistemas multimedia, clientes de redes sociales son ejemplos de aplicaciones utilizadas todos los días por millones de usuarios. Cada una tiene integradada una instancia de una base de datos embebida para almacenar e indexar datos de las aplicaciones. El desarrollo de procesadores multicore y de nuevas aplicaciones para sistemas embebidos como smartphones ponen especial énfasis en la necesidad de tomar ventaja de los recursos computaciones disponibles, pero además se lo deben lograr teniendo especial cuidado en el consumo de energía.
Para resolver una consulta SQL, cualquier motor de DBMS construye una planificación (*query planning*) siguiendo los pasos que se detallan a continuación:

i) *Parseo y Traducción*: La consulta es transformada a una expresión de álgebra relacional, que consiste en una representación formal de las operaciones involucradas en la consulta.

ii) *Optimización*: La expresión de álgebra relacional puede ser evaluada siguiendo diferentes estrategias. La elección de la estrategia de resolución puede impactar notablemente el tiempo de resolución. La salida de este paso es una expresión anotada conocida como *execution plan*. Si la DBMS soporta planificaciones paralelas, este paso debe incluir el código necesario para utilizar más de un core.

iii) *Evaluación*: El *execution plan* es ejecutado y el resultado es retornado al usuario.

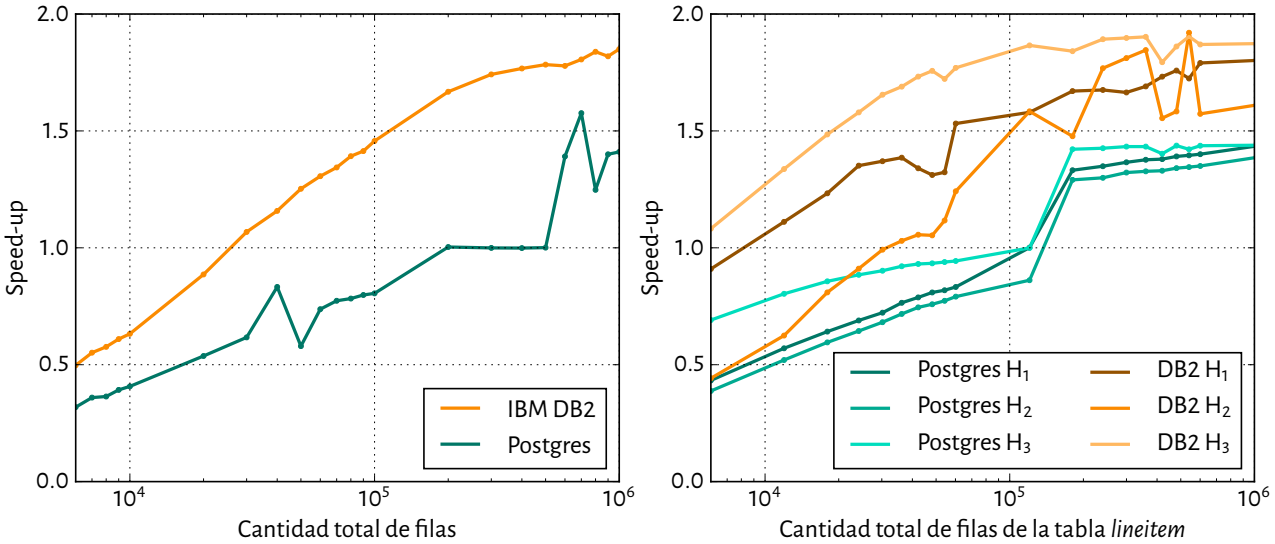
Una planificación consiste en la secuencia de pasos necesarios para resolver una consulta. Usualmente es escrita en términos de un lenguaje intermedio que puede ser ejecutado nativamente por el DBMS. En algunos casos, la ejecución de las consultas es aislada por medio de una máquina virtual dentro del motor de bases de datos. Las instrucciones dentro de una planificación corresponden a los pasos para iterar tablas por medio de cursores, comparar condiciones sobre tuplas, agregar datos, entre otras. Estas instrucciones utilizan registros temporales en los cuales almacenar resultados parciales de las operaciones, para luego generar los resultados que serán devueltos al usuario final o aplicación que lo requirió.

Para ejecutar una consulta en paralelo, la planificación debe dividir el problema, posiblemente junto con los datos de entrada (tablas), y asignarlos a los cores disponibles. En general, resulta necesario combinar los resultados parciales en una etapa intermedia (conocido como *reducción*), para luego continuar con la ejecución o generar los resultados finales de la consulta.

Tanto dividir la consulta, como la sincronización entre las etapas en la ejecución paralela, introduce overhead. Si el trabajo a realizar por la consulta es muy limitado, el overhead introducido por la paralelización puede superar al tiempo total de ejecución secuencial. El DBMS debe estimar este overhead y decidir si se optará por ejecutar una planificación paralela o secuencial. Típicamente, el tamaño de consulta necesario para superar el overhead es mayor que el usual esperado en consultas a bases de datos sobre dispositivos móviles, impidiendo así la adopción de la ejecución paralela en DBMS embebidas.

A continuación se muestra cómo el uso de *Mth* puede generar un enfoque efectivo en el uso de procesadores multicores, incluso cuando la cantidad de trabajo a realizar sea muy pequeña. De obtener resultados positivos, se permitirá el uso de recursos paralelos para consultas que están fuera de alcance del actual soporte de motores de bases de datos.

La figura 7.1 presenta resultados de speedup para la ejecución de consultas sobre IBM DB2 y PostgreSQL. Ambos motores poseen soporte para resolver planificación de consultas en paralelo. Para el análisis se forzó la ejecución de las consultas tanto en un procesador como en dos procesadores a fin de calcular el speedup. Las consultas utilizadas se detallan en la sección 7.2.4. La plataforma utilizada para estos experimentos fue un Intel i7-920 con 18 GB de memoria.



(a): Consulta S_2. Realiza un *filter* y *join* sobre un esquema de dos tablas.

(b): Consulta H_1, H_2 y H_3. Basadas en la consulta Q6 de TPC-H.

Figura 7.1: Speedup sobre la ejecución secuencial para distintas consultas definidas en la sección 7.2.4. Sobre el eje se indica el total de filas de cada tabla.

La figura 7.1a presenta el speedup de resolver una consulta sobre un esquema de dos tablas relacionadas. Para DB2, se requiere más de 5000 filas para compensar el overhead introducido por la paralelización, mientras que PostgreSQL se requiere de 2×10^5 filas para alcanzar el mismo objetivo. DB2 requiere más de 10^5 filas para superar el $75\,\%$ de eficiencia, mientras que PostgreSQL muestra un comportamiento inestable cerca de 5×10^5 filas, pero manteniendo un rendimiento pobre.

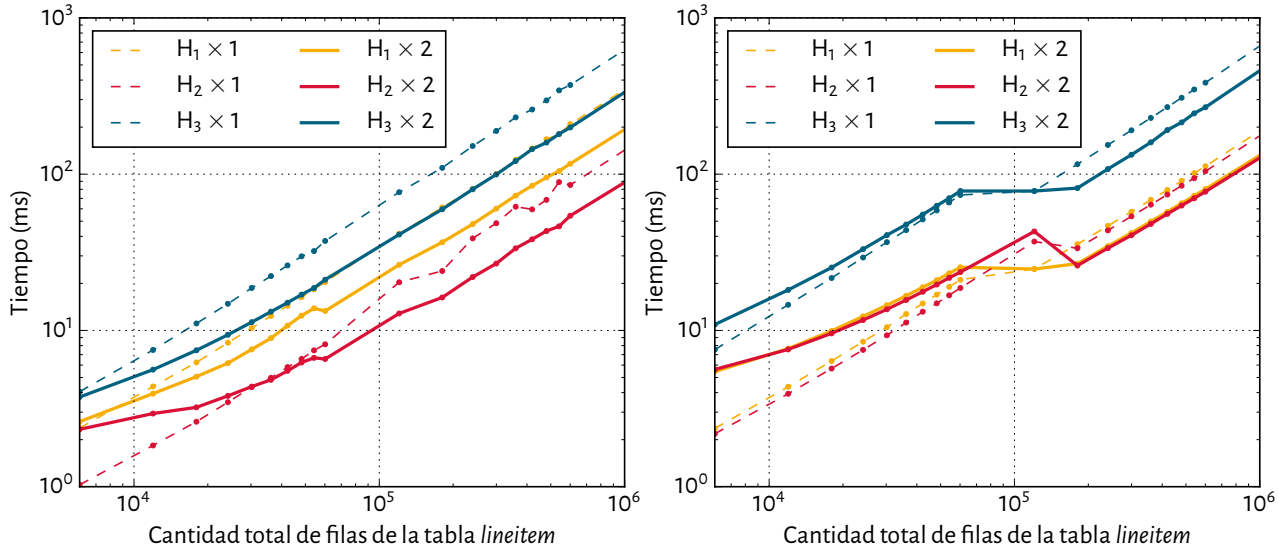
Los resultados para las tres consultas basadas en Q6 del benchmark TPC-H son ilustrados en la figura 7.1b. Las mismas son detalladas más adelante en la sección 7.2.4. El tamaño del trabajo de cada consulta se modifica incrementando la cantidad de filas en la tabla `lineitem` del esquema.

El speedup obtenido por PostgreSQL se mantiene debajo de uno para todas las consultas hasta alcanzar 2×10^5 filas, revelando un salto a $1{,}4$X y manteniéndose en ese valor para el resto del rango de tamaños estudiado.

Por otro lado, IBM DB2 presenta un comportamiento diferente para cada consulta, H_3 es la consulta que más operaciones por fila necesita para ser resuelta, por lo que solamente requiere de 2×10^4 filas para superar $1{,}5$X de speedup, mientras que H_2 requiere de 6×10^4 y H_1 de 1×10^5.

H_3 alcanza cerca de $1{,}8$X de speedup con 1×10^5 filas, mientras que H_2 presenta el menor speedup en todo el rango. El caso de H_3, presenta un comportamiento inestable en la última parte del rango estudiado.

Si bien el speedup da cuenta de la eficiencia, permitiendo reconocer cuándo resulta efectivo solucionar consultas en paralelo, comparar los tiempos de ejecución también provee información útil para el análisis. La figura 7.2 muestra la evolución del tiempo de ejecución en función de la cantidad de filas para cada consulta H, tanto para IBM DB2 (7.2a), como para PostgreSQL (7.2b). En cada figura se muestra el tiempo requerido para resolver la consulta en serie y en paralelo (utilizando dos procesadores). Entre el comienzo del rango estudiado y 5×10^4 filas, se presenta una notable diferencia entre los tiempos IBM DB2 a favor PostgreSQL para todas las consultas.



(a): IBM DB2 (b): PostgreSQL

Figura 7.2: Tiempos de ejecución secuenciales y paralelos para las mismas consultas de la figura 7.1b (basado en Q6 del benchmark TPC-H).

Sin embargo, PostgreSQL en el rango 5×10^4 a 2×10^5 filas, muestra un tiempo de ejecución constante, que impacta positivamente en el resto de los tiempos, obteniendo tiempos de ejecución equivalentes a los obtenidos por IBM DB2.

Desde el punto de vista del comportamiento en paralelo, los tiempos obtenidos son consistentes con el speedup presentado en la figura 7.1: durante el rango de tamaños inicial, PostgreSQL muestra una marcada pérdida de paralelismo con respecto a la versión secuencial, mientras que IBM DB2 presenta una mejora en su rendimiento paralelo. Solamente cuando el número de filas supera 2×10^5, ambos DBMSs alcanzan un rendimiento interesante, excepto para el caso de H_2 sobre IBM DB2, que exhibe algunas fluctuaciones favorables.

En lo que sigue de este capítulo se presentará cómo utilizar *Mth* para soportar la ejecución paralela de consultas, permitiendo el uso eficiente de múltiples procesadores, incluso para consultas que requieren poco trabajo.

7.1.1 Trabajo relacionado

Utilizar recursos de cómputo paralelo por parte de DBMS tiene una rica y larga historia, especialmente desde la llegada de plataformas multicores para el cómputo estándar. A continuación, se listan algunos avances relacionados con la propuesta.

Acker et al. [1] presentan una forma de encapsular paralelismo en la ejecución de consultas en bases de datos relacionales, por medio de un operador opaco que es incluido durante la planificación. Suponen que las planificaciones de las consultas son generadas basándose en primitivas estándar como `open()`, `next()`, y `close()`, ocultando entonces los detalles de implementación en el operador propuesto. La planificación de la consulta es analizada y transformada antes de ser ejecutada; un mecanismo de balance de carga es implementado para mantener los cores con cargas iguales de trabajo. Sus resultados muestran que solamente las consultas más complejas y grandes pueden obtener beneficios de la ejecución en paralelo.

Mühlbauer et al. [31] analizan el impacto de la diversidad de procesadores sobre las principales operaciones en bases de datos, concluyen que los detalles sobre los recursos computacionales deben ser expuestos directamente al DBMS para que éste tome sus propias decisiones sin intervención del sistema operativo. Centrándose en la ejecución simultánea de varias consultas, Zhang et al. [50] proponen un scheduler

de consultas que obtiene una mejora global en la eficiencia del sistema.

Kim et al. [22] proponen utilizar el hardware de discos SSD como parte de un sistema de procesamiento que asista en la resolución de consultas. A pesar de tener menos poder de cómputo, el procesamiento de los discos es más cercano a los datos y esto podría resulta más eficiente que enviar datos a memoria para ser procesados. Para los experimentos realizados sobre la propuesta, se utilizó una serie de consultas simplificadas basadas en el TPC-H benchmark [48], combinaciones de scan y scan-join. Salami et al. [38] diseñan e implementan DBMS basados en FPGA utilizando componentes estándar para obtener una solución eficiente y específica en términos de ahorro de energía en comparación con un DBMS estándar.

7.2 SQLite

Para agregar soporte para la planificación de consultas en paralelo, se requirió modificar sustancialmente SQLite. A pesar de aplicar este conjunto de cambios sobre un DBMS particular, este enfoque es general y puede ser aplicado sobre cualquier DBMS.

SQLite [47] es un DBMS totalmente contenido en una biblioteca, en general se puede encontrar embebido dentro de una aplicación y oculto para el usuario final. Es ampliamente utilizado por navegadores, sistemas operativos y sistemas embebidos.

Como SQLite es un motor de bases de datos embebido, éste no hace uso de un proceso independiente como servidor, escribe y lee directamente sobre archivos en disco. Está diseñado para tener una huella de memoria muy pequeña y su código es abierto.

Las razones para utilizar SQLite como motor de DBMS sobre el que implementar modificaciones son las siguientes:

- *Código abierto y simple*: Como SQLite no posee ningún tipo de soporte para paralelización, se requiere de grandes cambios al motor. Comenzar por un código base muy simple colabora en focalizar los esfuerzos en las modificaciones requeridas. Disponer del código abierto en este sentido resulta fundamental.

- *Diseñado para ser eficiente en memoria*: La propuesta de *Mth* puede soportar la ejecución en paralelo de pequeños fragmentos de código y acelerar la aplicación con una carga de trabajo limitada. El uso reducido de memoria es un aspecto clave a considerar y que el diseño de SQLite tiene en cuenta.

- *Ampliamente utilizado y en activo desarrollo*: La aplicación tiene soporte de una gran comunidad de desarrolladores que continuamente la mejoran. Además, es ampliamente utilizada en un gran número de plataformas y sistemas operativos.

7.2.1 Arquitectura de SQLite

La Arquitectura de software de SQLite es ilustrada en la figura 7.3 [2]. Se separa en tres módulos funcionales independientes descriptos a continuación:

1. *Núcleo*: Incluye la interfaz necesaria para la comunicación con aplicaciones, para acceder a estructuras de datos e interactuar con la biblioteca. También define la máquina virtual que ejecuta el código de las consultas, denominada *virtual database engine* (VDBE), que es orientada a registros y resuelve instrucciones en *bytecode*.

2. *Compilador de SQL*: La resolución de una consulta comienza con el *tokenizer* y el *parser*. Estos dos módulos transforman una consulta SQL en texto plano, a una estructura de datos que puede ser interpretada por el generador de código. El generador de código transforma el árbol sintáctico recibido a un programa en *bytecode* en el lenguaje específico de SQLite. Por último éste programa puede ser ejecutado en la máquina virtual.

3. *Backend*: La base de datos es almacenada en disco utilizando una implementación específica de B-tree. Cada tabla se almacena en un B-tree separado, mientras que todos los B-tree de la misma base de datos son almacenados en un mismo archivo. El módulo B-tree solicita información en páginas de tamaño fijo al `page cache`. Éste lee, escribe y salva temporariamente las páginas, mientras se realiza una transacción. La interfaz con el sistema operativo aísla las operaciones de acceso a archivos, permitiendo la portabilidad a través de diferentes plataformas.

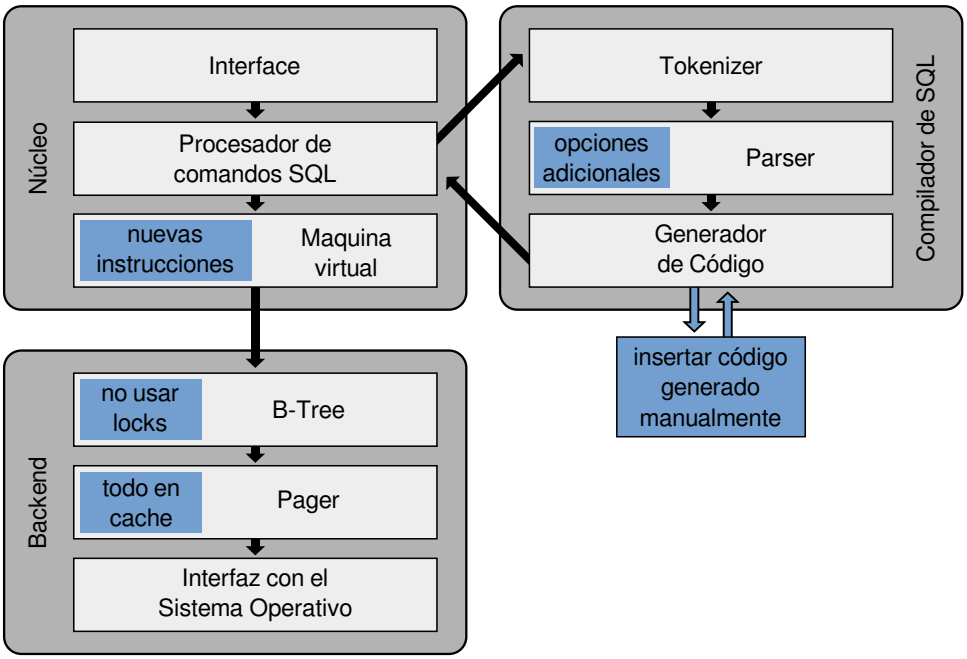


Figura 7.3: Arquitectura de SQLite y componentes principales. Se indican las modificaciones realizadas para soportar ejecución de consultas en paralelo.

7.2.2 Panificación de consultas para ejecución en paralelo

La figura 7.3 muestra los componentes de SQLite que fueron modificados. La implementación propuesta busca alterar la mínima cantidad de componentes, evitando rediseñar toda la arquitectura de la aplicación para soportar eficientemente varios procesadores.

Las modificaciones realizadas se pueden dividir en dos grupos:

Modificaciones funcionales
Nuevas funcionalidades para permitir ejecutar más de una máquina virtual simultáneamente.

- Instrucciones nuevas: La máquina virtual soporta nuevas instrucciones para paralelizar código.
- Opciones adicionales: El parser reconoce opciones adicionales que permiten por ejemplo, indicar la cantidad de procesadores para resolver una consulta.
- Impedir el *lock* de tablas: Nuevo esquema de locking que permite que más de un cursor en una consulta pueda leer la misma tabla, para esto se debieron modificar muchas funciones sobre el módulo `B-Tree`.

Modificaciones de bypass
Código adicional que resuelve funcionalidades particulares de experimentos. En el caso real, estas funcionalidades deberían ser reimplementadas por completo.

- Mantener las tablas en la cache de páginas: Para evitar el ruido generado por I/O se tomó como condición que todos los datos de la base de datos se encuentren en memoria. Para esto, se modificó el `Pager` para evitar desalojar páginas.

- Generación de código: Las consultas son pre-procesadas y el código generado por el compilador es remplazado por una versión paralela. En el caso real se debería reimplementar todo el generador de código.

Considerando estas modificaciones, la planificación de la consulta debe ser generada para utilizar múltiples procesadores simultáneamente. El proceso para resolver una consulta, consiste en abrir uno o más cursores, utilizarlos para acceder e iterar las tablas involucradas en la consulta, y guardar los resultados de la misma. La última parte del proceso implica cerrar los cursores y retornar la respuesta al usuario.

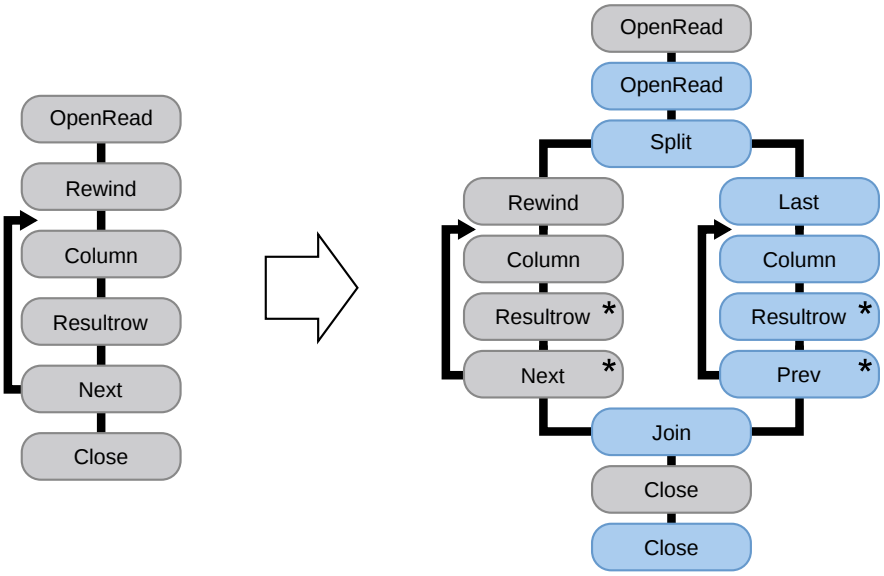


Figura 7.4: Comparación entre una planificación secuencial y una planificación paralela. Nuevas instrucciones son agregadas para controlar la colaboración entre dos máquinas virtuales que resuelven la consulta.

La figura 7.4 esquematiza las modificaciones necesarias para soportar ejecución en paralelo, para una planificación de una consulta simple. Dos instancias de máquinas virtuales son utilizadas para resolver la consulta, las instrucciones `Split` y `Join` fueron agregadas para crear y sincronizar el flujo de ejecución de la planificación. La primera de las máquinas virtuales, comienza recorriendo la tabla desde el comienzo, mientras que la segunda recorre la tabla comenzando por la última posición y en sentido opuesto. Esto sucede en la primer instrucción luego de split: la máquina virtual de la izquierda mueve su cursor al comienzo de la tabla, mientras que la otra mueve su cursor a la ultima fila de la tabla. Ambas terminarán de procesar la tabla de forma independiente cuando lleguen a la mitad del total de filas. El ciclo compuesto por las instrucciones `Column`, `Resulrow` y `Prev/Next`, se encargará de tomar un valor de la fila, retornar ese valor como respuesta y pasar a la siguiente fila. Ya sea, para la máquina virtual a derecha o a izquierda, el cursor se moverá en orden ascendente o descendente por medio de las instrucciones `Prev/Next`. La instrucción `Join` actúa como una barrera de sincronización, a la cual las dos máquinas virtuales deben llegar para poder continuar. La elección de las nuevas instrucciones y su semántica para construir planificaciones paralelas, se debe a la búsqueda en mantener la arquitectura de SQLite lo más simple posible. Además de articular su funcionamiento a las primitivas provistas por *Mth*.

El proceso para implementar un join en paralelo es más complejo, la figura 7.5 presenta un esquema ilustrativo del mismo. Dos cursores son utilizados en paralelo para recorrer la tabla A, esta es almacenada

usando una estructura de B-Tree. Para cada fila de la tabla A, se debe buscar en la tabla B un registro relacionado usando un índice, que es mantenido en memoria usando otra estructura de B-Tree. Una vez realizado el join, se evalúa una condición de filtro que determina si la fila debe ser descartada o almacenada como resultado.

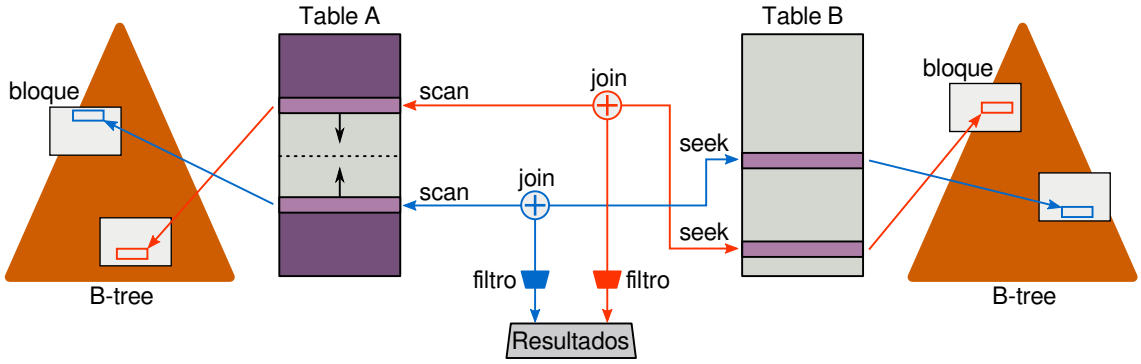


Figura 7.5: Resolución en paralelo de una consulta *join*. La tabla A es iterada usando dos cursores, cada uno accede a una fila en la tabla B por medio de un índice en memoria usando una estructura de B-Tree. Un filtro es evaluado determinando si el registro obtenido debe ser parte del resultado.

A continuación se listan las principales modificaciones al lenguaje de la máquina virtual de SQLite:

- `Split`: Nueva instrucción similar a un `fork`. Crea una nueva instancia de máquina virtual comenzando a ejecutar desde una instrucción dada. Ambas máquinas virtuales comparten variables en memoria.

- `Join`: Nueva instrucción, genera una barrera de sincronización entre las las máquinas virtuales y detiene a la instancia creada por `split`.

- `ResultRow`: Esta instrucción es modificada permitiendo que una sola de las máquinas virtuales ejecute la función *callback*.

- `Next` y `Prev`: Se agrega un contador que permite indicar cuando se alcanza una determinada fila. Permite determinar cuándo se debe terminar de iterar. En los experimentos este contador es inicializado en la mitad de la cantidad total de filas.

7.2.3 Modos de generar planificaciones

Para realizar los experimentos se utilizaron tres modos diferentes de generar planificaciones sobre SQLite:

- *Single core*: Corresponde al código original generado por SQLite, pero con modificaciones menores. La planificación original agrega instrucciones específicas para bloquear el acceso a tablas en uso. Este mecanismo no es utilizado cuando se ejecuta en paralelo. A fin de no interferir en la medición dando un sesgo a favor de ejecutar en paralelo, se remueven dichas instrucciones. Se debe tener en cuenta que remover estas instrucciones no genera ninguna alteración en el normal funcionamiento de SQLite, ya que no se utilizará concurrentemente otro thread que busque acceder a las mismas tablas.

- *Parallel w/Mth*: Es el código original extendido para ejecutar en paralelo. Este nuevo código contiene instrucciones especiales para administrar threads implementadas utilizando las primitivas de *Mth*.

- *Parallel w/Pthreads*: El mismo código extendido pero utilizando las directivas de Pthreads en lugar de las primitivas de *Mth*. Este caso es incluido a fin de comparar la ganancia de *Mth* en comparación con otro modelo de administración de threads. En el trabajo [16] se muestra que Pthreads resulta una buena referencia de comparación.

Es importante destacar que la interfaz para acceder a los resultados de una consulta opera como un iterador, es decir, el usuario debe llamar reiteradas veces la función `next` para obtener los datos de la consulta. La función encargada de esta tarea se denomina `callback`. La misma, fue modificada en todos los casos para no interferir con la ejecución de la consulta y almacenar los resultados en memoria sin interacción con I/O.

Además, se debe cambiar el código a ejecutar por las máquinas virtuales. El mismo fue compilado y optimizado por SQLite, durante este proceso se reservó un área de memoria para alojar al bytecode, junto con el banco de registros. Para ejecutar dos máquinas virtuales que hagan referencia al mismo código, éste deberá ser alterado. Como el código con el que se cuenta ya fue procesado, el área de memoria es insuficiente para alojar el código de dos máquinas. Por esta razón, se debe construir nuevamente todo el bytecode, incluyendo las referencias al nuevo banco de registros, ahora con espacio suficiente para las dos máquinas.

7.2.4 Consultas utilizadas

Las consultas utilizadas en los experimentos se basan en los siguientes esquemas:

i) *Simple (S)*: Dos tablas `Persons` y `Jobs`, relacionadas 1:N, `Person`→`Jobs`. El tamaño del problema se ajusta en función de la cantidad de filas de la tabla `Persons`.

Las consultas para este esquema son:

Join

S_1	`SELECT p.Name, j.Name FROM Jobs j, Persons p WHERE p.Job == j.Id`
S_{1a}	`SELECT sum(length(j.Name)) FROM Jobs j, Persons p WHERE p.Job == j.Id`

Join and filter

S_2	`SELECT p.Name FROM Jobs j, Persons p WHERE p.Job == j.Id AND j.Name="T4"`
S_{2a}	`SELECT sum(length(p.Name)) FROM Jobs j, Persons p WHERE p.Job == j.Id AND j.Name="T4"`

Scan

S_3	`SELECT p.Id, p.Job FROM Persons p`
S_{3a}	`SELECT sum(length(name)) FROM Persons`

En cada caso, las consultas S_1, S_2 y S_3 devuelven todas las filas que no son filtradas, mientras que las consultas S_{1a}, S_{2a} y S_{3a} realizan una sumatoria (i.e. reducción) sobre todas las filas resultado.

ii) *TPC-H (H)*: El benchmark TPC-H pertenece al conjunto de benchmarks de TPC, este provee un generador de bases de datos y describe un conjunto de consultas [48].

Para generar una base de datos se debe proveer un factor de escala, el mínimo permitido por la aplicación resultó generar una base de datos mayor del rango buscado para los experimentos. Para solucionar esto se quitaron filas de las tablas generadas hasta alcanzar el tamaño deseado.

Las consultas en este caso se seleccionaron siguiendo la metodología propuesta en Kim et al. [22]. Se tomó la consulta Q6 y se derivaron simplificaciones quitando condiciones en los filtros.

La consulta Q6 original del benchmark TPC-H es:

Q6	- TPC-H/TPC-R Forecasting Revenue Change Query (Q6) SELECT sum(l_extendedprice * l_discount) AS revenue FROM lineitem WHERE l_shipdate >= '1994-01-01' AND l_shipdate <'1995-01-01' AND l_discount between 0.06 - 0.01 AND 0.06 + 0.01 AND l_quantity <24

Y las consultas derivadas de esta son:

H_1	SELECT sum (l_extendedprice * l_discount) AS promo_revenue FROM lineitem WHERE l_shipdate >= '1994-01-01' AND l_shipdate <1995-01-01 AND l_discount <0.07 AND l_discount >0.05 AND l_quantity <24
H_2	SELECT sum(l_extendedprice * (1-l_discount)) AS promo_revenue FROM lineitem, part WHERE l_partkey = p_partkey AND l_shipdate >= 1995-09-01 AND l_shipdate <1995-10-01
H_3	SELECT sum(l_extendedprice * (1-l_discount)) AS promo_revenue FROM lineitem, part WHERE l_partkey = p_partkey

Las consultas están ordenadas en sentido ascendente en función de la cantidad de instrucciones que son requeridas para resolver cada una de ellas, desde H_1 a H_3.

H_1 es la consulta más restrictiva, los filtros descartan muchas de las filas, decrementando drásticamente la cantidad de trabajo requerido para resolverla. Las consultas H_2 y H_3 poseen menos filtros, pero adicionalmente la tabla `part` es agregada. Esto genera un filtro adicional, ya que se chequea si la referencia a esta tabla es nula o no. La combinación de las características de cada consulta produce el incremento de trabajo en cada una, que es independiente de la cantidad de instrucciones totales del código de la planificación de las consultas.

7.3 Resultados y discusión

Todas las consultas del experimento, fueron ejecutadas con distintos tamaños de bases de datos. En particular, modificando el tamaño de la tabla principal sobre la que se realiza la consulta.

Para simplificar el análisis sobre los distintos tamaños, se agruparon en casos dependiendo del tamaño de la tabla principal, como se muestra en la tabla 7.1.

Denominación	Cantidad de filas
Pequeño	10 to 60
Mediano	70 to 400
Grande	400 to 1000

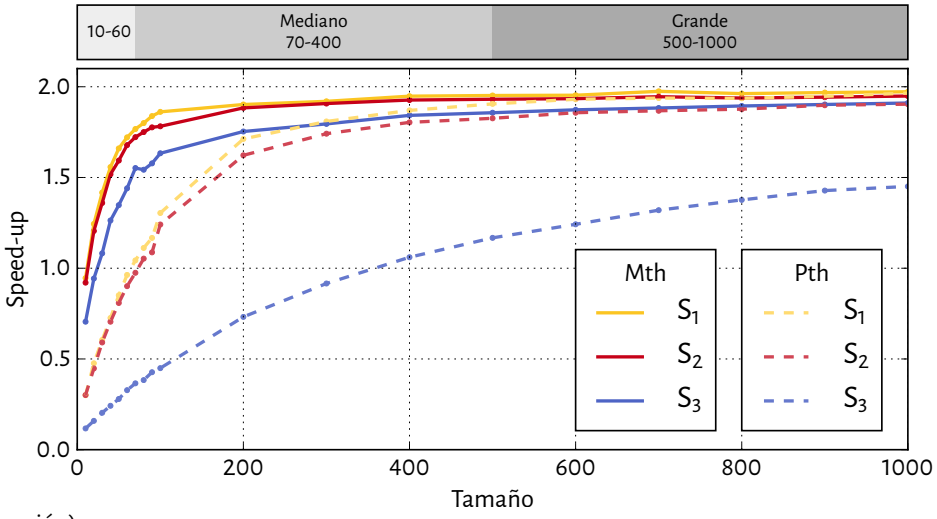
Tabla 7.1: Grupos dependiendo del tamaño de la tabla principal.

A continuación se presentan los resultados obtenidos haciendo foco en dos aspectos: rendimiento en términos de speedup y consumo de energía.

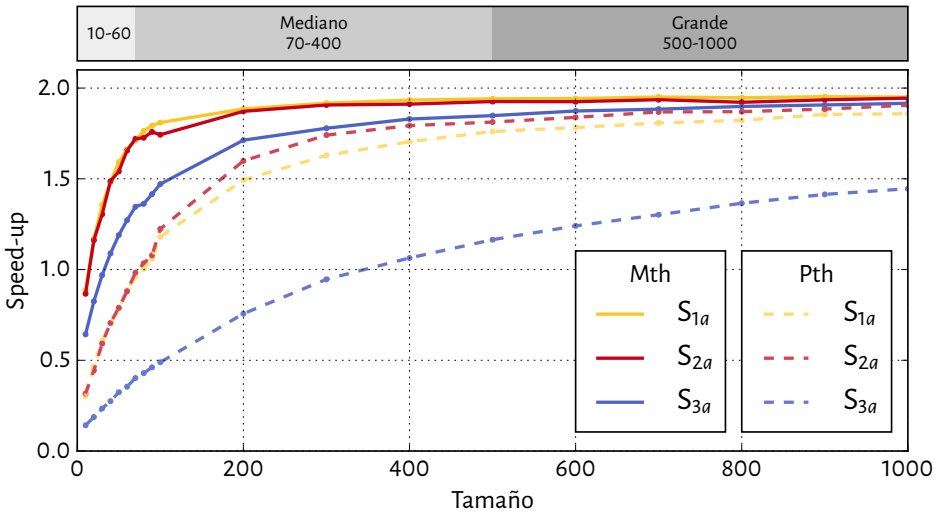
7.3.1 Rendimiento

Los gráficos presentados por la figura 7.6 muestran el speedup registrado por la ejecución en paralelo para ambos casos, tanto para *Mth* como para Pthreads (Pth) en comparación con el casos secuencial.

(a) Simple (Consulta completa)



(b) Simple (Consulta con agregación)



(c) TPC-H

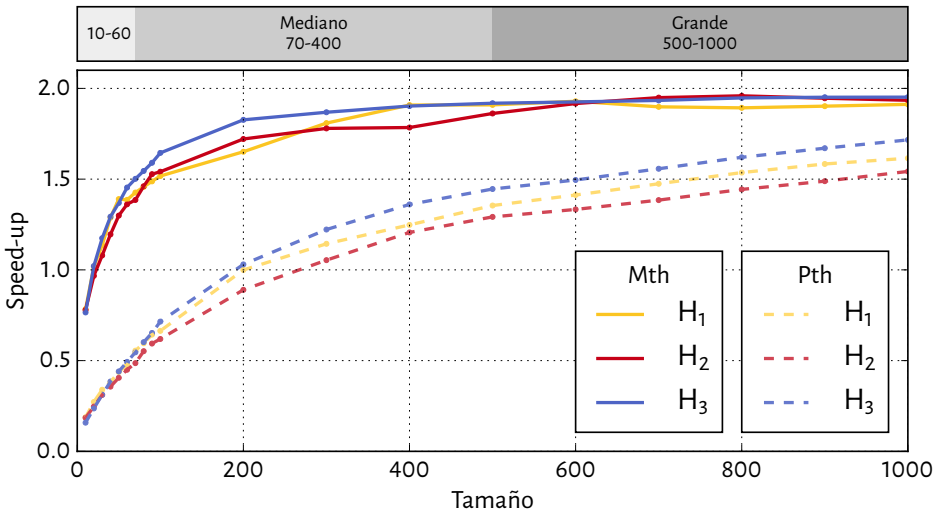


Figura 7.6: Speedup de resolución de diferentes tipos de consulta aumentando el tamaño de la tabla principal. La frecuencia de procesador utilzada fue de 2 GHz. Se compara la propuesta *Mth* con el framework de paralelización Pthreads.

Para estos experimentos se utilizó el procesador a una frecuencia de 2 GHz. La parte superior de cada gráfico muestra los grupos de tamaños registrados en el tabla 7.1.

Las figuras 7.6a y b muestran un comportamiento similar, revelando que la agregación de datos en las consultas S_{1a} a S_{3a} no produce un impacto observable en comparación con sus contrapartes no agregadas. Pthreads presenta un speedup cercano al ideal para los casos grandes en las consultas S_1, S_{1a}, S_2 y S_{2a}, mientras que para casos pequeños muestra un rendimiento pobre. En casos medianos se muestra una transición desde un rendimiento pobre, hasta un rendimiento casi ideal a medida que aumenta la cantidad de trabajo a realizar. Por otro lado, *Mth* muestra un speedup similar para casos grandes. Sin embargo su comportamiento para casos medianos es notablemente mejor que Pth. Las consultas S_1, S_{1a}, S_2 y S_{2a} para tamaños pequeños muestran una importante mejora respecto a Pthreads, obteniendo un speedup superior a uno, incluso para casos muy pequeños.

En los casos de las consultas S_3 and S_{3a} se observa una clara diferencia de rendimiento respecto a las consultas mencionadas anteriormente. Pthreads logra compensar el overhead de paralelización con más de 350 filas y no llega a alcanzar un speedup superior a $1{,}6$X incluso para el tamaño más grande de tabla. También S_3 y S_{3a} presentan el peor speedup para *Mth*, pero en este caso, *Mth* logra alcanzar un speedup de $1{,}6$X con tan solo 100 filas, mientras que para el resto del rango de tamaños estudiado, presenta un speedup menor que el obtenido por las consultas S_1, S_{1a}, S_2 y S_{2a}.

La figura 7.6c incluye el speedup obtenido por las consultas derivadas de TPC-H. El comportamiento observado resulta similar al presentado por la consulta S_3 en las figuras 7.6a y b: Pthreads requiere de más de 100 filas para compensar el overhead de paralelización, y solamente logra alcanzar un speedup alrededor de $1{,}6$X para el tamaño más grande estudiado. *Mth*, en cambio, muestra una mejora notable de speedup, alcanzando $1{,}5$X en casos pequeños y cerca del speedup ideal en casos grandes. En este conjunto de consultas, *Mth* obtiene un speedup notablemente mejor que Pthreads para todo el rango de tamaños estudiado.

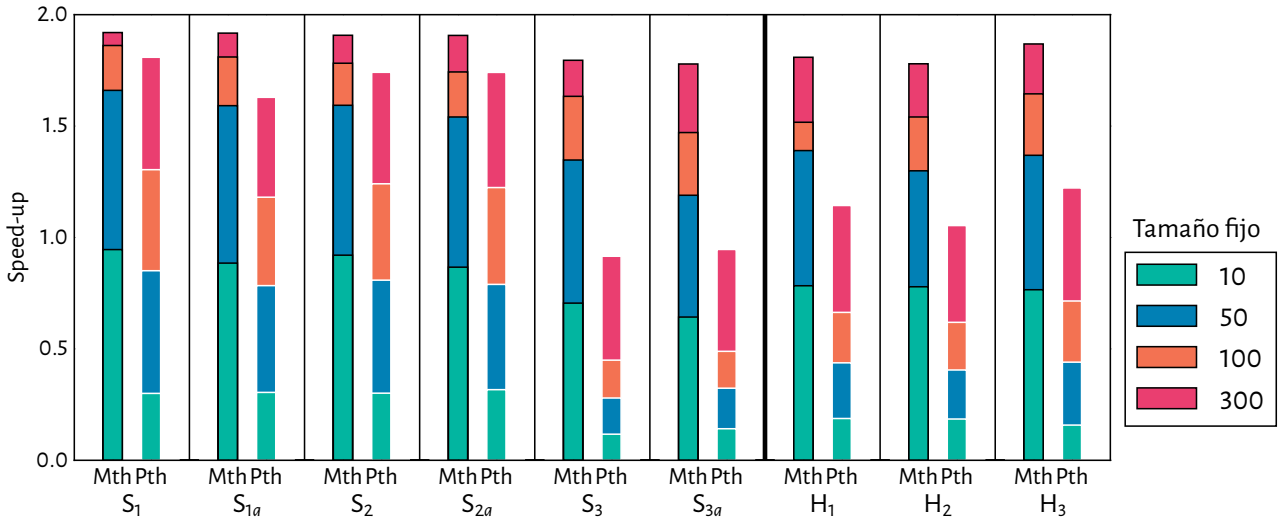


Figura 7.7: Speedup obtenido por *Mth* y Pth en casos pequeños y medianos. Cada valor es superpuesto mostrando la mejora obtenida por el incremento del tamaño en cada consulta.

Las mayores diferencias en speedup se aprecian en los casos pequeños y medianos para ambas implementaciones. La figura 7.7 hace foco en el speedup alcanzado para una cantidad determinada de filas, comparando para estos casos el speedup obtenido por *Mth* y Pthreads. Como es esperable, paralelizar una consulta que involucre solamente 10 filas se presenta como un reto por la limitada cantidad de trabajo a administrar. Con tal poca cantidad de trabajo, Pthreads presenta una notable pérdida de rendimiento respecto al caso secuencial. Por otro lado, *Mth* logra burlar tan desfavorable escenario obteniendo un tiempo de ejecución compatible al caso secuencial.

En el caso de resolver una consulta con 100 filas, Pthreads comienza a mostrar alguna ganancia sobre el secuencial para los casos S_1 y S_2, sin embargo para las consultas S_3 y H_1 a H_3, continúa obteniendo un speedup inferior a uno, es decir, demora más tiempo que el caso secuencial. Con 50 filas, *Mth* logra obtener un speedup cercano a $1{,}5$X, mientras que Pthreads aun necesita más tiempo que el caso secuencial. En estos casos, S_3 y H_1 a H_3, Pth logra obtener una ganancia marginal solo cuando se superan las 300 filas, al mismo tiempo que *Mth* alcanza un speedup cercano al ideal.

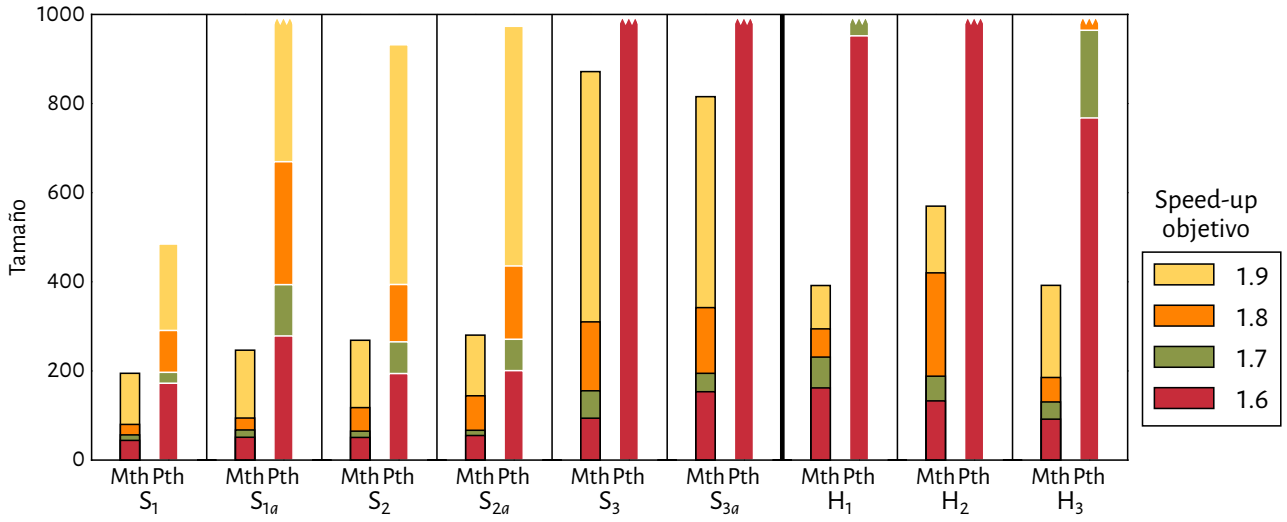


Figura 7.8: Tamaño de la consulta (cantidad de filas de la tabla principal) necesarias para obtener un speedup específico (menor valor es mejor).

Otro punto de vista acerca de la relación de rendimiento entre la ejecución secuencia y paralela, consiste en comparar el tamaño requerido para alcanzar un determinado speedup. En este caso, se buscará obtener la respuesta de cada técnica de paralelización en términos de la cantidad de filas que deben ser incluidas en una consulta para alcanzar un speedup dado ($1{,}6$, $1{,}7$, $1{,}8$ y $1{,}9$).

Estos resultados son ilustrados por la figura 7.8 que revela notables diferencias entre *Mth* y Pthreads. La consulta S_1 es el caso en que ambas tecnologías presentan un comportamiento similar: *Mth* alcanza $1{,}9$X con 200 filas, mientras que Pthreads necesita alrededor de 500. A pesar de la diferencia entre ambas, para el resto de las consultas se muestran, incluso, diferencias más importantes. Pthreads no puede alcanzar $1{,}9$X de speedup para los tamaños analizados con la consulta S_{1a}, este hecho se repite para $1{,}6$X de speedup para las consultas S_3, S_{3a} y H_2, para $1{,}7$X de speedup en la consulta H_1, y por ultimo, $1{,}8$X de speedup para la consulta H_3. *Mth* muestra en cambio un muy buen rendimiento para casos pequeños: para alcanzar $1{,}6$X de speedup, necesita entre 50 y 100 filas sobre todas las consultas analizadas. Además es destacable que para las consultas S_3 y S_{3a}, donde Pth no puede alcanzar ni $1{,}6$X de speedup, *Mth* alcanza $1{,}9$X con tan solo 800 filas. También se observan diferencias importantes en los casos H_1, H_2 y H_3.

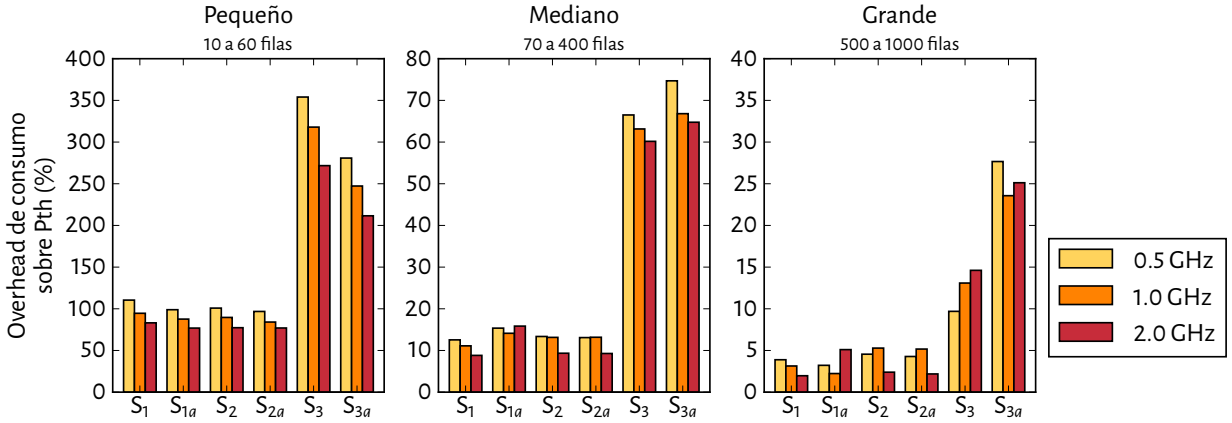
En términos de speedup la propuesta muestra interesantes resultados, especialmente cuando se enfrenta a consultas con poco volumen de trabajo. Cuando se resuelve una consulta con una cantidad extremadamente baja de filas, *Mth* logra obtener un tiempo de ejecución equivalente al tiempo secuencial, mientras que Pthreads solamente logra los mismos resultados cuando la cantidad de filas es de cinco veces o más.

Se puede concluir inicialmente, *Mth* puede ocuparse eficientemente de consultas pequeñas cuando se compara al tiempo de ejecución secuencial en términos de tiempo total, pero aun es necesario considerar el comportamiento en términos de consumo de energía.

7.3.2 Consumo de energía

En esta sección se exponen los resultados obtenidos de la estimación del consumo de energía utilizando la combinación de herramientas `gem5` y `McPAT`.

(a) Consultas S



(b) Consultas H

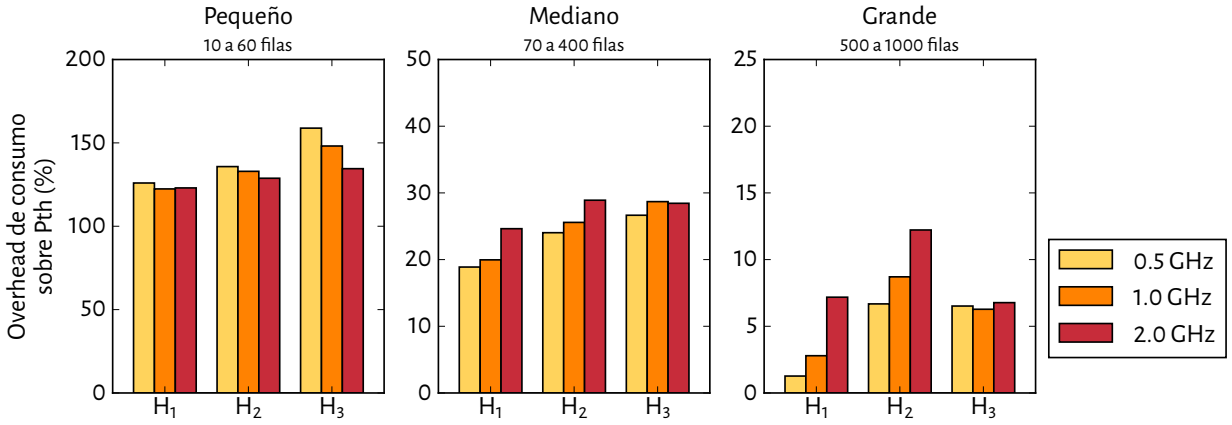


Figura 7.9: Consumo de energía promedio entre Pth y *Mth* para las tres frecuencias consideradas.

La figura 7.9 muestra la comparación en consumo de energía entre *Mth* y Pthreads, para tres frecuencias diferentes (0.5 GHz, 1.0 GHz y 2.0 GHz). Se puede observar la energía adicional utilizada por Pthreads en relación con *Mth*. Los datos se agrupan por tamaño siguiendo los casos definidos en el tabla 7.1.

En la figura 7.9a se incluyen las consultas S_1, S_{1a}, S_2, S_{2a}, S_3 y S_{3a}, mientras que la figura 7.9b incluye los resultados de las consultas derivadas de TPC-H, H_1, H_2 y H_3.

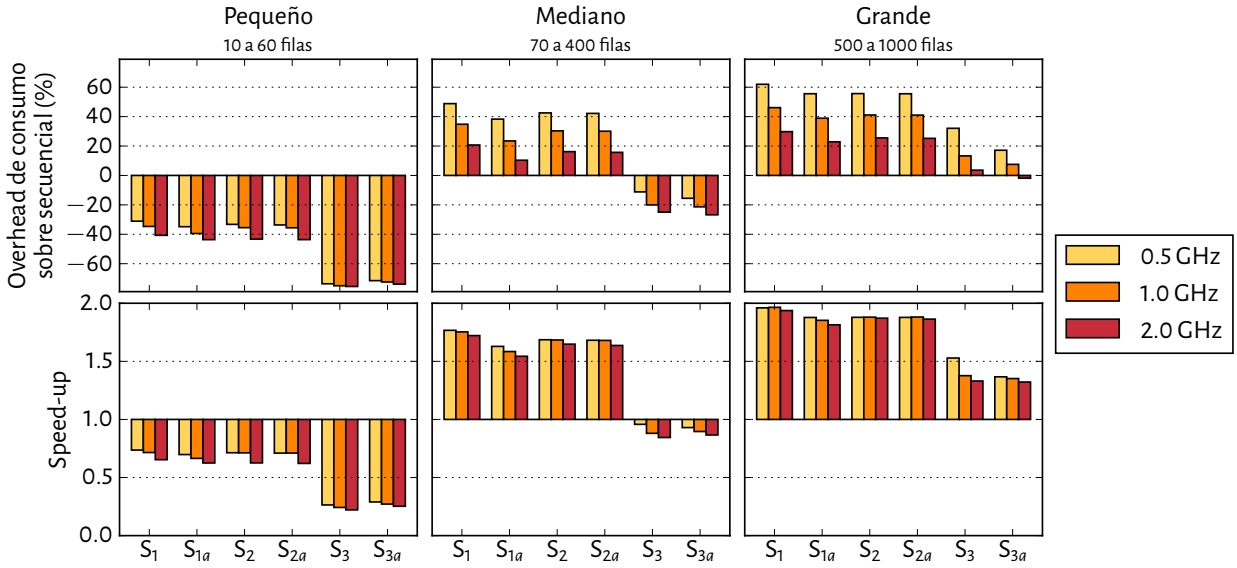
El overhead de consumo para casos pequeños es muy grande, alcanzando entre 200 % y 350 % para las consultas S_3 y S_{3a} en todas las frecuencias estudiadas. Para las consultas S_1, S_2, H_1 y H_2 el consumo adicional sobre *Mth* es del orden del 100 %, mientras que para la consulta H_3 es incluso mayor. Estos resultados, confirman la capacidad de *Mth* para administrar pequeños volúmenes de trabajo de forma eficiente.

Para tamaños medianos, la diferencia más grande se obtiene para la consulta S_3 en el rango de 60 % a 75 % de overhead de consumo. El resto de las consultas S obtienen un 10 %, mientras que para las consultas H se observa 20 % de diferencia.

La diferencia de consumo para tamaños grandes es incluso menor. A pesar de esta situación, la consulta S_{3a} alcanza un 25 % de overhead y la consulta S_3 supera el 10 % para 2 GHz. H_3 muestra diferencias de overhead cercanas a 5 % para todas las frecuencias analizadas, mientras que H_2 presenta diferencias entre 5 % y 10 % dependiendo de la frecuencia de operación. En el caso de H_1, solamente para la frecuencia de

2.0 GHz la diferencia es mayor al $5\,\%$ mientras que para el resto de las frecuencias se puede observar una ganancia marginal. Los resultados para H_1 and H_2 muestran el incremento de la diferencia de consumo a medida que se aumenta la frecuencia.

(a) Pth



(b) *Mth*

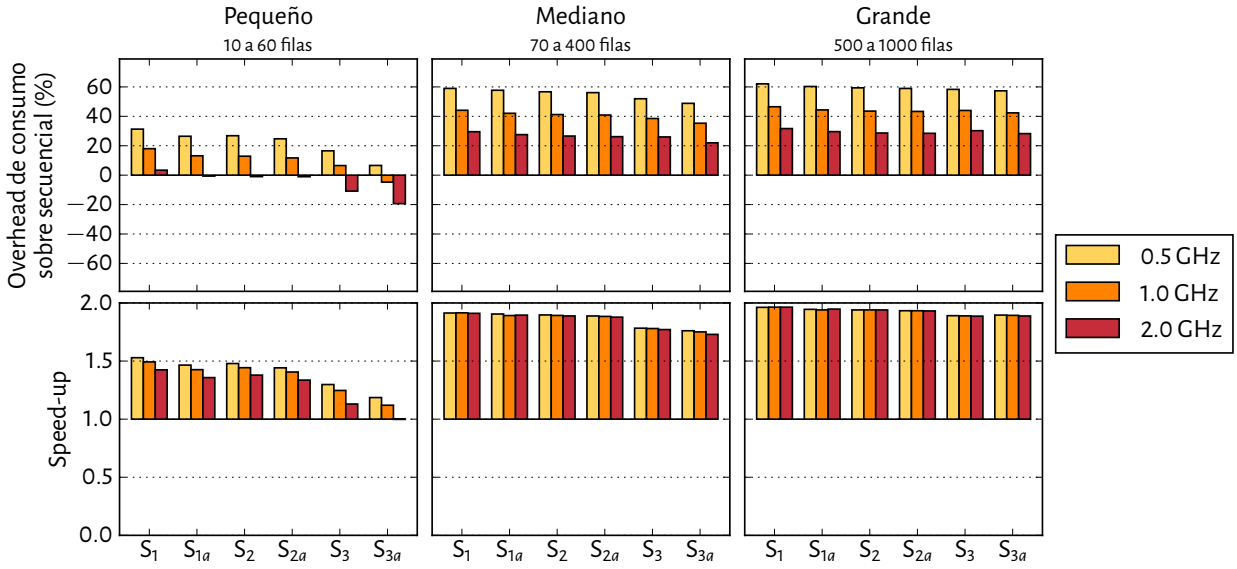
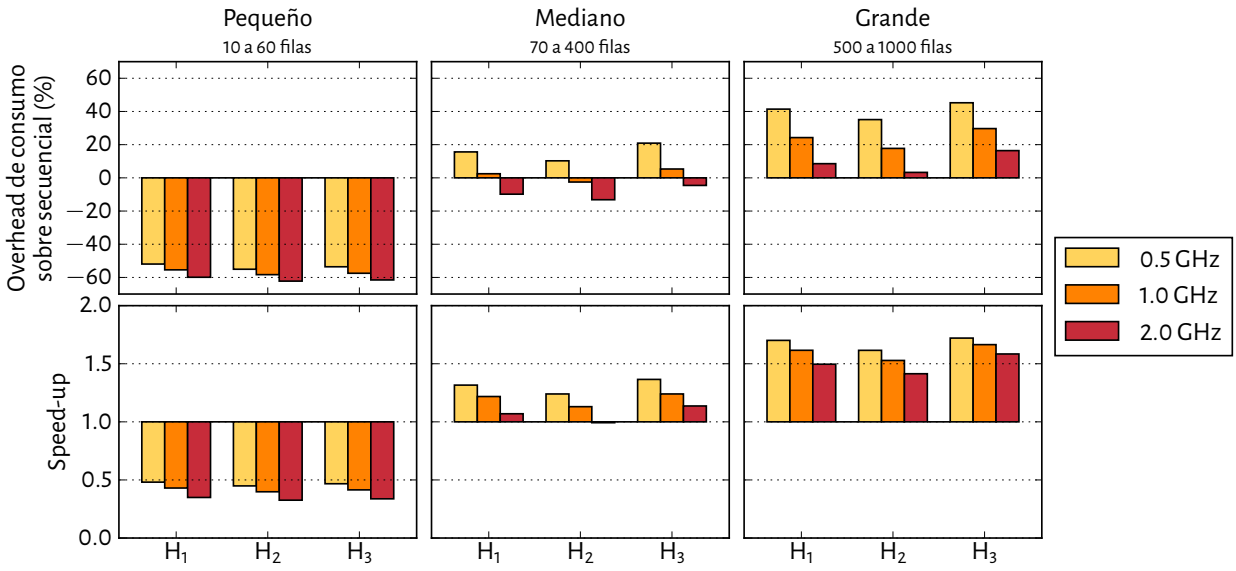


Figura 7.10: consultas S: Speedup y overhead de consumo sobre el caso secuencial en Pth y *Mth*.

Las figuras 7.10 y 7.11 muestran una comparación del consumo de energía de Pth y *Mth* en comparación con el caso secuencial para las frecuencias 0.5 GHz, 1.0 GHz y 2.0 GHz. Cada una presenta seis gráficos ordenados en dos filas, la primer fila corresponde al overhead comparado con caso secuencial, mientras que la segunda ilustra el speedup obtenido para cada caso. Las columnas corresponden a los diferentes conjuntos de tamaños.

Como fue mencionado anteriormente, resolver una consulta pequeña representa un desafío para cualquier framework de paralelización. El overhead introducido por paralelizar debe ser compensado por el trabajo realizado en paralelo, pero en casos pequeños no se cuenta con suficiente trabajo como para poder compensar el overhead. Esto resulta claro para la primer columna de la figura 7.11a, en donde la ejecución en paralelo de Pthreads toma más tiempo que el caso secuencial, especialmente para la consulta S_3 y S_{3a}.

(a) Pth



(b) *Mth*

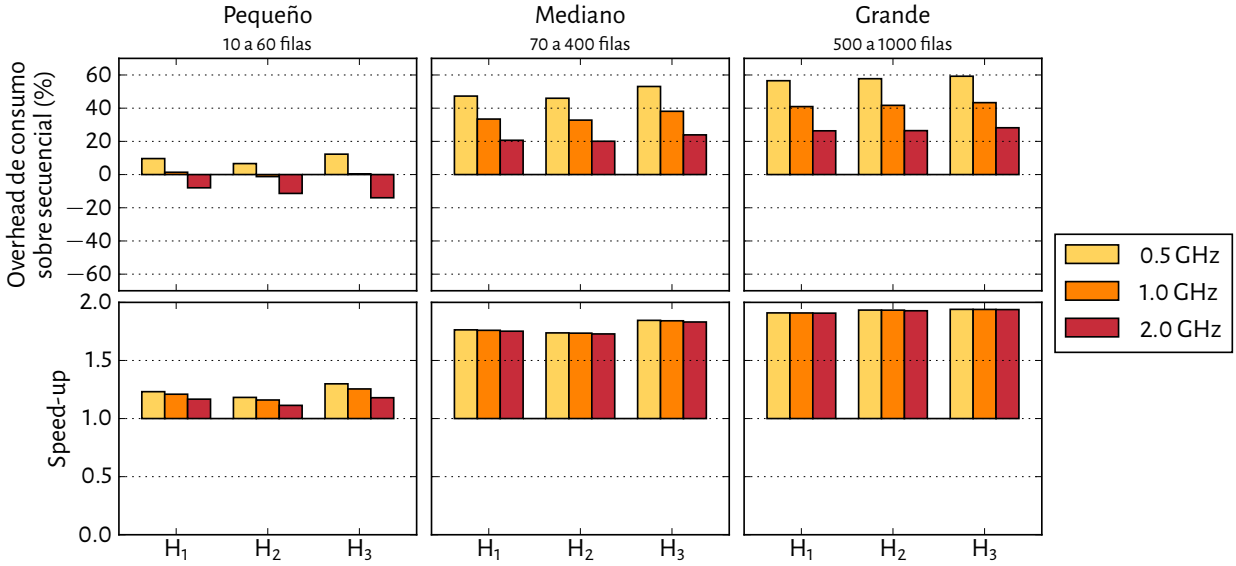


Figura 7.11: consultas H: Speedup y overhead de consumo sobre el caso secuencial en Pth y *Mth*.

Como es esperado, el consumo de energía también es mayor para el caso paralelo, consiguiendo el mismo patrón de rendimiento que para el speedup. Las consultas S_3 y S_{3a} nuevamente presentan el peor comportamiento. Para tamaños medianos, las consultas experimentan cambio de comportamiento, mientras que las consultas S_3 y S_{3a} continúan tomando más tiempo y energía que el caso secuencial, las consultas S_1 y S_2 muestran una clara ganancia, superando el $1{,}5$X de speedup y entre 20 % y 40 % de reducción en el consumo de energía. En casos grandes, las consultas S_3 y S_{3a} presentan el peor seedup y la menor reducción en consumo en comparación al resto de las consultas. Se observa incluso un leve aumento del consumo para la consulta S_{3a} a 2.0 GHz. El resto de las consultas se aproximan al speedup ideal manteniendo una buena relación de reducción de consumo. Es destacable la relación entre frecuencia y consumo: a frecuencias más altas, la relación de reducción de consumo se vuelve menor.

La figura 7.10b muestra los mismos resultados pero para *Mth*, el contraste con Pthreads es notable. Incluso para casos pequeños, el speedup obtenido por *Mth* alcanza valores cercanos a $1{,}5$X para las consultas S_1 y S_2 en sus dos variantes. Por otro lado, el speedup dado por las consultas S_3 y S_{3a} resulta más limita-

do. En cualquier caso, se observan interesantes ganancias en términos de rendimiento para los tres valores de frecuencia analizados. La relación de consumo de energía cambia notablemente con la frecuencia: para 0.5 GHz se obtiene una reducción de consumo mayor al $20\,\%$ para todas las consultas, con excepción de S_3 que no sobrepasa este valor y para S_{3a} que muestra una reducción marginal del consumo. Cuando se pasa a la frecuencia de 1.0 GHz, la relación de consumo es comparable a la implementación secuencial, ninguno de los casos supera el $20\,\%$, logrando una reducción marginal para S_3 y para S_{3a} se obtiene un consumo superior al caso secuencial. Esta situación incluso es peor cuando la frecuencia es de 2.0 GHz, la ejecución en paralelo obtiene una reducción del consumo solamente para la consulta S_1, mostrando un consumo superior al secuencial para el resto de las consultas, especialmente para S_{3a} el consumo alcanza un $20\,\%$ más que el consumo secuencial.

A pesar de esto, la comparación con Pthreads muestra una clara ganancia en la mayor parte de los casos pequeños, incluso reduciendo el consumo de energía. Para casos medianos y grandes se muestra un comportamiento similar en *Mth*: solamente las consultas S_3 y S_{3a} obtienen un speedup menor que para el resto, pero comparado con Pthreads, el rendimiento en tiempo y energía de *Mth* es destacado en estas ultimas.

Para las consultas derivadas de TPC-H, la figura 7.11a muestra un escenario similar: los casos pequeños son resueltos ineficientemente por Pthreads, tomando mucho más tiempo que el caso secuencial y consumiendo el $50\,\%$ más de energía. Los casos medianos son mejor administrados por Pthreads, pero aún se obtiene un reducido speedup y solo se ven mejoras en consumo para la frecuencia de 0.5 GHz, presentando mayor consumo en las otras dos frecuencias estudiadas. Solamente con casos grandes Pthreads logra obtener un mejor rendimiento y reducción del consumo. Resta considerar que para la frecuencia de 2.0 GHz el consumo aún para casos grandes se reduce de forma marginal, y solamente para las frecuencias de 0.5 GHz y 1.0 GHz el consumo se logra reducir en un $40\,\%$ y $20\,\%$ respectivamente..

Una vez más, *Mth* logra administrar casos pequeños presentando una reducción en consumo, especialmente cuando es comparado con Pthreads. Aunque el rendimiento no superó el $1{,}5$X de speedup y el consumo solo se vió reducido para 0.5 GHz, los resultados muestran un aumento del $20\,\%$ en consumo solamente en uno de los casos a la frecuencia de 2.0 GHz. En casos medianos y grandes, se presentó un patrón similar en que el rendimiento superó claramente el $1{,}75$X de speedup para todas las frecuencias. La reducción de consumo se observó cercana al $20\,\%$ para 2.0 GHz y alrededor del $50\,\%$ para 0.5 GHz.

Finalmente la figura 7.12 presenta la comparación en términos de consumo de energía para Pthreads y *Mth*, normalizada a la cantidad de energía promedio requerida para procesar una fila de la consulta H1. Al igual que en los experimentos anteriores y, a fin de estudiar su impacto, los resultados se presentan sobre las frecuencias 0.5 GHz, 1.0 GHz y 2.0 GHz. La figura 7.12a hace foco en los resultados de las consultas para los tamaños entre 10 a 100 filas, mientras que la figura 7.12b entre los tamaños 100 a 1000 filas. La escala en ambos casos es fija, de modo de facilitar la comparación entre Pthreads y *Mth*.

Como es esperado, el caso secuencial usando Pthreads y *Mth* consume una cantidad similar de energía. Las variaciones en los gráficos corresponden a las condiciones del experimento y pueden ser ignoradas.

En todo el rango analizado en la figura 7.12b, *Mth* presenta un menor consumo comparado con Pth, muestra además una consistente mejora en las tres frecuencias. Esto es más significativo para los tamaños pequeños, aun así aunque se aumente el tamaño, *Mth* muestra de forma sostenida un menor consumo de energía.

Los casos pequeños muestra una notable mejora en términos de consumo de energía, significativamente mejor que la observada para casos grandes. La figura 7.12a revela que *Mth* puede beneficiarse de su propia eficiencia cuando se trata de consultas pequeñas. Se observa, que cuando se resuelven consultas entre 10 a 50 filas, la energía utilizada por Pthreads duplica a la requerida por *Mth*. Para el resto del rango en la figura, *Mth* muestra un consumo menor de energía por fila. Esta situación es consistente en las tres frecuencias analizadas.

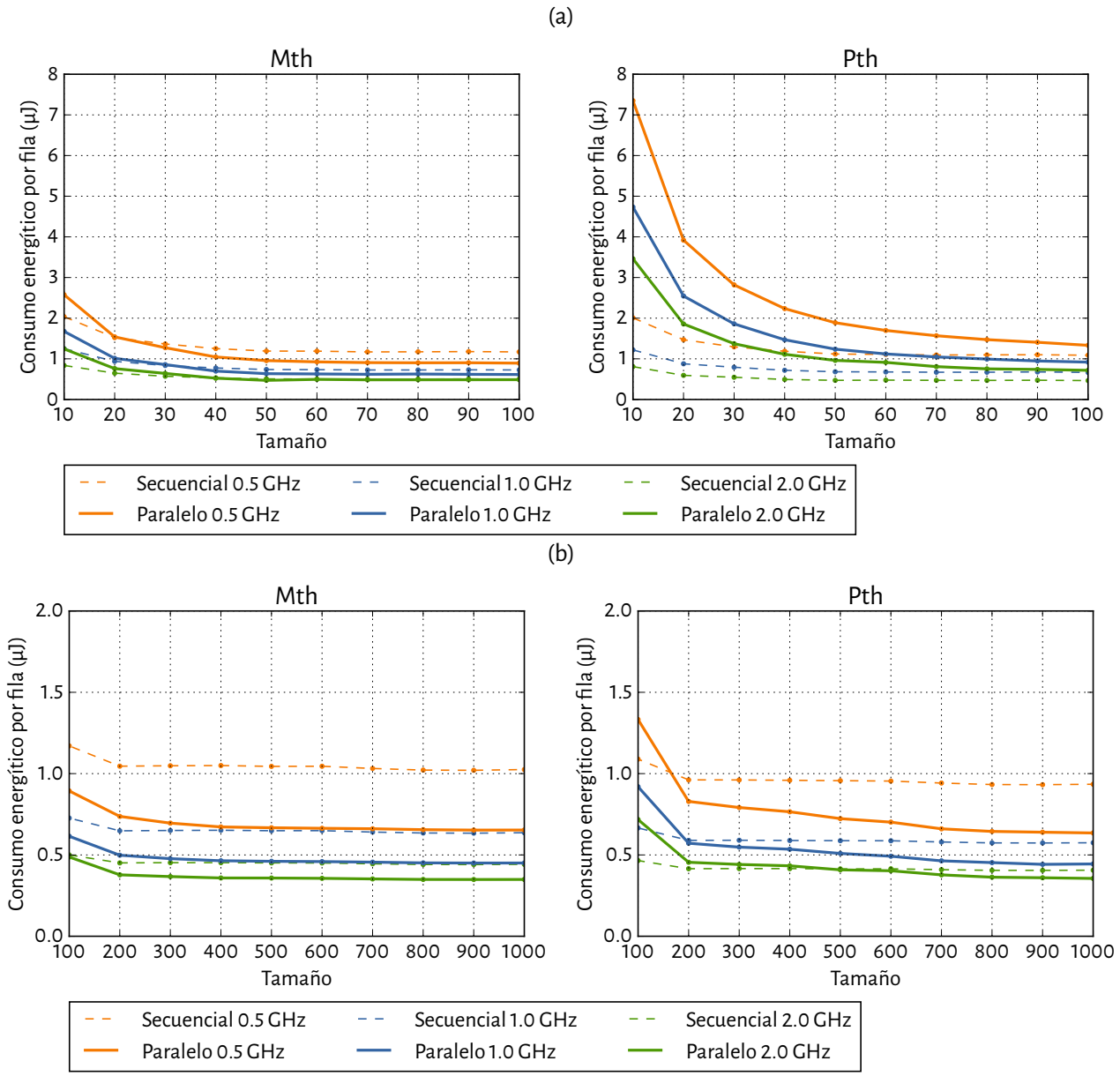


Figura 7.12: Consumo de energía por fila, para el caso secuencial y el caso paralelo, resolviendo la consulta H_1. Pth y *Mth* son comparados utilizando tres frecuencias distintas.

Capítulo 8

Conclusiones

La principal tarea de un procesador es ejecutar programas tan rápido como sea posible, dentro de limitaciones impuestas por el costo del propio sistema y su consumo energético. En este camino se han presentado grandes avances en la tecnología de fabricación y en arquitectura de procesadores. Algunos de estos llegaron para quedarse, como es el caso de los procesadores multicores que actualmente se pueden encontrar en, prácticamente, cualquier dispositivo de cómputo. Los sistemas operativos, por su parte, mantienen una forma tradicional de administrar estos recursos. Los modelos de programación tampoco han avanzado en el mismo grado que los procesadores: a pesar de existir nuevos lenguajes y plataformas orientados al uso de múltiples recursos de cómputo, éstos no son utilizados masivamente en aplicaciones. Existen excepciones como algunas aplicaciones de cálculo científico o multimedia, que basados en la regularidad y predictibilidad de sus operaciones pueden hacer uso de estas tecnologías.

El speedup de un programa paralelo está limitado por su fracción secuencial, es decir, por la parte que no puede ser paralelizada. Para complicar aún más esta situación, la implementación en paralelo introduce overhead adicional. En tiempo de ejecución, aparece otro overhead dado por la administración de los threads, la sincronización y la diferencia en el balance de carga entre cores. En este sentido, si la cantidad de trabajo disponible para ser paralelizado es muy limitada, el overhead podría llegar a ser tan grande que la solución paralela tomaría más tiempo que la solución secuencial. Esto se reflejaría en valores de speedup inferiores a la unidad.

En este trabajo se estudiaron soluciones codiseñadas de hardware-software cuyo objetivo es minimizar el overhead relacionado con la paralelización y la administración de threads. Buscando impactar particularmente en la creación, gestión y sincronización de threads. Se propone para esto una serie de extensiones a procesadores multicore denominada Micro-threads (*Mth*). Las extensiones incluyen un mecanismo para salvar contextos de ejecución, registros específicos de sincronización e instrucciones para controlar y administrar cores. *Mth* permite al programador tener control completo de la asignación de los procesadores, dejando la posibilidad de construir una eficiente política de scheduling de fracciones de código paralelo.

Para construir experimentos sobre la propuesta se realizaron aportes en la metodología de simulación. Construir un entorno controlado, en el cual poder combinar los resultados obtenidos de ejecutar en plataformas completas dentro de un sistema operativo, con soluciones por fuera del control de un sistema operativo, fue parte fundamental de los métodos desarrollados para este trabajo. Adicionalmente se tuvieron que desarrollar herramientas para realizar las mediciones y otorgar repetibilidad a los experimentos. Un problema general de los experimentos en arquitectura de procesadores, tiene que ver con las demoras en los simuladores de eventos discretos. Otro aspecto fundamental de la metodología, fue construir experimentos donde se aprovechará al máximo el tiempo de ejecución de las simulaciones. Permitiendo así realizar experimentos extensos sobre múltiples parámetros con un gran nivel de detalle en la simulación.

La implementación de este trabajo se realizó sobre un simulador fuertemente establecido dentro de la comunidad, que permite simular múltiples arquitecturas con variados niveles de detalle. A diferencia de desarrollar un sistema de simulación ad-hoc, usar una herramienta soportada por una gran comunidad de desarrolladores ayuda en el proceso de análisis y en la certeza de validaciones realizadas previamente. Las publicaciones mostrando los casos de uso de este simulador permiten profundizar en aspectos clave de la fidelidad lograda por los modelos implementados en el simulador.

Las extensiones dentro del simulador fueron implementadas como una solución modular, sin modificar el diseño de la arquitectura del simulador. Para lograr esto, se debieron estudiar las formas de implementación posibles, logrando una metodología de extensiones a sistemas completos que permitiría, no solo implementar esta solución, sino cualquier otro tipo de extensiones sobre sistemas completos.

El modelo resultado de este trabajo es un conjunto de extensiones simples, que pueden ser implementadas sobre diferentes arquitecturas. Estas extensiones resultan como una solución general que ataca puntualmente los problemas de overhead que presentan otro tipo de soluciones de paralelización. Además, es posible implementa cualquier framework de software sobre *Mth*, obteniendo así los beneficios de utilizar esta tecnología.

Los experimentos realizados sobre la propuesta se enfocan en dos aspectos. Por un lado un conjunto de algoritmos paradigmáticos bien definidos que cubren un amplio espectro de formas de sincronización y patrones de cómputo: HSL filter (trivialmente paralelizable), FFT Radix2 (algoritmo recursivo), LU decomposition (barrera de sincronización en cada ciclo) y Dantzig algorithm (basado en grafos y operaciones sobre matrices). El speedup y la eficiencia en la paralelización obtenida en todos los casos resultó notable, alcanzando valores casi ideales en un amplio rango de tamaños. Incluso cuando se opera con instancias muy chicas, del orden de cientos de bytes, y para tiempos de ejecución de décimas de segundo.

El segundo aspecto hace uso de una aplicación real. Se modificó SQLite, un motor de bases de datos embebido, para soportar la ejecución de planificaciones de consultas paralelas. La implementación se realizó utilizando las primitivas, tanto de *Mth* como el soporte de Pthreads (Pth). En este caso se analizó adicionalmente el consumo de energía utilizando la herramienta `McPAT`. Dos conjuntos de consultas fueron utilizados: i) join, join-filter y scan, sobre un esquema de dos tablas relacionadas (`Jobs`$\rightarrow$`Persons`); y ii) tres consultas derivadas de Q6 del TPC-H benchmark. Al comparar Pthreads con *Mth*, se observa una mejora notable en el rendimiento para instancias muy pequeñas. La propuesta obtuvo ganancias en términos de speedup (entre $1{,}5$X a $1{,}9$X) y consumo de energía (entre $20\,\%$ a $50\,\%$ menos) considerables sobre diferentes frecuencias de operación, incluso para consultas con menos de 100 filas. La comparación con Pthreads confirma la eficiencia que es posible obtener por medio de *Mth*, tanto en términos de consumo, como speedup, especialmente en casos pequeños.

La serie de extensiones a procesadores provistas por *Mth* prueban ser una solución eficiente y atractiva para enfrentar los problemas de overhead en la paralelización de aplicaciones. Es eficiente en términos de los speedup logrados, incluso en instancias muy pequeñas donde los frameworks de paralelización no logran un rendimiento aceptable. Resulta atractiva, a su vez, por ser simple y con posibilidades de ser implementada en procesadores multicore actuales. Además, los frameworks de paralelización que actualmente utilizan las primitivas provistas por el sistema operativo a fin de controlar threads, podrían utilizar las primitivas de *Mth* en forma de instrucciones. Haciendo uso así de toda la lógica y desarrollo dado por el propio framework, pero con los beneficios en eficiencia y reducción de overhead dado por *Mth*.

En el futuro se espera que la proliferación de recursos de cómputo disponibles en un solo sistema, continué creciendo. El desarrollo de aplicaciones, actualmente no hace uso de este tipo de recursos. Es por esto que se espera que surjan soluciones cooperativas entre hardware y software que logren resolver esta diferencia.

Capítulo 9

Trabajo Futuro

El trabajo de un doctorado se inicia con un plan que con el tiempo varía, muta y se adapta a los resultados y situaciones que van surgiendo. En esta búsqueda se abren ramas, caminos en la investigación que muchas veces quedan inconclusos. Algunos de estos, son prometedores en un principio y otros son descartados. Esta sección resumen alguno de estos senderos aun por recorrer producto de esta tesis.

Uno de los aspectos más relevantes es la implementación de *Mth* en un sistema real. Para esto, una solución posible es la utilización de FPGAs. Por medio de una implementación en FPGAs se obtendría un prototipo funcional sobre el cual realizar experimentos más realistas en términos de soporte en hardware. Con este tipo de soluciones, el paso siguiente consistiría en llevar la propuesta a un modelo de circuitos, e incluso a una implementación real en un encapsulado.

Parte del overhead que busca resolver *Mth*, está dado por el sistema operativo. Estudiar la posibilidad de implementar *Mth* en un sistema actual, haciendo foco en la agrupación de cores y el impacto en el rendimiento de aplicaciones actuales que coexistirían con la propuesta.

El sistema operativo, a fin de ejecutar concurrentemente tareas, implementa una política de scheduling. Éstas pasaron de operar con procesos a ejecutar threads. *Mth* propone un nuevo desafió en este aspecto tareas sobre conjuntos de cores.

Si bien *Mth* es ejecutado a nivel de aplicación, es esperable proveer de diferentes frameworks a los desarrolladores, cuyo objetivo sea facilitar la tarea de utilizar múltiples threads, tanto de forma explícita como con soporte de compilación. Su estudio abre la posibilidad de diseñar soportes específicos para tipos de aplicaciones, o incluso para tipos de problemas.

Como fue mencionado en el texto, los frameworks de paralelización actuales pueden ser implementados sobre *Mth*. Estudiar su impacto permitiría dar cuenta del alcance de la propuesta, accediendo a un sin número de aplicaciones que ya están paralelizadas.

Las herramientas de desarrollo se presentan como otro aspecto a estudiar. Los compiladores pueden hacer uso de *Mth* para paralelizar código de forma automática. Detectar pequeñas secciones de código independiente es posible, y gracias al reducido overhead, se obtendría una paralelización eficiente.

El análisis código y debug, es fundamental para el desarrollo, en este tipo de arquitecturas los problemas de concurrencia, junto con condiciones de carrera deben poder ser detectados fácilmente. Construir nuevas herramientas se presenta como otro paso adelante en este trabajo.

Es claro que queda mucho camino por delante, pero también es posible vislumbrar los beneficios futuros de la posibilidad de utilizar múltiples cores a gran escala por parte de aplicaciones generales. Creemos que *Mth* constituye un paso firme para afianzar estos objetivos.

Bibliografía

[1] Ralph Acker, Christian Roth, and Rudolf Bayer. Parallel query processing in databases on multicore architectures. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '08, pages 2–13, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] Grant Allen and Mike Owens. *The Definitive Guide to SQLite*. Expert's Voice in Open Source. Apress, NY, second edition, November 2010.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, August 2011.

[5] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. *ACM SIGARCH Computer Architecture News*, 38(3):302–313, June 2010.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[7] Fernando A. Endo, Damien Courousse, and Henri-Pierre Charles. Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 266–273, July 2014.

[8] Robert M. Farber. Topical perspective on massive threading and parallelism. *Journal of Molecular Graphics*, 30:82–89, 2011.

[9] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, Nov 1995.

[10] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, pages 140–150, New York, NY, USA, 1983. ACM.

[11] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[12] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[13] Karl Fürlinger and Michael Gerndt. Analyzing overheads and scalability characteristics of openmp applications. In *Proceedings of the 7th international conference on High performance computing for computational science*, VECPAR'06, pages 39–51, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] R. Gioiosa, S. A. McKee, and M. Valero. Designing os for hpc applications: Scheduling. In *2010 IEEE International Conference on Cluster Computing*, pages 78–87, Sept 2010.

[15] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[16] David A. González Márquez, Adrián Cristal, and Esteban E. Mocskos. Mth: Codesigned hardware/software support for fine grain threads. *IEEE Computer Architecture Letters*, 16(1):64–67, January 2017.

[17] John L. Hennessy, David Goldberg, and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 5th edition, September 2011.

[18] M. Horowitz and W. Dally. How scaling will change processor architecture. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, number 1, pages 132–133, 2004.

[19] Intel. Multiprocessor specification version 1.4, 1997.

[20] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):186–197, June 2007.

[21] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[22] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. In-storage processing of database scans and joins. *Inf. Sci.*, 327(C):183–200, January 2016.

[23] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[24] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. *ACM SIGARCH Computer Architecture News*, 35(2):162–173, June 2007.

[25] Sheng Li, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. Technical report: Mcpat 1.0: An integrated power, area, and timing modeling framework for multicore architectures, 1997.

[26] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. The mcpat framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Trans. Archit. Code Optim.*, 10(1):5:1–5:29, April 2013.

[27] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing*, 37(1):41–54, August 1996.

[28] Carlos Madriles, Pedro López, Josep M. Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raúl Martinez, and Antonio Gonzalez. Boosting single-thread performance in multi-core systems through fine-grain multi-threading. *ACM SIGARCH Computer Architecture News*, 37(3):474–483, June 2009.

[29] P. Marcuello and A. Gonzalez. Thread-spawning schemes for speculative multithreading. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 55–64, 2002.

[30] Roberto Maro, Yu Bai, and R. Iris Bahar. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, PACS '00, pages 97–111, London, UK, UK, 2001. Springer-Verlag.

[31] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Alfons Kemper, and Thomas Neumann. Heterogeneity-conscious parallel query execution: Getting a better mileage while driving faster! In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, pages 2:1–2:10, New York, NY, USA, 2014. ACM.

[32] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, Washington, DC, USA, 2003. IEEE Comp. Soc.

[33] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith. Multicore resource management. *IEEE Micro*, 28(3):6–16, May 2008.

[34] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[35] Gregory W. Price and David K. Lowenthal. A comparative analysis of fine-grain threads packages. *Journal of Parallel and Distributed Computing*, 63(11):1050–1063, November 2003.

[36] Mihai Pricopi and Tulika Mitra. Bahurupi: A polymorphic heterogeneous multi-core architecture. *ACM Trans. Archit. Code Optim.*, 8(4):22:1–22:21, January 2012.

[37] Mihai Pricopi and Tulika Mitra. Task scheduling on adaptive multi-core. *IEEE Trans. Comput.*, 63(10):2590–2603, October 2014.

[38] Behzad Salami, Gorker Alp Malazgirt, Oriol Arcas-Abella, Arda Yurdakul, and Nehir Sonmez. Axledb: A novel programmable query processing platform on fpga. *Microprocessors and Microsystems*, 51:142 – 164, 2017.

[39] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database engines on multicores, why parallelize when you can distribute? In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 17–30, New York, NY, USA, 2011. ACM.

[40] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 311–322. ACM, 2010.

[41] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. *SIGARCH Computer Architecture News*, 31(2):422–433, May 2003.

[42] João Soares, João Lourenço, and Nuno Preguiça. *MacroDB: Scaling Database Engines on Multicores*, pages 607–619. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[43] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. *SIGARCH Comput. Archit. News*, 23(2):414–425, May 1995.

[44] Mostafa I. Soliman. Design, implementation, and evaluation of a low-complexity vector-core for executing scalar/vector instructions. *Journal of Parallel and Distributed Computing*, 73(6):836 – 850, 2013.

[45] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. Stackthreads/mp: Integrating futures into calling standards. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 60–71, New York, NY, USA, 1999. ACM.

[46] Kenjiro Taura and Akinori Yonezawa. Fine-grain multithreading with minimal compiler support—a cost effective approach to implementing efficient multithreading languages. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 320–333, New York, NY, USA, 1997. ACM.

[47] SQLite Development Team. Sqlite computer software, July 2017. Last checked on 23 de febrero de 2018.

[48] TPC. Tpc benchmark h (decision support) standard specification revision 2.17.2, 2001.

[49] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS IV, pages 176–188, New York, NY, USA, 1991. ACM.

[50] Q. Zhang, S. Li, and J. Xu. Qscheduler: A tool for parallel query processing in database systems. In *Proceedings of 19th International Conference on Engineering of Complex Computer Systems*, pages 73–76, Tianjin University, China, August 2014. IEEE.

[51] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 25–36, Washington, DC, USA, 2007. IEEE Comp. Soc.

Apéndice A

Algoritmos e implementaciones paralelas

Los experimentos realizados en el capítulo 7, consisten en paralelizaciones de los códigos secuenciales presentados a continuación.

A.1 HSL filter

```
void rgbTOhsl(uint8_t *src, float *dst) {
  int b = (int)src[B];
  int g = (int)src[G];
  int r = (int)src[R];
  int cmax = max(b,g,r);
  int cmin = min(b,g,r);
  float d = (float)(cmax -cmin);
  float h=0,l=0,s=0;
  if ( cmax == cmin ) {
    h = 0.0f;
  } else if ( cmax == r ) {
    h = 60.0f*((float)(g-b)/d + 6.0f);
  } else if ( cmax == g ) {
    h = 60.0f*((float)(b-r)/d + 2.0f);
  } else if ( cmax == b ) {
    h = 60.0f*((float)(r-g)/d + 4.0f);
  }
  h = h>=360.0f ? h-360.0f : h;
  l = (float)(cmax+cmin)/510.0f;
  if ( cmax == cmin ) {
    s = 0.0f;
  } else {
    s = d/(1.0f-fabs(2.0f*l-1.0f))/255.0001f;
  }
  #if PIXEL == 4
    dst[A] = (float)src[A];
  #endif
  dst[H] = h;
  dst[S] = s;
  dst[L] = l;
}
```

```
void hslTOrgb(float *src, uint8_t *dst) {
  float h = src[H];
  float s = src[S];
  float l = src[L];
  float c = (1.0f - fabs(2.0f*l-1.0f)) * s;
  float x = c * (1.0f - fabs( fmod(h/60.0f,2.0f) - 1.0f ) );
  float m = l - c/2.0f;
  float r=0,g=0,b=0;
  if( 0.0f<=h && h<60.0f ) {
    r=c;    g=x;    b=0.0f;
  } else if(  60.0f<=h && h<120.0f ) {
    r=x;    g=c;    b=0.0f;
  } else if( 120.0f<=h && h<180.0f ) {
    r=0.0f; g=c;    b=x;
  } else if( 180.0f<=h && h<240.0f ) {
    r=0.0f; g=x;    b=c;
  } else if( 240.0f<=h && h<300.0f ) {
    r=x;    g=0.0f; b=c;
  } else if( 300.0f<=h && h<360.0f ) {
    r=c;    g=0.0f; b=x;
  }
  #if PIXEL == 4
    dst[A] = (int)src[A];
  #endif
  dst[B]=(b+m)*255.0f;
  dst[G]=(g+m)*255.0f;
  dst[H]=(r+m)*255.0f;
}

void hsl(uint32_t wh, uint8_t* data, float hh, float ss, float ll) {
  uint8_t (*m)[PIXEL] = (uint8_t(*)[PIXEL]) data;
  int i, t = wh;
  for(i=0;i<t;i++) {
      float dst[PIXEL]; // A H S L
      rgbTOhsl((uint8_t*)(&(m[i])),(float*)(&dst));
      dst[H] = (dst[H]+hh)>= 360 ? dst[H]+hh-360 : ((dst[H]+hh)<0 ? dst[H]+hh+360:dst[H]+hh);
      dst[S] = (dst[S]+ss)>=  1  ? 1:((dst[S]+ss)<0 ? 0:dst[S]+ss);
      dst[L] = (dst[L]+ll)>=  1  ? 1:((dst[L]+ll)<0 ? 0:dst[L]+ll);
      hslTOrgb((float*)(&dst),(uint8_t*)(&(m[i])));
  }
}
```

A.2 FFT radix2

Basado en el código de Mike Andrews de 6/29/1998 (`http://www.gweep.net/~rocko/FFT/node6.html`).

```
struct complex { double r; double i; };
#define PI 3.14159265359
#define MAXPOW 24
int pow_2[MAXPOW];

void setup() {
    int i;
    pow_2[0]=1;
    for (i=1; i<MAXPOW; i++)
        pow_2[i]=pow_2[i-1]*2;
}
```

```
void twiddle(struct complex *W, int N, double stuff) {
    W->r=cos(stuff*2.0*PI/(double)N);
    W->i=-sin(stuff*2.0*PI/(double)N);
}

void radix2(struct complex *data, int N) {
    int    n2, k1, N1, N2;
    struct complex W, bfly[2];
    N1=2;
    N2=N/2;
    for (n2=0; n2<N2; n2++) {
        twiddle(&W, N, (double)n2);
        bfly[0].r = (data[n2].r + data[N2+n2].r);
        bfly[0].i = (data[n2].i + data[N2+n2].i);
        bfly[1].r = (data[n2].r - data[N2+n2].r)*W.r - (data[n2].i - data[N2 + n2].i)*W.i;
        bfly[1].i = (data[n2].i - data[N2+n2].i)*W.r + (data[n2].r - data[N2 + n2].r)*W.i;
        for (k1=0; k1<N1; k1++) {
            data[n2 + N2*k1].r = bfly[k1].r;
            data[n2 + N2*k1].i = bfly[k1].i;
        }
    }
    if (N2!=1)
        for (k1=0; k1<N1; k1++)
            radix2(&data[N2*k1], N2);
}
```

A.3 LU decomposition

```
typedef double **mat;

void LU(int n, mat L, mat U, mat a) {
    int k, i, j;
    for(k=0;k<n;k++){
        L[k][k]=1;
        for(i=k+1;i<n;i++){
            L[i][k] = a[i][k] / a[k][k];
            for(j=k+1;j<n;j++){
                a[i][j] = a[i][j] - L[i][k] * a[k][j];
            }
        }
        for(j=k;j<n;j++){
            U[k][j]=a[k][j];
        }
    }
}
```

A.4 Dantzig shortest path

```
void sp(int n, int* matrix) {
    int i, j, k, c, min, t, n;
    int (*a)[n] = (int (*)[n]) matrix;

    for(k=1;k<n;k++) {

        for(j=0;j<k;j++) {
            min = INT_MAX;
            for(c=0;c<k;c++) {
                t = a[k][c] + a[c][j];
                if( min > t && a[k][c]!=-1 && a[c][j]!=-1 ) min = t;
            }
            a[k][j] = min;
        }

        for(i=0;i<k;i++) {
            min = INT_MAX;
            for(c=0;c<k;c++) {
                t = a[i][c] + a[c][k];
                if( min > t && a[i][c]!=-1 && a[c][k]!=-1 ) min = t;
            }
            a[i][k] = min;
        }

        for(i=0;i<k;i++) {
            for(j=0;j<k;j++) {
                if( a[i][k]!=-1 && a[k][j]!=-1) {
                    if( a[i][j] > a[i][k] + a[k][j])
                        a[i][j] = a[i][k] + a[k][j];
                }
            }
        }

    }
}
```

Apéndice B

Código de planificación de consultas

La planificación de las consultas consiste en una lista de instrucciones. Se presenta el código para la máquina virtual de SQLite de las consultas que fueron utilizadas en el capitulo 7.

Para simplificar la lectura, se remplazaron los números de cursores y registros, por nombres. Al comienzo del cada código se presenta una tabla de conversión de nombres a números.

Cada instrucción esta compuesta por:

- `num`: Número de la instrucción, valor que toma el PC.
- `opcode`: Nombre de la instrucción.
- `p1`, `p2` y `p3`: Primeros tres parámetros, permiten identificar cursores, registros o valores de inicialización.
- `p4type`: Tipo del parámetro 4, este es especial y puede contener multiples tipos no básicos.
- `p4union`: Valor del parámetro 4.
- `p5`: Parámetro númerico, permite identificar propiedades especiales de una instrucción.
- `flags`: Parámetro interno que siver para modificar el comportamiento de una instrucción.

La columna restante de la lista de instrucción, describe el funcionamiento de cada instrucción.

Los parámetros identificados por una letra `r` al comienzo de su nombre, corresponden a registros, mientras que si comienzan con cualquier otra letra hacen referencia a cursores. Cuando un parámetro se encuentra entre paréntesis, es porque el número en su interior refiere a la dirección de un salto. Esto sucede para el caso del parámetros número dos, este es utilizado como destino de saltos en el caso general.

Todos los códigos presentados están divididos en dos partes por el texto `- core auxiliar -`, esta división es solamente ilustrativa, ya que ambas maquinas virtuales ejecutan la misma sección de código.

B.1 Consultas Básicas (S)

S_1

```
SELECT p.Name, j.Name FROM Jobs j, Persons p WHERE p.Job == j.Id
```

```
Cursores: J0=0 P0=1 I0=2 J1=3 P1=4 I1=5
Memorias: r01=1 r02=2 r03=3 r04=4 r11=5 r12=6 r13=7 r14=8

num| opcode    | p1 | p2 | p3 |  p4type  |   p4union  | p5 |flags|
---|-----------|----|----|----|----------|------------|----|-----|
  0| Init      |   0| (1)|   0|         0|           0|   0|  0x1| Start at 1
  1| OpenRead  |  P0|   4|   0|  P4_INT32|           3|   0|    0| root=4 iDb=0  Persons
  2| OpenRead  |  J0|   2|   0|  P4_INT32|           2|   0|    0| root=2 iDb=0  Jobs
  3| OpenRead  |  I0|   3|   0|P4_KEYINFO|     keyInfo|   2|    0| root=3 iDb=0  autoindex_Jobs_1
  4| OpenRead  |  P1|   4|   0|  P4_INT32|           3|   0|    0| root=4 iDb=0  Persons
  5| OpenRead  |  J1|   2|   0|  P4_INT32|           2|   0|    0| root=2 iDb=0  Jobs
  6| OpenRead  |  I1|   3|   0|P4_KEYINFO|     keyInfo|   2|    0| root=3 iDb=0  autoindex_Jobs_1
  7| DSplit    |   0|{29}|   0|         0|           0|   0|    0| SPLIT(29)
  8| Rewind    |  P0|(21)|   0|         0|           0|   0|  0x1| rewindCurr(P0), error goto 21
  9| Column    |  P0|   2| r01|         0|           0|   0|    0| r[r01]=Persons.Job
 10| IsNull    | r01|(20)|   0|         0|           0|   0|  0x3| if r[r01]==NULL goto 20
 11| Affinity  | r01|   1|   0|P4_DYNAMIC|         'D'|   0|    0| affinity(r[r01])
 12| SeekGE    |  I0|(20)| r01|  P4_INT32|           1|   0|  0x9| key=r[r01]
 13| IdxGT     |  I0|(20)| r01|  P4_INT32|           1|   0|  0x1| key=r[r01]
 14| IdxRowid  |  I0| r02|   0|         0|           0|   0| 0x10| r[r02]=rowid
 15| Seek      |  J0| r02|   0|         0|           0|   0|  0x4| intkey=r[r02]
 16| Column    |  P0|   0| r03|         0|           0|   0|    0| r[r03]=Persons.Name
 17| Column    |  J0|   0| r04|         0|           0|   0|    0| r[r04]=Jobs.Name
 18| ResultRow | r03|   2|   0|         0|           0|   0|    0| output=r[r03..r04]
 19| Next      |  I0|(13)|   0|P4_ADVANCE|   BtreeNext|   0|  0x1| next(I0) goto 13
 20| Next      |  P0| (9)|   0|P4_ADVANCE|   BtreeNext|   1|  0x1| next(P0) goto 9
 21| DJoin     |   0|   0|   0|         0|           0|   0|    0| JOIN()
 22| Close     |  P0|   0|   0|         0|           0|   0|    0| close(P0)
 23| Close     |  J0|   0|   0|         0|           0|   0|    0| close(J0)
 24| Close     |  I0|   0|   0|         0|           0|   0|    0| close(I0)
 25| Close     |  P1|   0|   0|         0|           0|   0|    0| close(P1)
 26| Close     |  J1|   0|   0|         0|           0|   0|    0| close(J1)
 27| Close     |  I1|   0|   0|         0|           0|   0|    0| close(I1)
 28| Halt      |   0|   0|   0|         0|           0|   0|    0|
 - core auxiliar -
 29| Last      |  P1|(42)|   0|         0|           0|   0|  0x1| lastCurr(P1), error goto 42
 30| Column    |  P1|   2| r11|         0|           0|   0|    0| r[r11]=Persons.Job
 31| IsNull    | r11|(41)|   0|         0|           0|   0|  0x3| if r[r11]==NULL goto 41
 32| Affinity  | r11|   1|   0|P4_DYNAMIC|         'D'|   0|    0| affinity(r[r11])
 33| SeekGE    |  I1|(41)| r11|  P4_INT32|           1|   0|  0x9| key=r[r11]
 34| IdxGT     |  I1|(41)| r11|  P4_INT32|           1|   0|  0x1| key=r[r11]
 35| IdxRowid  |  I1| r12|   0|         0|           0|   0| 0x10| r[r12]=rowid
 36| Seek      |  J1| r12|   0|         0|           0|   0|  0x4| intkey=r[r12]
 37| Column    |  P1|   0| r13|         0|           0|   0|    0| r[r13]=Persons.Name
 38| Column    |  J1|   0| r14|         0|           0|   0|    0| r[r14]=Jobs.Name
 39| ResultRow | r13|   2|   0|         0|           0|   0|    0| output=r[r13..r14]
 40| Next      |  I1|(34)|   0|P4_ADVANCE|   BtreeNext|   0|  0x1| next(I1) goto 34
 41| Prev      |  P1|(30)|   0|P4_ADVANCE|   BtreePrev|   1|  0x1| prev(P1) goto 30
 42| Goto      |   0|(21)|   0|         0|           0|   0|  0x1| goto 21
```

S_2

```
SELECT p.Name FROM Jobs j, Persons p WHERE p.Job == j.Id AND j.Name="T4"
```

```
Cursores: J0=0 P0=1 I0=2 J1=3 P1=4 I1=5
Memorias: r01=1 r02=2 r03=3 r04=4 r05=5
          r11=6 r12=7 r13=8 r14=4 r15=10

num| opcode    | p1 | p2 | p3 |  p4type  |   p4union  | p5 |flags|
---|-----------|----|----|----|----------|------------|----|-----|
  0| Init      |   0| (1)|   0|         0|           0|   0|  0x1| Start at 1
  1| OpenRead  |  P0|   4|   0|  P4_INT32|           3|   0|    0| root=4 iDb=0  Persons
  2| OpenRead  |  J0|   2|   0|  P4_INT32|           2|   0|    0| root=2 iDb=0  Jobs
  3| OpenRead  |  I0|   3|   0|P4_KEYINFO|     keyInfo|   2|    0| root=3 iDb=0  autoindex_Jobs_1
  4| OpenRead  |  P1|   4|   0|  P4_INT32|           3|   0|    0| root=4 iDb=0  Persons
  5| OpenRead  |  J1|   2|   0|  P4_INT32|           2|   0|    0| root=2 iDb=0  Jobs
  6| OpenRead  |  I1|   3|   0|P4_KEYINFO|     keyInfo|   2|    0| root=3 iDb=0  autoindex_Jobs_1
  7| String8   |   0| r04|   0|P4_DYNAMIC|        'T4'|   0| 0x10| r[r04]='T4'
  8| DSplit    |   0|{31}|   0|         0|           0|   0|    0| SPLIT(31)
  9| Rewind    |  P0|(23)|   0|         0|           0|   0|  0x1| rewindCurr(P0), error goto 23
 10| Column    |  P0|   2| r01|         0|           0|   0|    0| r[r01]=Persons.Job
 11| IsNull    | r01|(22)|   0|         0|           0|   0|  0x3| if r[r01]==NULL goto 22
 12| Affinity  | r01|   1|   0|P4_DYNAMIC|         'D'|   0|    0| affinity(r[r01])
 13| SeekGE    |  I0|(22)| r01|  P4_INT32|           1|   0|  0x9| key=r[r01]
 14| IdxGT     |  I0|(22)| r01|  P4_INT32|           1|   0|  0x1| key=r[r01]
 15| IdxRowid  |  I0| r02|   0|         0|           0|   0| 0x10| r[r02]=rowid
 16| Seek      |  J0| r02|   0|         0|           0|   0|  0x4| intkey=r[r02]
 17| Column    |  J0|   1| r03|         0|           0|   0|    0| r[r04]=Jobs.Name
 18| Ne        | r04|(21)| r03|P4_COLLSEQ|     CollSeq|0x52|  0xb| if r[r04]!=r[r03] goto 21
 19| Column    |  P0|   0| r05|         0|           0|   0|    0| r[r05]=Persons.Name
 20| ResultRow | r05|   1|   0|         0|           0|   0|    0| output=r[r05]
 21| Next      |  I0|(14)|   0|P4_ADVANCE|   BtreeNext|   0|  0x1| next(I0) goto 14
 22| Next      |  P0|(10)|   0|P4_ADVANCE|   BtreeNext|   1|  0x1| next(P0) goto 10
 23| DJoin     |   0|   0|   0|         0|           0|   0|    0| JOIN()
 24| Close     |  P0|   0|   0|         0|           0|   0|    0| close(P0)
 25| Close     |  J0|   0|   0|         0|           0|   0|    0| close(J0)
 26| Close     |  I0|   0|   0|         0|           0|   0|    0| close(I0)
 27| Close     |  P1|   0|   0|         0|           0|   0|    0| close(P1)
 28| Close     |  J1|   0|   0|         0|           0|   0|    0| close(J1)
 29| Close     |  I1|   0|   0|         0|           0|   0|    0| close(I1)
 30| Halt      |   0|   0|   0|         0|           0|   0|    0|
 - core auxiliar -
 31| Last      |  P1|(45)|   0|         0|           0|   0|  0x1| lastCurr(P1), error goto 45
 32| Column    |  P1|   2| r11|         0|           0|   0|    0| r[r11]=Persons.Job
 33| IsNull    | r11|(44)|   0|         0|           0|   0|  0x3| if r[r11]==NULL goto 44
 34| Affinity  | r11|   1|   0|P4_DYNAMIC|         'D'|   0|    0| affinity(r[r11])
 35| SeekGE    |  I1|(44)| r11|  P4_INT32|           1|   0|  0x9| key=r[r11]
 36| IdxGT     |  I1|(44)| r11|  P4_INT32|           1|   0|  0x1| key=r[r11]
 37| IdxRowid  |  I1| r12|   0|         0|           0|   0| 0x10| r[r12]=rowid
 38| Seek      |  J1| r12|   0|         0|           0|   0|  0x4| intkey=r[r12]
 39| Column    |  J1|   1| r13|         0|           0|   0|    0| r[r13]=Jobs.Name
 40| Ne        | r14|(43)| r13|P4_COLLSEQ|     CollSeq|0x52|  0xb| if r[r14]!=r[r13] goto 43
 41| Column    |  P1|   0| r15|         0|           0|   0|    0| r[r15]=Persons.Name
 42| ResultRow | r15|   1|   0|         0|           0|   0|    0| output=r[r15]
 43| Next      |  I1|(36)|   0|P4_ADVANCE|   BtreeNext|   0|  0x1| next(I1) goto 36
 44| Prev      |  P1|(32)|   0|P4_ADVANCE|   BtreePrev|   1|  0x1| prev(P1) goto 32
 45| Goto      |   0|(23)|   0|         0|           0|   0|  0x1| goto 23
```

S_3

```
SELECT p.Id, p.Job FROM Persons p
```

```
Cursores: P0=0 P1=1
Memorias: r01=1 r02=2 r11=3 r12=4

num| opcode    | p1 | p2 | p3 |  p4type  |   p4union  | p5 |flags|
---|-----------|----|----|----|----------|------------|----|-----|
  0| Init      |   0| (1)|   0|         0|           0|   0|  0x1| Start at 1
  1| OpenRead  |  P0|   4|   0|  P4_INT32|           3|   0|    0| root=4 iDb=0  Persons
  2| OpenRead  |  P1|   4|   0|  P4_INT32|           3|   0|    0| root=4 iDb=0  Persons
  3| DSplit    |   0|{13}|   0|         0|           0|   0|    0| SPLIT(13)
  4| Rewind    |  P0| (9)|   0|         0|           0|   0|  0x1| rewindCurr(P0), error goto 9
  5| Column    |  P0|   0| r01|         0|           0|   0|    0| r[r01]=Persons.Id
  6| Column    |  P0|   2| r02|         0|           0|   0|    0| r[r02]=Persons.Job
  7| ResultRow | r01|   2|   0|         0|           0|   0|    0| output=r[r01..r02]
  8| Next      |  P0| (5)|   0|P4_ADVANCE|   BtreeNext|   1|  0x1| next(P0) goto 5
  9| DJoin     |   0|   0|   0|         0|           0|   0|    0| JOIN()
 10| Close     |  P0|   0|   0|         0|           0|   0|    0| close(P0)
 11| Close     |  P1|   0|   0|         0|           0|   0|    0| close(P1)
 12| Halt      |   0|   0|   0|         0|           0|   0|    0|
 - core auxiliar -
 13| Last      |  P1| (9)|   0|         0|           0|   0|  0x1| lastCurr(P1), error goto 9
 14| Column    |  P1|   0| r11|         0|           0|   0|    0| r[r11]=Persons.Id
 15| Column    |  P1|   2| r12|         0|           0|   0|    0| r[r12]=Persons.Job
 16| ResultRow | r11|   2|   0|         0|           0|   0|    0| output=r[r11..r12]
 17| Prev      |  P1|(14)|   0|P4_ADVANCE|   BtreePrev|   1|  0x1| prev(P1) goto 14
 18| Goto      |   0| (9)|   0|         0|           0|   0|  0x1| goto 9
```

S_1a

```
SELECT sum(length(j.Name)) FROM Jobs j, Persons p WHERE p.Job == j.Id
```

```
Cursores: J0=0 P0=1 I0=2 J1=3 P1=4  I1=5
Memorias: r01=1  r02=2  r03=3   r04=4   r05=5   r06=6   r07=7
          r11=8  r12=9  r13=10  r14=11  r15=12  r16=13  r17=14

num| opcode    | p1 | p2 | p3 |  p4type  |   p4union  | p5 |flags|
---|-----------|----|----|----|----------|------------|----|-----|
  0| Init      |   0| (1)|   0|         0|           0|   0|  0x1| Start at 1
  1| Null      |   0| r01| r02|         0|           0|   0|    0| r[r01..r02]=NULL
  2| Null      |   0| r11| r12|         0|           0|   0|    0| r[r11..r12]=NULL
  3| OpenRead  |  P0|   4|   0|  P4_INT32|           3|   0|    0| root=4 iDb=0  Persons
  4| OpenRead  |  J0|   2|   0|  P4_INT32|           2|   0|    0| root=2 iDb=0  Jobs
  5| OpenRead  |  I0|   3|   0|P4_KEYINFO|     keyInfo|   2|    0| root=3 iDb=0  autoindex_Jobs_1
  6| OpenRead  |  P1|   4|   0|  P4_INT32|           3|   0|    0| root=4 iDb=0  Persons
  7| OpenRead  |  J1|   2|   0|  P4_INT32|           2|   0|    0| root=2 iDb=0  Jobs
  8| OpenRead  |  I1|   3|   0|P4_KEYINFO|     keyInfo|   2|    0| root=3 iDb=0  autoindex_Jobs_1
  9| DFunction0|[23]| r06| r05|P4_FUNCDEF|   length(1)|   1|    0| getContext(length,23)
 10| DFunction0|[50]| r16| r15|P4_FUNCDEF|   length(1)|   1|    0| getContext(length,50)
 11| DAggStep0 |[24]| r05| r01|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,24)
 12| DAggStep0 |[51]| r15| r11|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,51)
 13| DSplit    |   0|{41}|   0|         0|           0|   0|    0| SPLIT(41)
 14| Rewind    |  P0|(27)|   0|         0|           0|   0|  0x1| rewindCurr(P0), error goto 27
 15| Column    |  P0|   2| r03|         0|           0|   0|    0| r[r03]=Persons.Job
 16| IsNull    | r03|(26)|   0|         0|           0|   0|  0x3| if r[r03]==NULL goto 26
 17| Affinity  | r03|   1|   0|P4_DYNAMIC|         'D'|   0|    0| affinity(r[r03])
```

```
 18| SeekGE    |  I0|(26)| r03|  P4_INT32|           1|   0|  0x9| key=r[r03]
 19| IdxGT     |  I0|(26)| r03|  P4_INT32|           1|   0|  0x1| key=r[r03]
 20| IdxRowid  |  I0| r04|   0|         0|           0|   0| 0x10| r[r04]=rowid
 21| Seek      |  J0| r04|   0|         0|           0|   0|  0x4| intkey=r[r04]
 22| Column    |  J0|   1| r06|         0|           0|  64|    0| r[r06]=Jobs.Name
 23| Function  |   0| r06| r05|P4_FUNCCTX|   length(1)|   1|    0| r[r05]=length(r[r06@1])
 24| AggStep   |   0| r05| r01|P4_FUNCCTX|      sum(1)|   1|    0| accum=r[r01] step(r[r05@1])
 25| Next      |  I0|(19)|   0|P4_ADVANCE|   BtreeNext|   0|  0x1| next(I0) goto 19
 26| Next      |  P0|(15)|   0|P4_ADVANCE|   BtreeNext|   1|  0x1| next(P0) goto 15
 27| DJoin     |   0|   0|   0|         0|           0|   0|    0| JOIN()
 28| Close     |  P0|   0|   0|         0|           0|   0|    0| close(P0)
 29| Close     |  J0|   0|   0|         0|           0|   0|    0| close(J0)
 30| Close     |  I0|   0|   0|         0|           0|   0|    0| close(I0)
 31| Close     |  P1|   0|   0|         0|           0|   0|    0| close(P1)
 32| Close     |  J1|   0|   0|         0|           0|   0|    0| close(J1)
 33| Close     |  I1|   0|   0|         0|           0|   0|    0| close(I1)
 34| AggFinal  | r01|   1|   0|P4_FUNCDEF|      sum(1)|   1|    0| accum=r[r01] N=1
 35| Copy      | r01| r07|   0|         0|           0|   0|    0| r[r07@1]=r[r01@1]
 36| AggFinal  | r11|   1|   0|P4_FUNCDEF|      sum(1)|   1|    0| accum=r[r11] N=1
 37| Copy      | r11| r17|   0|         0|           0|   0|    0| r[r17@1]=r[r11@1]
 38| DSum      | r17| r07|   0|         0|           0|   0|    0| r[r17]=r[r17]+r[r07]
 39| ResultRow | r17|   1|   0|         0|           0|   0|    0| output=r[r17]
 40| Halt      |   0|   0|   0|         0|           0|   0|    0|
 - core auxiliar -
 41| Last      |  P1|(54)|   0|         0|           0|   0|  0x1| lastCurr(P1), error goto 54
 42| Column    |  P1|   2| r13|         0|           0|   0|    0| r[r13]=Persons.Job
 43| IsNull    | r13|(53)|   0|         0|           0|   0|  0x3| if r[r13]==NULL goto 53
 44| Affinity  | r13|   1|   0|P4_DYNAMIC|         'D'|   0|    0| affinity(r[r13])
 45| SeekGE    |  I1|(53)| r13|  P4_INT32|           1|   0|  0x9| key=r[r13]
 46| IdxGT     |  I1|(53)| r13|  P4_INT32|           1|   0|  0x1| key=r[r13]
 47| IdxRowid  |  I1| r14|   0|         0|           0|   0| 0x10| r[r14]=rowid
 48| Seek      |  J1| r14|   0|         0|           0|   0|  0x4| intkey=r[r14]
 49| Column    |  J1|   1| r16|         0|           0|  64|    0| r[r16]=Jobs.Name
 50| Function  |   0| r16| r15|P4_FUNCCTX|   length(1)|   1|    0| r[r15]=length(r[r16@1])
 51| AggStep   |   0| r15| r11|P4_FUNCCTX|      sum(1)|   1|    0| accum=r[r11] step(r[r15@1])
 52| Next      |  I1|(46)|   0|P4_ADVANCE|   BtreeNext|   0|  0x1| next(I1) goto 46
 53| Prev      |  P1|(42)|   0|P4_ADVANCE|   BtreePrev|   1|  0x1| prev(P1) goto 42
 54| Goto      |   0|(27)|   0|         0|           0|   0|  0x1| goto 27
```

S_2a

```
SELECT sum(length(p.Name)) FROM Jobs j, Persons p
WHERE p.Job == j.Id AND j.Name="T4"
```

```
Cursores: J0=0 P0=1 I0=2 J1=3 P1=4 I1=5
Memorias: r01=1  r02=2  r03=3  r04=4  r05=5  r06=6  r07=7  r08=8  r09=9
          r11=10 r12=11 r13=12 r14=13 r15=14 r16=6  r17=16 r18=17 r19=18

num| opcode    | p1 | p2 | p3 |  p4type  |   p4union  | p5 |flags|
---|-----------|----|----|----|----------|------------|----|-----|
  0| Init      |   0| (1)|   0|         0|           0|   0|  0x1| Start at 1
  1| Null      |   0| r01| r02|         0|           0|   0|    0| r[r01..r02]=NULL
  2| Null      |   0| r11| r12|         0|           0|   0|    0| r[r11..r12]=NULL
  3| OpenRead  |  P0|   4|   0|  P4_INT32|           3|   0|    0| root=4 iDb=0  Persons
  4| OpenRead  |  J0|   2|   0|  P4_INT32|           2|   0|    0| root=2 iDb=0  Jobs
  5| OpenRead  |  I0|   3|   0|P4_KEYINFO|     keyInfo|   2|    0| root=3 iDb=0  autoindex_Jobs_1
  6| OpenRead  |  P1|   4|   0|  P4_INT32|           3|   0|    0| root=4 iDb=0  Persons
  7| OpenRead  |  J1|   2|   0|  P4_INT32|           2|   0|    0| root=2 iDb=0  Jobs
```

```
 8| OpenRead  |  I1|   3|   0|P4_KEYINFO|     keyInfo|   2|    0| root=3 iDb=0  autoindex_Jobs_1
 9| DFunction0|[26]| r08| r07|P4_FUNCDEF|   length(1)|   1|    0| getContext(length,26)
10| DFunction0|[55]| r18| r17|P4_FUNCDEF|   length(1)|   1|    0| getContext(length,55)
11| DAggStep0 |[27]| r07| r01|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,27)
12| DAggStep0 |[56]| r17| r11|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,56)
13| String8   |   0| r06|   0|P4_DYNAMIC|        'T4'|   0| 0x10| r[r06]='T4'
14| DSplit    |   0|{44}|   0|         0|           0|   0|    0| SPLIT(44)
15| Rewind    |  P0|(30)|   0|         0|           0|   0|  0x1| rewindCurr(P0), error goto 30
16| Column    |  P0|   2| r03|         0|           0|   0|    0| r[r03]=Persons.Job
17| IsNull    | r03|(29)|   0|         0|           0|   0|  0x3| if r[r03]==NULL goto 29
18| Affinity  | r03|   1|   0|P4_DYNAMIC|         'D'|   0|    0| affinity(r[r03])
19| SeekGE    |  I0|(29)| r03|  P4_INT32|           1|   0|  0x9| key=r[r03]
20| IdxGT     |  I0|(29)| r03|  P4_INT32|           1|   0|  0x1| key=r[r03]
21| IdxRowid  |  I0| r04|   0|         0|           0|   0| 0x10| r[r04]=rowid
22| Seek      |  J0| r04|   0|         0|           0|   0|  0x4| intkey=r[r04]
23| Column    |  J0|   1| r05|         0|           0|   0|    0| r[r05]=Jobs.Name
24| Ne        | r06|(28)| r05|P4_COLLSEQ|     CollSeq|0x52|  0xb| if r[r06]!=r[r05] goto 28
25| Column    |  P0|   1| r08|         0|           0|   0|    0| r[r08]=Persons.Name
26| Function  |   0| r08| r07|P4_FUNCCTX|   length(1)|   1|    0| r[r07]=length(r[r08@1])
27| AggStep   |   0| r07| r01|P4_FUNCCTX|      sum(1)|   1|    0| accum=r[r01] step(r[r07@1])
28| Next      |  I0|(20)|   0|P4_ADVANCE|   BtreeNext|   0|  0x1| next(I0) goto 20
29| Next      |  P0|(16)|   0|P4_ADVANCE|   BtreeNext|   1|  0x1| next(P0) goto 16
30| DJoin     |   0|   0|   0|         0|           0|   0|    0| JOIN()
31| Close     |  P0|   0|   0|         0|           0|   0|    0| close(P0)
32| Close     |  J0|   0|   0|         0|           0|   0|    0| close(J0)
33| Close     |  I0|   0|   0|         0|           0|   0|    0| close(I0)
34| Close     |  P1|   0|   0|         0|           0|   0|    0| close(P1)
35| Close     |  J1|   0|   0|         0|           0|   0|    0| close(J1)
36| Close     |  I1|   0|   0|         0|           0|   0|    0| close(I1)
37| AggFinal  | r01|   1|   0|P4_FUNCDEF|      sum(1)|   1|    0| accum=r[r01] N=1
38| Copy      | r01| r09|   0|         0|           0|   0|    0| r[r09@1]=r[r01@1]
39| AggFinal  | r11|   1|   0|P4_FUNCDEF|      sum(1)|   1|    0| accum=r[r11] N=1
40| Copy      | r11| r19|   0|         0|           0|   0|    0| r[r19@1]=r[r11@1]
41| DSum      | r19| r09|   0|         0|           0|   0|    0| r[r19]=r[r19]+r[r09]
42| ResultRow | r19|   1|   0|         0|           0|   0|    0| output=r[r19]
43| Halt      |   0|   0|   0|         0|           0|   0|    0|
- core auxiliar -
44| Last      |  P1|(59)|   0|         0|           0|   0|  0x1| lastCurr(P1), error goto 59
45| Column    |  P1|   2| r13|         0|           0|   0|    0| r[r13]=Persons.Job
46| IsNull    | r13|(58)|   0|         0|           0|   0|  0x3| if r[r13]==NULL goto 58
47| Affinity  | r13|   1|   0|P4_DYNAMIC|         'D'|   0|    0| affinity(r[r13])
48| SeekGE    |  I1|(58)| r13|  P4_INT32|           1|   0|  0x9| key=r[r13]
49| IdxGT     |  I1|(58)| r13|  P4_INT32|           1|   0|  0x1| key=r[r13]
50| IdxRowid  |  I1| r14|   0|         0|           0|   0| 0x10| r[r14]=rowid
51| Seek      |  J1| r14|   0|         0|           0|   0|  0x4| intkey=r[r04]
52| Column    |  J1|   1| r15|         0|           0|   0|    0| r[r15]=Jobs.Name
53| Ne        | r16|(57)| r15|P4_COLLSEQ|     CollSeq|0x52|  0xb| if r[r16]!=r[r15] goto 57
54| Column    |  P1|   1| r18|         0|           0|   0|    0| r[r18]=Persons.Name
55| Function  |   0| r18| r17|P4_FUNCCTX|   length(1)|   1|    0| r[r17]=length(r[r18@1])
56| AggStep   |   0| r17| r11|P4_FUNCCTX|      sum(1)|   1|    0| accum=r[r11] step(r[r17@1])
57| Next      |  I1|(49)|   0|P4_ADVANCE|   BtreeNext|   0|  0x1| next(I1) goto 49
58| Prev      |  P1|(45)|   0|P4_ADVANCE|   BtreePrev|   1|  0x1| prev(P1) goto 45
59| Goto      |   0|(30)|   0|         0|           0|   0|  0x1| goto 30
```

S_3a

```
SELECT sum(length(name)) FROM Persons
```

```
Cursores: c0=0 c1=1
Memorias: r01=1 r02=2 r03=3 r04=4 r05=5
          r11=6 r12=7 r13=8 r14=9 r15=10

num| opcode    | p1 | p2 | p3 |  p4type  |   p4union  | p5 |flags|
---|-----------|----|----|----|----------|------------|----|-----|
  0| Init      |   0| (1)|   0|         0|           0|   0|  0x1| Start at 1
  1| Null      |   0| r01| r02|         0|           0|   0| 0x10| r[r01..r02]=NULL
  2| Null      |   0| r11| r12|         0|           0|   0| 0x10| r[r11..r12]=NULL
  3| OpenRead  |  c0|   4|   0|  P4_INT32|           2|   0|    0| root=4 iDb=0  Persons
  4| OpenRead  |  c1|   4|   0|  P4_INT32|           2|   0|    0| root=4 iDb=0  Persons
  5| DFunction0|[12]| r04| r03|P4_FUNCDEF|   length(1)|   1|    0| getContext(length,12)
  6| DFunction0|[27]| r14| r13|P4_FUNCDEF|   length(1)|   1|    0| getContext(length,27)
  7| DAggStep0 |[13]| r03| r01|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,13)
  8| DAggStep0 |[28]| r13| r11|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,28)
  9| DSplit    |   0|{25}|   0|         0|           0|   0|    0| SPLIT(25)
 10| Rewind    |  c0|(15)|   0|         0|           0|   0|  0x1| rewindCurr(c0), error goto 15
 11| Column    |  c0|   1| r04|         0|           0|  64|    0| r[r04]=Persons.Name
 12| Function  |   0| r04| r03|P4_FUNCCTX|   length(1)|   1|    0| r[r03]=length(r[r04@1])
 13| AggStep   |   0| r03| r01|P4_FUNCCTX|      sum(1)|   1|    0| accum=r[r01] step(r[r03@1])
 14| Next      |  c0|(11)|   0|P4_ADVANCE|   BtreeNext|   1|  0x1| next(c0) goto 11
 15| DJoin     |   0|   0|   0|         0|           0|   0|    0| JOIN()
 16| Close     |  c0|   0|   0|         0|           0|   0|    0| close(c0)
 17| Close     |  c1|   0|   0|         0|           0|   0|    0| close(c1)
 18| AggFinal  | r01|   1|   0|P4_FUNCDEF|      sum(1)|   0|    0| accum=r[r01] N=1
 19| Copy      | r01| r05|   0|         0|           0|   0|    0| r[r05@1]=r[r01@1]
 20| AggFinal  | r11|   1|   0|P4_FUNCDEF|      sum(1)|   0|    0| accum=r[r11] N=1
 21| Copy      | r11| r15|   0|         0|           0|   0|    0| r[r15@1]=r[r11@1]
 22| DSum      | r15| r05|   0|         0|           0|   0|    0| r[r15]=r[r15]+r[r05]
 23| ResultRow | r15|   1|   0|         0|           0|   0|    0| output=r[r15]
 24| Halt      |   0|   0|   0|         0|           0|   0|    0|
 - core auxiliar -
 25| Last      |  c1|(30)|   0|         0|           0|   0|  0x1| lastCurr(c1), error goto 30
 26| Column    |  c1|   1| r14|         0|           0|  64|    0| r[r14]=Persons.Name
 27| Function  |   0| r14| r13|P4_FUNCCTX|   length(1)|   1|    0| r[r13]=length(r[r14@1])
 28| AggStep   |   0| r13| r11|P4_FUNCCTX|      sum(1)|   1|    0| accum=r[r11] step(r[r13@1])
 29| Prev      |  c1|(26)|   0|P4_ADVANCE|   BtreePrev|   1|  0x1| prev(c1) goto 26
 30| Goto      |   0|(15)|   0|         0|           0|   0|  0x1| goto 15
```

B.2 Consultas derivadas de TPC-H (H)

H_1

```
SELECT sum(l_extendedprice * l_discount) AS promo_revenue FROM lineitem
WHERE l_shipdate >= '1994-01-01' AND l_shipdate <1995-01-01 AND l_discount <0.07
AND l_discount >0.05 AND l_quantity <24
```

```
Cursores: cA0=0 cB0=1
Memorias: rA1=1   rA3=3   rA4=4   rA5=5   rA6=6   rA7=7   rA8=8   rA9=9
          rA10=10 rA11=11 rA12=12 rA13=13 rA15=15 rA16=16 rA17=17
          rB1=21  rB3=23  rB4=24  rB5=25  rB6=26  rB7=27  rB8=28  rB9=29
          rB10=30 rB11=31 rB12=32 rB13=33 rB15=35 rB16=36 rB17=37
```

```
num| opcode    | p1 | p2 | p3 |  p4type  |   p4union  | p5 |flags|
---|-----------|----|----|----|----------|------------|----|-----|
  0| Init      |   0| (1)|   0|         0|           0|   0|  0x1| Start at 1
  1| Null      |   0| rA1| rA3|         0|           0|   0| 0x10| r[rA1..rA3]=NULL
  2| Null      |   0| rB1| rB3|         0|           0|   0| 0x10| r[rB1..rB3]=NULL
  3| OpenRead  | cA0|  10|   0|  P4_INT32|          11|   0|    0| root=10 iDb=0  LineItem
  4| OpenRead  | cB0|  10|   0|  P4_INT32|          11|   0|    0| root=10 iDb=0  LineItem
  5| String8   |   0|rA16|   0|P4_DYNAMIC|'1994-01-01'|   0| 0x10| r[rA16]='1994-01-01'
  6| Function0 |   1|rA16| rA5|P4_FUNCDEF|     date(1)|   1|    0| r[rA5]=date(r[rA16@1])
  7| String8   |   0|rA17|   0|P4_DYNAMIC|'1995-01-01'|   0| 0x10| r[rA17]='1995-01-01'
  8| Function0 |   1|rA17| rA7|P4_FUNCDEF|     date(1)|   1|    0| r[rA7]=date(r[rA17@1])
  9| Real      |   0| rA9|   0|   P4_REAL|(double)0.05|   0| 0x10| r[rA9]=0.05
 10| Real      |   0|rA10|   0|   P4_REAL|(double)0.07|   0| 0x10| r[rA10]=0.07
 11| Integer   |  24|rA11|   0|         0|           0|   0| 0x10| r[rA11]=24
 12| String8   |   0|rB16|   0|P4_DYNAMIC|'1994-01-01'|   0| 0x10| r[rB16]='1994-01-01'
 13| Function0 |   1|rB16| rB5|P4_FUNCDEF|     date(1)|   1|    0| r[rb5]=date(r[rB17@1])
 14| String8   |   0|rB17|   0|P4_DYNAMIC|'1995-01-01'|   0| 0x10| r[rB17]='1995-01-01'
 15| Function0 |   1|rB17| rB7|P4_FUNCDEF|     date(1)|   1|    0| r[rB7]=date(r[rB17@1])
 16| Real      |   0| rB9|   0|   P4_REAL|(double)0.05|   0| 0x10| r[rB9]=0.05
 17| Real      |   0|rB10|   0|   P4_REAL|(double)0.07|   0| 0x10| r[rB10]=0.07
 18| Integer   |  24|rB11|   0|         0|           0|   0| 0x10| r[rB11]=24
 19| DAggStep0 |[33]|rA12| rA1|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,33)
 20| DAggStep0 |[56]|rB12| rB1|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,56)
 21| DSplit    |   0|{45}|   0|         0|           0|   0|    0| SPLIT(45)
 22| Rewind    | cA0|(35)|   0|         0|           0|   0|  0x1| rewindCurr(cA0), error goto 35
 23| Column    | cA0|  10| rA4|         0|           0|   0|    0| r[rA4]=LineItem[10]
 24| Lt        | rA5|(34)| rA4|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rA5]<r[rA4] goto 34
 25| Ge        | rA7|(34)| rA4|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rA7]>=r[rA4] goto 34
 26| Column    | cA0|   6| rA6|         0|           0|   0|    0| r[rA6]=LineItem[6]
 27| Lt        | rA9|(34)| rA6|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rA9]<r[rA6] goto 34
 28| Gt        |rA10|(34)| rA6|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rA10]>r[rA6] goto 34
 29| Column    | cA0|   4| rA8|         0|           0|   0|    0| r[rA8]=LineItem[4]
 30| Ge        |rA11|(34)| rA8|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rA11]>=r[rA8] goto 34
 31| Column    | cA0|   5|rA13|         0|           0|   0|    0| r[rA13]=LineItem[5]
 32| Multiply  | rA6|rA13|rA12|         0|           0|   0| 0x26| r[rA12]=r[rA13]*r[rA6]
 33| AggStep   |   0|rA12| rA1|P4_FUNCDEF|      sum(1)|   1|    0| accum=r[rA1] step(r[rA12@1])
 34| Next      | cA0|(23)|   0|P4_ADVANCE|   BtreeNext|   1|  0x1| next(cA0) goto 23
 35| DJoin     |   0|   0|   0|         0|           0|   0|    0| JOIN()
 36| Close     | cA0|   0|   0|         0|           0|   0|    0| close(cA0)
 37| Close     | cB0|   0|   0|         0|           0|   0|    0| close(cB0)
 38| AggFinal  | rA1|   1|   0|P4_FUNCDEF|      sum(1)|   0|    0| accum=r[rA1] N=1
 39| Copy      | rA1|rA15|   0|         0|           0|   0|    0| r[rA15@1]=r[rA1@1]
 40| AggFinal  | rB1|   1|   0|P4_FUNCDEF|      sum(1)|   0|    0| accum=r[rB1] N=1
 41| Copy      | rB1|rB15|   0|         0|           0|   0|    0| r[rB15@1]=r[rB1@1]
 42| DSum      |rA15|rB15|   0|         0|           0|   0|    0| r[rA15]=r[rA15]+r[rB15]
 43| ResultRow |rA15|   1|   0|         0|           0|   0|    0| output=r[rA15]
 44| Halt      |   0|   0|   0|         0|           0|   0|    0|
 - core auxiliar -
 45| Last      | cB0|(58)|   0|         0|           0|   0|  0x1| lastCurr(cB0), error goto 58
 46| Column    | cB0|  10| rB4|         0|           0|   0|    0| r[rB4]=LineItem[10]
 47| Lt        | rB5|(57)| rB4|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rB5]<r[rB4] goto 57
 48| Ge        | rB7|(57)| rB4|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rB7]>=r[rB4] goto 57
 49| Column    | cB0|   6| rB6|         0|           0|   0|    0| r[rB6]=LineItem[6]
 50| Lt        | rB9|(57)| rB6|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rB9]<r[rB6] goto 57
 51| Gt        |rB10|(57)| rB6|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rB10]>r[rB6] goto 57
 52| Column    | cB0|   4| rB8|         0|           0|   0|    0| r[rB8]=LineItem[4]
 53| Ge        |rB11|(57)| rB8|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rB11]>=r[rB8] goto 57
 54| Column    | cB0|   5|rB13|         0|           0|   0|    0| r[rB13]=LineItem[5]
```

```
 55| Multiply  | rB6|rB13|rB12|         0|           0|   0| 0x26| r[rB12]=r[rB13]*r[rB6]
 56| AggStep   |   0|rB12| rB1|P4_FUNCDEF|      sum(1)|   1|    0| accum=r[rB1] step(r[rB12@1])
 57| Prev      | cB0|(46)|   0|P4_ADVANCE|   BtreePrev|   1|  0x1| prev(cB0) goto 46
 58| Goto      |   0|(35)|   0|         0|           0|   0|  0x1| goto 35
```

H_2

```
SELECT sum(l_extendedprice * (1-l_discount)) AS promo_revenue
FROM lineitem, part
WHERE l_partkey = p_partkey AND l_shipdate >= 1995-09-01
AND l_shipdate <1995-10-01
```

```
Cursores: cA0=0 cA1=1 cB0=2 cB1=3
Memorias: rA1=1   rA3=3   rA4=4   rA5=5   rA6=6   rA7=7   rA8=8  rA9=9
          rA10=10 rA11=11 rA12=12 rA13=13 rA14=14 rA15=15
          rB1=21  rB3=23  rB4=24  rB5=25  rB6=26  rB7=27  rB8=28 rB9=29
          rB10=30 rB11=31 rB12=32 rB13=33 rB14=34 rB15=35

num| opcode    | p1 | p2 | p3 |  p4type  |   p4union  | p5 |flags|
---|-----------|----|----|----|----------|------------|----|-----|
  0| Init      |   0| (1)|   0|         0|           0|   0|  0x1| Start at 1
  1| Null      |   0| rA1| rA3|         0|           0|   0| 0x10| r[rA1..rA3]=NULL
  2| OpenRead  | cA0|  10|   0|  P4_INT32|          11|   0|    0| root=10 iDb=0  LineItem
  3| OpenRead  | cA1|   4|   0|  P4_INT32|           0|   0|    0| root=4  iDb=0  Part
  4| String8   |   0|rA14|   0|P4_DYNAMIC|'1995-09-01'|   0| 0x10| r[rA14]='1995-09-01'
  5| Function0 |   1|rA14| rA5|P4_FUNCDEF|     date(1)|   1|    0| r[rA5]=date(r[rA14@1])
  6| String8   |   0|rA15|   0|P4_DYNAMIC|'1995-10-01'|   0| 0x10| r[rA15]='1995-10-01'
  7| Function0 |   1|rA15| rA7|P4_FUNCDEF|     date(1)|   1|    0| r[rA7]=date(r[rA15@1])
  8| Integer   |   1|rA11|   0|         0|           0|   0| 0x10| r[rA11]=1
  9| Null      |   0| rB1| rB3|         0|           0|   0| 0x10| r[rB1..rB3]=NULL
 10| OpenRead  | cB0|  10|   0|  P4_INT32|          11|   0|    0| root=10 iDb=0  LineItem
 11| OpenRead  | cB1|   4|   0|  P4_INT32|           0|   0|    0| root=4  iDb=0  Part
 12| String8   |   0|rB14|   0|P4_DYNAMIC|'1995-09-01'|   0| 0x10| r[rB14]='1995-09-01'
 13| Function0 |   1|rB14| rB5|P4_FUNCDEF|     date(1)|   1|    0| r[rB5]=date(r[rB14@1])
 14| String8   |   0|rB15|   0|P4_DYNAMIC|'1995-10-01'|   0| 0x10| r[rB15]='1995-10-01'
 15| Function0 |   1|rB15| rB7|P4_FUNCDEF|     date(1)|   1|    0| r[rB7]=date(r[rB15@1])
 16| Integer   |   1|rB11|   0|         0|           0|   0| 0x10| r[rB11]=1
 17| DAggStep0 |[31]| rA9| rA1|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,31)
 18| DAggStep0 |[56]| rB9| rB1|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,56)
 19| DSplit    |   0|{45}|   0|         0|           0|   0|    0| SPLIT(45)
 20| Rewind    | cA0|(33)|   0|         0|           0|   0|  0x1| rewindCurr(cA0), error goto 33
 21| Column    | cA0|  10| rA4|         0|           0|   0|    0| r[rA4]=LineItem[10]
 22| Lt        | rA5|(32)| rA4|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rA5]<r[rA4] goto 32
 23| Ge        | rA7|(32)| rA4|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rA7]>=r[rA4] goto 32
 24| Column    | cA0|   1| rA8|         0|           0|   0|    0| r[rA8]=LineItem[1]
 25| MustBeInt | rA8|(32)|   0|         0|           0|   0|  0x3| int(r[rA8]), error goto 32
 26| NotExists | cA1|(32)| rA8|         0|           0|   0|  0x9| intkey=r[rA8], error goto 32
 27| Column    | cA0|   5| rA6|         0|           0|   0|    0| r[rA6]=LineItem[5]
 28| Column    | cA0|   6|rA12|         0|           0|   0|    0| r[rA12]=LineItem[6]
 29| Subtract  |rA12|rA11|rA10|         0|           0|   0| 0x26| r[rA10]=r[rA11]-r[rA12]
 30| Multiply  |rA10| rA6| rA9|         0|           0|   0| 0x26| r[rA9]=r[rA6]*r[rA10]
 31| AggStep   |   0| rA9| rA1|P4_FUNCDEF|      sum(1)|   1|    0| accum=r[rA1] step(r[rA9@1])
 32| Next      | cA0|(21)|   0|P4_ADVANCE|   BtreeNext|   1|  0x1| next(cA0) goto 21
 33| DJoin     |   0|   0|   0|         0|           0|   0|    0| JOIN()
 34| Close     | cA0|   0|   0|         0|           0|   0|    0| close(cA0)
 35| Close     | cA1|   0|   0|         0|           0|   0|    0| close(cA1)
 36| Close     | cB0|   0|   0|         0|           0|   0|    0| close(cB0)
```

```
 37| Close     | cB1|   0|   0|         0|           0|   0|    0| close(cB1)
 38| AggFinal  | rA1|   1|   0|P4_FUNCDEF|      sum(1)|   0|    0| accum=r[rA1] N=1
 39| Copy      | rA1|rA13|   0|         0|           0|   0|    0| r[rA13@1]=r[rA1@1]
 40| AggFinal  | rB1|   1|   0|P4_FUNCDEF|      sum(1)|   0|    0| accum=r[rB1] N=1
 41| Copy      | rB1|rB13|   0|         0|           0|   0|    0| r[rB13@1]=r[rB1@1]
 42| DSum      |rA13|rB13|   0|         0|           0|   0|    0| r[rA13]=r[rA13]+r[rB13]
 43| ResultRow |rA13|   1|   0|         0|           0|   0|    0| output=r[rA13]
 44| Halt      |   0|   0|   0|         0|           0|   0|    0|
 - core auxiliar -
 45| Last      | cB0|(58)|   0|         0|           0|   0|  0x1| lastCurr(cB0), error goto 58
 46| Column    | cB0|  10| rB4|         0|           0|   0|    0| r[rB4]=LineItem[10]
 47| Lt        | rB5|(57)| rB4|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rB5]<r[rB4] goto 57
 48| Ge        | rB7|(57)| rB4|P4_COLLSEQ|     CollSeq|  83|  0xb| if r[rB7]>=r[rB4] goto 57
 49| Column    | cB0|   1| rB8|         0|           0|   0|    0| r[rB8]=LineItem[1]
 50| MustBeInt | rB8|(57)|   0|         0|           0|   0|  0x3| int(r[rB8]), error goto 57
 51| NotExists | cB1|(57)| rB8|         0|           0|   0|  0x9| intkey=r[rB8], error goto 57
 52| Column    | cB0|   5| rB6|         0|           0|   0|    0| r[rB6]=LineItem[5]
 53| Column    | cB0|   6|rB12|         0|           0|   0|    0| r[rB12]=LineItem[6]
 54| Subtract  |rB12|rB11|rB10|         0|           0|   0| 0x26| r[rB10]=r[rB11]-r[rB12]
 55| Multiply  |rB10| rB6| rB9|         0|           0|   0| 0x26| r[rB9]=r[rB6]*r[rB10]
 56| AggStep   |   0| rB9| rB1|P4_FUNCDEF|      sum(1)|   1|    0| accum=r[rB1] step(r[rB9@1])
 57| Prev      | cB0|(46)|   0|P4_ADVANCE|   BtreePrev|   1|  0x1| prev(cB0) goto 46
 58| Goto      |   0|(33)|   0|         0|           0|   0|  0x1| goto 33
```

H_3

```
SELECT sum(l_extendedprice * (1-l_discount)) AS promo_revenue
FROM lineitem, part WHERE l_partkey = p_partkey
```

```
Cursores: cA0=0 cA1=1 cB0=2 cB1=3
Memorias: rA1=1  rA3=3  rA4=4  rA5=5  rA6=6  rA7=7  rA8=8  rA9=9  rA10=10
          rB1=11 rB3=13 rB4=14 rB5=15 rB6=16 rB7=17 rB8=18 rB9=19 rB10=20

num| opcode    | p1 | p2 | p3 |  p4type  |   p4union  | p5 |flags|
---|-----------|----|----|----|----------|------------|----|-----|
  0| Init      |   0| (1)|   0|         0|           0|   0|  0x1| Start at 1
  1| Null      |   0| rA1| rA3|         0|           0|   0| 0x10| r[rA1..rA3]=NULL
  2| Integer   |   1| rA8|   0|         0|           0|   0| 0x10| r[rA8]=1
  3| OpenRead  | cA0|  10|   0|  P4_INT32|           7|   0|    0| root=10 iDb=0  LineItem
  4| OpenRead  | cA1|   4|   0|  P4_INT32|           0|   0|    0| root=4  iDb=0  Part
  5| Null      |   0| rB1| rB3|         0|           0|   0| 0x10| r[rB1..rB3]=NULL
  6| Integer   |   1| rB8|   0|         0|           0|   0| 0x10| r[rB8]=1
  7| OpenRead  | cB0|  10|   0|  P4_INT32|           7|   0|    0| root=10 iDb=0  LineItem
  8| OpenRead  | cB1|   4|   0|  P4_INT32|           0|   0|    0| root=4  iDb=0  Part
  9| DAggStep0 |[20]| rA5| rA1|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,20)
 10| DAggStep0 |[42]| rB5| rB1|P4_FUNCDEF|      sum(1)|   1|    0| getContext(sum,42)
 11| DSplit    |   0|{34}|   0|         0|           0|   0|    0| SPLIT(34)
 12| Rewind    | cA0|(22)|   0|         0|           0|   0|  0x1| rewindCurr(cA0), error goto 22
 13| Column    | cA0|   1| rA4|         0|           0|   0|    0| r[rA4]=LineItem[1]
 14| MustBeInt | rA4|(21)|   0|         0|           0|   0|  0x3| int(r[rA4]), error goto 21
 15| NotExists | cA1|(21)| rA4|         0|           0|   0|  0x9| intkey=r[rA4], error goto 21
 16| Column    | cA0|   5| rA6|         0|           0|   0|    0| r[rA6]=LineItem[5]
 17| Column    | cA0|   6| rA9|         0|           0|   0|    0| r[rA9]=LineItem[6]
 18| Subtract  | rA9| rA8| rA7|         0|           0|   0| 0x26| r[rA7]=r[rA8]-r[rA9]
 19| Multiply  | rA7| rA6| rA5|         0|           0|   0| 0x26| r[rA5]=r[rA6]*r[rA7]
 20| AggStep   |   0| rA5| rA1|P4_FUNCDEF|      sum(1)|   1|    0| accum=r[rA1] step(r[rA5@1])
 21| Next      | cA0|(13)|   0|P4_ADVANCE|   BtreeNext|   1|  0x1| next(cA0) goto 13
 22| DJoin     |   0|   0|   0|         0|           0|   0|    0| JOIN()
```

```
23| Close     | cA0|   0|   0|         0|           0|   0|    0| close(cA0)
24| Close     | cA1|   0|   0|         0|           0|   0|    0| close(cA1)
25| Close     | cB0|   0|   0|         0|           0|   0|    0| close(cB0)
26| Close     | cB1|   0|   0|         0|           0|   0|    0| close(cB1)
27| AggFinal  | rA1|   1|   0|P4_FUNCDEF|      sum(1)|   0|    0| accum=r[rA1] N=1
28| Copy      | rA1|rA10|   0|         0|           0|   0|    0| r[rA10@1]=r[rA1@1]
29| AggFinal  | rB1|   1|   0|P4_FUNCDEF|      sum(1)|   0|    0| accum=r[rB1] N=1
30| Copy      | rB1|rB10|   0|         0|           0|   0|    0| r[rB10@1]=r[rB1@1]
31| DSum      |rA10|rB10|   0|         0|           0|   0|    0| r[rA10]=r[rA10]+r[rB10]
32| ResultRow |rA10|   1|   0|         0|           0|   0|    0| output=r[rA10]
33| Halt      |   0|   0|   0|         0|           0|   0|    0|
- core auxiliar -
34| Last      | cB0|(44)|   0|         0|           0|   0|  0x1| lastCurr(cB0), error goto 44
35| Column    | cB0|   1| rB4|         0|           0|   0|    0| r[rB4]=LineItem[1]
36| MustBeInt | rB4|(43)|   0|         0|           0|   0|  0x3| int(r[rB4]), error goto 43
37| NotExists | cB1|(43)| rB4|         0|           0|   0|  0x9| intkey=r[rB4], error goto 43
38| Column    | cB0|   5| rB6|         0|           0|   0|    0| r[rB6]=LineItem[5]
39| Column    | cB0|   6| rB9|         0|           0|   0|    0| r[rB9]=LineItem[6]
40| Subtract  | rB9| rB8| rB7|         0|           0|   0| 0x26| r[rB7]=r[rB8]-r[rB9]
41| Multiply  | rB7| rB6| rB5|         0|           0|   0| 0x26| r[rB5]=r[rB6]*r[rB7]
42| AggStep   |   0| rB5| rB1|P4_FUNCDEF|      sum(1)|   1|    0| accum=r[rB1] step(r[rB5@1])
43| Prev      | cB0|(35)|   0|P4_ADVANCE|   BtreePrev|   1|  0x1| prev(cB0) goto 35
44| Goto      |   0|(22)|   0|         0|           0|   0|  0x1| goto 22
```