Studienjahresarbeit

Modell zur Transparenz der Verfügbarkeit von Hardware-Ressourcen auf der Systembusebene

Martin Stein
Dresden den 7. Dezember 2011

Eingereicht bei Prof. Dr.-Ing. habil. Armin Zimmermann
Institut für Technische Informatik und Ingenieurinformatik
Technische Universität Ilmenau

Betreuer
Dr.-Ing. Norman Feske
Prof. Dr. Volker Zerbe

Besonderer Dank gilt
Anja und Michael
die mir zur Seite standen.

Inhaltsverzeichnis

1 Einleitung

Mit der zunehmenden Optimierung der Fertigungsverfahren in der Digitaltechnik wächst auch stetig die Komplexität der damit umsetzbaren Hardware-Systeme. Im Jahr 2001 vereinte zum Beispiel ein Intel Itanium Mikroprozessor etwa 25 Millionen Transistoren, bereits 9 Jahre später hatte sich dieser Wert beim Nachfolgemodell Intel Itanium 2 Tukwila auf ca. 2 Milliarden verachtzigfacht. Dieser Vergleich ist zwar oberflächlich, verdeutlicht aber dennoch die Situation in welcher sich die Entwicklung darauf aufbauender Produkte befindet. Die stärksten Auswirkungen hat dieser Umstand auf die Entwicklung eingebetteter Systeme beziehungsweise die Enwticklung von Treibersoftware in modularen Rechnersystemen. Hier laufen die Entwicklung von Hardware und Software kombiniert ab, sie beinflussen sich in Umfang und Funktionsweise also gegenseitig. Um solche Systeme bei wachsender Komplexität und in gleichbleibender Zeit zu entwickeln und dabei den Anforderungen an Ausfallsicherheit und Fehlerfreiheit gerecht zu werden, ist es nötig, Spezifikation und Implementierung beidseitig möglichst früh testen zu können. Diese Arbeit verfolgt zu diesem Zweck einen Ansatz, welcher die Emulation von Gerätespezifikationen auf der Basis moderner Betriebssystemkonzepte ermöglicht. Das Software-System, welches das Gerät nutzt, kann in dem Modell, transparent zur physischen Verfügbarkeit des Geräts, auf Kommunikationsmittel der Systembusebene aufsetzen. Die Implementierung des Emulators beschränkt sich, im Gegensatz zu anderen Software-Emulationen, auf die Spezifikation des Geräts hinter einer intuitiven und generischen Schnittstelle, welche nur auf Betriebssystemmechanismen aufsetzt. Die Emulation kann bei modularen Rechnersystemen somit bereits in der Umgebung des Zielsystems betrieben werden. Zur effizienten Fehlersuche bietet die Schnittstelle zwischen emulierter Hardware und dem Softwaresystem umfangreiche Möglichkeiten, die entsprechende Kommunikation zu beobachten und zu steuern. Zudem hält das Konzept die Vertrauensbasis des Software-Systems auch im Emulationsfall zweckorientiert. Das Vertrauen in den Emulator, welches durch das Software-System mit dessen Nutzung impliziert wird, bleibt das selbe, welches auch in die physische Umsetzung des Geräts gesetzt würde. Außerdem lässt sich das vorgestellte Modell einfach in das Zielsystem integrieren, wodurch über rein funktionale Tests hinaus auch die Anwendung realitätsnaher Szenarien und Extremsituationen ermöglicht wird.

Die Arbeit ist wie folgt aufgebaut. In Kapitel 2 werde ich einen Überblick über die Problematik der Transparenz von Ressourcen-Verfügbarkeit geben. Es werden heutige Ansätze zur Transparenz von Verfügbarkeit auf verschiedenen Abstraktionsebenen dargestellt. Daraus leitet sich schließlich die Motivation zu dieser Arbeit ab. Kapitel 3 erläutert dann die Funktionsweise des Modells welches ich während dieser Arbeit entwickelt habe. In Kapitel 4 werde ich dann eine beispielhafte Implementierung auf Basis des Genode Betriebssystems darstellen. Kapitel 5 behandelt schließlich die Eigenschaften des Modells. Es werden potentielle Anwendungsbereiche und Ausbaumöglichkeiten agezeigt. Abschließend möchte ich zu einigen ausstehenden Herausforderungen bezügich des Modells motivieren.

2 Grundlagen

Die kommenden Abschnitte bieten einen Überblick über bekannte Ansätze zur Emulation von Hardware-Ressourcen. Bei der Integration von Hardware-Ressourcen betrachte ich die folgenden Abstraktionsebenen:

- *Systembusebene*
  Kommunikation mit dem entsprechenden Gerät, durch synchrone Schreib- und Leseoperationen auf seinen Input-Output-, kurz I/O-Registern. Dies wird durch den Systembus zum Beispiel als memory-mapped I/O, kurz MMIO umgesetzt. Behandlung von asynchronen Ereignissen wie Interrupts, welche vom Geräte ausgelöst werden.
- *Prozesslokale Funktionsaufrufe*[1]
  Nutzung prozesslokal verfügbarer Ressourcen über die abstrakte Schnittstelle einer Funktion deren Implementierung mittels Austausch von Parametern und Rückgabewerten kommuniziert.
- *Prozesse und Interprozesskommunikation*
  Nutzung von Ressourcen zu denen ein anderer Prozess Zugriff besitzt, welcher diese über eine Schnittstelle der Interprozesskommunikation zur Verfügung stellt.

2.1 Systembusebene

2.1.1 Grundlagen

Es wird von der physischen Umsetzung der Funktionalitäten eines Gerätes, auf Operationen des Hauptprozessors, kurz CPU für *Central Processing Unit*, abstrahiert. Üblich sind dabei die beiden Ansätze memory-mapped I/O sowie port-mapped I/O, auch Isolated I/O genannt. Bei memory-mapped I/O wird der Zugriff auf die Geräte-IO-Pins über den Adressraum des Systembus integriert. Der Zugriff erfolgt dann mit den selben Instruktionen, welche auch für andere Speicherzugriffe genutzt werden. Port-mapped I/O hingegen verfolgt das Konzept, diese in einem isoliertem Adressraum, über dedizierte CPU-Instruktionen anzusprechen. Durch die Anleihe der vorhandener CPU-Instruktionen für Speicherzugriffe und deren Adressierungsart, integriert sich MMIO in die Methodik imperativer und objektorientierter Hochsprachen, und vereinfacht den Prozessoraufbau gegenüber PMIO. PMIO hingegen kann insbesondere bei Systemen mit ohnehin eingeschränkter Breite des Adressbus vorteilhaft sein.

Desweiteren ist das Konzept der Interruptports verbreitet um, zusätzlich zur synchronen Kommunikation über MMIO oder PMIO, asynchrone Kommunikation effizient umzusetzen. Statt Software-seitig stets auf Ereignisse an entsprechenden I/O-Adressen zu prüfen, kann ein Gerät, über das Setzen eines Interrupt-Signals am Hauptprozessor, diesen zur Unterbrechung der aktuellen Ausführung

[1] In dieser Arbeit ist mit einem Prozess, wenn nicht anders erwähnt, immer ein Software-Prozess auf Basis des entsprechenden Betriebssystems gemeint.

anregen. Hardware-seitig wird daraufhin in der Ausführung zu einer, vom Entwickler vorgesehenen Interrupt-Behandlung gesprungen. Dabei muss von der CPU die vollständige Sicherung des unterbrochenen Ausführungskontextes zur späteren Wiederaufnahme ermöglicht werden.

Ich betrachte desweiteren Geräte, welche der Nutzer auf der Systembusebene über eine Schnittstelle aus MMIO und Interrupts wie erläutert anspricht. Ein Ansatz, die Verfügbarkeit solcher Geräte für den Nutzer auf der Systembusebene transparent zu gestalten, besteht darin, das gewünschte Verhalten hinter der Schnittstelle zu emulieren. Das Verhalten wird dazu durch einen Emulator unter Einsatz anderer, meist weniger spezialisierter Systemkomponenten nachgebildet. Konkrete Umsetzungen solcher Emulatoren existieren sowohl hardware- als auch Software-seitig [7].

2.1.2 Hardware-Emulation mittels FPGA

Eine weit verbreitete Art der Hardwareemulation ist die Umsetzung durch wiederprogrammierbare Logikgatter-Chips, im englischen *Field Programmable Gate Arrays* oder kurz FPGAs. Ein FPGA besteht aus verhältnismäßig einfachen Logikbausteinen und Speicherzellen von hoher Stückzahl. Ihr jeweiliges Verhalten kann über eine nicht-statische Konfiguration programmiert werden. Die Verschaltung der Logikbausteine untereinander ist ebenfalls programmierbar. Die Flexibilität dieser konfigurierbaren Verschaltung, sowie die Anzahl und Art der Logikbausteine variiert je nach FPGA. Kommunikation über die Chipgrenzen hinaus geschieht in einem FPGA über die Verschaltung von Logikblöcken mit den I/O-Pins direkt, oder speziellen Logikblöcken, welche den jeweiligen I/O-Signalfluss verwalten. [9]

Ausgangspunkt für die Emulation durch FPGAs ist im Allgemeinen die plattformunabhängige Beschreibung der gewünschten Funktionalität in einer Hardwarebeschreibungssprache wie VHDL oder Verilog. Mittels entsprechender Synthesesoftware wird diese dann in eine FPGA-spezifische Konfiguration der Logikbausteine und Verschaltung dekompositioniert. Die so gewonnnene Konfiguration kann dann auf die einbezogenen Komponenten des FPGAs geladen werden und über dessen I/O-Pins die gewünschte Schnittstelle emulieren. [11] und [10] beschreiben diesen Vorgang anhand der Xilinx Integrated Software Enviroment, kurz ISE, und dem Xilinx Spartan 3 FPGA.

Ein Problem von FPGAs ist der erhöhte Gatterbedarf für Logik gegenüber der nativen Umsetzung als anwendungspezisch integrierte Schaltung. Letztere werden auch als *Application-specific Integrated Circuit* oder kurz ASIC bezeichnet. Je nach Aufbau des FPGAs und Effizienz der Konfigrationssynthese, resultiert dies laut [12, S. 209-211] in einem 10 bis 70 mal höherem Flächenbedarf, bei gleicher physischer Gatterdichte. Damit schränkt die Größe des FPGAs die Komplexität der umsetzbaren Funktionalität maßgeblich ein. Als Antwort darauf werden davon betroffene Designs partitioniert und auf mehrere, entsprechend miteinander verbundene FPGAs verteilt. In dem resultierenden Chipnetz fällt die Kommunikation zwischen den FPGAs, zusätzlich zu der eigentlichen Kommunikation mit der Außenwelt, auf die I/O-Pins ab. Dies stößt, da I/O-Pins im Allgemeinen eine kritische Ressource von FPGAs darstellen, an die Gren-

zen des verfügbaren Durchsatzes. Folge dessen ist dass die durchschnittliche Flächenausnutzung der einzelnen Chips drastisch sinkt, da nicht genug I/O-Pins zur Speisung der eigentlich umzusetzenden Logik bleiben [1, S. 8].

Bezüglich der Berechnungsgeschwindigkeit sind FPGA-Umsetzungen, im Bereich heutiger Emulatoren führend. Einer Umsetzung des Geräts als spezialisierte Schaltung, kurz ASIC, stehen sie naheliegender Weise nach. Moderne ASICs erreichen Taktungen über 4 GHz wohingegen moderne FPGAs, wie die Speedster-Serie von Achronix [17], mit bis zu 1.5 GHz betrieben werden können. Studien wie [12, S. 211-212] zeigen zudem, dass die Berechnungsdauer, durch die weniger optimale Umsetzung in Logikgatter bei FPGAs gegenüber dem funktionalen Äquivalent als ASIC, durchschnittlich 3 bis 4 mal höher ausfällt.

Vorteil neuerer FPGAs, wie denen der Virtex Reihe von Xilinx gegenüber anderen Hardware-Emulatoren, ist die partielle dynamische Rekonfigurierung. Dabei kann ein Teil der FPGA-Komponenten, während des unbeeinflussten Betriebs der restlichen Komponenten und sogar durch eigens aufkonfigurierte Logik durchgeführt werden. Konzepte wie in [19] ermöglichen so zwar eine bedarfsorientierte Instanziierung von Hardware-Ressourcen, es bleibt jedoch der verhältnismäßig hohe Zeitaufwand für die Konfigurierung von FPGA-Komponenten. Zusätzlich wird aus [18, S. 20 ff.] ersichtlich, dass die dynamische Rekonfigurierung in der Praxis bislang Einschränkungen bezüglich der Form und Formdynamik der adressierbaren Regionen, sowie der Wiederverwendbarkeit der partiellen Konfigurationen unterliegt.

2.1.3 Zeitmultiplexbasierte Hardware-Emulatoren

Zeitmultiplexte Hardware-Emulatoren, desweiteren TMEs für *Time-Multiplexed Hardwaresided Logic Emulators*, bestehen, ähnlich den FPGAs, aus mehreren konfigurierbaren, flexibel vernetzten Logikblöcken und Speicherzellen. Ein Logikblock einer TME kann jedoch logische Komponenten realisieren deren Umfang über seine Kapazität hinausgeht. Eine solche Komponente wird vorab, ebenso wie bei einem FPGA, auf einzelne Konfigurationen vorhandener Blockarten dekompositioniert. Dabei werden Blöcke, insofern die Spezifikation dies zulässt, mehrfach belegt. Zur Laufzeit werden diese Mehrfachbelegungen dann abfolgend, entsprechend ihrer Abhängigkeiten, zyklisch auf die Blöcke konfiguriert und instruiert. Der jeweilige Kontext eines solchen Berechnungszyklus wird, zur Initialisierung des nächsten Zyklus, über die Konfigurierungsphasen hinweg in einer lokalen Speichereinheit gehalten. Das Konfigurationsarray eines Blocks wird ebenso lokal gespeichert, um die Dauer der Konfurierungen zu minimieren [1, S. 8-9].

Ein Ansatz für TMEs wird in [13] mit den sogenannten *Dynamically Programmable Gate Arrays*, kurz DPGAs vorgestellt. Der Autor optimiert hierbei Platzbedarf und Berechnungsgeschwindigkeit einer FPGA-Konfiguration, durch den Einsatz zeitmultiplexender Blöcke, in parallel geschalteten, unterschiedlich schnellen Berechnungspfaden. In [16] wird der Ansatz verallgemeinert und die gesamte Logik eines zu emulierenden Systems in Level unterteilt, welche dann zeitmultiplex auf den Logikblöcken des TMEs emuliert werden. Beide Konzepte verringern, gegenüber der Emulation mit klassischen FPGAs,

den Verschaltungsaufwand von Logikblöcken untereinander, sowie den Bedarf an Logikblöcken. Dafür steigt der Bedarf an internem Speicher durch das lokale Halten der Berechnungskontexte und Blockkonfigurationen erheblich.

Die neuere VEGA-Architektur in [1] beschreibt ein Netz zeitmultiplexbasierter Emulationsprozessoren. Ähnlich den Logikblöcken bei anderen TME-Ansätzen werden hier je Prozessor mehrere Logische Funktionen umgesetzt, deren Abfolge für eine konkrete Emulation in einem lokalen Instruktionsspeicher, kurz LIM, gehalten werden. Der sogenannte *Node Memory*, kurz NM, hält dabei die jeweiligen Berechnungskontexte. Auch hier nehmen die lokalen Speicher, LIM und NM, mit über 30% der Chipfläche einen hohen Anteil an der Gesamtlogik ein. Eine Verringerung des Aufwands an spezieller Emulations-Hardware gegenüber FPGA-Lösungen, gemessen an der umgesetzten Design-Komplexität, ist also nur durch eine umfangreiche Nutzung der zeitmultiplexen Dekomposition möglich.

2.1.4 Software-Emulatoren

Software-Emulatoren bilden ein ganzes Prozessorsystem auf das System ab auf welchem sie betrieben werden. Das emulierte Prozessorsystem wird in dem Zusammenhang auch Zielsystem genannt. Das System auf welches der Emulator abbildet nenne ich desweiteren Betriebsystem. Zur Emulationszeit werden bei diesem Ansatz Befehlsfolgen des Nutzers übersetzt auf Befehlsfolgesfolgen am Betriebssystem. Dies geschieht so, dass die Auswirkungen Letzterer, rückübersetzt ins Zielsystem, äquivalent zu den Auswirkungen der ursprünglichen Befehlsfolge sind wenn diese im Zielsystem ausgeführt würden. Auf dieser Basis lassen sich beliebige Geräte emulieren, zu denen eine Kommunikation per MMIO, Interrupts und PMIO durch das Zielsystem stattfindet.

Ein bekannter Software-Emulator ist das in [8] vorgestellte System namens Qemu, welches für verschiedene Betriebssysteme wie Linux, Windows und Mac OS existiert. Aufbauend auf dem jeweiligen Betriebssystem, bietet Qemu bereits Emulationen einiger verbreiteter Zielsysteme und Geräte an. Der Arbeitsfluss zur Integration eines weiteren zu emulierenden Zielsystems, beschränkt sich auf das Schreiben einer Übersetzung für jeden Befehl des Zielssystem, in entsprechende Befehlsblöcke für das Betriebssystem.

Unter Anderem die Mehrfachnutzung von Laufzeitübersetzungen wirkt sich in Qemu zu Gunsten der Emulationsgeschwindigkeit aus, einem eher kritischen Merkmal von Software-Emulatoren. Systememulationen mittels Qemu sind laut [8, S. 5], durchscnittlich 6-12 mal langsamer als die native Ausführung[2]. Dies kann als Referenz unter den nicht-spezialisierten Software-Emulatoren angesehen werden. Ein Problem heutiger Software-Emulatoren ist, dass das Zielsystem, welches die erforderlichen Kommunikationsmittel anbietet, ebenfalls emuliert wird. So muss zum Beispiel für das emuliertes MMIO eines Geräts jede Instruktion des Benutzers, Software-seitig auf entsprechende Speicherzugriffe geprüft werden.

[2]Für die Berechnung wird vorrausgesetzt, daß QEMU seine eigene MMU nutzt

2.2 Prozesslokale und Prozessübergreifende Funktionsaufrufe

2.2.1 Grundlagen

Funktionen abstrahieren von der Umsetzung abstrakter Funktionalitäten, auf Befehlssfolgen an physischen und virtuellen Geräten, zu denen ein Prozess Zugriff besitzt. Eventuelle Argumentwerte sowie der aktuelle Befehlszeiger, werden dazu vom Aufrufer für die Funktion hinterlegt. Daraufhin wird der Ausführungsfluß auf den Beginn der Funktion umgelenkt. Die Funktion hinterlegt ihrerseits gegebenenfalls Rückgabewerte für den Aufrufer und lenkt den Ausführungsfluss letztendlich wieder auf den ursprünglichen Befehlszeiger um. Diese Methodik kann auch thread- oder prozessübergreifend angewandt werden, wenn geeignete Kommunikationsmittel zur Übertragung der Argument- und Rückgabewerte existieren. Vor der Schnittstelle der Funktion kann dies, wie bei dem *Remote Procedure Call* Protokoll in [5], kurz RPC, transparent gehalten werden. Die Ansätze zur Transparenz von Geräteverfügbarkeit auf der Ebene prozesslokaler Funktionenaufrufe, sind somit auf die prozessübergreifender Funktionsaufrufe übertragbar. Speicherbereiche, welche prozessübergreifend referenziert werden, müssen allerdings synchronisiert werden.

2.2.2 Emulation

Hinter der abstrakten Schnittstelle einer Funktion, lassen sich Hardware-Ressourcen mit den Mitteln höherer Programmiersprachen emulieren. Die Instanziierung eines Software-Emulators kann zur Laufzeit somit in Form eines parallelen Prozesses, oder prozesslokal geschehen. Wird der Emulator durch einen parallelen Prozess umgesetzt, so kann die nötige Interprozesskommunikation zudem hinter der Funktionsschnittstelle transparent für den Nutzer gehalten werden. Vorteilhaft gegenüber der Emulation auf der Systembusebene ist die einfache Implementierung des Emulators, ohne Eingriff in die Umgebung des Nutzers oder spezielle Hardware. Die Abkapselung der Emulation eines Geräts in einen Prozess, vereinfacht die Verhaltensauswertung durch das Betriebssystem. Zudem kann der Nutzer durch die Adressraumvirtualisierung der Prozesse, gegen Fehlverhalten der Emulation robust gehalten werden. Ein wesentlicher Nachteil ist, dass auf diesem Weg die Entwicklung maschinennaher Treiber nicht möglich ist, da die Emulation nicht auf nativem Weg mit dem Nutzer kommuniziert. Kombinierte Hard- und Softwareentwicklung auf Basis der Emulation, wird somit, gegenüber der Umsetzung auf physischer Basis verfälscht.

2.2.3 Die Genode Betriebssystemarchitektur

Die Genode Betriebssystemarchitektur, kurz Genode, aus [2] bietet eine Umgebung, in der sich die beschriebene Integration von Hardware-Ressourcen, unter Transparenz der Verfügbarkeit für den Nutzer umsetzen lässt. Sie setzt auf verschiedene Kernel wie Linux, Fiasco und NOVA als Userland, also Anwendung

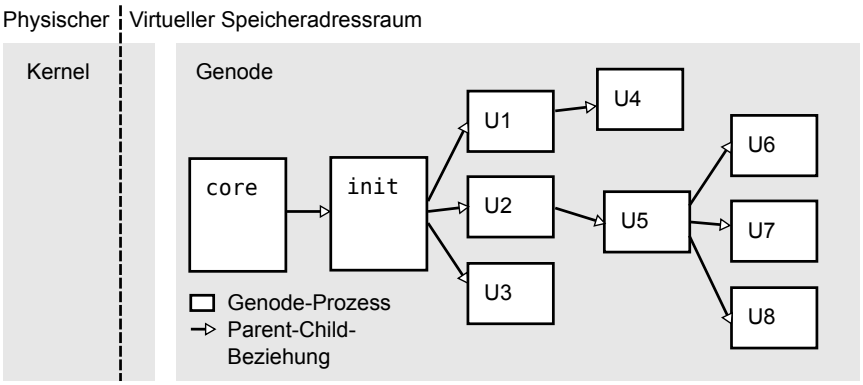

Abbildung 1: Struktur und Prozesshierarchie von Genode

niedriger Privilegierungsstufe auf. Da Genode Basis der Implementierung in Kapitel 4 ist, werde ich hier einen detaillierteren Einblick dazu geben. Die Prozessstruktur von Genode ist in Abbildung 1 beispielhaft dargestellt. Sie beschreibt, graphentheoretisch gesehen, immer einen gewurzelten Baum, mit Kanten welche von der Wurzel weg zeigen. Jeder Knoten stellt einen Prozess dar. Die Kanten zwischen den Knoten stellen je eine Eltern-Kind-Beziehung dar. Sie gehen jeweils vom Elternprozess aus und zeigen auf den Kindprozess welcher vom Elternprozess erzeugt wurde und verwaltet wird. Die Wurzel des Baums ist immer `core`, dessen einziges Kind `init` ist. Alle Ressourcen welche ein Prozess nutzt, müssen, mit Ausnahme von `core`, von seinem Elternprozess gestellt werden. `core` hingegen bekommt seine Ressourcen vom Kernel zugeteilt, von welchem er auch erzeugt wird. Alle Ressourcen die in Genode zum Einsatz kommen, sind somit durch `core` verwaltet. Ein Prozess kann jeden direkten Kindrozess, und somit implizit alle Prozesse in dessen Teilbaum, destruieren, da er über all seine Ressourcen verfügt.

Genode verwaltet die Rechte, welche ein Prozess auf den Ressourcen besitzt, durch sogenannte Capabilitys. Eine Capability ist eine systemweit eindeutige Identität für ein Objekt im Sinne der objektorientierten Programmierung. Die Methoden-Schnittstelle des Objekts reflektiert die Rechte welche durch die Capability vermittelt werden. Durch die prozessübergreifende Eindeutigkeit, dient die Capability als Referenz bei prozessübergreifenden aber auch prozesslokalen Funktionsaufrufen, welche die Schnittstelle des Objekts invokieren. Kreiert ein Prozess einen neuen Kindprozess, so gibt er diesem initial eine Capability mit, auch Parent-Capability genannt. Das Objekt welches durch die Parent-Capability assoziiert wird, wird auch Parent-Objekt genannt und bietet dem neuen Prozess grundlegende Funktionalitäten. Diese sind jedoch stets der Verwaltung durch den Elternprozess untergeben. Zu ihnen gehört unter Anderem das Anfordern weiterer Capabilities über den Elternprozess.

Darauf aufbauend kommt in Genode das Konzept von Diensten und Verbindungen zum Einsatz. Ein Dienst ist eine Schnittstelle, zu der ein Prozess, mitunter mehreren anderen Prozessen gleichzeitigen Zugriff bieten möchte. Die Vermittlung eines solchen Dienstes zwischen zwei Prozessen, dem Dienstanbieter und

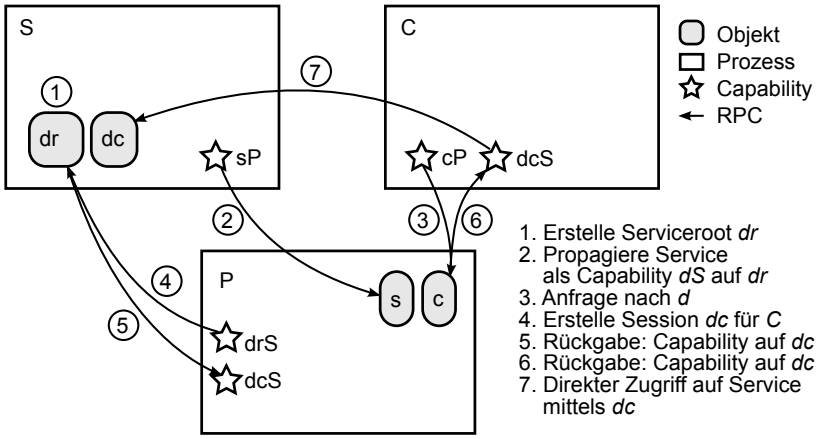

Abbildung 2: Vermittlung eines Dienstes in Genode

dem Nutzer des Dienstes, ist in Abbildung 2 beispielhaft dargestellt. In der Abbildung sind Anbieter und Nutzer, oder auch Server und Client durch S und C dargestellt. Beide sind Kindprozesse von P. Initial erzeugt der Anbieter ein sogennantes Root-Objekt zu dem Dienst, welches in der Abbildung durch dr dargestellt wird. Dieses Objekt implementiert eine Schnittstelle zum Erstellen und Schließen von Verbindungen zu dem Dienst. Über sein Parent-Objekt, kann der Anbieter den Dienst nun bei seinem Elternprozess bekanntmachen. Dabei reicht er dem Elternprozess eine Capability zu dem Root-Objekt des Dienstes. In der Abbildung ist dies drS. Die Bekanntmachung ist durch den Pfeil von der Parent-Capability cP zu dem Parent-Objekt c dargestellt. Ein Kind des Elternprozesses kann nun über seine Parent-Capability eine Verbindung zu dem Dienst anfordern. Daraufhin kann der Elternprozess, da ihm der angefragte Dienst bekannt ist, über die Root-Capability des Dienstes eine neue Verbindung für den Nutzer anfragen. Am Anbieter wird dafür ein neues Objekt erzeugt, welches die Schnittstelle des Dienstes für den Nutzer implementiert. Es ist in der Abbildung durch dc dargestellt. Auf dieses Objekt erzeugt der Anbieter nun noch eine Capability, welche er als Antwort auf die Anfrage nach einer Verbindung liefert. In der Abbildung sieht man wie diese Verbindungs-Capability dcS über den Elternprozess an den Nutzer, als Antwort auf dessen Anfrage weitergereicht wird. Es besteht nun eine direkte Verbindung zwischen Nutzer und Anbieter bezüglich des Dienstes welche der Indirektion über den Elternprozesse nicht mehr bedarf. Wird die Verbindung vom Elternprozess oder dem Anbieter geschlossen, so wird die Verbindungs-Capability invalidiert.

Der Elternprozess kann den Dienst auch bei seinem Elternprozess, durch erneutes Weiterreichen der Root-Capability bekanntmachen. Ebenso kann er Verbindungsanfragen des Nutzers an seinen Elterprozess übermitteln und die entsprechende Antwort dann zum Nutzer zurückleiten. Durch die rekursive Anwendung dieses Systems lässt sich ein Dienst in Genode über den gesamten Prozessbaum vermitteln. Die Kontrolle darüber, zwischen welchen Prozessen welche Verbindungen aufgebaut werden dürfen, bleibt dabei immer in der Hand der Elternprozesse, über welche der Dienst vermittelt wird. Ergänzend sei hier erwähnt, dass

ein Prozess eine Verbindung zu einem Dienst, welcher von einem seiner Kinder angeboten wird, auch für seine eigenen Zwecke aufbauen kann. Dazu benötigt er lediglich Zugriff auf das entsprechende Root-Objekt, was potentiell immer der Fall ist. Da das Dienstekonzept in Genode umfassend zur Vermittlung von Capability-basierten Rechten eingesetzt wird, basieren alle angebotenen Dienste direkt oder indirekt auf Diensten, welche `core` anbietet.

Zur prozessübergreifenden Kommunikation bietet Genode synchrone RPC-Mechanismen [23, K. 1], sowie asynchrone Benachrichtigung von Threads [21, K. 2.1.5] durch sogenannte Signale. Auch die Kommunikation über gemeinsame Speicherbereiche [22, K. 4.3] und ein darauf aufbauendes Paketfluss-Protokoll [20, K. 4.2] werden angeboten. Der Zugriff auf Arbeitsspeicher geschieht über Objekte, Dataspaces genannt, welche einen Speicherbereich und entsprechende Zugriffsrechte assoziieren. Somit lässt sich ein gemeinsamer Speicherbereich durch die Weitergabe der Dataspace-Capability an den Kommunikationspartner realisieren. Beim RPC wird der nötige Datenaustausch über Kernel-Mechanismen realisiert. Die asynchronen Signale hingegen transportieren keine zusätzlichen Daten. Der Sender muss bei einem Signal auf keine Rückmeldung blockieren, der Empfänger bekommt dennoch jede Benachrichtigung mit, welche an ihn adressiert ist.

2.2.4 Emulationsmodell in Genode

Ich nehme desweiteren an daß der Nutzer ein direkten Kindprozess von `core` ist. Die Vorgehensweise ist in anderen Fällen die selbe, solange der Elternprozess die benötigten Dienste vermitteln kann..

Für die Ressourcenintegration eines physischen Geräts welches über MMIO-Speicherbereiche und Interrupt-Signale mit dem Nutzer kommuniziert, sind insbesondere die beiden `core`-Dienste mit den Namen `IRQ` und `IO_MEM` relevant. In `core` verfügt der `IO_MEM`-Dienst über die Teile des physischen Speicheradressraums, welche nicht auf den *Random-Access Memory*, kurz RAM, abbilden. Entsprechende Verbindungen beziehen sich jeweils auf einen zusammenhängenden Teilbereich dieser Adressräume. Der `IRQ`-Dienst von `core` verfügt über den Adressraum aller Interrupts welche vom Kernel an `core` propagiert werden. Verbindungen an diesem Dienst werden auf ein Interrupt-Signal beschränkt und sind, solange sie gehalten werden, exklusiv auf diesem. Eine Verbindung vermittelt dem Nutzer das Recht auf das nächste Auftreten des Interrupts zu blockieren. Somit kann ein Nutzer, bei physischer Verfügbarkeit des Geräts, von den `IRQ`-Verbindungen und den `IO_MEM`-Verbindungen auf eine Funktionsschnittstelle abstrahieren.

Nun soll das Gerät, wie in Kapitel 2.2.2 beschrieben, in einem simultanen Prozess emuliert werden. Dazu benötigt der Nutzer, analog zu [2, S. 14 ff.] Verbindungen, welche ihm das Starten eines Programms in einem weiteren Prozess ermöglichen. In diesem Fall ein Emulatorprogramm welches das Gerät auf den, vom Nutzer vermittelten Diensten emuliert. Nun abstrahiert der Emulator auf die Funktionsschnittstelle welche, über die Genode-Mechanismen, auch anderen Prozessen zur Verfügung gestellt werden kann. Durch die hierarchische Ressourcenverwaltung von Genode behält der Nutzer die Kontrolle über alle, vom

Emulator verwandten Ressourcen, da er selbst sämtliche Rechte darauf besitzt. Durch die notwendigen Leserechte auf den Hintergrundspeicher zum Adressraum des Emulators, lässt sich dessen Verhalten vom Nutzer nachvollziehen. Die Überwachung von Ausführungskontexten des Emulators ist in Genode auf einigen Platformen möglich.

2.3 Motivation

Die vorausgegangenen Kapitel hatten einen Überblick zu den bisher verfügbaren Ansätzen und Möglichkeiten bezüglich der Geräteemulation zum Ziel. Hardware-Emulatoren die auf die Systembusebene abstrahieren, sind hinsichtlich der Berechnungsgeschwindigkeit die derzeit beste Lösung zur Transparenz der Verfügbarkeit. Die starken Limitationen bezüglich der Designgröße, sowie Kosten und Unflexibilität durch die zum Einsatz kommende spezielle Hardware, motivieren jedoch zu einer Emulation durch gewöhnliche Prozessorarchitekturen. Heutige Software-Emulatoren die auf die Systembusebene abstrahieren, setzen jedoch eine Emulation des gesamten Systems auf das Betriebssystem auf und betten den gewünschten Geräteemulator darin ein. Diese Vorgehensweise beansprucht unnötig zusätzliche Ressourcen und verkompliziert, durch die starke Abhängigkeit zur Systememulation, die emulationsgestützte Entwicklung des Geräts. Es wäre somit anstrebenswert den Geräteemulator in einem eigenen Prozess umzusetzen von welchem möglichst wenig Abhängigkeiten ausgehen bzw. zu dem keine Abhängigkeit vom restlichen System besteht. Dies würde auch die bedarfsorientierte Instanziierung emulierter Ressourcen vereinfachen. Umgesetzt ist es so bisher nur auf der Abstraktionsebene von Funktionsaufrufen wie in Kapitel 2.2.4 beschrieben. Diese Arbeit stellt deshalb ein Modell vor, welches prozessbasierte Emulatoren auf Prozessoren der Harvard-Architektur umsetzt, zu welchen der Nutzer über virtuelle MMIO Bereiche und Softwaregenerierte Interrupts kommuniziert. Die nötigen Kommunikationsmechanismen werden dabei durch den Kern des Betriebssystems umgesetzt.

3 Das Modell

3.1 Software-Emulation auf der Systembusebene

3.1.1 Prozessorarchitektur

Um den Zugriff auf emulierte MMIO-Bereiche unter der Annahme zu realisieren, dass Nutzer und Geräteemulator auf unterschiedlichen Speicheradressräumen laufen, ist es nötig, dass die Prozessorarchitektur Möglichkeiten zur transparenten Synchronisation solcher Zugriffe bereitsstellt. Bestimmte Speicherzugriffe sollen immer durch die Software abgefangen und behandelt werden können. Die CPU muss demzufolge einen sogenannten virtuellen Modus besitzen, in welchem sie Zugriffe auf Speicheradressen nicht, wie im physischen Modus, direkt über den Adressbus aufzulöst. Stattdessen sollen Zugriffe auf die emulierten

MMIO-Bereiche, im virtuellen Adressraum des Nutzers, zu Unterbrechung seiner Ausführung führen. Die Ausführung soll dazu zu einer statisch definierten Behandlungsroutine springen und in den physischen Modus wechseln. Dabei muss die CPU Vorkehrungen treffen, welche die vollständige Sicherung des unterbrochenen Ausführungskontextes ermöglichen. Zusätzlich muss sie Zugriffsadresse und Adressraum für die Behandlungsroutine identifizieren. Dennoch sollen Zugriffe auf die restlichen Adressbereiche möglichst effizient, also durch Hardware aufgelöst werden. Dies geschieht zum Beispiel durch eine Softwareseitig gesteuerte MMU, welche bekannte Auflösungen hält und im virtuellen Modus bei allen Speicherzugriffen von der CPU instruiert wird. Liefert die MMU keine Auflösung wird der Nutzer, wie auch an emulierten Bereichen unterbrochen, um die MMU Software-seitig zu instruieren. Somit muss von Seiten der Software nur sichergestellt werden, dass Zugriffe innerhalb des emulierten MMIO nie direkt durch die MMU aufgelöst werden.

Um die Auswertung der Zugriffe, auf emuliertes MMIO durch die Software möglichst effizient, also einfach zu gestalten, setze ich in diesem Modell ein RISC-Prozessorsystem voraus. Eine typische Eigenschaft von RISC-Prozessoren ist eine feste Befehlslänge und die einfache Formatierung der Befehle als Drei-Adress-Code Ein weiteres wichtiges Merkmal von RISC-Prozessoren ist die Load-Store-Architektur. Damit sind Kombination von Speicherzugriffen mit arithmetischen oder logischen Operationen beziehungsweise Operationen zum direkten Datentransfer im Speicher ausgeschlossen. Auch Ausführungssprünge können somit während eines Speicherzugriffs nicht vorkommen. Der Zugriff auf den CPU-externen Speicher beschränkt sich also auf solche Operationen, welche Daten entweder von CPU-Registern in CPU-externen Speicher oder von CPU-externem Speicher in CPU-Register transferieren. Dadurch ist ausgeschlossen dass zur Emulation, Adressen des Nutzeradressraums außerhalb des emulierten Bereichs ausgewertet beziehungsweise aufgelöst werden müssen. Die Auflösung von Nutzeradressen außerhalb des emulierten Bereichs würde ein weiteres Problem aufwerfen. Denn der Nutzer möchte die nötigen Vergabe der Rechte zu dieser Art Auflösung, nicht durch die Nutzung von MMIO impliziert sehen, da es für ihn transparent bleibt, ob dieses emuliert oder physisch vorhanden ist.

Zur Umsetzung emulierter Interrupts sollte es am Prozessor möglich sein, software-seitig Interrupts auszulösen.

3.1.2 Betriebssystem

Für die Zwecke dieses Modells wird davon ausgegangen, dass der Betriebssystemkern der erste laufende Prozess ist, welcher unter anderem auf den physischen Adressraum direkten Zugriff besitzt. Alle weiteren Prozesse sollen in virtuellen Adressräumen betrieben werden. Der Kern stellt eine Schnittstelle zur synchronen Interprozesskommunikation bereit. Darüber können Prozesse beim Betriebssystem die Assoziation von Teilen ihres Adressraums, mit Teilen des physischen Adressraums anfordern, beziehungsweise Assoziationen bezüglich ihres Adressraums aufheben. Die dazu nötigen Ressourcen werden einem Prozess initial vom Betriebssystem in seinem Adressraum bereitgestellt. Ein Prozess kann weitere Prozesse starten, deren Adressraum unter anderem durch seinen eigenen hinterlegt werden können. Prozesse müssen dazu die Möglichkeit besitzen,

über Speicherzugriffe auf Adressen anderer Adressräumen, durch das Betriebssystem informiert zu werden. Außerdem müssen sie diese zumindest temporär aufzulösen können.

Wie in Kapitel 3.1.1 beschrieben ist es nötig, Prozesse bei bestimmten MMIO-Speicherzugriffen immer zu unterbrechen. Das setzt voraus, dass das Betriebssystem Mittel zur Deklaration von Adressraumbereiche anbietet, welche nicht durch das Betriebssystem aufgelöst werden. Desweiteren soll das Betriebssystem, nachdem es durch die Unterbrechung, infolge eines solchen Zugriffs instruiert worden ist, Informationen über den Zugriff an einen Prozess weiterreichen, damit dieser die Synchronisation mit dem Emulator vornehmen kann. Dazu kommt die synchrone Interprozesskommunikation zum Einsatz.

3.1.3 Prozesse

Aufbauend auf dem Betriebssystem existieren nun drei Arten von Prozessen, welche für das Modell von Bedeutung sind. Dies sind die Nutzer, der sogenannte Virtual-Device-Monitor und die Emulatoren. Der Nutzer-Prozess spricht ein Gerät auf der Systembusebene im lokalen Adressraum, unabhängig von der physikalischen Verfügbarkeit des Geräts an. Er kann zum Beispiel, als Treiber für das Gerät, auf eine abstrakte Funktionsschnittstelle für weitere Gerätenutzer abstrahieren.

3.1.4 Memory-mapped I/O

Die Realisierung des emulierten MMIO eines Gerätetyps wird vom Virtual-Device-Monitor, kurz VDM, vorgenommen. Dieser Prozess soll hier durch das Betriebssystem erzeugt werden und besitzt im Emulationsfall Zugriff auf Ressourcen, welche es ihm ermöglichen, einen Emulator als weiteren Prozess zu erzeugen. Ist, oder wird, sobald nötig, der Emulator durch den VDM gestartet, erwirbt er initial bei seinem Erzeuger die Zugriffsrechte auf einen RAM-Speicherbereich entsprechend des emulierten MMIO-Bereichs. Auch der VDM besitzt in seinem Adressraum die angeforderten Rechte auf diesen Speicherbereich. Nun bietet der VDM dem Nutzer die Assoziation eines nutzerlokalen Adressbereichs mit dem MMIO-Bereich des Gerätes mit an. Dies tut er unabhängig davon, ob Letztere physisch durch das Gerät hinterlegt sind, beziehungsweise er, gesetzt den Fall, Rechte darauf besitzt. Der Nutzer kann nun bei dem VDM, ebenso wie beim Betriebssystem, Zugriff auf die MMIO-Bereiche anfordern, nur dass der VDM gegenüber dem Betriebssystem von der tatsächlichen Verfügbarkeit abstrahiert. Der VDM vermerkt sich daraufhin eine bijektive Abbildung, welche die gegenseitige Zuordung der MMIO-Bereiche von Nutzer und VDM wiederspiegelt.

In Abbildung 3 ist die Vorgehensweise, bei einem Schreibzugriff seitens des Nutzers U, auf eine Adresse ep_U eines emulierten MMIO-Bereichs A_U^D veranschaulicht. Die durchgehenden Pfeile stellen die Kommunikation der Prozesse untereinander, durch synchrone *Inter-Process-Communication*, kurz IPC, dar. Der Speicherzugriff hat eine unmittelbare Unterbrechung des Nutzers durch die

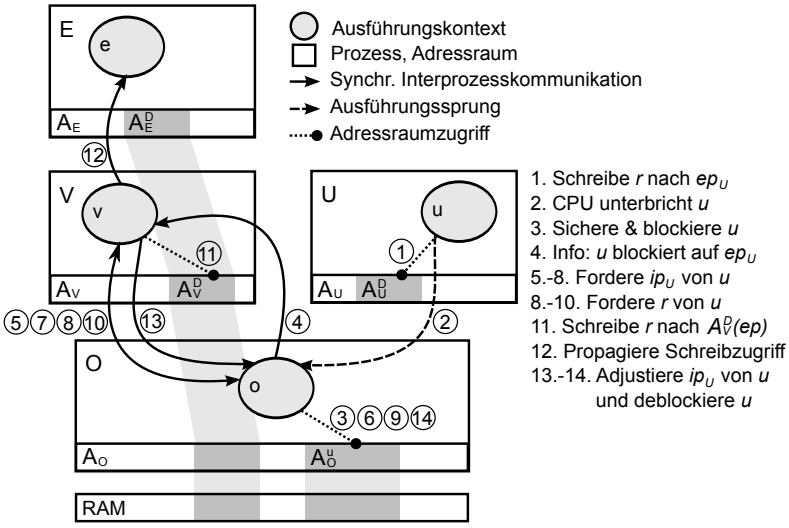

Abbildung 3: Emulation eines Geräts mit MMIO an einem Nutzer

CPU zufolge. Der dabei stattfindene Sprung in das Betriebssystem O ist durch den gestrichelten Pfeil an seinem Nutzerkontextkontext u dargestellt. Daraufhin sichert das Betriebssystem den unterbrochenen Ausführungskontext. Außerdem wird der Ausführungskontext als blockiert gekennzeichnet, so, dass er vorerst nicht mehr zur Ausführung kommt. Das Betriebssystem benachrichtig nun den VDM über die Unterbrechung und schickt dabei die Identität des Unterbrochenen und die Adresse des Speicherzugriffs mit. Diese Kommunikation ist als durchgehender Pfeil vom Betriebssystemkontext o zum VDM-Kontext v dargestellt. Zudem bietet es dem VDM Zugriff auf den gesicherten Kontext, welcher unter Anderem den Befehlszähler des Nutzers, hier ip_u genannt, enthält. Der VDM testet seinerseits, ob die Zugriffsadresse des Nutzers in einem Adressbereich liegt welchen er emuliert. Ist dies der Fall, so fordert er dessen Befehlszähler, aufgelöst auf den physischen Adressraum an. Der Binärcode des letzten Nutzerbefehls kann dann vom Betriebssystem in den Adressraum des VDM abgebildet und von diesem ausgelesen werden.

Nun kann der VDM den Befehl, welcher zu dem Speicherzugriff geführt hat dekodieren. Im Fall eines einfachen RISC-Prozessors, wie in Kapitel 3.1.1 beschrieben, beschränkt sich der Zugriff auf den Transfer zwischen einem CPU-Register, dem Zielregister, und der Zugriffsadresse. Der letzte Wert des Zielregisters vor dem Zugriff ist im gesicherten Ausführungskontext des Nutzers enthalten. Der VDM löst nun die Zugriffsadresse des Nutzers, über die Abbildung von Nutzer- auf VDM-MMIO-Bereiche, auf den emulierten MMIO-Bereich auf. Dann fordert er beim Betriebssystem den Wert des Zielregisters und informiert den Emulator per IPC. In der Nachricht enthalten sind Zugriffsart, das Offset des Zugriffs im MMIO-Bereich und der zu schreibende Wert. Dies stellt der Pfeil zu sei-

nem Ausführungskontext e dar. Hat der Emulator den Schreibzugriff auf seinen MMIO-Bereich verarbeitet, vermeldet er dies bei dem VDM durch seine IPC-Antwort, um ihm zu signalisieren dass nun weitere Zugriffe möglich sind. Der VDM muss nachträglich noch Sorge dafür tragen, dass der Befehlszähler des Nutzers adjustiert wird. Dazu weist er das Betriebssystem an, die Befehlsweite des Schreibzugriffs im gesicherten Kontext auf den Befehlszähler zu addieren. Dadurch wird sichergestellt, dass nicht erneut der Zugriff, sondern der unmittelbar folgende Befehl ausgeführt wird, sobald der Nutzer wieder zur Ausführung kommt. Befehlssprünge können bei Speicherzugriffen, aufgrund der vereinbarten RISC-Architektur nicht vorkommen. Da der Emulator die Nachricht über den Zugriff verarbeitet hat, kann der Nutzer nun durch den VDM über das Betriebssystem deblockiert werden. Durch diese Vorgehensweise bleiben MMIO-Zugriff synchronisiert zwischen Nutzer und Emulator.

Im Fall eines Lesezugriffs durch den Nutzer ist die Vorgehensweise anfangs analog zu der, welche im Falle eines Schreibzugriffs zum Einsatz kommt. Nach der Dekodierung des Befehls fordert der VDM jedoch das Betriebssystem auf, den Wert der lokal aufgelösten Adresse des Zugriffs, in das gesicherte Zielregister zu schreiben. Der Wert ist korrekt bezüglich des Emulatorstatus, solange sich der Emulator nicht in einer Prozessierungsphase eines Schreibzugriffs befindet. Da nur ein, zum Emulator synchronisierter Nutzer vorhanden ist, ist dies der Fall. Eine Benachrichtigung des Emulators ist bei Lesezugriffen nicht nötig. Ist der Schreibvorgang an Zielregister erfolgt, adjustiert der VDM, wie bei Speicherzugriffen, wieder den Befehlszähler des Nutzers und deblockiert diesen.

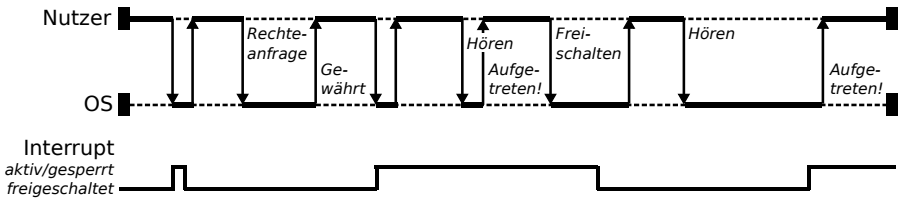

Abbildung 4: Sequenzdiagramm zur nativen Interrupt-Behandlung

3.1.5 Asynchrone Ereignisse

Geräte-Interrupts sollen am Nutzer nach der Semantik emuliert werden, welche durch das Sequenzdiagramm 4 verdeutlich wird. Sobald ein Interrupt auftritt, wird er in den gesperrten Status überführt, in welchem er nicht erneut auftreten kann. Der Interrupt muss dann Software-seitig freigeschaltet werden um wieder aus dem gesperrten Zustand zu gelangen und erneut auftreten zu können. Zu Beginn besitzt der Nutzer keine Rechte auf dem Interrupt, folglich wird er nicht über dessen erstes Auftreten informiert. Statt dessen schaltet hier das Betriebssystem, kurz OS, den Interrupt frei. Nun erwirbt der Nutzer die Rechte, auf das Auftreten des Interrupts zu hören und ihn freizschalten zu können. Er wird in diesem Zustand noch nicht über das Auftreten des Interrupts informiert, wie beim zweiten Übergang des Interrupts in den gesperrten Zustand ersichtlich wird. Nun teilt er dem Betriebsystem per synchroner IPC mit, dass er auf den Interrupt

hören möchte. Solange das Betriebssystem nicht antwortet, bleibt der Nutzer durch die IPC-Anfrage blockiert. Die Antwort erfolgt, sobald der Interrupt nach der Anfrage im gesperrten Zustand ist. Ist dies bereits der Fall wenn der Nutzer die Anfrage stellt, erfolgt die Antwort direkt. Der gesperrte Interrupt kann nur explizit vom Nutzer freigeschaltet, also entsperrt werden. Bei der zweiten Anfrage des Nutzers, auf den Interrupt zu hören, ist dieser jedoch nicht aktiv. Somit wartet das Betriebssystem bis der Zustand eintritt und reaktiviert den Nutzer dann.

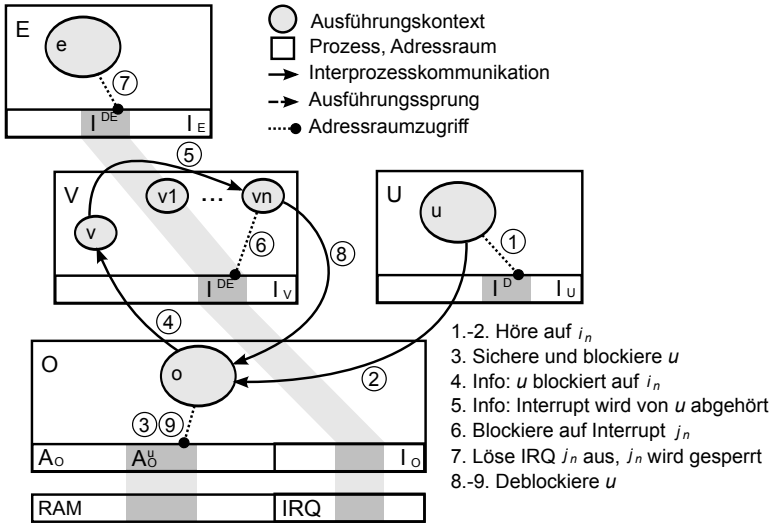

Abbildung 5: Emulation eines Geräts mit Interrupts an einem Nutzer

Basis für die Umsetzung dieses Verhaltens für mehrere emulierte Interrupts an einem Nutzer, ist erneut der VDM-Prozess. Er bietet dem Nutzer die Rechte zum Hören und Freischalten auf die Interrupts des emulierten Geräts, unabhängig davon, ob diese physisch existent, beziehungsweise in Gebrauch sind. Hat der Nutzer die Rechte erworben, werden Anfragen auf diese Interrupts vom Betriebssystem infolgedessen mit dem VDM assoziiert. Abbildung 5 stellt den Vorgang dar, welcher durch das Hören des Nutzers auf einem der Interrupts angestoßen wird. Die Anfrage des Nutzers ist durch den durchgehenden Pfeil vom Nutzer U zum Betriebssystem O dargestellt. Dieses benachrichtigt dann den VDM V, wie der durchgehender Pfeil von O zu V zeigt. Zur Emulation der Nutzeranfrage wird der Emulatorprozess nötig, welcher, insofern dies nicht bereits geschehen ist, durch den VDM kreiert wird. In der Abbildung ist dies der Prozess E. Der VDM muss desweiteren, zur Kommunikation mit dem Emulator, die Rechte zum Hören, Auslösen und Freischalten auf ungenutzte Interrupts aus dem physischen Adressraum besitzen. Hierzu sei erwähnt, dass diese nicht mit den angeforderten Interrupts des Nutzers übereinstimmen müssen. Diese Rechte vermittelt er nun auch dem Emulator. Der Emulator wirft dann die Interrupts, wie auch das Gerät zur asynchronen Benachrichti-

gung des Nutzers, nur Software-seitig. Anders als beim Gerät jedoch, hört nicht der Nutzer auf die Interrupts, sondern der VDM. Dazu benötigt er, neben seinem Hauptausführungskontext zu jedem emulierten Interrupt einen weiteren, parallel laufenden Ausführungskontext. In der Abbildung sind diese durch $v1$ bis vn dargestellt. Jeder dieser Ausführungskontexte legt sich initial schlafen, da er erst bei einer entsprechenden Nutzeranfrage gebraucht wird. Außerdem hält der VDM eine bijektive Abbildung von den Emulator-Interrupts zu den Nutzer-Interrupts. Möchte der Nutzer nun auf einen der emulierten Interrupts hören, löst der Hauptausführungskontext diesen zu einem der Emulator-Interrupts auf und benachrichtigt den entsprechenden VDM-Kontext darüber. Dieser blockiert infolgedessen auf den ihm zugewiesenen Interrupt. Wirft der Emulator nun den Interrupt, wird der VDM-Kontext reaktiviert und löst den aufgetretenen Interrupt wieder zu dem emulierten Interrupt des Nutzers auf. Der Nutzer muss dann noch geweckt werden. Modifikationen am Befehlszähler des Nutzers sind nicht nötig, da der Nutzer über synchrone Kommunikation mit dem VDM in Verbindung steht und nicht, wie bei emuliertem MMIO, unterbrochen wurde. Der Nutzer wartet explizit auf den Interrupt, womit auch der Transfer weiterer Informationen zum Nutzer entfällt. Somit muss der VDM ihn lediglich durch eine leere Antwort reaktivieren.

Der VDM lässt einen geworfenen Emulator-Interrupt im gesperrten Zustand, so dass er vorerst nicht erneut vom Emulator geworfen werden kann. Möchte Der Nutzer den Interrupt nun wieder freischalten, so wird die Anfrage wieder durch das Betriebssystem an den VDM geleitet. Dieser schaltet den entsprechenden Emulator-Interrupt dann über das Betriebssystem frei und vermeldet dem Nutzer daraufhin die erfolgreiche Freischaltung des emulierten Interrupts.

3.2 Mehrere Nutzer

In der Praxis kommt es vor, dass mehrere Nutzer zur gleichen Zeit ein Gerät nutzen wollen. Dabei möchte man diesen Aspekt vor dem Nutzer verbergen und wendet zum Beispiel betriebssystemintern Scheduling-Methoden an, um eine gegenseitige Beeinflussung zu verhindern. Das vorgestellte Modell lässt sich, bezüglich der Anzahl der Nutzer verallgemeinern. Damit ist es möglich zur Laufzeit Ersatzimplementierungen eines Gerätes zu erzeugen um diese Nutzern exklusiv und transparent zuzuweisen. Dies stellt eine Alternative zu Methoden wie dem Scheduling dar. Abbildung 6 veranschaulicht die Konstellation für den Fall, dass die Nutzer U_1 bis U_m Zugriff auf ein Gerät bei dem VDM V erwerben. Die Adressräume werden in der Abbildung der Einfacheit halber auf die, für das Modell relevanten Bereiche beschränkt. An den Nutzern sind dies sind die emulierten MMIO-Bereiche A^D und die Interrupt-Bereiche I^D. Sie sind an jedem Nutzer gleich benannt da sie es sich aus Sicht der Nutzer um die selben Ressourcen handelt. Auch die Hauptausführungskontexte der Prozesse sind hier nicht die dargestellt.

Im Hintergrund agiert der VDM bei Zugriffen wie in Kapitel 3.1.4 und 3.1.5 beschrieben mit ein paar Anpassungen. Zu jedem Nutzer des emulierten Geräts wird von dem VDM nun genau ein Emulatorprozess gestartet, welcher nur für die Emulation des Geräts an diesem Nutzer verwantwortlich ist. Die Emula-

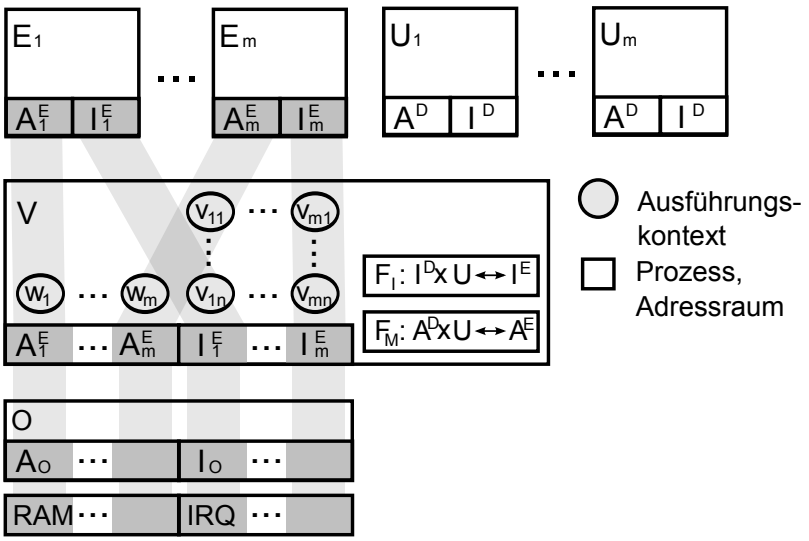

Abbildung 6: Emulation eines Geräts mit MMIO und Interrupts an mehreren Nutzern

toren zu den Nutzern in der Abbildung werden durch die Prozesse E_1 bis E_n dargestellt. Betrachten wir zuerst die Emulation des MMIO an den Nutzern. Bei nur einem Nutzer, in Kapitel 3.1.4, hat der Hauptausführungskontext der VDM auf MMIO-Zugriffe des Nutzers gewartet. Nun muss der VDM mit mehreren, womöglich parallelen MMIO-Zugriffen durch die Nutzer rechnen. Somit benötigt er für jeden Nutzer einen parallel laufenden Ausführungskontext, welcher auf einen MMIO-Zugriff des, ihm zugewiesenen Nutzers blockiert. Für jeden dieser Kontexte hält der VDM eine Abbildung von den MMIO-Adressen des Nutzers zu den Speicheradressen des jeweiligen Emulators, wie sie auch schon in Kapitel 3.1.4 zum Einsatz kam. In der Grafik sind die zusätzlichen Ausführungskontexte durch w_1 bis w_n dargestellt. Die Erweiterung der Abbildungen wird durch das Kreuzprodukt mit der Nutzermenge U in der Abbildung F_M verdeutlicht.

Nun sehen wir uns die Emulation der Interrupts für mehrere Nutzer an. Im Einzelnutzerbetrieb in Kapitel 3.1.5 wartete zu jedem Interrupt des Nutzers ein Ausführungskontext am VDM auf, welcher die Behandlung von entsprechenden Anfragen durchführte. Nun benötigen wir solche Ausführungskontexte also für jeden Nutzer. Das emulierte Gerät nutzt in der Abbildung n viele Interrupts. Für den ersten Nutzer warten dementsprechend Kontext v_{11} bis v_{1n} auf, für den zweiten v_{21} bis v_{2n} und so weiter, bis zum letzten, hier den $m$ten Nutzer. Jeder dieser Kontexte braucht nun auch einen realen Interrupt, über welchen der entsprechende Emulatorprozess, wie in Kapitel 3.1.5, mit ihm kommuniziert. Der VDM muss also je Interrupt des emulierten Geräts, auf ebensoviele reale Interrupts zugreifen können, wie Nutzer existieren. Oder anders ausgedrückt, je Nutzer muss er ebensoviele reale Interrupts besitzen wie das emulierte Gerät

nutzen würde. Für den ersten Nutzer sind dies in der Abbildung an dem V die Adressen I_1^E, für den Zweiten entsprechend I_2^E und so weiter, bis zum letzten Nutzer und I_m^E. Die grauen Verbindungen zu den jeweils selben Adressen an den Emulatoren E_1 bis E_m verdeutlichen, dass auch jeder Emulator Zugriff auf die Interrupts besitzt, welche mit seinem Nutzer assoziiert werden. Somit ist die Kommunikation zwischen VDM und Emulatoren auf mehrere Nutzer adaptiert. Der VDM muss sich, wie auch im Einzelnutzerbetrieb, für jeden emulierten Interrupt eines Nutzers merken, mit welchem realen Interrupt des entsprechenden Emulators er jeweils assoziiert wird. Dazu wird die bijektive Abbildung, welche bereits im Einzelnutzerbetrieb dafür genutzt wurde, erweitert. Diese Erweiterung ist in der Grafik durch das Kreuzprodukt mit der Nutzermenge U in F_I dargestellt.

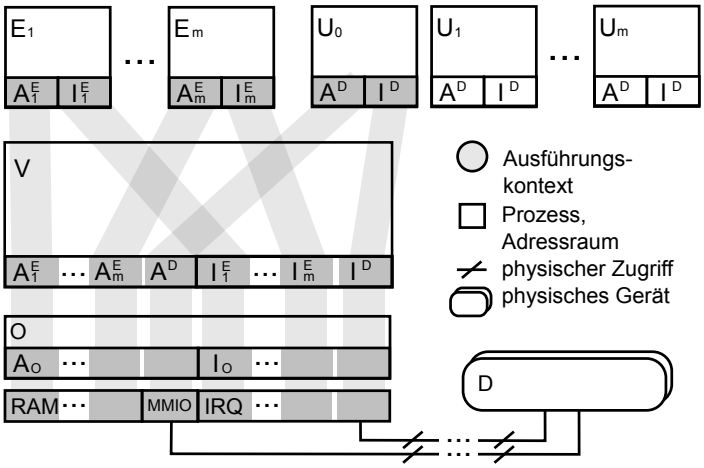

Abbildung 7: Emulation eines Geräts mit MMIO und Interrupts an mehreren Nutzern

3.3 Einbindung des physischen Geräts

Es liegt nahe den VDM so zu modifizieren, dass er die transparente Einbindung des physischen Geräts neben den Emulationen realisiert. Abbildung 7 stellt den modellhaften Aufbau für diesen Fall dar. Es existieren wieder, wie in Kapitel 3.2, mehrere Nutzer welche gleichzeitigen exklusiven Zugriff auf das Gerät über den VDM fordern. Der VDM ist wie gewohnt als V, die Nutzer als U_0 bis U_m und die Emulatoren als E_1 bis E_m dargestellt. Nun ist auch eine reale Version des Geräts vorhanden. In der Abbildung wird dies durch D verdeutlicht. Die Zugriffsrechte auf dessen MMIO-Bereiche und Interrupts erwirbt der VDM über das Betriebssystem. Im Adressraum des VDM ist der MMIO-Bereich des Geräts als A^D und die Interrupts als I^D dargestellt. Die grauen Verbindungen welche

in der Abbildung davon ausgehen weisen darauf hin daß der VDM dem Nutzer U_0 den Zugriff auf die physischen Ressourcen gewährt hat. Folglich verringert sich die Anzahl der benötigten Emulatoren. Diese bedienen in der Abbildung nur die Nutzer U_1 bis U_m. Die Vorgehensweise bei den emulierten Ressourcen bleibt die selbe wie im Modell aus Kapitel 3.2.

Da die Einbindung einer physischen Instanz des Geräts keine Auswirkung auf die, bereits laufenden Emulationen hat, kann sie dynamisch erfolgen. Dies kann sich zum Beispiel danach richten ob das Betriebssystem die Verfügbarkeit eines Geräts ausweist und der VDM die entsprechenden Zugriffsrechte dazu erwerben kann. Dabei ist zu beachten dass das Modell nicht vorsieht, eine laufende Emulation transparent durch ein physisches Gerät zu ersetzen. Dies folgt da der Zustand des Emulators nicht in eindeutigem Bezug zu einem Zustand des Geräts steht und somit nicht übertragbar ist. Ebenso verhält es sich bei dem Versuch ein Gerät, welches genutzt wird transparent durch einen Emulator zu ersetzen. Ein physisches Gerät welches von einem Nutzer freigegeben wird kann jedoch, unter Umständen erneut vermittelt werden. Dazu muss ein Initialzustand des Gerätes vereinbart werden. Der VDM versetzt das Gerät dann jedesmal in diesen Zustand, wenn er es einem neuen Nutzer zusprechen möchte. Auch auf mehrere Instanzen des Geräts können so vermittelt werden. Für das Emulationsmodell bringt dies keine Änderungen mit sich. Es greift so, wie beschrieben, für alle Nutzer, welchen kein physisches Gerät durch den VDM zukommt.

4 Beispielimplementierung

4.1 Zielsetzung

Ich möchte das beschriebene Modell nun, wie in der Einleitung erwähnt, in einem Beispielszenario auf dem Genode OS Framework verdeutlichen. Als Ziel habe ich mir dafür die Implementierung eines einfachen 32-Bit Addierers gesetzt. Der Addierer soll einerseits als Hardware-Emulation durch eine FPGA-Konfiguration und andererseits als emulierender Prozess in Genode vorliegen. Er hört, auf zwei 32-Bit Registern, die Summanden als vorzeichenlose Integerwerte ab. Diese werden unmittelbar addiert und die Summe module 2^{32} in ein drittes 32-Bit Register geschrieben. Die Summandenregister behalten vom Nutzer geschriebene Werte, während Schreibzugriffe auf das Summenregister ohne Effekt bleiben. Alle drei Register sind als ein zusammenhängender MMIO-Bereich im physischen Adressraum verfügbar. Für dieses Szenario gehe ich vorerst davon aus, dass die Frequenz der Summandenabfrage, und somit der unmittelbar ausgeführten Berechnung, im Falle des FPGA-Addierers mehr als doppelt so hoch ist, wie die schnellstmögliche Frequenz von Schreibzugriffen durch den Microblaze. Softwareseitig soll diese Annahme durch eine transparente Synchronisation der Schreibzugriffe zu den entsprechenden Berechnungen wiedergespiegelt werden.

Ein exemplarischer Nutzerprozess soll nun über seinen Elternprozess die Rechte, zum Schreib- und Lesezugriff auf den MMIO-Bereich des Addierers, in seinem virtuellen Adressraum vermittelt bekommen. Daraufhin schreibt dieser einige Summandenkonstellationen und überprüft unmittelbar den resultierenden Wert

im Summenregister. Der Nutzerprozess soll von Genode mindestens in zwei, pseudoparallel laufenden Instanzen gestartet werden. Würden beide auf dem FPGA-Addierer agieren, bestände bei dem Test die Wahrscheinlichkeit einer Überschneidung der Zugriffe. Es wäre möglich dass der Addierer für eine Berechnung Summanden von zwei unterschiedlichen Nutzern entgegennimmt oder ein Nutzer nicht die Gelegenheit bekommt, sein Ergebnis rechtzeitig zu lesen, ehe es von einer weiteren Berechnung überschrieben wird. Mittels des vorgestellten Modells sollen die Nutzer deshalb transparent auf unterschiedlichen Implementierungen des Addierers agieren. Ist der FPGA-Addierer in Benutzung, soll er dabei präferiert vermittelt werden. Ist dies jedoch nicht der Fall kommt eine dynamisch erzeugte, exklusive Instanz des Addiereremulators zum Einsatz.

Der Prozess welcher den [Virtual Device Monitor] umsetzt, soll in der Genode-Prozesshierarchie möglichst unabhängig von den Nutzerprozessen gehalten werden. Während die Emulationsprozesse direkte Kindprozesse des VDM-Prozess sind sollen die Nutzer unabhängig von ihrer Position im Genode-Prozessbaum durch den Dienst der VDM bedient werden können. Dabei sollten die Rechte der VDM gegenüber Ressourcen der Nutzer möglichst gering und vor allem zweckbezogen gehalten werden. Aus diesen Gründen, welche durch die Restriktionen des Genode-Rechtesystems bestärkt werden, habe ich mich entschieden, das Modell für dieses Szenario leicht zu modifizieren. So wird die Dekodierung des emulierten Speicherzugriffs direkt durch das Betriebssystem vorgenommen, um die VDM davor zu bewahren, Kenntnis und Zugriff zu dem Adressraum des Nutzers zu besitzen. Statt dessen vermittelt das Betriebssystem, in einem möglichst platformunabhängigen Format, lediglich Art und Parameter des Zugriffs an die VDM. Auch die Modifikation des Ausführungskontextes am Nutzer durch die VDM ist in Genode aus Sicherheitsgründen nicht vorgesehen. Deshalb vermeldet die VDM eine erfolgreiche emulatierte Instruktion beim Betriebssystem, wodurch dieses die nötige Modifikation vornimmt. Die dritte Abweichung betrifft die Kommunikation zwischen Emulator und VDM. Da sich hierfür Genodes Dienst-Methodik anbietet, wird der zuständige Ausführungskontext der VDM nicht selbst auf die Speicherbereiche des Emulators zugreifen, sondern Schreib und Lesezugriffe durch einen Dienst am Emulator prozessieren lassen. Dadurch bleibt zudem die Synchronisation zwischen den beiden Prozessen gewahrt, am Aufbau der VDM ändert sich jedoch desweiteren nichts.

4.2 Basissystem

4.2.1 Das Board

Hardware-seitig ist die Basis für mein Szenario ein S3AN Starter Kit von Xilinx, welches in [24] dokumentiert ist. Der zentrale FPGA dieses Boards ist ein Spartan 3AN, ebenfalls von Xilinx. Die Konfigurierung des FPGAs findet über die integrierte *Joint Test Action Group Interface* Schnittstelle, kurz JTAG, mittels USB statt. Textausgaben des Systems höre ich über die serielle RS-232-Schnittstelle des Boards ab, welche über die I/O-Pins des FPGA direkt angesprochen werden kann. Zum Vorhalten größerer Datenmengen wie der Programmdaten wird das, direkt angebundene DDR2-SDRAM Modul des Boards genutzt. DDR2-SDRAM steht für die *Double Data Rate Synchronous Dyna-*

mic Random Access Memory Spezifikation für Speichermodule, in der zweiten Version.

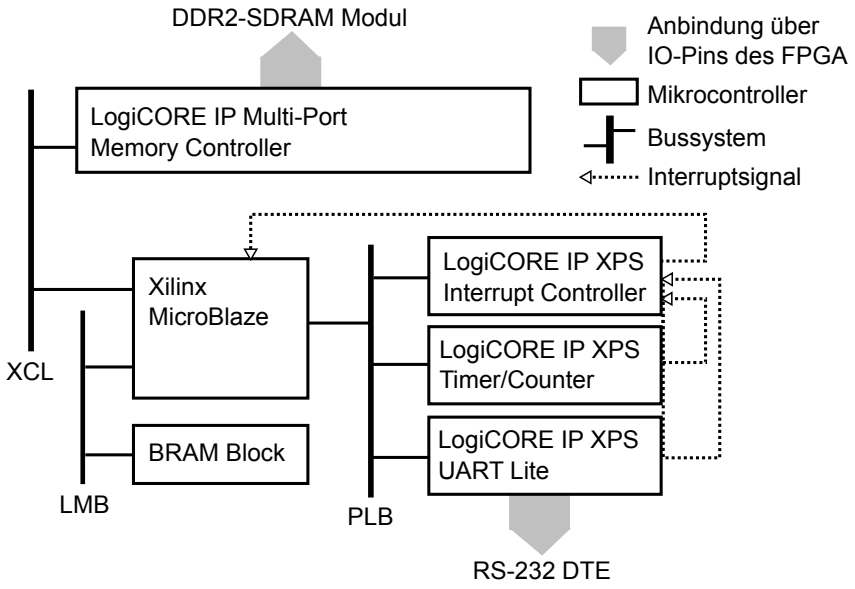

Abbildung 8: Schematische Darstellung des SoC für das Beispielszenario

4.2.2 Das System on Chip

Der Spartan 3AN FPGA wird für das Szenario mit einem passendem *System-on-Chip*, kurz SoC, konfiguriert. Die Konfiguration, welche dieses SoC auf dem FPGA umsetzt, wurde zuvor mittels des Xilinx Platform Studio aus dem Embedded Developement Kit von Xilinx generiert. Der interne Aufbau des SoC ist in Abbildung 8 schematisch dargestellt. Zentrale Komponente des resultierenden SoC ist der MicroBlaze Mikrocontroller welcher als CPU des Systems fungiert. Er ist mit einem Cachespeichersystem für Adressauflösungen, dem Translation Lookaside Buffer ausgestattet. Dieser Software-seitig steuerbare Cache dient dem System zur effizienten Umsetzung der virtuellen Adressräume. Außerdem nutze ich einige Optimierungsmethoden des MicroBlaze, wie den Hardwaremultiplizierer und Hardwarebeschleunigte Gleitkommaberechnungen, welche nicht essentiell für die Beispielimplementierung sind. Die MicroBlaze-Architektur folgt dem RISC-Prinzip wie im Modell vorausgesetzt. Sein Befehlssatz sieht je 6 Instruktionen zum Speichern in, beziehungsweise Laden aus dem Adressraum vor. Die Instruktionen zum Laden, beziehungsweise Speichern sind zum einen jeweils 3, absolut adressierte Befehle für die Transferbreiten Wort, Halbwort und Byte. Zum Anderen existieren auch je 3, per Register adressierte Befehle für ebendiese Transferbreiten. Diese Befehle sind in Tabelle 1 und 2 nach der Dokumentation in [3] ausgeführt.

Direkt an den IRQ-Eingang des MicroBlaze angebunden, ist der

Befehlsformat	**Beschreibung**
`swi rA, I`	Speichere Inhalt (32 Bit) von Register `rA` an Adresse `I`
`shi rA, I`	Speichere niederwertigere 16 Bit aus Register `rA` an Adresse `I`
`sbi rA, I`	Speichere niederwertigere 8 Bit aus Register `rA` an Adresse `I`
`lwi rA, I`	Lade 32 Bit von Adresse `I` in Register `rA`
`lhui rA, I`	Lade 16 Bit, vorzeichenfrei interretiert, von Adresse `I` niederwertig in Register `rA`
`lhui rA, I`	Lade 8 Bit, vorzeichenfrei interpretiert, von Adresse `I` niederwertig in Register `rA`

Tabelle 1: Auszug des MicroBlaze Befehlssatzes für absolut adressierte Zugriffe auf den Speicheradressraum (32-Bit Architektur)

`sw rA, rB, rC`	Speichere Inhalt (32 Bit) von Register `rA` an die Adresse welche sich als Summe der Inhalte der Register `rB` und `rC` ergibt
`sh rA, rB, rC`	Speichere niederwertige 16 Bit aus Register `rA` an die Adresse, welche sich als Summe der Inhalte der Register `rB` und `rC` ergibt
`sb rA, rB, rC`	Speichere niederwertige 8 Bit aus Register `rA` an die Adresse, welche sich als Summe der Inhalte der Register `rB` und `rC` ergibt
`lw rA, rB, rC`	Lade 32 Bit von der Adresse, welche sich als Summe der Inhalte der Register `rB` und `rC` ergibt, in Register `rA`
`lhu rA, rB, rC`	Lade 16 Bit von der Adresse, welche sich als Summe der Inhalte der Register `rB` und `rC` ergibt, vorzeichenfrei interpretiert, niederwertig in Register `rA`
`lbu rA, rB, rC`	Lade 8 Bit von der Adresse, welche sich als Summe der Inhalte der Register `rB` und `rC` ergibt, vorzeichenfrei interpretiert, niederwertig in Register `rA`

Tabelle 2: Auszug des MicroBlaze Befehlssatzes für indirekte adressierte Zugriffe auf den Speicheradressraum (32-Bit Architektur)

LogiCORE IP XPS Interrupt Controller, beschrieben durch [26]. Da der Microblaze nur ein Interrupt-Signal kennt, werden die Interrupts, maximal 32 weiterer Peripheriegeräte über diesen Mikrocontroller auf das CPU-Signal gemultiplext. Dies nutzt ein eingebauter LogiCORE IP XPS Timer/Counter, ein Mikrocontroller welcher unter Anderem, wie in [25] beschrieben, als Zeitgeber fungiert. Der Zeitgeber wird später von der Software zum Zeitmultiplexen der Ausführungskontexte auf dem Microblaze genutzt. Weiter enthalten ist ein Mikrocontroller LogiCORE IP XPS UART Lite. Dieser bietet, wie in [28] beschrieben, asynchronen Datenaustausch nach dem *Universal Asynchronous Receiver Transmitter* Protokoll, kurz UART, über die RS-232-Schnittstelle des Boards. Die bisher genannten Komponenten, Interrupt-Controller, Zeitgeber und UART-Controller, sollen Software-seitig über MMIO angesprochen werden. Zu diesem Zweck integrieren sie den entsprechend genutzten, lokalen Speicher über den Datenbus Processor Local Bus von IBM nach [30], in den Adressraum des MicroBlaze.

Zur Kommunikation mit dem DDR2-SDRAM Modul des Boards enthält das SoC einen LogiCORE IP Multi-Port Memory Controller. Dieser integriert, wie in [27] beschrieben, den Speicher des Moduls mittels des Xilinx Cache Link in den Adressraum des MicroBlaze. Die Anbindung durch den XCL ermöglicht zur Laufzeit die dynamische Zwischenschaltung von Cachespeichern, für Daten- und Instruktionstransfers mit dem DDR2-SDRAM Modul. Unter Anderem zur Umsetzung dieser Cachespeicher besitzt der MicroBlaze zudem, über ein zusätzliches Bussystem, den Local Memory Bus, Zugriff auf ein lokales RAM-Speichermodul. Dieses besitzt zwar weit geringere Zugriffslatenzen als der DDR2-SDRAM, ist jedoch in der Kapazität stark begrenzt, da es durch FPGA-interne Komponenten realisiert wird.

4.2.3 Der Mikrokern

Die Genode Betriebssystemarchitektur implementiert bereits konzepteigen Funktionalitäten, wie eine Speicherverwaltung und eine RPC-Methodik, welche in anderen Architekturen eher dem Kern zugeordnet werden. Aus diesem Grund habe ich mich entschlossen einen Mikrokern für den Microblaze zu entwerfen, welcher die Vertrauensbasis des privilegierten und physischen Modus an den Anforderungen von Genode minimiert. Der Kern beschränkt sich auf den privilegierten und physischen Modus, arbeitet in nur einem Ausführungskontext und besitzt einen statischen Ressourcenverbrauch. Durch diese Eigenschaften entfallen Risiken wie kerninterne Ressourcenknappheit und Wettlaufsituationen. Der Kern ist in C++ und Maschinensprache geschrieben. Eine schematische Darstellung der grundlegenden statischen Objekte und Funktionen des Kerns liefert Abbildung 9.

Der Kern implementiert pseudoparallele Ausführung über mehreren Ausführungskontexten nach dem Round-Robin-Verfahren. Die Ausführungskontexte werden in Thread-Objekte gekapselt, in einem statischen Thread Allokator Objekt verwaltet. Jeder Platz des Allokators ist fest mit einer eindeutigen Thread-ID verknüpft welcher gleichzeitig zur Indentifikation des entsprechenden Ausführungskontextes, auch über den Kern hinaus dient.

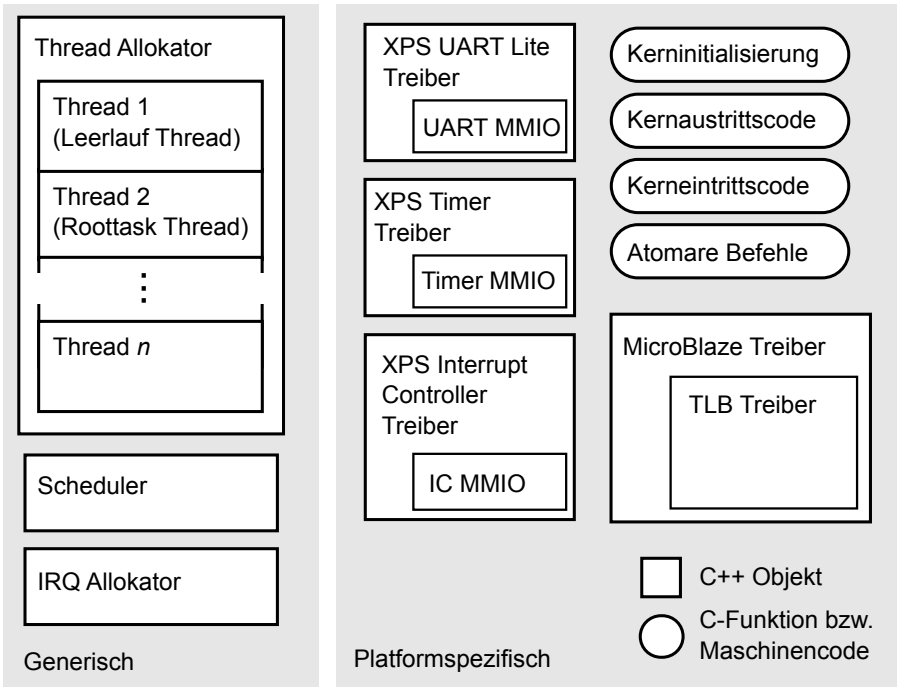

Abbildung 9: Grundlegende Statische Objekte und Codeparts des MicroBlasze Mikrokerns für Genode

Nachdem der Kerninitialisierungscode den Kernkontext und den Kernspeicher initialisiert hat, wird das statische Scheduler-Objekt erzeugt. Dies impliziert die Erstellung zweier Thread-Objekte, eines zur Leerlaufausführung und den sogenannten Roottask Thread, welcher den initialen Userland-Code zur Ausführung bringt.

Das Scheduler-Objekt fängt nun mit diesen Thread-Objekten an, über die statischen Treiberobjekte zum MicroBlaze und dem Zeitgeber, die entsprechenden Ausführungskontexte auf der CPU zu zeitmultiplexen. Ein Userland-Durchlauf startet mit der Übermittlung des Ausführungskontextes an den MicroBlaze-Treiber welcher diesen zum Laden vorhält. Daraufhin wird der Zeitgeber über den entsprechenden Treiber auf die Zeitscheibe des Ausführungskontextes eingestellt und sein Interrupt-Signal über den Treiber des Interrupt Controllers freigegeben. Der Kernaustrittscode läd nun den eingestellten Ausführungskontext, startet den konfigurierten Zeitgeber und springt in den virtuellen User-Modus. Die Ausführung im Userland kann, durch eine Nutzeranweisung, präventives Eingreifen der CPU bei Fehlverhalten oder das Interrupt-Signal der CPU unterbrochen werden. In diesem Fall gerät die Ausführung, wieder im physischen, privilegierten Modus, in den Kerneintrittscode. Dieser wertet die Ursache der Unterbrechung aus und sichert die nötigen Informationen zusammen mit dem Ausführungskontext zurück in das entsprechende Thread-Objekt. Abschließend stellt der Kerneintrittscode den Kontext des Kerns wieder her, welcher dann erneut den Scheduler instruiert.

Da der MicroBlaze aufgrund getrennter Daten- und Instruktionsbussysteme, keinen atomaren Befehl zum Lesen, Vergleichen und Schreiben eines Wertes zur Verfügung stellt, setzt der Kern dies manuell um. Dazu existiert ein Speicherbereich, welcher in jedem Prozess auf den selben physischen Bereich und nur lesbar aufgelöst wird. Der physische Hintergrundspeicher wird vom Kern mit der entsprechenden Funktion belegt, und im Folgenden jeder Ausführungskontext bei Unterbrechungen auf den Befehlszeiger getestet. Liegt dieser in dem genannten Bereich, gelangt der entsprechende Ausführungskontext unmittelbar zurück zur Ausführung. Daraufhin wird er jedoch direkt unterbrochen sobald er anstrebt den Bereich wieder zu verlassen. Mittels des so umgesetzten atomaren Tests auf Speicherwerte setzt Genode das Thread-übergreifende Sperren kritischer Ressourcen um.

Um die RPC-Schnittstelle von Genode umzusetzen, stellt der Kern zwei Syscalls zur Verfügung, welche Interprozesskommunikation ermöglichen. Der Syscall `ipc_request` nimmt den aufrufenden Ausführungskontext, desweiteren als Sender bezeichnet, von der Ausführung aus und propagiert sein Bestreben, eine Nachricht an den, per Thread-ID angegebenen Ausführungskontext zu senden. Instruiert ein Ausführungskontext, desweiteren Empfänger, den Syscall `ipc_serve`, wird er ebenfalls von der Ausführung ausgenommen bis sich ein anderer Ausfürhungskontext findet, welcher bestrebt ist, ihm eine Nachricht zu senden. Daraufhin erhält er die Nachricht und kommt zur Ausführung. Instruiert er den Syscall `ipc_serve` nun erneut, wird die erstellte Antwort automatisch an den Sender versandt und dieser geweckt. Mehrere Sender werden an einem Empfänger in der chronologischen Reihenfolge der Anfragen abgearbeitet. Zum Senden und Empfangen besitzt jeder Ausführungskontext ein Stück Speicher, welches statisch an ihn gebunden, und dem Kern bekannt ist. Sender sind somit selbstverwaltend, so dass, zum Beispiel Denial of Service Attacken erschwert werden.

Der Kern bietet zudem Unterstützung für Genodes dynamisches Speichermanagement durch dedizierte Ausführungskontexte. Zu diesem Zweck implementiert er Syscalls, um über den eigenen TLB-Treiber Adressauflösungen zum Translation Lookaside Buffer des MicroBlaze hinzuzufügen, als auch aus diesem zu entfernen. Desweiteren kann jedem Ausführungskontext, auch Faulter genannt, dynamisch per Syscall ein anderer Ausführungskontext, auch Pager genannt, zugeordnet werden. Der Pager wird dann informiert werden, wenn der Faulter einen sogenannten Seitenfehler auslöst. Das heißt dass er durch Zugriff auf eine virtuelle Adresse unterbrochen wird, weil noch keine entsprechende Auflösung im TLB existiert. Der Faulter bekommt dies wie eine normale IPC-Nachricht vermittelt wenn er den Syscall `ipc_serve` instruiert. Den Faulter kann er dann über den Syscall `thread_wake` wieder zur Ausführung bringen, sobald die Auflösung von ihm geladen worden ist.

Die Rechteverwaltung bezüglich der Nutzung von Syscalls, wird derzeit nur durch die Unterscheidung privilegierter und nicht-privilegierter Ausführungskontexte umgesetzt. So können privilegierte Ausführungskontexte jeden Syscall durchführen, während nicht-privielgierten Kontexten im Allgemeinen lediglich solche Syscalls erlaubt sind, welche kein Sicherheitsrisiko für andere Prozesse darstellen. Dies umfasst unter Anderem die IPC-Syscalls. Der erste Ausführungskontext für den initialen Userland-Code ist

privilegiert. Alle weiteren Ausführungskontexten können nur von privilegierten Ausführungskontexten als privilegiert oder nicht-privilegiert gekennzeichnet werden. Gehört ein Ausführungskontext dem virtuellen Adressraum des ersten Ausführungskontextes an und besitzt keinen Ausführungskontext zur Auflösung seiner Adresszugriffe, werden diese vom Kern automatisch identisch zum physischen Adressraum aufgelöst. Dies ist nötig damit der erste Prozess im Userland mit nur einem initialen Ausführungskontext anlaufen kann.

4.2.4 Oberhalb des Mikrokerns

Es kam gerade zum Ende der Entwicklung dieses Szenarios zu Schwierigkeiten mit der Kompilierung durch die Microblaze Toolchain, welche lediglich durch Xilinx selbst gepflegt wird. Da die Probleme im Zeitrahmen nicht beseitigt werden konnten, habe ich mich entschlossen, kurzfristig und im Nachinein auf eine andere Plattform zu wechseln. Die Wahl fiehl dabei auf den Cortex-A9 welcher die ARM Architektur v7-AR implementiert. Diese Architektur kommt der des Microblaze recht nahe, folgt ebenfalls dem RISC Prinzip und verwendet im Normalfall eine statische Befehlslänge. Außerdem existiert mit dem Fiasco OC Microkernel der TU Dresden als Basis, eine entsprechende Portierung von Genode auf diese Architektur. Die Differenzen der Implementierung zu der Microblaze-Variante lassen sich insgesamt, mit zwei plattformspezifischen C++ Methoden, gut in eine, entsprechend auszutauschende Quelldatei kapseln. Die Umsetzung habe ich so gestaltet dass sich diese Differenzen lediglich auf den `core`-Prozess auswirken, und der entscheidende konzeptuelle Teil somit unbedacht übernommen werden kann.

Genodes initial angestrebter Prozessbaum kann durch einen Konfigurationstext in der *Extensible Markup Language*, kurz XML, vorgegeben werden. Die Konfiguration für das Beispielszenario ist in Listing 1 dargestellt. Diese Konfiguration wird von Genodes erstem Prozess oberhalb des `core`-Prozesses, dem `init`-Prozess ausgelesen. Zuerst erscheint hier in `parent-provides` eine Auflistung der Dienste durch `service`, welche über den Elternprozess in Anspruch genommen werden können. Die Dienste sind:

- `ROM` - *Read Only Memory Service* zum Anfordern bestimmter lediglich lesbarer Speicherbereiche.
- `RAM` - *Random Access Memory Service* zum Anfordern beliebiger beschreibbarer Speicherbereiche.
- `IO_MEM` - *IO Memory Service*, zum Anfordern bestimmter MMIO-Speicherbereiche.
- `RM` - *Region Manager Service*, zum Einhängen der über `RAM`, `ROM` und `IO_MEM` angeforderten Speicherbereiche an eine Adresse im eigenen Adressraum, beziehungsweise zum Entfernen bereits einghängter Bereiche.
- `PD` - *Protection Domain Service* zum Erstellen, Konfigurieren und Destruieren weiterer Prozesse, implizit des zugehörigen virtuellen Adressraums.

- `CPU` - *Processing Unit Service*, zum Erstellen, Konfigurieren und Destruieren weiterer pseudoparalleler Ausführungskontexte in, per `PD` erstellten Adressräumen.
- `LOG` - *Log Service*, für die Textausgabe auf die Standartausgabe des Systems, in diesem Fall die UART-Schnittstelle.

Zu jedem weiteren Kindprozess welcher von `init` gestartet werden soll enthält die Konfiguration nun einen `start`-Block. Dieser deklariert über `ressource` die Ressourcen welche dem Prozess initial durch `init` zugewiesen werden sollen. Das Tag `binary` deklariert zudem das ELF-Image welches als Basis für den Prozess genutzt werden soll. Desweiteren gibt der Unterblock `route` an in welche Richtung Dienstanfragen von diesem Kindprozess weitergeleitet werden sollen. Der Tag `parent` deutet dabei auf den eigenen Elternprozess, also hier den `core`-Prozess, `child` hingegen auf den Kindprozess entsprechend des `name`-Parameters hin. Dies wird bei den Kindern `user_1` und `user_2` genutzt, deren Zugriff auf MMIO-Bereiche des Addierers, vom Prozess `server` verwaltet werden soll. Der MMIO-Bereich des Addierers ist in dem Szenario an Basisadresse `0x71000000` mit Größe `0x60` integriert, Werte, welche auch als Identifikation des angefragten Bereichs bei `IO_MEM`-Anfragen genutzt werden. Derzeit werden sämtliche `IO_MEM`-Anfragen der beiden Nutzer auf die VDM umgelenkt. Ein Modell welches anhand der Bereichsparameter des MMIO differenziert, ist geplant. Die abschließende `any-service`-Regel greift für Anfragen aller anderen Dienste und leitet diese hier Richtung Elternprozess.

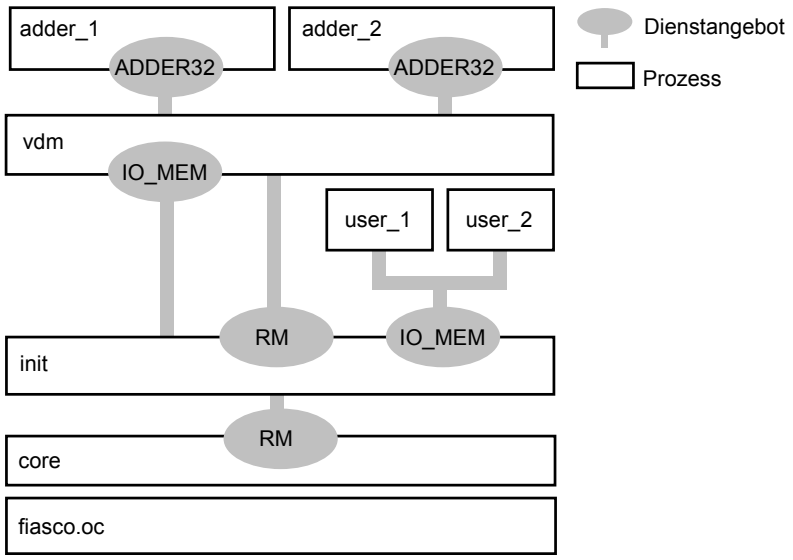

Abbildung 10: Überblick über die Prozesse und Dienste im Beispielszenario

Als Überblick zu den kommenden Erläuterungen soll Abbildung 10 dienen. Die Prozesse werden nacheinander, einzeln und eigenständig vertieft. Ausgenommen ist dabei der `init`-Prozess, da er unverändert übernommen wird und somit durch die Genode-Dokumentation erklärt ist. Das Selbe gilt für große Teile

des `core`-Prozesses, weshalb ich mich hier auf die wesentlichen Modifikationen beschränke.

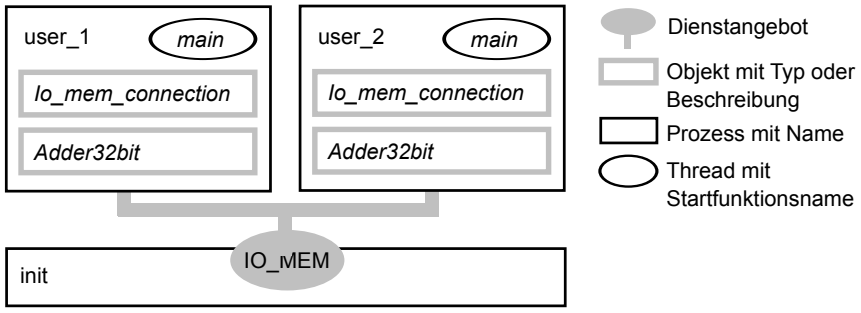

Abbildung 11: Die Nutzerprozesse mit den wichtigsten lokalen Objekten und Threads

Nutzer

Wie aus der Konfiguration ersichtlich, werden von `init` drei weitere Prozesse gestartet, `user_1` und `user_2` je vom Programm `test/vadd/client`, sowie eine Instanz `vdm` vom Programm `test/vadd/server`. Die `client`-Prozesse sollen je einen Nutzer des Addierers darstellen, welche um dessen Ressourcen konkurrieren. Einen Überblick über ihre wichtigsten Objekte, Threads und Dienstabhängigkeiten der Nutzer bietet Abbildung 11. Im Folgenden werden diese anhand von Code-Ausschnitten erklärt. Die Hauptroutine `main` ist in Listing 2 dargestellt. Zeile 25 fordert in Form einer `IO_MEM`-Verbindung Zufriff auf die MMIO-Bereiche des Addierers. `Io_mem_connection` dient dabei als Hülle für eine Verbindung zu dem entsprechenden Dienst des Elterprozesses. Als Rückgabe erhält der Prozess im Erfolgsfall eine Capability, welche ihm die entsprechenden Rechte zugesteht. In den folgenden zwei Zeilen wird diese genutzt, um sich bei einer `RM`-Verbindung auszuweisen. `env` stellt hier den Zugriff auf die Verbindung bereit, welche während der Prozesserzeugung über den Elternprozess bereits hergestellt wurden und zur sogenannten Enviroment des Prozesses gehören. Die instruierung von `attach` bewirkt nun, durch den `RM`-Dienst, dass der entsprechende MMIO-Bereich desweiteren in den lokalen Adressraum integriert ist. Die resultierende lokale Basisadresse liefert der Aufruf als Rückgabewert, auf welchem dann eine Instanz von `Adder32bit`, dem Treiber des Addierers, erzeugt wird. Die in Zeilen den 3-20 beschriebene `Adder32bit`-Klasse implementiert aufbauend auf der MMIO Basis Adresse nun eine einfache Summenberechnung `sum` zweier 32 Bit Werte. Die Schleife in den Zeilen 29-36 verdeutlicht dann, mit Hilfe von `sum`, die Funtionsweise des Addierers, indem sie die naive Rechnung des Kleinen Gauß bis 5 auf der Standartausgabe prozessiert. Abschließend geht der Nutzer in Zeile 36 in den Leerlauf über.

Core

Beim ersten Zugriff auf eine Adresse innerhalb des Addierer MMIO-Bereichs in Zeile 14, löst ein Nutzer einen Seitenfehler am Prozessor aus, da dieser keine Auflösung dazu besitzt. Als erster Genode-Prozess wird `core` davon durch den Kern informiert. Damit der Virtual Device Monitor den Zugriff emulieren kann

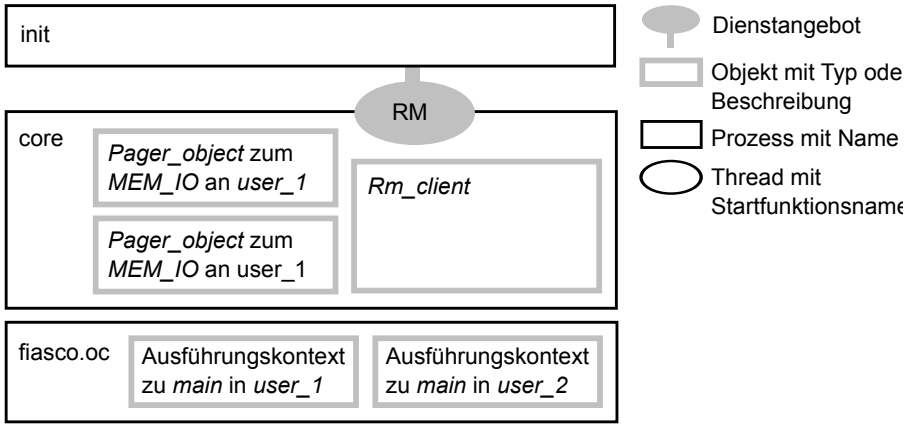

Abbildung 12: Der `core`-Prozess mit den wichtigsten lokalen Objekten und Threads

benötigt er Information darüber, welcher Wert wohin geschrieben werden sollte, beziehungsweise von wo gelesen werden sollte. Außerdem muss die Möglichkeit bestehen, im Fall des Lesezugriffs, den gelesenen Wert entsprechend zurück in den Kontext des Nutzers zu schreiben. Da diese Funktionalitäten inhärent sicherheitskritisch und architektur-, beziehungsweise kernabhängig sind, habe ich mich entschieden, diese möglichst hardwarenah in den `core`-Prozess auszulagern und dem VDM durch eine generischen Schnittstelle zur Verfügung zu stellen. Genodes Betriebssystemkern wird in Abbildung 12 vertieft, hier sind die wichtigsten Objekte, Threads und Dienstabhängigkeiten der Prozesse aufgeführt, welche im Folgenden erklärt werden. Für die Umsetzung der Emulationsschnittstelle zum VDM bietet sich die Einbettung in das Konzept der sogenannten verschachtelten `RM`-Dienste an. Im Normalfall wird das gesamte Speichermanagement in Genode durch den `RM`-Dienst von `core` und die darin eingehängten "normalen" Dataspaces abgebildet, da eine Assoziation der Kernbenachrichtigungen zu Seitenfehlern mit anderen Prozessen nicht vorgesehen ist. Ein Prozess besitzt jedoch die Möglichkeit eine eigene `RM`-Verbindung als `Dataspace` zu vermitteln und sich selbst bei `core` als Pager für diesen einzutragen. Das heißt, als Pager bekommt er in Folge eine asynchrone Benachrichtigung über solche Seitenfehler generiert, welche nicht von `core` aufgelöste werden können.

Der Vorgang dazu ist im Detail wie folgt. Wirft der Nutzer des verschachtelten Dataspace auf diesem einen Seitenfehler, instruiert `core` wie bei jedem eingehenden Seitenfehler die Methode `Rm_client::pager` welche in Listing 12 dargestellt ist. Die entsprechenden Informationen über den Seitenfehler stellt dabei die angegebene `Ipc_pager`-Instanz bereit. Entscheidend ist daraufhin der Funktionsaufruf `lookup_attachment` in Zeile 24. Er ermittelt, ob bereits eine Auflösung der Fehleradresse `pf_ip` durch das Speichermanagement gegeben ist. Die `RM`-Verbindung `curr_rm_session`, hier diejenige zu `core`'s `RM` Dienst, wird dazu initial nach dem vorhandenen Fehlerbereich im Zieladressraum `dst_fault_area` befragt. Ist die Auflösung verschachtelt, wie beschrieben, liefert dies einen Dataspace welcher die `RM`-Verbindung des ersten Pagers darstellt. Die Auflösung auf dessen RM-Verbindung könnte erneut verschach-

telt sein. Die Funktion iteriert also in die Verschachtelungen bis ein "normaler" Dataspace gefunden wird und hinterlegt die nötige Tiefe der Verschachtelungen in `level`, sowie den Erfolg der Suche in `lookup`. Ist letzteres abschließend gegeben, so enthält `src_fault_area` daraufhin den, in den `core`-Adressraum aufgelösten Bereich und `src_dataspace` den entsprechenden Dataspace. Für das VDM Szenario interessant ist nun der Fall dass keine Auflösung existiert, wie durch Zeile 32 gegeben, und spezifischer der Fall, wie durch Zeile 40-42 gegeben, dass es sich um einen verschachtelten Dataspace handelt, auf welchem ein Schreib- beziehungsweise Lesezugriff stattfinden soll. Dieser Ausgang von `lookup_attachment` deutet darauf hin, dass der Maschinenbefehl, welcher als Ursache des Fehlers gilt, auf höherer Ebene emuliert werden könnte. Ziel ist es in Zeile 68-70, da keine Auflösung existiert, die `RM` Verbindung des Pagers, also `curr_rm_session`, in den Fehlerzustand inklusive aller nötigen Informationen zu versetzen. So, dass der Pager in der Lage ist, den Seitenfehler durch eine Auflösung oder Emulation zu behandeln.

Der `core` Prozess lässt dafür ein weiteres `lookup_attachment` in Zeile 51-52 durchlaufen. Diesmal wird jedoch jedoch versucht, den Adressbereich zum aktuellen Befehlszeiger des Nutzers, `ip_dst_area`, aufzulösen. Die resultierende `ip_src_area` verweist im Erfolgsfall `core`-lokal auf den Maschinenbefehl, welcher den Seitenfehler ausgelöst hat und wird zur Ermittlung weiterer Befehlsparameter genutzt. Handelt es sich um einen Schreibzugriff wird in Zeile 60-61 die Architekturspezifische Funktion `decode_instruction` genutzt um den zu schreibenden Wert für `curr_rm_session` in `pf_writes` zu hinterlegen. Die ARM-spezifische Implementierung dieser Funktion ist in Listing 11 dargestellt. Sie ermittelt, über den Befehlscode `arm_instr`, in Zeile 27 den Index `reg` des Quellregisters. Dann instruiert sie den Kern in Zeile 30-31, um den Schreibwert `writes_value` aus dem gespeicherten Ausführungsontext des inaktiven Nutzers zu lesen. Den dazu nötigen Systemaufruf `l4_thread_ex_regs_ret` habe ich im Kernel modifiziert, da er eigentlich nur zum Lesen spezieller Register gedacht ist. Handelt es sich in Listing 12, Zeile 65 hingegen um einen Lesezugriff, muss später eventuell ein emulationsbasiert gelesener Wert in ein Zielregister des Nutzers geschrieben werden. Da der Befehl dann ebenfalls dekodiert werden muss, wird der lokal aufgelöste Befehlszeiger `pf_instr` aus `ip_src_area` an `curr_rm_session` übergeben. Der VDM kann diese Informationen nun aus dem `Rm_session::State`, wie in Listing 14 deklariert, über die RPC-Methode `Rm_session::state` auslesen.

Im Normalfall löst ein Pager nun den Seitenfehler an `curr_rm_session`, durch das Einhängen eines entsprechenden Dataspace, mittels `Rm_session::attach` auf. Dann würde der `RM` Dienst dies erkennn, die Auflösung an die Hardware übermitteln und daraufhin den Nutzer wieder aktivieren. Im Zuge einer Emulation ist die Einhängung eines Speicherbereichs nicht nötig, deshalb erweitere ich die `RM`-Schnittstelle um die RPC-Methode `processed`, wie in Listing 14 ersichtlich. Sie nimmt vom Pager nach einer erfolgreichen Emulation einen `Rm_session::State`, entsprechend des emulierten Befehls als Identifikator entgegen. Dies hat die Bewandnis dass ein `RM` Dienst mitunter mehrere Seitenfehler gleichzeitig behandeln kann. Diesen Fall schließe ich zwar später durch das Modell aus, behalte die Behandlung hier jedoch konformitätshalber bei. In `processed` wird nun zuerst getestet ob die architekturspezifische Methode

`Pager_object::instruction_processed` aus Listing 11 für den Nutzer erfolgreich ist. Ist dies der Fall so konnte die Methode den Ausführungskontext des Nutzers bezüglich der Emulationsergebnisse aktualisieren. Der Methode wird der `core`-lokale Befehlszeiger des Nutzers in `instr` und, bei einem Lesezugriff, der gelesene Wert aus `Rm_session::State` in `read` übergeben. Sie versucht dann, in Zeile 51 `ip`, den nutzerlokalen Befehlszeiger des Nutzers, durch den Kernel lesen zu lassen. Dieser wird durch Zeile 65 und 68 auf den nächsten Befehl erhöht und dann durch den Kernel zurück geschrieben. Handelt es sich zudem um einen Lesenden Befehl, ermittelt `Arm::Instruction::reads`, in Zeile 55, den Index des Zielregisters. Daraufhin wird der Kernel erneut instruiert, um den gelesenen Wert, in Zeile 58-59, in den gesicherten Ausführungskontext des Nutzers zu schreiben. Ist dies alles erfolgreich, ist der Nutzer im Nachinein in dem Status, den die native Ausführung des letzten Befehls impliziert hätte. Er wird daraufhin von `Rm_session::processed`, wie auch in `Rm_session::attach`, wieder zur Ausführung freigegeben.

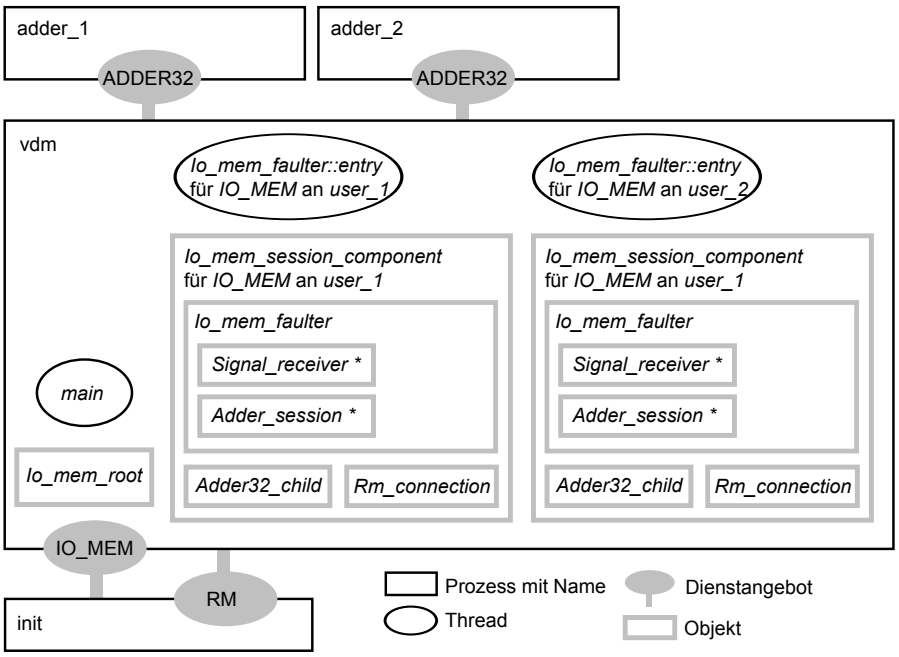

Abbildung 13: Der `vdm`-Prozess mit den wichtigsten lokalen Objekten und Threads

VDM

Nun, da über den `RM`-Dienst an `core` die Voraussetzungen zur generischen Behandlung eines Seitenfehlers vorhanden sind, wende ich mich dem Aufbau des darauf aufsetzenden VDM zu. Abbildung 13 gibt einen Überblick über die wichtigsten Objekte, Threads und Dienstabhängigkeiten des `vdm`-Prozesses. Seine Hauptroutine `main` ist in Listing 3 aufgeführt. Ihr einziges Ziel ist es, in Zeile 12 ein `Root`-Objekt für den `IO_MEM`-Dienst zu erstellen, um damit das Dienstangebot, in Zeile 14, beim Elternprozess bekannt zu machen. Die Klasse dieses RPC-

Objekts ist von Root_component abgeleitet, dessen RPC-Schnittstelle in Listing 4 dargestellt ist. Entscheidend für dieses Beispiel ist ihre Methode session, welche eine neue Verbindung erstellt und eine entsprechende Capability dazu liefert. Zu Erzeugung des Root-Objekts wird ein weiterer Ausführungskontext, ein sogenannter Rpc_entrypoint benötigt. Dieser empfängt, parallel zur Hauptroutine, RPC-Anfragen an das Root-Objekt des Dienstes und verarbeitet sie. Er wird in Zeile 10-11 des Listings 3 erzeugt und benötigt eine Verbindung zum CAP-Dienst cap des Elternprozesses. Diesen benötigt er um Capabilities zu den IO_MEM-Verbindungen zu erzeugen, welche er dann an die Anfragenden ausliefert. Abschließend deaktiviert sich der Hauptausführungskontext des VDM in Zeile 15.

Liegt nun die, vom Nutzerprozess gestellte Anfrage nach IO_MEM vor, leitet init diese, laut seiner XML Konfiguration, an den VDM weiter. Dieser erzeugt in Io_mem_root::session eine entsprechende Instanz von Io_mem_session_component, welche dann durch die resultierende Verbindungs-Capability assoziiert wird. Die Klasse ist in Listing 5 aufgeführt. Ihr Konstruktor verfolgt zwei Ziele. Zuerst müssen die Seitenfehler des Nutzers auf dem IO_MEM-Bereich empfangen und ausgewertet werden. Dazu wird in Zeile 30 die verschachtelte RM-Verbindung _rm mit Seitengröße erzeugt. Sie wird somit, bei entsprechenden Seitenfehlern, als curr_rm_session nach dem ersten lookup_attachment in core resultieren. Empfänger der Seitenfehler, also Pager, soll der Ausführungskontext hinter _fault_handler, in Zeile 13 werden. Er erhält dazu einen einen sogenannten Signal_receiver für das asynchrone Signal, welches core generiert, um einen Seitenfehler zu propagieren. Ein Signal_receiver blockiert den Anwender auf den Empfang bestimmter Typen von Signalen. Typisiert werden Signale durch einen Signal_context. Die Methode manage in Zeile 66 bewirkt, dass _fault_sreceiver Signale vom Typ _fault_scontext entgegennimmt. Sie liefert zudem eine Capability, welche den Signaltyp und das Signalziel assoziiert und durch die Methode fault_handler an _rm zu core propagiert. Dieser instruiert die Capability später um die Seitenfehler-Signal an den VDM zu senden.

Nun da die Signale von _fault_handler empfangen werden können, gilt es diese entsprechend zu behandeln. Dazu ist, wie aus dessen Konstruktor in Zeile 64 ersichtlich, _adder32_session, der zweite Hauptbestandteil der Io_mem_session nötig. Es handelt sich dabei um eine Verbindung zu dem Dienst ADDER32, welcher, über eine möglichst generische RPC-Schnittstelle, MMIO-Zugriffe serverseitig emuliert. Er stellt somit die Schnittstelle zwischen VDM und Emulatorprozess dar. Seine Deklaration wird durch die entsprechende Adder32_session_component-Implementierung in Listing 10 ersichtlich. Sie beschränkt sich auf die beiden Funktionen read_mmio, welche den gelesenen Wert liefert, sowie write_mmio, welche den Schreibwert als Parameter value entgegennimmt. Geplant ist diese zu einem MMIO-Emulationsdienst EMMIO für beliebige Geräte zu generalisieren, welchen ADDER32 dann spezialisieren würde. Den, mit der IO_MEM-Verbindung assoziierten Emulator soll der VDM als Kindprozess erzeugen. Dazu besitzt Io_mem_session_component die Member-Variablen _adder32_child, _adder32_elf und _adder32_elf_ds, wie aus Zeile 15-18 des Listings 5 ersichtlich. Letztere beiden stellen in Zeile 34-40, über den ROM-Dienst von core, den Zugriff auf das ELF-Image des Emu-

lators her. Der Adder32_child-Konstruktor in Zeile 45-47 nimmt das Resultat und einen durchnummerierten Namen, wie adder_1, entgegen und startet durch die Genode-Klasse Child in Listing 7, Zeile 35-36, einen entsprechenden Kindprozess. Der Konstruktor stellt zudem sicher, dass eine Anfrage auf den ADDER32-Dienst erst stattfindet, wenn der neue Kindprozess diesen erfolgreich initialisiert hat. Dazu wird in Zeile 34 das Lock _ready gesperrt initialisiert. Daraufhin wird, in Zeile 38, durch _activate an _entrypoint, der Dienst des VDM Parent-Objekts für den Emulator eröffnet. Dadurch wiederum kann dieser seinen Dienst anmelden und impliziert damit die Funktion announce_service aus Zeile 44-53 am VDM. Wichtig ist dass diese parallel zu dem Konstruktor im Entrypoint-Kontext des Parent-Dienstes ausgeführt. Sie vermekt sich in Zeile 49 das Root-Objekt des neuen ADDER32 Dienstes und gibt in Zeile 50 das Lock wieder frei. Dadurch kann der Konstruktor, welcher in Zeile 39 auf das Lock wartet, fortfahren. Zu dem erhaltenen Root-Objekt stellt die Io_mem_session_component, in Listing 5, Zeile 49-60, nun eine RPC-Verbindung _adder32_session her. Diese wird dem _fault_handler Konstruktor, neben dem bereits beschriebenen fault_sreceiver in Zeile 62-64 übergeben. Die Klasse Io_mem_fault_handler ist in Listing 6 beschrieben. Ihre Methode entry wird in einem neuen Ausführungskontext parallel gestartet. Sie wartet in Zeile 52 durchgängig auf das Eintreffen von Seitenfehler-Signalen und instruiert für jedes Signal in Zeile 53 die Funktion handle_fault. Diese liest in Zeile 19 den Status der verschachtelten RM-Verbindung aus, auf welcher der Fehler auftrat, und instruiert dann in Zeile 22-24 beziehungsweise 30-32 den ADDER32 Dienst des entsprechenden adder_x-Prozesses. Abschließend wird, insofern die Instruktion emuliert werden konnte Zeile 43 erreicht, welche dem RM-Dienst den Seitenfehler als prozessiert meldet. Daraufhin kommt in unserem Fall der besprochene Pfad in core zum Einsatz, welcher den Nutzer reaktiviert.

Für jede IO_MEM-Verbindung existieren also 3 parallel laufende Ausführungskontexte. Einer wird durch Io_mem_session impliziert, um auf Anfragen des Klienten zu blockieren. Einen Weiteren erzeugt Io_mem_fault_handler, um auf Seitenfehler-Signale von core zu warten. Der dritte ist der, welcher durch den Parent-Dienst des VDM für den Emulator der Verbindung erzeugt wird, um diesbezüglich auf Anfragen durch den Emulator zu blockieren. Desweiteren besitzt die VDM noch seinen Hauptausführungskontext, welcher abschließend inaktiv ist, und einen Ausführungskontext welcher auf RPC-Anfragen zu der Io_mem_root-Instanz wartet.

Emulator

Ein Überblick über die wichtigsten Objekte und Dienstabhängigkeiten der Emulatoren des Szenarios ist durch Abbildung 14 gegeben. Die Hauptroutine eines Addierer-Emulators ist in Listing 8 aufgeführt. Ihr einziges Anliegen ist die Erzeugung eines Root-Objektes des ADDER32-Dienstes in Zeile 11, welcher dann in Zeile 13 beim Parent-Prozess angemeldet wird. Die Methodik ist analog zu den bereits besprochenen Genode-Diensten. Interessant bleibt die Implementierung der Dienst-Methoden, read_mmio und write_mmio, durch die Adder32_session_component in Listing 10. Die Klasse hält einen Speicherbereich _mmio in Größe des emulierten MMIO und projeziert darauf einen ebenso großen Adressbereich, welcher bei 0 startet. Die Basisadresse des emulierten MMIO ist somit nicht von Belang für den Emulator. Dem kommt der VDM

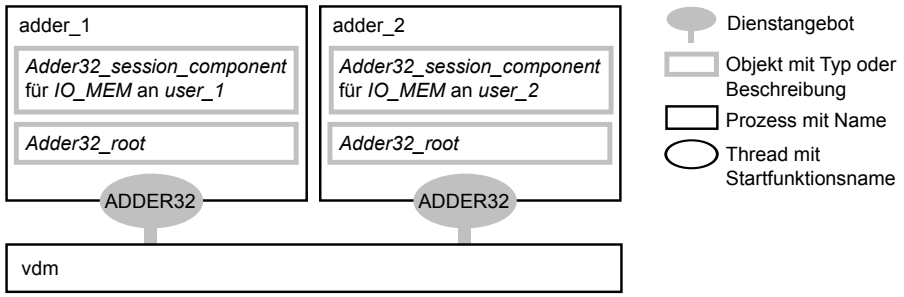

Abbildung 14: Die Emulatorprozesse mit den wichtigsten lokalen Objekten und Threads

nach, indem er die Adressen aus `Rm_session::State` unverändert durchreicht. Diese sind eben durch `core`, relativ zur Basis des verschachtelten Dataspace berechnet worden. Die beiden `ADDER32`-Methoden arbeiten ähnlich, je nach Format `a` des Zugriffs wird in den Zeilen 36, 43, 50, 76, 80, beziehungsweise 84 die Quelladresse des Zugriffs `ic` ermittelt. Im Falle eines Schreibzugriffs liegt diese im Parameter `value`, ansonsten in `_mmio`. Wird geschrieben, muss zudem eine Zieladresse `dest` innerhalb von `_mmio` durch die Zeilen 35, 42, beziehungsweise 49 ermittelt werden. Nun wird der Wert der Quelladresse in den Zeilen 77, 81 und 85 per RPC zurückgegeben, beziehungsweise in den Zeilen 38, 45 und 52 durch den Wert der Zieladresse überschrieben. Wurde dabei der Inhalt von `_mmio` verändert, kommt, wie in Zeile 61, noch eine emulatorspezifische Funktion zum Einsatz, welche die eigentliche Arbeit des Addierers wiederspiegelt. Sie aktualisiert den MMIO-Bereich der Summe in `_mmio` entsprechend der Summandenbereiche. Wie in Zeile 95 zu sehen, erfolgt hier die Addition, welche, durch die RPC-Synchronisierung des Dienstes, für andere Prozesse als unmittelbar erscheint.

Testlauf

In Listing 13 ist die Standartausgabe eines Testlaufs des beschriebenen Szenarios dargestellt. Der Testlauf wurde auf einer emulierten Umsetzung des Cortex-A9 in Qemu durchgeführt, da ein reales ARM-Board nicht zur Verfügung stand. Die Ausgabe des Fiasco OC Kernels sind ausgeblendet. Zu Beginn jeder Zeile ist in eckigen Klammern aufgeführt, von welchem Prozess die jeweilige Ausgabe stammt und vorangehend, welchem Hierarchiezweig er in Genodes Prozessbaum angehört. So stammt eine Ausgabe hinter `[init -> vdm -> adder_1]` von `adder_1`, einem Kindprozess von `vdm`, welcher wiederum Kindprozess von `init` ist. `core` wird dabei nicht aufgeführt, da er immer nur `init` zum Kind hat. In Zeile 12-24 sieht man die Ausgabe zum Start der Kindprozesse von `init` durch ebendiese. Darauf vermelden in Zeile 25 und 27 die beiden Nutzer je eine Anfrage nach einer `IO_MEM`-Verbindung, auf den Addierer-MMIO. Dies hat die Erzeugung von zwei `IO_MEM` Verbindungen an `vdm` zur Folge, wie Zeile 35 und 42 zeigen. Die beiden Verbindungen implizieren zudem je den Start eines Emulationsprozesses, hier `adder_1` und `adder_2`, deren Hauptroutinen sich in Zeile 30 und 37 zu Wort meldet. Sie erstellen im Folgenden, wie in den Zeilen 31, 33, 38 und 40 ersichtlich, den `ADDER32` Dienst und öffnen je eine Verbin-

dung dazu. Diese wiederrum werden von den `Io_mem_fault_handler` Objekten der beiden `IO_MEM` Verbindungen in `vdm` genutzt. Ist diese Basis einmal erzeugt, liefert die `vdm` wie in Zeile 43 und 45 ersichtlich, jeweils den `IO_MEM` Dataspace, welcher durch die Nutzerhauptroutinen, über die `IO_MEM` Verbindungen angefordert wurde. Nun wird bei den Nutzern der Addierertreiber instruiert, dessen Konstruktor in Zeile 44 und 48 Erfolg vermeldet. Ab Zeile 49 sieht man dann die Logausgabe der `vdm` über die einzelnen Schreib- und Lesezugriffe auf den Addierer MMIO-Bereichen, wie sie durch die Umsetzung des Kleinen Gauß, in den Nutzerhauptroutinen durchlaufen werden. Da zu jedem Nutzer ein Emulator mit eigenständigem Zustand existiert, stellt der pseudoparallele Zugriff auf die beiden, für die Nutzer nicht voneinander unterscheidbaren MMIO Bereiche, kein Problem dar.

5 Auswertung

In dieser Arbeit habe ich ein Modell zur transparenten Geräteemulation, auf der Systembusebene vorgestellt, welches auf moderne Betriebssystemkonzepte, wie virtualisierte Adressräume und Capability-basierte Rechteverwaltung aufbaut. Es bewahrt den Entwickler vor einer unnötigen Emulation der gesamten Architektur, in welche das Zielgerät eingebettet ist, so wie es bei heutigen Software-Emulatoren wie Qemu der Fall ist. Dazu nutzt es Kernmechanismen zur Emulation von Maschinenbefehlen, an gesicherten Ausführungskontexten. Die Emulation beschränkt sich dabei auf die Betriebsmittel und Befehle, welche zur Kommunikation mit dem Gerät genutzt werden. Dies spart neben Entwicklungs- und Pflegeaufwand auch Emulations-Overhead. Außerdem veringert es das Risiko von Ungenauigkeiten durch die Differenz zwischen Real- und Emulatorumsetzung. Das Modell bedarf zudem keiner speziellen Emulationshardware. Voraussetzungen an die Architektur bleiben jedoch. Einerseits muss die Software-seitige Auslösung von Interrupts möglich sein. Andererseits müssen Speicherzugriffe in einen Schreibzugriff mit einem lesbaren Quellregister, oder einen Lesezugriff mit einem schreibbaren Zielregister dekodierbar sein. Desweiteren benötigt man die Umsetzung virtueller Adressräume unter Software-seitiger Kontrolle.

Ein weiteres Merkmal dieses Modells ist die gute Beobachtbarkeit, sowohl des Emulators, als auch der Nutzerzugriffe auf dessen Ressourcen, durch den zwischengeschalteten Virtual Device Monitor. Hinzu kommt, dass dieser auch die Kontrolle über den Arbeitsfluss des Nutzers an dem emulierten Gerät hält. Dadurch entsteht das Potential, den VDM zur umfangreichem Fehlerbeseitigung, durch Techniken wie Speicherauswertung und -modifikation, Protokollierung, bedingte Haltepunkte und Einzelschrittverarbeitung des Nutzers am MMIO zu verwenden. Dabei bleibt die Sicherheit des Nutzers durch eine entsprechende Aufgabenverteilung wie folgt gewahrt. Die sicherheitskritischen Eingriffe in den Ausführungskontext des Nutzers können, wie im Beispielszenario auf Genode in den Kernprozess des Betriebssystems, hier `core` ausgelagert werden. Auf diesen Prozess muss der Nutzer in einer vergleichbaren Prozesshierarchie immer vertrauen können, da dieser auf sämtliche, von ihm verwandte Ressourcen Zugriff besitzt. Hält man, ebenfalls wie auf der Genode-Implementierung, auch die Dekodierung der Maschinenbefehle im Kernprozess des Betriebssystems, so

ist desweiteren, über den MMIO hinaus, keine implizite Rechtevergabe auf den Adressraum des Nutzers nötig. Für den Nutzer impliziert die, für ihn transparente Nutzung von emuliertem MMIO, also lediglich bezüglich der Rückgabewerte bei Lesezugriffen und der Reaktivierung seines Ausführungskontextes, Vertrauen in den VDM. Dies ist analog zu dem Vertrauen in die korrekte Funktionsweise eines realen Gerätependants. Durch die verlagerte Befehlsdekodierung wird der VDM zudem, abgesehen von den Schreib- und Lesewertformaten, generisch bezüglich der eingesetzten Architektur gehalten.

Das Modell lässt sich, durch die genannten Eigenschaften, zur Parallelisierung von Phasen der kombinierten Hard- und Softwareentwicklung nutzen. Existiert ein Modell der Geräteschnittstelle mittels MMIO und Interrupt-Signalen, so lässt sich dieses bereits vorläufig durch einen Emulatorprozess umsetzen. Die Implementierung eines speziellen Emulators, lässt sich, hinter einer recht generischen RPC-Schnittstelle, einfach und intuitiv gestalten, wie das Beispiel in Genode zeigt. Seiteneffekte durch die Nutzung des Geräts können durch den VDM protokolliert oder, gegebenenfalls, an dafür vorgesehene weitere Geräte übermittelt werden. Dadurch wird es zum Einen möglich, eine Validierung der Gerätespezifikation für die Hardwareentwicklung vorzunehmen. Zum Anderen kann in der parallel laufenden Softwareentwicklung bereits eine Verifikation des Treibers vorgenommen werden, insofern die Kommunikation zwischen Software und Gerät nicht auf beidseitigen Zeitannahmen basiert.

5.1 Leistungsvergleich

In diesem Abschnitt war ein beispielhafter Leistungsvergleich einer Geräteimplementierung, als FPGA-Konfiguration und als Emulationsprozess nach dem vorgestellten Modell geplant. Die FPGA-Umsetzung ist derzeit jedoch auf Microblaze-basierte Platformen beschränkt, auf welchen die Umsetzung des Emulationsmodells noch nicht lauffähig ist. Es bliebe somit ein Vergleich zwischen der Gerätenutzungsphase auf dem ARM-basierten Fiasco OC Kernel und ebendieser auf dem Microblaze-basierten Kernel. Ich habe mich gegen diesen Vergleich entschieden, da den beiden Kernen recht unterschiedliche Konzepte, zum Beispiel im Bezug auf das Scheduling der Ausführungskontexte und der Speicherverwaltung, zugrunde liegen. Dadurch, und natürlich durch die unterschiedlichen CPU-Architekturen, ergeben sich kaum berechenbare Differenzen in der Performanz der beiden Basisysteme, wodurch die Vergleichbarkeit der Szenarien diesbezüglich nicht gegeben ist.

5.2 Potential und zukünftige Herausforderungen

Eine wesentliches Problem welches bleibt, ist die umfangreiche Leistungsanalyse des vorgestellten Modells. Dazu ist geplant, die `Microblaze`-basierte Umsetzung des Modells abzuschließen, um ein möglichst breites Spektrum an Geräteumsetzungen als FPGA-Konfiguration, jeweils mit der emulierte Variante zu vergleichen. Ziel dieser Betrachtung sollte eine representative Gegenüberstellung der durchschnittlichen Dauer von Speicherzugriffsbefehlen, auf

realem und emuliertem MMIO sein. Dabei sollte zwischen Zugriffen auf Adressen, welche bereits aufgelöst wurden und solchen, welche am realen Gerät noch einen Seitenfehler erzeugen, differenziert werden. Die gerätespezifische Verarbeitungsdauer von Eingaben ist für die Gegenüberstellung nicht von Interesse. Es sollte daher beachtet werden, dass sie im Emulationsfall, auch bei asynchronen Geräteschnittstellen, durch die Synchronisation der VDM, vollständig Teil der Befehlsdauer ist. Aufbauend auf den Vergleich, können Möglichkeiten der Optimierung in der Implementierung, als auch in der Deklaration des Modells, evaluiert und umgesetzt werden. Ein solcher Vergleich der Latenzen bei Hardware-seitigen und emulierten, also Software-seitigen Interrupts, erscheint nur in besonderen Fällen sinnvoll, in denen das Betriebssystem oder die Architektur Software-seitig einen entscheidenden Mehraufwand bedingt.

Eine weitere Überlegung ist es, die RPC-Schnittstelle zwischen dem Emulator und dem VDM zu generalisieren. Die Richtung dazu wird bereits vom `ADDER32`-Dienst des Beispielszenarios gezeigt, welcher sich lediglich zweier geräteunabhängiger Funktionen, zum Lesen und Schreiben an MMIO-Adressen bedient. So wäre für eine vollständige Umsetzung des Modells, voraussichtlich noch eine RPC-Methode, angeboten durch den VDM, zur Handhabung von Interrupts nötig. Die Funktion würde von einem emulatorlokalen Adressraum für Interrupts, auf den globalen Interrupt-Adressraum abbilden und einen entsprechenden Interrupt für den Emulator auslösen. Desweiteren stellt sie eine Synchronisation des Emulators, mit der folgenden Interrupt-Freigabe, durch dessen Blockierung auf die RPC-Antwort her. Die Emulatorumsetzung eines konkreten Geräts würde sich dann auf Implementierung der Funktion beschränken, welche nach dem Schreiben an einer lokalen MMIO-Adresse instruiert wird. Im Beispielszenario ist dies `Adder32_session_component::_update_sum` aus Listing 10.

Existiert eine generische Schnittstelle zwischen Emulatoren und VDM, besteht zudem die Möglichkeit, den Virtual Device Manager, für die Emulation von Adressbereichen mit verschiedenen Gerätespezifikationen, als Virtual Bus Monitor zu generalisieren. Der VBM würde dann, der Zuordnung von Nutzern zu Emulatoren vorgelagert, eine Zuordnung der Adressen zu den Emulatorentypen entsprechend der Spezifikationen vornehmen. Die Differenzierung der Zugriffsrechte der Nutzer, würde bei diesem Modell in der Genode-basierten Implementierung, durch den Einsatz verschiedener Capability-gebundener Verbindungen zu den entsprechenden Adressbereichen gewahrt bleiben.

Offen bleibt auch eine konkrete Umsetzung der transparenten Einbindung realer Geräte, zur Nutzung des Modells für die Transparenz von Ressourcenverfügbarkeit. Auf Genode könnten die entsprechenden Verbindungen dazu, vor Erzeugung eines Emulators an `core` auf Verfügbarkeit der Ressource testen und diese dann direkt an den Nutzer vermitteln. Ein Ziel der Implementierung sollte dabei sein, den Zusatzaufwand durch die potentielle Emulation der Ressource, im Falle der Verfügbarkeit minimal zu halten, verglichen mit der Implementierung ohne potentielle Emulation. Dazu würde in dieser Variante beitragen, dass der VDM lediglich als Vermittler der entsprechenden Capabilities in Erscheinung tritt und bei den folgenden Zugriffen nicht mehr involviert würde.

Abschließend ist unter Genode vorgesehen, die Funktionalitäten, welche derzeit

in `core` angesiedelt sind, konzeptionell zu überdenken. Die kernel- und platformabhängigen Pfade sind zwar gut vom generischen Teil abgekapselt. Allerdings sollten diese Aufgabenfelder nicht in dem, sonst eher grundlegenden Funktionsumfang von `core` angesiedelt sein. Dazu zählt insbesondere die Erweiterung des `RM`-Dienstes. Der Umstand dass derzeit potentiell emulierte Ressourcen bei der Behandlung realer Ressourcen, wie in `Rm_client::pager` in Listing 12 mit berücksichtigt werden, wirft tatsächlich mehrere Probleme auf. So wird ein ständiger Zusatzaufwand in der zentralen Ressourcenverwaltung impliziert und die Wartung eines, bereits sehr komplexen Bestandteils des Betriebssystems erschwert. Es ist deshalb angedacht einen Prozess `emucore` als eine Art Aufsatz zu `core` zu entwerfen. `emucore` soll seinen Kindprozessen die selbe Dienste-schnittstelle bieten, wie `core` es täte. Bei Diensten, welche zur Geräteemulation speziell abgehandelt werden müssen, wie `IO_MEM`, schaltet sich der Prozess jedoch zwischen und handhabt die zusätzlichen Aufgaben. Alle anderen Dienste würde dieser Prozess direkt an `core` vermitteln, so dass, für den Prozessteilbaum unter `emucore`, der Zusatzaufwand diesbezüglich konstant bleibt.

Abkürzungsverzeichnis

ASIC	*Application-Specific Integrated Circuit*
CPU	*Central Processing Unit*
DDR2-SDRAM	*Double Data Rate Synchronous Dynamic Random Access Memory*
DPGA	*Dynamically Programmable Gate Array*
ELF	*Executable and Linking Format*
FPGA	*Field Programmable Gate Array*
IO	*Input-Output*
IPC	*Inter-Process-Communication*
JTAG	*Joint Test Action Group Interface*
LIM	*Local Instruction Memory*
MMU	*Memory Management Unit*
PMIO	*Port-mapped I/O*
SoC	*System-on-Chip*
RAM	*Random-Access Memory*
RTL	*Register-Transfer Level*
RPC	*Remote Procedure Call*
TME	*Time-Multiplexed Hardwaresided Logic Emulator*
UART	*Universal Asynchronous Receiver Transmitter*
VDM	*Virtual Device Monitor*
XCL	*Xilinx Cache Link*
XML	*Extensible Markup Language*

Anhang

```
<config verbose="yes">
  <parent-provides>
    <service name="ROM" />
    <service name="RAM" />
    <service name="CAP" />
    <service name="PD" />
    <service name="RM" />
    <service name="CPU" />
    <service name="IO_MEM" />
    <service name="LOG" />
    <service name="SIGNAL" />
  </parent-provides>
  <start name="vdm">
    <binary name="vadd_server" />
    <resource name="RAM" quantum="10M" />
    <provides> <service name="IO_MEM" /> </provides>
    <route><any-service><parent/></any-service></route>
  </start>
  <start name="user-1">
    <binary name="vadd_client" />
    <resource name="RAM" quantum="3M" />
    <route>
      <service name="IO_MEM">
        <child name="vdm" />
      </service>
      <any-service><parent/></any-service>
    </route>
  </start>
  <start name="user-2">
    <binary name="vadd_client" />
    <resource name="RAM" quantum="3M" />
    <route>
      <service name="IO_MEM">
        <child name="vdm" />
      </service>
      <any-service><parent/></any-service>
    </route>
  </start>
</config>
```

Listing 1: Auszug aus 'base–foc/run/vadd.run': XML Konfiguration für Genode im Beispielszenario

```
...

class Adder32bit {

  volatile uint32_t * _addend1;
  volatile uint32_t * _addend2;
  volatile uint32_t * _sum;

  public:
  ...

  uint32_t sum(uint32_t a, uint32_t b)
  {
    *_addend1 = a;
    *_addend2 = b;
    return *_sum;
  }

  ...
};

int main(int argc, char **argv)
{
  PINF("IO_MEM client to 0x71000000");
  Io_mem_connection adder_iom(0x71000000, Adder32bit::mmio_size());
  Adder32bit a((addr_t)env()->rm_session()
```

```
27                                  ->attach(adder_iom.dataspace()));
28
29    static unsigned i=2, j=1, k=0;
30    for(; j<=10;) {
31
32      k = a.sum(i,j);
33      PINF("Adder32 sum: %i + %i = %i", j, i++, k);
34      j = k;
35    }
36    while(1);
37  }
```

Listing 2: Auszug aus 'base–foc/src/test/vadd/client/main.cc': Hauptroutine eines Nutzers

```
1  ...
2
3  int main(int argc, char **argv)
4  {
5    PINF("Virtual device monitor");
6
7    static Cap_connection cap;
8    static Sliced_heap sliced_heap(env()->ram_session(),
9                                   env()->rm_session());
10   static Rpc_entrypoint
11          io_mem_ep(&cap, STACK_SIZE, "io_mem_pf_ep");
12   static Io_mem_root io_mem_root(&io_mem_ep, &sliced_heap);
13
14   env()->parent()->announce(io_mem_ep.manage(&io_mem_root));
15   sleep_forever();
16   return 0;
17 }
```

Listing 3: Auszug aus 'base–foc/src/test/vadd/server/main.cc': Hauptroutine des VDM

```
1  ...
2
3    template <typename SESSION_TYPE, typename POLICY = Multiple_clients>
4    class Root_component : public Rpc_object<Typed_root<SESSION_TYPE> >,
5                           private POLICY
6    {
7      private:
8
9        Rpc_entrypoint *_ep;
10       Allocator *_md_alloc;
11       ...
12
13     public:
14       ...
15
16       Session_capability session(Root::Session_args const &args)
17       {
18         ...
19       }
20
21       void upgrade(Session_capability session,
22                    Root::Upgrade_args const &args)
23       {
24         ...
25       }
26
27       void close(Session_capability session)
28       {
29         ...
30       }
31   };
32
33 ...
```

Listing 4: Auszug aus 'base/include/root/component.h': VDM-seitige Initialklasse des `IO_MEM` Dienstes

```
1  ...
2
3  class Io_mem_session_component :
4    public Genode::Rpc_object<Genode::Io_mem_session>
5  {
6    private:
7
8      Genode::Rm_connection  _rm;
9      Genode::Cap_connection _cap;
10
11     Genode::Signal_receiver _fault_sreceiver;
12     Genode::Signal_context  _fault_scontext;
13     Io_mem_fault_handler *  _fault_handler;
14
15     Adder32_child *               _adder32_child;
16     Genode::Rom_connection *      _adder32_elf;
17     Genode::Dataspace_capability _adder32_elf_ds;
18
19     Genode::Root_client *     _adder32_root;
20     Adder32_session_client * _adder32_session;
21
22   ...
23 };
24
25
26 Io_mem_session_component::Io_mem_session_component(
27   Genode::addr_t base,
28   Genode::size_t size)
29 :
30   _rm(0, 0x1000)
31 {
32   using namespace Genode;
33
34   try {
35     _adder32_elf    = new (env()->heap())
36                         Rom_connection("adder32");
37     _adder32_elf_ds = _adder32_elf->dataspace();
38   }
39   catch (...) {
40     PERR("Konnte Zugriff auf 'adder32' ROM nicht herstellen.\n"); }
41
42   char name[32];
43   snprintf(name, sizeof(name), "adder_%i", unique_child_id());
44
45   _adder32_child = new (env()->heap())
46                      Adder32_child(name, _adder32_elf_ds,
47                                    ADDER32_RAM_QUOTA, &_cap);
48
49   _adder32_root = new (env()->heap())
50                     Root_client(_adder32_child->root());
51
52   char args[32];
53   snprintf(args, sizeof(args), "ram_quota=%i",
54            ADDER32_SESSION_RAM_QUOTA);
55
56   Adder32_session_capability c;
57   c = static_cap_cast<Adder32_session>(_adder32_root->session(args));
58
59   _adder32_session = new (env()->heap())
60                        Adder32_session_client(c);
61
62   _fault_handler = new (env()->heap())
63                      Io_mem_fault_handler(&_rm, &_fault_sreceiver,
64                                           _adder32_session);
65
66   _rm.fault_handler(_fault_sreceiver.manage(&_fault_scontext));
67   _fault_handler->start();
68   PINF("IO_MEM Session");
```

```
69 }
70
71 ...
```

Listing 5: Auszug aus 'base–foc/src/test/vadd/server/io_mem_session_component.h': VDM-seitige Implementierung der `IO_MEM` Schnittstelle

```
1  ...
2
3  class Io_mem_fault_handler : public Genode::Thread<8*1024>
4  {
5    Genode::Rm_session *        _rm;
6    Genode::Signal_receiver *   _receiver;
7    Adder32_session_client *    _adder32;
8
9    ...
10 };
11
12 void Io_mem_fault_handler::handle_fault()
13 {
14   using namespace Genode;
15   Rm_session::State state = _rm->state();
16
17   ...
18
19   switch(state.type) {
20
21   case Rm_session::WRITE_FAULT:
22     _adder32->write_mmio(state.addr,
23                          Adder32_session::LSB32,
24                          state.value);
25     PINF("Write to Adder32 MMIO 0x%X: 0x%X",
26          (unsigned)state.addr, (unsigned)state.value);
27     break;
28
29   case Rm_session::READ_FAULT: {
30     state.value = (unsigned long)
31                   _adder32->read_mmio(state.addr,
32                                       Adder32_session::LSB32);
33     PINF("Read from Adder32 MMIO 0x%X: 0x%X",
34          (unsigned)state.addr, (unsigned)state.value);
35     break;
36   }
37
38   default:
39     PERR("MMIO Operation not permitted");
40     return;
41   }
42
43   _rm->processed(state);
44 }
45
46
47 void Io_mem_fault_handler::entry()
48 {
49   PINF("IO_MEM fault handler");
50
51   while (1) {
52     Genode::Signal signal = _receiver->wait_for_signal();
53     for (int i = 0; i < signal.num(); i++) handle_fault();
54   }
55 }
56
57 ...
```

Listing 6: Auszug aus 'base–foc/src/test/vadd/server/io_mem_fault_handler.h': Empfänger für Seitenfehler auf dem `IO_MEM` Bereich

```
1  ...
```

```
2
3  class Adder32_child : public Genode::Child_policy,
4                        public Init::Child_policy_enforce_labeling
5  {
6    private:
7
8      struct Resources { ... } _resources;
9
10     const char *              _name;
11     Genode::Rpc_entrypoint    _entrypoint;
12     Genode::Root_capability   _root;
13     Genode::Lock              _ready;
14     Genode::Child             _child;
15     Genode::Service_registry  _parent_services;
16
17   public:
18
19     ...
20
21     Genode::Service * resolve_session_request(const char *service_name,
22                                               const char *args);
23 };
24
25 Adder32_child::Adder32_child(const char *                  name,
26                              Genode::Dataspace_capability  elf_ds,
27                              Genode::size_t                ram_quota,
28                              Genode::Cap_session *         cap_session)
29 :
30   Init::Child_policy_enforce_labeling(name),
31   _resources(name, ram_quota),
32   _name(name),
33   _entrypoint(cap_session, STACK_SIZE, name, false),
34   _ready(Genode::Lock::LOCKED),
35   _child(elf_ds, _resources.ram.cap(), _resources.cpu.cap(),
36          _resources.rm.cap(), &_entrypoint, this)
37 {
38   _entrypoint.activate();
39   _ready.lock();
40   PINF("Adder32 child");
41 }
42
43
44 bool Adder32_child::announce_service(const char *             name,
45                                      Genode::Root_capability  root,
46                                      Genode::Allocator *      alloc)
47 {
48   if (Genode::strcmp(name, "ADDER32")) return false;
49   _root = root;
50   _ready.unlock();
51   PINF("ADDER32 announced");
52   return true;
53 }
54
55 ...
```

Listing 7: Auszug aus 'base–foc/src/test/vadd/server/adder32_child.h': Klasse welche einen Emulator-Kindprozess erzeugt und verwaltet

```
1  ...
2
3  int main(int argc, char **argv)
4  {
5    PINF("Adder32 emulator");
6
7    static Cap_connection cap;
8    static Sliced_heap sliced_heap(env()->ram_session(),
9                                   env()->rm_session());
10   static Rpc_entrypoint adder32_ep(&cap, STACK_SIZE, "adder32_ep");
11   static Adder32_root adder32_root(&adder32_ep, &sliced_heap);
12
13   env()->parent()->announce(adder32_ep.manage(&adder32_root));
14   sleep_forever();
15   return 0;
```

```
16 }
17
18 ...
```

Listing 8: Auszug aus 'base–foc/src/test/vadd/adder32/main.cc': Hauptroutine eines Emulators

```
1  ...
2
3  class Adder32_root :
4    public Genode::Root_component<Adder32_session_component>
5  {
6    private:
7
8      bool _session_exists;
9
10     Adder32_session_component *_create_session(const char *args);
11
12   public:
13
14     class Multiple_sessions_not_allowed : public Genode::Exception { };
15
16     Adder32_root(Genode::Rpc_entrypoint *entrypoint,
17                  Genode::Allocator      *md_alloc);
18 };
19
20 ...
```

Listing 9: Auszug aus 'base–foc/src/test/vadd/adder32/adder32_root.h': Initialklasse des Dienstes `ADDER32`

```
1  ...
2
3  class Adder32_session_component :
4    public Genode::Rpc_object<Adder32_session>
5  {
6    private:
7
8      byte_t _mmio[MMIO_SIZE+sizeof(word_t)];
9
10     word_t * const _addend1;
11     word_t * const _addend2;
12     word_t * const _sum;
13
14     inline void _update_sum();
15
16   public:
17
18     Adder32_session_component();
19
20     ...
21 };
22
23 void Adder32_session_component::write_mmio(Genode::addr_t off,
24                                            Access a, word_t value)
25 {
26   using namespace Genode;
27
28   if(off > sizeof(_mmio)/sizeof(_mmio[0])) return;
29
30   switch(ENDIANNESS) {
31   case BIG_ENDIAN: {
32
33     switch(a) {
34     case LSB32: {
35       uint32_t * dest = (uint32_t *)&_mmio[off];
36       uint32_t * src = ((uint32_t *)&value)
37                        +sizeof(word_t)/sizeof(uint32_t)-1;
38       *dest=*src;
39       break;
40     }
```

```
41      case LSB16: {
42        uint16_t * dest = (uint16_t *)&_mmio[off];
43        uint16_t * src = ((uint16_t *)&value)
44                         +sizeof(word_t)/sizeof(uint16_t)-1;
45        *dest=*src;
46        break;
47      }
48      case LSB8: {
49        uint8_t * dest = (uint8_t *)&_mmio[off];
50        uint8_t * src = ((uint8_t *)&value)
51                        +sizeof(word_t)/sizeof(uint8_t)-1;
52        *dest=*src;
53        break;
54      }
55      default: return;
56      }
57
58      break;
59    } }
60
61    _update_sum();
62  }
63
64  Adder32_session_component::word_t
65  Adder32_session_component::read_mmio(Genode::addr_t off, Access a)
66  {
67    using namespace Genode;
68
69    if(off > sizeof(_mmio)/sizeof(_mmio[0])) return 0;
70
71    switch(ENDIANNESS) {
72    case BIG_ENDIAN: {
73
74      switch(a) {
75      case LSB32: {
76        uint32_t * src = (uint32_t *)&_mmio[off];
77        return (word_t)*src;
78      }
79      case LSB16: {
80        uint16_t * src = (uint16_t *)&_mmio[off];
81        return (word_t)*src;
82      }
83      case LSB8: {
84        uint8_t * src = (uint8_t *)&_mmio[off];
85        return (word_t)*src;
86      } }
87
88      break;
89    } }
90    return 0;
91  }
92
93  void Adder32_session_component::_update_sum()
94  {
95    *_sum=*_addend1+*_addend2;
96  }
97
98  ...
```

Listing 10: Auszug aus 'base–foc/src/test/vadd/adder32/adder32_session_component.h': Implementierung der ADDER32 Schnittstelle am Emulator

```
1  ...
2
3  namespace Arm {
4
5    struct Instruction {
6
7      unsigned long code;
8
9      ...
```

```
10
11      bool writes(unsigned long & reg_id);
12
13      bool reads(unsigned long & reg_id);
14    };
15  }
16
17
18  int Genode::decode_instruction(Genode::addr_t instr,
19                                 unsigned long & writes_value,
20                                 unsigned long thread)
21  {
22    ...
23
24    Arm::Instruction * arm_instr = (Arm::Instruction *)instr;
25    l4_addr_t reg = 0;
26
27    if(!arm_instr->writes(reg))
28      return INSTR_WRITES_NO_VALUE;
29
30    l4_umword_t flags = Fiasco::L4_THREAD_EX_REGS_READ_REG;
31    l4_msgtag_t tag = l4_thread_ex_regs_ret(thread, &reg,
32                                            &writes_value, &flags);
33
34    return SUCCESS;
35  }
36
37
38  int Genode::Pager_object::instruction_processed(unsigned long * instr,
39                                                  unsigned long read)
40  {
41    ...
42
43    Fiasco::l4_cap_idx_t faulter = _badge;
44    Fiasco::l4_addr_t    ip, sp, dst_reg;
45    Fiasco::l4_umword_t  flags;
46    Fiasco::l4_msgtag_t  mtag;
47
48    ip = ~0UL;
49    sp = ~0UL;
50    flags = 0;
51    mtag = Fiasco::l4_thread_ex_regs_ret(faulter, &ip, &sp, &flags);
52    if(l4_msgtag_has_error(mtag)) return ERROR_WHILE_UPDATING_FAULTER;
53
54    Arm::Instruction * instr_arm = (Arm::Instruction *)instr;
55    if(instr_arm->reads(dst_reg)) {
56
57      flags = Fiasco::L4_THREAD_EX_REGS_WRITE_REG;
58      mtag  = Fiasco::l4_thread_ex_regs_ret(faulter, &dst_reg,
59                                            &read, &flags);
60      if(l4_msgtag_has_error(mtag))
61        return ERROR_WHILE_UPDATING_FAULTER;
62
63    }
64
65    ip += sizeof(unsigned long);
66    sp = ~0UL;
67    flags = 0;
68    mtag = Fiasco::l4_thread_ex_regs_ret(faulter, &ip, &sp, &flags);
69    if(l4_msgtag_has_error(mtag)) return ERROR_WHILE_UPDATING_FAULTER;
70
71    return SUCCESS;
72  }
```

Listing 11: Auszug aus 'base–foc/src/core/arm/platform_arm.cc': CPU-spezifische Implementierungen für 'core'

```
1  ...
2
3  int Rm_client::pager(Ipc_pager &pager)
4  {
5    Rm_session::Fault_type pf_type;
6    pf_type = pager.write_pf() ? Rm_session::WRITE_FAULT
```

```
7                                   : Rm_session::READ_FAULT;
8
9     addr_t pf_addr = pager.pf_addr();
10    addr_t pf_ip   = pager.pf_ip();
11
12    if (verbose_page_faults)
13      print_page_fault("page fault", pf_addr, pf_ip, pf_type, badge());
14
15    Rm_session_component             *curr_rm_session;
16    addr_t                            curr_rm_base = 0;
17    Dataspace_component              *src_dataspace = 0;
18    Rm_session_component::Fault_area  src_fault_area;
19    Rm_session_component::Fault_area  dst_fault_area(pf_addr);
20    bool                              lookup;
21    unsigned                          level;
22
23    curr_rm_session = member_rm_session();
24    lookup_attachment(curr_rm_session, curr_rm_base, src_dataspace,
25                      src_fault_area, dst_fault_area, level, lookup);
26
27    if (level == MAX_NESTING_LEVELS) {
28      PWRN("Too many nesting levels of managed dataspaces");
29      return -1;
30    }
31
32    if (!lookup) {
33
34      if (curr_rm_session == member_rm_session())
35        print_page_fault("no RM attachment",
36                         pf_addr, pf_ip, pf_type, badge());
37
38      unsigned long    pf_writes = 0;
39      unsigned long * pf_instr  = 0;
40      if(level > 0 &&
41         (pf_type == Rm_session::WRITE_FAULT ||
42          pf_type == Rm_session::READ_FAULT)) {
43
44        Rm_session_component             *ip_rm = member_rm_session();
45        addr_t                            ip_rm_base = 0;
46        Dataspace_component              *ip_src_dsc = 0;
47        Rm_session_component::Fault_area  ip_src_area, ip_dst_area(pf_ip);
48        bool                              ip_attached;
49        unsigned                          ip_level;
50
51        lookup_attachment(ip_rm, ip_rm_base, ip_src_dsc, ip_src_area,
52                          ip_dst_area, ip_level, ip_attached);
53
54        if(!ip_attached || ip_level >= MAX_NESTING_LEVELS) {
55          PERR("Emulatable memaccess:
56                Faulting instruction not attached");
57
58        } else if(pf_type == Rm_session::WRITE_FAULT) {
59
60          if(decode_instruction(ip_src_area.fault_addr(),
61                                pf_writes, badge()))
62            PERR("Emulatable memaccess: Faulting
63                  instruction of non-emulatable type");
64
65        } else pf_instr = (unsigned long *)ip_src_area.fault_addr();
66      }
67
68      curr_rm_session->fault(this, dst_fault_area.fault_addr()
69                             - curr_rm_base, pf_type, pf_writes,
70                             pf_instr);
71      return 1;
72    }
73
74    ...
75  }
76
77  ...
```

Listing 12: Auszug aus 'base/src/core/rm_session_component.cc': Vorbereitung

der Emulation durch den ersten Empfänger des Seitenfehlers in 'core'

```
[init] Could not open file "ld.lib.so"
[init] parent provides
[init]   service "ROM"
[init]   service "RAM"
[init]   service "CAP"
[init]   service "PD"
[init]   service "RM"
[init]   service "CPU"
[init]   service "IO_MEM"
[init]   service "LOG"
[init]   service "SIGNAL"
[init] child "vdm"
[init]   RAM quota:  10321920
[init]   ELF binary: vadd_server
[init]   priority:   0
[init]   provides service IO_MEM
[init] child "user_1"
[init]   RAM quota:  2981888
[init]   ELF binary: vadd_client
[init]   priority:   0
[init] child "user_2"
[init]   RAM quota:  2981888
[init]   ELF binary: vadd_client
[init]   priority:   0
[init -> user_1] IO_MEM client to 0x71000000
[init -> vdm] Virtual device monitor
[init -> user_2] IO_MEM client to 0x71000000
[init -> vdm] IO_MEM root
[init] child "vdm" announces service "IO_MEM"
[init -> vdm -> adder_1] Adder32 emulator
[init -> vdm -> adder_1] ADDER32 root
[init -> vdm] Adder32 child
[init -> vdm -> adder_1] ADDER32 session
[init -> vdm] ADDER32 announced
[init -> vdm] IO_MEM Session
[init -> vdm] IO_MEM fault handler
[init -> vdm -> adder_2] Adder32 emulator
[init -> vdm -> adder_2] ADDER32 root
[init -> vdm] Adder32 child
[init -> vdm -> adder_2] ADDER32 session
[init -> vdm] ADDER32 announced
[init -> vdm] IO_MEM Session
[init -> vdm] IO_MEM dataspace
[init -> user_2] Adder32 driver at 0x1000
[init -> vdm] IO_MEM dataspace
[init -> vdm] IO_MEM fault handler
[init -> vdm] Write to Adder32 MMIO 0x0: 0x2
[init -> user_1] Adder32 driver at 0x1000
[init -> vdm] Write to Adder32 MMIO 0x0: 0x2
[init -> vdm] Write to Adder32 MMIO 0x4: 0x1
[init -> vdm] Write to Adder32 MMIO 0x4: 0x1
[init -> vdm] Read from Adder32 MMIO 0x8: 0x3
[init -> user_2] Adder32 sum: 1 + 2 = 3
[init -> vdm] Read from Adder32 MMIO 0x8: 0x3
[init -> vdm] Write to Adder32 MMIO 0x0: 0x3
[init -> user_1] Adder32 sum: 1 + 2 = 3
[init -> vdm] Write to Adder32 MMIO 0x4: 0x3
[init -> vdm] Write to Adder32 MMIO 0x0: 0x3
[init -> vdm] Write to Adder32 MMIO 0x4: 0x3
[init -> vdm] Read from Adder32 MMIO 0x8: 0x6
[init -> user_2] Adder32 sum: 3 + 3 = 6
[init -> vdm] Read from Adder32 MMIO 0x8: 0x6
[init -> vdm] Write to Adder32 MMIO 0x0: 0x4
[init -> user_1] Adder32 sum: 3 + 3 = 6
[init -> vdm] Write to Adder32 MMIO 0x4: 0x6
[init -> vdm] Write to Adder32 MMIO 0x0: 0x4
[init -> vdm] Read from Adder32 MMIO 0x8: 0xa
[init -> vdm] Write to Adder32 MMIO 0x4: 0x6
[init -> user_2] Adder32 sum: 6 + 4 = 10
[init -> vdm] Read from Adder32 MMIO 0x8: 0xa
[init -> vdm] Write to Adder32 MMIO 0x0: 0x5
```

```
72 [init -> user_1] Adder32 sum: 6 + 4 = 10
73 [init -> vdm] Write to Adder32 MMIO 0x4: 0xa
74 [init -> vdm] Write to Adder32 MMIO 0x0: 0x5
75 [init -> vdm] Read from Adder32 MMIO 0x8: 0xf
76 [init -> vdm] Write to Adder32 MMIO 0x4: 0xa
77 [init -> user_2] Adder32 sum: 10 + 5 = 15
78 [init -> vdm] Read from Adder32 MMIO 0x8: 0xf
79 [init -> user_1] Adder32 sum: 10 + 5 = 15
```

Listing 13: Auszug aus 'vadd.log': Ausgabe des Testszenario auf der seriellen Schnittstelle

```
1  ...
2
3  namespace Genode {
4
5    struct Rm_session : Session
6    {
7      enum Fault_type {
8        READY = 0, READ_FAULT = 1,
9        WRITE_FAULT = 2, EXEC_FAULT = 3 };
10
11     enum Access_format { LSB8, LSB16, LSB32 };
12
13     struct State
14     {
15       Fault_type type;         // Typ des fehlererzeugenden Zugriffs
16       Access_format format;    // Bitformat des Zugriffs
17       addr_t addr;             // Adresse auf die zugegriffen wurde
18       unsigned long value;     // Schreibzugriff: Zu schreibender Wert
19       unsigned long * instr;   // Lesezugriff: Zeiger auf Befehl in Core
20
21       bool operator== (State & s)
22       {
23         return type==s.type && format==s.format && addr==s.addr;
24       }
25
26       ...
27     };
28
29     virtual Local_addr attach(Dataspace_capability ds,
30                               size_t size = 0, off_t offset = 0,
31                               bool use_local_addr = false,
32                               Local_addr local_addr = 0) = 0;
33
34     virtual void processed(State state) = 0;
35
36     virtual void fault_handler(Signal_context_capability handler) = 0;
37
38     virtual State state() = 0;
39
40     virtual Dataspace_capability dataspace() = 0;
41
42     ...
43   };
44 }
```

Listing 14: Auszug aus 'base/include/rm_session/rm_session.h': Die Schnittstelle zu dem Dienst `RM`

Literaturverzeichnis

[1] David Jones, *A Time-Multiplexed FPGA Architecture For Logic Emulation*, 1995.

[2] Christian Helmuth Norman Feske, *Design of the Bastei OS Architecture* (2010), available at `http://os.inf.tu-dresden.de/papers_ps/bastei_design.pdf`.

[3] Xilinx Inc., *MicroBlaze Processor Reference Guide, Embedded Development Kit EDK 10.1i*, 2008.

[4] Inc. Sun Microsystems, *RPC: Remote Procedure Call Protocol Specification Version 2*, 1988.

[5] Sun Microsystems R. Thurlow, *RPC: Remote Procedure Call Protocol Specification Version 2*, 2009.

[6] Bruce Jay Nelson Andrew D. Birrell Xerox Palo Alto Research Center, *Implementing Remote Procedure Calls*, 1983.

[7] Stephen Schmitt, *Integrierte Simulation und Emulation eingebetteter Hardware/Software-Systeme*, 2005.

[8] Fabrice Bellard, *QEMU a Fast and Portable Dynamic Translator*, 2005.

[9] Stephen Trimberger, *Field Programmable Gate Array Technology*, 1994.

[10] Pong P. Chu, *FPGA Prototyping by Verilog Examples: Xilinx Spartan-3 Version*, 2008.

[11] ———, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*, 2008.

[12] Ian Kuon and Jonathan Rose, *Measuring the Gap Between FPGAs and ASICs*, IEEE transactions on computer-aided design of integrated circuits and systems, Vol. 26, No. 2 (February 2007).

[13] Edward Tau, Derrick Chen, Ian Eslick, and Jeremy Brown, *A First Generation DPGA Implementation*, Third Canadian Workshop of Field-Programmable Devices (Montreal, Canada, May 1995).

[14] André DeHong, *DPGA Utilization and Application*, ACM/SIGDA Fourth International Symposium on FPGAs (Monterey, CA, February 1996).

[15] Tristan Gingold, *GHDL User Guide* (2010), available at `http://ghdl.free.fr/site/uploads/Main/ghdl_user_guide/AA_ghdl_guide.html`.

[16] Narasimha B. Bhat, Kamal Chaudhary, and Ernest S. Kuh, *Performance-Oriented Fully Routable Dynamic Architecture For A Field Programmable Logic Device* (1993).

[17] Achronix Semiconductor Corp., *Speedster FPGA Family Product Brief* (2009), available at `www.achronix.com/docs/Speedster_Product_Brief_PB001.pdf`.

[18] Inc. Xilinx, *Early Access Partial Reconfiguration User Guide* (2008).

[19] Armin Jeyrani Mamegani, *Erprobung und Evaluierung von Methoden zur partiellen und dynamischen Rekonfiguration eines SoC-FPGAs*, 2010.

[20] Genode Labs GmbH, *Release notes for the Genode OS Framework 9.11* (2009), available at `http://genode.org/documentation/release-notes/9.11`.

[21] ———, *Release notes for the Genode OS Framework 8.11* (2008), available at `http://genode.org/documentation/release-notes/8.11`.

[22] ———, *Genode base API documentation* (2009), available at `http://genode.org/documentation/api/base_index`.

[23] ———, *Release notes for the Genode OS Framework 11.05* (2011), available at `http://genode.org/documentation/release-notes/11.05`.

[24] Inc. Xilix, *Spartan-3A/3AN FPGA Starter Kit, Board User Guide* (2008), available at `http://www.xilinx.com/support/documentation/boards_and_kits/ug334.pdf`.

[25] ———, *LogiCORE IP XPS Timer/Counter, Product Specification* (2010), available at `http://www.xilinx.com/support/documentation/ip_documentation/xps_timer.pdf`.

[26] ———, *LogiCORE IP XPS Interrupt Controller, Product Specification* (2010), available at `http://www.xilinx.com/support/documentation/ip_documentation/xps_intc.pdf`.

[27] ———, *LogiCORE IP Multi-Port Memory Controller, Product Specification* (2011), available at `http://www.xilinx.com/support/documentation/ip_documentation/mpmc/v6_04_a/mpmc.pdf`.

[28] ———, *LogiCORE IP XPS UART Lite, Product Specification* (2011), available at `http://www.xilinx.com/support/documentation/ip_documentation/xps_uartlite/v1_02_a/xps_uartlite.pdf`.

[29] Betriebssysteme Lehrstuhl der TU Dresden, *Fiasco OC Microkernel, Features* (2011), available at `http://os.inf.tu-dresden.de/fiasco/features.html`.

[30] International Business Machines Corporation, *128-Bit Processor Local Bus, Architecture Specifications* (2007), available at `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4/$file/PlbBus_as_01_pub.pdf`.