Arquitectura de Computadores II, proyecto 1: Procesadores MP y False-Sharing

1° Ignacio Morales Chang
Ingeniería en Computadores
Tecnológico de Costa Rica
Cartago, Costa Rica
ig.morales@estudiantec.cr

***Resumen*—Este proyecto tiene como objetivo principal abordar el concepto de false sharing en sistemas multiprocesador (MP) y su impacto en el rendimiento. Para lograrlo, se plantea, en primer lugar, comprender a fondo el concepto de false sharing y cómo afecta el desempeño de un sistema MP. Esto implica analizar la importancia de las cachés para mejorar el rendimiento de un procesador y cómo la coherencia de caché influye en el manejo correcto de la memoria compartida. Se requiere diseñar pruebas de exploración que permitan detectar y observar el efecto de false sharing en diferentes situaciones. Se utilizan herramientas de perfilado y análisis: Google Benchmark con libpfm y Perf, y se evalua el fenómeno de false sharing en C++. Además, se analiza las características específicas del hardware objetivo al implementar software, para ver el efecto del false sharing en diferentes configuraciones de hardware. Por último, se requiere implementar al menos dos medidas para mitigar el false sharing y evaluar su efectividad en la mejora del rendimiento.**

***Palabras clave*—False Sharing, Cache Coherency, Benchmarks, Profiling**

I. Introducción

En programación MP, las cachés desempeñan un papel muy importante en la optimización del rendimiento del procesador. Esto se debe a que las cachés actúan como un almacenamiento intermedio de datos y permiten a la CPU acceder a información más rápido que si tuviera que acceder a ella directamente desde memoria principal.

La coherencia de caché garantiza que todos los núcleos de procesador tengan una vista de la memoria compartida, esto es muy importante porque sin ella, los diferentes núcleos podrían tener versiones desactualizadas de los datos en sus cachés, lo que llevaría a errores en la ejecución del software.

Cuando se implementa software con paralelismo, como aplicaciones multiproceso o multithreading, el fenómeno de false sharing se convierte en una preocupación importante. False sharing ocurre cuando múltiples hilos o núcleos comparten la misma línea de caché, aunque solo están accediendo a partes separadas de la línea. Esto puede provocar invalidaciones frecuentes de caché y una competencia innecesaria por los recursos de caché, lo que reduce significativamente el rendimiento del sistema.

Por lo tanto, en el proyecto de MP Programming and False Sharing, se aborda todo lo mencionado anteriormente. Los objetivos incluyen la investigación en profundidad de false sharing, la creación de pruebas para detectarlo, el uso de herramientas de perfilado para evaluar su impacto y la implementación de medidas para disminuir sus efectos perjudiciales.

II. Componentes del sistema

II-A. *Benchmarks*

Los benchmarks fueron escritos en C++ y corren con ayuda de Google Benchmarks. Se realizaron 5 benchmarks, pero antes veremos las variables globale definidas y la operación principal que ejecutarán los benchmarks.

II-A1. Variables globales: Inicialmente se declaran variables globales necesarias para los algoritmos para maximizar el efecto de false sharing, esto se muestra en la figura 1. Entre estas variables se encuentran:

- num_cores: Cantidad de núcleos disponibles
- cache_lsize: Tamaño de línea de cache
- num_threads: Cantidad de hilos disponibles
- int_per_line: Cantidad de atomic¡int¿que caben dentro de una línea de cache
- int_per_thread: Cantidad de elementos que le corresponde a cada hilo para una distribución uniforme

```
int num_cores = sysconf(_SC_NPROCESSORS_ONLN);
long cache_lsize = sysconf(_SC_LEVEL1_DCACHE_LINESIZE);

unsigned int num_threads = std::thread::hardware_concurrency();

const int int_per_line = cache_lsize / sizeof(std::atomic<int>);
const int int_per_thread = int_per_line / num_threads;
```

Figura 1. Variables globales

II-A2. Work: Esta es la función principal y la que se encarga de modificar los valores de las variables con las que se está trabajando. Como se muestra en la figura 2, la función se encarga de sumar 1 a la variable 100000 veces.

II-A3. Single Thread: El benchmark mostrado en la figura 3 utiliza un solo hilo por lo que no se presenta el efecto de false sharing. Este benchmark llama la operación work() pasando como parámetro .ªtomic¡int¿a". La operación se llama ïnt_per_line"veces para que se realice la misma cantidad de veces que en el resto de benchmarks como veremos a continuación.

```cpp
void work(std::atomic<int>& a) {
  for (int i = 0; i < 100000; i++) {
    a++;
  }
}
```

Figura 2. Función work()

```cpp
void single_thread() {
  std::atomic<int> a;
  a = 0;

  for (int i = 0; i < int_per_line; i++){
    work(a);
  }
}

static void singleThread(benchmark::State& s) {
  while (s.KeepRunning()) {
    single_thread();
  }
}
BENCHMARK(singleThread)->Unit(benchmark::kMillisecond);
```

Figura 3. Benchmark single thread

II-A4. Direct Sharing: Este benchmark mostrado en la figura 4 muestra el efecto de *direct sharing*. Se crean num_threads hilos y estos hilos comparten la variable .ª". Aquí se crean "num_threads"hilos, los cuales llaman la operación "work(a)int_per_thread"veces. Los hilos se guardan en un vector llamado "threadVecz este se itera para realizar el "join"de los hilos.

```cpp
void shared_var() {

  std::vector<std::thread> threadVec;

  std::atomic<int> a;
  a = 0;

  // Create threads and use lambda to launch work
  for (int i = 0; i < num_threads; i++) {
    threadVec.emplace_back([&]() {
      for (int j = 0; j < int_per_thread; j++){
        work(a);
      }
    });
  }

  // Join the threads
  for (std::thread& thread : threadVec) {
    thread.join();
  }

}

static void directSharing(benchmark::State& s) {
  while (s.KeepRunning()) {
    shared_var();
  }
}
BENCHMARK(directSharing)->UseRealTime()->Unit(benchmark::kMillisecond);
```

Figura 4. Benchmark direct sharing

II-A5. False Sharing: Este benchmark es el que muestra el efecto de false sharing. Como se observa en la figura 5, ahora se declaran dos vectores, uno para los hilos y otro para las variables que se van a procesar, este último se llama intVec y es de tamaño ïnt_per_line". Ahora en lugar de trabajar con variable compartida, se utilizan distintas variables por cada hilo. Incluso dentro de cada hilo, cada vez que se llama "work()", se pasa como parámetro una variable distinta.

```cpp
void diff_var() {

  std::vector<std::atomic<int>> intVec(int_per_line);
  std::vector<std::thread> threadVec;

  for (int i = 0; i < int_per_line; i++) {
    intVec[i] = 0;
  }

  // Create four threads and use lambda to launch work
  for (int i = 0; i < num_threads; i++) {
    threadVec.emplace_back([&]() {
      for(int j = 0; j < int_per_thread; j++){
        work(intVec[i + j]);
      }
    });
  }

  // Join the threads
  for (std::thread& thread : threadVec) {
    thread.join();
  }
}

static void falseSharing(benchmark::State& s) {
  while (s.KeepRunning()) {
    diff_var();
  }
}
BENCHMARK(falseSharing)->UseRealTime()->Unit(benchmark::kMillisecond);
```

Figura 5. Benchmark false sharing

II-A6. Padding: Este benchmark se muestra en la figura 6 y consiste en mitigar los efectos de false sharing. Aquí la variable se guarda dentro de un struct de tamaño de la línea de cache para asegurar que cada variable se encuentre en una línea de cache distinta y así no tener false sharing.

II-A7. Thread Local: El benchmark mostrado en la figura 7 es otra forma de eliminar false sharing. Aquí lo se se hace es guardar la variable como thread local, lo que lo hace accesible únicamente por el thread con el que se está tratando.

II-B. Automatización

Este componente tiene la función de correr los benchmarks repetidas veces con el fin de realizar el perfilado de los benchmarks, extraer los resultados y analizarlos. Para esto se utiliza Python y la biblioteca "subprocessçomo se muestra en la figura !!!!!!!!!!!!!!!!!!!!!!!!!!!!.

III. Propuesta de diseño

III-A. Benchmarks

Para mostrar el efecto de *false sharing* se propone implementar cinco benchmarks diferentes en un sistema con múltiples hilos. Para el benchmark de false sharing se tomó en cuenta el tamaño de la línea de caché y crear cuantas variables quepan y se distribuyeron entre los hilos disponibles, esto para

```cpp
struct alignas(64) AlignedType {
  AlignedType() { val = 0; }
  std::atomic<int> val;
};

void diff_line() {

  std::vector<AlignedType> intVec(num_cores);
  std::vector<std::thread> threadVec;


  for (int i = 0; i < num_threads; i++) {
    threadVec.emplace_back([&intVec, i]() {
      for (int j = 0; j < int_per_thread; j++){
        work(intVec[i].val);
      }
    });
  }

  // Join the threads
  for (std::thread& thread : threadVec) {
    thread.join();
  }
}

static void padding(benchmark::State& s) {
  while (s.KeepRunning()) {
    diff_line();
  }
}
BENCHMARK(padding)->UseRealTime()->Unit(benchmark::kMillisecond);
```

Figura 6. Benchmark padding

```cpp
void diff_var_local() {

  std::vector<std::atomic<int>> intVec(int_per_line);
  std::vector<std::thread> threadVec;
  thread_local std::atomic<int> a;

  for (int i = 0; i < int_per_line; i++) {
    intVec[i] = 0;
  }

  for (int i = 0; i < num_threads; i++) {

    threadVec.emplace_back([&]() {
      for(int j = 0; j < int_per_thread; j++){
        a = intVec[i+j].load();
        work(a);
      }
    });
  }

  // Join the threads
  for (std::thread& thread : threadVec) {
    thread.join();
  }

}

// A simple benchmark that runs our single-threaded implementation
static void noSharingLocal(benchmark::State& s) {
  while (s.KeepRunning()) {
    diff_var_local();
  }
}
BENCHMARK(noSharingLocal)->UseRealTime()->Unit(benchmark::kMillisecond);
```

Figura 7. Benchmark thread local

maximizar la competencia entre los threads y aumentar el efecto de False Sharing. Los demás benchmarks se adaptaron para que realizaran la misma cantidad de operaciones ”work()”que el benchmark falseSharing para realizar una comparativa justa. A continuación se explica cada benchmark:

III-A1. single_thread: Este benchmark realiza incrementos en una variable atómica en un solo hilo. No hay posibilidad de False Sharing aquí, ya que solo un hilo accede a la variable. Esta es una referencia para medir el rendimiento básico de una operación atómica en un solo hilo.

III-A2. directSharing: En este benchmark, varios hilos acceden directamente a una variable atómica compartida. Este enfoque puede llevar a Direct Sharing ya que múltiples hilos escriben en la misma variable, entonces cada vez que un hilo modifica esta variable, la variable entra en estado de modificado en los demás hilos.

III-A3. falseSharing: Similar al benchmark anterior, pero en lugar de acceder directamente a una variable compartida, cada hilo tiene su propia variable atómica en un vector, lo que puede llevar al efecto de False Sharing, ya que es muy probable que estas variables dentro del vector se guarden en una misma línea de caché.

III-A4. padding: En este benchmark, cada hilo tiene su propia variable atómica, pero se agregan rellenos para alinear las variables en líneas de caché separadas. Esto hace que cada variable esté en una líne de caché distinta, eliminando así el False Sharing en L1.

III-A5. noSharingLocal: Similar al benchmark de falseSharing, pero utiliza una variable local thread_local para que cada hilo trabaje con una variable única del hilo mitigando así el efecto de False Sharing.

Para todos los benchmarks se elige ejecutarlos utilizando Google Benchmark. El bucle ”while (s.KeepRunning())”se ejecutará repetidamente durante la fase de medición del benchmark. ”BENCHMARK(singleThread)-¿Unit(benchmark::kMillisecond)”declara el benchmark y establece la unidad de tiempo en milisegundos para la salida de los resultados. Al final del programa se llama ”BENCHMARK_MAIN();”para que Google Benchmark ejecute los benchmarks definidos.

Google Benchmark tiene por defecto la salida de el nombre del benchmark, el tiempo de promedio por iteración, el tiempo promedio de CPU por iteración y el número de iteraciones para obtener un resultado estable. Sin embargo, al integrarlo con ”libpfm”se puede acceder a los PMU.

III-B. Profiling

Para realizar el análisis de perfilado de utiliza Google Benchmark junto con la biblioteca libpfm y por aparte se utiliza perf.

Google Benchmark junto con libpfm permite realizar mediciones precisas de rendimiento ya que Google Benchmark proporciona una infraestructura robusta para medir el tiempo de ejecución de fragmentos de código de manera precisa y repetible. Además se pueden definir varios benchmarks que representen diferentes escenarios. Por ejemplo, tener benchmarks que utilizan variables compartidas directamente, benchmarks que utilizan variables separadas, benchmarks con relleno de alineación, entre otros, y ejecutarlos todos de manera sencilla.

Al integrarle libpfm se permite acceder a los contadores de rendimiento de hardware en arquitecturas de CPU específicas. Esto es valioso para medir aspectos más detallados del rendimiento, como los ciclos de CPU, los accesos a la caché y otros eventos de hardware relevantes para el análisis de False Sharing.

También se utiliza perf. La herramienta perf en Linux es una potente utilidad de perfilado que puede proporcionar información detallada sobre el uso de la CPU, la memoria y otros recursos del sistema durante la ejecución de tus benchmarks. Además, se utiliza perf c2c. Perf ç2c"se refiere a una herramienta y función de análisis de rendimiento en el conjunto de herramientas Perf del sistema operativo Linux donde ç2c"significa Çache-to-Cache". Perf c2c se utiliza principalmente para analizar transferencias de datos entre cachés en las CPUs modernas ya que muestra las líneas de cache que presentan la mayor contención y detalles de estas.

III-C. Automatización

Para la etapa de automatización se decide utilizar Python, lenguaje de alto nivel para tener como opción futura capturar de manera automática los resultados y graficarlos. Para la automatización lo que se hace es, haciendo uso de la biblioteca "subprocess"se ejecutan los benchmarks con las banderas necesarias para obtener la información deseada en los distintos casos planteados.

IV. Proceso de diseño

1. **Investigación:**
   - Se realizó una investigación exhaustiva sobre el concepto de false sharing y su impacto en sistemas multiprocesador.
   - Se estudiaron los fundamentos de las cachés y la coherencia de caché en sistemas MP.
   - Se revisaron publicaciones y recursos relevantes para obtener una comprensión sólida del problema.
2. **Diseño del Sistema:**
   - Se elaboró una propuesta de diseño que incluyó la arquitectura general del sistema.
   - Se identificaron los componentes clave, como los benchmarks, la etapa de perfilado y la etapa de automatización.
3. **Implementación de Benchmarks:**
   - Se desarrollaron benchmarks representativos en C++ para demostrar el false sharing.
   - Estos benchmarks incluyeron implementaciones tanto de hilos individuales como de múltiples hilos.
4. **Perfilados:**
   - Se utilizaron herramientas de perfilado como perf y Google Benchmark (con libpfm), para analizar el comportamiento del sistema y detectar el false sharing.
   - Se realizaron pruebas exhaustivas con los benchmarks implementados y se recopilaron datos de perfilado.
5. **Automatización:**
   - Se creó una etapa de automatización para ejecutar los benchmarks de manera automática.
6. **Graficado de Resultados de Perfilado:**
   - Los datos recopilados de perfilado se utilizaron para generar gráficos y visualizaciones que representaron claramente el impacto del False Sharing en el rendimiento.
   - Estas visualizaciones ayudaron en la presentación de resultados y presentación de los efectos de False Sharing.

V. Resultados

De resultados se obtuvieron las salidas de perf stat, perf c2c y Google Benchmark haciendo uso de libpfm para los performance counters. De estos datos se contruyen las gráficas en las figuras 8 y 9.

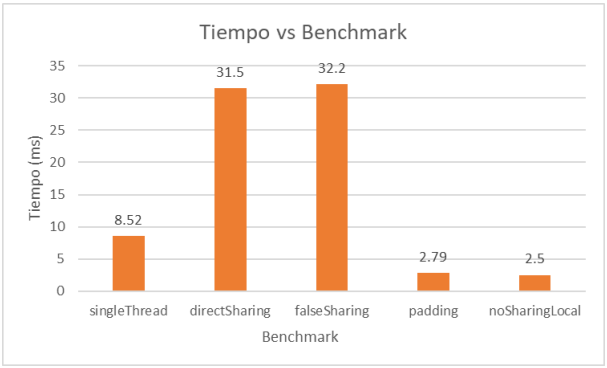

Figura 8. Tiempo por Benchmark utilizando Google Benchmark

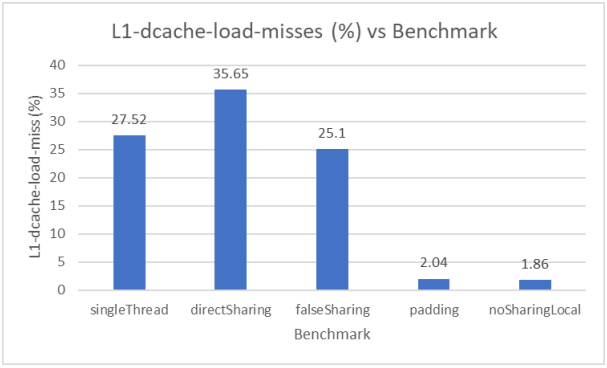

Figura 9. Load Misses en L1D por Benchmark utilizando perf stat

VI. Análisis de resultados

Tanto de la gráfica en la figura 8 como en 9 se muestra una notable diferencia cuando se presenta False Sharing y cuando no. El rendimiento es similar a cuando hay direct sharing y cuando hay false sharing, y el tiempo que toman estos benchmarks es mucho mayor a cuando no hay false sharing e incluso mayor a cuando es un programa single thread.

Es importante resaltar la gran mejora de arreglar los problemas de contención de memoria ya que los resultados que se obtienen de evitar este problema hace que se logre aprovechar el paralelismo del programa y mostrar mejora con respecto al programa utilizando un solo hilo.

VII. Conclusiones

El efecto de false sharing es dificil de identificar a un nivel superficial, por esto las herramientas de perfilado son tan valiosas, ya que hacen que sea notable que haya false sharing analizando lo que está pasando a nivel de hardware utilizando los performance counters.

El efecto de false sharing no solo hace que no se aproveche la paralelización a nivel de threads, si no que hace que el programa se vuelva mucho más ineficiente debido al mal uso de estos hilos.

Referencias

[1] Kandemir, Mahmut, et al. .On reducing false sharing while improving locality on shared memory multiprocessors."1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00425). IEEE, 1999.

[2] Raman, Easwaran, Robert Hundt, and Sandya Mannarswamy. "Structure layout optimization for multithreaded programs.International Symposium on Code Generation and Optimization (CGO'07). IEEE, 2007.

[3] Bolosky, William J., and Michael L. Scott. "False sharing and its e ect on shared memory performance."4th symposium on experimental distributed and multiprocessor systems. 1993.

[4] Freeh, Vincent W., and Gregory R. Andrews. "Dynamically controlling false sharing in distributed shared memory."Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing. IEEE, 1996.