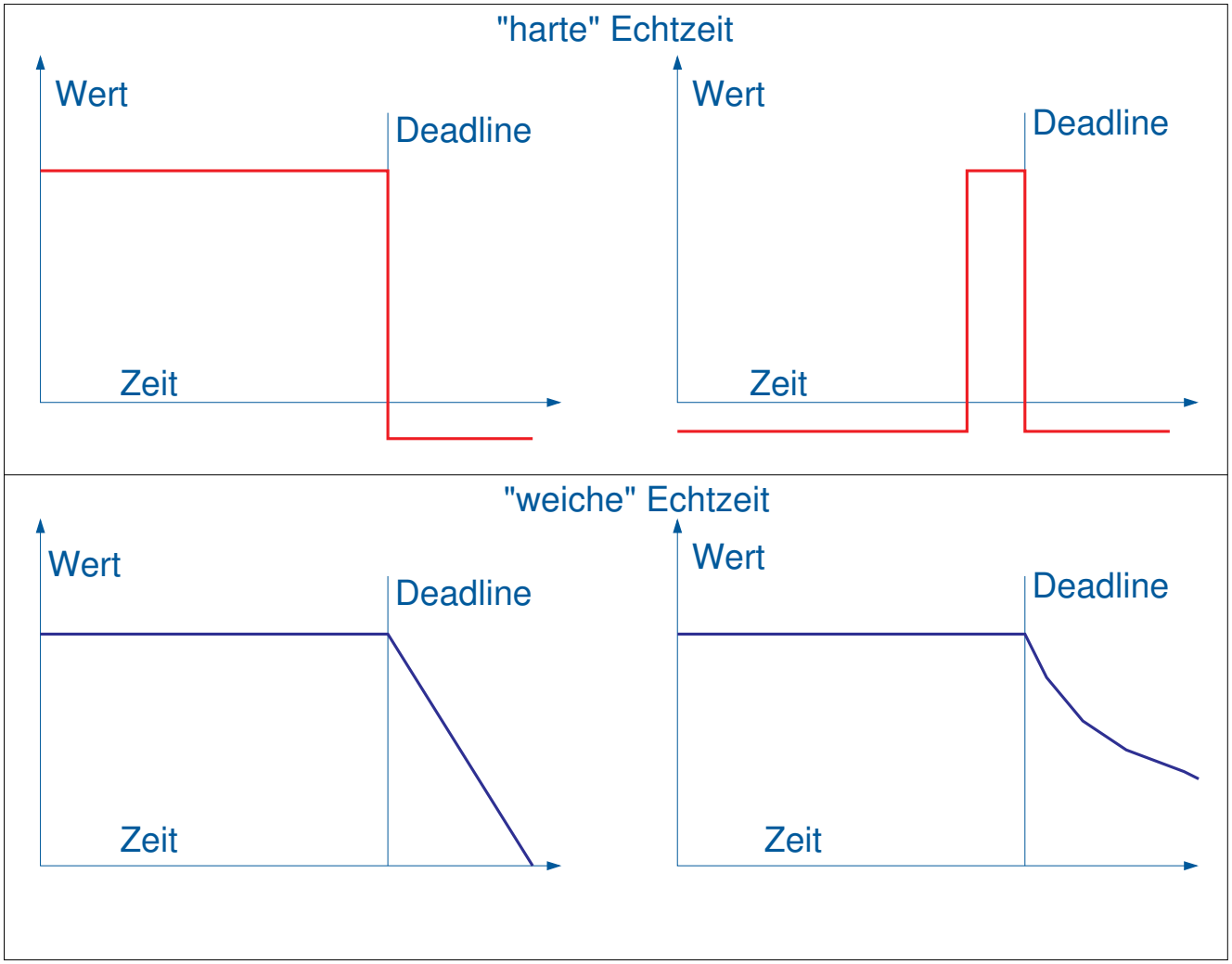
EMES: Eigenschaften mobiler und eingebetteter Systeme

Weiche Echtzeit

Dr. Felix Salfner, Dr. Siegmar Sommer

Wintersemester 2010/2011

Wiederholung - Resultat/Wert-Funktion



Echtzeitsysteme sind Computersysteme, bei denen der Nutzen eines Resultates nicht nur vom Resultat abhängt, sondern auch vom Zeitpunkt der Auslieferung des Resultats.

Soft Realtime

Doing hard Realtime is hard...

Soft Realtime

Doing hard Realtime is hard...

... but doing soft Realtime is much harder!

Soft-Realtime-Anwendungen

- Multimedia-Applikationen
  - fehlende Frames reduzieren Qualität, sind aber keine Katastrophe
- Geschäftsprozesse (Banken, Börsen, Online-Handel)
  - zu spät kommende Daten können kostspielige Folgen haben, aber keine Katastrope
  - In Überlastsituationen sind längere Antwortzeiten akzeptabel
- Telekommunikation
  - Anrufton kann in Einzelfällen verspätet kommen
  - Abbruch einer Verbindung ist in gewissen Grenzen akzeptabel

Performancemaße

- Durchschnittswerte sind im Gegensatz zu harten Echtzeitsystemen als Maß akzeptabel
- Verteilungsfunktionen definieren Qualität des Ergebnisses
- Statistische Betrachtungen statt Garantien

Ziel: Möglichst hoher Anteil eingehaltener Deadlines

Problem: Schon bei recht kleinen Problemen NP-vollständig

Jitter

- Ungenaues Einhalten von Deadlines (oder Verpassen) führt zu erhöhtem Jitter
- Multimediaanwendungen:
  - Jitter ist störender als geringfügig inkorrekte Geschwindigkeit
  - Jitter senkt die Qualität
- Geschäftsprozesse:
  - Jitter stört im allgemeinen Fall nicht
  - Jitter kann stören, wenn dadurch die Anwendungslogik verletzt wird
- Telekommunikation
  - Bei Verarbeitung von audiovisuellen Daten: wie Multimedia
  - Beim Schalten von Verbindungen: Jitter stört weniger als abgebrochene Prozesse

Quality of Service (QoS)

"QoS ist die Fähigkeit eines Systems, Anforderungen an den gelieferten Service zu erfüllen"

Parameter:

- Jitter
- eingehaltene Deadlines
- Vorhersagbarkeit
- Bandbreite
- Durchsatz
- End-to-End Parameter

Klar:
Ein hartes Echtzeitsystem liefert (in den meisten Fällen) den besten QoS

Aber: Harte Echtzeitsysteme sind teuer, denn...

Kosten harter Echtzeitsysteme

- Hoher Ressorcenbedarf durch
  - Auslegung für Worst-Case-Szenarien
  - WCET-Analysen mit (notwendigen) pessimistischen Einschätzungen
  - Beschränkung “erlaubter” Programmiermethoden
- Hohe Kosten durch
  - Ressourcenüberdimensionierung (s.o.)
  - Validierungsverfahren, Zertifizierungen, Nachweise
  - unflexible Architekturen (im Fall nachträglicher Erweiterungen)
  - spezielle Hardware
  - spezielle Programmierumgebungen (hoher Entwicklungsaufwand)

Idee: Bei weicher Echtzeit wird vieles davon nicht benötigt!

Soft Realtime auf COTS-Systemen I

COTS: Commercial off-the-shelf

Vorteile:

- Durch Massenmarkt billige Hardware
- Durch Massenmarkt statistische Daten vorhanden (Zuverlässigkeit)
- Betriebssysteme ebenfalls off-the-shelf
- Keine oder kaum Abhängigkeiten von speziellen Herstellern

aber:

- Zuverlässigkeit ist nicht vergleichbar mit harten Echtzeitsystemen
- Betriebssysteme sind auf “best effort” optimiert, nicht auf zeitliche Vorhersagbarkeit
- Kein vollständiges Wissen über Abläufe im System

Soft Realtime auf COTS-Systemen II

- Harte Echtzeit nicht oder nur mit grossem Aufwand erreichbar (aber auch nicht nötig für Soft Realtime Systeme)
- Kompromisse sind erforderlich
  - dedizierte Systeme mit eingeschränkter Funktion erhöhen Wissen über Systemabläufe
  - Einfachere Hardware und/oder Systemsoftware erhöhen Wissen über Systemabläufe
  - zwischen Performance und Quality of Service
  - zwischen Vorhersagbarkeit und Funktionalität
- Klare Trennung ist erforderlich zwischen "Echtzeit"-Prozessen und übrigen Abläufen auf dem System

Idee: Erweiterung des OS-Scheduler

- Nieh and Lam 1997: SMART
  - „Scheduler for Multimedia And Real-Time“
  - Erweiterung / Ersatz des SunOS 2.5 Schedulers
  - Basiert auf einer *importance* (abgeleitet von priorities und weighted fair queueing) und einer *urgency* (angelehnt an EDF)

- Verwendung eines RT-Betriebssystem, verpasste Deadlines werden in Kauf genommen

Zusammenfassender Artikel:
Nieh, J. & Lam, M. S., 2003. A SMART scheduler for multimedia applications. *ACM Transactions on Computer Systems*, 2(21), pp. 117-163

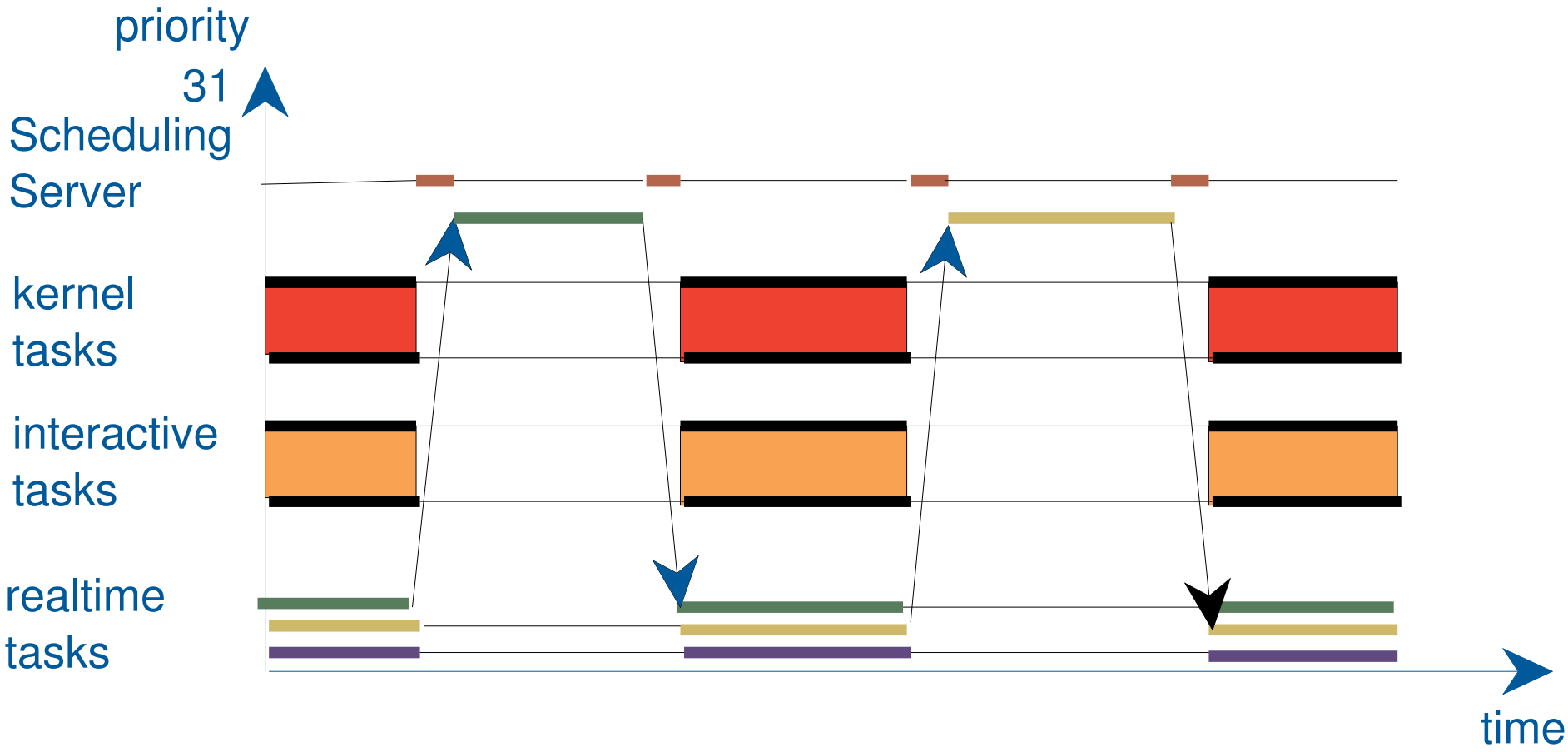
Idee: Benutzung von OS-Prioritäten

- (Fast) jedes Betriebssystem arbeitet mit verschiedenen Prioritätsebenen und Prioritäts-Scheduling
- Diese Prioritäten können benutzt werden, wenn man Mechanismen wie Aging und Priority-Boosts abschaltet (*fixed priority scheduling*)
- Scheduling obliegt OS-Scheduler
- Probleme:
  - Betriebssysteminterne Prozesse hoher Priorität können "stören"
  - Abbildung auf Prioritäten ist nicht einfach, da Bereich begrenzt ist (muß oberhalb aller anderen Prioritäten liegen)
  - Implementation des OS-Schedulers nicht immer bekannt
  - Systeminterne Abläufe sind nicht immer bekannt

Idee v2: Benutzung eines “Scheduling Servers”

- Benutzung der Prioritäten des Betriebssystems als Grundlage
- Nutzung der beiden höchsten Prioritätsebenen für die Implementation eines “Echtzeit-Schedulers” für die Verwaltung der zeitkritischen Tasks
- Scheduling der “Echtzeit-Tasks” obliegt Scheduling Server
- Funktion:
  - Scheduling Server läuft periodisch auf höchster Priorität
  - Manipuliert Prioritäten der anderen Prozesse so, daß zweite Prioritätsstufe für Echtzeit-Tasks benutzt wird
- Idee wurde im Rahmen von weichen Echtzeitsystemen auf Basis von COTS-Systemen mehrfach parallel entwickelt, u.a. durch Dr. Andreas Polze am Institut für Informatik der HU

HU Scheduling Server — Funktion



Scheduling Server — Vorteile

- Keine Modifikationen am Betriebssystem erforderlich
- Keine Kenntnis des Source-Codes erforderlich
- Scheduling Server kann intern beliebige Prioritäten benutzen
- Scheduling Server kann intern beliebige Schedulingverfahren implementieren
- System bleibt (mit reduzierter Leistung) für Nicht-Echtzeit-Aufgaben benutzbar
- Leistung für Echtzeit-Tasks *weitgehend* konstant

Scheduling Server — Nachteile

- Abläufe im Betriebssystemkern sind nicht vollständig bekannt, damit unvorhersehbare Einflüsse möglich
- Benutzung von Systemaufrufen kann unvorhersehbare Auswirkungen auf das zeitliche Verhalten haben
- Scheduling Server kann Systemprozesse behindern (z.B. grafische Oberflächen)
- Für harte Echtzeit nicht geeignet

Scheduling Server — Implementationen

Auf folgenden Plattformen wurde die Idee des Scheduling Servers implementiert und untersucht (u.a.):

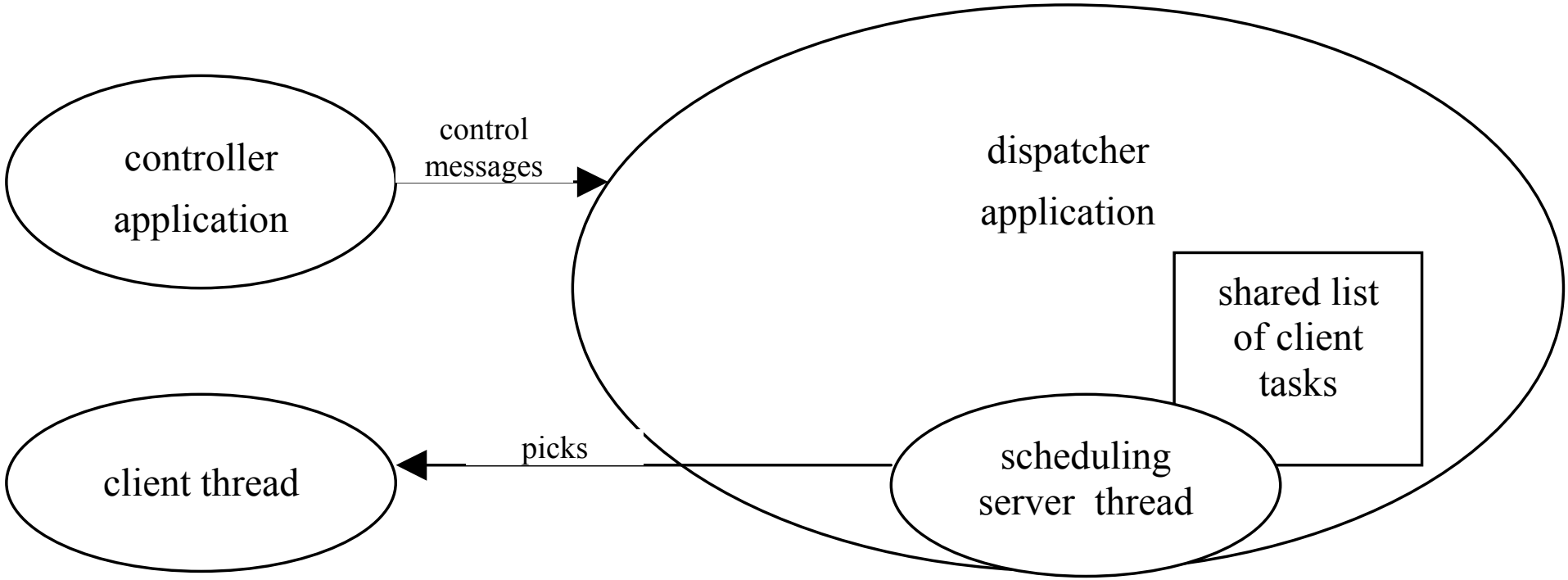
- Mach (NeXTStep 3.3)
- Solaris
- rtLinux 0.x
- LynxOS 3.0
- Windows NT
- Voraussetzungen: Fixed Priority Scheduling, Möglichkeit, Scheduling-Politiken umzuschalten, Möglichkeit, Prioritäten umzuschalten

Scheduling Server — NeXTStep

Scheduling Server - main loop

```
thread_t        th_id;
mutex_t         th_list_lock;
condition_t     new_th_reg;

while(1) {
   mutex_lock( th_list_lock );
   while ((th_id = next_edf( thread_list )) == THREAD_NULL)
         condition_wait( new_th_reg, th_list_lock );

   /* set scheduling policy and quantum, resume thread */
   thread_policy(th_id, POLICY_FIXEDPRI, time_rt);
   thread_priority(th_id , real_time_prio, FALSE);
   thread_resume( th_id );

   /* handoff scheduling, real-time thread runs until its quantum expires */
   thread_switch(th_id, SWITCH_OPTION_DEPRESS, 0);

   thread_suspend( th_id );
   mutex_unlock( th_list_lock );

   /* give rest of the system a chance, invoke Mach scheduler */
   thread_switch(THREAD_NULL, SWITCH_OPTION_WAIT, time_os);
}
```

Scheduling Server — Windows / Solaris



Scheduling Server — Windows / Solaris

```
// save actual values and free access to threadList array
actHandle = threadList[actIndex].hThread;
actHighPhase = threadList[actIndex].highPhase;
actLowPhase = threadList[actIndex].lowPhase;
LeaveCriticalSection(&lock);
if (ResumeThread(actHandle) == -1) {
  EnterCriticalSection(&lock);          // error, thread doesn't exist
  threadList[actIndex].hThread = NULL;  // we continue, trying next thread
}
else  {
  COUNT_START;                          // start timing log
  Sleep(actHighPhase);                  // suspend ss for high-period
  // ss wakes up, stop counter & suspend client thread
  COUNT_END;
  timerLog[logIndex++] = *((__int64*)(count_s));
  timerLog[logIndex++] = *((__int64*)(count_e));
  if (SuspendThread(actHandle) == -1)  {
    EnterCriticalSection(&lock);          // thread doesn't exist anymore
    threadList[actIndex].hThread = NULL;
    LeaveCriticalSection(&lock);
    // thread used his high phase time , so we leave
    // the low phase time to the rest of the system
  }
  COUNT_START;                          // start timing log
  Sleep(actLowPhase);                   // suspend ss for low-period
  COUNT_END;                            // ss wakes up, stop counter
  timerLog[logIndex++] = *((__int64*)(count_s));
  timerLog[logIndex++] = *((__int64*)(count_e));
  EnterCriticalSection(&lock);          // necessary on loop start
}
```

Scheduling Server — Analyse

Interessante Probleme:

- Wie stabil ist die Leistung, die einer Echtzeit-Task zur Verfügung steht?
- Wie hoch ist der Overhead des Scheduling Servers?
- Welchen Einfluß haben Non-RT-Lasten im Hintergrund?
- Welchen Einfluß haben I/O-Prozesse?

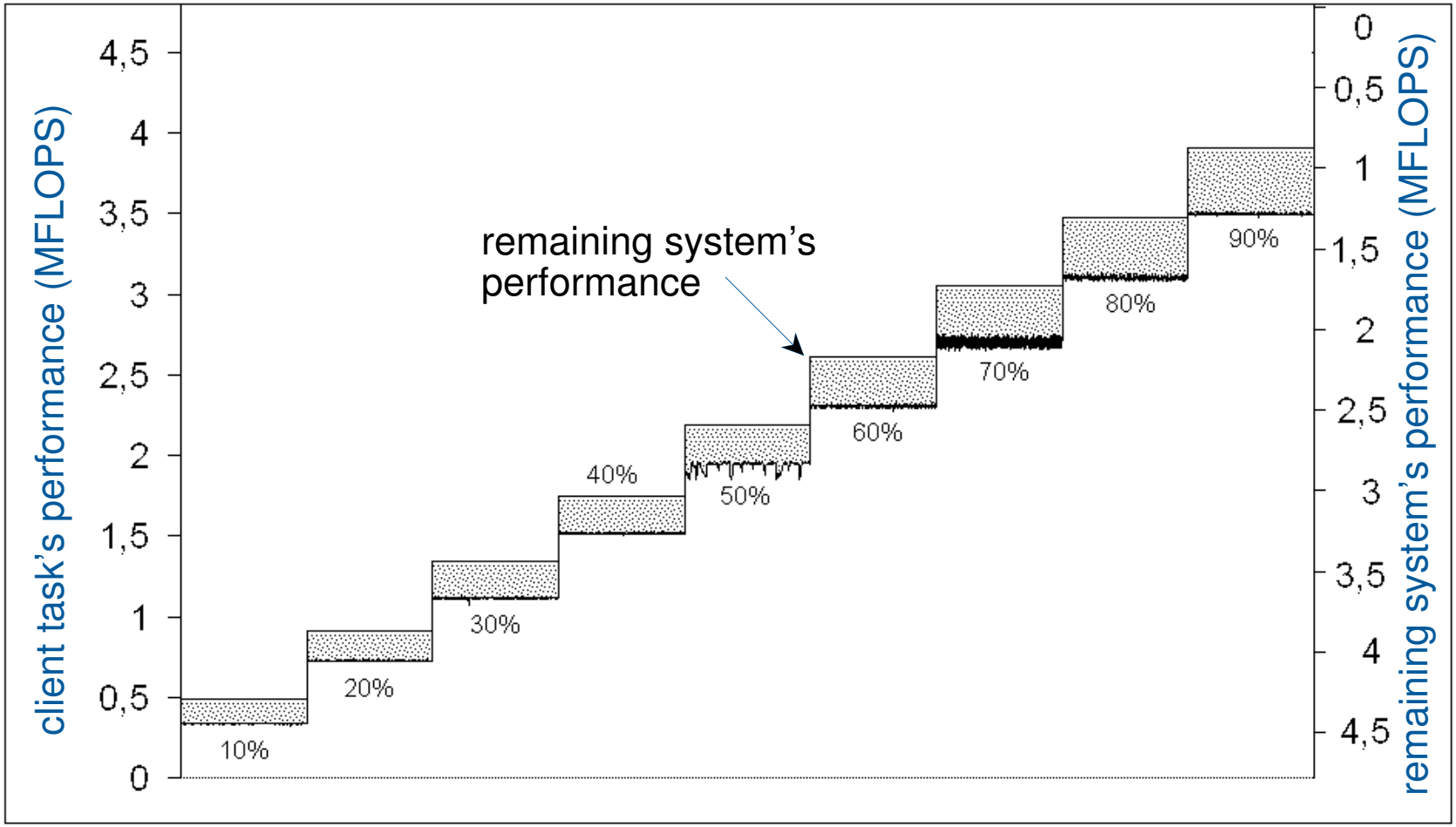
Messung der Rechenleistung: Linpack / SciMark

Metriken

- Gesamtleistung G, kontrollierte Leistung L_i, unkontrollierte Leistung L_e
- Programm wird unter Kontrolle mit Anteil p der CPU ausgeführt, ohne Overhead gilt $L_i = pG$ und $L_e = (1-p)G$
- Interner Overhead für die kontrollierte Applikation: $O_i = 1 - \frac{L_i}{pG}$
- Externer Overhead für die unkontrollierten Applikationen: $O_e = 1 - \frac{L_e}{(1-p)G}$
- Negativer Overhead bedeutet Überbuchung

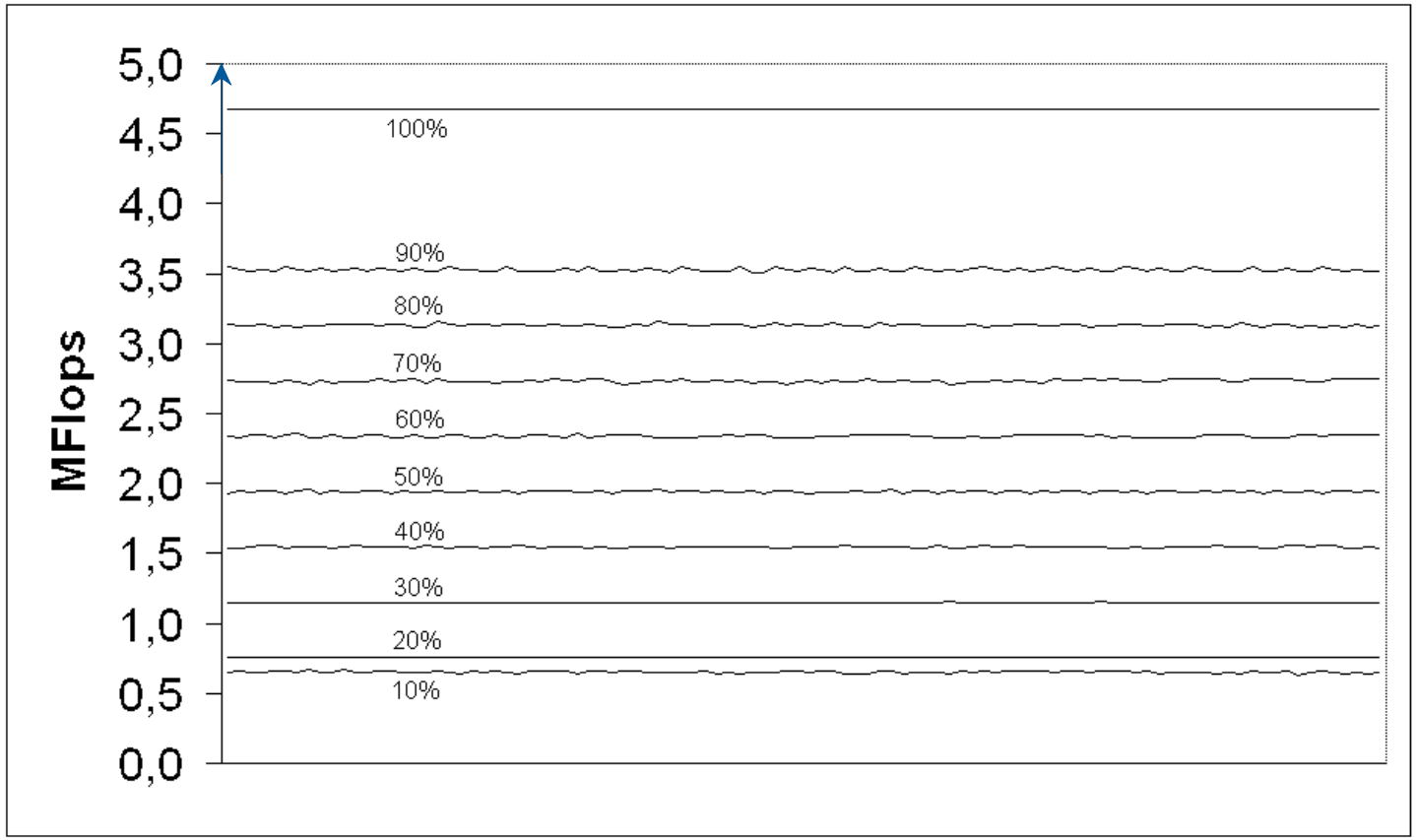
Scheduling Server — Overhead

System: NeXTStep 3.3



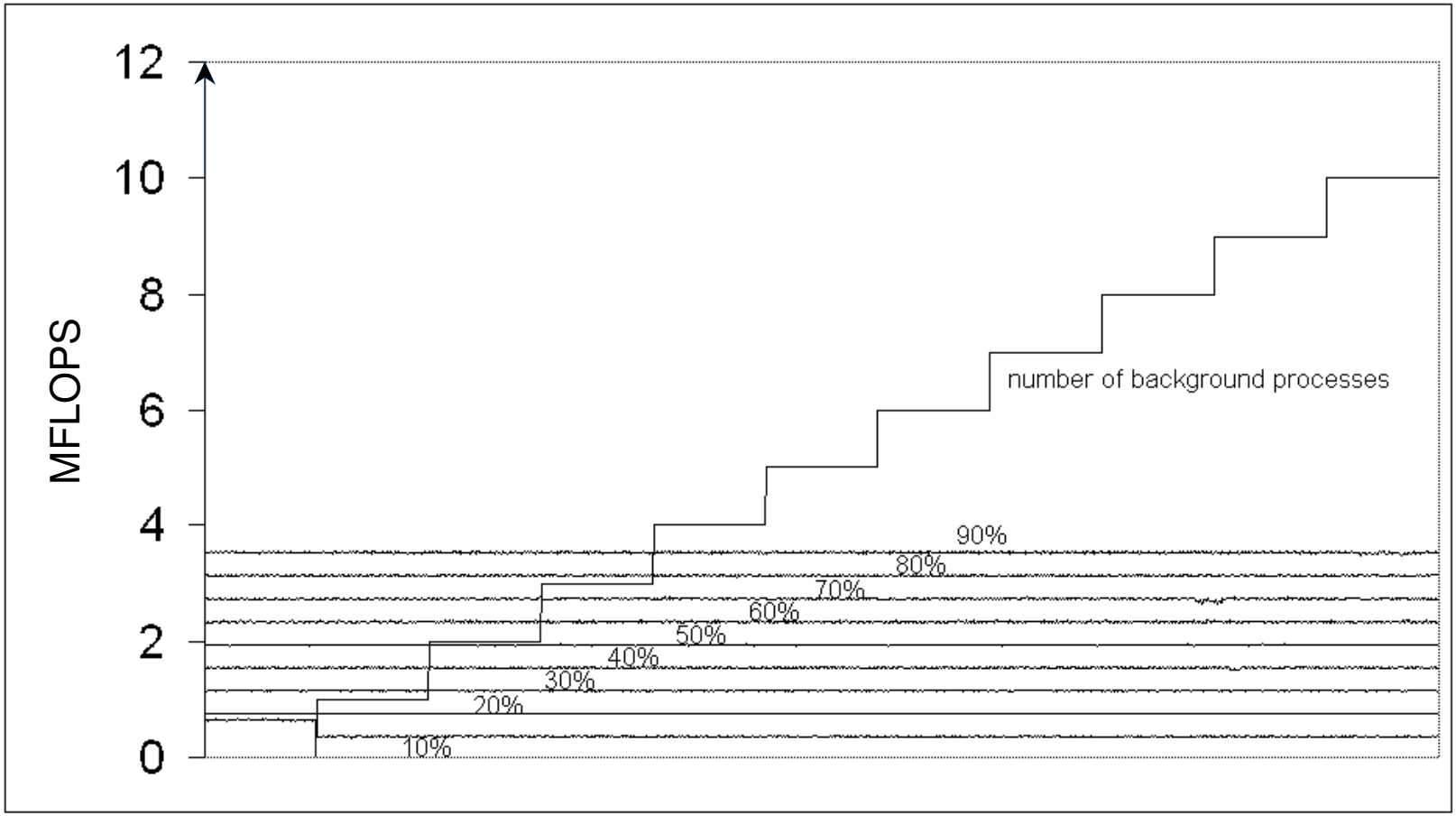
Scheduling Server — Stabilität ohne Last

System: NeXTStep 3.3



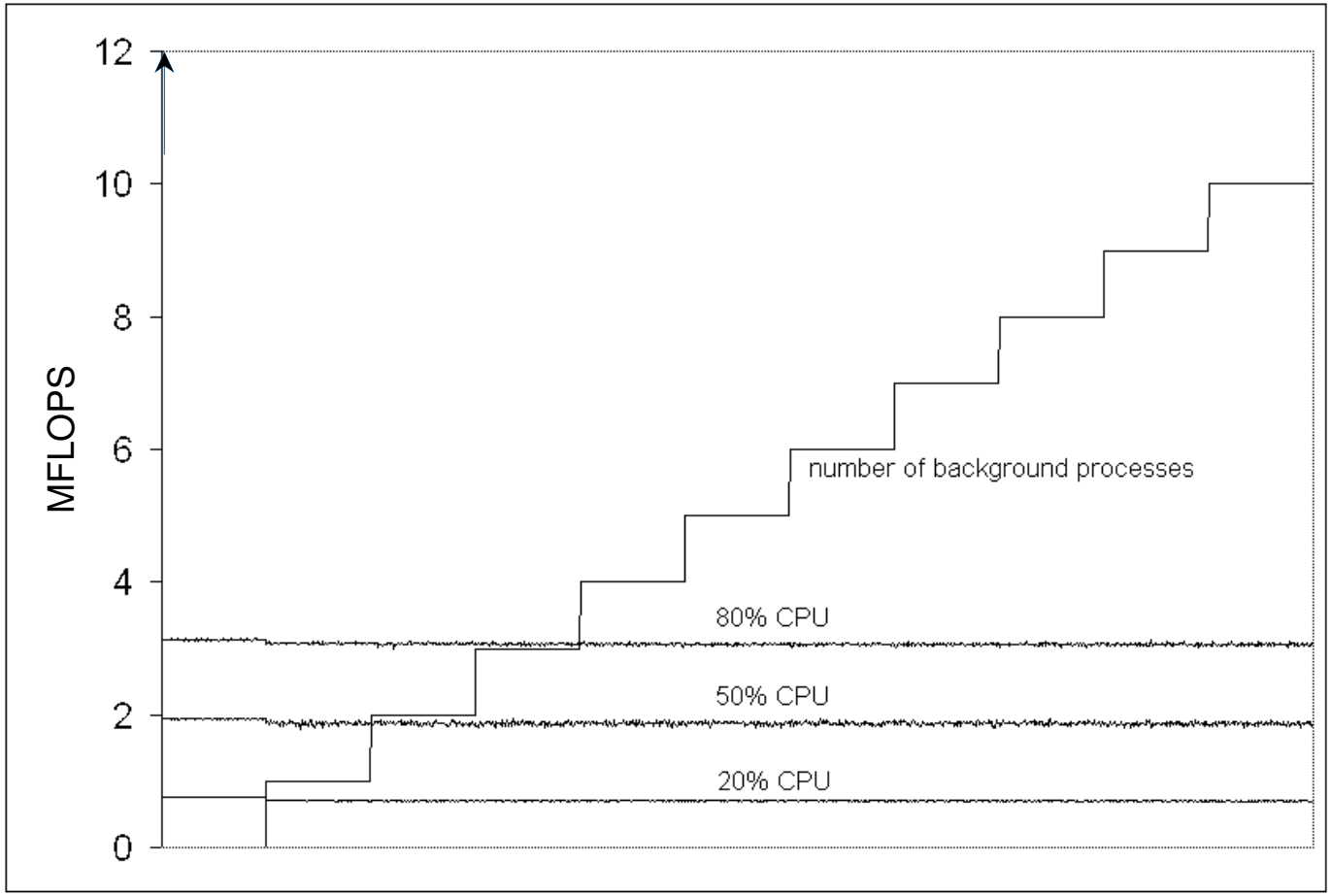
Scheduling Server — Stabilität mit Last

System: NeXTStep 3.3



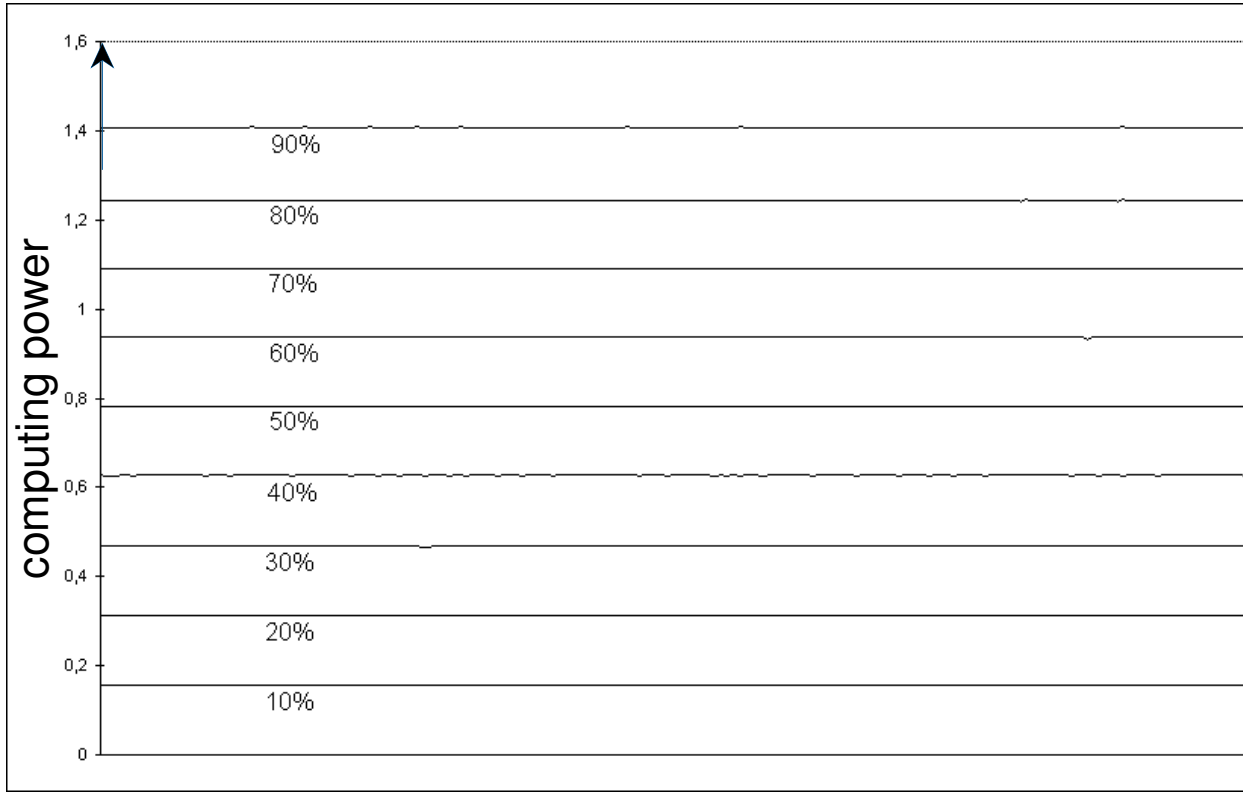
Scheduling Server — Stabilität mit I/O-Last

System: NeXTStep 3.3



Scheduling Server — Stabilität

System: rtLinux 0.x



Scheduling Server — Stabilität

System: SunOS 2.5

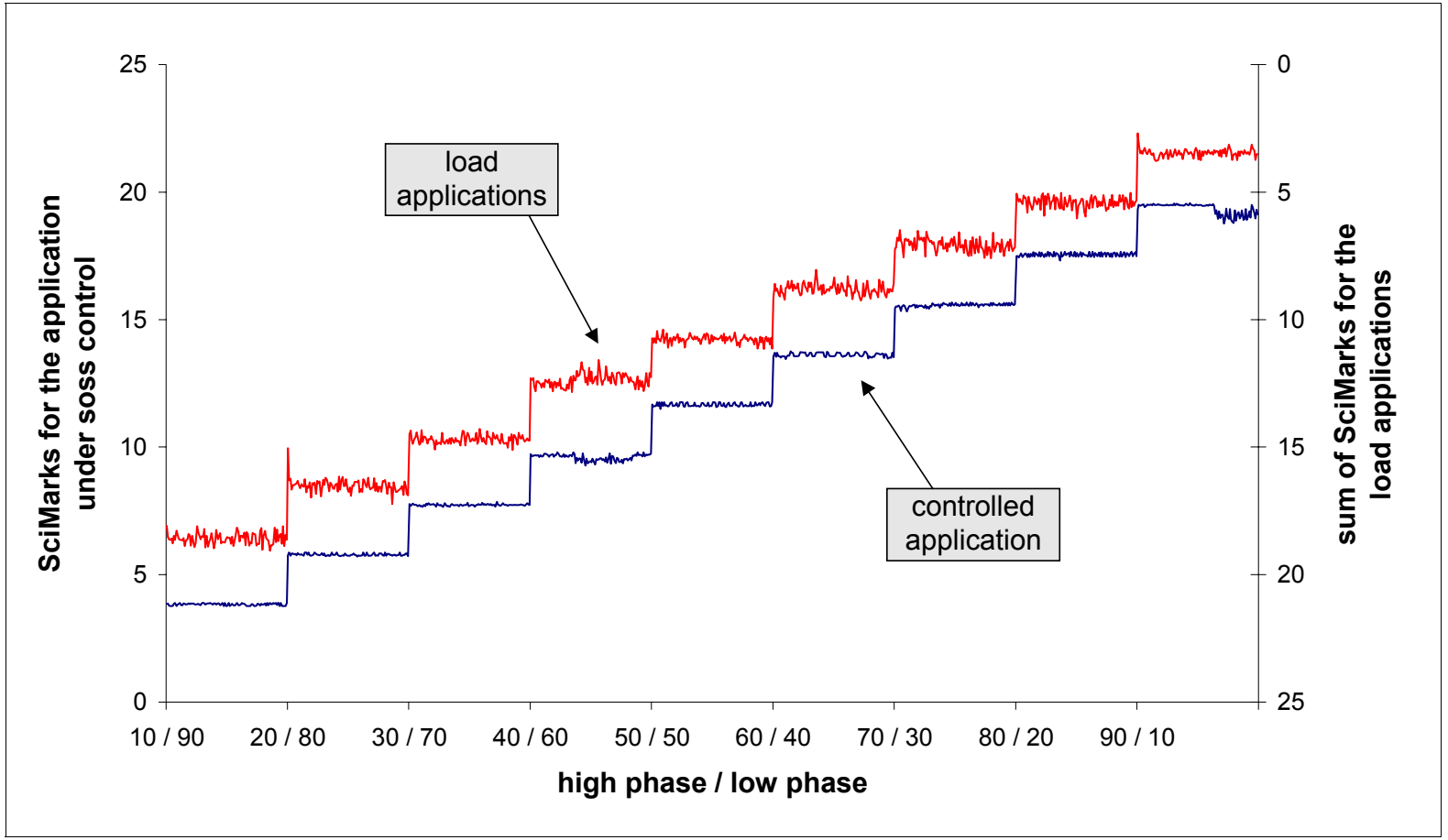
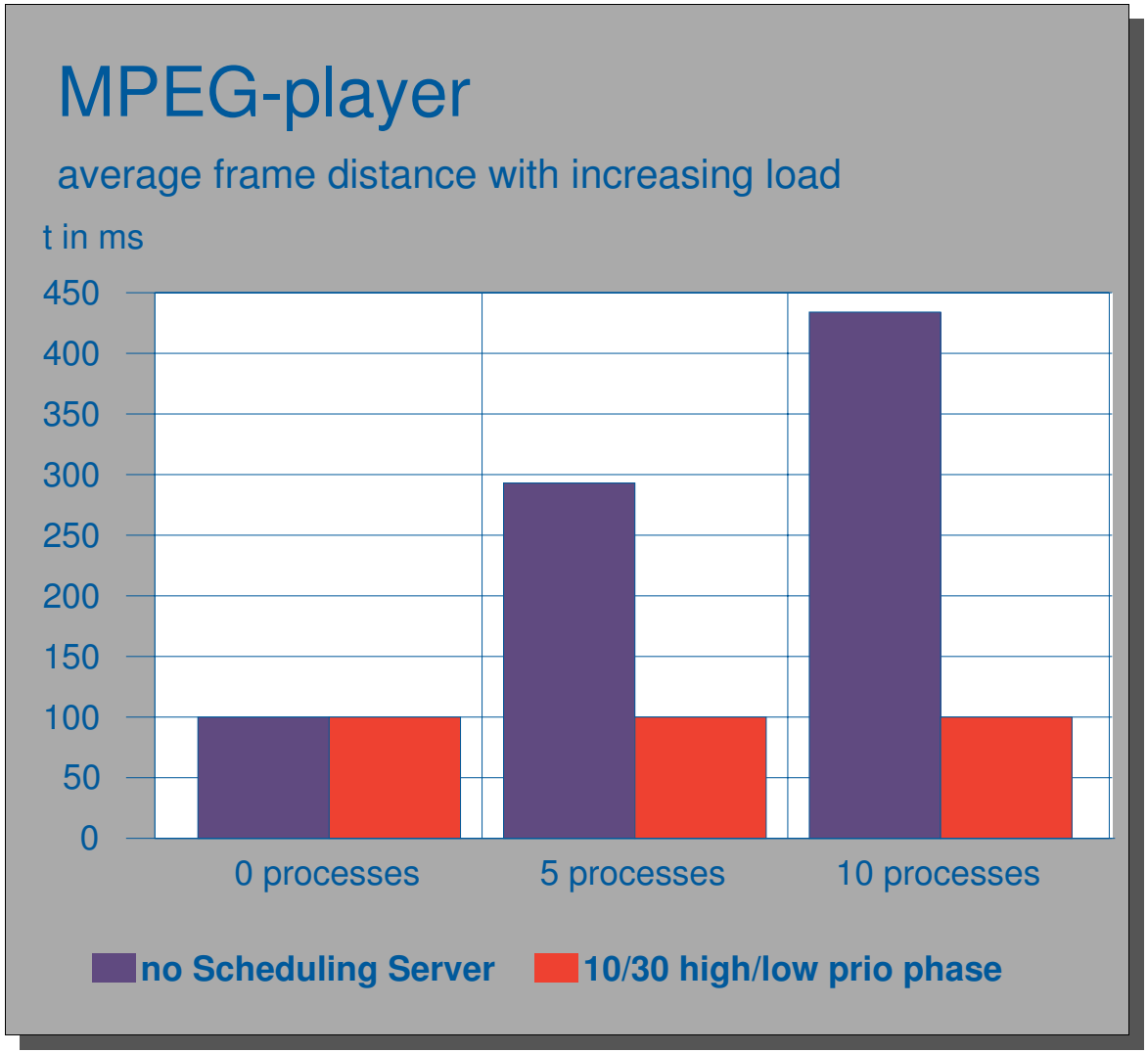


Figure 6.5: SciMark benchmark under SOSS control

Anwendung auf eine Multimedia-Anwendung I

- Multimedia-Anwendungen haben weiche Echtzeit-Anforderungen
- Verletzung dieser Anforderungen führt zu verminderter oder inakzeptabler Qualität
- Beispiel: MPEG-Player
  - auf unbelastetem System benutzbar
  - Belastung des Systems führt zu reduzierter verfügbarer Rechenleistung: unakzeptables Resultat
  - Scheduling Server sorgt für garantierte Rechenzeit für MPEG-Player
- Problem: Window-Server wird benutzt (Systemdienst)

Anwendung auf eine Multimedia-Anwendung II



Scheduling Server — Erfahrungen

- Alle Teile einer zeitkritischen Anwendung müssen unter Kontrolle des Scheduling Servers laufen
- Benutzung von Systemdiensten vermeiden, die auch von anderen Prozessen benutzt werden
- Synchronisation in einer Applikation muß mit Scheduling Server abgestimmt werden
- Parameter des Scheduling Servers (Scheduling-Periode und High/Low-Verhältnis) bestimmen Vorhersagbarkeit sehr und müssen an Applikation angepaßt werden

Streaming

- Kontinuierliche Übertragung von Datenströmen
- Verschiedene Klassen: Stored audio/video, live audio/video, interactive audio/video
- Streaming vs. progressives Herunterladen
- Verwendung von UDP-basierten Soft-RT Protokollen (keine automatische Flusskontrolle)
- Herausforderungen: Variierende Datenrate, Verzögerungen, Paketverlust

Streaming Protokolle

- Real-Time Transport Protocol (RTP) - Sequenznummern, Timestamps, keine Garantien
- Real-Time Control Protocol (RTCP) - Regelmässiger Austausch von Statistiken (Anzahl gesendeter Pakete, Paketverlust, Jitter, ...)
- Real-time Streaming Protocol (RTSP) - Session-Verwaltung für Video-Streaming, verwendet RTP
- Session Description Protocol (SDP) - Beschreibt Initialisierungsparameter (Codec, Datenrate, Dauer, ...), von RTSP verwendet