Dynamische Verwaltung Virtuellen Speichers für Echtzeitsysteme

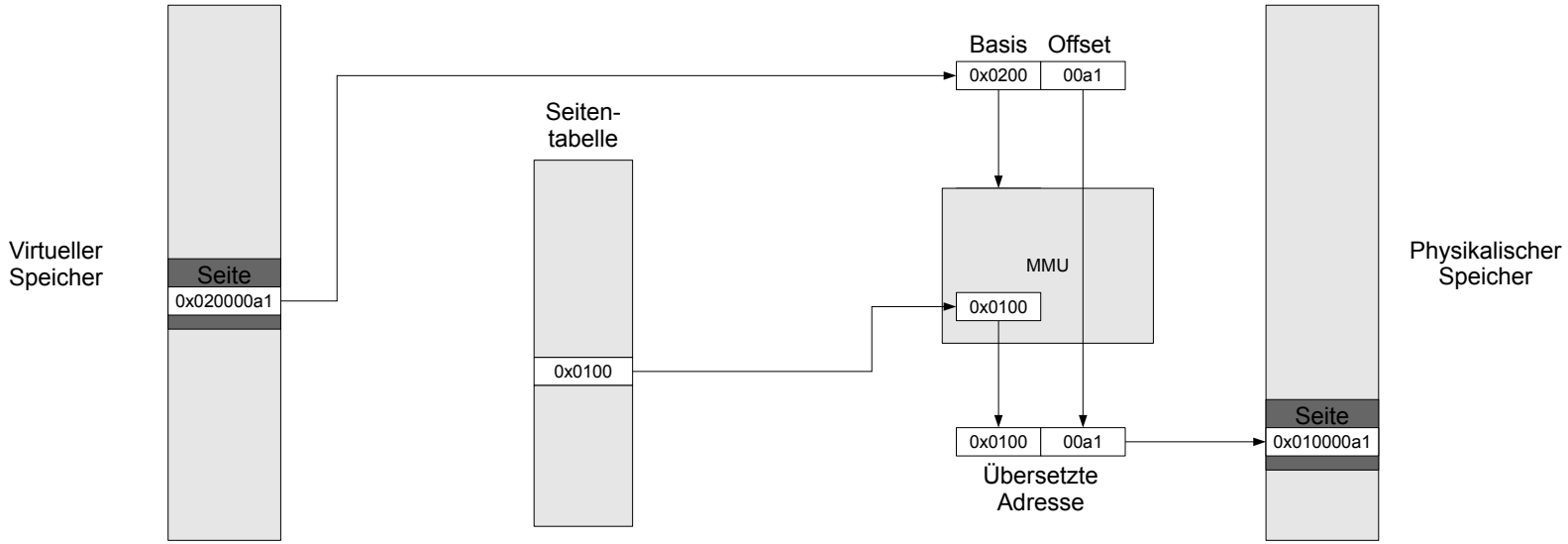
Martin Böhnert, Thorsten Zitterell, Christoph Scholl

Albert-Ludwigs-Universität Freiburg
Fakultät für Angewandte Wissenschaften
Institut für Informatik
Lehrstuhl für Betriebssysteme

28. November 2008

Einführung

- Virtueller Speicher für individuelle Adressräume
- Feste Einteilung der Adressräume in Seiten
- Abbildung von virtueller auf physikalische Speicherseite
- Übersetzung mittels Memory Management Unit (*MMU*)
- Gültigkeit und Zugriffsschutz durch Seitentabelle



Motivation

- Betriebssystem kümmert sich um Bereitstellung freier Seiten
- Implizite Freigabemechanismen werden genutzt
- Explizite Freigabe würde Laufzeit $O(n)$ haben
  (n Speicherseiten einer De-/Allokation)
- Virtuelle Speicherverwaltung in Echtzeitsystemen üblicherweise *nicht* verwendet
  - Schlechte zeitliche Abschätzbarkeit
  - $\Rightarrow$ Oft statische Speicherverwaltung

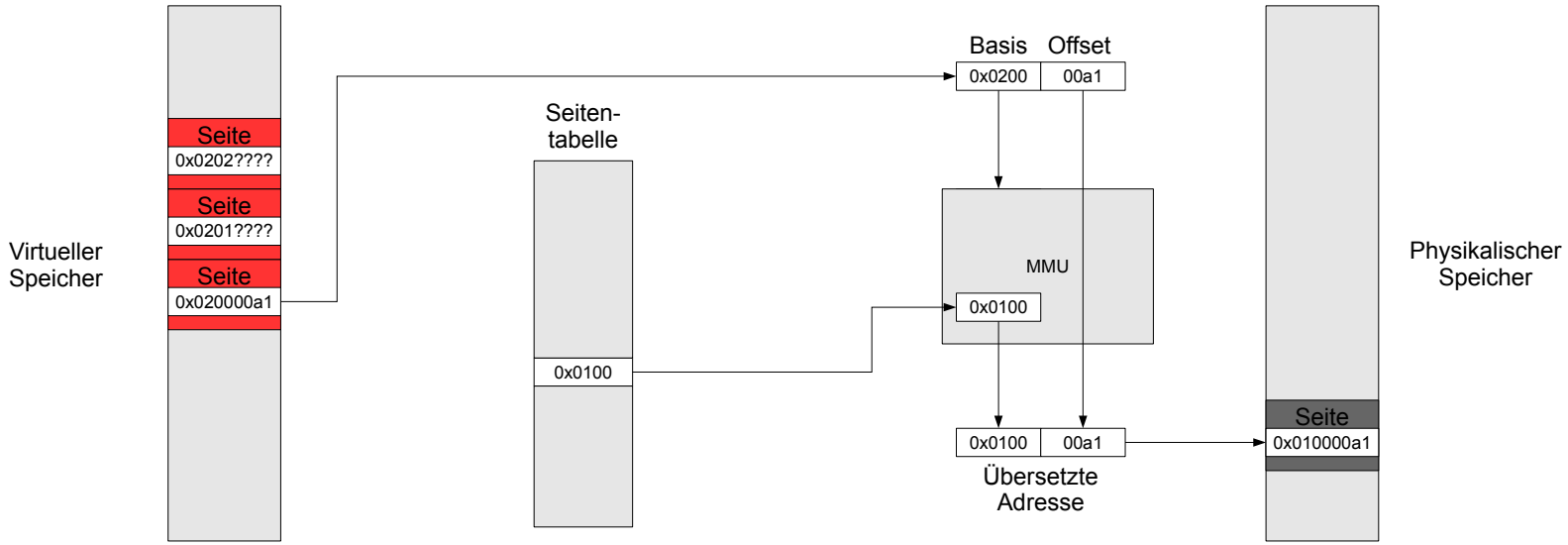
Motivation

- Betriebssystem kümmert sich um Bereitstellung freier Seiten
- Implizite Freigabemechanismen werden genutzt
- Explizite Freigabe würde Laufzeit $O(n)$ haben
  (n Speicherseiten einer De-/Allokation)
- Virtuelle Speicherverwaltung in Echtzeitsystemen üblicherweise *nicht* verwendet
  - Schlechte zeitliche Abschätzbarkeit
  - $\Rightarrow$ Oft statische Speicherverwaltung

- *Ziel:* Programmierung vereinfachen durch dynamischen und virtuellen Speicher auch in *Echtzeitsystemen*

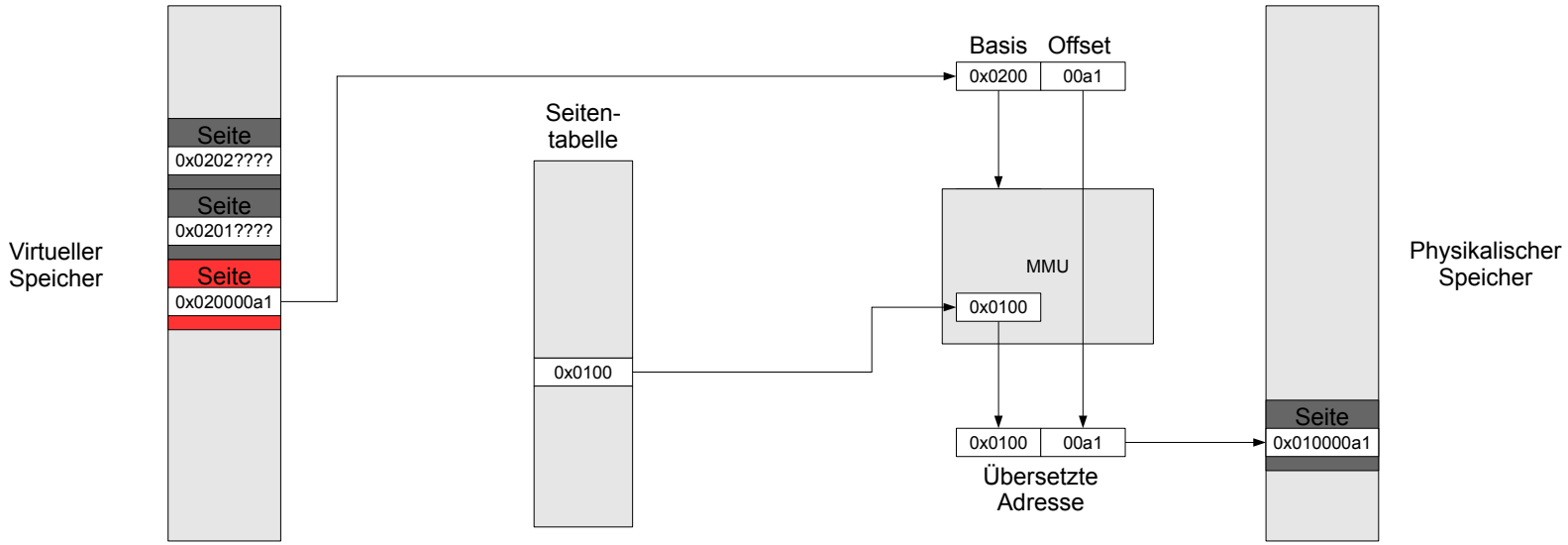
Unser Ansatz

- Dynamische Verwaltung virtuellen Speichers für Echtzeitsysteme
  - Jeder Task in isoliertem Adressraum
  - Tasks können zur Laufzeit Speicher anfordern und zurückgeben
- Laufzeit (quasi-)konstant
  - $O(\log \log m)$ mit Anzahl m virtueller Speicherseiten im System
  - Anzahl Speicherseiten fest in realem System
- Verbesserung der zeitlichen Abschätzbarkeit



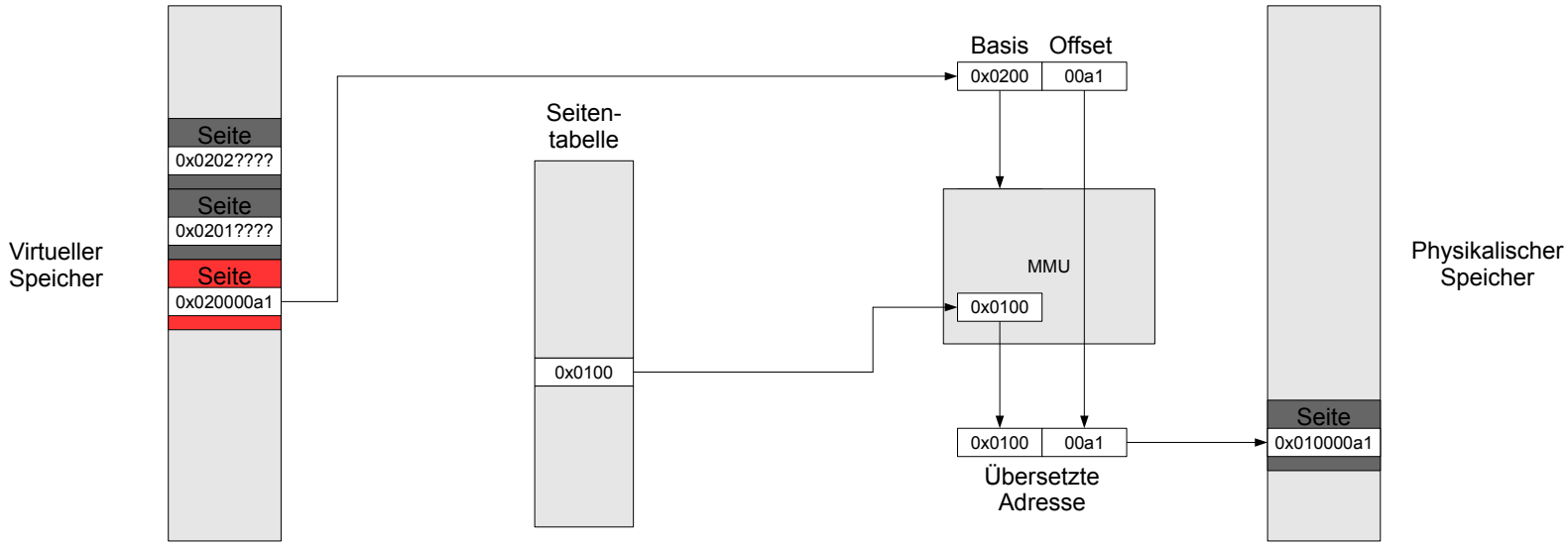
GRUNDFUNKTIONEN

- `vmalloc` (A_i, a, n)
  - Reservierung von n Speicherseiten ab Speicherseite a (Speicherregion)
- `vfree` (A_i, a)
  - Freigabe von Speicherregion a
- `vmap` (A_i, b)
  - Speicherseite b zuordnen



Allokation von Seiten

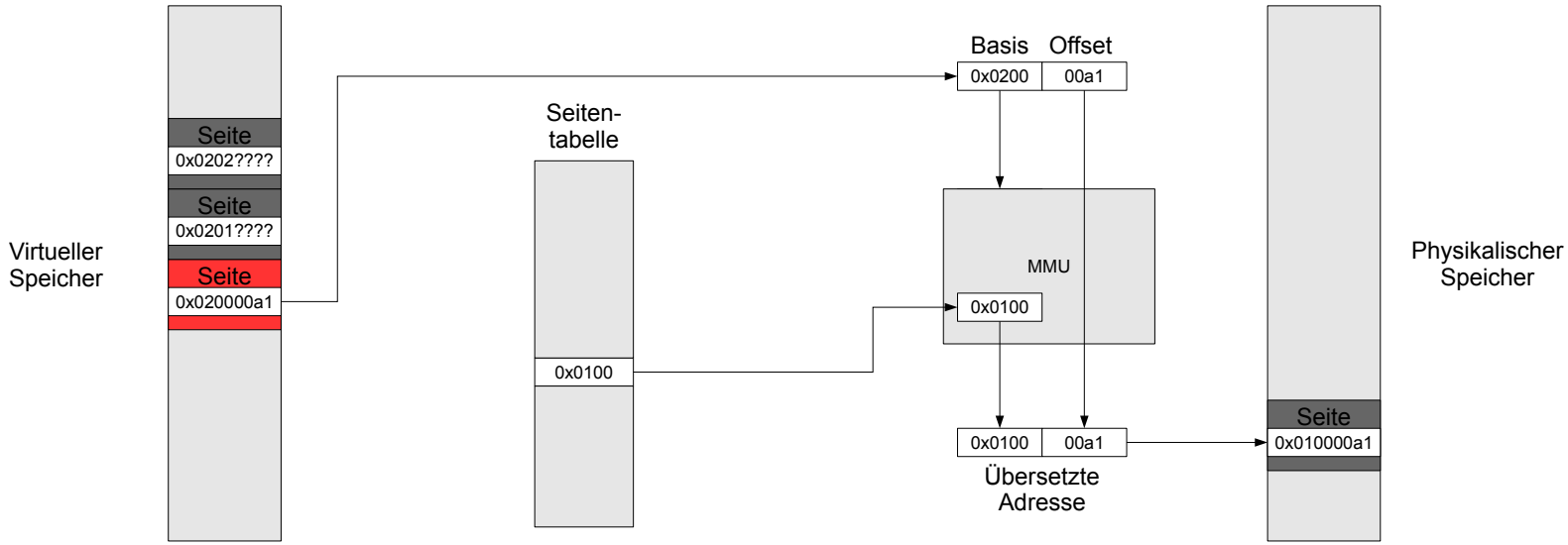
- Allokation mittels `vmalloc` (A_i, a, n)
- Belegen aller n Speicherseiten
  - Freie Seiten aus Pool entnehmen und mappen
  - $\Rightarrow$ Laufzeit $O(n)$, Speicherverschwendung(!)
  - $\Rightarrow$ Seiten erst bei Erstzugriff belegen



Allokation von Seiten

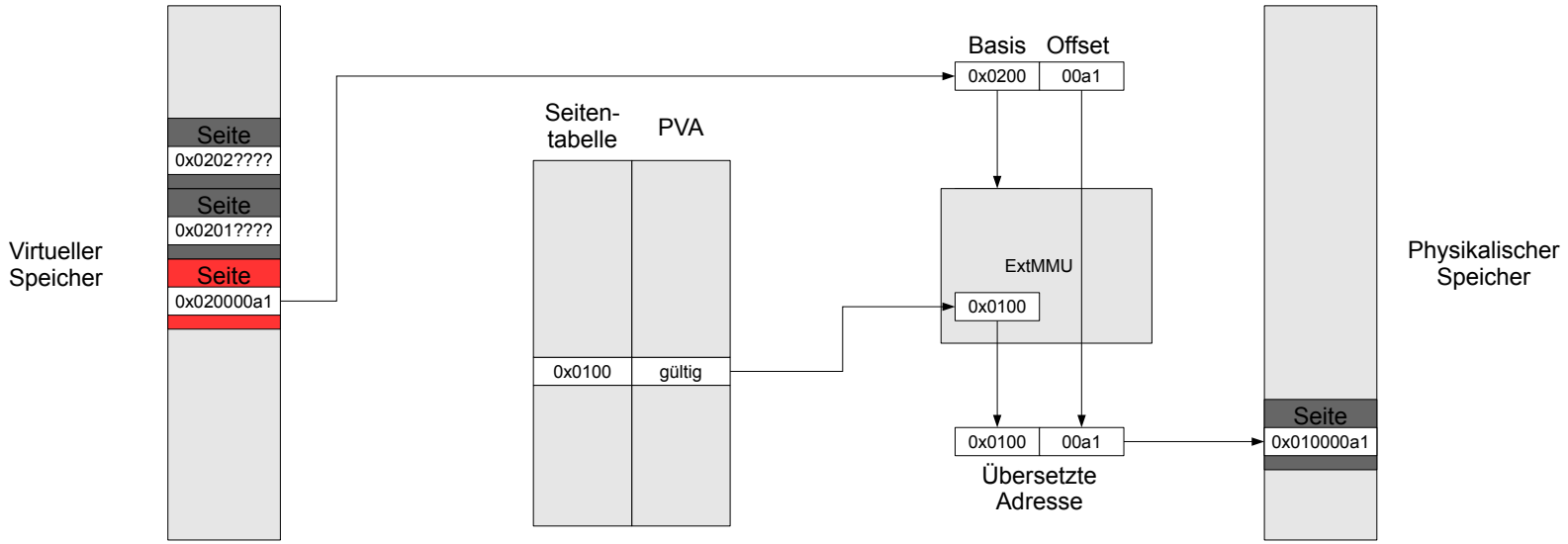
- Allokation mittels `vmalloc` (A_i, a, n)
- Belegen aller n Speicherseiten
  - Freie Seiten aus Pool entnehmen und mappen
  - $\Rightarrow$ Laufzeit $O(n)$, Speicherverschwendung(!)
  - $\Rightarrow$ Seiten erst bei Erstzugriff belegen

- Seite a wird direkt belegt
  - Speichern von Verwaltunginformationen (*Virtual Region Header*)



SPEICHERZUGRIFFE

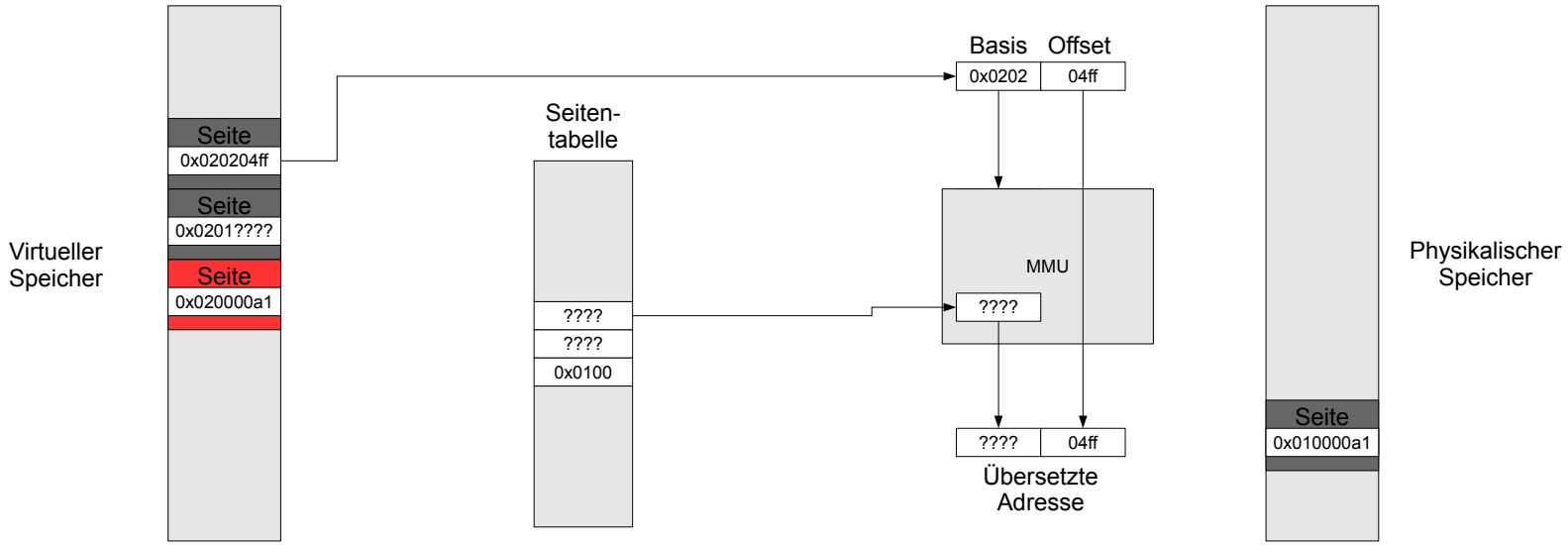
- Übersetzung von Speicheradressen transparent in MMU



Speicherzugriffe

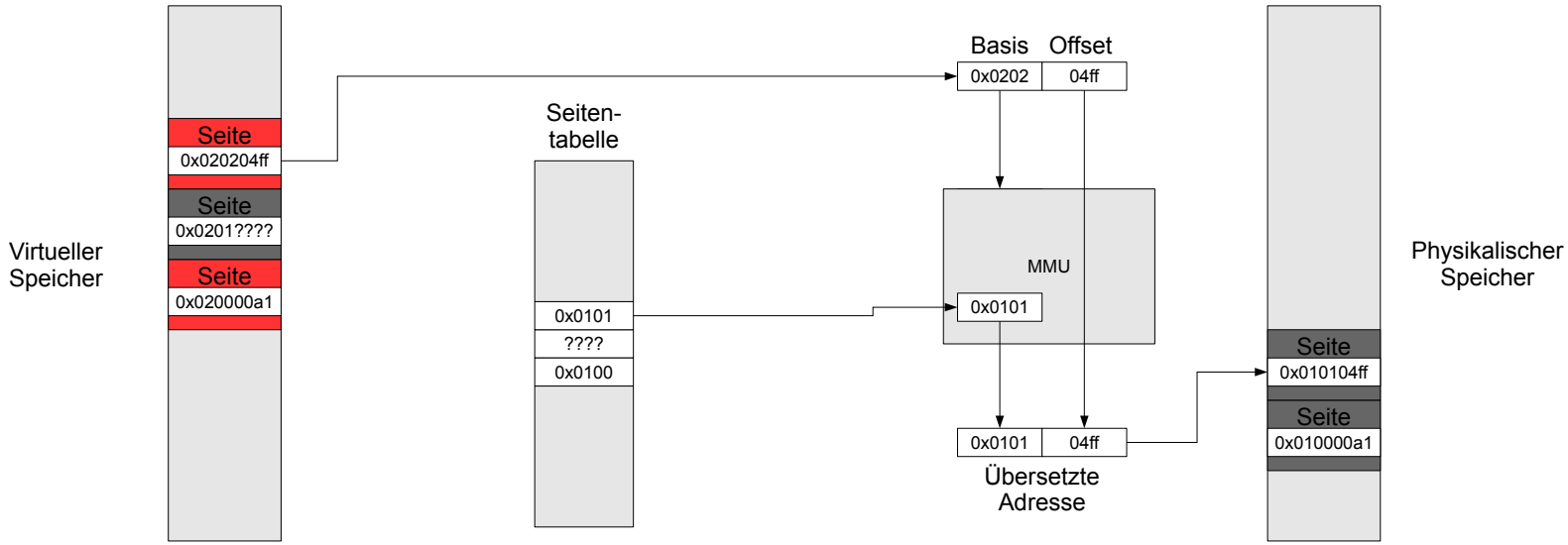
- Übersetzung von Speicheradressen transparent in MMU

- *Extended MMU* prüft bei Zugriff auf Seite den Seitentabelleneintrag auf Gültigkeit
- Überprüfung mittels *Pages Validity Array*
  - Indikator ob Speicherseite gültig ist
  - Neu-/Initialisierung des Arrays in $O(1)$ für Freigabe wichtig
  - $\Rightarrow$ initialisierungsfreies Array



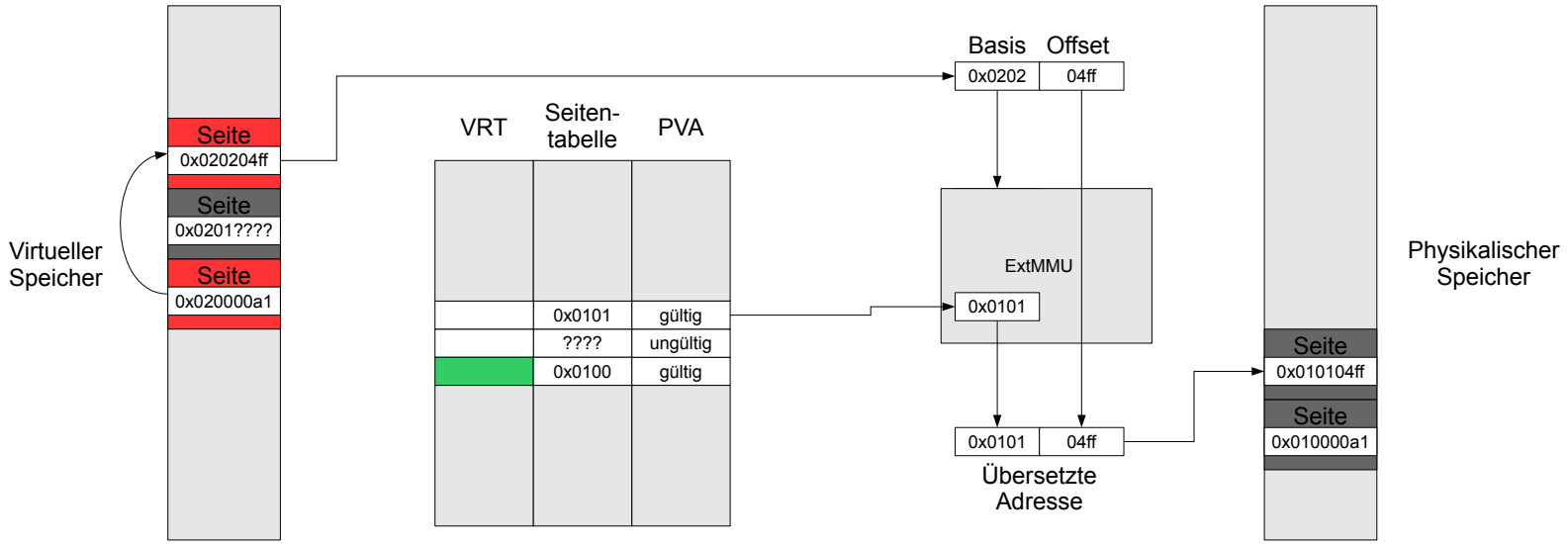
SEITENFEHLER

- Seite muss gemappt werden (`vmap` (A_i, b))
- Freie physikalische Seite aus Pool nehmen
- Eintrag in Seitentabelle schreiben



SEITENFEHLER

- Seite muss gemappt werden (`vmap` (A_i, b))
- Freie physikalische Seite aus Pool nehmen
- Eintrag in Seitentabelle schreiben



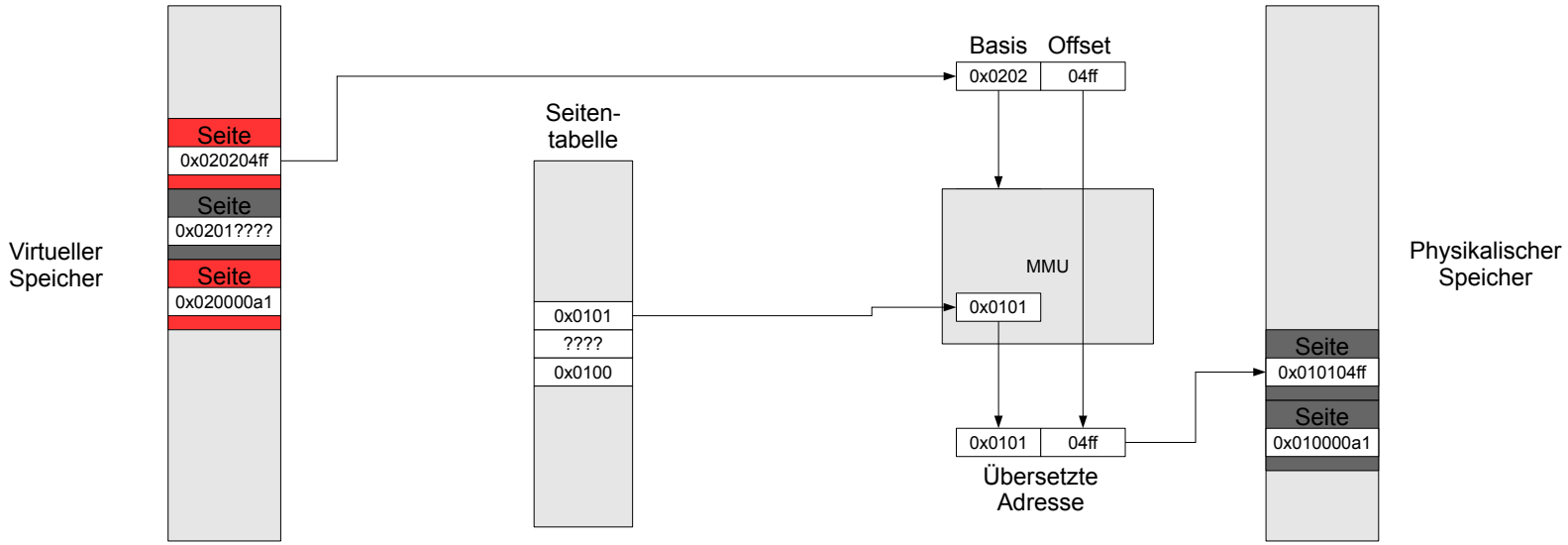
SEITENFEHLER

- Seite muss gemappt werden (`vmap` (A_i, b))
- Freie physikalische Seite aus Pool nehmen
- Eintrag in Seitentabelle schreiben

- Freie Seite im *Virtual Region Header* verketten
  - *Virtual Region Header* wird mittels *Virtual Regions Tree* gesucht
- Seite im *Pages Validity Array* als gültig markieren

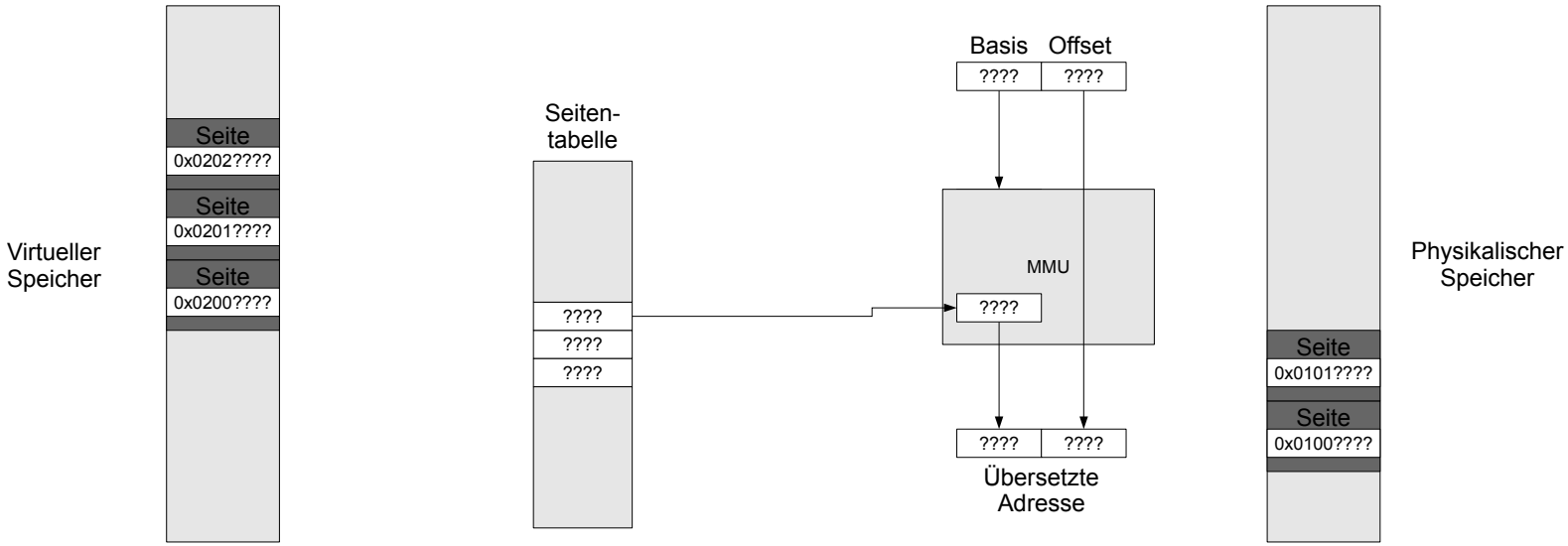
Virtual Regions Tree

- Markierung des Anfangs einer Speicherregion
- Auffinden des zugehörigen *Virtual Region Header*
- Basierend auf geschichteten Baum
  - `union()`-, `find()`-, `split()`-Operationen auf Baum in $O(\log \log m)$ mit m als Anzahl der Seiten im virtuellen Adressraum
- Baum statisch
  - Zeiger innerhalb des Baumes implizit
  - $\Rightarrow$ Implementierung vereinfacht



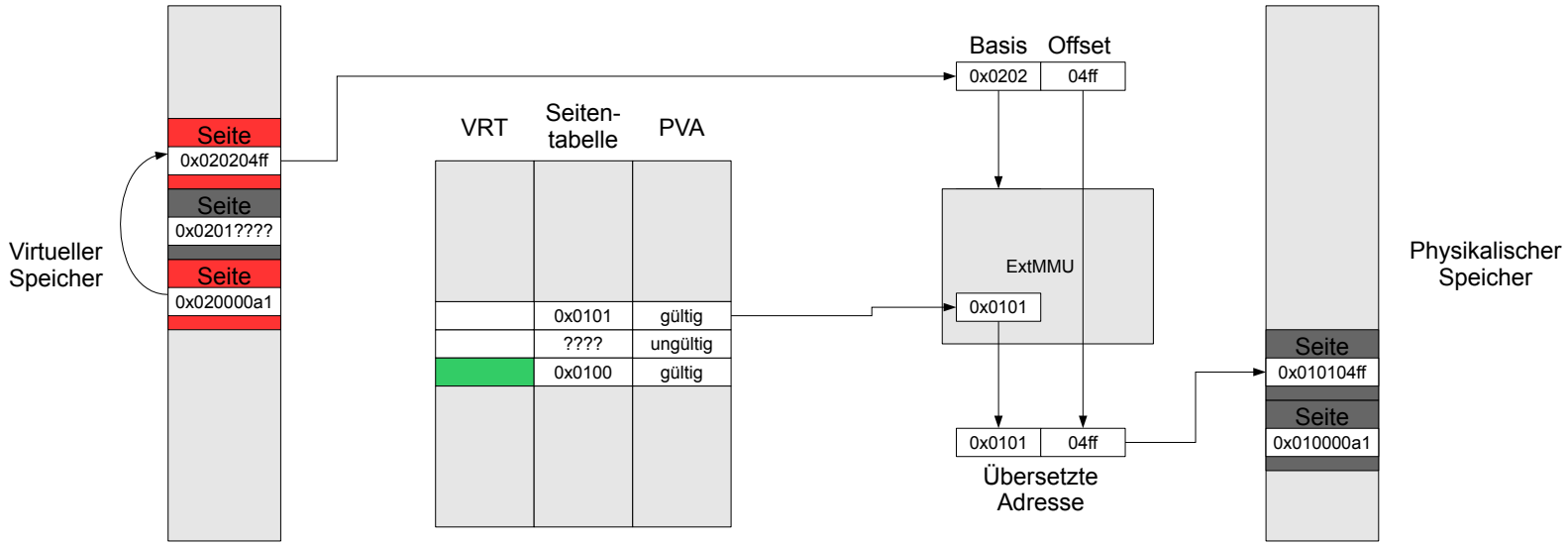
FREIGABE

- Speicherseiten an System zurückgeben (`vfree`(A_i, a))



FREIGABE

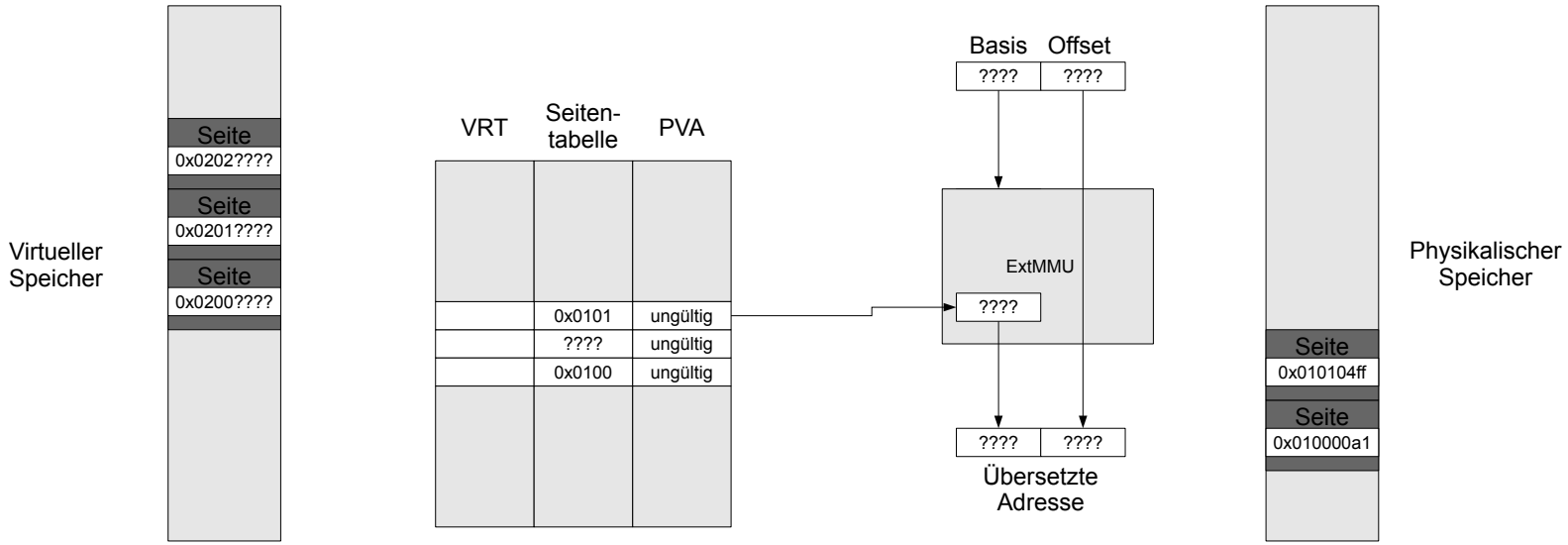
- Speicherseiten an System zurückgeben (`vfree(`A_i, a`)`)



Freigabe

- Speicherseiten an System zurückgeben (`vfree`(A_i, a))

- Speicherbereich im *Pages Validity Array* als ungültig markieren
  - $O(1)$ durch initialisierungsfreies Array
- Benutzte Speicherseiten der Region in Pool einhängen
  - Liste im *Virtual Region Header* wird in Liste freier Speicherseiten eingehangen ($O(1)$)
- Markierung im *Virtual Regions Tree* löschen ($O(\log \log m)$)



Freigabe

- Speicherseiten an System zurückgeben (`vfree`(A_i, a))

- Speicherbereich im *Pages Validity Array* als ungültig markieren
  - $O(1)$ durch initialisierungsfreies Array
- Benutzte Speicherseiten der Region in Pool einhängen
  - Liste im *Virtual Region Header* wird in Liste freier Speicherseiten eingehangen ($O(1)$)
- Markierung im *Virtual Regions Tree* löschen ($O(\log \log m)$)

Zwischenbilanz

- Bisher reine virtuelle Speicherverwaltung
  - Größe der Speicherstücken nur Vielfache der Seitengröße
- Laufzeit von $O(\log \log m)$ über die virtuelle Adressraumgröße m im Gegensatz zu $O(n)$ in Bezug auf Größe der Speicherregion n in klassischen Allokatoren
- Deallokation ist explizit
  - Keine Verschwendung von Speicherplatz
  - Laufzeitanalyse wird vereinfacht
- Nicht behandelt: Kompakte Darstellung der Datenstrukturen

Arbiträre Speicherblockgrössen

- Anwendungsebene nutzt `malloc()` und `free()` für beliebige Speicherbereichsgrößen
- Virtuelle Speicherverwaltung arbeitet transparent im OS mit `vmalloc()` und `vfree()`
- TLSF-Allokator wurde dafür angepasst
- $O(\log \log m)$ wird dabei gehalten

Arbiträre Speicherblockgrössen

- Anwendungsebene nutzt `malloc()` und `free()` für beliebige Speicherbereichsgrößen
- Virtuelle Speicherverwaltung arbeitet transparent im OS mit `vmalloc()` und `vfree()`
- TLSF-Allokator wurde dafür angepasst
- $O(\log \log m)$ wird dabei gehalten

TLSF-Allokator

- Speicherallokator für beliebig große Stücke phys. Speichers
- Komplexität von $O(1)$
- Entwickelt zur Nutzung in Systemen ohne MMU
- Verwaltungsstrukturen im Nutzspeicher von allokierten Blöcken

Arbiträre Speicherblockgrössen

- Anwendungsebene nutzt `malloc()` und `free()` für beliebige Speicherbereichsgrößen
- Virtuelle Speicherverwaltung arbeitet transparent im OS mit `vmalloc()` und `vfree()`
- TLSF-Allokator wurde dafür angepasst
- $O(\log \log m)$ wird dabei gehalten

TLSF-Allokator

- Speicherallokator für beliebig große Stücke phys. Speichers
- Komplexität von $O(1)$
- Entwickelt zur Nutzung in Systemen ohne MMU
- Verwaltungsstrukturen im Nutzspeicher von allokierten Blöcken

- Mechanismus notwendig um virt. Seiten dynamisch zu mappen in denen Verwaltungsstrukturen bzw. Nutzdaten liegen

Situation

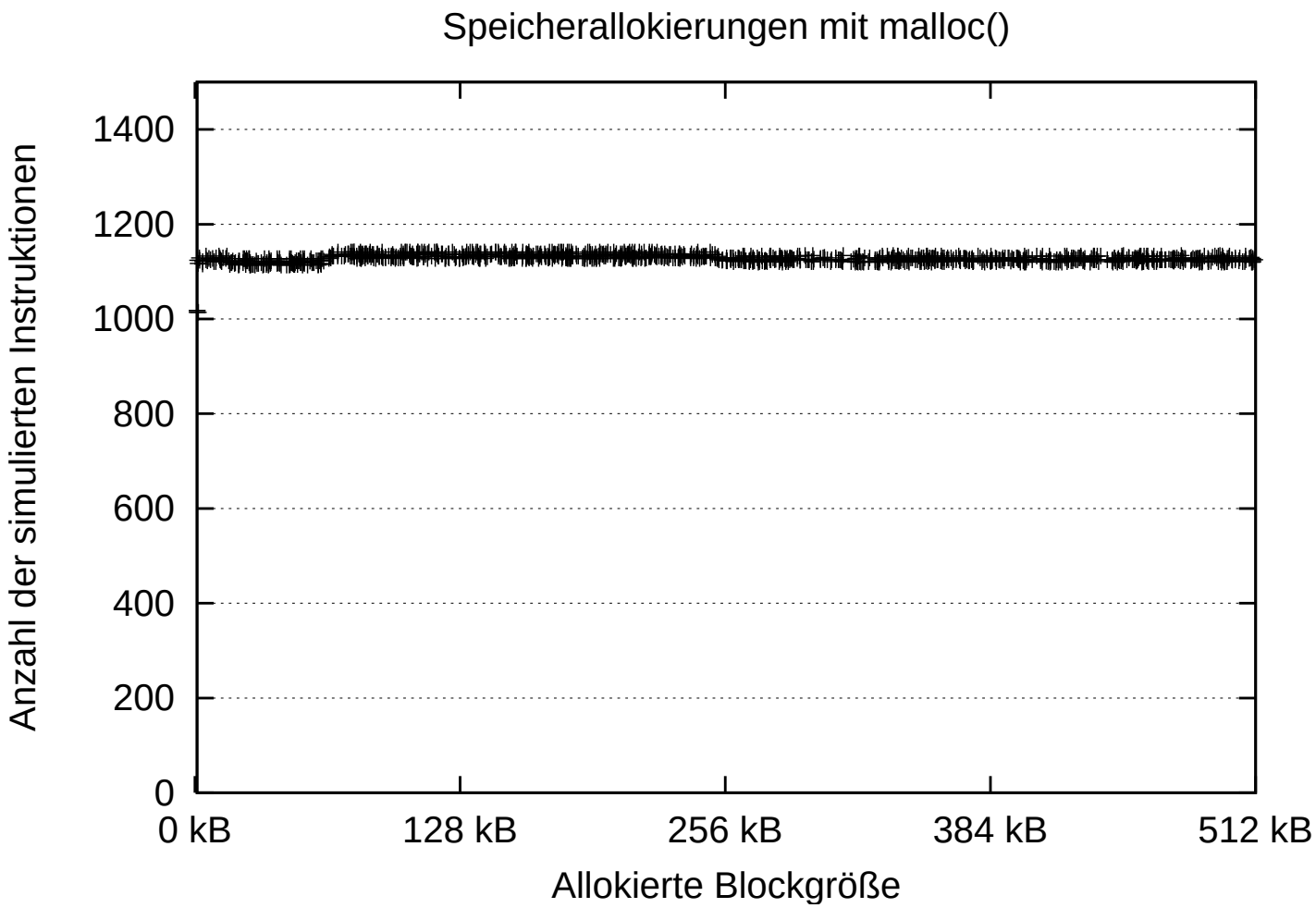
- Reale Hardware für *ExtMMU* nicht vorhanden
- Implementierung einer Bibliothek in C
- Simulation von *ExtMMU* und Allokator
- Angepasster TLSF-Allokator

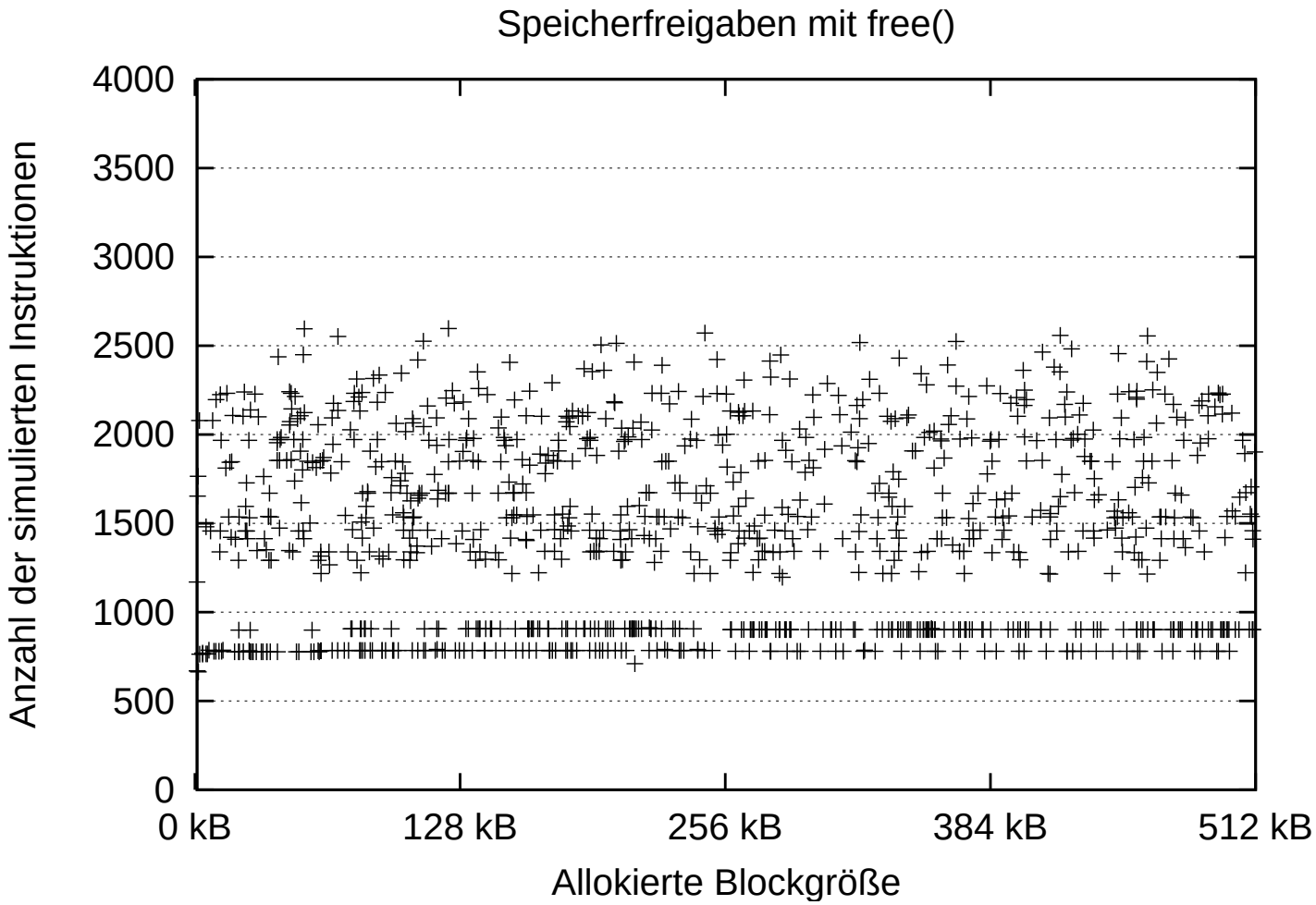
Situation

- Reale Hardware für *ExtMMU* nicht vorhanden
- Implementierung einer Bibliothek in C
- Simulation von *ExtMMU* und Allokator
- Angepasster TLSF-Allokator

Messmethode

- 1000 Allokationen
- Speicherstücke zwischen 16 Bytes (2^4) und 512 kBytes (2^{19})
- Anzahl der Instruktionen abhängig von Allokationsgröße gemessen


Speicherallokierungen mit malloc()
Anzahl der simulierten Instruktionen
1400
1200
1000
800
600
400
200
0
0 kB
128 kB
256 kB
384 kB
512 kB
Allokierte Blockgröße


Speicherfreigaben mit free()
Anzahl der simulierten Instruktionen
4000
3500
3000
2500
2000
1500
1000
500
0
0 kB
128 kB
256 kB
384 kB
512 kB
Allokierte Blockgröße

Fazit

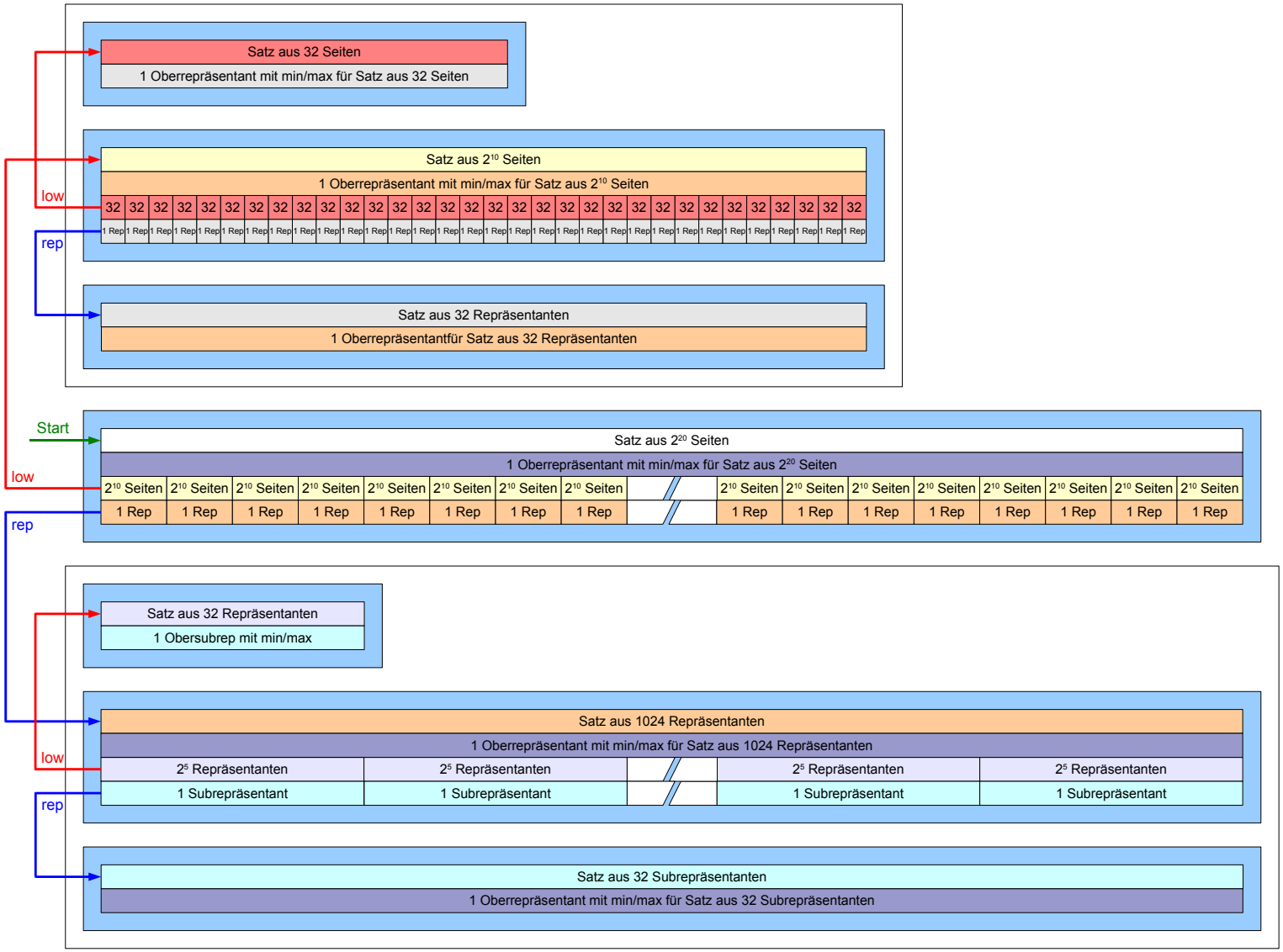
- Neues Verfahren für Virtuelle Speicherverwaltung
- Alle Operation in $O(\log \log m)$ über die Größe m des virtuellen Adressbereichs
- Zeitaufwand unabhängig von allokierter Speicherregion
- Speicheraufwand nur in konstantem Faktor größer (in Bezug auf Seitentabelle)
- Gute Abschätzbarkeit von Laufzeit und Speicherbedarf
- Gute Vorhersagbarkeit in Echtzeitbetriebssystemen durch expliziten Freigabemechanismus

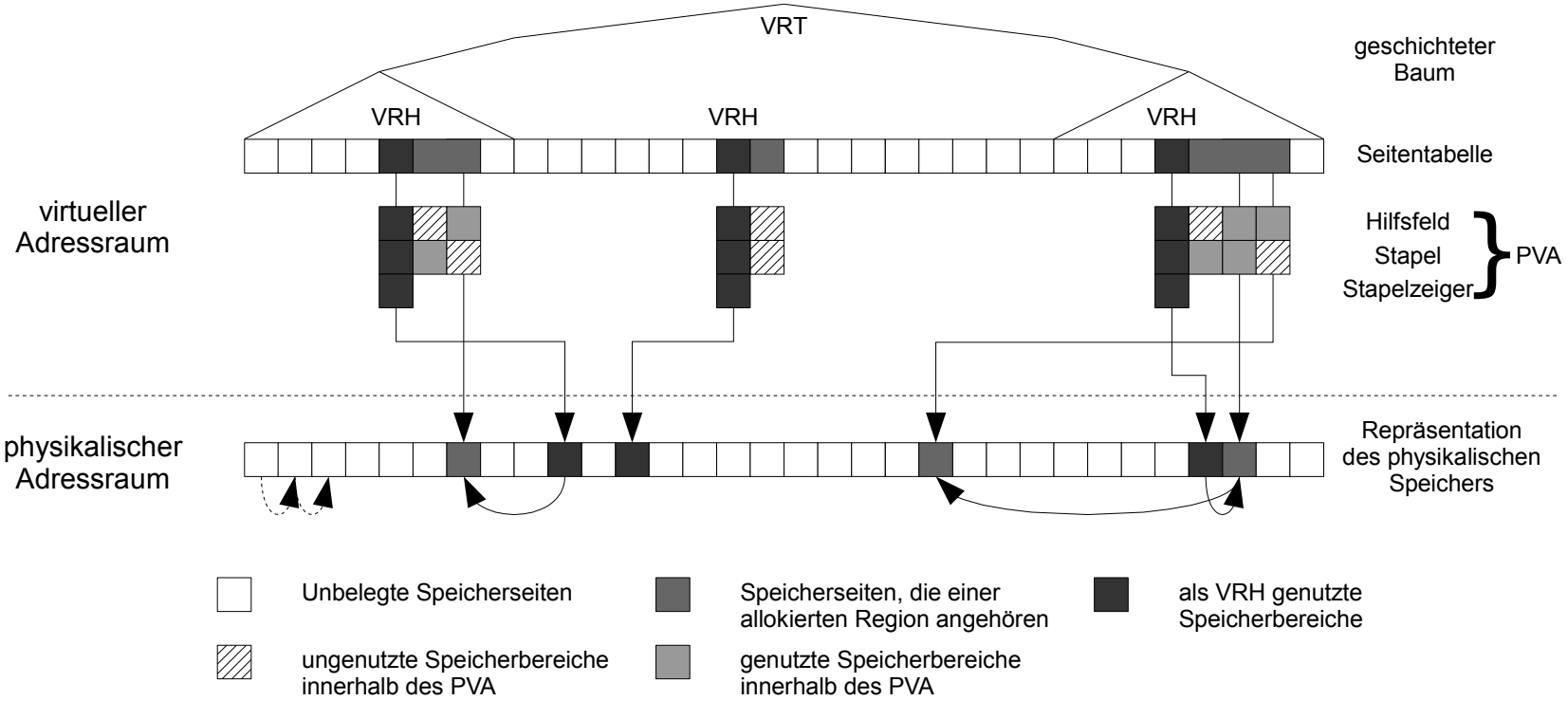
Fazit

- Neues Verfahren für Virtuelle Speicherverwaltung
- Alle Operation in $O(\log \log m)$ über die Größe m des virtuellen Adressbereichs
- Zeitaufwand unabhängig von allokierter Speicherregion
- Speicheraufwand nur in konstantem Faktor größer (in Bezug auf Seitentabelle)
- Gute Abschätzbarkeit von Laufzeit und Speicherbedarf
- Gute Vorhersagbarkeit in Echtzeitbetriebssystemen durch expliziten Freigabemechanismus

Fortführende Arbeiten

- Realisierung der *ExtMMU* in Hardware
- Forschungskernel um Allokator und Behandlung der *ExtMMU* erweitern
- Beschleunigung und Optimierung des Allokators


Satz aus 32 Seiten
1 Oberrepräsentant mit min/max für Satz aus 32 Seiten
Satz aus 2^{10} Seiten
1 Oberrepräsentant mit min/max für Satz aus 2^{10} Seiten
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
1 Rep 1 Rep
low
rep
Satz aus 32 Repräsentanten
1 Oberrepräsentantfür Satz aus 32 Repräsentanten
Start
low
rep
Satz aus 2^{20} Seiten
1 Oberrepräsentant mit min/max für Satz aus 2^{20} Seiten
2^{10} Seiten
1 Rep
Satz aus 32 Repräsentanten
1 Obersubrep mit min/max
low
rep
Satz aus 1024 Repräsentanten
1 Oberrepräsentant mit min/max für Satz aus 1024 Repräsentanten
2^5 Repräsentanten
1 Subrepräsentant
Satz aus 32 Subrepräsentanten
1 Oberrepräsentant mit min/max für Satz aus 32 Subrepräsentanten


VRT
geschichteter Baum
VRH
VRH
VRH
Seitentabelle
virtueller Adressraum
Hilfsfeld
Stapel
Stapelzeiger
PVA
physikalischer Adressraum
Repräsentation des physikalischen Speichers
Unbelegte Speicherseiten
Speicherseiten, die einer allokierten Region angehören
als VRH genutzte Speicherbereiche
ungenutzte Speicherbereiche innerhalb des PVA
genutzte Speicherbereiche innerhalb des PVA

Systemeigenschaften

- 32 Bit-Architektur
- 4 KB Seitengröße
- 4 GB adressierbarer Speicher
- 4 GB physikalisch vorhandener Speicher

Systemeigenschaften

- 32 Bit-Architektur
- 4 KB Seitengröße
- 4 GB adressierbarer Speicher
- 4 GB physikalisch vorhandener Speicher

Overhead

- *Virtual Regions Tree* kompaktiert: 137 KB
- *Pages Validity Array* kompaktiert:
  - 4096 KB je Task
  - 8192 KB systemweit
- Gesamtverbrauch rund 4233 KB je Task
  einmalig 8192 KB systemweit