Ausgewählte Kapitel der Systemsoftware

Betriebssystemkonzepte für moderne Arbeits- und Festspeichertechnologien

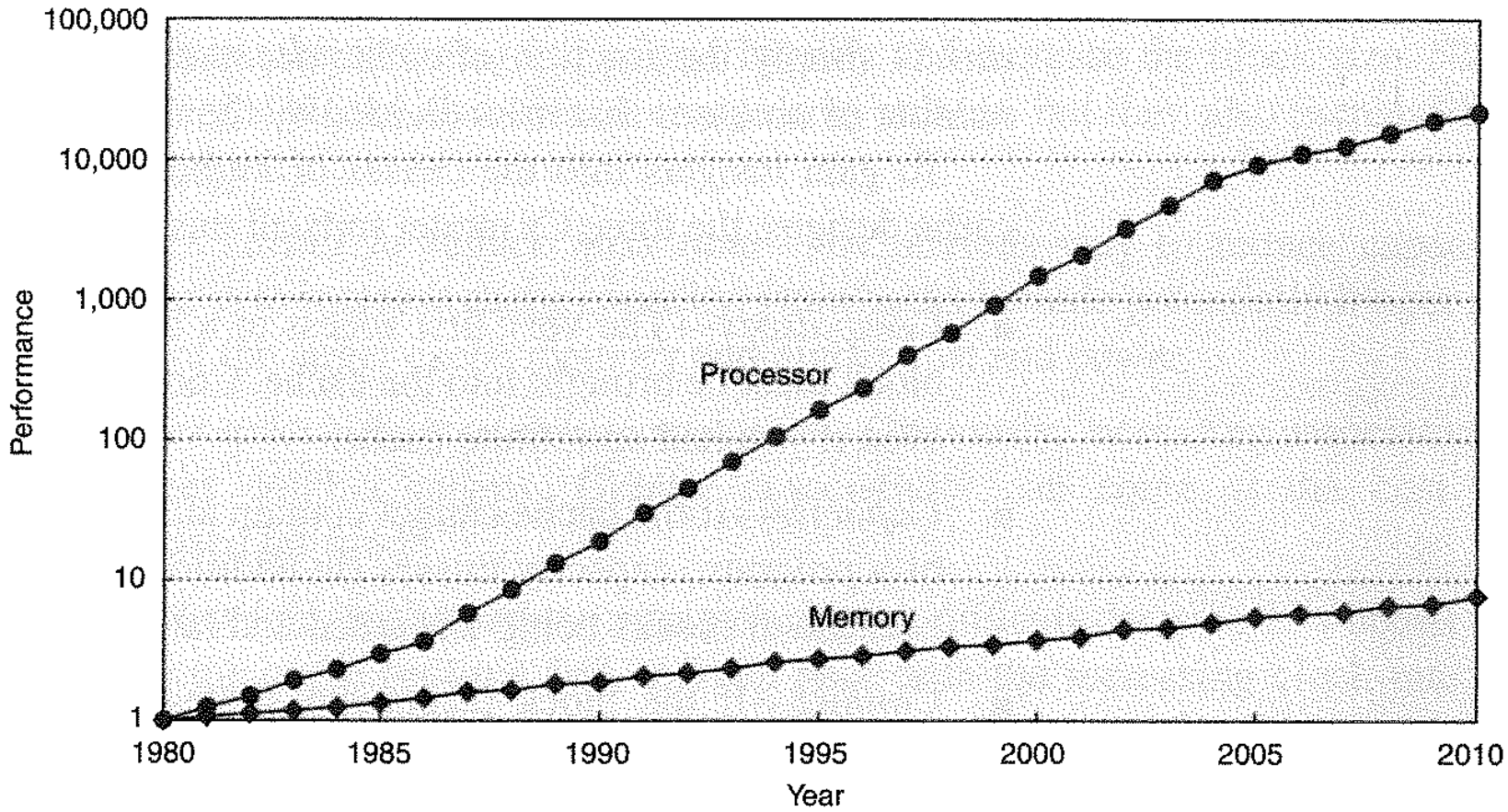
II. Disruptive Hauptspeichertechnologien

Wolfgang Schröder-Preikschat

5. November 2019

Gliederung

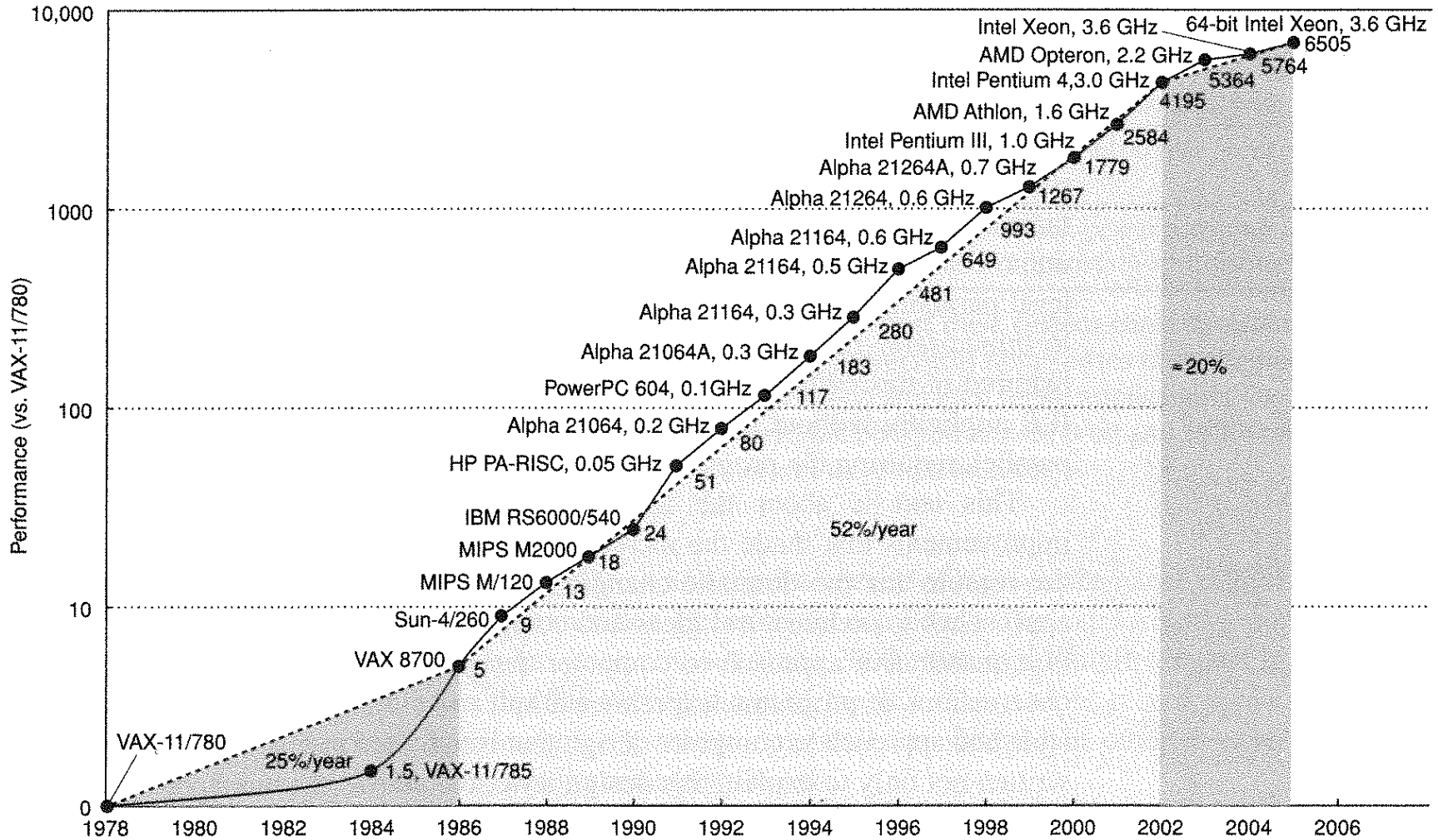
Leistungslücke zwischen CPU und RAM [7, S. 289]



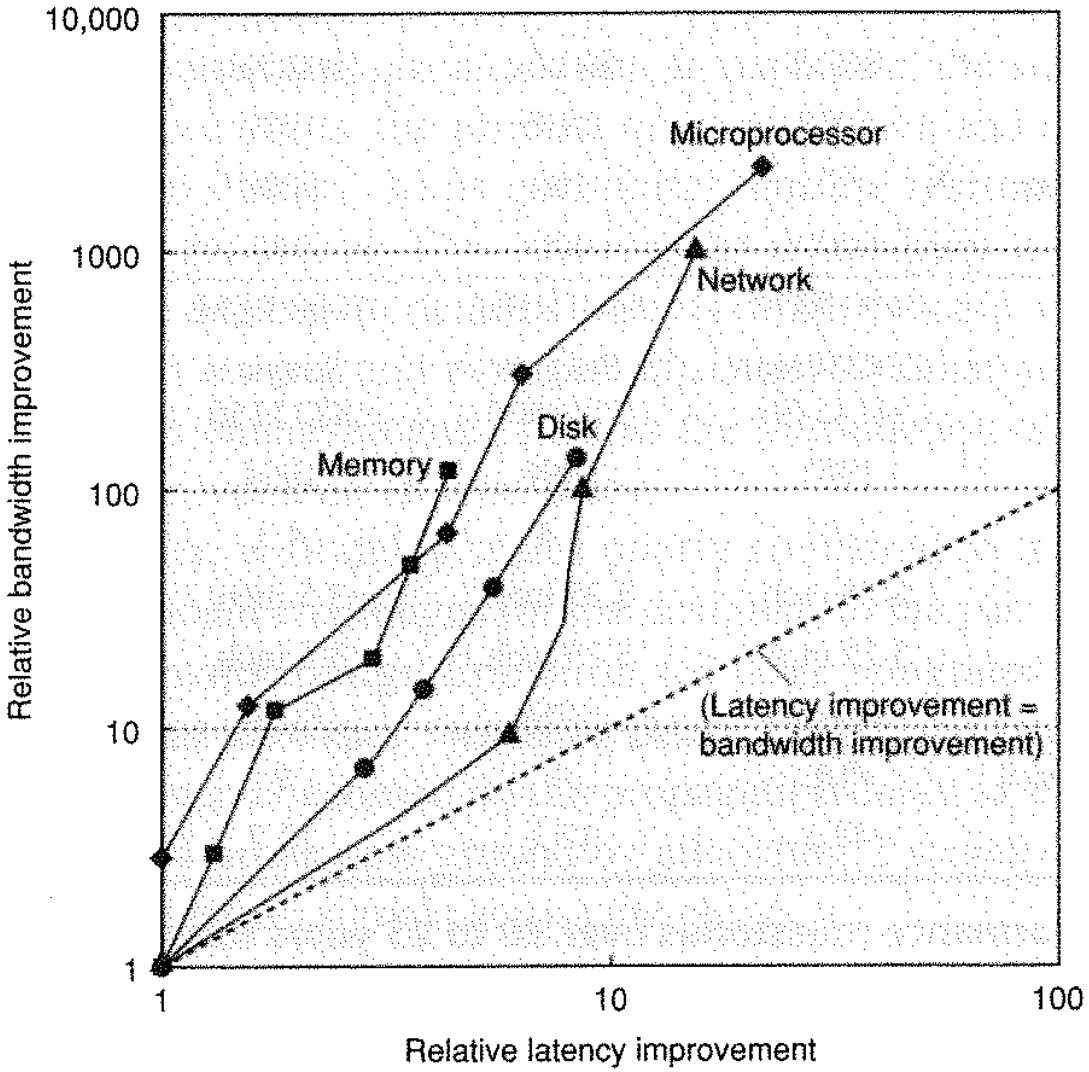
- angenommene Faktoren des jährlichen Leistungsanstiegs [7, S. 289]:
  1.07 bei der Speicherlatenz — versus Prozessorleistung (vgl. S. 4):
  - 1.25 bis 1986
  - 1.52 bis 2004
  - 1.20 danach

Wachstum der Prozessorleistung

[7, S. 3]



Leistungstrends: Bandbreite über Latenz [7, S. 15]



Clearly, bandwidth has outpaced latency across these technologies and will likely continue to do so.

A simple rule of thumb is that bandwidth grows as at least the square of the improvement in latency.

Bedeutung von „disruptiv“

- Neudeutsch, aus dem Englischen übernommen: (en.) dis·rup·tive
  - störend, spaltend, trennend, auflösend
  - Unruhe stiftend
- von (lat.) *disrumpere*
  - platzen, zerbrechen, zerreißen
- die Dis·rup·tion betreffend, beinhaltend oder schaffend
  - ein **Prozess**, der zur Ablösung oder Zerschlagung eines bestehenden Geschäftsmodells oder eines gesamten Marktes führt
  - verursacht durch eine **Innovation**, die etwas Gegebenes möglicherweise vollständig verdrängt
  - hier insbesondere eine bestehende **Technologie**, ein bestehendes Produkt oder eine bestehende Dienstleistung

↪ (en.) ***disruptive technology***

- durchschlagende, umwälzende Technologie

Direktzugriffsspeicher

- gemeinhin **RAM** (Abk. für (en.) *random access memory*)

  Bezeichnung für einen Speicher mit wahlfreiem/direktem Zugriff auf jedes Speicherwort über eine jeweils eindeutig zugeordnete Adresse. [12]

  - gleichfalls ist damit die Auslegung als **Schreib-lese-Speicher** gemeint
  - üblicherweise als **flüchtiger Speicher** (*volatile memory*) gesehen

- aber auch **nichtflüchtiger Speicher** (*nonvolatile memory*) ist „RAM“
  - permanent in Form von ROM oder PROM
  - semi-permanent als EPROM, EEPROM oder **NVRAM**

↪ **Zugriffsart** und **Fristigkeit** gespeicherter Daten sind zu trennen!

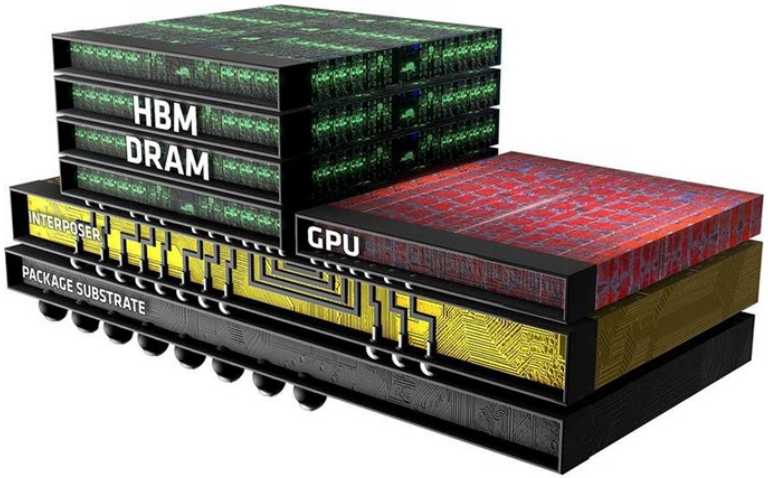
Sequenzieller Zugriff

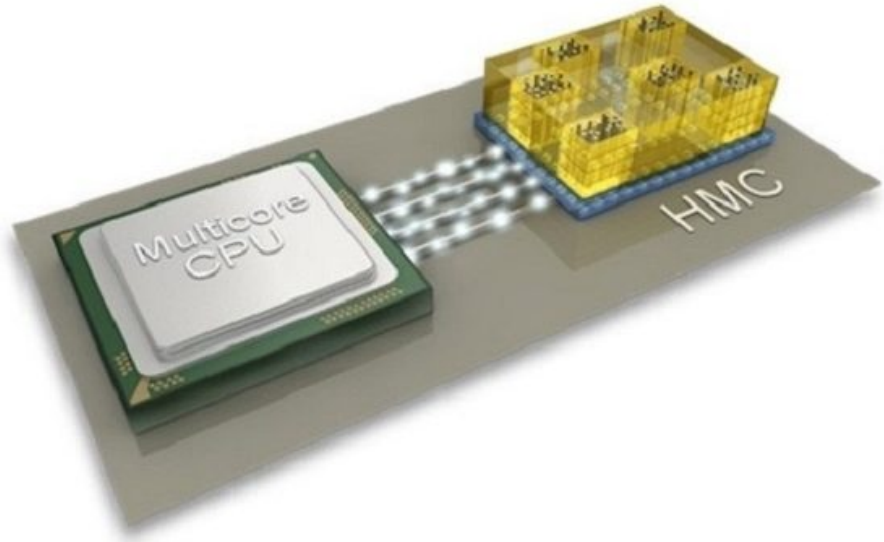
- fortlaufender, nacheinander erfolgender Zugriff auf die Speicherworte
  - eines einzelnen Prozesses oder mehrerer, gleichzeitiger Prozesse
  - über einen allen Prozess(or)en gemeinsamen Bus
- auch bekannt als **von-Neumann-Flaschenhals** [2, S. 615]

↪ klassisches Thema der **Rechnerarchitektur** ⇝ Globalsicht ✓
- Parallelverarbeitung, . . . , mehr-/vielkernige Prozessoren
- eine nicht mehr ganz so disruptive Technologie — sollte man meinen!

↪ herausfordernd dazu die **Hauptspeicherarchitektur** ⇝ Detaillsicht
- einerseits das Aufbauprinzip, um die Bandbreite über den Bus zu steigern
- andererseits das Funktionsprinzip, um den Bus seltener zu strapazieren

Aufbauprinzip





- High-Bandwidth Memory
- Hybrid Memory Cube

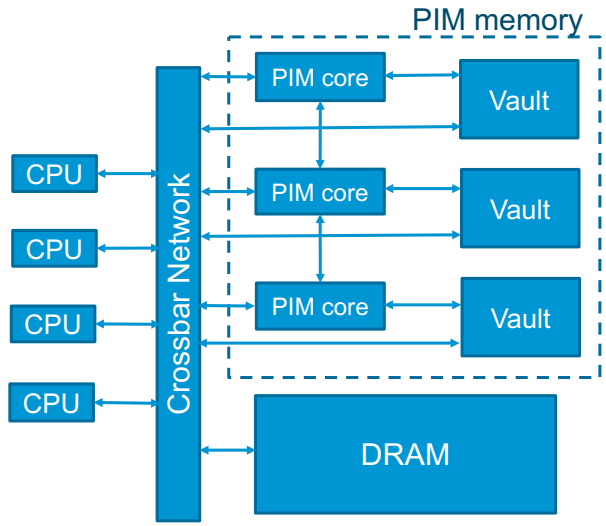
Setzen vertikal durchkontaktierte Speicherchip-Stapel ein, um sowohl die ***Speicherbandbreite*** *zu erhöhen als auch die* ***Zugriffslatenz*** *zu minimieren.*

↪ intelligente Nutzung der Logikschicht eines 3D-DRAM ⇝ PIM [5, 11]

Funktionsprinzip

Prozessorelemente (z.B. diverse dedizierte Beschleuniger) werden in die Nähe des Speichers verlagert (NMC) beziehungsweise direkt in den Speicher integriert (PIM), um die zu verarbeitenden Daten nicht immer zeit- und energieaufwendig durch ein Prozessor-Speichernetzwerk zu einer im Vergleich weit entfernten CPU transportieren zu müssen.

- Near Memory Computing (NMC)
- Processing in Memory (PIM)
  - PIM-Kammer (*vault*) ist lokaler Speicher eines PIM-Rechenkerns (*core*)
    - HMC mit 16 oder 32 solcher Kammern
    - mit (heute) etwa 128 MiB pro Kammer
  - Datenaustausch der PIM-Kerne durch Nachrichtenversenden (*message passing*)
  - jeder Prozessor (CPU) hat direkten Zugriff auf die Kammern und auf den zusätzlichen gemeinsamen Speicher



↪ **nebenläufige Datenstrukturen** (*concurrent data structures*) [9]

Dauer der Festlegung gespeicherter Informationen

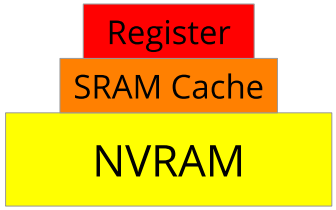
- DRAM (Abk. für (en.) *dynamic* RAM)
  - Leckströme verändern die in Kondensatoren gespeicherte Ladungsmenge
  - verliert Information, Auffrischen (*refresh*) notwendig alle 32ms – 64ms
  - mittlere Zugriffszeiten 20ns – 70ns
- SRAM (Abk. für (en.) *static* RAM)
  - jedes Bit wird in einer Transistorschaltung (Inverter) gespeichert
  - behält seine Information, solange die Betriebsspannung anliegt
  - mittlere Zugriffszeiten 0.5ns – 5ns

⋮

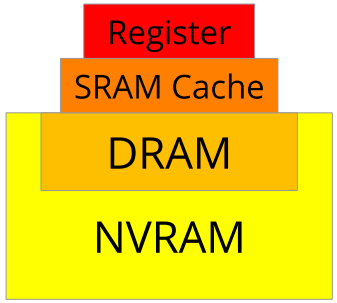
- NVRAM (Abk. für (en.) *non-volatile* RAM)
  - behält seine Information, auch ohne anliegende Betriebsspannung
    - SRAM mit Pufferbatterien (Selbstentladung) mehrere Jahre $\rightsquigarrow$ NVSRAM†
    - FRAM‡ (Abk. für (en.) *ferroelectric* RAM) 10^2 Jahre
    - andere spezielle Halbleitermaterialien 10^6 Jahre
  - mittlere Zugriffszeiten 15ns†/100ns‡ (Lesen) – 25ns†/150ns‡ (Schreiben)

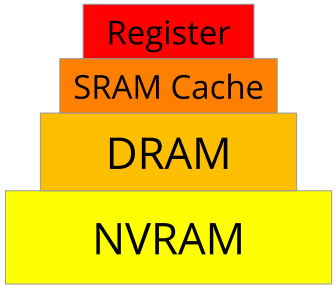
NVRAM in der Speicherhierarchie

Eingebettete Systeme

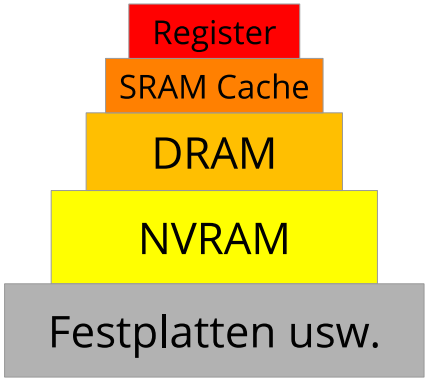


Smartphones/Tablets/Notebooks

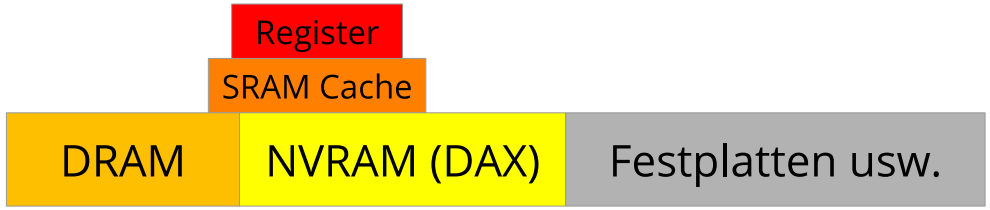




Workstations/Server



Spezialsysteme/Datenbankserver



Gliederung

Grundlegende Veränderung

Urladen	■ persistente Programme (NVRAM)
	■ Schadsoftware und Fehlzustände *loswerden*
Datenkonsistenz	■ globale Zwischenzustände (NVRAM, PIM)
	■ Inkonsistenzen *vorbeugen*, sofern möglich
Funktionszerlegung	■ logisch dedizierte Rechenkerne (PIM)
	■ Aufgabe innerhalb eines Komplexes *vereinzeln*
Spezialisierung	■ softwaregeführte Koprozessoren (PIM)
	■ Funktionen *erschaffen* und *zuordnen*
Integrität	■ widersprüchliche Systemzustände (NVRAM)
	■ Inkonsistenzen *erkennen* und *auflösen*
⋮	

Datenkonsistenz

Hinweis (Transaktionskonsistenz)

Entweder kommt die gesamte Transaktion vollständig zum Abschluss oder die sie definierende Aktionsfolge bleibt ohne Auswirkung, als wenn sie nie stattgefunden hätte.
Bedingt ein Systemabsturz/-ausfall unvollständige Transaktionen, sind diese beim Wiederanlauf zu erkennen und zurückzusetzen oder alle bereits erledigten Teile sind rückgängig zu machen.

- eine korrekte Transaktion (im RAM) ist gegen **Wettlaufsituationen** immun, allerdings (im NVRAM) nicht gegen **Stromausfall**
- sofern es eine Wahl gibt, ist **Vorbeugen von Fehlzuständen** besser als Heilen nach einer ohnehin schwierigen Diagnose
  - dauerhafte (*durable*) Datenstrukturen [4] für NVRAM
  - nebenläufige (*concurrent*) Datenstrukturen [10, 3] für PIM

Datentyp I

- Untersuchungsgegenstand sei eine grundlegende, wie folgt modellierte **dynamische Datenstruktur**
  - ein Bündel (*bunch*) von Elementen als **einfach verkettete Liste**:

```
1 typedef struct bunch {
2     chain_t head;         /* top of the list */
3 } bunch_t;
```

  - wobei ein einzelnes **Kettenglied** (*chain link*) folgende Struktur hat:

```
1 typedef struct chain {
2     struct chain *link; /* next list element */
3 } chain_t;
```

- **stapelbasierte** (*last in, first out*: LIFO) **Listenmanipulation** umfasst hier die Aktualisierung nur des Kopfzeigers (`head`)
- jedoch erfordert auch dann das Einfügen (*push*) oder Austragen (*pull*) des Kopfelements immer noch zwei zusammenhängende Schritte

Prinzipielle Funktionsweise I

- **Grundvoraussetzung**: Ein zu listendes Element ist noch nicht auf der Liste verzeichnet $\rightsquigarrow$ der Verknüpfungszeiger (`link`) ist frei

```
1  inline void push(bunch_t *this, chain_t *item) {
2      item->link = this->head.link;
3      this->head.link = item;
4  }
```

- 2 kopieren des Kopfzeigers auf das Element, das „gestapelt“ werden soll
- 3 aktualisieren des Kopfzeigers mit der Adresse dieses Elements

```
5  inline chain_t *pull(bunch_t *this) {
6      chain_t *node;
7      if ((node = this->head.link))
8          this->head.link = node->link;
9      return node;
10 }
```

- 7 merken des oben auf der Liste stehenden Elements, falls vorhanden
- 8 aktualisieren des Kopfzeigers mit der Adresse des nächsten Elements

Problemanalyse I

- Datenkonsistenz bei unvollständigen Transaktionen im nichtflüchtigen Hauptspeicher und nebenläufigen Aktionen durch Koprozessoren

  NVRAM
  - nur Einzelwortschreibzugriffe auf globale Daten ☺
  - ein geschriebenes Datum muss aber den RAM erreichen ☹

  ↪ Zwischenspeicher (*cache*) selektiv **Ausspülen** (*flush*)

  PIM
  - gleichzeitige Prozesse könnten Fehlzustände hinterlassen ☹

    `push` ‖ `push` zwei Elemente erhalten denselben Nachfolger
    `push` ‖ `pull` ein Element erhält einen ungültigen Nachfolger
    `pull` ‖ `push` ein eben eingehängtes Element geht verloren
    `pull` ‖ `pull` zwei Prozesse erhalten dasselbe Element
  - Synchronisation nötig, die beteiligten Rechenkerne haben aber kein gemeinsames Betriebssystem ☹

  ↪ Befehlssatzebene ↦ **nichtblockierende Synchronisation**

- die zur Konsistenzwahrung nötigen Aktionen sind prozessorabhängig und für die (System-) Software querschneidend

Datentyp II

- eine gemeine **Warteschlange**, jedoch mit einer technischen Feinheit:

```
1  typedef struct queue {
2      chain_t  head;                  /* first item */
3      chain_t *tail;                  /* insertion point */
4  } queue_t;
```

  - der Endzeiger (`tail`) adressiert das Bindeglied (`link`) des nächsten in die Warteschlange zu stellenden Elements, nicht das Element

- damit hat auch eine leere Warteschlange einen gültigen Endzeiger:

```
1  inline chain_t *zap(queue_t *this) {
2      chain_t *head = this->head.link;
3
4      this->head.link = 0;            /* null item */
5      this->tail = &this->head;       /* linkage item */
6
7      return head;
8  }
```

4 - der Kopfzeiger (`head`) ist die Attrappe (*dummy*) eines Listenelements

5 - den Endzeiger auf diese Elementattrappe verweisen lassen

Prinzipielle Funktionsweise II

- gleiche Voraussetzung wie bisher: ein einzustellendes Element steht noch nicht in der Warteschlange
  - Elemente in **Ankunftsreihenfolge** (*first-in*, *first-out*: FIFO) einstellen

```
1  inline void enq(queue_t *this, chain_t *item) {
2      item->link = 0;                /* finalise chain */
3      this->tail->link = item;       /* append item */
4      this->tail = item;             /* set insertion point */
5  }
```

  - der Kopfzeiger einer leeren Schlange zeigt implizit auf das erste Element

```
6  inline chain_t* deq(queue_t *this) {
7      chain_t *node;
8      if ((node = this->head.link)          /* filled? */
9      && !(this->head.link = node->link)) /* last item? */
10         this->tail = &this->head;         /* reset */
11     return node;
12 }
```

- der Endzeiger muss immer gültig sein, auch bei leerer Warteschlange

Problemanalyse II

- **unvollständige Transaktionen** im nichtflüchtigen Hauptspeicher als mögliche Zugabe mit unangenehmen Konsequenzen

  NVRAM
  - teilbare Mehrwortschreibzugriffe auf globale Daten ☹
  - jedes geschriebene globale Datum muss den RAM erreichen ☹

  ↪ **Transaktionskonsistenz** ist zu gewährleisten

  ↪ Zwischenspeicher (*cache*) selektiv **Ausspülen** (*flush*)

  PIM
  - gleichzeitige Prozesse könnten Fehlzustände hinterlassen ☹

    `enq` ‖ `enq` ein eben eingehängtes Element geht verloren & weitere Elemente werden ebenda angehängt

    `deq` ‖ `enq` ein eben eingehängtes Element geht verloren

    `deq` ‖ `deq` zwei Prozesse erhalten dasselbe Element
  - wie bereits zuvor muss die benötigte Synchronisation auf ein allen Rechenkernen gemeinsames Betriebssystem verzichten ☹

  ↪ Befehlssatzebene ↦ **nichtblockierende Synchronisation**

- **wiederherstellbare kritische Abschnitte** [6, 8] sind zwar bekannt, jedoch zwecklos, da sie ein Betriebssystem „benutzen“

Nebenläufige, dauerhafte Lösung?

```
1  void enq_lfs(queue_t *this, chain_t *item) {
2      chain_t *last, *hook;
3
4      item->link = item;
5
6      do hook = (last = this->tail)->link;
7      while (!CAS(&this->tail, last, item));
8
9      if (!CAS(&last->link, hook, item))
10         this->head.link = item;
11 }


12 chain_t* deq_lfs(queue_t *this) {
13     chain_t *node, *next;
14
15     do if ((node = this->head.link) == 0) return 0;
16     while (!CAS(&this->head.link, node,
17         ((next = node->link) == node ? 0 : next)));
18
19     if (next == node) {
20         if (!CAS(&node->link, next, 0))
21             this->head.link = node->link;
22         else CAS(&this->tail, node, &this->head);
23     }
24
25     return node;
26 }
```

- neuralgische Punkte:
  - 9–10 zusammenhängende Schreibvorgänge
  - 20–21 dito
    - ganz abgesehen von fehlenden Barrieren
    - auch das ABA-Problem wäre noch zu lösen
      ↪ CS
    - atomare RMW-Befehle allein reichen nicht
      - für NVRAM
- dann noch **Performanz**
  - vor allem CAS & Co.
  - geht für PIM
  - ungewiss für NVRAM
- zu guter Letzt die **Programmierbarkeit**: Sprache? Übersetzer? . . . ?

Gliederung

Lehren aus der Geschichte ziehen

- wie schon immer RMW-Befehle mit Bedacht nutzen, hier allerdings PIM und NVRAM unterschiedlich begegnen
  - für PIM gilt weitestgehend Bekanntes aus der Welt der Vielkerner
    - wiederholende Häufung von Bussperren (*bus-lock burst*) vorbeugen
    - ein Bewusstsein für das Problem von **Interferenz** entwickeln
  - bei NVRAM auf Eigenschaften des Zwischenspeichers setzen
    - wenn Zwischenspeicherzeilen lesen/schreiben atomar geschieht
    - dann Daten mittels ***cache-line transaction*** [13], CLT, verdauern
- viel größer ist jedoch die Herausforderung, Datenstrukturen für CLT aufzubereiten, um sie dauerhaft auslegen zu können
  - jegliche Form von Wartelisten im Betriebssystem, inklusive Bereitliste
  - jegliche Form von Deskriptoren oder Verbünden, . . .

  ↪ dies alles krempelt bestehende (System-) Software komplett um
- so stellt sich auch die Frage, was von alledem von einem **Übersetzer** übernommen werden könnte oder sollte
  - denkbar wäre eine **domänenspezifische Programmiersprache**
  - aber Abstraktion schützt nicht vor Änderung: **Neugestaltung** ist nötig

„Luja sog i“...

Literaturverzeichnis I

[1] ACM (Veranst.):
Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC'16).
New York, NY, USA : ACM, 2016

[2] Backus, J. :
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
In: *Communications of the ACM* 21 (1978), Aug., Nr. 8, S. 613–641

[3] Ellen, F. ; Brown, T. :
Concurrent Data Structures.
In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC'16)*[1], S. 151–153

[4] Friedman, M. ; Herlihy, M. ; Marathe, V. ; Petrank, E. :
A Persistent Lock-Free Queue for Non-Volatile Memory.
In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*.
New York, NY, USA : ACM, 2018, S. 28–40

Literaturverzeichnis II

[5] Ghose, S. ; Hsieh, K. ; Boroumand, A. ; Ausavarungnirun, R. ; Mutlu, O. : *Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions*.
https://arxiv.org/pdf/1802.00320.pdf, Febr. 2018

[6] Golab, W. ; Ramaraju, A. :
Recoverable Mutual Exclusion.
In: *P*roceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC'16)[1], S. 65–74

[7] Hennessy, J. L. ; Patterson, D. A.:
Computer Architecture: A Quantitative Approach.
Vierte Auflage.
Elsevier, Inc., 2007

[8] Jayanti, P. ; Joshi, A. :
Recoverable FCFS Mutual Exclusion with Wait-Free Recovery.
In: *Proceedings of the 31st International Symposium on Distributed Computing (DISC 2017)*.
Schloss Dagstuhl, Germany : Leibniz International Proceedings in Informatics, 2017. –
ISBN 978–3–95977–053–8, S. 30:1–30:15

Literaturverzeichnis III

[9] Liu, Z. ; Calciu, I. ; Herlihy, M. ; Mutlu, O. :
Concurrent Data Structures for Near-Memory Computing.
In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '17)*.
New York, NY, USA : ACM, 2017, S. 235–245

[10] Moir, M. ; Shavit, N. :
Concurrent Data Structures.
In: Mehta, D. P. (Hrsg.) ; Sahni, S. (Hrsg.): *Handbook of Data Structures and Applications*.
Chapman & Hall/CRC, 2005 (Chapman & Hall/CRC Computer and Information Science Series). –
ISBN 1–58488–435–5, Kapitel 47, S. 47:1–47:30

[11] Mutlu, O. ; Ghose, S. ; Gómez-Luna, J. ; Ausavarungnirun, R. :
Enabling Practical Processing in and near Memory for Data-Intensive Computing.
In: *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*.
New York, NY, USA : ACM, 2019

[12] Schröder-Preikschat, W. :
Sachwortverzeichnis zur Systemprogrammierung.
`https://www4.cs.fau.de/~wosch/glossar.pdf`, 2018 *ff.*

Literaturverzeichnis IV

[13] Traue, J. :
Fine-Grained Transactions for NVRAM, Brandenburgische Technische Universität Cottbus-Senftenberg, Fakultät 1, Diss., 2018