Datenlokalitätsoptimierung für Architekturen mit tiefen Speicherhierarchien

Christian Weiß

Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik
Technische Universität München

Email: weissc@in.tum.de
WWW: http://wwwbode.informatik.tu-muenchen.de/~weissc/

Motivation

- *Gewünscht*: unbegrenzte Menge an schnellem Speicher
- *Prinzip der Lokalität*
- Ansatz: Speicherhierarchie
  - CPU nah: sehr schnelle aber kleine Speicher
    (z.B. 8 KByte on chip Cache, 1 Takt Latenzzeit)
  - CPU fern: große langsame Speicher
    (z.B. 1 GByte Hauptspeicher, $\approx$ 100 Takte Latenzzeit)
  - Speicherstufe beinhalten einander
- Prinzip gewinnt immer mehr an Bedeutung

Hitting the Memory Wall

- Durchschnittliche Zugriffszeit: $t_{avg} = p \times t_c + (1 - p) \times t_m$

p:	Cache Trefferrate
t_c:	Zugriffszeit zum Cache
t_m:	Zugriffszeit zum Hauptspeicher

- Annahmen:
  - $t_c = 1$
  - perfekter Cache
  - 20 - 40 % der Operationen Speicherzugriffe

$\Rightarrow$ Hitting the wall: $t_{avg} > 5$

$\Rightarrow$ Wenn Entwicklungstrend anhält: in den nächsten 5 Jahren zu erwarten

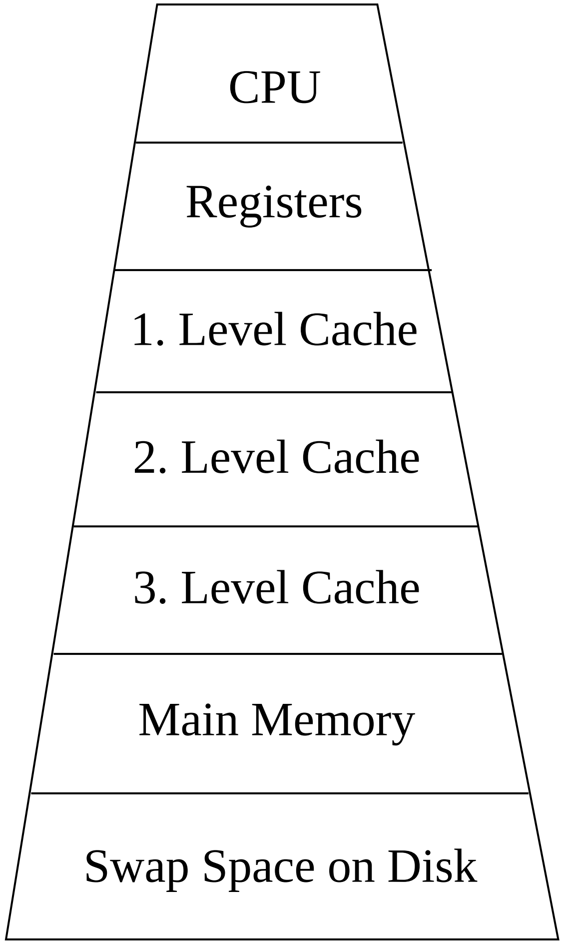
Strategien

- Latenzzeit bei *hit* und *miss* reduzieren
  - Second Level Cache
  - Speicherzugriffe überlappt ausführen (Pipelining)
  - Diverse andere Hardwaretechniken (s. "Computer Architecture: A ...")
- Cache-Trefferrate erhöhen
  - Hardwaretechniken
  - Datenzugriffs-Transformationen
  - Datenlayout-Transformationen
- Latenzzeit verdecken
  - Out-of-Order Execution
  - Multithreading

Cacheaufbau und Charakteristika

- Cache-Lines (Blockgröße als Parameter; typisch z.B. 32 Bytes)
- Caches deutlich kleiner als Hauptspeicher
  $\Rightarrow$ Mapping von HS-Blöcken notwendig
  - direct mapped $\Rightarrow$ jeder Block wird auf festen Bereich abgebildet
  - fully associative $\Rightarrow$ Block kann überall im Cache abgelegt werden
  - m-way set associative:
    * Block kann beliebig in einer Menge von m Cache-Lines abgelegt werden
    * Replacement: zufällig oder LRU
    * Verallgemeinert die beiden anderen Prinzipien
- Weiter Charakteristika:
  - Latenzzeit
  - Bandbreite
  - Cache-Größe

Speicherhierarchie der DEC PWS 500 au



Capacity	Bandwidth	Latency
256 Bytes	24000 MB/s	2 ns
8 KBytes	16000 MB/s	2 ns
96 KBytes	8000 MB/s	6 ns
2 MBytes	888 MB/s	24 ns
1536 MBytes	1000 MB/s	112 ns

Cache-Miss-Typen

- Compulsory miss:
  - Auch *cold start miss* genannt
  - Erster Zugriff auf Block führt zu einem Cache-Miss
- Capacity miss:
  - Cache nicht groß genug für alle benötigten Blöcke
  - Cache-Miss wird auftreten, wenn ein Block verdrängt wird, der später noch benötigt wird.
- Conflict miss:
  - Auch *collision miss* bzw. *interference miss* genannt
  - Später noch benötigter Block wird verdrängt, weil ein anderer Block auf dieselbe Cache-Line gemapped wird.
  - Ping-Pong Effekt

Compulsory Misses

Problem:

- Erster Zugriff nicht cachebar
- Folge: Zugriff potentiell sehr teuer

Optimierungen:

- Größere Cache-Lines
  - Erhöht Latenzzeit bei Cache-Miss
  - Kann Anzahl der *conflict misses* erhöhen
- Prefetching

Prefetching

Prinzip:

- Datenwort laden bevor es tatsächlich benötigt wird
- Überlappung von Berechnung und Ladeoperation
- Nur sinnvoll mit nichtblockierenden Caches

Varianten:

- Nächstes Datenwort laden (*unit stride access*)
- Wiederkehrende Zugriffsmuster erkennen und laden (*stream prefetch*)
- Pointerstrukturen laden

Implementierungsvarianten

Hardware:

- Meist nur *unit stride access*
- Stream Prefetch bei Cray T3E
- Bei Datenzugriffen meist nicht sehr präzise

Software:

- Grundlage: FETCH-Instruktion
- Vollautomatisch durch Compiler (z.B. SGI CC oder SUIF)
- Dank Compiler-Support potentiell sehr präzise
- Erlaubt auch Prefetching von Zeigerstrukturen
- Evtl. auch Interaktion durch Compileranweisungen (Pragmas)

Prefetching: Beispiel

```
for( i=0; i<n; i++){
    a += b[i];
}
```

- Cache-Miss führt zu Stall
- Cache-Miss zum Hauptspeicher kostet ≈ 60 Cycles

```
for( i=0; i<n; i++){
    prefetch b[i+16]
    a += b[i];
}
```

- Datum $16 \times 8 = 128$ [Bytes] im voraus holen
- Aber: 15 Prefetch-Instruktionen überflüssig
- Außerdem: Latenzzeit nur halb verdeckt.

```
for( i=0; i<n; i+=8){
    prefetch b[i+32]
    a += b[i];
    a += b[i+1];
    ...
}
```

- Loop Unrolling eliminiert überflüssige Prefetches
- Prefetch von mehr als einer Cache-Line im voraus

Prefetching: Bewertung

Pro:

- Reduziert teuere *cold misses*
- Kann Latenzzeiten mit Berechnung verdecken

Contra:

- Erzeugt evtl. unnötig Speicherzugriffe ($\rightarrow$ Bandbreite)
- Kann zusätzliche *conflict* und *capacity misses* erzeugen.

Resümee:

- Sinnvoll, wenn Prefetching präzise
- einfache SW-Implementierungen in vielen Compilern vorhanden

Capacity Misses

Problem:

- Daten passen nicht komplett in den Cache
- Daten werden verfrüht aus den Cache verdrängt

$\Rightarrow$ Ziel: Wiederverwendung der Daten maximieren

Lösung:

- Zugriffe auf gleiches Datum näher zusammen ziehen
- Schleifentransformationen
- *aber:*
  - Datenabhängigkeiten müssen erhalten bleiben
  - Strategien für Transformationen gesucht

Capacity Misses

Optimierungen:

- Loop Interchange:
  - Umordnung von verschachtelten Schleifen
  - Lineares Durchlaufen des Speichers
  - Maximiert Ausnützung einer Cache-Line
- Loop Fusion
  - Verschiedene Berechnungen auf selben Daten
  - Berechnungen verschiedener Schleifen zusammenziehen
  - Daten laden, mehrmalige Verwendung und anschließendes Verdrängen
- Loop Blocking

Loop Blocking

Problem:

- Mehrere Arrays werden gleichzeitig verwendet
- Zugriff auf einige *zeilenweise* auf die anderen *spaltenweise*

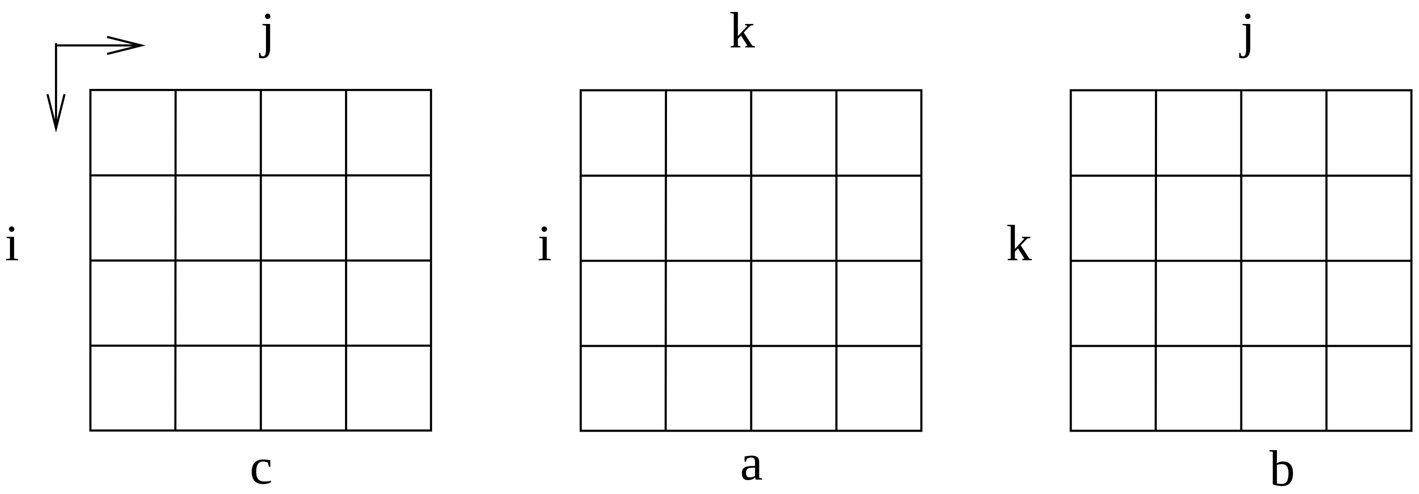
$\Rightarrow$ Loop Interchange zwecklos

Prinzip:

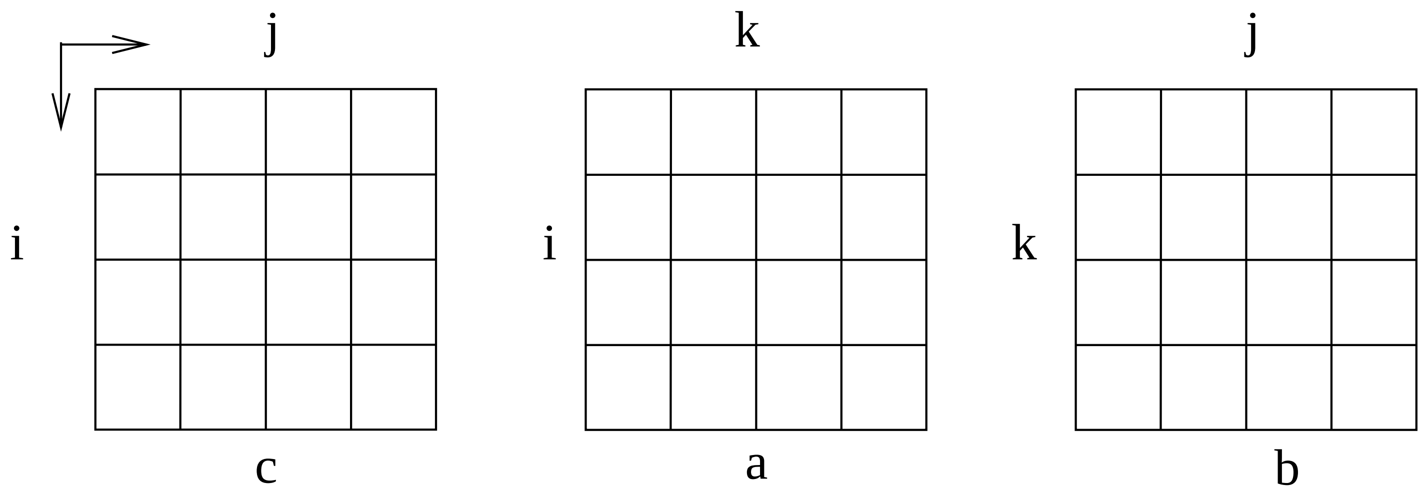
- Zerlegung der Arrays in Blöcke
- Maximiere Zugriffe auf diese Blöcke bevor sie verdrängt werden

Beispiel: Matrixmutiplikation

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {

       c[i][j] += a[i][k]*b[k][j];

}}}
```



```
for (jj=0; jj<N; jj+=B) {
  for (kk=0; kk<N; kk+=B) {

    for (i=0; i<N; j++) {
      for (j=jj; j<min(jj+B,N); j++) {
        for (k=kk; i<min(kk+B,N); k++) {
          c[i][j] += a[i][k]*b[k][j];

}}}}}
```



Capacity Misses

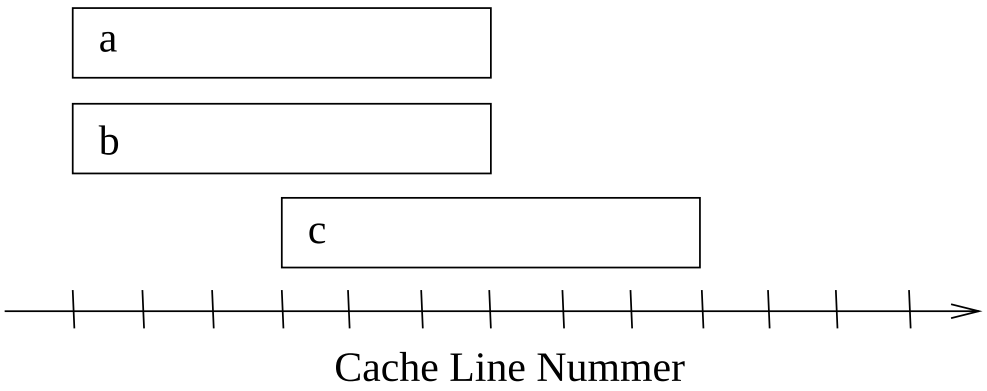
Resümee:

- Äußerst komplexe Transformationen nötig
- Compiler-Support erwünscht
- Hohes Potential für Effizienzgewinn
- Keine Veränderung der Hardware nötig: Nur recompilieren
- Implementierungen:
  - Native SGI Compiler
  - SUIF (Stanford)
  - Memoria (Rice; MTU)

Conflict Misses

Problem:

- 2 Datenstrukturen mit ungünstigem Mapping
- Wechselseitige Zugriffe auf die Datenstrukturen
- Ping-Pong Effekt
- wird verschlimmert durch große Cache-Lines



Conflict Misses

Hardware:

- Assoziativität erhöhen
- Victim Buffer
- Pseudo Assoziative Caches

Datenlayout-Transformationen:

- Array Padding
- Array Merging
- Block Copy

Array Padding

Beispiel:

double a[1024];
double b[1024];

for (i=0; i<1024; i++) {
  sum += a[i]*b[i];
}

DEC A21164
L1 Cache-Mapping:

addr(a) = 0
addr(a) → 0 **mod** 8192 = 0

addr(b) = 8192
addr(b) → 8192 **mod** 8192 = 0

Padding:

// *X < 1024*
double a[1024+X];
double b[1024];

addr(a) = 0
addr(a) → 0 **mod** 8192 = 0

addr(b) = 8192+8*X
addr(b) → 8192+8*X **mod** 8192 = 8*X

Array Padding

Wahl der Padding-Größe:

- Mapping in verschiedene Cache-Lines
- Arrays in völlig verschiedenen Bereichen mappen
- Falls nicht möglich: abhängig vom Algorithmus

$\Rightarrow$ Von Maschine und Algorithmus abhängig

Array Merging

Prinzip:

- Selbe Problemstellung wie beim Array Padding
- Verschmelzen zweier Arrays zu einem Array einer Struktur
- Prinzipiell einfache Umstrukturierung

Vorher:

```
double a[1024];
double b[1024];

a[42]=1.0;
```

Nachher:

```
struct merge {
    double a;
    double b;
};
struct merge m[1024];

m[42].a=1.0;
```

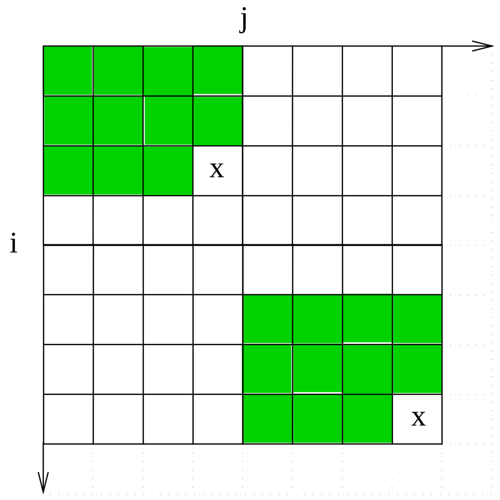
Array Padding und Array Merging

Resümee:

- Kann erhebliche Leistungssteigerung zur Folge haben
- Pointerarithmetik problematisch
- Ansonsten: einfach und gefahrlos durchzuführen
- Wird nicht in kommerziellen Compilern angewendet

Block Copy

am Beispiel MatMult



Ursachen für Conflict Misses:

- Cross Interference
- Self Interference

Prinzip:

- Blöcke in neuen Speicherbereich kopieren
- Immer möglich, wenn Loop Blocking möglich
- Keine *self interference* mehr
- Kopieraufwand steht Reduzierung der Conflict Misses gegenüber

Zusammenfassung

Inhalt:

- Motivation und grundsätzliche Optimierungsstrategien
- Cache-Miss-Arten (Compulsory, Capacity und Conflict)
- Hard- und Software-Optimierungen vorgestellt und bewertet

Schlußbemerkung:

- Relativ großes Optimierungspotential
- Optimierungen sind (bedingt) maschinenabhängig
- Optimierungen meist sehr kompliziert
- Compiler-Support noch immer dürftig

Trend

Leistungsunterschied zwischen CPU und RAM wird weiter wachsen

1. Speicherhierarchien mit mehr als 10 Stufen
   - Stärkere Unterstützung durch Compiler und Tools gefordert
   - Stärkere Einbindung von Programmiererwissen
2. Intelligente RAMs (IRAM)
   - Univ. of California at Berkeley
   - RAMs mit CPU on chip
   - Kommunikation zwischen RAM-Chips wie bei Parallelrechnern
3. Programmierbare Caches
   - Z.B. MORPH Projekt (Univ. of Illinois)
   - Volle Kontrolle über Caches für Compiler (und Programmierer)
   - Realisiert durch rekonfigurierbare Architekturen

Noch Fragen?

Literatur

[1] Ed Anderson, Jeff Brooks, and Tom Hewitt. The Benchmarker's Guide to Single-processor Optimization for CRAY T3E Systems. Technical report, Benchmarking Group, Gray Research, June 1997.

[2] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM computing surveys*, 26(4):345ff, December 1994.

[3] David Bernstein, Doron Cohen, and Ari Freund. Compile Techniques for Data Prefetching on the PowerPC. In *Parallel Architectures and Compilation Techniques*, pages 19–26, 1995.

[4] D. C. Burger, J. R. Goodman, and A. Kägi. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Technical Report TR-95-1261, University of Wisconsin, Dept. of Computer Science, Madison, 1995.

[5] S. Carr, K.S. McKinley, and C. Tseng. Compiler Optimizations for Improving Data Locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, San Jose, CA, October 1994.

[6] J. Ferrante, V. Sarkar, and W. Trash. On Estimating and Enhancing Cache Effectiveness. In U. Banerjee, editor, *Languages and Compilers for Parallel Computing, 4th Int'l Workshop*. Springer, August 1991.

[7] Jeff Fier. *Performance Tuning Optimization for Origin2000 and Onyx2*. Silicon Graphics, Technical Publications, 1996. Document Number 007-3430-001.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publisher, Inc., second edition, 1996.

[9] Norman Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. Technical Report TN-14, Western Research Laboratory, March 1990.

[10] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, April 1991.

[11] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, December 1995.

[12] K. S. McKinley and O. Temam. A Quantitative Analysis of Loop Nest Locality. In *Proceedings ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachussets, October 1996.

[13] Subbarao Palacharla and R.E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[14] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent dram: Iram. *IEEE Micro*, April 1997.

[15] O. Temam, E. Granston, and W. Jalby. To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Proceedings Supercomputing'93*, Portland, November 1993.

[16] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.

[17] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implication of the Obvious. *ACM Computer Architecture News*, 23(1):20–24, March 1995.

[18] X. Zhang, A. Dasdan, M. Schulz, R. Gupta, and A. Chien. Architectural adaptation for application–specific locality optimizations. In *Proceedings of the International Conference on Computer Design ICCD*. IEEE, October 1997.

[19] Z.Zhang and J.Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *Proceedings of 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 188–199, June 1995.