Tema 1. Fundamentos de diseño de computadores: sistemas multinúcleo.

Arquitectura de Computadores

Dora Blanco Heras
Área de Arquitectura de Computadores
CITIUS
Universidad de Santiago de Compostela

Optimizaciones avanzadas de memoria caché

- Métricas a reducir:
  - Tiempo de búsqueda (*hit time*). Tasa de fallos (*miss rate*).
  - Penalización de fallos (*miss penalty*).

- Métricas a aumentar:
  - Ancho de banda de caché (*cache bandwidth*).

- Observación: Todas las optimizaciones avanzadas buscan mejorar alguna de estas métricas.

Optimizaciones avanzadas aplicada a memorias caché

Optimizaciones avanzadas

- Cachés pequeñas y simples
- Predicción de vía
- Acceso segmentado a la caché
- Cachés no bloqueantes
- Cachés multi-banco
- Palabra crítica primero y reinicio temprano
- Mezclas en búfer de escritura
- **Optimizaciones del compilador**
- **Lectura adelantada hardware**

El programador puede aplicar manualmente algunas de las optimizaciones que aplica el compilador.

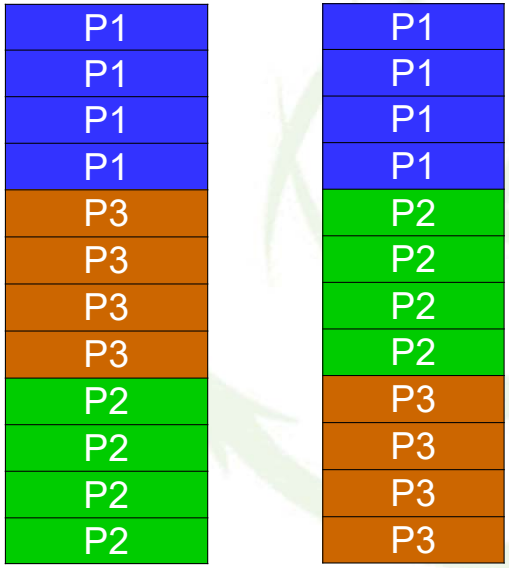
Optimizaciones del compilador

- Mejoran la tasa de fallos
- Con complejidad baja para el hardware.
- **El desafío está en que el software permita realizarlas.**

Referencia sobre optimizaciones avanzadas de memoria cache:
Computer Architecture. A Quantitative Approach
5th Ed.
Hennessy and Patterson.
Secciones: 2.1, 2.2.

Reordenación de procedimientos

- **Objetivo:** Reducir los fallos por conflicto debidos a que dos procedimientos coincidentes en el tiempo se corresponden con la misma línea de caché.
- **Técnica:** Reordenar los procedimientos en memoria.



Alineación de bloques básicos

- **Definición**: Un **bloque básico** es un conjunto de instrucciones que se ejecuta secuencialmente (no contiene saltos).

- **Objetivo**: Reducir la posibilidad de **fallos de caché** para código secuencial.

- **Técnica**: Hacer coincidir la **primera instrucción** de un bloque básico con la **primera palabra** de una línea de caché.

Linearización de saltos

- **Definición:** Un bloque básico es un conjunto de instrucciones que se ejecuta secuencialmente (no contiene saltos).

- **Técnica:** Si el compilador sabe que es probable que se tome un salto, puede cambiar el sentido de la condición e intercambiar los bloques básicos de las dos alternativas.
  - Algunos compiladores definen extensiones para dar pistas al compilador.
  - **Ejemplo**: **gcc** (**__likely__**).

Distribución optimizada de datos en las estructuras

Arrays paralelos

```
struct datos {
float coef, b, c;
char id[20];
float posicion[3];
double v;
};

struct datos ar[N];

for (i=0; i<M; i++)
r += ar[ d[i] ].coef * ar[ d[i] ].v;
```

Array fusionado

```
struct datos {
double v;
float coef, b, c;
char id[20];
float posicion[3];
};

struct datos ar[N];

for (i=0; i<M; i++)
r += ar[ d[i] ].coef * ar[ d[i] ].v;
```

- Cambiamos ubicación de los campos para poner adyacentes los que se usan conjuntamente

Intercambio de bucles

Acceso con saltos

```
for  ( int  j=0;  j<100; ++j)  {
for  ( int  i=0;  i<5000; ++i) {
v[ i ][ j ]  = k * v[ i ][ j ];
   }
}
```

Accesos secuenciales

```
for  ( int  i=0;  i<5000; ++i) {
for  ( int  j=0;  j<100; ++j)  {
 v[ i ][ j ]  = k * v[ i ][ j ];
   }
}
```

- **Objetivo:** Mejorar localidad espacial.
- Dependiente del modelo de almacenamiento vinculado al lenguaje de programación (por filas o por columnas):
  - FORTRAN versus C.

Fusión de bucles

Bucles separados

```
for ( int i=0; i<rows; ++i) {
for ( int j=0; j<cols; ++j) {
a[ i ][ j ] = b[ i ][ j ] * c[ i ][ j ];
  }
}
for ( int i=0; i<rows; ++i) {
for ( int j=0; j<cols; ++j) {
d[ i ][ j ] = a[ i ][ j ] + c[ i ][ j ];
}
}
```

Bucles fusionados

```
for ( int i=0; i<rows; ++i) {
for ( int j=0; j<cols; ++j) {
a[ i ][ j ] = b[ i ][ j ] * c[ i ][ j ];
d[ i ][ j ] = a[ i ][ j ] + c[ i ][ j ];
}
}
```

- **Objetivo**: Mejorar localidad temporal.
- **Cuidado**: Puede reducir localidad espacial.
- Puede ser aplicada por el programador de manera manual.

Acceso por bloques (*blocking* o *tiling*)

Producto original

```
for ( int i=0; i<N; ++i) {
  for ( int j=0; j<N; ++j) {
    r=0;
    for ( int k=0; k<N; ++k) {
      r+= b[ i ][ k] * c[k ][ j ];
    }
    a[i ][ j ] = r;
  }
}
```

Producto con acceso por bloques

```
for ( bj=0; bj<N; bj+=bsize) {
  for (bk=0; bk<N; bk +=bsize) {
    for ( i=0; i<N; ++i) {
      for ( j=bj; j<min(bj+bsize,N); ++j) {
        r=0;
        for (k=bk; k<min(bk+bsize,N); ++k){
        r +=b[i ][ k] * c[k ][ j ];
        }
        a[i ][ j ] += r;
      }
    }
  }
}
```

- **bsize:** tamaño de bloque
- En lugar de acceder filas o columnas enteras, las subdividimos en bloques y reusamos datos antes de que el bloque sea reemplazado de la cache.
- Requiere más acceso de memoria pero mejora la localidad en los accesos.
- Cuidado: hay aumento de tamaño de código y de cálculos-> no siempre se reducirá el tiempo de ejecución

Lectura adelantada (prebúsqueda o prefetching) de instrucciones

- **Observación**: Las instrucciones presentan alta localidad espacial.

- **Instruction prefetching:** Lectura adelantada de instrucciones.
  - Lectura de dos bloques en caso de fallo.
    - Bloque que provoca el fallo.
    - Bloque siguiente.

- **Ubicación:**
  - Bloque que provoca el fallo → **caché de instrucciones**.
    Bloque siguiente → **búfer de instrucciones**.

Lectura adelantada (prebúsqueda o prefetching) de datos por hardware

Ejemplo concreto del Pentium 4:

- **Data prefetching:** Permite lectura adelantada de páginas de 4KB a caché L2.

- Se invoca lectura adelantada si:
  - 2 fallos en L2 debidos a una misma página.
  - Distancia entre fallos menor que 256 bytes.

En los procesadores de Intel actuales se realizan precarga software y hardware.