Eingebettete Systeme

S. Ryffel

14. Oktober 2006

Inhaltsverzeichnis

1 Einführung

Definition

Embedded Systems sind informationsverarbeitende Systeme, die in ein grösseres Produkt eingebettet sind.

Eigenschaften von eingebetteten Systemen

- *Zuverlässig*:
  - **Reliability** $R(t)$ ist die Whkeit, dass System noch korrekt funktioniert wenn es dies bei $t = 0$ tat.
  - **Maintainability** $M(d)$ ist die Whkeit, dass System bei Ausfall nach d wieder korrekt funktioniert.
  - **Availability** ist die Whkeit, dass System zur Zeit t funktioniert.
  - **Safety** System richtet keinen Schaden an
  - **Security** bedeutet verlässliche und authentische Kommunikation
- *Effizient*:
  - **Energie** effizient
  - **Code-size** effizient
  - **Run-time** effizient
  - **Weight** effizient
  - **Cost** effizient
- Zugeschnitten und optimiert auf ganz bestimmte Anwendung und Benutzerschnittstelle.
- *Echtzeit-Anforderungen*

2 Einführung in Software Design

2.1 Ausführungsdauer von Tasks

Worst Case Execution Time (WCET) ist die obere Schranke für die Ausführungszeit eines Tasks.
Kann im allgemeinen nicht berechnet werden, da Architektur zu wenig deterministisch: Piplines, Caches, Interrupts, VM, etc.

Durchschnittliche Ausführungszeiten

Simulation: Schwierig Systemzustand und -Umgebung zu simulieren.
Emulation: Hardwareunterstützte Simulation
Profiling: Messung der Ausführungszeit
Im allgemeinen kann die WCET nicht beobachtet werden.

2.2 Multitasking

2.2.1 Co-Routines

Durch Funktionsaufrufe kann ein Task die Kontrolle freiwillig an eine Co-Routine abgeben.

2.2.2 Zeitgesteuerte Systeme

- Interrupt nur vom Timer
  → Interaktion mit Umgebung durch Polling
- Scheduler als offline Algorithmus → deterministisch, zuverlässig

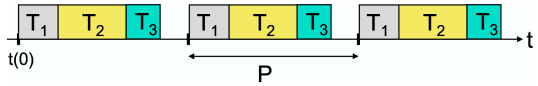


Abbildung 1: Einfacher periodischer TT Scheduler

Einfacher periodischer TT Scheduler (vgl Abb. 1)

- Gleiche Periode t für alle Prozesse
- Keine Probleme mit IPC und Ressourcen
- $$\sum_k \mathrm{WCET}(T_k) < P$$

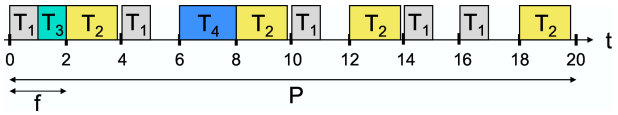


Abbildung 2: Zyklisch ausführender TT Scheduler

Zyklisch ausführender TT Scheduler (vgl Abb. 2)

- Tasks mit verschiedenen Perioden
- Problem: Lange Prozesse ($> f$) müssen in kleinere Teile gespalten werden!
- *Notwendige Bedingungen an* f für Tasks mit Perioden $p(k)$ und Deadlines $D(k)$

  a)
  $$f \leq \min\{p(k)\}$$
  Einhaltung der Deadline kann nur überprüft werden, wenn Task höchstens ein mal pro Frame vorkommt.

  b)
  $$\exists\, k \quad p(k) \mod f = 0$$

  c)
  $$f \geq \max\{\mathrm{WCET}(k)\}$$
  Jeder Tasks muss in ein Frame passen.

  d)
  $$2f - \mathrm{ggT}(p(k), f) \leq D(k) \quad \forall k$$
  Tasks sollten vor ihrer Deadline enden können.
- *Hinreichende Bedingung*: Zeichnen eines Ablaufs.

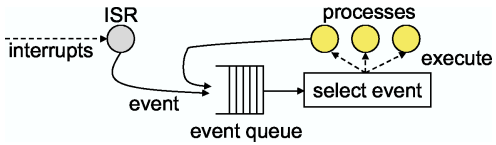


Abbildung 3: Ereignisgesteuerte Systeme

2.2.3 Ereignisgesteuerte Systeme

Prinzip (vgl Abb. 3)

- Jeder Event besteht aus einem auszuführenden Task.
- Sowohl Prozesse als auch Interrupts können Events generieren.
- Events werden in einer Queue gespeichert.
- *preemptive*: Laufende Tasks können unterbrochen werden.
  - Gemeinsame Ressourcen müssen geschützt werden.
  - Stack-based context Mechanismus
- *non-preemptive*:
  - IPC, gemeinsame Ressourcen kein Problem
  - Buffer overflow bei zu vielen Events
  - Event-queue ist gemeinsame Ressource

Definitionen

Prozess ist eine einzigartige Ausführung eines Programmes.
Ein Prozess hat einen Zustand: register values, memory stack.
Activation record ist eine Kopie eines Prozesszustandes im Speicher.
Context switch ist ein Wechsel des aktiven Prozesseses in der CPU.

Co-operative Multitasking

Prinzip: Jeder Prozess kann einen Contetxt switch mit `cswitch()` erlauben.
Vorteile:

- Context switchs vorhersehbar.
- Weniger Fehler mit gemeinsamen Ressourcen

Nachteile:

- Es ist möglich, dass ein Task die CPU nicht mehr freigibt.
- Echtzeitverhalten gefährdet, wenn Task CPU lange nicht freigibt.

Typisches Programmierinterface: NutOS Für ein Beispiel von co-operativem Multitasking siehe im Skript S. 2-40.

3 Echtzeit-Modelle

3.1 Definitionen

3.1.1 Echtzeit-Systeme

Hard Überschreitung einer Deadline führt zu einer Katastrophe.
Soft Deadline wird meistens eingehalten.

3.1.2 Schedule

gegeben die Tasks $J = \{J_1, J_2, \dots\}$ (vgl Abb. 4)

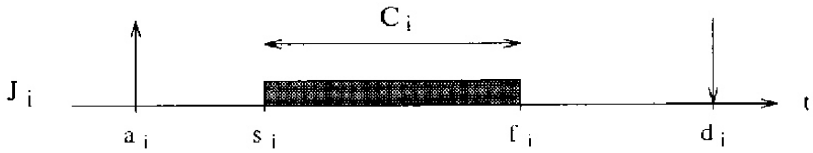


Abbildung 4: Zeiten eines Tasks

Schedule $\sigma : \mathbb{R} \to \mathbb{N}$ ist eine Zuordnung von Tasks zum Prozessor, so dass jeder Task abgearbeitet wird.
$\sigma(t) = n$: zum Zeitpunkt t wird n abgearbeitet.
Time Slice ist das Intervall, während dem $\sigma(t)$ konstant ist.
Feasible / Zulässig ist ein Schedule, wenn alle Tasks gemäss den Anforderungen ausgeführt werden.
Schedulable / Ablauffähig ist eine Menge von Tasks, wenn ein zulässiger Schedule existiert.
Arrival Time a_i **/ Release Time** r_i ist die Zeit zu welcher Task bereit zur Ausführung ist.
Computation Time C_i ist die Zeit, die der Prozess die CPU braucht.
Deadline d_i: absolute Deadline; D_i: relative Deadline; $d_i = D_i + a_i$
Start Time s_i Task startet Ausführung.
Finishing Time f_i Task ist fertig
Lateness $L_i = f_i - d_i$; $L_i \leq 0 \to$ Task hat Deadline eingehalten
Tardiness / Execution Time $E_i = \max\{0, L_i\}$
Laxity / Slack Time $X_i = d_i - a_i - C_i$ maximale Zeit um die ein Task noch verzögert werden kann.

3.2 Ablauf-Bedingungen

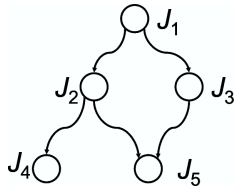


Abbildung 5: Ablauf Bedingungen

Nach Ablauf eines Tasks werden alle durch eine Kante verbundenen Tasks ausgeführt. (Precedence Constraints, vgl Abb. 5)

3.3 Klassifizierung von Scheduling Algorithmen

3.3.1 Klassen

- Preemptive Algorithmen
- Non-preemtive Algorithmen
- Statische Algorithmen
- Dynamische Algorithmen
- Optimale Algorithmen haben minimale Kosten
- Heuristische Algorithmen finden meistens aber nicht immer den optimalen Ablauf.

3.3.2 Metriken

Average Response Time

$$\overline{t_r} = \frac{1}{n}\sum_{i=1}^{n}(f_i - a_i)$$

Total Completion Time

$$t_c = \max_i(f_i) - \min_i(a_i)$$

Average Waiting Time

$$\overline{t_w} = \frac{1}{n}\sum_{i=1}^{n} s_i - a_i$$

Maximum Lateness

$$L_{\max} = \max_i(f_i - d_i)$$

Maximum Number of Late Tasks

$$N_{\text{late}} = \sum_{i=1}^{n} \text{miss}(f_i)$$

4 Aperiodische und Periodische Tasks

4.1 Aperiodische Tasks mit Echtzeitanforderungen

Tabelle bekannter Algorithmen für Scheduling von aperiodischen Tasks.

	Equal arrival times non preemptive	Arbitrary arrival times preemtive
Independent tasks	EDD (Jackson)	EDF (Horn)
Dependent tasks	LDF (Lawler)	EDF* (Chetto)
	LDD	

4.1.1 Earliest Deadline Due (EDD)

Jackon's rule: Gegeben sind n Tasks. Wenn man jeweils den Task mit der frühesten Deadline abarbeitet, minimiert man die Maximum Lateness. (*non preemtive*)

4.1.2 Earliest Deadline First (EDF)

Horns's rule: Gegeben sind n unabhängige Task mit beliebigen Ankunftszeiten. Ein Algorithmus, der *preemtive* am Task mit der frühesten Deadline arbeitet, minimiert die Maximum Lateness.

Garantie
Dieser Algorithmus prüft, ob man J_{new} noch zu den Tasks J hinzufügen kann, so dass diese ablauffähig bleiben.

$c_k(t)$	verbleibende WCET vom Task k
$f_i = \sum_{k=1}^{i} c_k(t)$	worst case finishing time of Task i
$\sum_{k=1}^{n} c_k(t) \leq d_i \quad \forall\, i$	EDF Garantie-Bedingung

Alle Tasks müssen auf jeden Fall vor ihrer Deadline ausgeführt werden können.

4.1.3 Latest Deadline Due (LDD) or Latest Deadline First (LDF)

LDD nimmt eine Menge von Tasks mit Ablaufbedingungen und generiert daraus ein *non-preemtive* Schedule:

1. Starte bei den Blättern des Ablaufbaumes.
2. Wähle die Menge der Tasks ohne oder mit geschedulten Nachfolgern.
3. Schedule denjenigen mit der spätesten Deadline zuletzt.
4. Wiederhole 2., bis alle Tasks gescheduled.

4.1.4 Earliest Deadline First (EDF*)

Der EDF* Algorithmus findet einen zulässigen Schedule für Tasks mit Ablaufbedingungen und gleichzeitiger Aktivierung.

Wenn ein zulässiger Schedule existiert, dann ist garantiert dass

- Tasks einander nicht unterbrechen ($f_i < s_{i+1}$)
- Tasks ihre Deadline einhalten

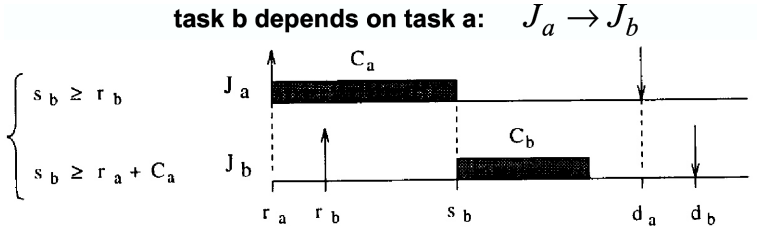


Abbildung 6: Modifizierung der Release Times

Modifizierung der Release Times

Ziel: Tasks dürfen nicht vor ihrer Release Time oder bevor ihre Vorgänger fertig sind starten.

$$r_j^* = \max\left(r_j,\ \max(r_i^* + C_i : J_i \to J_j)\right)$$

Algorithmus: von vorne nach hinten

1. Für den ersten Knoten: $r_i^* = r_i$
2. Wähle ein Task j dessen Vorgänger alle schon bearbeitet sind.
3. Set $r_j^* = \max\left(r_j,\ \max(r_i^* + C_i : J_i \to J_j)\right)$
4. Gehe zu 2.

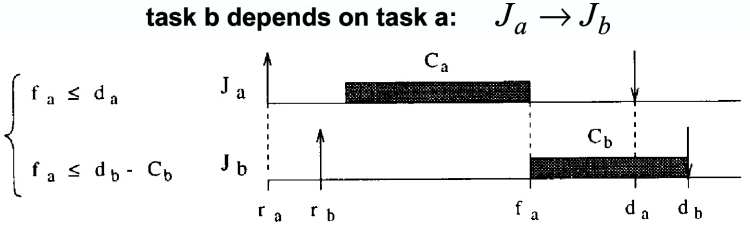


Abbildung 7: Modifizierung der Deadlines

Modifizierung der Deadlines

Ziel: Task muss vor der Deadline und vor der spätesten Startzeit der Nachfolger fertig sein.

$$d_i^* = \min\left(d_i,\ \min(d_j^* - C_j : J_i \to J_j)\right)$$

Algorithmus: von hinten nach vorne

1. Wähle ein Blatt des Baumes und setze: $d_j^* = d_j$.
2. Wähle ein Task i dessen Nachfolger schon bearbeitet sind.
3. Set $d_i^* = \min\left(d_i,\ \min(d_j^* - C_j : J_i \to J_j)\right)$
4. Gehe zu 2.

4.1.5 Diverse Algorithmen

FCFS First Come First Served:
Führe in der Reihenfolge aus, in der die Jobs ankommen.

SJF Shortest Job First:
Führe den kürzesten Job aus, non-preemtpive.

SRTN Shortest Remaining Time Next:
Führe preemtive den Job aus, der am wenigsten Zeit zum Ende braucht.

RR Round Robin:
Gib jedem Job reihum ein Timeslice zur Ausführung.

4.2 Preemptive Scheduling Algorithmen für periodische Tasks

	Deadline equals period	Deadline smaller than period
static priority	RM (rate-monotonic)	DM (deadline-monotonic)
dynamic priority	EDF	EDF*

4.2.1 Modell der periodischen Tasks

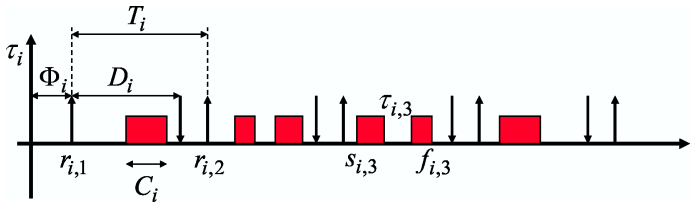


Abbildung 8: Modell der periodischen Tasks

Definitionen siehe Abbildung 8.

Γ Menge von periodischen Tasks
$\tau_{i,j}$ Instanz j des Tasks i
$r_{i,j}$, $s_{i,j}$, $f_{i,j}$, $d_{i,j}$, C_i dito, zB $r_{i,j} = \Phi_i + (j-1)T_i$
Φ_i relative Verschiebung, Phase, $\Phi_i = r_{i,1}$
T_i Periode des Tasks i
D_i relative Deadline

Hypothesen

- Regelmässige Aktivierung: $r_{i,j} = \Phi_i + (j-1)T_i$
- Gleiche relative Deadline → absolute Deadline: $d_{i,j} = r_{i,j} + D_i$

4.2.2 Rate Monotonic Scheduling (RM)

Annahmen

a) Statische Prioritäten
b) Preemtive
c) $D_i = T_i$

Algorithmus: Tasks mit höheren Raten unterbrechen solche mit tieferen.
RM ist *optimal*.

Schedulability Analyse: Hinreichende aber nicht notwendige Bedingung

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

Für $n \to \infty$ sind Tasks schedulable, wenn CPU-Auslastung $\leq 70\%$ ist.

4.2.3 Deadline Monotonic Scheduling (DM)

Annahmen: Gleich wie bei RM aber $C_i \leq D_i \leq T_i$.

Algorithmus: Tasks mit kleinerer (nicht früherer!) Deadline unterbrechen solche mit höheren.

Schedulability analysis: 2 Methoden

a) Hinreichende aber nicht notwendige Bedingung:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{1/n} - 1\right)$$

b) Hinreichende und notwendige Bedingung:

$$\begin{aligned} R_i &\leq D_i \qquad \forall i \\ R_i &= C_i + I_i \qquad \text{longest response time} \\ &= C_i + \underbrace{\sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j}_{I_i} \end{aligned}$$

- Kritischer Augenblick ist, wenn alle Tasks zur gleichen Zeit instaziiert werden.
- *Worst Case Interference* I_i: maximale Zeit, um die ein Task i auf Grund der höher priorisierten ($j < i$) verzögert werden kann.
- *Berechnung* von R_i kann iterativ erfolgen:
  1. $R_i = C_i$
  2. Berechne I_i für dieses R_i
  3. Wenn I_i geändert $\Rightarrow R_i = C_i + I_i$ und gehe zu 2.

4.2.4 EDF Scheduling

Annahmen: Dynamische Prioritäten, preemtive, $D_i \leq T_i$

Algorithmus: Der laufende Task wird unterbrochen, wenn ein Task mit früherer Deadline kommt.

Schedulability analysis:

- Hinreichend und notwendiger Test für $D_i = T_i$:

$$U = \sum_{i=1} \frac{C_i}{T_i} \leq 1 \qquad \text{„CPU-Auslastung kleiner als 100\%“}$$

- Hinreichender Test für $D_i \leq T_i$:

$$\sum_{i=1} \frac{C_i}{D_i} \leq 1$$

4.3 Scheduling von periodischen und aperiodischen Tasks

Periodische Tasks: Zeitgesteuert, harte Echtzeitanforderungen
Aperiodische Tasks: Ereignisgesteuert, harte, weich oder ohne Echtzeitanforderungen
Sporadische Tasks: Aperiodische Tasks welche durch eine maximale Ankunftsrate charakterisiert sind.

4.3.1 Background Scheduling

Aperiodische Tasks werden nur ausgeführt, wenn keine periodischen vorhanden sind.

4.3.2 RM-Polling Server

Ein künstlicher periodischer *„polling server“* Task (mit C_s, T_s) führt regelmässig aperiodische Tasks aus.

- *Vorteile*:
  - Nur eine Queue / Scheduling Algorithmus
  - Kein Starving aperiodischer Tasks
  - Wenn keien aperiodischen Tasks vorhanden sind, kann Zeit für periodische genutzt werden.
- *Nachteile*:
  - Aperiodische Task müssen vor der Periode des polling server eintreffen.
    Denn wenn sie ankommen, nachdem er suspended hat, müssen sie warten, obwohl der Prozessor eventuell frei währe!

Schedulability analysis: Hinreichender aber nicht notwendiger Test:

$$\frac{C_s}{T_s} + \sum_{i=1}^{n} \frac{C_i}{T_i} \leq (n+1)\left(2^{1/(n+1)} - 1\right)$$

Aperiodic Guarantee: Hinreichende Bedingung für aperiodischen Task (C_a, D_a).

$$\left(1 + \left\lceil \frac{C_a}{C_s} \right\rceil\right) T_s \leq D_a$$

unter der Annahme, dass jeweils nur ein aperiodischer Task existiert.

4.3.3 EDF - Total Bandwidth Server

Total Bandwidth Server: Der k-te aperiodische Request (C_k, D_k) zur Zeit $t = r_k$ bekommt die Deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

wobei U_s die Bandbreite des Servers oder die Auslastung des Servers ist. $d_0 = 0$

Schedulability Test: Mit n periodischen Tasks mit der Prozessor Auslastung U_p und einem total bandwidth server mit Auslastung U_s

$$U_p + U_s \leq 1$$

5 Ressource Sharing

5.1 Definitionen

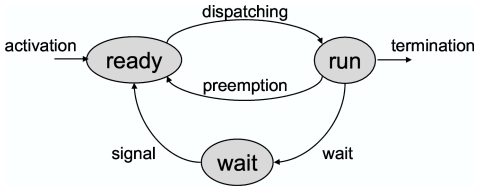


Abbildung 9: Zustände eines Tasks

Exclusive Ressource R_i benötigt mutual exclusion
Critical Section Codeabschnitt, welches eine kritische Ressource benützt.
Blocked Task wartet auf die Freigabe einer Ressource
Semaphore S_i ein kritischer Codeabschnitt wird mit `wait(`S_i`)` und `signal(`S_i`)` geschützt.

5.2 Priority Inversion

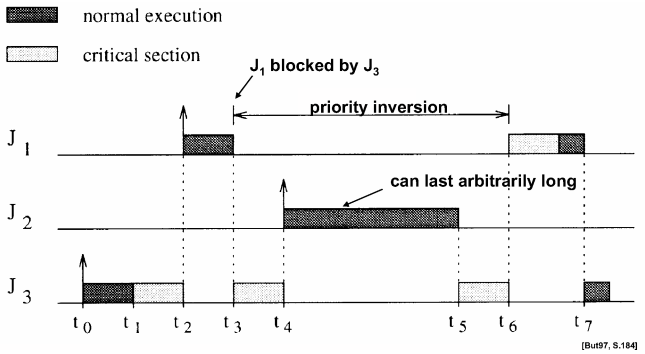


Abbildung 10: Priority Inversion

Wenn ein hoch priorisierter Task mit `wait(`S_i`)` blockiert, weil er auf `signal(`S_i`)` eines niedrig priorisierten wartet. Der niedrig priorisierte bekommt aber keine Rechenzeit, da mittler priorisierte zuerst ausgeführt werden. Siehe Abbildung 10.

5.2.1 Lösung I: Verbiete Preemption

Während der Ausführung von kritischen Codestücken. Einfach, doch unbeteiligte Tasks werden auch geblockt.

5.2.2 Lösung II: Priority Inheritance Protocol (PIP)

Idee Setze die Priorität von Tasks in kritischen Codeabschnitten auf die Stufe des höchstpriorisierten der blockierten Tasks.

Definitionen
Nominal Priority P_i fixe Priorität
Active Priority p_i ist grösser oder gleich P_i
Direct Blocking Hoch priorisierter Task will Ressource eines niedriger priorisierten Tasks.
Push-through Blocking Mittler priorisierter Task wartet auf einen niedrig priorisierten, welcher Priorität von hoch priorisiertem geerbt hat.

Algorithmus
- Jobs werden nach ihrer active priority p_i gescheduled.
- `wait(`S_i`)`: Warte, bis Ressource S_i freigegeben wird.
  Übergebe eigene Priorität an den Task, der den Semaphore hat.
- `signal(`S_i`)`: Der höchstpriorisierte der wartenden Tasks wird aktiviert.
  Wenn keine Tasks mehr warten, wird $p_i = P_i$, sonst erbe neue höchste Prioriät.
- Priority Inheritance ist *transitiv*: wenn 1 auf 2 und 2 auf drei wartet, erbt 3 die Priorität von 1 über 2.

Deadlock System ist blockiert. Zum Beispiel wollen zwei kritische Codeabschnitte die Ressource des anderen.

6 Real-Time Operating Systems

6.1 Embedded OS

Wieso kein Desktop-OS?
- Zu viele Features
- Nicht fehler-tolerant, konfigurierbar
- Zu gross
- Nicht power optimized

Konfigurierbarkeit Keinn OS erfüllt ohne Overhead alle Anforderungen.
Anpassung über: entfernen von Funktionen, conditional compilation ...

Device Drivers als Prozesse anstatt ins OS integriert
- Erhöht Vorhersehbarkeit, alles geht durch den Scheduler
- Kein Gerät muss von allen OS-Versionen unterstützt werden.

Interrupts can be employed by any process. Interrupts können direkt Tasks starten oder stoppen. Ist effizienter als durch die OS-Services.

6.2 Real-time Operating Systems

Definition Ein Real-time Operating System ist ein OS, welches die Konstruktion von Echtzeitsystemen unterstützt.

6.2.1 Anforderungen

a) Das Zeitverhalten muss vorhersehbar sein.
- Obere Schranke für Ausführungszeiten
- Interrupts nur während kurzen Zeiten deaktiviert
- Fast alles durch Scheduler kontrolliert

b) OS muss das Timing und Scheduling verwalten.
- OS muss Deadlines kennen.
- Hohe Zeitauflösung

c) OS muss schnell sein.
- In Praxis nützlich

6.2.2 Hauptfunktion: Process Management

Ausführen von qausi-parallelen Tasks
- Verwaltung von Prozess-Zuständen:
  run: wird ausgeführt
  ready: könnte ausgeführt werden
  wait: wartet auf Event: Semaphore, Signal
  idle: periodischer Task wartet auf nächste Ausführung
- Context switching
- Interrupt handling
- *Threads*: Programmteil mit eigenem Context (Thread State).
  - shared memory: Threads eines Tasks teilen einen Teil ihres Zustandes
  - Haben eigenen Register und Stack
  - Thread Control Block: Speichert Informationen fürs Management und Scheduling des Threads

Scheduling Garantie von Deadlines, Minimierung von Wartezeiten, Fairness

Prozess Synchronisation Semaphores, Monitors, Mutex

Inter Process Communication (IPC)
- Synchrone Kommunikation: Prozesse warten
- Asynchrone Kommunikation: OS verwaltet eine Queue von Messages

Real-time Clock interne Zeitreferenz

6.2.3 Klassen von Echtzeitsystemen

a) Schnelle proprietäre Kernel:
Designt für Schnelligkeit, nicht Vorhersehbarkeit

b) Zusätze zu Standard-OS:
RT-Kernel führt RT-Tasks und Betriebssystem aus

c) Forschungs-Systeme:

7 System Komponenten

7.1 Spezialisierung

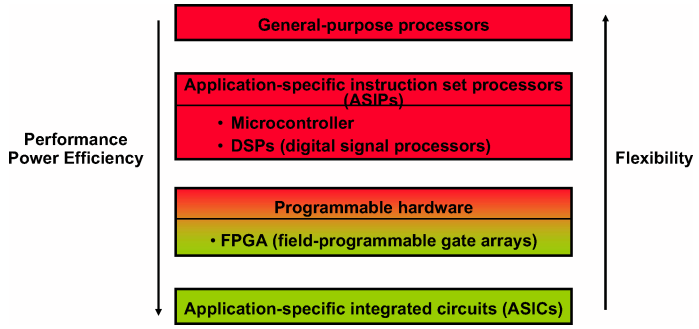


Abbildung 11: Performance versus Flexibility

(Siehe Abbildung 11)

General-purpose Processors
- High Performance: hoch optimiert, superscalar, komplexe Memoryverwaltung, pipelining
- Nicht geeignet für Echtzeit: Ausführungszeiten sind unberechenbar.
- Hoher Energieverbrauch

Spezialisierung
- Flexibilität erhalten: Unterstützung für ganze Applikations-Klasse, Platz für späte Änderungen, Debugging
- Analyse wichtig

Beispiele
- Codegrösse Effizienz: Komprimierung des ROM-Inhalts
- Multimedia-Instruktionen: spezielle Funktionen
- Mehrere Memory-Bänke: paralleles Laden
- Address Generation Units: Eigenen Adder für Addressen
- Modulo addressing

7.2 Applikationsspezifische Instruktions-Sets

Micro Controller
- Control-dominant Applications
  ereignisgesteuert, Finite State Machine
- Scheduling, synchronisation, interrupts, context switch, timers
- Niedriger Energieverbrauch
- Geeignet für Echtzeit

Digital Signal Processors
- Flussgesteuerte Systeme, datengesteuert
- Parallell hardware units, spezielle Instruktions-Sets
- Hoher Durchsatz, breiter Memory-Bus
- Geeignet für Echtzeit

Very Long Instruction Word (VLIW) Parallelisierung mehrerer Instruktionen in ein parallell ausführbares Word (Instruktionsset) durch den Compiler.

7.3 Programmierbare Hardware

7.3.1 FPGA

Klassifikation
- Granularität: Gate, Tabelle, Memory, funktionale Blöcke (ALU, data path, CPUs)
- Kommunikation: Crossbar, Mesh, Tree
- Programmierung: zur Produktionszeit, dynamisch zur Laufzeit

7.4 Application Specific Circuits (ASIC)

Vorteile
- Ultimative Geschwindigkeit
- Ultimative Energieeffizienz

Nachteile
- Aufwendiges Design, Entwurf
- Nicht flexibel
- Hohe Kosten, grosse Serie nötig

7.5 System-on-Chip

...

8 Kommunikation

8.1 Anforderungen

- Echtzeit
- Effizient, ökonomisch
- Geeignete Bandbreite, Latenz
- Robustheit, Fehlertoleranz
- Verwaltung, Diagnose
- Security, Safety

8.1.1 Echtzeit

- CSMA/CD, Ethernet nicht geeignet,
  keine garantierte Antwortzeit
- Alternativen:
  Tocken rings, busses
  CSMA/CA
  Time Triggered Protocol (TTP)

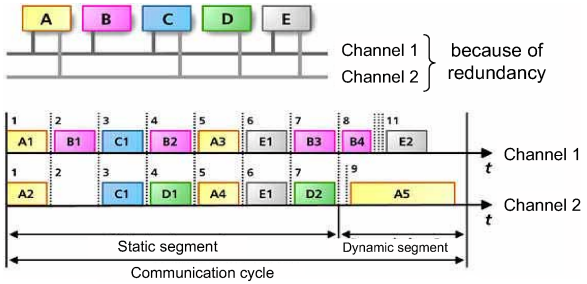


Abbildung 12: FlexRay

FlexRay (Siehe Abbildung 12)
- Jeder Zyklus ist ein ein statisches, kollisionsfreies und ein dynamische (token ring) Segment unterteilt.
- Echtzeit
- $\approx$ 10Mbit/sec

8.2 Bluetooth

8.2.1 Technische Daten

- 2.4 GHz Band, 79 MHz Bandbreite
- Frequency-Hopping:
  - 1600 hops/s (0.625ms), jedes Packet auf anderer Frequenz
  - Frequenzen: $(2402 + k)$ MHz, $k = 0 \ldots 78$
  - Sequenz wird durch ID des Masters bestimmt
- 10-100m Reichweite
- 1Mbit/s Bandbreite für jede Verbindung
- Übertragung:
  Synchronous Connection-Oriented (SCO)
  Asynchronous Connection-Less (ACL)

8.2.2 Topologien

Ad-hoc Netzwerke potentiell mobiler Knoten.

Piconet
- 1 Master und max 7 Slaves
- Benutzen selbes Frequency Hopping Muster
- Broadcast, unicast
- Verbindungsartgen:
  432kBit/s (duplex) oder 721/56 kBit/s (asymmetrisch) oder
  3 Audiokanäle oder
  eine Kombination

Scatternet
- Mehrere Piconets mit gemeinsamen Knoten
- Knoten können gleichzeitig Slave in mehreren Piconets und Master in maximal einem sein.

8.2.3 Protokoll Hierarchie

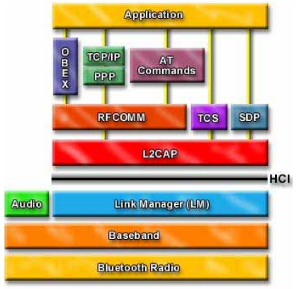


Abbildung 13: Bluetooth Protokoll Hierarchie

(Siehe Abbildung 13)
1. *Radio Specification*: definiert Frequenzbänder
2. *Baseband*: Packetformat, physikalischer Layer, Link Layer
3. *Audio Specification*: coding, decoding
4. *Link Manager* : Authentifikation, Management des Piconets, Verbindungen
5. *Host Controller Interface (HCI)*: definiert Schnittstellen zum Bluetooth-Modul
6. *Link Layer Control and Adaptation Layer (L2CAP)*: abstrakte Schnittstelle für Daten Kommunikation, Segmentierung, Multiplexing, QoS
7. *RFCOMM*: simuliert serielle Schnittstelle
8. *other protocols*: Telephony control protocol specification (TCS), service discovery protocol (SDP), object exchange protocol (OBEX)

8.2.4 Adressierung

BD_ADDR Bluetooth Device Address 48 Bit, eindeutig für jedes Gerät

AM_ADDR Active Member Address 3 Bit, 0 ist Broadcast
PM_ADDR Parked Member Address 8 Bit für geparkte Slaves

8.2.5 Packet Format

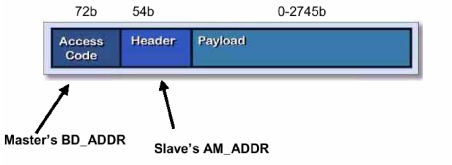


Abbildung 14: Bluetooth Packet Format

(siehe Abbildung 14.)

8.2.6 Verbindungstypen

Synchronous Connection-Oriented (SCO) peroidischer Service
- Symmetrischer synchroner Service
- Reservierte Slots für regelmässige Übertragung
- Der Master fordert den Slave regelmässig zum Senden auf.

Asynchronous Connection-Less (ACL) aperiodischer Service
- Keine Reservation von Slots
- Der Master überträgt spontan, der Slave antwortet im nächsten Intervall.

8.2.7 Moden und Zustände

Moden
Connection: Verbindung zwischen Master und Slave steht.
Page: Master versucht Verbindung mit BD_ADDR aufzunehmen.
Inquiry: Master identifiziert Nachbarknoten.

Zustände der Verbindung
active: mit Master verbunden
hold: verarbeitet keine Packete
sniff: erwacht regelmässig
park: keine Verbindung zum Master, synchronisiert

Der Page Modus
1. *page*: Der Master sendet seine Adresse an den Slave. Er benutzt eine spezielle Kanalsequenz (Synchronisation!).
2. *page scan*: Der Slave wartet auf Adresse.
3. *master page response*: Der Slave antwortet mit seiner eigenen Adresse.
4. *slave page response*: Der Master sendet ein Frequency Hop Synchronization (FHS) Packet. Dies enthält Kanalsequenz und Phase.

9 Energiesparendes Design

„Leistung gilt als die wichtigste Einschränkung von eingebetteten Systemen."

9.1 Leistung und Energie

$$\mathbf{E} = \int \mathbf{P}\,\mathbf{dt}$$

Arten von Energieverbrauch:
Dynamisch: Energeiverbrauch beim Schalten
Statisch: Leckströme, von Strukturgrössen abhängig

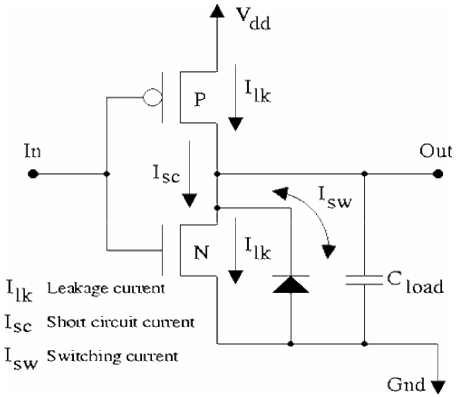


Abbildung 15: Gate

Leistungsverbrauch eines Gates (Siehe Abbildung 15)

$$P = \alpha \cdot C_L \cdot V_{dd}^2 \cdot f$$
$$E = \alpha \cdot C_l \cdot V_{dd}^2 \cdot \underbrace{f \cdot t}_{\text{Cycles}}$$

α: Switching Aktivität, Schaltungen pro Takt
C_L: Ladungskapazität
V_{dd}: Versorgungsspannung
V_t: Threshhold Spannung
f: Frequenz

Verzögerung von CMOS-Schaltungen

$$\tau = k \cdot C_L \cdot \frac{V_{dd}}{(V_{dd} - V_t)^2} \quad \approx \quad \frac{k \cdot C_L}{V_{dd}}$$
$$f_{max2} = f_{max1} \cdot \frac{V_{dd2}}{V_{dd1}}$$

9.2 Grundlegende Techniken

Parallelisierung Zwei paralelle Einheiten mit halber Versorgungsspannung und halber Frequenz brauchen zusammen nur halb so viel Energie.

VLIW Als simple Hardware Architektur, siehe Kapitel 7.2.

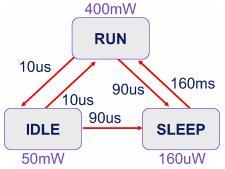


Abbildung 16: Dynamic Power Management

Dynamic Power Management (Siehe Abbildung 16)
RUN lauffähig
IDLE CPU angehalten, warten auf Interrupts
SLEEP Abschaltung aller Chip-Aktivität

Dynamic Voltage Scaling
- Anpassung der Versorgungsspannung und Frequenz an die Last.
- Verschiedene Module mit verschiedenen Versorgungsspannungen / Frequenzen.

10 Architektur Design - Modelle

Definitionen
Formales Modell Ein Modell repräsentiert bestimmte Aspekte des modellierte Systems (Abstraktion). Ein formales Modell repräsentiert diese Aspekte in eindeutiger Form.
Nebenläufigkeit bezeichnet das Verhältnis von Ereignissen, die nicht kausal abhängig sind

10.1 Task Graph oder Dependence Graph (DG)

Definitionen
Dependence Graph gerichteter Graph $G = (V, E)$, wobei $E \subseteq V \times V$ eine Ordnung ist.
Knoten entsprechen Tasks und Kanten den Ablaufbedingungen.
Achtung: ein DG repräsentiert Parallelismus und nicht logische Verzweigungen.

Predecessor zB v_1 wenn $(v_1, v_2) \in E$
Sucessor zB v_2 wenn $(v_1, v_2) \in E$

10.2 Control-Data Flow Graph (CDFG)

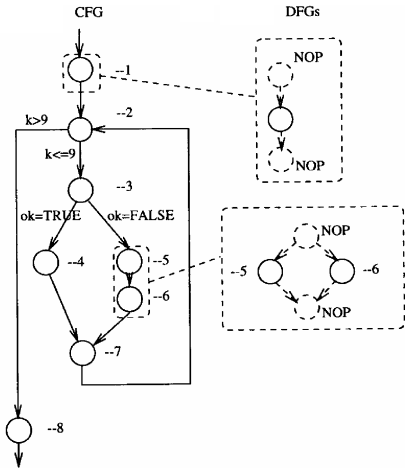


Abbildung 17: Control Flow Graph und Dependence Graph

Zweck Beschreibung von Kontroll-Strukturen und Datenabhängigkeiten.

Control Flow Graph Siehe Abbildung 17.
- Endlicher Zustandsautomat der den logischen Ablauf einer Programmes repräsentiert.
- Verzweigungen sind oft Knoten.
- Die Operationen in den Knoten sind im Dependence Graph aufgezeichnet.

Dependence Graph Siehe Abbildung 17.
- Beginnt und entdet mit einem `NOP`.

10.3 Sequence Graph

(Siehe Abbildung 18)

Ein Sequence Graph ist eine Hierarchie von Graphen mit den folgenden Eigenschaften:
- Knoten sind entweder:
  - Operationen, Tasks
  - Hierarchie Knoten
    CALL Modulauftuf
    BR Verzweigung (branch)
    LOOP Iteration
- Jeder Graph hat einen Start- und einen Endknoten mit einem `NOP`.

```
x := a*b;
y := x*c;
z := a+b;
submodul(a,z);

PROCEDURE submodul(m,n);
    p := m+n;
    q := m*n
END submodul;
```

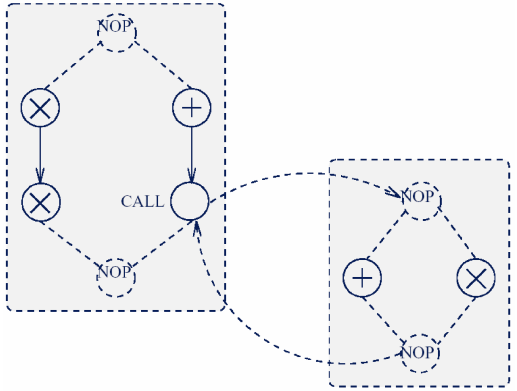


Abbildung 18: Sequence Graph

10.4 Marked Graphs (MG)

Definitionen

Marked Graph $G = (V, A, del)$ besteht aus
- Knoten (actors) $v \in V$
- Kanten $a = (v_i, v_j) \in A \subseteq V \times V$
- Initialverteilung von Tokens: $del : A \to \mathbb{N}$

Marking ist die Verteilung von Tokens, geschrieben als Vektor $del = (\dots del_k \dots) \in \mathbb{Z}^{1 \times |A|}$

Endlicher Automat ist ein Marked Graph mit
1. Jede Transition hat genau einen Ein- und einen Ausgang.
2. Es gibt nur ein Token.
3. Jede Transition ist eine Bedingung zugeschrieben, ohne deren Erfüllung sie nicht schalten kann.

Synchroner Datenflussgraph ist ein markierter Graph mit gewichteten Eingangs- und Ausgangskanten.

Aktionen
- Die Tokens auf den Kanten repräsentieren Daten in FIFO-Buffern.
- Eine Transition ist *aktiviert* oder *schaltbereit*, wenn auf jeder Input-Kante mindestens ein Token ist.
- Eine Transition kann *feuern*, wenn sie aktiviert ist.
- Wenn eine Transition *feuert*, wird auf jeder Input-Kante ein Token entfernt und auf jeder Output-Kante eins addiert.

Eigenschaften

Transitionen:

Tot: unter keiner erreichbaren Bedingung schaltbereit

Aktivierbar: unter einer Markierung schaltbereit

Lebendig: immer schaltbereit

Marked Graph:

Konservativ: die Anzahl Marken bleibt immer konstant

Tot: alle Transitionen sind tot

Beschränkt: es gibt nie mehr als B Marken

Beispiele Im Skript Seite 10-19ff.

11 Architekturentwurf

11.1 Modelle

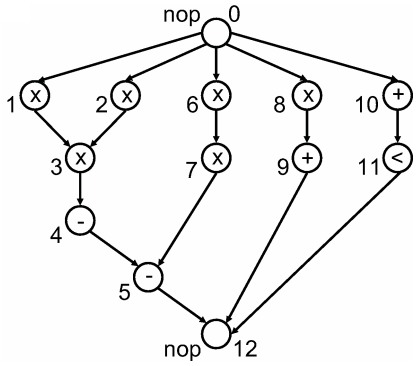


Abbildung 19: Sequenz-Graph $G_S = (V_s, E_S)$

Sequenz-Graph $G_S = (V_s, E_S)$ (Abb. 19)

V_S Operationen

E_S Abhängigkeiten

Ressourcen-Graph $G_R = (V_R, E_R)$ $V_R = V_S \cup V_T$ (Abb 20)

V_T Ressourcen-Typen

$(v_s, v_t) \in E_R =$ Zuordnung einer Operation v_s zu einer Ressource v_t. Auf der Kante steht oft die zugehörige Ausführungszeit.

Kosten-Funktion $c : V_T \to \mathbb{Z}$

Ausführungs-Zeit $w : E_R \to \mathbb{Z}$ Kosten werden den Operationen auf einer Ressource (v_s, v_t) zugeordnet.

Allokation $\alpha : V_T \to \mathbb{Z}$

Gibt zu jeder Ressource v_t die Anzahl $\alpha(v_t)$ *verfügbaren Instanzen* an.

Bindung $\beta : V_S \to V_T$ **and** $\gamma : V_S \to \mathbb{Z}$

- $\beta(v_s) = v_t$ ordnet der Operation v_s die *Ressource* v_t zu.
- $\gamma(v_s) = r$ ordnet der Operation v_s die *Instanz* r zu.

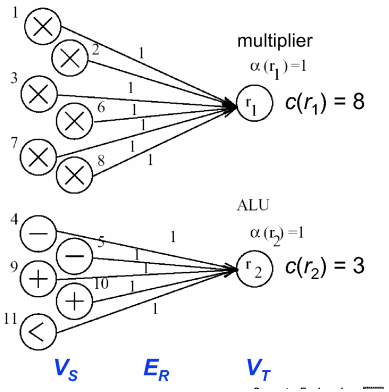


Abbildung 20: Ressourcen-Graph $G_R = (V_R, E_R)$, $V_R = V_S \cup V_T$

11.2 Scheduling

Schedule $\tau : V_S \to \mathbb{Z}$

Gibt die Startzeiten der Operationen v_s an.

Ein Schedule ist *zulässig*, wenn

$$\tau(v_j) - \tau(v_i) \geq w(v_i) \quad \forall (v_i, v_j) \in E_S$$

wobei $w(v_i) = w(v_i, \beta(v_i))$ die Ausführungszeit ist

Latency $L = \tau(v_n) - \tau(v_0)$

11.3 Scheduling ohne Ressourceneinschränkungen

Latenz Minimierung gegeben ein Sequenzgraph G_S und ein Ressourcengraph G_R.

$$L = \min\left\{\tau(v_n) : \tau(v_j) - \tau(v_i) \geq w(v_i) \quad \forall (v_i, v_j) \in E_S\right\}$$

11.3.1 ASAP

Algorithmus: (immer mit unbeschränkten Ressourcen)

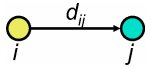
```
ASAP (G_S(V_S, E_S), w) {
    τ(v_0) = 1;
    REPEAT {
        Finde v_i dessen Vorgänger geplant sind
        τ(v_i) = max{τ(v_j) + w(v_i)   ∀(v_j, v_i) ∈ E_S}
    } UNTIL (v_n geplant);
    RETURN (τ);
}
```

11.3.2 ALAP

Algorithmus: (immer mit unbeschränkten Ressourcen)

```
ALAP (G_S(V_S, E_S), w, L_max) {
    τ(v_n) = L_max
    REPEAT {
        Finde v_i dessen Nachfolger geplant sind
        τ(v_i) = min{τ(v_j)   ∀(v_j, v_i) ∈ E_S} − w(v_i)
    } UNTIL (v_0 geplant);
    RETURN (τ);
}
```

11.3.3 Zeiteinschränkungen



$$\tau(v_j) - \tau(v_i) \geq d_{i,j}$$

Abbildung 21: Timing Constraints

Klassen von Zeiteinschränkungen (Abb. 21):

Deadline: $\tau(v_i) + w(v_i) \leq d_i$
$\Rightarrow \quad \tau(v_0) - \tau(v_i) \geq -d_i + w(v_i)$

Release Time: $\tau(v_i) \geq a$
$\Rightarrow \quad \tau(v_i) - \tau(v_0) \geq a$

Minimum Constraint: „v_j mindestens l_{ij} nach v_i“
$\Rightarrow \quad \tau(v_j) - \tau(v_i) \geq l_{ij}$

Maximum Constraint: „v_j mindestens l_{ij} vor v_i“ oder „v_i spätestens l_{ij} nach v_j“
$\Rightarrow \quad \tau(v_j) - \tau(v_i) \geq -l_{ij}$

Differenz von ASAP und ALAP bezeichnet vorhandene Flexibilität.

Weighted Constraint Graph (Abb. 21)

- $G_C = (V_C, E_C, d)$ zu einem Sequenzgraphen $G_S = (V_S, E_S)$ mit den Knoten $V_C = V_S$ und einer gewichteten Kante für jede Zeiteinschränkung.
- Jede Kante $(v_i, v_j) \in E_C$ bezeichnet die Beschränkung $\tau(v_j) - \tau(v_i) \geq d(v_i, v_j)$.

Bellmann-Ford Algorithmus findet Startzeitpunkte τ für alle v_i.

```
Anfangswerte: v_i = −∞ und v_0 = 0
Setze iterativ τ(v_j) := max{τ(v_j), τ(v_i) + d(v_i, v_j) :   (v_i, v_j) ∈ E_C}
für alle v_j ∈ V_C
```

Es dürfen in G_S keine Zyklen vorkommen. Bellmann-Ford ist $O\left(|V_S|^3\right)$

11.3.4 Pareto-Punkte

Pareto-Punkte sind eine Abbildung verschiedener Konfigurationen im Entwurfsraum (zB. Kosten-Latenz-Diagramm). Ein Pareto Punkt ist optimal, das heisst, dass es keinen Punkt gibt, welcher in in allen Eigenschaften übertrifft.

Zum Beispiel gibt es keinen Punkt mit weniger Kosten bei gleicher Latenz und keinen mit kleinerer Latenz bei gleichen Kosten.

11.4 Scheduling mit Ressourcenbeschränkungen

Gegeben sind ein Sequenz-Graph $G_S = (V_S, E_S)$, ein Ressourcen-Graph $G_R(v_R, E_R)$ und die zugehörigen Allocation α und Binding β

Minimal Latency

$$L = \min\{\tau(v_n) : \underbrace{\left(\tau(v_j) - t(v_i) \geq w\left(v_i, \beta(v_i)\right) \quad \forall (v_i, v_j) \in E_S\right)}_{\text{Schedule feasible}} \wedge \underbrace{\left(\left|\{v_s : \beta(v_s) = v_t \wedge \tau(v_s) \leq t \leq \tau(v_s) + w(v_s, v_t)\}\right| \leq \alpha(v_t)\right)}_{\text{Allokation}} \quad \forall v_t \in V_T, \forall 1 \leq t \leq L_{max}\}$$

11.4.1 List Scheduling

Ein greedy Algorithmus, welcher nicht immer die optimale Lösung findet.

```
LIST (G_S(V_S, E_S), G_R(V_R, E_R), α, β, priorities) {
    t = 1
    REPEAT {
        FORALL v_k ∈ V_T {
            U_k = Ops die auf v_k gescheduled werden können
            T_k = auf v_k laufende Operationen
            wähle S_k ⊆ U_k mit maximaler Priorität
                und |S_k| + |T_k| ≤ α(v_k)
            τ(v_i) = t   ∀v_i ∈ S_k
        }
        t = t + 1
    } UNTIL (v_n geplant)
    RETURN (τ);
}
```

11.4.2 Integer Linear Programming

Minimierung einer Zielfunktion mit linearen Ungleichungen. Findet immer eine optimale Lösung.

Integer Linear Programm

- minimiere:

$$\tau(v_n) - \tau(v_0)$$

- mit

$$x_{i,t} \in \{0,1\} \quad \forall v_i \in V_S \quad \forall t : l_i \le t \le h_i \tag{1}$$

$$\sum_{t=l_i}^{h_i} x_{i,t} = 1 \quad \forall v_i \in V_S \tag{2}$$

$$\sum_{t=l_i}^{h_i} t \cdot x_{i,t} = \tau(v_i) \quad \forall v_i \in V_S \tag{3}$$

$$\tau(v_j) - \tau(v_i) \ge w(v_i) \quad \forall (v_i, v_j) \in E_S \tag{4}$$

$$\underbrace{\sum_{\forall i:(v_i,v_k)\in E_R}}_{\text{Operationen } v_i \text{ auf } v_k} \underbrace{\sum_{p'=\max\{0,t-h_i\}}^{\min\{w(v_i)-1,t-l_i\}}}_{1 \text{ wenn } v_i \text{ auf } v_k} x_{i,t-p'} \le \alpha(v_k) \tag{5}$$

$$\forall v_k \in V_T \quad \forall t : 1 \le t \le \max\{h_i : v_i \in V_S\}$$

- Erklärungen
  - (1) Variablen $x_{i,t}$ sind binär.
    - Mit ASAP und ALAP (mit unbeschränkten Ressourcen) die obere und untere Schranke h_i und l_i für die Startzeitpunkte der Operation v_i berechnen.
    - L_{max} für ALAP zum Beispiel mit ASAP und Ressourcenbeschränkung bestimmen.
  - (2) Genau eine Variable $x_{i,t}$ ist 1, die anderen sind 0.
    $x_{i,t}$ ist 1, wenn die Operation v_i zum Zeitpunkt t startet.
  - (3) Definiert $\tau(v_i)$ als den Startzeitpunkt der Operation v_i.
  - (4) Ablaufbedingungen müssen erfüllt sein.
  - (5) Ressourceneinschränkungen müssen für alle Ressourcen $v_k \in V_T$ zu allen Zeiten t erfüllt sein.

11.5 Iterative Algorithmen

Repräsentationen

a) Gleichung mit diskreten Indizes: $y[l] = au[l] + by[l-1] + cy[l-2] + dy[l-3]$
b) Gleichungssystem: $x_1[l] = au[l] \quad \wedge \quad y[l] = x_1[l] + by[l-1] \quad \wedge \ldots$
c) Extended sequence graph: $G_S = (V_S, E_S, d)$ Auf jeder Kante steht ein Index Displacement d_{ij}. Dies bedeutet, dass die Varible in v_j abhängt von der Variable v_i aber d_{ij} Zeitpunkte früher. Siehe Abbildung 22.
d) Marked Graph
e) Signalflussgraph
f) Programm mit Schleifen

Definitionen

Iteration Set von Operationen um $y[l]$ zu berechnen.
Latency L Zeitdifferenz zwischen dem Start der ersten und dem Ende der letzten Operation der gleichen Iteration.
Iteration Interval P Zeit zwischen zwei Iterationen.
Throughput $1/P$

Implementationen

a) *Einfache Möglichkeit*: Sequenzgraphen implementieren.
$\Rightarrow L = P$
b) *Functional Pipelining*: Parallelle Ausführung von Iterationen, siehe Abbildung 22.

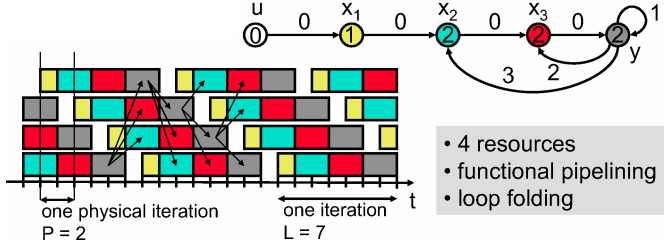


Abbildung 22: Functional Pipelining

(a) Benutze den Extended Sequence Graph.
(b) ASAP und ALAP für die obere und untere Schranke h_i und l_i benützen nur Kanten mit $d_{ij} = 0$.
(c) (4) wird ersetzt durch

$$\tau(v_j) - \tau(v_i) \ge w(v_i) - d_{ij} \cdot P \qquad \forall (v_i, v_j) \in E_S$$

Grund: $\underbrace{\tau(v_i) + w(v_i)}_{\text{Erzeugungszeitpunkt}} \le \underbrace{\tau(v_j) + d_{ij} \cdot P}_{\text{Benutzungszeitpunkt}}$

(d) (5) wird ersetzt durch

$$\sum_{\forall i:(v_i,v_k)\in E_R} \sum_{p'=0}^{w(v_i)-1} \sum_{\forall p: l_i \le t-p'+p\cdot P \le h_i} x_{i,t-p'+p\cdot P} \le \alpha(v_k)$$

$$\forall 1 \le t \le P, \qquad \forall v_k \in V_T$$

Denn eine Operation belegt die Ressource während der Kontrollschritte t mit

$t = \tau(v_i) + p' - p \cdot P$ „Startzeitpunkt + Rechenzeit + frühere Perioden“
$\forall p' : 0 \le p' \le w(v_i) - 1$ „Rechenzeit“
$\forall p : l_i \le t - p' + p \cdot P \le h_i$ „Instanzen früherer Perioden, welche bei t noch rechnen.“

11.6 Dynamic Voltage Scaling

Optimiere den Energieverbrauch mit Hilfe eines ILPs.

$d(v_i)$**:** Deadline der Operation v_i
$w_k(v_i)$**:** Ausführungszeit des Energylevels $k \in K$
$e_k(v_i)$**:** Energylevel des Taks v_i

Integer Linear Programm

- Gesamtenergie:

$$\sum_{k \in K} \sum_{v_i \in V_S} y_{ik} \cdot e_k(v_i)$$

- mit

$$y_{ik} \in \{0,1\} \qquad \forall v_i \in V_S, k \in K \tag{1}$$

$$\sum_{k \in K} y_{ik} = 1 \qquad \forall v_i \in V_S \tag{2}$$

$$\tau(v_j) - \tau(v_i) \geq \underbrace{\sum_{k \in K} y_{ik} \cdot w_k(v_i)}_{\text{Ausführungszeit}} \quad \forall (v_i, v_j) \in E_S \tag{3}$$

$$\underbrace{\tau(v_i) + \sum_{k \in K} y_{ik} \cdot w_k(v_i)}_{\text{Rechenzeit}} \leq d(v_i) \quad \forall v_i \in V_S \tag{4}$$

Index