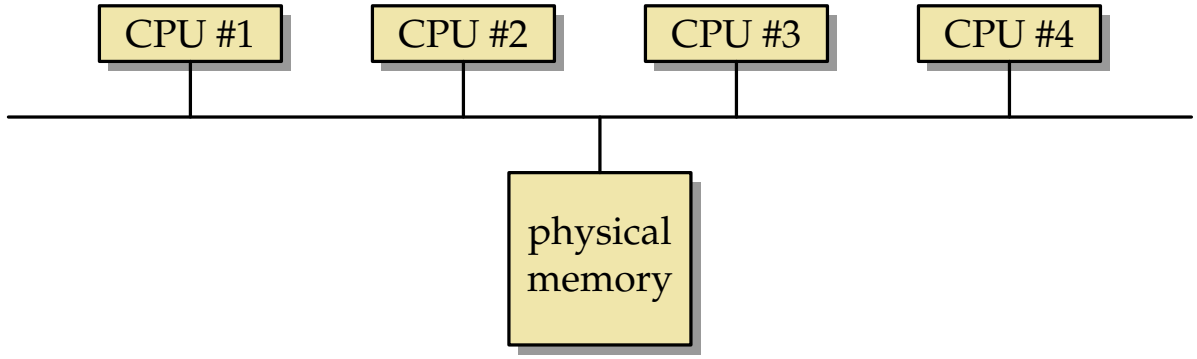
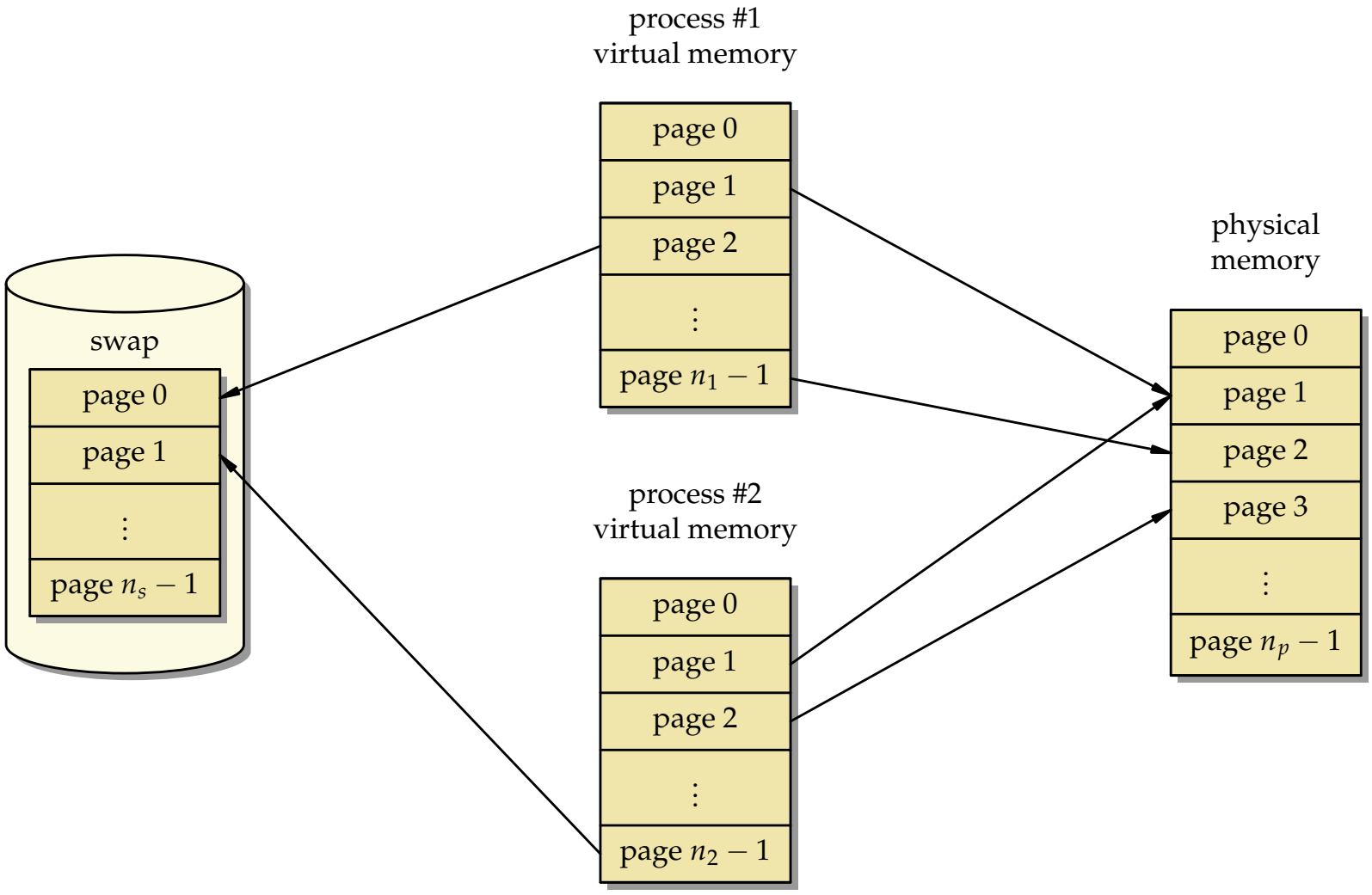


- Sofern gemeinsamer Speicher zur Verfügung steht, so bietet dies die effizienteste Kommunikationsbasis parallelisierter Programme.
- Die Synchronisation muss jedoch auf anderem Wege erfolgen. Dies kann entweder mittels der entsprechenden Operationen für Threads (etwa mit *join* und auf Basis von Bedingungsvariablen), über lokale Netzwerkkommunikation oder anderen Synchronisierungsoperationen des Betriebssystems erfolgen.
- Am häufigsten anzutreffen ist die UMA-Variante (*uniform memory access*, siehe Abbildung). Diese Architektur finden wir auf der Thales und der Theseus vor.


process #1
virtual memory
page 0
page 1
page 2
⋮
page $n_1 - 1$
physical
memory
page 0
page 1
page 2
page 3
⋮
page $n_p - 1$
swap
page 0
page 1
⋮
page $n_s - 1$
process #2
virtual memory
page 0
page 1
page 2
⋮
page $n_2 - 1$

- Der Adressraum eines Prozesses ist eine virtuelle Speicherumgebung, die von dem Betriebssystem mit Unterstützung der jeweiligen Prozessorarchitektur (MMU = *memory management unit*) umgesetzt wird.
- Die virtuellen Adressen eines Prozesses werden dabei in physische Adressen des Hauptspeichers konvertiert.
- Für diese Abbildung wird der Speicher in sogenannte Kacheln (*pages*) eingeteilt.
- Die Größe einer Kachel ist systemabhängig. Auf der Theseus sind es 8 KiB, auf Thales und Hochwanner 4 KiB (abzurufen über den Systemaufruf *getpagesize*()).
- Wenn nicht genügend physischer Hauptspeicher zur Verfügung steht, können auch einzelne Kacheln auf Platte ausgelagert werden (*swap space*), was zu erheblichen Zeitverzögerungen bei einem nachfolgendem Zugriff führt.

- Jeder Prozess hat unter UNIX einen eigenen Adressraum.
- Mehrere Prozesse können gemeinsame Speicherbereiche haben (nicht notwendigerweise an den gleichen Adressen). Die Einrichtung solcher gemeinsamer Bereiche ist möglich mit den Systemaufrufen *mmap* (*map memory*) oder *shm_open* (*open shared memory object*).
- Jeder Prozess hat zu Beginn einen Thread und kann danach (mit *pthread_create* bzw. *std::thread*) weitere Threads erzeugen.
- Alle Threads eines Prozesses haben einen gemeinsamen virtuellen Adressraum. Gelegentlich wird bei Prozessen von Rechtegemeinschaften gesprochen, da alle Threads die gleichen Zugriffsmöglichkeiten und -rechte haben.

Zwei Aspekte in Bezug auf Korrektheit und Performance sind besonders relevant:

- Wie ist das Verhalten des gemeinsamen Speichers, wenn mehrere Threads konkurrierend und unsychronisiert (d.h. insbesondere ohne Mutex-Variablen) auf die gleichen Datenbereiche zugreifen?
- Welche Auswirkungen hat das Zugriffsverhalten der Threads auf die Performance?

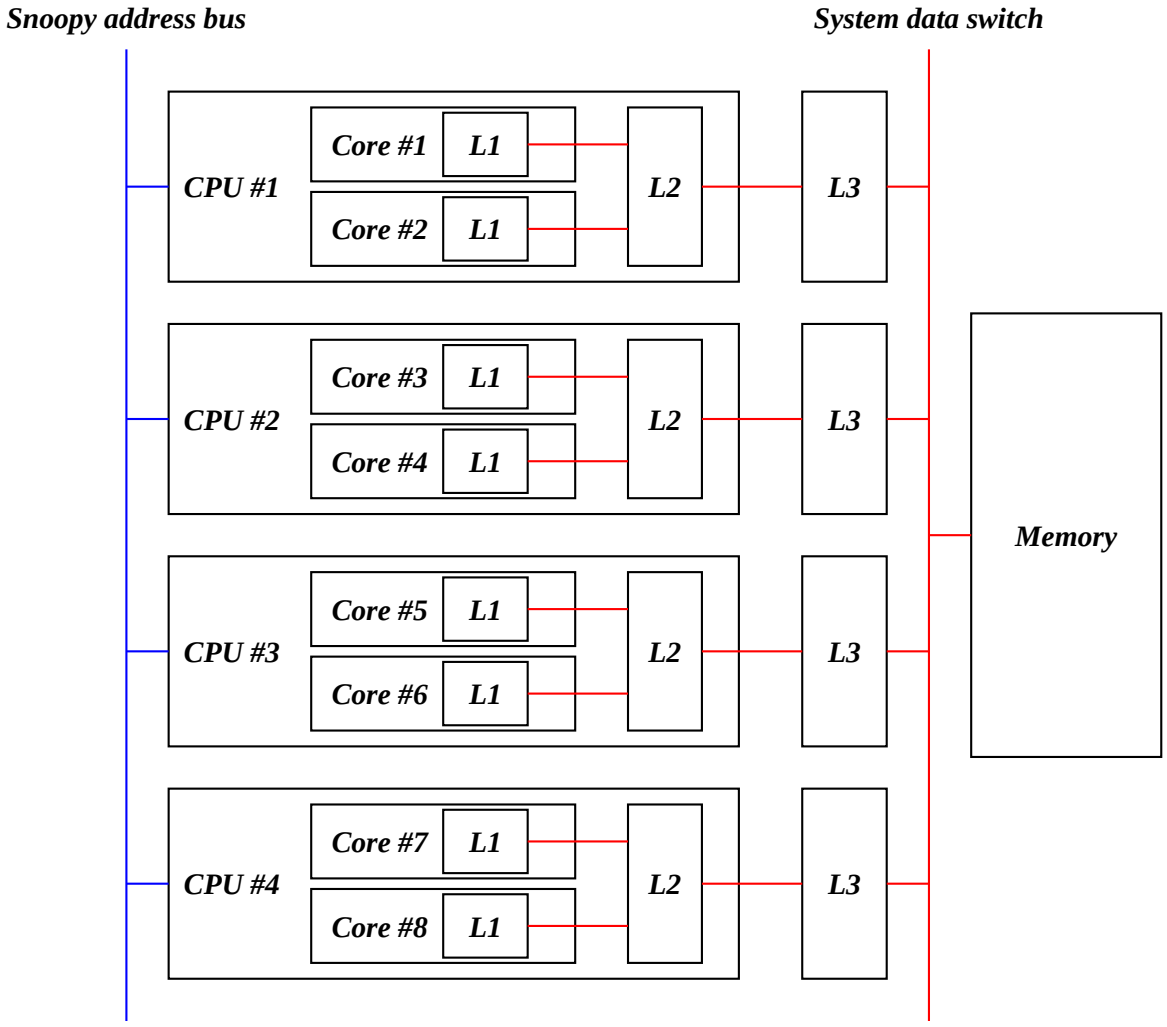
Die Relevanz ergibt sich daraus, dass

- gemeinsamer Speicher sich nicht mehr „intuitiv“ verhält bzw. ein „intuitives“ Verhalten so restriktiv wäre, dass ein erhebliches Optimierungspotential verloren gehen würde, und dass
- ohne sorgfältige Planung und Koordinierung von Speicherzugriffen in Abhängigkeit der gegebenen Architektur die Laufzeit ein Vielfaches im Vergleich mit einer optimierten Konfiguration betragen kann.

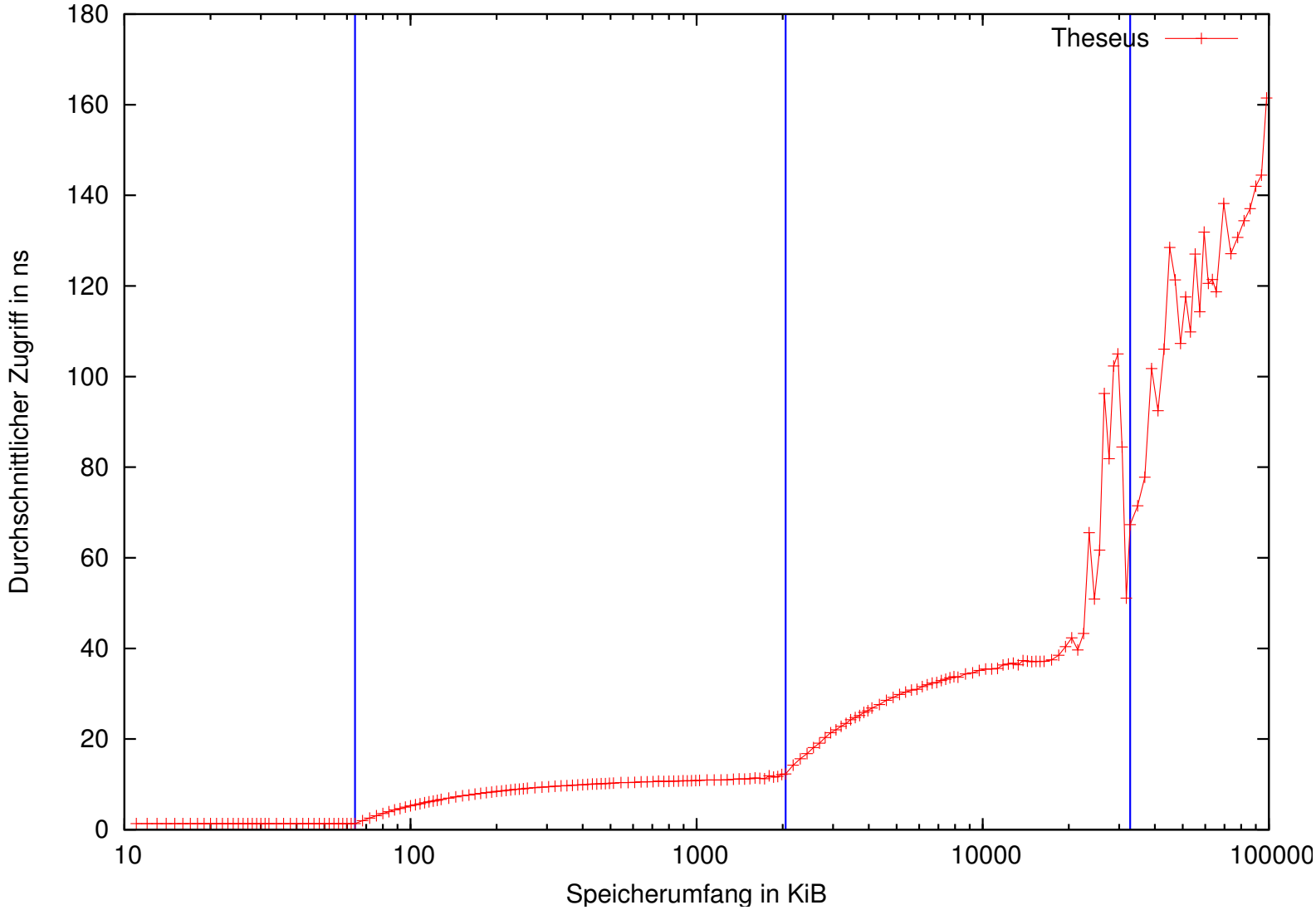
- Zugriffe einer CPU auf den primären Hauptspeicher sind vergleichsweise langsam. Obwohl Hauptspeicher generell schneller wurde, behielten die CPUs ihren Geschwindigkeitsvorsprung.
- Grundsätzlich ist Speicher direkt auf einer CPU deutlich schneller. Jedoch lässt sich Speicher auf einem CPU-Chip aus Komplexitäts-, Produktions- und Kostengründen nicht beliebig ausbauen.
- Deswegen arbeiten moderne Architekturen mit einer Kette hintereinander geschalteter Speicher. Zur Einschätzung der Größenordnung sind hier die Angaben für die Theseus, die mit Prozessoren des Typs UltraSPARC IV+ ausgestattet ist:

Cache	Kapazität	Taktzyklen
Register		1
L1-Cache	64 KiB	2-3
L2-Cache	2 MiB	um 10
L3-Cache	32 MiB	um 60
Hauptspeicher	32 GiB	um 250

Cache-Struktur der Theseus

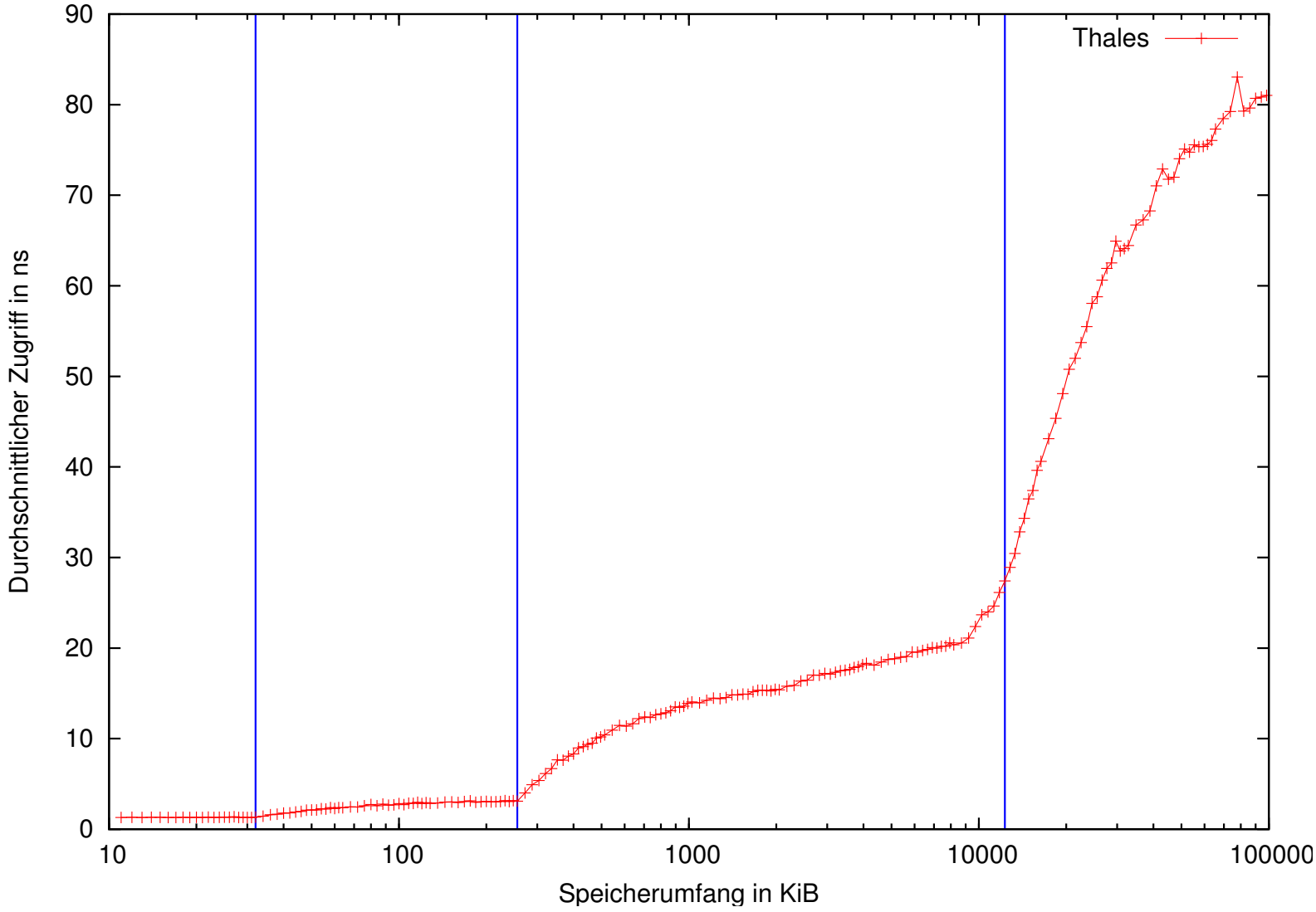


Gemessene Speicherzugriffszeiten: Theseus



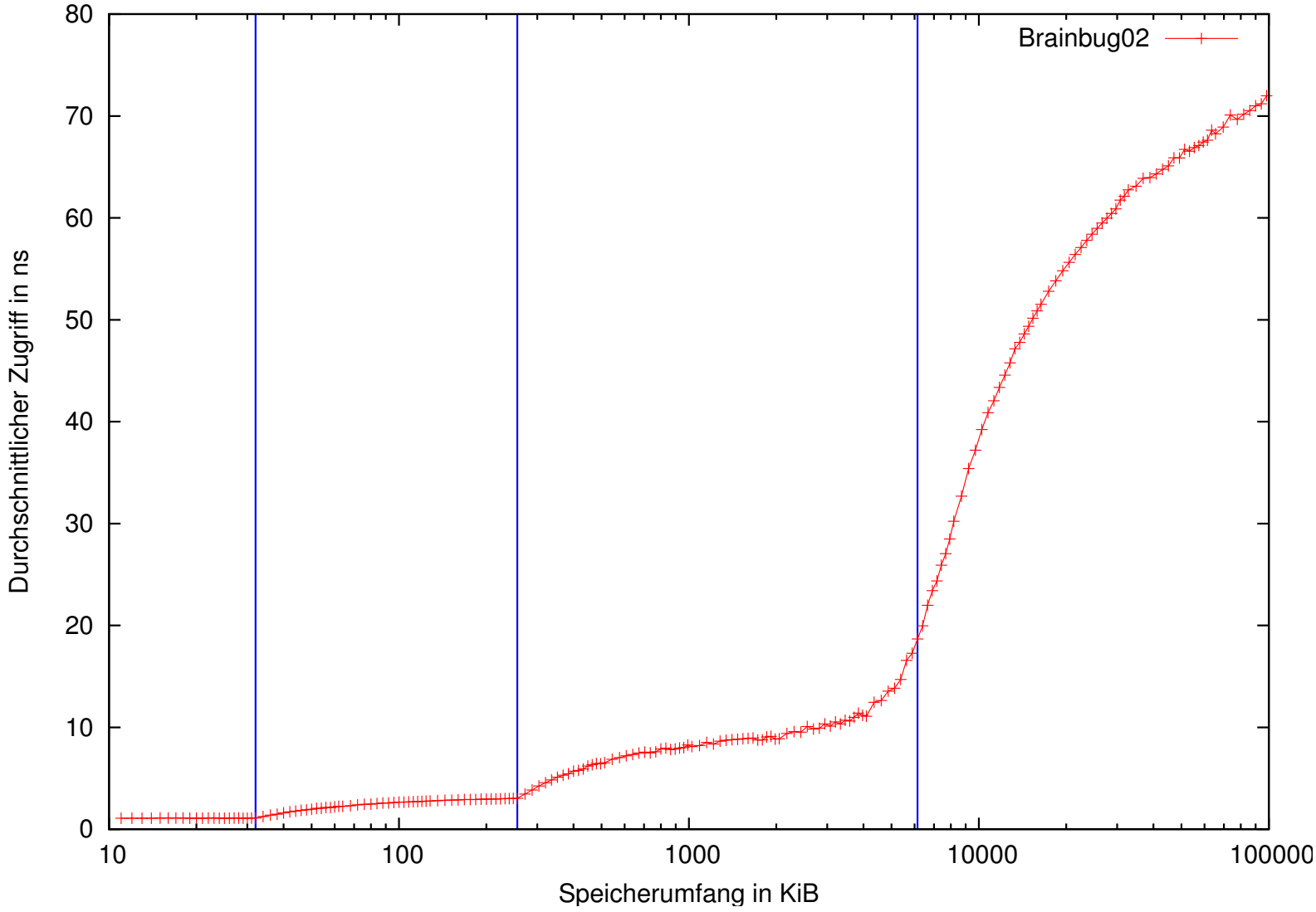
Theseus: 4 UltraSPARC IV+ Prozessoren mit je zwei Kernen
Caches: L1 (64 KiB), L2 (2 MiB), L3 (32 MiB)

Gemessene Speicherzugriffszeiten: Thales



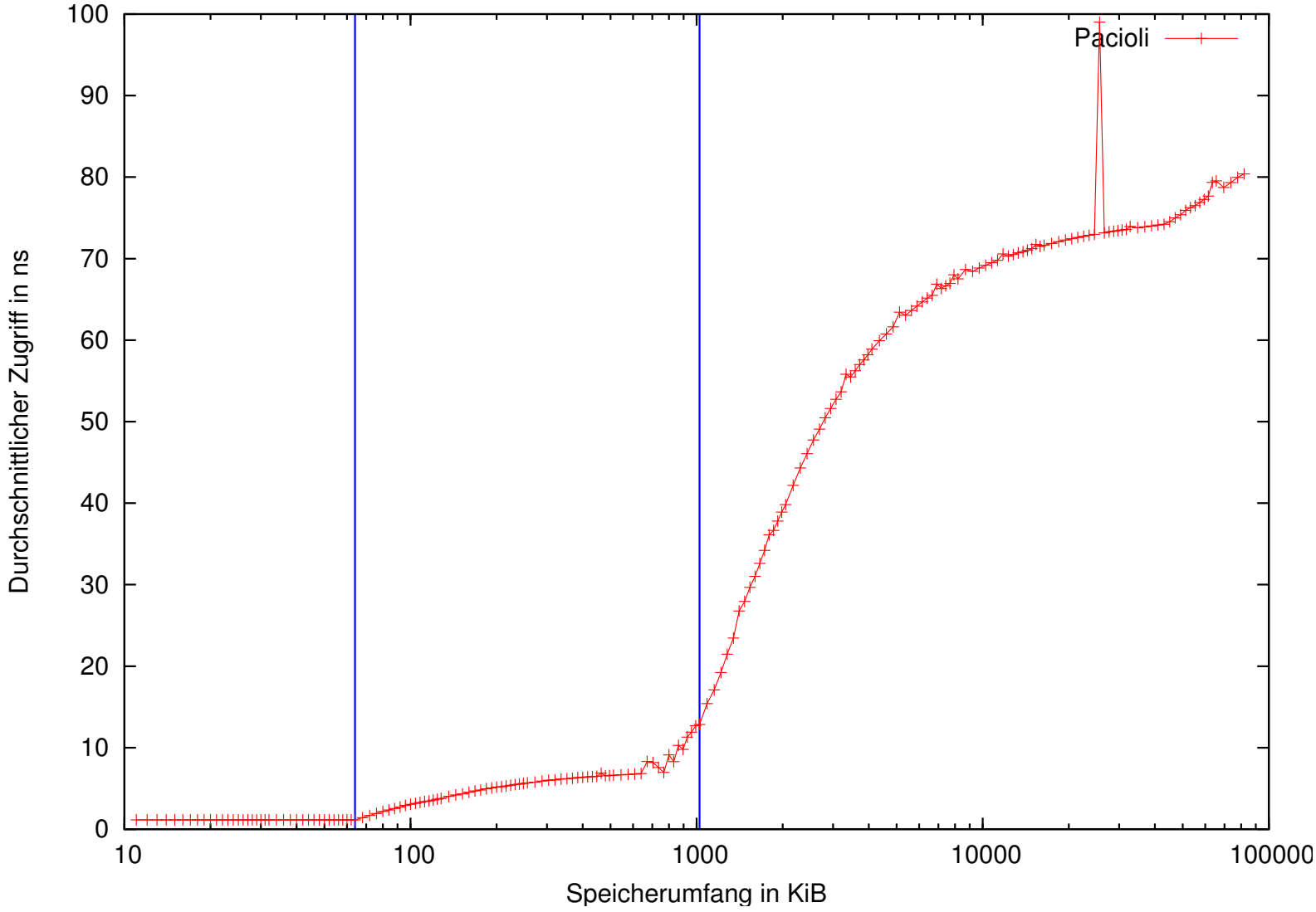
Thales: 2 Intel X5650-Prozessoren mit je 6 Kernen mit je 2 Threads
Caches: L1 (32 KiB), L2 (256 KiB), L3 (12 MiB)

Gemessene Speicherzugriffszeiten: Brainbug02



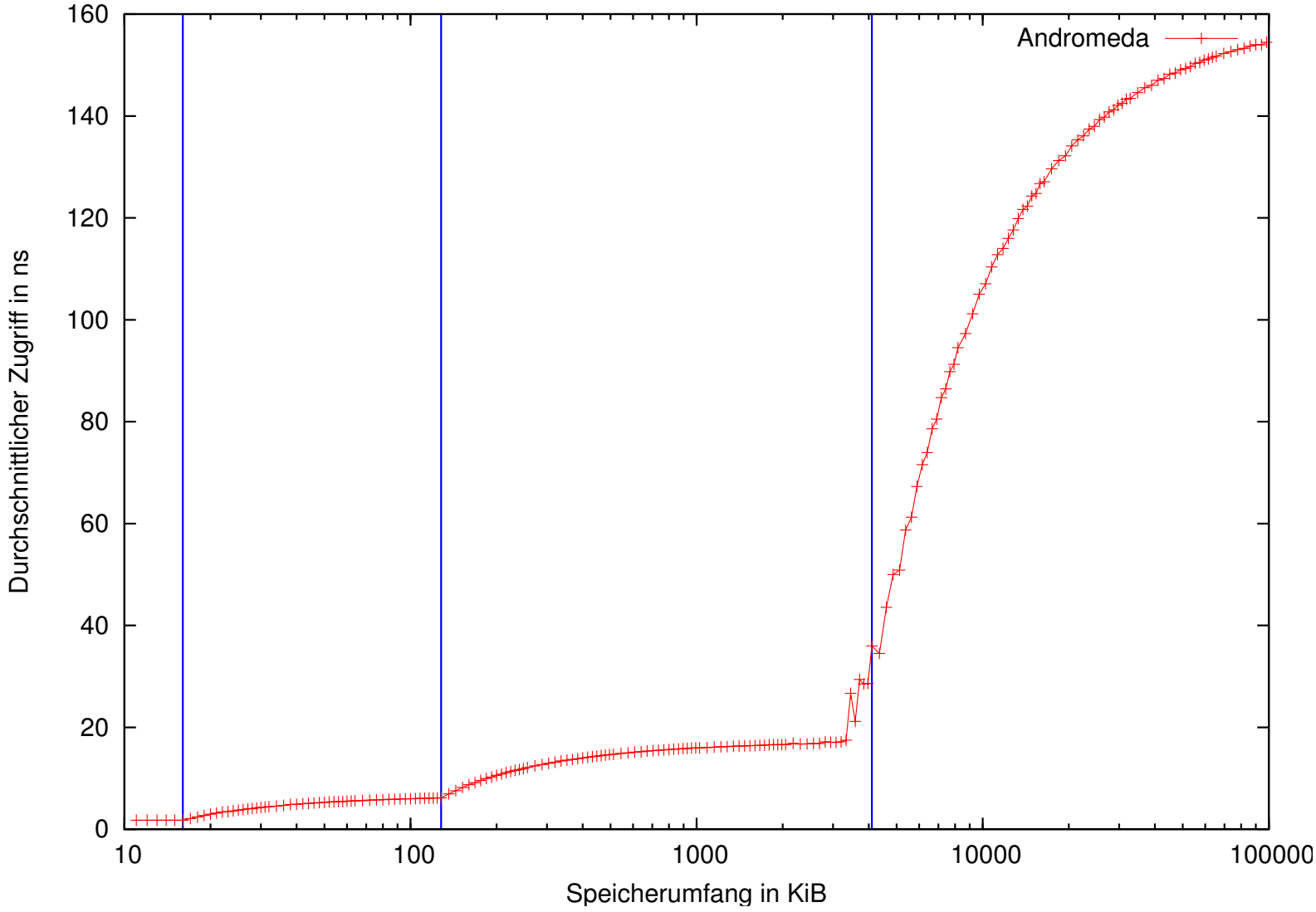
Brainbug02: Intel Quadcore i5-2500S CPU
Caches: L1 (32 KiB), L2 (256 KiB), L3 (6 MiB)

Gemessene Speicherzugriffszeiten: Pacioli



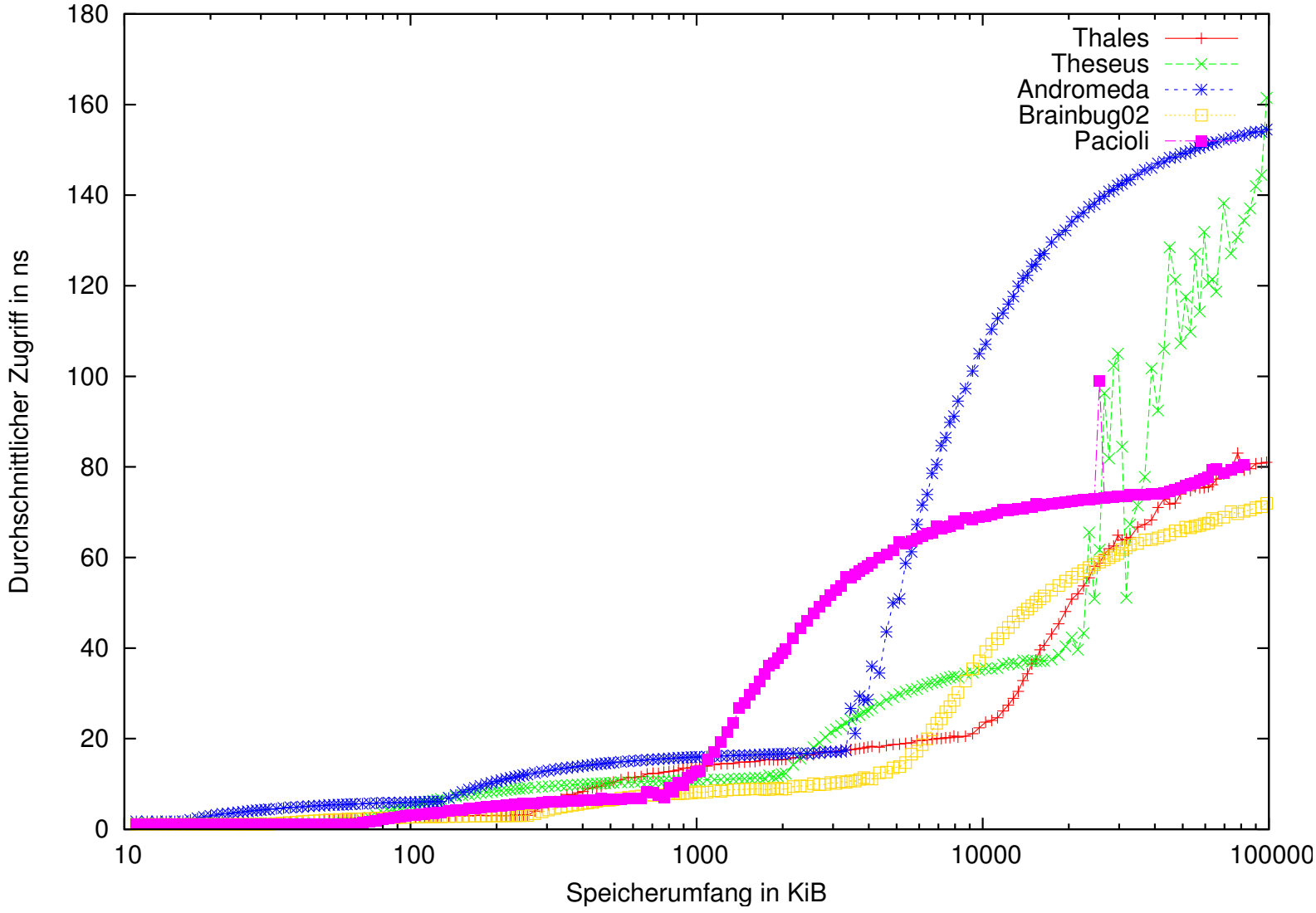
Pacioli: AMD Opteron 252
Caches: L1 (64 KiB), L2 (1 MiB)

Gemessene Speicherzugriffszeiten: Andromeda

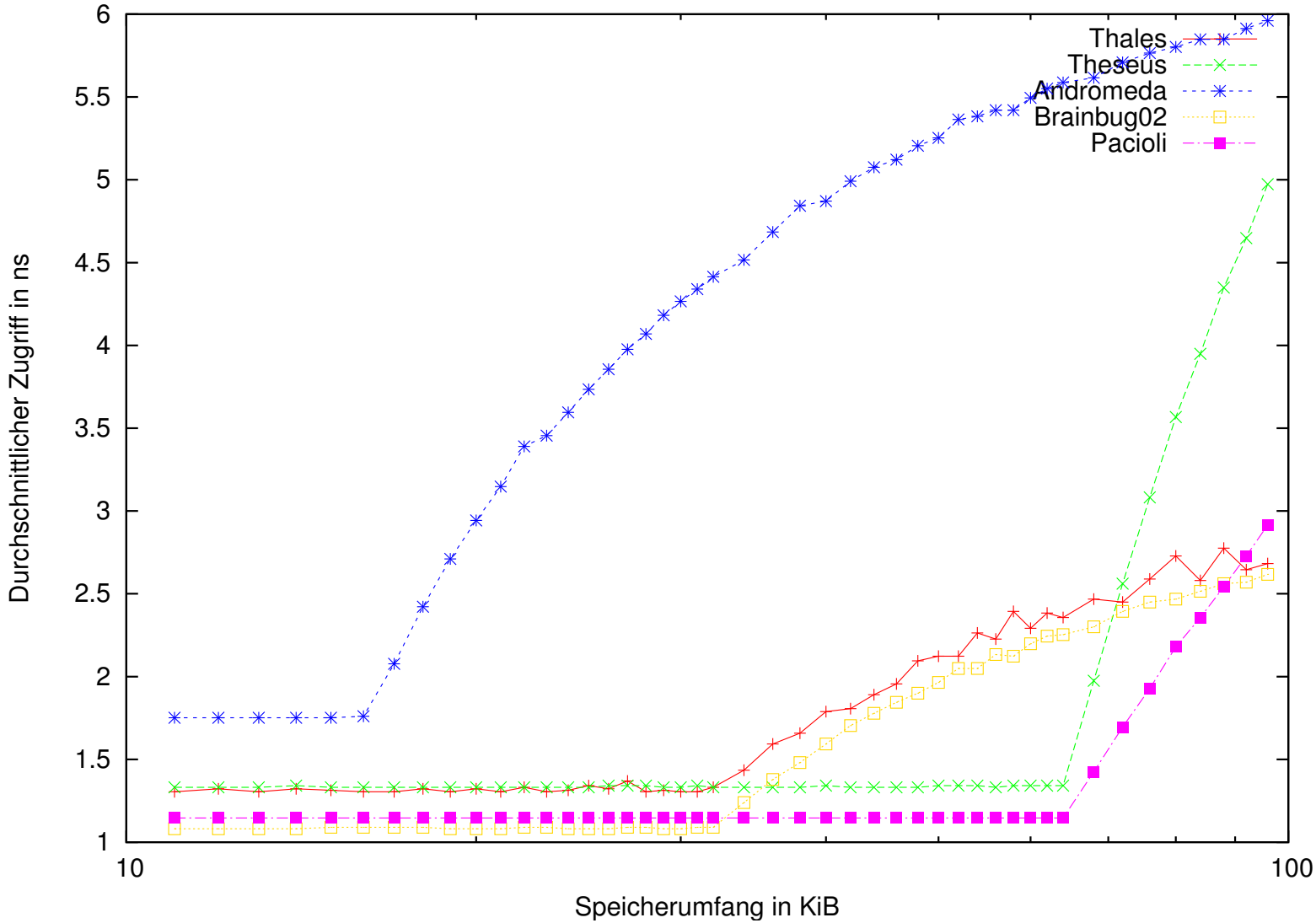


Andromeda: 2 SPARC-T4-Prozessoren mit je 8 Kernen mit je 8 Threads
Caches: L1 (16 KiB), L2 (128 KiB), L3 (4 MiB)

Gemessene Speicherzugriffszeiten: Vergleich



Gemessene Speicherzugriffszeiten im L1-Bereich



- Ein Cache ist in sogenannten *cache lines* organisiert, d.h. eine *cache line* ist die Einheit, die vom Hauptspeicher geladen oder zurückgeschrieben wird.
- Jede der *cache lines* umfasst – je nach Architektur – 32 - 128 Bytes. Auf der Theseus sind es beispielsweise 64 Bytes.
- Jede der *cache lines* kann unabhängig voneinander gefüllt werden und einem Abschnitt im Hauptspeicher entsprechen.
- Das bedeutet, dass bei einem Zugriff auf `a[i]` mit recht hoher Wahrscheinlichkeit auch `a[i+1]` zur Verfügung steht.
- Entweder sind Caches vollassoziativ (d.h. jede *cache line* kann einen beliebigen Hauptspeicherabschnitt aufnehmen) oder für jeden Hauptspeicherabschnitt gibt es nur eine *cache line*, die in Frage kommt (*fully mapped*), oder jeder Hauptspeicherabschnitt kann in einen von *n* *cache lines* untergebracht werden (*n-way set associative*).

- Diese Optimierungsechnik des Übersetzers bemüht sich darum, die Instruktionen (soweit dies entsprechend der Datenflussanalyse möglich ist) so anzuordnen, dass in der Prozessor-Pipeline keine Stockungen auftreten.
- Das lässt sich nur in Abhängigkeit des konkret verwendeten Prozessors optimieren, da nicht selten verschiedene Prozessoren der gleichen Architektur mit unterschiedlichen Pipelines arbeiten.
- Ein recht großer Gewinn wird erzielt, wenn ein vom Speicher geladener Wert erst sehr viel später genutzt wird.
- Beispiel: `x = a[i] + 5; y = b[i] + 3;`
  Hier ist es für den Übersetzer sinnvoll, zuerst die Ladebefehle für `a[i]` und `b[i]` zu generieren und erst danach die beiden Additionen durchzuführen und am Ende die beiden Zuweisungen.

axpy.c

```
// y = y + alpha * x
void axpy(int n, double alpha, const double* x, double* y) {
   for (int i = 0; i < n; ++i) {
      y[i] += alpha * x[i];
   }
}
```

- Dies ist eine kleine Blattfunktion, die eine Vektoraddition umsetzt. Die Länge der beiden Vektoren ist durch n gegeben, x und y zeigen auf die beiden Vektoren.
- Aufrufkonvention:

Variable	Register
n	%o0
alpha	%o1 und %o2
x	%o3
y	%o4

Resultat von gcc -O3

axpy.s

```
        add     %sp, -120, %sp
        cmp     %o0, 0
        st      %o1, [%sp+96]
        st      %o2, [%sp+100]
        ble     .LL5
        ldd     [%sp+96], %f12
        mov     0, %g2
        mov     0, %g1
.LL4:
        ldd     [%g1+%o3], %f10
        ldd     [%g1+%o4], %f8
        add     %g2, 1, %g2
        fmuld   %f12, %f10, %f10
        cmp     %o0, %g2
        faddd   %f8, %f10, %f8
        std     %f8, [%g1+%o4]
        bne     .LL4
        add     %g1, 8, %g1
.LL5:
        jmp     %o7+8
        sub     %sp, -120, %sp
```

- Ein *loop unrolling* fand hier nicht statt, wohl aber ein *instruction scheduling*.

- Der C-Compiler von Sun generiert für die gleiche Funktion 241 Instruktionen (im Vergleich zu den 19 Instruktionen beim gcc).
- Der innere Schleifenkern mit 81 Instruktionen behandelt 8 Iterationen gleichzeitig. Das orientiert sich exakt an der Größe der *cache lines* der Architektur: $8 * \textbf{sizeof}(\textbf{double}) == 64$.
- Mit Hilfe der prefetch-Instruktion wird dabei jeweils noch zusätzlich dem Cache der Hinweis gegeben, die jeweils nächsten 8 Werte bei x und y zu laden.
- Der Code ist deswegen so umfangreich, weil
  - die Randfälle berücksichtigt werden müssen, wenn n nicht durch 8 teilbar ist und
  - die Vorbereitung recht umfangreich ist, da der Schleifenkern von zahlreichen bereits geladenen Registern ausgeht.

- Bei einem Mehrprozessorsystem hat jede CPU ihre eigenen Caches, die voneinander unabhängig gefüllt werden.
- Problem: Was passiert, wenn mehrere CPUs die gleiche *cache line* vom Hauptspeicher holen und sie dann jeweils verändern? Kann es passieren, dass konkurrierende CPUs von unterschiedlichen Werten im Hauptspeicher ausgehen?

- Die Eigenschaft der Cache-Kohärenz stellt sicher, dass es nicht durch die Caches zu Inkonsistenzen in der Sichtweise mehrerer CPUs auf den Hauptspeicher kommt.
- Die Cache-Kohärenz wird durch ein Protokoll sichergestellt, an dem die Caches aller CPUs angeschlossen sind. Typischerweise erfolgt dies durch Broadcasts über einen sogenannten Snooping-Bus, über den jeder Cache beispielsweise den anderen mitteilt, wenn Änderungen durchgeführt werden.
- Das hat zur Folge, dass bei konkurrierenden Zugriffen auf die gleiche *cache line* einer der CPUs sich diese wieder vom Hauptspeicher holen muss, was zu einer erheblichen Verzögerung führen kann.
- Deswegen sollten konkurrierende Threads nach Möglichkeit Schreibzugriffe auf die gleichen *cache lines* vermeiden.

vectors.cpp

```
void axpy(int n, double alpha,
      const double* x, int incX, double* y, int incY) {
   for (int i = 0; i < n; ++i, x += incX, y += incY) {
      *y += alpha * *x;
   }
}
```

- Die Funktion *axpy* berechnet

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \leftarrow \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} + \alpha \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

- Sowohl $\vec{x}$ als auch $\vec{y}$ liegen dabei nicht notwendigerweise als zusammenhängende Arrays vor. Stattdessen gilt $x_i = x[i*incX]$ (für $\vec{y}$ analog), so dass auch etwa die Spalte einer Matrix übergeben werden kann, ohne dass deswegen eine zusätzliche (teure) Umkopieraktion notwendig wird.

Verpackung der Vektoraddition als Thread

vectors.cpp

```
class AxpyThread {
   public:
      AxpyThread(int _n, double _alpha, double* _x, int _incX,
         double* _y, int _incY) :
            n(_n), alpha(_alpha), x(_x), incX(_incX),
            y(_y), incY(_incY) {
      }
      void operator()() {
         axpy(n, alpha, x, incX, y, incY);
      }
   private:
      int n;
      double alpha;
      double* x; int incX;
      double* y; int incY;
};
```

- Wie gehabt werden die Parameter wieder in eine Klasse verpackt.
- Doch wie werden die zu addierenden Vektoren auf die einzelnen Threads aufgeteilt?

vectors.cpp

```
void mt_axpy(int n, double alpha, double* x, int incX, double* y,
             int incY, int nofthreads) {
   assert(n > 0 && nofthreads > 0);
   thread axpy_thread[nofthreads];

   // spawn threads and pass parameters
   int chunk = n / nofthreads;
   int remainder = n % nofthreads;
   int nextX = 0; int nextY = 0;
   for (int i = 0; i < nofthreads; ++i) {
      int len = chunk;
      if (i < remainder) ++len;
      axpy_thread[i] = thread(AxpyThread(len, alpha,
         x + nextX * incX, incX, y + nextY * incY, incY));
      nextX += len; nextY += len;
   }

   // wait for all threads to finish
   for (int i = 0; i < nofthreads; ++i) {
      axpy_thread[i].join();
   }
}
```

bad-vectors.cpp

```
void mt_axpy(int n, double alpha, double* x, int incX, double* y,
             int incY, int nofthreads) {
   assert(n > 0 && nofthreads > 0);
   if (nofthreads > n) nofthreads = n;
   struct axpy_parameters* params = new axpy_parameters[nofthreads];
   pthread_t* thread = new pthread_t[nofthreads];

   int chunk = n / nofthreads;
   int remainder = n % nofthreads;
   int nextX = 0; int nextY = 0;
   for (int i = 0; i < nofthreads; ++i) {
      int len = chunk; if (i < remainder) ++len;
      params[i].n = len; params[i].alpha = alpha;
      params[i].x = x + i * incX; params[i].incX = incX * nofthreads;
      params[i].y = y + i * incY; params[i].incY = incY * nofthreads;
      if (pthread_create(&thread[i], 0, run_axpy_thread, &params[i])) {
         cerr << "unable to create threads, aborting" << endl; exit(1);
      }
      nextX += len; nextY += len;
   }

   for (int i = 0; i < nofthreads; ++i) {
      struct axpy_parameters* rp;
      if (pthread_join(thread[i], (void**) &rp)) {
         cerr << "pthread_join failed, aborting" << endl; exit(1);
      }
   }

   delete[] params; delete[] thread;
}
```

Laufzeiten je nach Aufteilung


```
theseus$ time vectors 10000000 1

real    0m0.78s
user    0m0.57s
sys     0m0.16s
theseus$ time bad-vectors 10000000 8

real    0m0.70s
user    0m2.33s
sys     0m0.17s
theseus$ time vectors 10000000 8

real    0m0.44s
user    0m0.66s
sys     0m0.17s
theseus$
```

- Der erste Parameter spezifiziert die Länge der Vektoren $n = 10^7$, der zweite die Zahl der Threads. (Dieses Beispiel könnte noch besser skalieren, wenn auch die Initialisierung der Vektoren parallelisiert worden wäre.)
- Die ungünstige alternierende Aufteilung führt dazu, dass Threads um die gleichen *cache lines* konkurrieren, so dass erhebliche Wartezeiten entstehen, die insgesamt den Vorteil der vielen Threads fast vollständig aufhoben.

```
theseus$ cputrack -c pic0=L2_rd_miss,pic1=L2L3_snoop_inv_sh \
> vectors 10000000 8
   time lwp      event      pic0      pic1
  0.302   2 lwp_create         0         0
  0.321   3 lwp_create         0         0
  0.382   4 lwp_create         0         0
  0.382   2   lwp_exit       224      1192
  0.382   3   lwp_exit       143       556
  0.442   5 lwp_create         0         0
  0.442   4   lwp_exit       134       278
  0.504   6 lwp_create         0         0
  0.504   5   lwp_exit        46       642
  0.562   7 lwp_create         0         0
  0.562   6   lwp_exit       128       373
  0.623   8 lwp_create         0         0
  0.623   7   lwp_exit       116       909
  0.664   9 lwp_create         0         0
  0.683   8   lwp_exit       126       400
  0.719   9   lwp_exit       172       486
  0.720   1       exit      4659      5437
theseus$
```

- Moderne CPUs erlauben die Auslesung diverser Cache-Statistiken. Unter Solaris geht dies mit *cputrack*. Hier liegt die Zahl der L2- und L3-Cache-Invalidierungen wegen Parallelzugriffen bei 5.437, während bei der ungünstigen Aufteilung diese auf 815.276 ansteigt...

```
theseus$ cputrack -c pic0=L2_rd_miss,pic1=L2L3_snoop_inv_sh \
> bad-vectors 10000000 8
   time lwp      event      pic0      pic1
  0.294   2 lwp_create         0         0
  0.301   3 lwp_create         0         0
  0.432   4 lwp_create         0         0
  0.518   5 lwp_create         0         0
  0.518   2   lwp_exit      1076    195702
  0.518   3   lwp_exit       186      1222
  0.692   6 lwp_create         0         0
  0.693   4   lwp_exit      4508    304384
  0.820   7 lwp_create         0         0
  0.821   5   lwp_exit       109      2176
  0.932   8 lwp_create         0         0
  0.932   6   lwp_exit       196       856
  0.947   9 lwp_create         0         0
  0.948   7   lwp_exit       131       959
  1.007   1       tick      3230       646
  1.067   8       tick       276    308152
  1.077   9       tick        25       702
  1.079   8   lwp_exit       369    308404
  1.110   9   lwp_exit       123       893
  1.112   1       exit     10263    815276
theseus$
```

- Es sind hier nicht alle Threads gleichermaßen betroffen, da jeweils zwei Threads sich einen Prozessor (mit zwei Kernen) teilen, die den L2- und L3-Cache gemeinsam nutzen. (Den L1-Cache gibt es jedoch für jeden Kern extra.)

- Auf dem Prozessor CPU_i führt der Prozess P_i sequentiell Instruktionen aus, zu denen auch Speicher-Instruktionen gehören.
- Die Speicher-Instruktionen lassen sich vereinfacht einteilen:
  - Schreib-Instruktionen, die den Inhalt einer Speicherzelle an einer gegebenen Adresse ersetzen und als abgeschlossen gelten, sobald der neue Inhalt für alle anderen Prozessoren sichtbar ist.
  - Lade-Instruktionen, die den Inhalt einer Speicherzelle an einer gegebenen Adresse auslesen und als abgeschlossen gelten, sobald der geladene Inhalt durch Schreib-Instruktionen anderer Prozessoren nicht mehr beeinflusst werden kann.
  - Atomare Lade/Schreib-Instruktionen, die beides miteinander integrieren und sicherstellen, dass die adressierte Speicherzelle nicht mittendrin durch andere Instruktionen verändert wird.

$$X_i <_p Y_i$$

- X_i und Y_i seien beliebige Speicher-Instruktionen des Prozesses P_i.
- Dann gilt die auf den Prozess P_i bezogene Relation $X_i <_p Y_i$, wenn die Operation X_i vor der Instruktion Y_i ausgeführt wird.
- Die Relation $<_p$ ist eine Totalordnung, d.h. es gilt entweder $X_i <_p Y_i$ oder $Y_i <_p X_i$, falls $X_i \neq Y_i$.
- (Diese und die folgenden Definitionen und Notationen wurden dem Kapitel D des *SPARC Architecture Manual* entnommen.)

$$X_i <_d Y_i$$

- Es gilt $X_i <_d Y_i$ genau dann, wenn $X_i <_p Y_i$ und mindestens eine der folgenden Bedingungen erfüllt ist:
  - Y_i ist oder enthält eine Schreib-Instruktion, die einer bedingten Sprunganweisung folgt, die von X_i abhängt.
  - Y_i liest ein Register, das von X_i abhängt.
  - Die Schreib-Instruktion X_i und die Lade-Instruktion Y_i greifen auf die gleiche Speicherzelle zu.
- $<_d$ ist eine partielle Ordnung, die transitiv ist.

$$X <_m Y$$

- Die Relation $<_m$ reflektiert die Reihenfolge, in der die Speicher-Instruktionen X und Y erfolgen.
- Die Reihenfolge ist im allgemeinen nicht deterministisch.
- Stattdessen gibt es in Abhängigkeit des zur Verfügung stehenden Speichermodells einige Zusicherungen.
- Ferner können auch gewisse Ordnungen erzwungen werden.

$$M(X, Y)$$

- Alle gängigen Prozessorarchitekturen bieten Instruktionen an, die eine Ordnung erzwingen. Diese werden *memory barriers* bzw. bei den Intel/AMD-Architekturen *fences* genannt.
- Eine Barrier-Instruktionen steht zwischen den Instruktionen, auf die sie sich bezieht.
- Eine Barrier-Instruktion spezifiziert, welche Sequenz von Speicher-Instruktionen geordnet wird. Prinzipiell gibt es vier Varianten bzw. Kombinationen davon: Lesen-Lesen, Lesen-Schreiben, Schreiben-Lesen und Schreiben-Schreiben.
- $M(X, Y)$ gilt dann, wenn wegen einer dazwischenliegende Barrier-Instruktion zuerst X und dann Y ausgeführt werden muss.

- Lesen-Lesen: Alle Lese-Operationen vor der Barrier-Instruktion müssen beendet sein, bevor folgende Lese-Operationen durchgeführt werden können.
- Lesen-Schreiben: Alle Lese-Operationen vor der Barrier-Instruktion müssen beendet sein, bevor die folgenden Schreib-Operationen für irgendeinen Prozessor sichtbar werden.
- Schreiben-Lesen: Die Schreib-Operationen vor der Barrier-Instruktion müssen abgeschlossen sein, bevor die folgenden Lese-Operationen durchgeführt werden.
- Schreiben-Schreiben: Die Schreib-Operationen vor der Barrier-Instruktion müssen alle beendet sein, bevor die folgenden Schreib-Operationen ausgeführt werden.

Eine Speicherordnung $<_m$ ist RMO genau dann, wenn

- $X <_d Y \wedge L(X) \Rightarrow X <_m Y$
- $M(X, Y) \Rightarrow X <_m Y$
- $Xa <_p Ya \wedge S(Y) \Rightarrow X <_m Y$

Xa steht für eine Speicher-Instruktion auf der Speicherzelle an der Adresse a; die Prädikate $L(X)$ und $S(Y)$ treffen zu, wenn X eine Lese-Instruktion ist oder umfasst bzw. Y eine Schreib-Instruktion ist oder umfasst.

- *Partial Store Order* (PSO) erfüllt alle Bedingungen der RMO. Hinzu kommt, dass allen ladenden Speicher-Instruktionen implizit eine Barrier-Instruktion für Lesen-Lesen und Lesen-Schreiben folgt.
- *Total Store Order* (TSO) erfüllt alle Bedingungen der PSO. Hinzu kommt, dass allen schreibenden Speicher-Instruktionen implizit eine Barrier-Instruktion für Schreiben-Schreiben folgt.
- *Sequential Consistency* (SQ) erfüllt alle Bedingungen der TSO. Hinzu kommt, dass allen schreibenden Speicher-Instruktionen eine Barrier-Instruktion für Schreiben-Lesen folgt.
- SQ ist das einfachste Modell, bei dem die Reihenfolge der Instruktionen eines einzelnen Prozesses sich direkt aus der Programmordnung ergibt. Andererseits bedeutet SQ, dass bei einem Schreibzugriff sämtliche Speicherzugriffe blockiert werden, bis die Cache-Invalidierungen abgeschlossen sind. Das ist sehr zeitaufwendig.

- Gegeben seien die beiden globalen Variablen a und b mit einem Initialwert von 0 und drei Prozesse mit den folgenden Instruktionen:

P_1	P_2	P_3
$a = 1;$	**int** $r1 = a;$	**int** $r1 = b;$
$b = 1;$	**int** $r2 = b;$	**int** $r2 = a;$

- Mögliche Szenarien (bei SQ und TSO nicht vollständig):

	P_2: $r1$	P_2: $r2$	P_3: $r1$	P_3: $r2$
Unter SQ:	0	0	0	0
	1	1	1	1
Unter TSO, !SQ:	1	0	1	1
Unter PSO, !TSO:	0	0	1	0
	0	1	1	0
	1	1	1	0
Unter RMO, !PSO:	1	0	1	0