Rechnerarchitekturen Protokoll
Teil 2

Herbert Valerio Riedel
e9725348@student.tuwien.ac.at
E881

26. Mai 2000

Inhaltsverzeichnis

1 Übung 3: Cache Simulation **1**

1.1 Aufgabe 1 1

1.2 Aufgabe 2 4

1.3 Aufgabe 3 7

1.3.1 Aufbereitung der Daten 7

1.3.2 Unified Cache Statistiken 8

1.3.3 Non-Unified Cache Statistiken 12

1.4 Aufgabe 4 25

Abbildungsverzeichnis

1 unified 256 byte cache 8
2 unified 512 byte cache 9
3 unified 1024 byte cache 10
4 unified 2048 byte cache 11
5 non-unified 512 byte cache (256 byte data/256 byte instr) 13
6 non-unified 768 byte cache (256 byte data/512 byte instr) 14
7 non-unified 1280 byte cache (256 byte data/1024 byte instr) . . . 15
8 non-unified 1792 byte cache (256 byte data/1536 byte instr) . . . 15
9 non-unified 2048 byte cache (256 byte data/1792 byte instr) . . . 16
10 non-unified 768 byte cache (512 byte data/256 byte instr) 17
11 non-unified 1024 byte cache (512 byte data/512 byte instr) . . . 18
12 non-unified 1536 byte cache (512 byte data/1024 byte instr) . . . 18
13 non-unified 2048 byte cache (512 byte data/1536 byte instr) . . . 19
14 non-unified 1280 byte cache (1024 byte data/256 byte instr) . . . 20
15 non-unified 1536 byte cache (1024 byte data/512 byte instr) . . . 21
16 non-unified 2048 byte cache (1024 byte data/1024 byte instr) . . 22
17 non-unified 1792 byte cache (1536 byte data/256 byte instr) . . . 22
18 non-unified 2048 byte cache (1536 byte data/512 byte instr) . . . 23
19 non-unified 2048 byte cache (1792 byte data/256 byte instr) . . . 24

Tabellenverzeichnis

1 unified 256 byte cache 8
2 unified 512 byte cache 9
3 unified 1024 byte cache 9
4 unified 2048 byte cache 10
5 non-unified 512 byte cache (256 byte data/256 byte instr) 13
6 non-unified 768 byte cache (256 byte data/512 byte instr) 13
7 non-unified 1280 byte cache (256 byte data/1024 byte instr) . . . 14
8 non-unified 1792 byte cache (256 byte data/1536 byte instr) . . . 14
9 non-unified 2048 byte cache (256 byte data/1792 byte instr) . . . 16
10 non-unified 768 byte cache (512 byte data/256 byte instr) 16
11 non-unified 1024 byte cache (512 byte data/512 byte instr) . . . 17
12 non-unified 1536 byte cache (512 byte data/1024 byte instr) . . . 17
13 non-unified 2048 byte cache (512 byte data/1536 byte instr) . . . 19
14 non-unified 1280 byte cache (1024 byte data/256 byte instr) . . . 20
15 non-unified 1536 byte cache (1024 byte data/512 byte instr) . . . 20
16 non-unified 2048 byte cache (1024 byte data/1024 byte instr) . . 21
17 non-unified 1792 byte cache (1536 byte data/256 byte instr) . . . 21
18 non-unified 2048 byte cache (1536 byte data/512 byte instr) . . . 23
19 non-unified 2048 byte cache (1792 byte data/256 byte instr) . . . 23

1 Übung 3: Cache Simulation

1.1 Aufgabe 1

Gegeben war der folgende MIPS-Code:

```
compute:
        move    $9, $0
        la      $8, a
        la      $7, b
        la      $6, c
        la      $5, d
        la      $4, e
$L11:
        lw      $3, 0($6)
        addiu   $6, $6, (4*N)
        lw      $2, 0($7)
        addiu   $7, $7, 4
        addu    $2, $2, $3
        sw      $2, 0($8)
        addiu   $9, $9, 1
        lw      $3, 0($4)
        addiu   $4, $4, 4
        lw      $2, 0($5)
        addiu   $5, $5, 4
        addu    $2, $2, $3
        sw      $2, 0($6)
        addiu   $8, $8, 4
        slt     $2, $9, 1000
        addiu   $6, $6, (-4*(N-1))
        bne     $2, 0, $L11
        j       $31
        .end    compute
```

Mittels eines C-Programms[1] wurde ein Tracefile erstellt, aus dem ein Auszug[2] folgt:

```
2 00001000  move $9,$0
2 00001004  la   $8,a
2 00001008  la   $7,b
2 0000100c  la   $6,c
2 00001010  la   $5,d
2 00001014  la   $4,e
2 00001018  lw   $3,0($6)
0 00004000
2 0000101c  addu $6,$6,(4*N)
2 00001020  lw   $2,0($7)
0 00003000
2 00001024  addu $7,$7,4
2 00001028  addu $2,$2,$3
```

[1] auf das listing des Programms wurde bewußt verzichtet, weil es in meinen Augen irrelevant und darüberhinaus eher unschön anzusehen ist…

[2] 23007 Zeilen abzudrucken wird sicher nicht im Sinne der Angabe gewesen sein

```
2 0000102c  sw    $2,0($8)
1 00002000
2 00001030  addu  $9,$9,1
2 00001034  lw    $3,0($4)
0 00006000
2 00001038  addu  $4,$4,4
2 0000103c  lw    $2,0($5)
0 00005000
2 00001040  addu  $5,$5,4
2 00001044  addu  $2,$2,$3
2 00001048  sw    $2,0($6)
1 00004020
2 0000104c  addu  $8,$8,4
2 00001050  slt   $2,$9,1000
2 00001054  addu  $6,$6,(-4*(N-1))
2 00001058  bne   $2,$0,$L11
2 00001018  lw    $3,0($6)
0 00004004
2 0000101c  addu  $6,$6,(4*N)
2 00001020  lw    $2,0($7)
0 00003004
2 00001024  addu  $7,$7,4
2 00001028  addu  $2,$2,$3
2 0000102c  sw    $2,0($8)
1 00002004
...
2 00001030  addu  $9,$9,1
2 00001034  lw    $3,0($4)
0 00006f98
2 00001038  addu  $4,$4,4
2 0000103c  lw    $2,0($5)
0 00005f98
2 00001040  addu  $5,$5,4
2 00001044  addu  $2,$2,$3
2 00001048  sw    $2,0($6)
1 00004fb8
2 0000104c  addu  $8,$8,4
2 00001050  slt   $2,$9,1000
2 00001054  addu  $6,$6,(-4*(N-1))
2 00001058  bne   $2,$0,$L11
2 00001018  lw    $3,0($6)
0 00004f9c
2 0000101c  addu  $6,$6,(4*N)
2 00001020  lw    $2,0($7)
0 00003f9c
2 00001024  addu  $7,$7,4
2 00001028  addu  $2,$2,$3
2 0000102c  sw    $2,0($8)
1 00002f9c
2 00001030  addu  $9,$9,1
2 00001034  lw    $3,0($4)
0 00006f9c
2 00001038  addu  $4,$4,4
2 0000103c  lw    $2,0($5)
```

```
0 00005f9c
2 00001040  addu $5,$5,4
2 00001044  addu $2,$2,$3
2 00001048  sw   $2,0($6)
1 00004fbc
2 0000104c  addu $8,$8,4
2 00001050  slt  $2,$9,1000
2 00001054  addu $6,$6,(-4*(N-1))
2 00001058  bne  $2,$0,$L11
2 0000105c  j    $31
```

1.2 Aufgabe 2

Reverse Engineering ergab:

```
/*
 * Rechnerarchitekturen Uebung 3
 */

/* #include <assert.h> */

#define N 8

int a[1024], b[1024], c[1024], d[1024], e[1024];

void compute(void)
{
  register int *pe=e, *pd=d, *pc=c, *pb=b, *pa=a,
               i=0, r3, r2;
  /* assert(sizeof(int) == 4); */

  do {
    r3 = *pc;
    pc += N;
    r2 = *(pb++);
    r2 += r3;
    *pa = r2;
    i++;
    r3 = *(pe++);
    r2 = *(pd++);
    r2 += r3;
    *pc = r2;
    pa++;
    r2 = i < 1000;
    pc -= N-1;
  } while(r2);

  return;
}
```

Recompilen auf einer IRIX Maschine mittels des MIPS C Compilers[3] ergab folgendes Ergebnis:

```
        .globl  a
        .lcomm  a 4096
        .globl  b
        .lcomm  b 4096
        .globl  c
        .lcomm  c 4096
        .globl  d
        .lcomm  d 4096
```

[3]Zum Compilen wurde `cc -mips1 -32 -S source.c` verwendet, und der Assembler Output manuell geringfügig nachbearbeitet

```
9           .globl  e
10          .lcomm  e 4096
11          .text
12          .align  2
13          .globl  compute
14   #  12  {
15          .ent    compute 2
16  compute:
17          .option O1
18          .set     noreorder
19          .cpload $25
20          .set     reorder
21          subu    $sp, 32
22          .frame  $sp, 32, $31
23   #  13     register int *pe=e, *pd=d, *pc=c, *pb=b, *pa=a,
24   #  14                  i=0, r3, r2;
25          la      $4, e
26          la      $5, d
27          la      $6, c
28          la      $7, b
29          la      $8, a
30          move    $9, $0
31   #  15     /* assert(sizeof(int) == 4); */
32   #  16
33   #  17     do {
34  $32:
35   #  18       r3 = *pc;
36          lw      $3, 0($6)
37   #  19       pc += N;
38          addu    $6, $6, 32
39   #  20       r2 = *(pb++);
40          lw      $2, 0($7)
41          addu    $7, $7, 4
42   #  21       r2 += r3;
43          addu    $2, $2, $3
44   #  22       *pa = r2;
45          sw      $2, 0($8)
46   #  23       i++;
47          addu    $9, $9, 1
48   #  24       r3 = *(pe++);
49          lw      $3, 0($4)
50          addu    $4, $4, 4
51   #  25       r2 = *(pd++);
52          lw      $2, 0($5)
53          addu    $5, $5, 4
54   #  26       r2 += r3;
55          addu    $2, $2, $3
56   #  27       *pc = r2;
57          sw      $2, 0($6)
58   #  28       pa++;
59          addu    $8, $8, 4
60   #  29       r2 = i < 1000;
61          slt     $2, $9, 1000
62   #  30       pc -= N-1;
```

```
63          addu    $6, $6, -28
64          bne     $2, 0, $32
65  #  31     } while(r2);
66  #  32
67  #  33     return;
68          .livereg        0x0000FF0E,0x00000FFF
69          addu    $sp, 32
70          j       $31
71          .end    compute
```

Eine funktionell äquivalente, jedoch elegantere Variante stellt das folgende Listing dar[4]:

```
1  /*
2   * Rechnerarchitekturen Uebung 3
3   */
4
5  #define N 8
6
7  int a[1024], b[1024], c[1024], d[1024], e[1024];
8
9  void compute(void)
10 {
11   int i;
12
13   for(i=0; i<1000; i++) {
14     a[i] = c[i] + b[i];
15     c[i+N] = e[i] + d[i];
16   }
17
18   return;
19 }
```

[4]Diese Variante ließ sich jedoch nicht auf den vorgegebenen Assemblercode zurückführen.

1.3 Aufgabe 3

1.3.1 Aufbereitung der Daten

Allen Cacheconfigurationen lag laut *DineroIII/Dos* folgende gemeinsamme Zugriffstatistik zugrunde:

```
Metrics            Access Type:
(totals,fraction)  Total   Instrn   Data    Read    Write    Misc
-----------------  ------  ------  ------  ------  ------  ------
Demand Fetches      23007   17007    6000    4000    2000       0
                   1.0000  0.7392  0.2608  0.1739  0.0869  0.0000
```

Die Einstellungen waren:

Prefetching	none
Blocksize	16 byte
Buswidth	1 word

Die Einstellungen von *XCache* zur Berechnung der gesammten Länge der Speichzugriffe ergaben, daß die Länge proportional zum *Total Traffic* ist; daher wurde in den Diagrammen auch diese statt jener verwendet.

	LRU	FIFO	RAND
1-way	30684	30684	30684
2-way	27524	27772	28116
4-way	29536	30032	24100
8-way	21032	15024	19172
16-way	7032	10508	13580

Tabelle 1: unified 256 byte cache

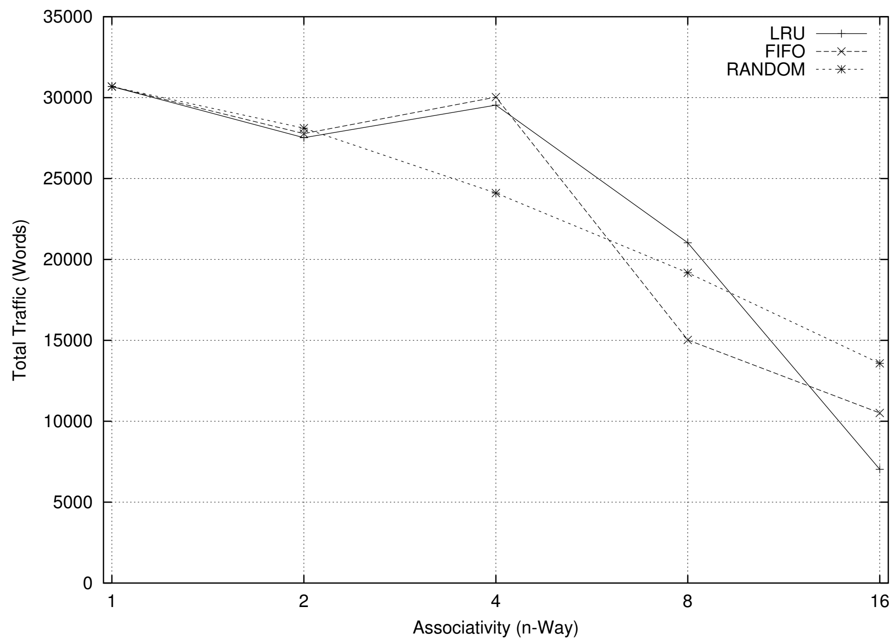

Abbildung 1: unified 256 byte cache

1.3.2 Unified Cache Statistiken

Folgende Cachegrößen wurden für einen nicht unterteilten Cache simuliert:

- 256 byte
- 512 byte
- 1024 byte
- 2048 byte

	LRU	FIFO	RAND
1-way	27836	27836	27836
2-way	26312	26440	25992
4-way	27276	27524	19676
8-way	7032	7676	11880
16-way	7032	7852	9936

Tabelle 2: unified 512 byte cache

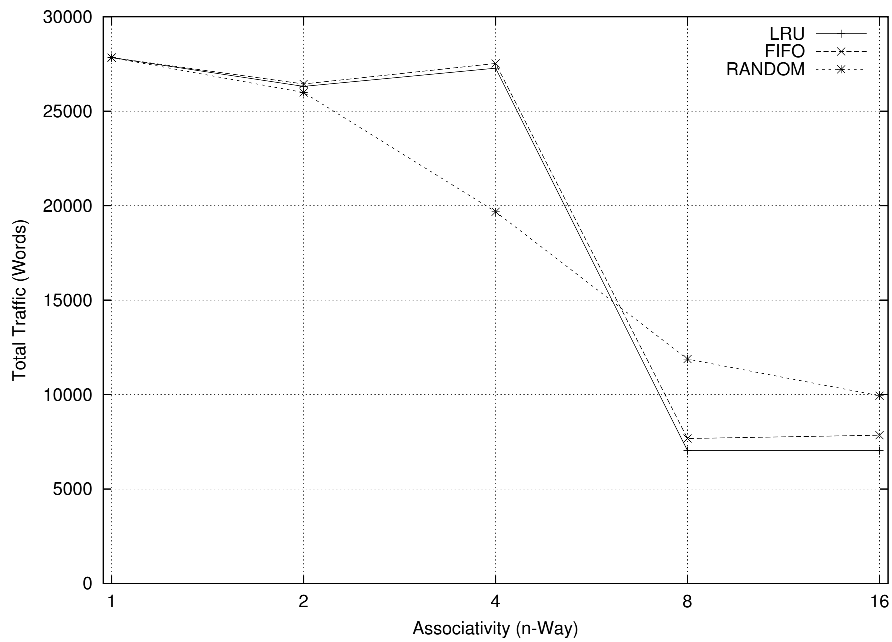

Abbildung 2: unified 512 byte cache

	LRU	FIFO	RAND
1-way	26412	26412	26412
2-way	25672	25736	24916
4-way	26184	26312	18180
8-way	7032	7348	10424
16-way	7032	7416	8428

Tabelle 3: unified 1024 byte cache

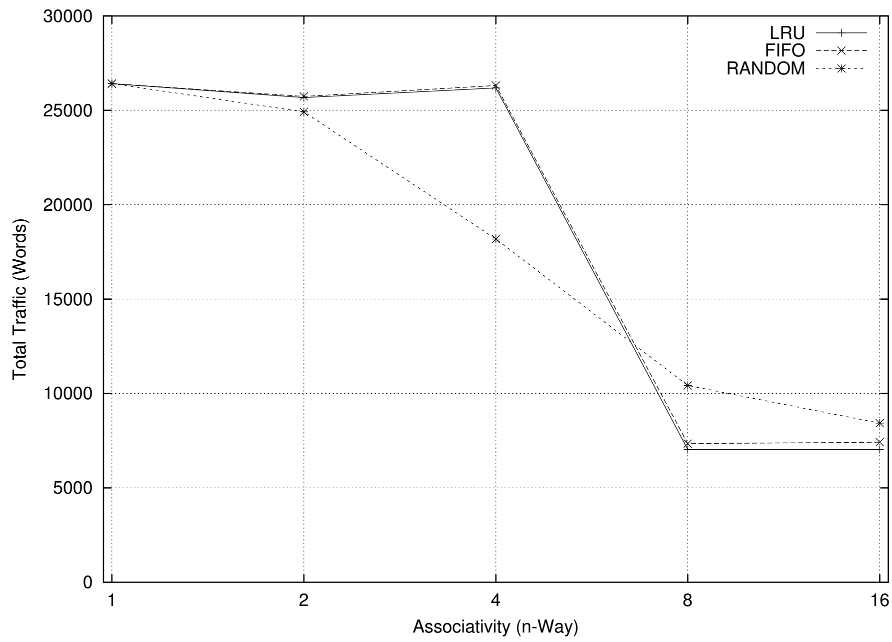

Abbildung 3: unified 1024 byte cache

	LRU	FIFO	RAND
1-way	25700	25700	25700
2-way	25352	25384	24352
4-way	25608	25672	16772
8-way	7032	7196	9820
16-way	7032	7204	8064

Tabelle 4: unified 2048 byte cache

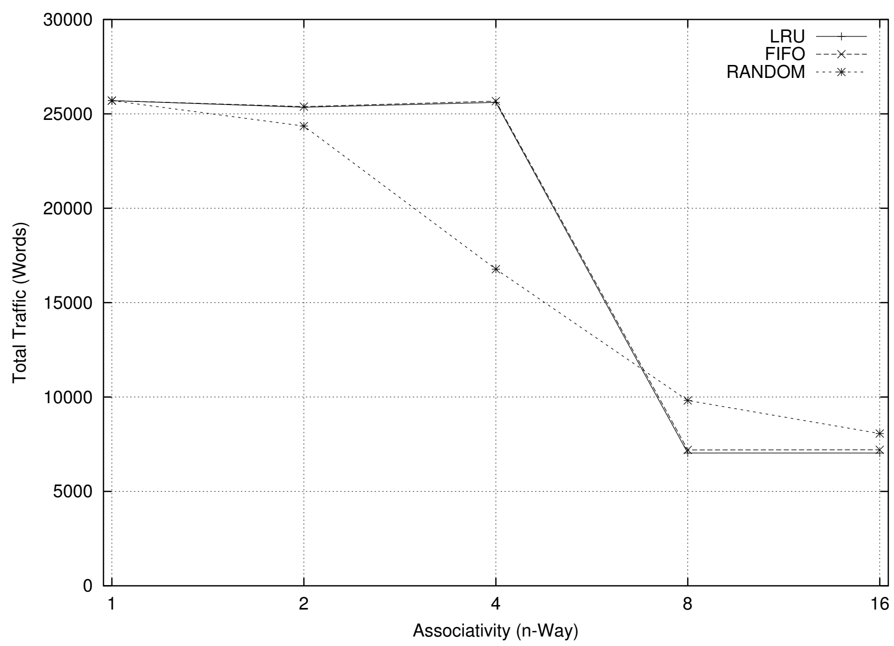

Abbildung 4: unified 2048 byte cache

1.3.3 Non-Unified Cache Statistiken

Folgende Cachegrößen wurden simuliert:

- 512 byte
- 768 byte
- 1024 byte
- 1536 byte
- 1792 byte
- 2048 byte

Für folgende (jeweils in bytes angegebene) Unterteilungen des non-unified Caches:

data	instr.	total
256	256	512
256	512	768
512	256	768
512	512	1024
256	1024	1280
1024	256	1280
512	1024	1536
1024	512	1536
256	1536	1792
1536	256	1792
256	1792	2048
512	1536	2048
1024	1024	2048
1536	512	2048
1792	256	2048

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23952
4-way	25032	25032	16308
8-way	7032	7032	12056
16-way	7032	7032	8928

Tabelle 5: non-unified 512 byte cache (256 byte data/256 byte instr)

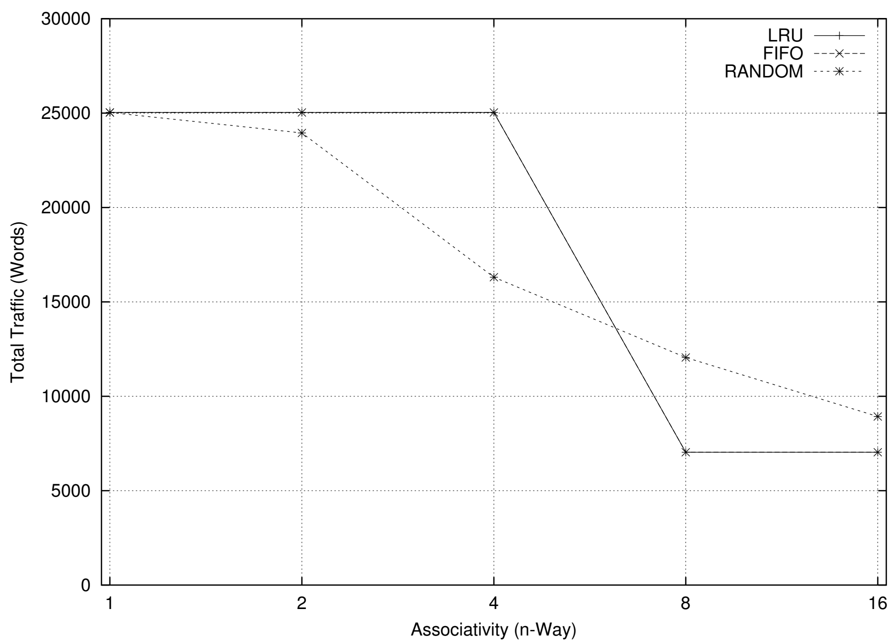

Abbildung 5: non-unified 512 byte cache (256 byte data/256 byte instr)

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23952
4-way	25032	25032	16320
8-way	7032	7032	11636
16-way	7032	7032	8748

Tabelle 6: non-unified 768 byte cache (256 byte data/512 byte instr)

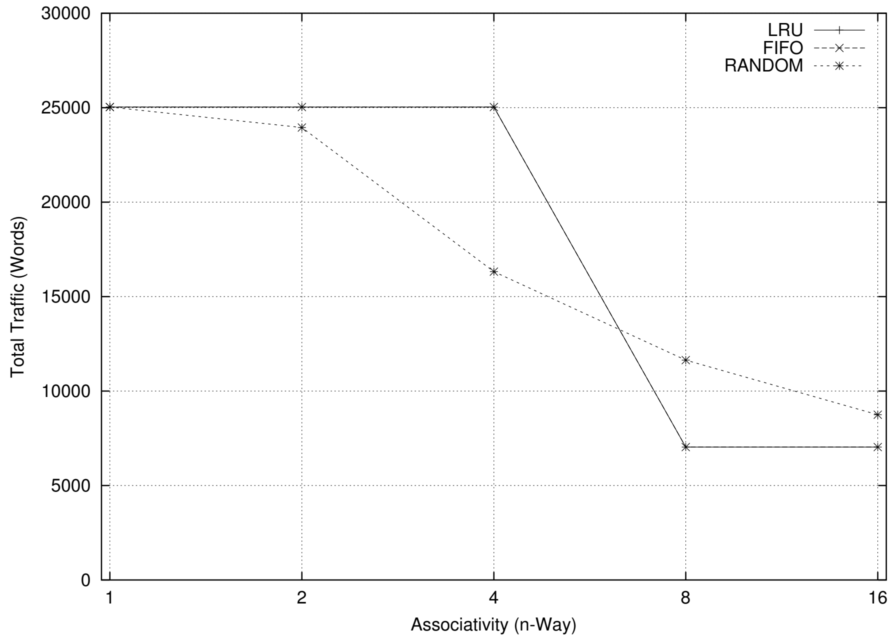

Abbildung 6: non-unified 768 byte cache (256 byte data/512 byte instr)

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23952
4-way	25032	25032	16320
8-way	7032	7032	11636
16-way	7032	7032	8992

Tabelle 7: non-unified 1280 byte cache (256 byte data/1024 byte instr)

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23952
4-way	25032	25032	16320
8-way	7032	7032	11636
16-way	7032	7032	8944

Tabelle 8: non-unified 1792 byte cache (256 byte data/1536 byte instr)

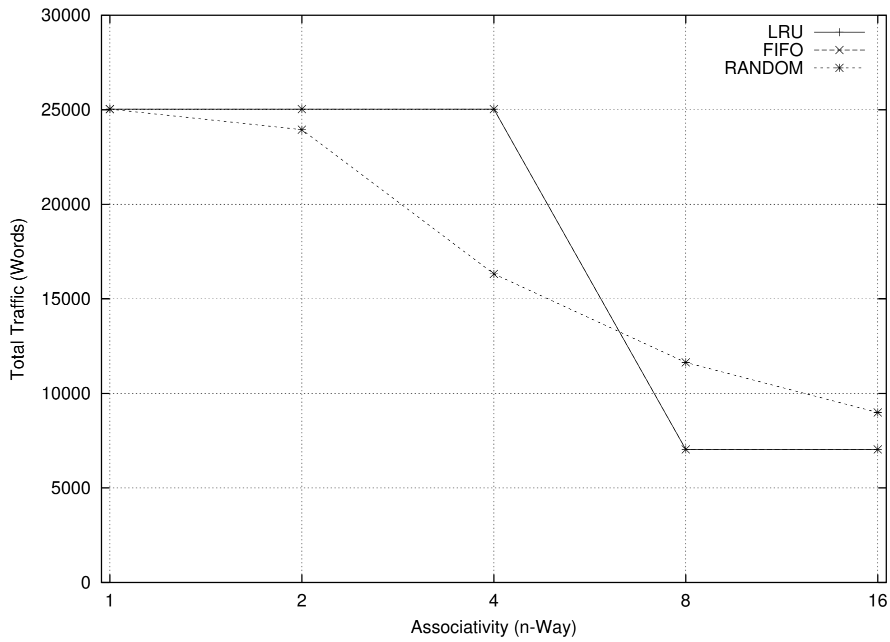

Abbildung 7: non-unified 1280 byte cache (256 byte data/1024 byte instr)

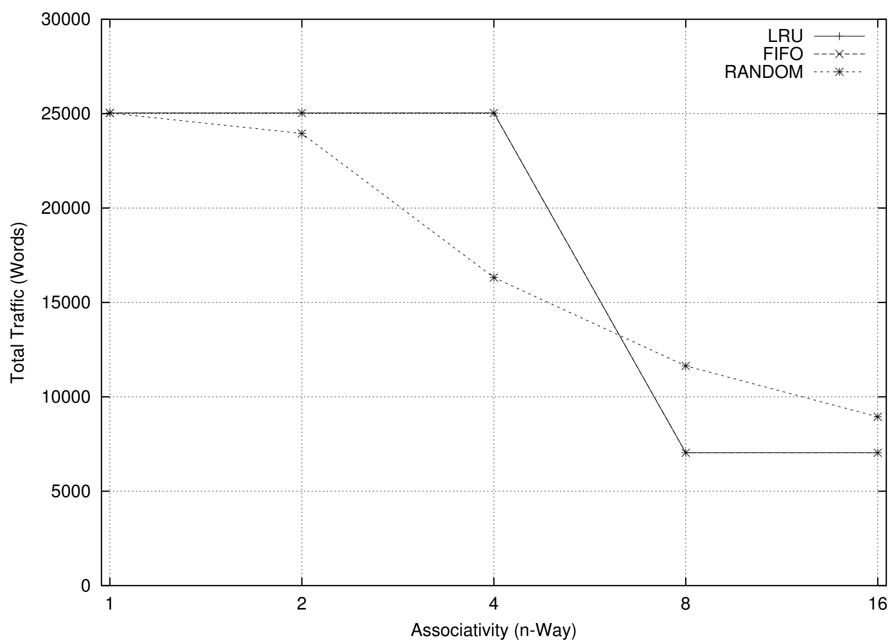

Abbildung 8: non-unified 1792 byte cache (256 byte data/1536 byte instr)

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23952
4-way	25032	25032	16320
8-way	7032	7032	11636
16-way	7032	7032	8944

Tabelle 9: non-unified 2048 byte cache (256 byte data/1792 byte instr)

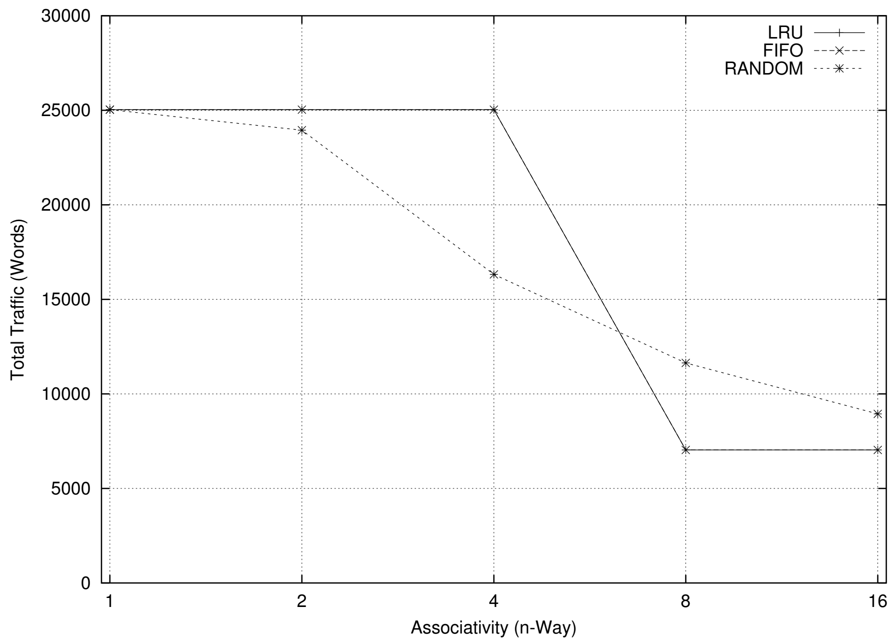

Abbildung 9: non-unified 2048 byte cache (256 byte data/1792 byte instr)

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23976
4-way	25032	25032	16244
8-way	7032	7032	9340
16-way	7032	7032	8524

Tabelle 10: non-unified 768 byte cache (512 byte data/256 byte instr)

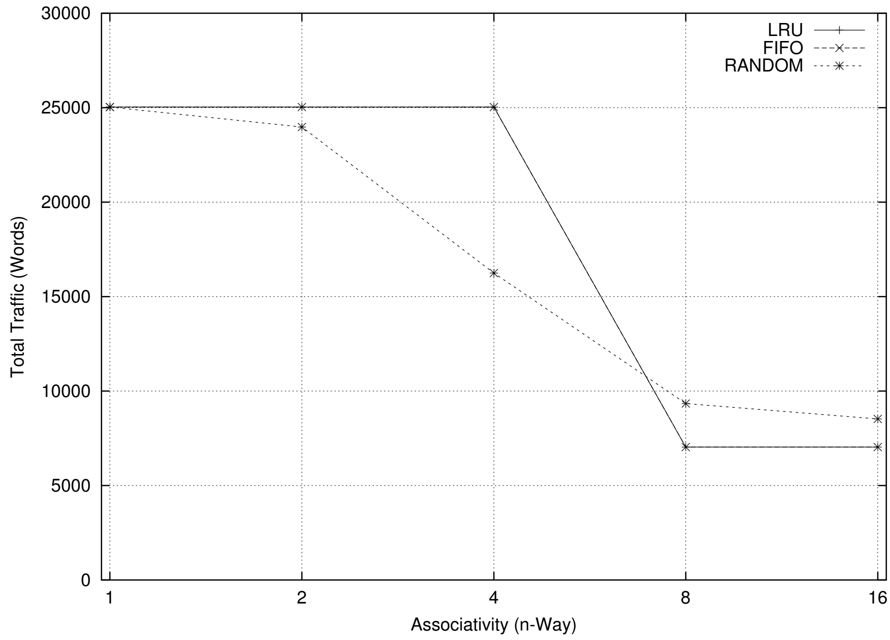

Abbildung 10: non-unified 768 byte cache (512 byte data/256 byte instr)

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23976
4-way	25032	25032	16232
8-way	7032	7032	9200
16-way	7032	7032	8604

Tabelle 11: non-unified 1024 byte cache (512 byte data/512 byte instr)

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23976
4-way	25032	25032	16232
8-way	7032	7032	9308
16-way	7032	7032	8600

Tabelle 12: non-unified 1536 byte cache (512 byte data/1024 byte instr)

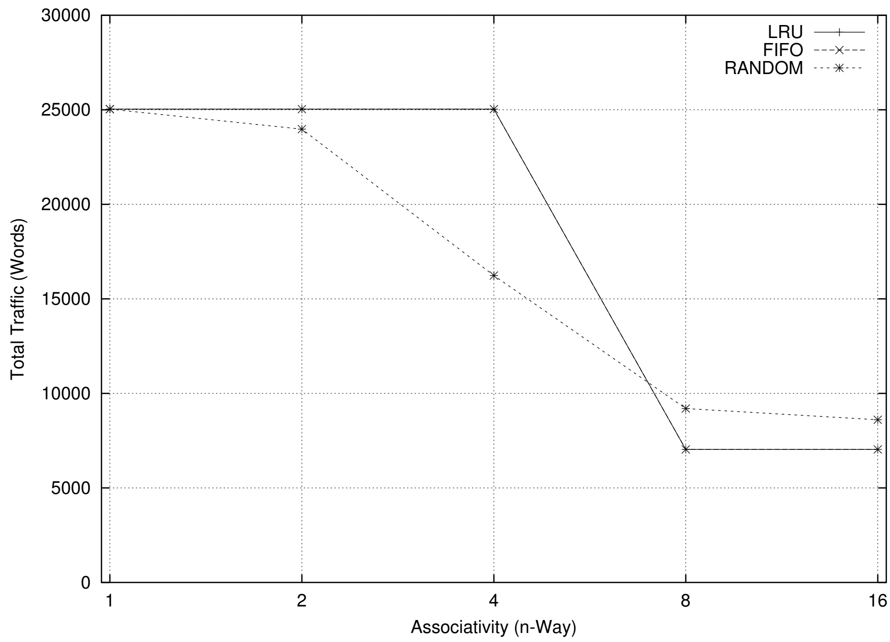

Abbildung 11: non-unified 1024 byte cache (512 byte data/512 byte instr)

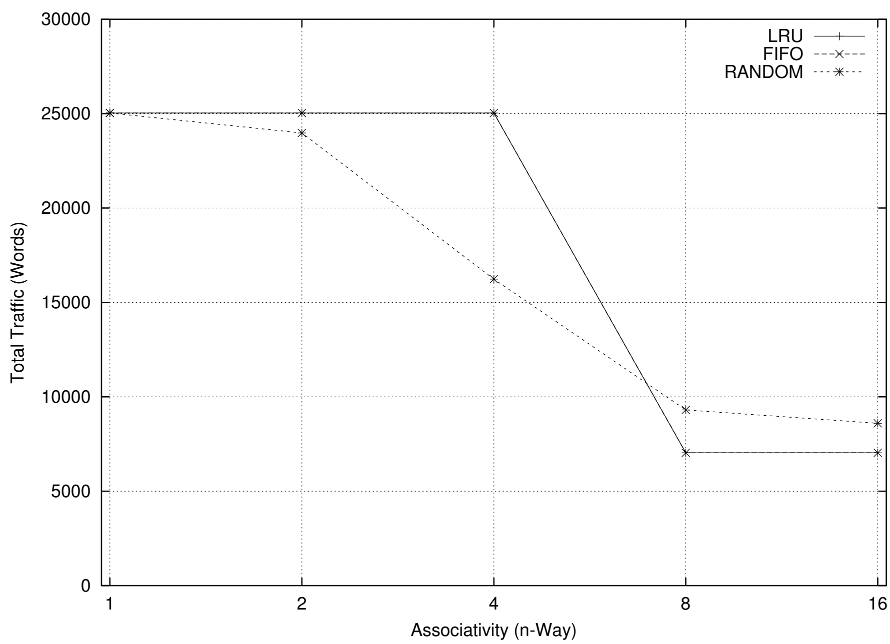

Abbildung 12: non-unified 1536 byte cache (512 byte data/1024 byte instr)

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23976
4-way	25032	25032	16232
8-way	7032	7032	9308
16-way	7032	7032	8600

Tabelle 13: non-unified 2048 byte cache (512 byte data/1536 byte instr)

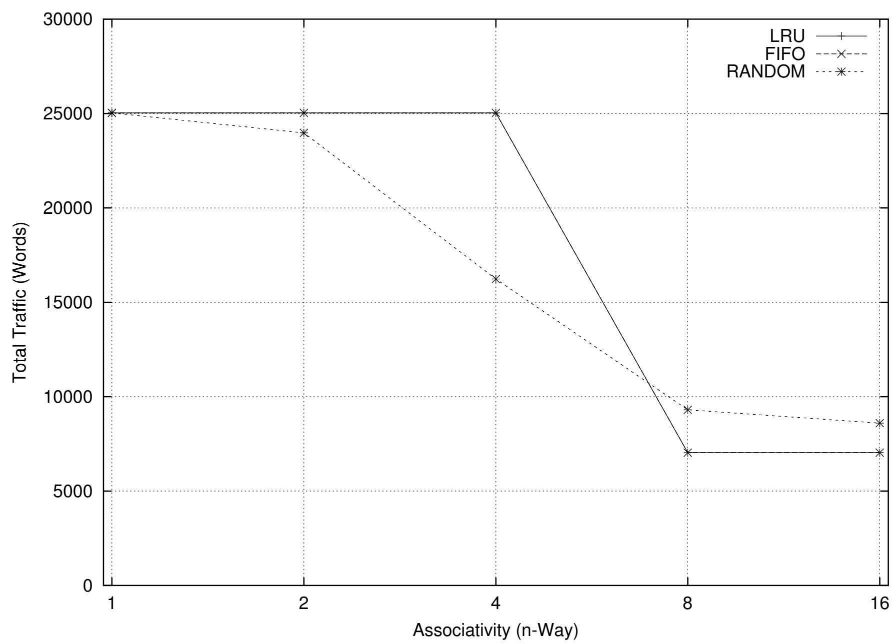

Abbildung 13: non-unified 2048 byte cache (512 byte data/1536 byte instr)

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23964
4-way	25032	25032	16496
8-way	7032	7032	9392
16-way	7032	7032	7928

Tabelle 14: non-unified 1280 byte cache (1024 byte data/256 byte instr)

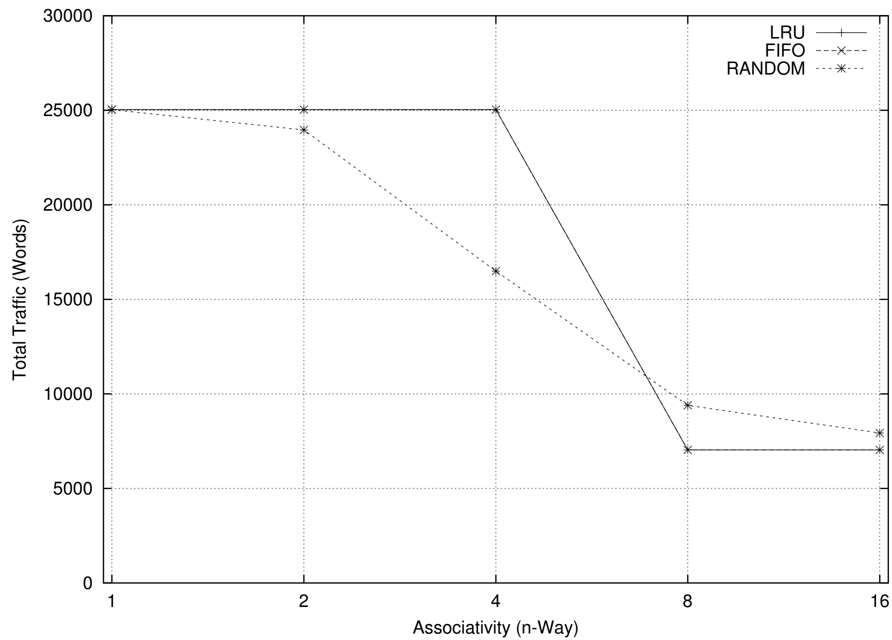

Abbildung 14: non-unified 1280 byte cache (1024 byte data/256 byte instr)

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23964
4-way	25032	25032	16212
8-way	7032	7032	9256
16-way	7032	7032	7956

Tabelle 15: non-unified 1536 byte cache (1024 byte data/512 byte instr)

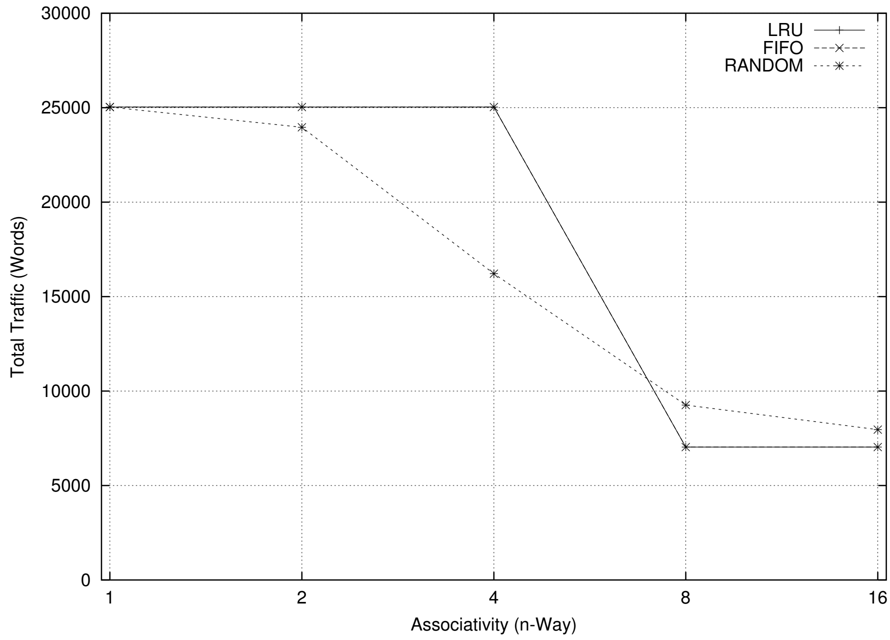

Abbildung 15: non-unified 1536 byte cache (1024 byte data/512 byte instr)

	LRU	FIFO	RAND
1-way	25032	25032	25032
2-way	25032	25032	23964
4-way	25032	25032	16212
8-way	7032	7032	9256
16-way	7032	7032	7892

Tabelle 16: non-unified 2048 byte cache (1024 byte data/1024 byte instr)

	LRU	FIFO	RAND
1-way	22032	22032	22032
2-way	7032	7032	9436
4-way	7032	7032	7820
8-way	7032	7032	7412
16-way	7032	7032	7356

Tabelle 17: non-unified 1792 byte cache (1536 byte data/256 byte instr)

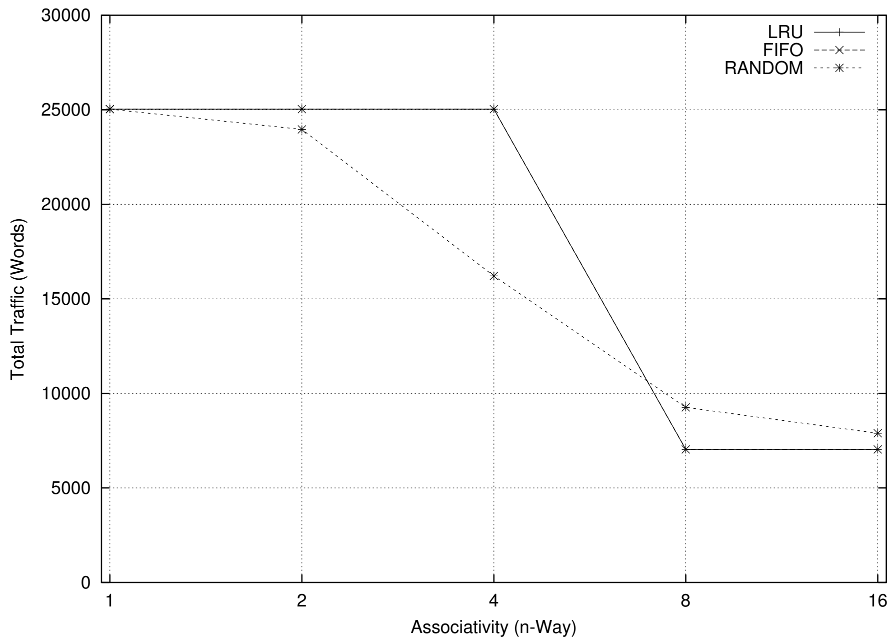

Abbildung 16: non-unified 2048 byte cache (1024 byte data/1024 byte instr)

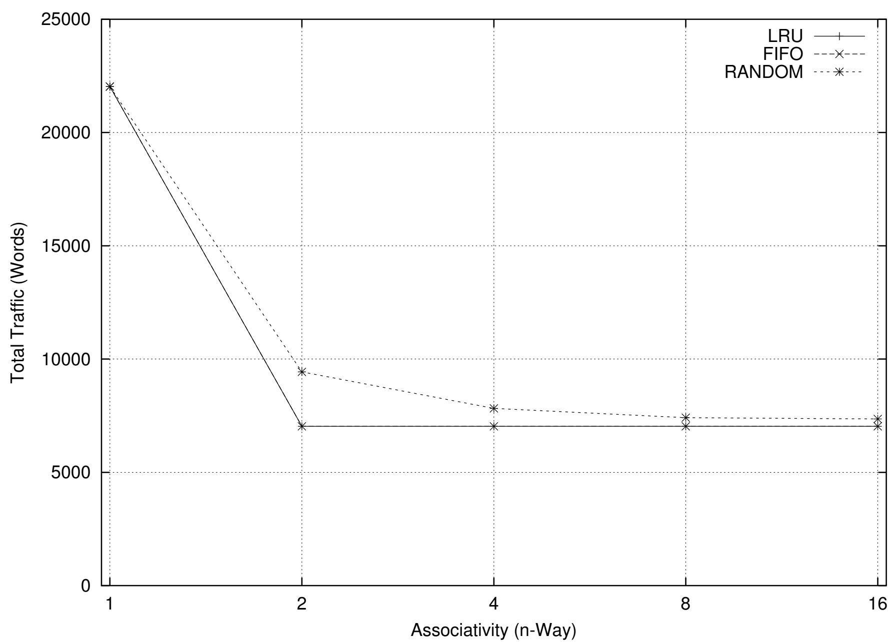

Abbildung 17: non-unified 1792 byte cache (1536 byte data/256 byte instr)

	LRU	FIFO	RAND
1-way	22032	22032	22032
2-way	7032	7032	9436
4-way	7032	7032	7820
8-way	7032	7032	7440
16-way	7032	7032	7324

Tabelle 18: non-unified 2048 byte cache (1536 byte data/512 byte instr)

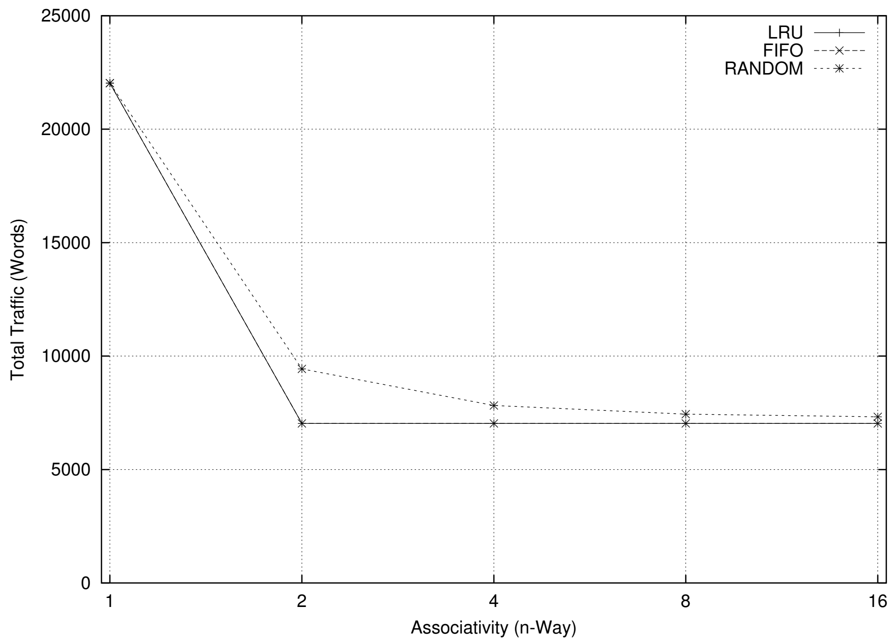

Abbildung 18: non-unified 2048 byte cache (1536 byte data/512 byte instr)

	LRU	FIFO	RAND
1-way	7032	7032	7032
2-way	7032	7032	7032
4-way	7032	7032	7032
8-way	7032	7032	7032
16-way	7032	7032	7096

Tabelle 19: non-unified 2048 byte cache (1792 byte data/256 byte instr)

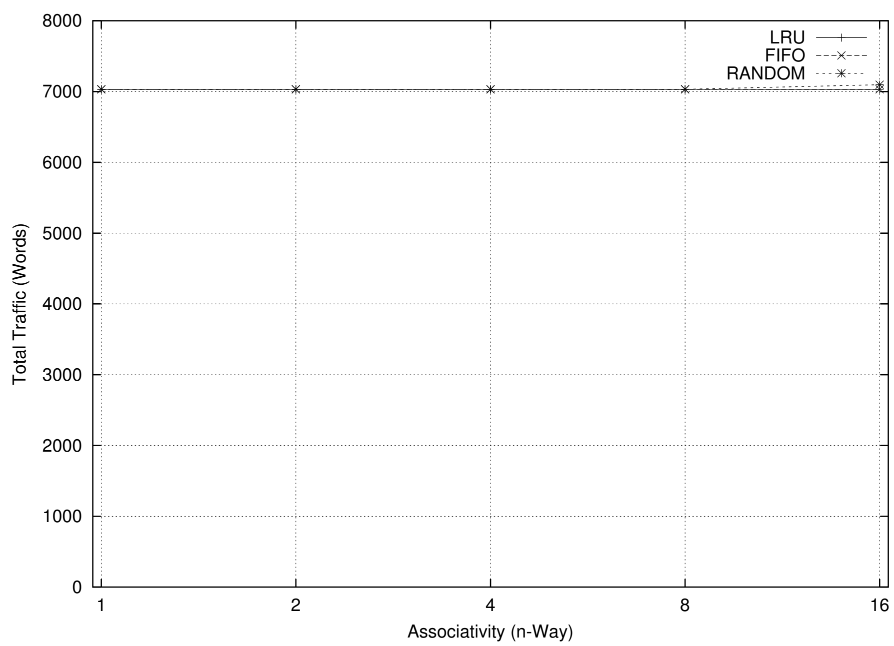

Abbildung 19: non-unified 2048 byte cache (1792 byte data/256 byte instr)

1.4 Aufgabe 4

Je größer der Cache und je höher die Assoziativität ist, desto geringer ist die Anzahl der Cachemisses. Jedoch wurde bei den Berechnungen nicht die höhere Komplexität bei höherer Assoziativität miteinbezogen.

Die Unified-Cache Variante neigt bei geringer Assoziativität dazu, daß die Instruktionen vorzeitig im Cache aufgrund der Datenzugriffe ersetzt werden.

Ab einer bestimmten Größe des Caches, unabhängig davon, ob unified oder non-unified, finden nurmehr bestenfalls einmalig, weitgehend unabhängig von der Ersetzungsstrategie, Zugriffe auf den langsammeren Hauptspeicher statt.

Die beiden Strategien LRU und FIFO verhalten sich aufgrund der Zugriffsmuster im vorgegeben Programm weitgehend identisch. Für höhere Assoziativitäten verhält sich auch die Random-Strategie in der selben Effizienz wie die beiden anderen. Daß die Random-Strategie bei kleineren Assoziativitäten effizienter arbeitet, liegt darin begründet, daß das Programm viele Speicherzugriffe auf Adressen durchführt, deren niederwertige Bits, die ja gerade die Position im Cache bestimmen, gleich sind, und somit die Cache-Einträge bei zu geringer Assoziativität zu früh ersetzt werden. Dadurch erklärt sich auch für den Non-Unified Cache, daß sich dort eine Verbesserung auf etwa 1/4 der Cachemisses von 4-way auf 8-way einstellt, weil dann nur bei jedem 4ten Schleifendurchlauf die ansich hintereinander liegenden Arraypositionen über einen 4er Word Block Transfer als Cachemiss vom Hauptspeicher geholt werden müssen.

Da das Programm selber deutlich unter 256 byte groß ist, würde ein größerer Instruktionscache keinen Vorteil bringen, daher bringt die Maximierung des Datencaches wesentlich mehr.

Schlußfolgerung

Bezieht man sich nur auf die ermittelten Werte, und läßt somit die Implementierungsproblematik außer Acht, so ergibt sich die non-unified 2048 byte Variante, mit 1792 byte für Daten und den verbleibenden 256 byte für Instruktionen als vorteilhaft, großteils unabhängig von der gewählten Zugriffspolitik und der Assoziativität, weil der Datencache groß genug.