FAKULTÄT FÜR INFORMATIK UND AUTOMATISIERUNG
INSTITUT FÜR TECHNISCHE INFORMATIK UND INGENIEURINFORMATIK
FACHGEBIET SYSTEM- UND SOFTWARE-ENGINEERING

Diplomarbeit

Ausbau einer Umgebung für das Codesign von Hardware und Software

Verfasser	Martin Stein Rothenburger Straße 37 01099 Dresden
Matrikelnummer	39665
Studiengang, Matrikel	Informatik, 2005
Verantw. Professor	Prof. Dr.-Ing. habil. Armin Zimmermann
Hochschulbetreuer	Prof. Dr. Volker Zerbe – FH Erfurt
Betrieblicher Betreuer	Dr.-Ing. Norman Feske – Genode Labs GmbH
Zeitraum	Oktober 2012 – März 2013

Abstract

Modern Hardware-Software Codesign lives from the capabilities of holistic simulation frameworks and specialized FPGA-platforms. The latter are struggling with a low yield per unit area, high acquisition costs and the problem of realizing application-level security in hardware. Holistic simulation frameworks, on the other hand, suffer from the balancing act between accuracy, speed and the catching-up with chip designers. Therefore, this work introduces a microkernel-based approach that enables the seamless integration of HDL simulations on top of the real target system, making them available to software tests. Special hardware is not required in this context. Simulators communicating among each other as well as with hardware and software of the target system, by using the natural interfaces. Apart from that, simulators, devices and applications are isolated against each other to stem failure propagtion. Furthermore, the system provides an approach to observe and analyse any communication a simulator is involved in. The foundation of these features is a minimal-invasive extension beyond the operating system that may be used flexibly. Conceptually, this work is an advancement of the modular emulation framework VDM [4].

Abstract

Modernes Codesign von Hardware und Software lebt von den Fähigkeiten ganzheitlicher Simulationsumgebungen und spezialisierter FPGA-Plattformen. Letztere kämpfen mit einer niedrigen Ergiebigkeit, hohen Beschaffungskosten und dem Problem, Anwendungssicherheit in Hardware realisieren zu müssen. Auf der anderen Seite stehen ganzheitliche Simulationsumgebungen im ständigen Spagat zwischen Genauigkeit, Geschwindigkeit und der Aufholjagd zu Chipdesignern. Diese Arbeit stellt deshalb einen mikrokernelbasierten Ansatz vor, der es ermöglicht, Software-Tests im Zielsystem aufwandsarm mit HDL-Simulationen anzureichern. Spezielle Hardware wird dazu nicht benötigt. Die Simulationen können auf natürliche Weise untereinander, sowie mit Geräten und Anwendungen des Zielsystems kommunizieren. Darüber hinaus bleiben Simulationen, Geräte und Anwendungen gegeneinander abgeschottet, so dass Fehlerfortpflanzung vermieden wird. Außerdem stellt das System Mittel für die Beobachtung und Auswertung jedweder Kommunikation bereit, in die ein Simulator involviert ist. Der Unterbau für die Integration ist eine minimal-invasive Erweiterung oberhalb des Betriebssystems, die sehr flexibel eingesetzt werden kann. Konzeptionell ist diese Arbeit eine Weiterentwicklung des modularen Emulations-Frameworks VDM [4].

Inhaltsverzeichnis

Einführung

Modelle

Implementierung

Analyse

Schluss

Anhang

1 Einführung

Das Problem des Codesigns von Hardware und Software ist seit nunmehr zwanzig Jahren Gegenstand der Forschung. Ende der 80er Jahre hatten eingebettete Systeme eine Komplexität erreicht, welche die vorherrschende Separation von mikroprozessorbasiertem Design und IC-Design in Frage stellte. Mikroprozessoren vereinfachten die Entwicklung eingebetteter Systeme. Sie machten es möglich, komplexe Teilprobleme durch wiederverwendbare Hardware auf eine höhere Abstraktionsebene zu verlagern. Außerdem ließ sich der Trade-Off zwischen Kosten und Performanz durch die Partitionierung auf Hardware und Software besser beeinflussen. Damit kommen jedoch die grundlegenden Probleme des Codesigns auf. Die Möglichkeit, die Lösung eines Problems auf zwei Welten ganz unterschiedlicher Natur aufteilen zu können, stellt Entwickler vor neue Entscheidungen, deren Auswirkungen nicht ohne weiteres absehbar sind. Zu diesen Entscheidungen gehört nicht nur die Aufteilung der eigentlichen Funktionalität, sondern auch das Schnittstellendesign zwischen den beiden Welten. Hinzu kommt, dass Hardware wegen ihrer hochparallelen Arbeitsweise gemeinhin als Datenfluss beschrieben wird, während Software für die meisten Mikroprozessoren einen seriellen Kontrollfluss darstellt. Die Systembeschreibung eines Codesigns ist also nicht zwangsläufig homogen. Herkömmliche Analysewerkzeuge wie SPICE [37] oder verschiedene Instruction-Set-Simulatoren sind jedoch auf einen homogenen Lösungsansatz spezialisiert. Damit kommt das Problem auf, dass die Systembeschreibung während der Entwicklung nur noch partiell analysiert werden kann. Fehlentscheidungen bei der Partitionierung und partitionsübergreifende Funktionsfehler werden so erst nach der Fertigung sichtbar. Das macht ihre Behebung mitunter sehr kostspielig. Anfang der 90er Jahre kam deshalb eine verstärktes Interesse dafür auf, wie sich Codesign durch digitale Technik unterstützen lässt. [20]

1.1 Verwandte Arbeiten

Frühe Codesign-Arbeiten wie COSYMA [20 S. 1f] oder VULCAN [22] konzentrierten sich noch ausschließlich auf das Partitionierungsproblem. Deshalb setzen sie bei einer homogenen Beschreibung des Systems an. Um die automatisierte Partitionierung zu ermöglichen, kommen für diese Beschreibung aber spezielle,

C-ähnliche Sprachen zum Einsatz. Sie sind so beschaffen, dass Teilfunktionen sowohl in einen Schaltungsentwurf, als auch ein C-Konstrukt überführt werden können. Die entstandenen Partitionen werden dann getrennt analysiert, um aus den Ergebnissen Werte für das gesamte System zu schätzen. Dadurch lassen sich iterativ Funktionen umsiedeln, bis das System den gewünschten Trade-Off aus Performanz und Produktionskosten erfüllt. Als Ausgangspunkt dieser Prozedur wird einfach eine rein software- oder hardware-basierte Lösung gewählt. Diese Cosynthese ermöglicht zwar komplexe Partitionierungskriterien, die auf den Ergebnissen der Einzelanalyse basieren, ohne eine Möglichkeit der Coverifikation jedoch bleibt der Ansatz stets auf Heuristiken angewiesen.

Ein bis heute grundlegender Ansatz zur Coverifikation eingebetteter Systeme, wurde 1992 durch PTOLEMY [21] aufgezeigt. Die Idee besteht darin, die unterschiedlichen Beschreibungsmodelle eines heterogenen Systems durch eine Software-Umgebung zusammenzuführen. Dazu benötigt jedes Beschreibungsmodell ein abstraktes Gerüst, das gemeinhin Domäne genannt wird. Die Domäne übersetzt externe Effekte des Beschreibungsmodells auf ein abstraktes Kommunikationsmodell. Durch diese künstlich angelegten und untereinander kompatiblen Kanäle, kann zwischen den Domänen modellübergreifende Kommunikation betrieben werden. Innerhalb einer Domäne können Entwurf und Analyse aber mit den gewohnten Mitteln vonstatten gehen. Dadurch wird die Cosimulation heterogener Systeme möglich. Spätere Arbeiten, wie FORMAGGIO [25] spezialisierten den Ansatz auf typische Daten- und Kontrollflussdomänen, um die Komplexität des Frameworks für eingebettete Systeme zu reduzieren. Heute finden sich diese ersten software-basierten Ansätze zur Cosynthese und zur Cosimulation vereint, in umfrangreichen Codesign-Frameworks wie PEACE [24] und SEAMLESS CVE [23]. Sie begleiten die Entwickler heterogener Systeme von der homogenen Beschreibung des Systems bis zur Schaltungssynthese und Kompilierung. Zu dieser Kategorie können auch Erweiterungen gängiger Virtual Machines wie QEMU gezählt werden, die, wie bei YEH [27] die Brücke zwischen HDL-Design und virtuellem Systembus schlagen.

Einen anderen, rein software-basierten Weg zur Coverifikation greift LISA+, eine Weiterentwicklung der ISS-Beschreibungssprache LISA [26] auf. Die grundlegende Idee besteht darin, das Prozessormodell mit den selben Mitteln zu beschreiben wie die individuelle Peripherie. Die Simulationen werden gewissermaßen auf natürliche Weise zusammengeführt, da der Software-Simulator nun Teil der Hardware-Beschreibung ist. Gewöhnliche HDLs wie VERILOG sind für einen performanten ISS, wie er bei dem Ansatz gebraucht wird, zu generisch gehalten. Rein ISS-orientierte Sprachen hingegen, wie der Vorgänger von LISA+, vernachlässigen das allgemeine Gerätedesign. Deshalb besteht die Herausforderung dieser Coverifikation darin, den Umfang gängiger HDLs mit der Optimierung oft

genutzter Geräte zu vereinen. Dieses Ziel kann aber auch anders, als durch einen konzeptionellen Umbau der Sprache erreicht werden. SÉMÉRIA [30] steigerte die Performanz eines SYSTEMC-Cosimulators durch Optimierung von Design- und Simulationsablauf. Zum Beispiel ist der Simulator, den diese Arbeit beschreibt, modular aufgebaut, so dass Komponenten geringerer Abstraktion ausgelassen werden können, wenn sie für den Abstraktionsgrad der Systembeschreibung noch nicht erforderlich sind. Diese Konzepte haben sich bis heute aber offenbar kaum durchgesetzt. Ein Grund dafür könnte die geringe Verfügbarkeit von HDL-Beschreibungen für kommerzielle Mikroprozessoren sein.

Seit den späten 90er Jahren bildet sich noch ein weiterer Zweig der Forschung heraus. Er basiert auf der Idee, alle Aspekte des Codesigns in einem Beschreibungsmodell umzusetzen. Ein Beispiel dafür ist die Sprache NAPA C [36]. Anders als bei der automatischen Partitionierung aus einer homogenen Systembeschreibung, erlauben solche Sprachen die manuelle Partitionierung und das Design der Hardware-Software-Schnittstelle direkt im Quellcode.

Die bislang vorgestellten Lösungen haben allesamt mit einem grundlegendem Problem rein software-basierter Coverifikation zu kämpfen: Die Komplexität einer komplett künstlichen Umgebung nimmt schnell ein kritisches Maß an. Die Entwickler solcher Systeme stehen vor der Aufgabe, eine wachsende Zahl immer umfangreicherer Architekturen performant nachahmen zu müssen. Es liegt nahe, dass Abweichungen gegenüber dem realen Vorbild schnell herhalten, um den zeitlichen und technischen Einschränkungen gerecht zu werden. So werden zum Beispiel optionale Komponenten wie Caches oder Koprozessoren bei der Simulation oft vernachlässigt. Während diese Ungenauigkeiten bei der Verifikation von Hardware bislang unvermeidbar sind, hat sich die Lage bei eingebetteter Software über die Jahre verändert. Eingebettete Mikroprozessoren stießen anfangs noch schnell an die Ressourcenschranken ihres Umfelds. Ein hoher Grad an Wiederverwendbarkeit bei den Chips war so nur schwer zu erreichen und Software-Entwickler waren zu sehr sparsamen Lösungen angehalten. Doch mit dem Fortschritt der Fertigungstechnik und der RISC-Prozessoren während der 90er Jahre, hielten auch langlebigere Architekturen, wie die POWERPC-400-, oder die ARM-Familie Einzug in den Embedded-Bereich. Diese zeichnen sich unter anderem durch stabile Standards und einen großen Verbreitungsgrad aus, so dass bei der Entwicklung eingebetteter Systeme heute oft schon ein Referenzboard der Ziel-CPU existiert. Es liegt also nahe, die Software während der Entwicklung nicht in einem künstlichen, sondern einem realen Umfeld zu analysieren. Das Problem der Coverifikation lässt sich dadurch auf die Virtualisierung und Bereitstellung von Gerätedesigns reduziert.

Einen bedeutenden Ansatz nach diesem Prinzip, stellen hybride Systeme aus Hardware-Emulator und Mikroprozessor dar. Sie gehen auf die frühen 80er Jahre zurück, als erstmals die Kombination von Mikroprozessoren und FPGAs diskutiert wurde [38, S. 27]. Erst Anfang des neuen Jahrtausends hatte sich die Technik soweit entwickelt, dass industrielle FPGAs, wie der VIRTEX II PRO komplexe Prozessoren wie den POWERPC 405 integrierten [28]. Doch schon vorher existierten solche Rapid-Prototyping-Plattformen in der Forschung. Das Modell der Coverifikation ist bei all diesen Systemen ähnlich. Die Software kommuniziert auf natürliche Weise mit der Hardware, welche von FPGAs emuliert wird. Dazu ist der FPGA mit den nativen Kommunikationsmitteln, wie dem Systembus und dem Interrupt-Controller verbunden. Diese Form der Coverifikation wird heute auch mit software-basierten Lösungen kombiniert. Dazu bedarf es, wie bei SLOMKA [29], einer Hardware-Synthese die mit unterschiedlichen Architekturmodellen auskommt. Ist die Entwicklung abgeschlossen, wird das FPGA-Modell einfach durch das Modell der Zielarchitektur ersetzt. Ein weiterer Vorteil der Rapid-Prototyping-Plattformen ist, dass sie den Komponenten, die mittels FPGA emuliert werden, andere, bereits existierende Geräte zur Verfügung stellen können. Somit müssen nur noch solche Komponenten emuliert werden, die tatsächlich der Entwicklung unterliegen. Mit BORPH [33] existiert mittlerweile sogar ein LINUX-basiertes Betriebssystem das diesen Ansatz integriert. BORPH kann Gerätedesigns auf einem rekonfigurierbaren FPGA starten, so als wären sie normale LINUX-Prozesse. Die Kommunikation mit dieser Gateware wird – wie in LINUX üblich – durch Dateien umgesetzt.

Ein Nachteil der FPGAs ist jedoch der enorme Flächenverbrauch. Designs beanspruchen, auf einem FPGA emuliert um ein Vielfaches mehr an Logik, als wenn sie durch einen ASIC umgesetzt würden. Deshalb geht ihr Einsatz heute noch mit Einschränkungen einher. Das wird auch dadurch verdeutlicht, dass aktuelle Arbeiten, wie bei SCHELLE [34] zunehmend auf verteilte, und somit langsamere FPGAs zurückgreifen müssen. Im Gegenzug wurde aber auch die Forschung im Bereich verteilter FPGAs vorangetrieben. Mittlerweile finden deshalb, wie bei KULMALA [35], ganze SoCs ihren Platz auf den FPGA-Stacks. Das ist vor allem dann von Bedeutung, wenn für den Zielprozessor keine Rapid-Prototyping-Plattform vorhanden ist, weil er z.B. selbst noch in der Entwicklung steht, oder nicht genug Verbreitung findet. Die Software wird in diesem Fall wie gewohnt von einem externen Speichermedium geladen. Diese Variante bringt jedoch erneut Abweichungen des Prozessormodells mit sich und leidet unter dem Overhead der FPGA-übergreifenden Kommunikation.

Ein anderes Problem, das der Einsatz von FPGAs generell aufwirft, liegt bei der Sicherheit des restlichen Systems. Designs die sich noch in der Entwicklung befinden erlangen über den Systembus mitunter die Möglichkeit, die Integrität

des umliegenden Systems zu stören. Ohne eine IOMMU zum Beispiel, die derzeit eher selten zur Verfügung steht, ist es software-seitig nicht möglich solchen Szenarien zuvorzukommen. Fehlerauswirkungen können also nicht eingegrenzt werden, wodurch das Auffinden des verursachenden Designfehlers erschwert wird. BORPH geht dieses Problem an, indem jeder Geräteentwurf hinter ein restriktives Frontend aus Gateware gesperrt wird. Bisher ist BORPH aber auf rekonfigurierbare FPGAs angewiesen und wahrscheinlich auch deshalb nur für wenige Plattformen verfügbar [33].

Dennoch stellt BORPH einen Wegweiser für die Weiterentwicklung von Codesign-Umgebungen dar, da es das Problem der Isolation angeht, indem emulierte Komponenten in das Sicherheitskonzept des Betriebssystems integriert werden. Das Sicherheitsmodell des monolithischen LINUX-Kernels ist jedoch überholt: moderne Mikrokernel, wie GENODE oder die L4-Kernel abstrahieren nicht nur von niederen Aspekten wie der Speicherverwaltung, sie stellen dem Entwickler auch ein feingranulares System der Least-Privilege-Rechtevergabe zur Verfügung [1][31][32, S. 389]. Dazu wird jeder Prozess[1] in eine virtuelle Umgebung eingesperrt, welche lediglich die Möglichkeiten bietet, die ihr der Mikrokernel zugesteht. Diese in das Systemkonzept integrierte Virtualisierung bietet sich an, eine Schnittstelle zwischen Software-Test und Hardware-Simulation zu schaffen, die kontrollierbarer und flexibler ist. So kommt es, dass gerade bei den Fragen der Sicherheit und der Hardware-Anforderungen ungenutztes Potential für die Coverifikation besteht.

Bezüglich der Cosynthese gibt es gegenläufige Argumentationen. Die automatisierte Partitionierung mag dazu dienen, den individuellen Trade-Off zwischen Performanz und Produktkosten möglichst schnell und genau zu erreichen. Für den Erfolg muss das System aber feingranular aufteilbar sein. Dieser Granularitätsgrad ist auch von der Beschreibung des Systems abhängig. Ist er zu fein wird der Entwickler bei der manuellen Optimierung zunehmend eingeschränkt. Ist er zu grob steigt das Risiko, dass eine manuelle Partitionierung effizienter wäre. Aus diesem Grund sollte ein allgemeines Modell zur Coverifikation die Automatisierung zwar ermöglichen, aber nicht vorraussetzen.

1 Als Prozess wird eine Software-Komponente mit homogenem Vertrauensprofil bezeichnet. D.h., eine Zusammenschluss von Funktionalitäten, die sich gegenseitig vollständig vertrauen können bzw. müssen.

1.2 Zielstellung

Die genannten Umstände motivierten dazu, über eine sicherere, und zugleich flexiblere Bereitstellung von Geräteentwürfen nachzudenken. Es kam deshalb die Idee auf, Software-Tests im realen Zielsystem mit software-basierten Geräte-Simulationen anzureichern. Diese Arbeit beschäftigt sich mit der Umsetzung dieser Idee und orientiert sich dabei an folgenden Thesen:

1.2.1 Codesign im Zielsystem

Rapid-Prototype-Testing von Software und software-basierte Gerätesimulation lassen sich durch einen modernen Mikrokernel transparent vereinen. Die Entwürfe kommunizieren beiderseits auf natürliche Art und Weise mit der jeweils anderen Welt, zum Beispiel hardware-seitig durch Bustransaktionen und software-seitig durch MMIO-Zugriffe. Es ist keine Hardware speziell für die Simulation nötig. Das Testboard muss also, abgesehen von den Geräten, die in Entwicklung sind, nur die Anforderungen erfüllen, die auch das Zielboard erfüllt. Geräte die bereits real existieren lassen sich von den simulierten Geräten nativ nutzen. Das heißt, die Simulation von Konzepten wie DMA lässt sich natürlich integrieren. Außerdem müssen nur die Geräte simuliert werden, die sich tatsächlich noch in der Entwicklung befinden.

1.2.2 Bedienbarkeit

Die Schnittstelle zwischen der Verifikation und dem Entwickler lässt sich in drei eigenständige Aspekte unterteilen. Die Synthese der Gerätesimulationen kann vollständig automatisiert werden. Als Eingabe genügt die Datenflussbeschreibung des Designs, welche auch bei der Fertigung genutzt wird. Die Synthese der Software hingegen, lässt sich auf die selbe Art und Weise durchführen wie bei der abschließenden Inbetriebnahme. Sie ist also nicht vom Verifikationsmodell abhängig, das in dieser Arbeit vorgestellt werden soll, sondern hauptsächlich von der nativen Toolchain und dem verwendeten Mikrokernel. Der dritte Aspekt ist die Synthese der Schnittstelle zwischen Software und simulierter Hardware. Sie ist abhängig von den Kommunikationsmodellen, welche zum Einsatz kommen sollen. Solche Modelle, wie die Kommunikation über einen bestimmten Speicherbus oder Interrupt-Controller folgen einer Spezifikation, die dem Entwickler gemeinhin Freiheitsgrade lässt. Dazu gehören etwa die Bandbreite oder die Nummern der Interrupts. Die Steuerung der Synthese solcher Modelle wirft zuerst zwei grundlegende Fragen auf. Einerseits, welche Teile der Geräteschnittstelle ein Modell der übergreifenden Kommunikation bilden und andererseits, welche Teile der Software-Umgebung auf ein solches Modell abbilden. Darüber hinaus lässt sich der Konfigurationsaufwand beiderseits auf die Freiheitsgrade

des Kommunikationsmodells beschränken. Als Beispiel sei hier die Kommunikation über einen MMIO-Bereich aufgeführt. Laut der These braucht der Entwickler hardware-seitig nur anzugeben, welche Signale des Geräts die Busanbindung darstellen und – je nach Bus – welche Bandbreite, Timeouts etc. zum Einsatz kommen. Software-seitig hingegen muss er festlegen, welche Speicheradressen mit MMIO-Bereichen des Geräts assoziiert werden sollen und unter Umständen welche Zugriffe ihm erlaubt sind.

Dadurch, dass alle drei Aspekte eigenständig behandelt werden und sowohl die Hardware- als auch die Software-Synthese auf der nativen Beschreibung beruhen, lassen sich Entwurf und Verifikation enger zueinander führen. Die Übergänge von der Komponentenanalyse zur Systemanalyse und von der Systemanalyse zur Fertigung werden vereinfacht.

1.2.3 Exklusivität

Durch den Einsatz eines modernen Mikrokernels, lassen sich für nebenläufige Subsysteme der Software verschiedene Zustände eines simulierten Geräts verwalten. Durch diese Simulationskontexte lassen sich zugleich verschiedene Software-Entwürfe auf dem selben Hardware-Entwurf testen, ohne dass eine gegenseitige Beeinflussung möglich ist. Außerdem läßt sich die Ursache eines Fehlers somit besser eingrenzen, da sich den Nutzerkreis eines Kontextes künstlich ausdünnen lässt. Die Verwaltung der Simulationskontexte ist für die betroffenen Prozesse transparent.

1.2.4 Kosten-Accounting

Die Gerätesimulation wirft Kosten auf. Nicht nur in Form des zusätzlichen Ressourcenverbrauchs, sondern unter Umständen auch in Form von zusätzlichem Vertrauen. Diese Kosten lassen sich in drei Klassen unterteilen. Zum einen gibt es die Kosten, die dadurch entstehen, dass die Simulation bestimmter Geräte überhaupt möglich ist. Sie lassen sich auf die Prozesse einschränken, die zumindest von einem der Geräte abhängig sind. Das bedeutet, dass sich parallel zu der Verifikation z.B. Analysesoftware betreiben lässt, die vollkommen unbeeinflusst von der Verifikation bleibt. Es bedeutet aber auch, dass verschiedene Subsysteme der Software zugleich unterschiedliche Gerätedesigns testen können, da sich der Geltungsbereich einer Simulation einschränken lassen muss. Desweiteren gibt es die Kosten, die durch die Existenz eines Simulationskontextes entstehen. Sie lassen sich auf die Prozesse einschränken die den Kontext benötigen. Die übrigen Kosten sind die der Simulation selbst. Sie lassen sich auf das Maß einschränken, welches die Spezifikation der Geräteschnittstelle zulässt. Das heißt unter ande-

rem, dass sie nur dann aufkommen, wenn die Schnittstelle wirklich benutzt wird. Außerdem werden so die Auswirkungen eines Designfehlers, auf Software oder andere Geräte, im Sinne der Geräteschnittstelle beschränkt.

1.2.5 Beobachtbarkeit

Zu jedem Zeitpunkt zu dem Informationen zwischen der Software und einem simuliertem Gerät ausgetauscht werden, kann der Austausch von außen über eine einfache Schnittstelle protokolliert und je nach Art auch modifiziert werden. Austausch und Teilnehmer lassen sich dafür beliebig lange anhalten. Abschließend lässt sich der Austausch mit den neuen Modalitäten beenden, woraufhin auch die Teilnehmer fortfahren. Das heißt, dass sich gerade domänenübergreifende Fehler, die erst durch die Kooperation von Software und Hardware aufkommen, gut verfolgen lassen.

1.2.6 Skalierbarkeit

Der zusätzliche Ressourcenverbrauch, der durch die Kommunikation mit einer Simulation entsteht, skaliert über der Anzahl vergleichbarer, nebenläufiger Kommunikationsvorgänge. Das heißt, langfristig gesehen wächst der Ressourcenverbrauch proportional, wenn die Zahl der parallel genutzten Geräte zunimmt. Es heißt aber auch, dass der Ressourcenverbrauch höchstens proportional wächst, wenn die Anzahl der parallelen Zugriffe auf ein einziges simuliertes Gerät zunimmt.

1.3 Struktur der Arbeit

Das zweite Kapitel wird die theoretischen Modelle vorstellen, aus denen sich die Codesign-Umgebung zusammensetzt, und die Zusammenhänge zwischen ihnen erläutern. Kapitel 3 zeigt, wie die Modelle auf dem GENODE OS-Framework umgesetzt wurden. Im vierten Kapitel wird das entstandene System an realen Beispielszenarien getestet. Neben der Demonstration der Handhabung, liegt das Augenmerk auch auf der Performanz und Stabilität der Umgebung. Anhand der Ergebnisse dieses Kapitels sollen im fünften Kapitel die Thesen überprüft werden, die der Arbeit als Zielstellung dienen. Abschließend soll resümiert werden, welche Probleme und Möglichkeiten diese Arbeit für die Zukunft offen lässt.

2 Modelle

Die Studie "*Modell zur Transparenz der Verfügbarkeit von Hardware-Ressourcen auf der Systembusebene"* [4] geht dieser Arbeit thematisch voraus. Sie beschreibt die Grundlagen der Emulation peripherer Geräte beim Einsatz moderner Mikrokernel wie GENODE. Das Ergebnis dieser Studie ist VDM, ein Monitor für virtuelle Geräte.

2.1 Vorraussetzungen

Für die Erweiterung zu einem Codesign-Framework, werden an die Zielplattform ähnliche Vorraussetzungen gestellt wie bei VDM. Das Abfangen von MMIO-Zugriffen soll durch die Hardware-Virtualisierung des Speichers, kurz MMU möglich sein. Bei einem Zugriff, der nicht hinterlegt ist, wird der Auslöser gestoppt und dem Betriebssystem eine Fehlermeldung zugestellt. Das Betriebssystem kann dann das Codesign-Framework aktivieren. Um dort die Dekodierung des Zugriffs vorerst zu vereinfachen, wird zusätzlich von einer CPU mit fester Befehlsbreite und Load-Store-Architektur ausgegangen. Außerdem soll ausgeschlossen sein, dass Zugriffe auf virtuellen MMIO-Bereichen eine oder mehrere Seiten der MMU überspannen. Das vermeidet nicht nur im Codesign-Framework unnötige Komplexität bei der Auswertung und Weiterleitung. Auch bekannte Load-Store-Architekturen, wie ARM [13, S. 112 (A3-6)] oder MICROBLAZE [14, S. 52] setzen die Einschränkung aus diesem Grund um. [4, S. 13ff]

Nun sollen die Vorraussetzungen auf der Betriebssystemebene betrachtet werden. VDM wurde ohne strikten Bedacht auf Eigenständigkeit entwickelt. So zieht es auch die Erweiterung einiger Betriebsysteminterna mit sich, was die Wahl des Unterbaus zusätzlich einschränkt. Nicht zuletzt wegen dieser Einschränkung sollen solche Eingriffe in der Neuauflage vermieden werden. Das Modell des Betriebssystems kann also vereinfacht werden. Wie auch bei VDM ist es wichtig, dass die Organisation und Rechteverwaltung von Prozessen einer Baumhierarchie folgt, deren Verzweigungen auf der Eltern-Kind-Beziehung beruhen. Elternprozesse erzeugen Kindprozesse und haben die komplette Gewalt über deren Rechte. Diese Rechte werden durch Capabilities vermittelt. Jede Capability adressiert eine individuelle Menge von Aktionen auf einer bestimmten Ressource und kann auch an andere Prozesse weitergereicht werden. Für den Start erhält ein

Prozess eine Grundmenge an Capabilities von seinem Elternteil. Darüber hinaus kann er weitere anfordern, indem er per Interprozesskommunikation bei seinem Elternteil anfragt. Der Elternprozess muss den Dienst hinter der gewünschten Capability aber nicht selbst erbringen. Er fragt stattdessen einfach sein Elternteil und seine Kinder, ob sie jemanden mit diesem Dienst kennen. Ist ein passender Anbieter zu dem Nachfrager gefunden, wird eine Server-Client-Verbindung zwischen den beiden Prozessen hergestellt. Diese Abkürzung ermöglicht dann den direkten Austausch von Nachrichten. Die Verbindung zwischen Server und Client ist selbst auch nur eine Capability. Sie dient dem Client als Server-Adresse wenn er Anweisungen zur eigentlichen Capability verschicken will. Auch das Betriebssystem, als initialer, oberster Prozess soll die physischen Ressourcen der Zielplattform auf diese Weise bereitstellen. Es ergibt sich also ein Modell welches die Basis für alle virtuellen Räume und somit die Isolation von Anwendungen bildet. Abbildung 1 verdeutlicht die Funktionsweise des Dienstmodells noch einmal an einer einfachen Prozesshierarchie, denn es ist auch die Grundlage für die Emulation von Geräteschnittstellen.

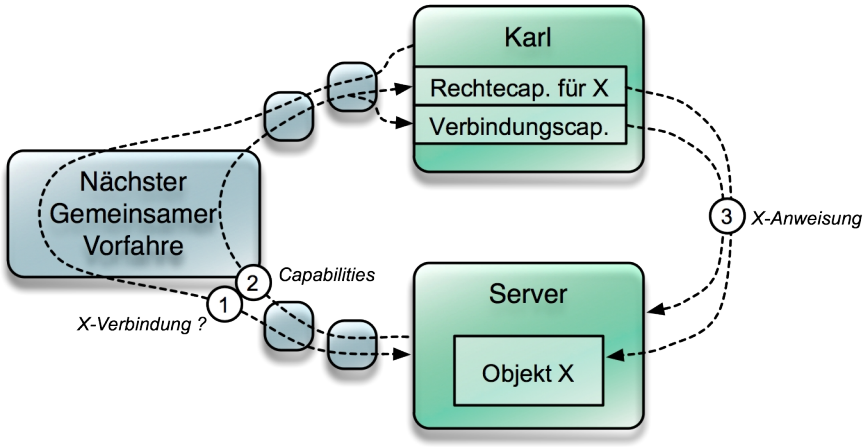


Abb. 1: Karl erfragt Verbindungs- und Rechte-Capability eines Serverobjekts

Betrachtet man das Dienstemodell etwas genauer, fällt auf, dass zum Auffinden eines Servers nur von Elternteil zu Elternteil und von Elternteil zu Kind kommuniziert wird. Die Vermittlung eines Dienstes läuft deshalb zwangsläufig über den nächsten gemeinsamen Vorfahren von Server und Client – ein Umstand, der für diese Arbeit noch sehr wichtig ist. Außerdem setzt die Vermittlung und Nutzung der Dienste die synchrone Interprozesskommunikation voraus. Diese Arbeit wird sich aber – zur Vereinfachung der Interrupt-Emulation – auch der asynchronen Interprozesskommunikation bedienen. Dabei muss jedoch keine Datenlast übermittelt werden. Der Empfänger benennt stattdessen im Vorfeld eine Menge

an Ereignistypen die auftreten dürfen. Dann muss das Betriebsystem nur sicherstellen, dass der Empfänger im Nachhinein jedes Ereignis mitbekommt und einem seiner Typen zuordnen kann.

Abschließend soll noch ein Unterschied zwischen den gemachten Annahmen hervorgehoben werden. Die meisten Bedingungen an die Zielplattform wurden nur zur Vereinfachung der Arbeit eingeführt. Hier bleibt demnach Potential für den zukünftigen Ausbau. Der Rahmen des Betriebssystems jedoch, ist ein fester Bestandteil des Modells. Die Motivation dieser Arbeit liegt nicht zuletzt darin, das Codesign von Hardware und Software mit den Eigenschaften moderner Betriebssystemkonzepte zu bereichern.

2.2 Emulationsumgebung

Die Emulationsumgebung ist die Basis für kooperative Tests von Hardware und Software-Komponenten vor der Fertigung. Sie bildet das Bindeglied zwischen dem Software-Design und Emulatoren, welche die Funktionalität der Hardware imitieren. Dazu bildet sie Software-Zugriffe wenn nötig auf die generische Schnittstelle eines Emulators ab. Diese Schnittstelle bildet in den nachfolgenden Kapiteln auch die Basis für die automatisierte Erzeugung und Anbindung von Emulatoren.

2.2.1 Ausgangspunkt

Das Konzept der vorgestellten Emulationsumgebung entstand auf Basis des VDM-Konzepts. Emulierte MMIO-Bereiche werden in den virtuellen Adressräumen nie physisch hinterlegt. Dadurch blockiert die MMU den Treiber bei jedem Zugiff, um das Betriebssystem darüber zu benachrichtigen. Diese Berichte greift die Emulationsumgebung auf und instruiert einen passenden Emulator, den Zugriff zu verarbeiten. Damit die Emulationsumgebung die MMU-Berichte erhält, wird sie schon bei der Vermittlung des Speichers aktiv. Sie gibt vor, die Rechte an dem emulierten Speicher zu besitzen und liefert dem Treiber eine Capability, die nach außen hin ganz gewöhnlich wirkt, im Hintergrund verursacht die Capability aber die gewünschte Umleitung. Auch für das Abhören emulierter Interrupts, hängt sich die Emulationsumgebung schon bei der Vermittlung der Zugriffsrechte ein. Als Antwort auf die Rechteanfrage erhält der Treiber wieder eine konform wirkende Capability, der aber, wie schon beim MMIO keine reale Ressource zugrunde liegt. Wenn der Treiber am vermeintlichen Interrupt blockiert, benachrichtigt die Capability stattdessen die Emulationsumgebung. Diese wartet dann, bis der entsprechende Emulator den Interrupt aktiviert und deblockiert den Treiber wieder. [4, S. 15ff]

Diese Vorgehensweise soll grundsätzlich gleich bleiben. In VDM sind damit aber mindestens fünf verschiedene Prozesse beschäftigt. Neben Treiber, Emulator und Virtual Device Monitor – namensgebend für VDM – muss auch das Betriebssystem – kurz OS – und der nächste gemeinsame Vorfahre von VDM und Treiber – auch Nearest Common Root, kurz NCR genannt – mitspielen. Abbildung 2 veranschaulicht diese Zusammenhänge an einer typischen Konstellation. Der VDM übernimmt die Verwaltung und Zuordnung der Emulatoren, sowie den Aufbau emulierter Dienste. Allgemein werden ihm all die Aspekte der Emulation zuge-

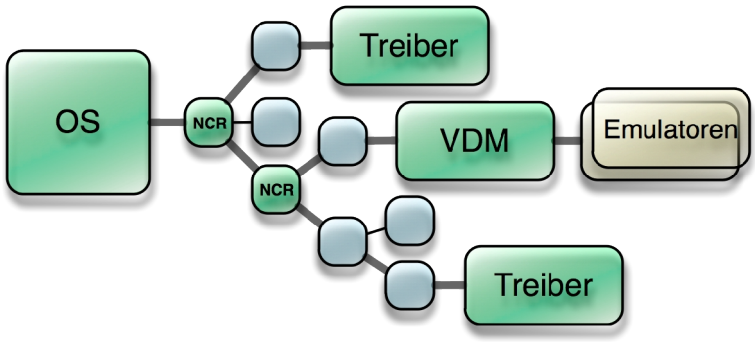


Abb. 2: Prozesshierarchie mit der VDM-Emulationsumgebung

rechnet, die dem Treiber nicht mehr Vertrauen abverlangen als die Benutzung eines echten Geräts. Das beschränkt seine Gewalt auf das Blockieren und Deblockieren des Ausführungskontextes, sowie das Lesen und Zurückschreiben der zu übertragenden Daten. Im Gegenzug kann die Rolle des VDM von jedem beliebigen Prozess eingenommen werden, ohne auf ein besonderes Vertrauensverhältnis zum Treiber achtzugeben. Andere Aspekte hingegen, wie die Dekodierung, welche nur unter vollständigem Vertrauen des Treibers erreicht werden können, übernimmt das Betriebssystem. Übrig bleibt zum Beispiel die Entscheidung, bei welchen Ressourcen VDM statt der normalen Umwelt genutzt wird. Diese Aufgabe wird dem NCR zuteil.

2.2.2 Motivation zur Überarbeitung

Bei VDM entsteht im Betriebssystem durch die Arbeitsaufteilung ein Overhead an Laufzeit und Umfang, der sich nicht nur in der Implementierung zeigt, sondern auch in der Schnittstelle. Jede Anwendung, ob emuliert oder nicht, leidet darunter. Um dem entgegenzuwirken, wird zum Beispiel bei der Behandlung von MMU-Fehlern ein Filter angewandt. Der Filter soll unnötige Dekodierung vermeiden, wenn ersichtlich ist, dass der Speicherzugriff garnicht emuliert werden kann, weil er zum Beispiel keinem MMIO-Bereich angehört. Aufgrund der eingeschränkten Sichtweise können solche Entscheidungen im Betriebssystem aber

nicht präzise getroffen werden. Außerdem erzeugen sie selbst wieder Overhead bei allen Zugriffen. Andererseits ist dieses integrierte Modell bei emulierten Zugriffen schneller, als die vollständige Verlagerung in einen eigenen Prozess. Gewöhnliche Prozesse müssen also einen dauernden Nachteil erdulden, damit bei der Emulation zwei Ziele erreicht werden. Einerseits sollen emulierte Zugriffe für einen Treiber möglichst kostengünstig sein und andererseits ist der VDM auf diese Weise unabhängig davon, wo sich der Treiber im Prozessbaum befindet. Bei einer Coverifikation möchte man den Overhead der Emulation jedoch einschränken können, damit die Software zur Analyse und Bedienung des Tests parallel und unbeeinflusst laufen kann. Die Flexibilität des VDM und die Emulationskosten spielen demgegenüber eine untergeordnete Rolle.

Ein weiterer Nachteil, der durch die Arbeitsaufteilung entsteht, ist die schwer überschaubare Trennung verschiedener Testumgebungen. Möchte man mehrere Software-Tests parallel betreiben, so müssen die Rechteanfragen verschiedener Treiber unterschiedlichen Emulationswelten zugeordneten werden. Das heißt, dass sowohl die NCRs als auch die VDM-Prozesse entsprechend konfiguriert sein müssen und zusätzlich, dass kein Prozess zwischen NCR und VDM zuwiderhandelt. Diese unhandlichen Indirektionen stellen in VDM – ein auf homogene Emulationswelten ausgerichtetes System – kein Problem dar. Für die Erforschung verschiedener Gerätekonzepte wäre es aber wünschenswert, die Testumgebung über verschiedenen Teilbäumen der Prozesshierarchie diversifizieren zu können.

Um diese Mängel zu beheben, musste das Konzept der Emulationumgebung überarbeitet werden. Das Misstrauen des Treibers gegen den VDM soll aufgegeben werden, damit die Emulationsumgebung im Gegenzug zu einer eigenständigen Anwendung vereint werden kann. Diese Entscheidung ging mit der Überlegung einher, dass der VDM ein System mit hohem Wiederverwendungsgrad ist. Wie bereits im ursprünglichen Konzept festgestellt, ist der gut abgrenzbare Dekoder die einzige plattformspezifische Komponente der Emulationsumgebung. Diese Eigenschaft führt vorraussichtlich schnell zu einer Verringerung der Code-Fluktuation und zu einer Erhöhung der Code-Qualität. Ganz im Gegenteil zum Beispiel zu den Emulatoren, welche im Hinblick auf das Codesign potentiell nie vertrauenswürdig sind. Ein anderes Argument ergibt sich aus den Projektzielen. VDM wurde nicht nur für das Anwendungsfeld der Geräteentwicklung geschrieben, es sollte auch die Fallback-Emulation in fertigen Systemen ermöglichen, in denen es mitunter mehr Gerätenutzer als Geräte gibt. Das neue Projekt jedoch, konzentriert sich ausschließlich auf das Codesign. Für den Fallback-Mechanismus ist, angesichts der damit einhergehenden Nachteile eine dedizierte Lösung sinnvoller. Eine wichtige Konsequenz daraus ist, dass die Emulationsumgebung nun nicht mehr auf die Rolle eines rein optionalen Werkzeugs reduziert werden muss. Vielmehr erlaubt ihr die Coverifikation, ein integraler Bestandteil des Trei-

berumfelds zu werden, dessen Einsatz gewollt ist. Das prinzipielle Misstrauen des Treibers ist also nicht mehr zweckgemäß. Abschließend sei erwähnt, dass die Umsiedelung der ehemaligen Betriebssystemerweiterung in einen eigenen Prozess sogar Stabilitätsvorteile mit sich bringt. Denn es bedeutet, dass der betroffene Code von einer hardware-nahen, komplexen Umgebung mit geringem Abstraktionslevel, in einen spezialisierten Prozess der Anwendungsebene übergeht. Letzterer deckt ein weit überschaubareres Aufgabenfeld ab und operiert zudem auf einer standartisierten Schnittstelle.

2.2.3 Die überarbeitete Emulationsumgebung

Eine der wichtigsten Aufgaben, die das Betriebssystem für VDM übernimmt, ist die Aufarbeitung der MMU-Berichte. Das Ziel ist eine Beschreibung des Speicherzugriffs, mit der in VDM eine plattformunabhängige Übersetzung auf die Emulatorschnittstelle gelingt, ohne dass erneut in die Privatsphäre des Treibers eingegriffen werden muss. Mit Privatsphäre sind z.B. der Speicher und der Ausführungskontext des Treibers gemeint. Da diese Ressourcen aber gerade Befehlscode und -argumente eines Zugriffs beherbergen, muss z.B. die Dekodierung bereits im Betriebssystem stattfinden. Dort sind zwangsläufig alle Zugriffsrechte und Übersetzungen der virtuellen Adressräume verfügbar. Um diese Vorrausset-

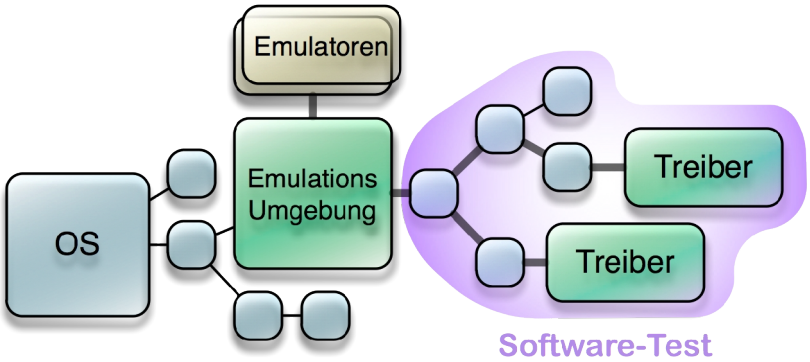


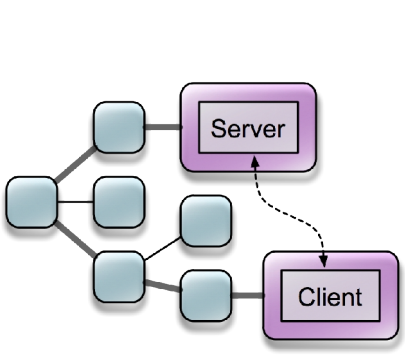
Abb. 3: Prozesshierarchie mit der neuen Entwicklungsumgebung

zungen auch in einem Prozess oberhalb des Betriebssystems herzustellen, müssen die Dienste des Treibers zumindest über den Prozess vermittelt werden. In dieser Rolle des Mittelsmanns wäre es ihm möglich, die Kommunikation zwischen OS und Treiber auch für eigene Zwecke, wie die Emulation zu nutzen. Wie bereits erwähnt, verläuft die Vermittlung von Diensten – ob emuliert oder nicht – über den nächsten gemeinsamen Vorfahren von Server und Client. Ihm müssen zudem beide Prozesse, sogar ohne Emulation vollstes Vertrauen entgegenbringen. Deshalb liegt es nahe, den NCR von Treiber und Emulator als neue Emulationsumgebung zu wählen. Da Emulatoren aber immer direkte Kinder der Emulationsumge-

bung sind, bedeutet diese Neuwahl vor allem, dass die Position des Treibers nun eine viel offensichtlichere Rolle spielt. Der Anwendungsradius einer Emulationsumgebung entscheidet sich nun nicht mehr je nach Routing des gesamten restlichen Systems. Er beschränkt sich einfach auf den Teilbaum der Prozesshierarchie, dessen Wurzel die Emulationsumgebung ist. Diese neue Sichtweise spart Indirektionen bei der Emulationsvermittlung und macht Systeme mit mehreren Emulationsumgebungen verständlicher. Abbildung 3 verdeutlicht das neue Konzept von Emulationsumgebung und Software-Test-Subsystem, anhand einer beispielhaften Prozesshierarchie.

Da die Emulationsumgebung nun alle Verbindungen des Treibers vermittelt, kennt sie auch die Verbindungs-Capabilities. Damit kann sie Anweisungen an die Server schicken und Einfluss auf die Ressourcen des Treibers nehmen. Aber wie die Ressourcen beim Treiber heißen, weiß sie trotzdem nicht. Als Beispiel soll die MMIO-Emulation dienen. Über den Dienst für Adressraumverwaltung kann sich die Emulationsumgebung alle MMU-Berichte zusenden lassen, die zu dem MMIO-Bereich gehören. Angaben in dem Bericht, wie der Befehlszeiger sind aber treiberlokal. Die Emulationsumgebung muss solche Adressen erst auf lokale Capabilities übersetzen und diese in den eigenen Adressraum einhängen. Damit das funktioniert, ist eine gewisse Vorbereitung nötig. Der Treiber muss eine Ressource, wie den Speicher des Befehlszeigers irgendwann vor dem MMIO-Zugriff lokal eingehängt haben. Dies geschah zwangsläufig über einen Dienst den die Emulationsumgebung kennt. Bislang ist sie aber nur beim Aufbau dieser Verbindung involviert, während die Verbindung selbst direkt zwischen Server und Client verläuft. Die Emulationsumgebung kann das Einhängen des Speichers also nicht mithören. Dieses Problem lässt sich mit einem kleinen Trick lösen. Statt dem Treiber bei einer Dienstanfrage die echte Verbindung zuzusenden, erzeugt die Emulationsumgebung eine sogenannte Monitoring-Verbindung. Diese Verbindung, die aussieht wie die echte, erhält der Treiber als Antwort. Anweisungen, die er der Verbindung dann schickt, landen direkt bei der Emulationsumgebung, die sie unverändert an den eigentlichen Server weiterschickt. Durch diese Indirektion kann eine Monitoring-Verbindung die gesamte Kommunikation der eigentlichen Verbindung abhören und die benötigten Informationen sammeln. Die Abbildungen 4 und 5 verdeutlichen das Konzept eines Monitors. Der Weg der Dienstvermittlung wird dort mit dicken Linien markiert, während die Kommunikation eines laufenden Dienstes durch Pfeile dargestellt ist. Der reine Aufwand der Monitor-Indirektion ist gering. Es müssen lediglich eine weitere RPC-Anfrage und ihre Antwort übertragen werden. Darüber hinaus entsteht nur Overhead in den Funktionen, deren Daten tatsächlich von Bedeutung sind.

Mit diesen Vorüberlegungen soll der Ablauf der MMIO-Emulation nun vertieft werden. Die Emulationsumgebung hält, wie besprochen, einen Monitoring-Dienst für die Adressraumverwaltung. Dieser merkt sich, immer dann, wenn ein Kind Speicher einhängt, die lokale Adresse und die Speicher-Capability dazu. Für das Ergebnis und die Argumente eines MMIO-Zugriffs, muss die Emulationsumgebung außerdem den Ausführungskontext des Treibers lesen bzw. schreiben können. Die dafür nötigen Zugriffsrechte werden im allgemeinen nicht vom Treiber selbst in Anspruch genommen, auch wenn er sie grundsätzlich besitzt. Ausführungskontexte stehen aber zumindest mit der Zuordnung und Steuerung von CPU-Zeit im Zusammenhang. Ein Treiber besitzt also zwangsläufig je Ausführungskontext eine Capability, über die er den entsprechenden Thread starten und pausieren kann. Diese Capabilities erhält der Treiber an dem Dienst, an dem er die Threads auch erzeugt, und der desweiteren einfach CPU-Dienst genannt wird. Der CPU-Dienst muss die Zuordnung zwischen Thread-Capabilities



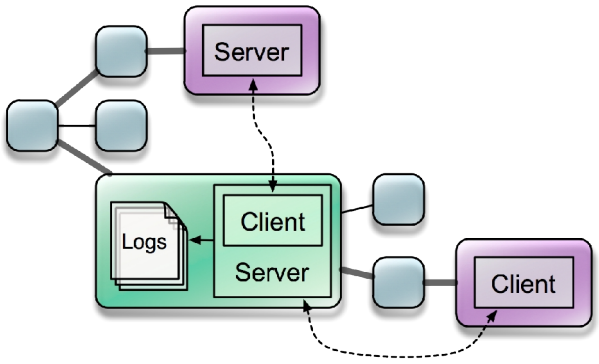


Abb. 5: Normale Dienstkommunikation *Abb. 4: Dienstkommunikation mit Monitor*

und Ausführungskontexten kennen und letztere auf Hardware-Ebene zur Anwendung bringen. Deshalb liegt es nahe, dass er zu einer Thread-Capability auch die Zugriffsrechte des Ausführungskontextes herausgeben kann. Will die Emulationsumgebung diese Rechte also erlangen, merkt sie sich erst einmal die Thread-Capabilities ihrer Kinder, indem sie ein Monitoring für den CPU-Dienst einrichtet. Die erlangten Capabilities muss sie nun nur noch mit den MMU-Berichten des Treibers in Zusammenhang bringen. Dazu greift sie wieder auf den Dienst der Adressraumverwaltung zurück. Dieser muss die MMU-Berichte nämlich selbst zuordnen, ehe er sie weiterschickt. Ansonsten wäre eine individuelle Weiterleitung, an Prozesse wie die Emulationsumgebung nicht möglich. Die Threads deren MMU-Berichte der Adressraum-Dienst verwaltet, muss ihm der Treiber also zuvor genannt haben. Der Dienst darf eine Thread-Capability aber auf keinem Fall mit einem MMU-Bericht verschicken. Der Bericht würde sonst mit Rechten angereichert, die weit über seinen Zweck hinausreichen. Ein Fehlerbericht müsste also eine Art Signatur tragen die einzig zur Wiedererkennung taugt. Da die Emu-

lationsumgebung einen Monitor zwischen Treiber und Adressraum-Dienst geschaltet hat, kann sie eine solche Signatur einführen. Nennt der Treiber dem Adressraum-Dienst einen neuen Thread, so erzeugt der Monitor eine neue Signatur die er der Anweisung hinzufügt und merkt sich die Thread-Capability. Eine Identifikation der Threads, anhand der MMU-Berichte, bleibt so dem Erzeuger der Signaturen vorbehalten. Nun ist die Emulationsumgebung selbstständig in der Lage, Dekodierung und Emulation des Zugriffs nach dem ursprünglichen Modell zu erledigen.

In VDM ist die Emulation von Interrupts – im Gegensatz zu der von MMIO – noch nicht implementiert. Das Modell sieht jedoch vor, dem Emulator freie Software-Interrupts als seine eigenen zu vermitteln. Der VDM soll dann einen Thread auf dem realen Interrupt blockieren lassen und Statuswechsel auf den virtuellen Interrupt des Treibers übersetzen. Mit Hilfe der Annahmen über das Betriebssystem, lässt sich dieses Modell jedoch vereinfachen. Software-gesteuerte Interrupts stellen eine sehr beschränkte Ressource dar, wenn die Zielplattform sie überhaupt anbietet. Ihre Benutzung lässt sich aber einsparen. Wie bereits erwähnt, ist für jeden emulierten Interrupt ein eigener Thread nötig. Sonst würde das Warten auf den Interrupt andere Teile der Emulationsumgebung blockieren. Statt nun auf einen realen Interrupt zu blockieren, kann dieser Thread auch die Mittel der asynchronen Kommunikation nutzen. Dazu gibt er dem Emulator – statt des Interrupts – die Capability eines Ereignisses, das er zuvor kreiert hat. Über diese Capability kann der Emulator dann Statusänderungen seines Interrupt-Ausgangs asynchron vermerken. Da die Emulationsumgebung so zumindest die Anzahl der Statusänderungen vollständig registriert, lässt sich der Interrupt-Verlauf ihrerseits nachvollziehen. Dieser optimierte Ansatz kann auch ohne weiteres in die NCR von Treiber und Emulator übernommen werden. [4, S. 17ff]

2.3 Erzeugung von Emulatoren

Ein großer Vorteil der Emulationsumgebung ist, dass jeder Emulator über die gleiche, generische Schnittstelle mit ihr kommuniziert. Das Interface, an dem die Emulationsumgebung Aufträge erteilt wird jedoch vom Emulator betrieben. Deshalb ist für die Anbindung eines neuen Emulators, der vergleichsweise geringe Aufwand eines RPC-Servers nötig. Aufgabe des Servers ist es, die interne Funktionalität des Geräts auf die Emulatorschnittstelle abzubilden.

2.3.1 Vorüberlegung zur Automatisierung

Die Beschreibung der Gerätefunktionalität ist ohne weiteres also nur in der, vorraussichtlich imperativen bzw. objektorientierten Hochsprache des Servers möglich. Auf diese Art ist die Beschreibung von typischen Hardware-Aspekten

nicht nur umständlich, sondern auch ungeeignet für die spätere Fertigung. Eine viel praktischere Alternative würden Hardwarebeschreibungssprachen – kurz HDLs – darstellen. Mit ihnen lassen sich Aspekte der Register-Transfer-Ebene und Transaktionsebene direkt modellieren. HDLs wie VHDL, SYSTEMC oder VERILOG sind außerdem darauf ausgelegt, Hardware so zu beschreiben, dass die Fertigung danach automatisiert erfolgen kann. Diese sogenannte High-Level-Synthese – kurz HLS – wird durch Werkzeuge wie CATAPULT C [39] oder ICARUS VERILOG [40] möglich. Es ist dazu aber nötig, je nach Sprache und HLS-Programm einige Regeln bei der Beschreibung des Gerätes einzuhalten. Diese beschränken zwar für gewöhnlich die syntaktischen Mittel der Sprache, nicht aber ihre Mächtigkeit. HDLs stellen also einen vollwertigen Ansatz zur textuellen Entwicklung und Synthese von Geräten dar. Deshalb sollten Emulatoren auf die HDL-Beschreibung des Geräts setzen.

Der Gluecode hingegen, der das Gerät mit der Emulationsumgebung verbindet, sollte nicht mit einer solchen Gerätebeschreibung interferieren müssen. Bei der späteren Synthese spielt schließlich nur die interne Funktionalität des Emulators eine Rolle. Für die Gestaltung von Aspekten wie IPC ist eine HDL ohnehin denkbar ungeeignet, da sie nur Datenflüsse ohne expliziten Zugriff auf Speicher oder CPU-Register beschreibt. Wünschenswert wäre also eine automatische Überführung des HDL-Codes in ein Programm, mit dem der Gluecode über eine einfache Schnittstelle interagieren kann. Über diese Schnittstelle muss auch das zeitliche Modell des Gerätes gesteuert werden. Dies ist eine bekannte Problematik, die der Erzeugung von Simulationen aus HDL-Code sehr ähnlich ist. Die Simulation war ursprünglich die eigentliche Motivation für die Entwiclung von HDLs. Bei der Emulatorenerzeugung ist nun lediglich die Testumgebung anders geartet. Statt eine Test Bench, oder eine künstliche Oberfläche zur menschlichen Bedienung anzubinden, kommt hier indirekt der reale Treiber zum Einsatz.

2.3.2 Modell des Generators

Die Idee, den Generator für die Simulation komplett selbst zu entwickeln, war bereits nach kurzer Recherche verworfen. Wie Abbildung 6 für existierende Lösungen zeigt[2], hätte dieses Vorhaben vorraussichtlich den Rahmen der Arbeit überschritten. Dabei stellte sich aber auch heraus, das konkrete Ansätze bereits ausreichend vorhanden sind. Im Grunde sind nur deren äußere Eigenschaften, also Schnittstelle, Performanz und Umfang interresant für diese Arbeit. Deshalb war die nächste Überlegung, einen vorhandenen Simulationsgenerator zu integrieren. Zur Compile-Zeit kann dann vorhandener HDL-Code erkannt und dem Simulationsgenerator zugeführt werden. Dieser soll so modifiziert sein, dass er

2 Quelle: Code-Only-Sparte des Linux-Shell-Kommandos `cloc`, angewandt auf die Quellcodes, Stand 24.10.2012

zum Beispiel keine Testumgebung kompiliert. Dafür soll die Simulation mit einer geeigneten Schnittstelle zur Steuerung ausgestattet sein. An dieser Stelle wird klar, dass eine gut überlegte Wahl des Simulators von Vorteil ist. Er sollte zum Beispiel – zumindest zwischenzeitlich – eine Repräsentation der Simulation erzeugen, die auf dem Level einer Hochsprache mit dem Server kommunizieren kann. Sonst läßt sich die Anbidnung nur durch die Interpretation des erzeugten Maschinencodes bewerkstelligen – ein ungleich komplexeres Unterfangen.

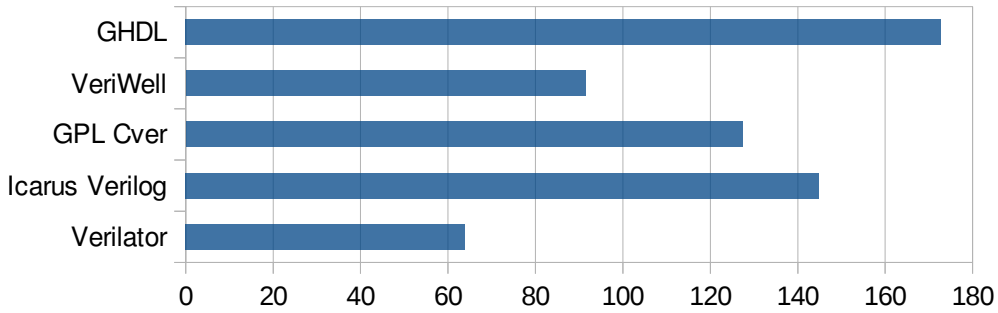


Abb. 6: Größe bekannter HDL-Simulatoren in tausend Lines Of Code

Für die Kommunikation mit anderen Sprachen bieten einige Tools, wie GHDL den Einsatz fremdsprachiger Funktionen [11, S. 25ff]. Diese sind im HDL-Code deklariert und können dort genutzt werden. Ihre Implementierung findet jedoch in einer anderen Sprache statt und wird als Bibliothek gegen die Simulation gelinkt. Vorteil dieser Methode ist die größtmögliche Flexibilität bei der Gestaltung der Schnittstelle. Lagert man den HDL-Code einer solchen Anbindung in eine eigene Komponente aus, bleibt die Gerätebeschreibung zudem unberührt. Ein anderer Ansatz – wie ihn z.B. VERILATOR umsetzt [12, S. 23f] – besteht darin, die Simulation erst selbst in eine andere Hochsprache zu transkompilieren. Dabei lassen sich wieder zwei Kategorien von Simulatoren unterscheiden. Bei der einen Kategorie muss die Testumgebung bereits als HDL-Beschreibung mitgeliefert werden. Sie wird dann, mit den zu testenden Komponenten in die Simulationssprache überführt. Dabei muss jedoch keine klare Schnittstelle zwischen beiden existieren. Designexterne Effekte können so auch in die Gerätebeschreibung hineingeraten. Undokumentiert wendet diese Vorgehensweise zum Beispiel GHDL an. In diesem Fall ist die zwischenzeitliche Repräsentation natürlich nur nutzbar, wenn der Entwickler das Design so aufarbeitet, dass sich eine geeignete Schnittstelle ergibt. Die praktischere Alternative dazu sind solche Simulatoren, die die Testumgebung erst in der Zweitsprache integrieren. Es liegt im Konzept begründet, dass hier eine klare Abstraktionsschicht zwischen Simulation und Testumgebung geschaffen werden muss. Diesen Ansatz verfolgt VERILATOR, bei dem ein Design durch eine Klasse repräsentiert wird, deren Attribute und Methoden eine umfassende Steuerung ermöglichen.

In dem Modell der Arbeit fiehl die Wahl schließlich auf die letzte Variante der Anbindung. Dafür gab es mehrere Gründe. Simulatoren, die HDL-fremde Funktionsaufrufe im Design zulassen, waren eigentlich die erste Wahl während der Ausarbeitung. Dass diese Entscheidung bald verworfen wurde lag jedoch nicht am Konzept, sondern daran, dass alle betrachteten Tools Abhängigkeiten zu anderen, umfangreichen Bibliotheken und Programmen aufwiesen. Der enorme Portierungsaufwand führte zu der Erwägung, eine Alternative zu suchen. Es blieb also nur noch die Anbindung über die Zwischenrepräsentation. Hier lag der Fokus von Anfang an auf den Tools, die eine klare Abstraktionsschicht einführen. Dies ging mit der Überlegung einher, dass eine konzeptionelle Limitierung gegenüber Vereinbarungen weniger Spielraum für Fehler lässt.

Auch die Integration des Tools soll so weit wie möglich automatisiert werden. Bereits die Vorbereitungsphase lässt sich intuitiv gestalten. Der Simulator erzeugt, wie besprochen eine geeignete Schnittstelle, unabhängig vom konkreten HDL-Design. Somit kann angenommen werden, dass die integrationsbedingten Änderungen am Quellcode des Generators statisch sind. Es kann also bereits im Vorraus ein unabhängiges Programm den Quellcode laden, modifizieren und kompilieren. Dabei müssen vor allem drei Probleme gelöst werden. Einerseits muss der Generator, wenn er die Simulation kompiliert, die betriebssystembedingten Toolchain-Eigenarten und -Einstellungen zur Anwendung bringen. Zum anderen soll keine Testumgebung im Sinne einer Simulation kompiliert werden. Der Generator kann also nach dem Transkompilieren abgebrochen werden, da die zweite Kompilierung und das Linken gegen eine Simulationsumgebung entfällt.

Nach dieser Vorbereitung kann der Generator beim Bau der Testumgebung, von der nativen Toolchain auf alle HDL-Quellen angesetzt werden. Für jedes Top-Level-Design wird er angewiesen, temporäre Objekt-Dateien zu transkompilieren. Diese enthalten eine imperative Simulation des Geräts und werden dann jeweils, wiederum von der nativen Toolchain, gegen den Gluecode zur Emulationsumgebung und die notwendigen Bibliotheken gelinkt. Damit ergibt sich je Design ein eigenständiges Programm, das in der Testumgebung als Emulator genutzt werden kann.

2.4 Integration der Emulatoren

Nun ist es möglich, die interne Funktionalität eines Geräts direkt durch HDL-Code in das Testszenario einzubinden. Es bleibt aber die Übersetzung der Emulatorschnittstelle auf die Abstraktionsschicht der Simulation. Diese stellt für den Entwickler immer noch gerätespezifischen Aufwand dar, dessen Ergebnis nur für die Verifikation gebraucht wird. Deshalb soll der Gluecode nun genauer betrachtet werden.

2.4.1 Bus-Clients

Die Schnittstelle, die VERILATOR an der Simulation erzeugt, vermittelt expliziten Lese- und Schreibzugriff auf den Ein- und Ausgangssignalen des Geräts. Bei realen Geräten ist diese Schnittstelle ebenfalls vorhanden. Üblicherweise adressiert ein Treiber aber nicht die einzelnen Signale des Geräts. Sonst müsste die CPU für jedes Gerät einen speziellen Befehlssatz und entsprechende IO-Signale implementieren. Wie bereits in Kapitel 2.2 besprochen, greift der Treiber hingegen über einen generischen Befehlssatz auf Adressen des Speicherbus zu. In der Realität ordnet der Bus die Adressen dann seinen Clients zu. Bei Peripherie, wie sie in dieser Arbeit betrachtet wird, ist das der Bus-Client, welcher dem Gerät vorgelagert ist. Er übernimmt das Protokoll des Bussystems und tauscht darüber die Informationen mit dem Master aus. Der ist in diesem Fall Teil der CPU, auf der der Treiber ausgeführt wird. Damit Client und Gerät die Nutzdaten austauschen können, muss das Gerät also statt einer individuellen Schnittstelle die des Bus-Clients aufweisen. Die Client-Schnittstelle ist durch die Busspezifikation geräteunabhängig deklariert. Dieser Umstand eröffnet auch in der Entwicklungsumgebung Potential zur Vereinheitlichung.

Statt jedes Gerät individuell anzusteuern, soll eine Sammlung von Software-Adaptern für häufig genutzte Bussysteme zur Verfügung stehen. Aus dieser Sammlung kann der Entwickler, je nach Emulator und Szenario eine passende Zusammenstellung wählen. Außerdem kann er die Auswahl schon vornehmen, bevor er das Szenario kompilliert, da sowohl die Emulatorschnittstelle, als auch die Gerätesignale statisch deklariert sind. Nun gibt es unter Umständen mehrere Instanzen einer Busanbindung. Es kann sogar nötig sein, Zugriffe zu synchronisieren, da Signale von unterschiedlichen Adaptern zugleich genutzt werden. Es bietet sich deshalb an, den Bus-Client als Klasse zu implementieren. Sie bekommt vom Entwickler mitgeteilt, welche Signale des Designs eine Aufgabe des Bus-Clients erfüllen. Falls nötig nimmt sie auch andere Parameter des Bus-Protokolls entgegen. Zum Beispiel die Breite von Daten- und Adress-Input. Kommt dann ein MMIO-Zugriff an der Emulatorschnittstelle auf, treibt der Adapter die Signale des Designs entsprechend des Busprotokolls. Zu erwähnen bleibt noch, dass es sich bei VERILATOR-Simulationen um taktgenaue State Machines handelt. Der Adapter muss also jedes Eingabe-Array explizit evaluieren lassen. Damit liegt, neben der funktionalen Übersetzung auch die Bedienung des Bustaktes in der Hand der Klasse.

2.4.2 Taktgeber

Nun lassen sich zwar MMIO-Zugriffe auf das Gerät abbilden, ein paralleler Arbeitsfluss entsteht jedoch nicht in der Simulation. Das liegt daran, dass Takteingänge bisher nur bei Zugriffen des Treibers betrieben werden. Finden keine

Transaktionen statt, steht die Simulation still. Um dieser unnötigen Abweichung von der Realität entgegenzuwirken, muss es möglich sein an beliebigen Eingangssignalen einen stetigen Takt zu simulieren. Dazu kann eine neue Klasse – der Taktgeber – eingeführt werden. Ihr wird vom Entwickler die Taktfrequenz und das betroffene Signal übergeben. Außerdem wird jedem Taktgeber mitgeteilt, ob der Takt auf der steigenden oder der fallenden Flanke des Signals läuft. Ein typischer Takt von mehreren Kilo- oder Megahertz würde jedoch, genau umgesetzt zuviel CPU-Zeit in Anspruch nehmen. Deshalb erzeugt der Taktgeber die Flanken einfach gebündelt, mit einer weit niedrigeren Frequenz. Diese niedrigere Frequenz gibt der Entwickler ebenfalls, als sogenannte Update-Frequenz an. Jeder Taktgeber stellt für sein Signal einen eigenen Thread an. Der Thread nutzt dann einen realen Timer um periodisch, entsprechend der Update-Frequenz geweckt zu werden.

Nun kann es auch vorkommen, dass ein Taktsignal von einem anderen Tool und somit einem anderen, nebenläufigen Thread genutzt wird. Dieses Tool ist unter Umständen darauf angewiesen, seine Taktflanken asynchron zur Update-Frequenz des Taktgebers zu betreiben. So eine Situation tritt zum Beispiel ein, wenn ein Bus-Client Transaktionen über den selben Takt abwickelt, wie ihn auch die funktionalen Teile des Geräts nutzen. In diesem Fall muss der Taktgeber mit den anderen Tools synchronisiert werden. Der Entwickler kann z.B. ein Lock erzeugen, welches er beiden Anwärtern mitgibt. Nun ist zwar der Zugriff synchronisiert, aber die zusätzlichen Flanken der anderen Tools beeinflussen die Genauigkeit des Taktgebers trotzdem. Um dieses Problem zu lösen, wird das Konzept des Taktgebers noch einmal überarbeitet. Der neue Taktgeber hält einen Zähler über die ausstehenden Taktflanken und implementiert außerdem die Schnittstelle eines HDL-Signals. Dadurch kann er selbst als vermeintliches Signal an andere Tools, wie den Bus-Client weitergereicht werden. Betätigen diese dann den Takt in gewohnter Manier, wird der Zähler des Taktgebers entsprechend herabgesetzt. Der Taktgeber selbst löst, wenn er aufwacht, nur die verbleibenden Flanken aus und setzt den Zähler wieder auf das Rundensoll. Das Zugriffs-Lock wird – auch wenn es nun direkt vom Takgeber verwaltet werden könnte – weiterhin außerhalb erzeugt und den zu synchronisierenden Tools mitgegeben. So kann jedes Tool individuell CPU-Zeit sparen, indem es schnell aufeinanderfolgende Zugriffe in nur einem Lock-Zyklus erledigt.

2.4.3 Interrupt-Listener

Der nächste Schritt besteht darin, Signale des Designs als Interrupt zu deklarieren. Ihr Zustand soll dann automatisch erfasst und wartende Clients gegebenfalls informiert werden. Dadurch steht zuerst die Frage im Raum, wann ein solches Signal abgehört werden soll. Der Schluss liegt nahe, dass ein Interrupt immer

synchron zu einem Taktgeber auftreten muss. Ohne einen autonomen Takt – wenn dieser zum Beispiel nur vom Bus-Client getrieben wird – macht ein asynchrones Kommunikationsmittel wenig Sinn. Aus diesem Grund soll eine alternative Version des Taktgebers – der Interrupt-Taktgeber – geschaffen werden. Dieser erhält, zusätzlich zu den Taktgeber-Parametern ein sogenanntes Interrupt-Array. Durch ein Interrupt-Array kann ein Entwickler mehrere Interrupts zu einer Gruppe vereinen. Interrupts können auch in mehreren Arrays enthalten sein. Der Interrupt selbst setzt sich aus zwei Information zusammen – dem HDL-Signal und dem asynchronen Ereignis der Emulationsumgebung, das bei einer Interrupt-Flanke ausgelöst werden soll. Das HDL-Signal teilt ihm der Entwickler mit. Die

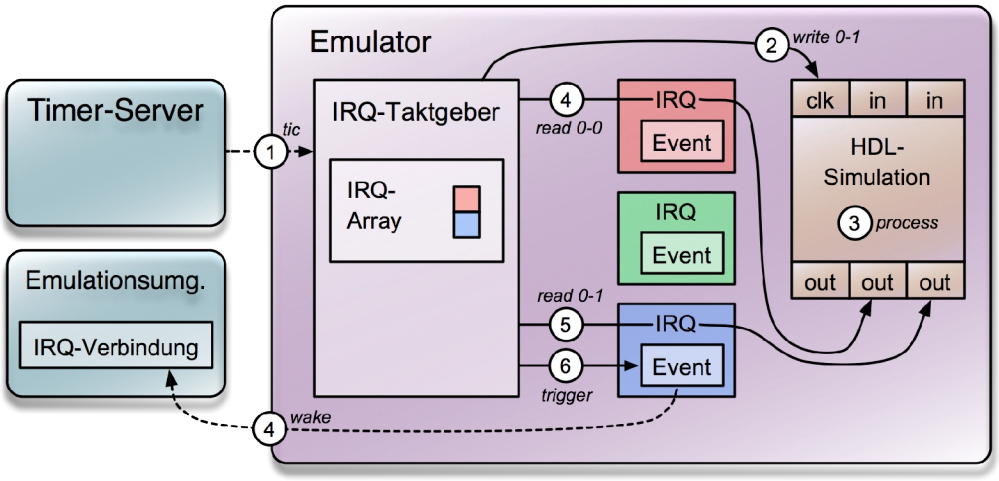


Abb. 7: Ein IRQ-Taktgeber betreibt den roten und den blauen IRQ

Capability des asynchronen Ereignisses hingegen, wird on-demand durch die Emulatorschnittstelle gesetzt bzw. wieder gelöscht. Dadurch kann die Emulationsumgebung selbst bestimmen, wann sie einen Interrupt erwartet und wann nicht. Der Interrupt-Taktgeber prüft nun bei jeder Takflanke alle Interrupts aus seinem Array. Ist ein Interrupt über der Taktflanke gekippt, löst der Interrupt-Taktgeber das asynchrone Ereignis aus. Durch diese Vorgehensweise werden zusätzliche Threads für die Interrupt-Auswertung gespart. Abbildung 7 zeigt das Zusammenspiel von Timer, Interrupt-Taktgeber, Interrupts und Simulation. Die Abkürzung “IRQ” wird in der Abbildung stellvertretend für “Interrupt” benutzt.

Offen bleibt bisher aber, wie genau die Emulatorschnittstelle die asynchronen Ereignisse installiert und deinstalliert, wenn ein Treiber auf eine bestimmte Interrupt-Nummer warten möchte. Erforderlich ist eine Instanz, der sämtliche Interrupts des Emulators unter Namen bekannt sind, die auch die Emulationsumgebung je Gerätetyp kennt. Da diese Namen keine Rolle für den Entwickler spielen, sollen sie je Gerätetyp einfach von 0 aufwärts zählen. Der Entwickler bewerkstel-

ligt diese Nummerierung durch ein neues, allumfassendes Interrupt-Array. Dessen Index entspricht den Interrupt-Namen, die nach außen hin wirksam sind. Dieses Array übergibt der Entwickler dann einem sogenannten Interrupt-Listener. Der Listener kümmert sich darum, die eingehenden Anfragen der Emulatorschnittstelle auf die Capabilities in dem allumfassenden Array zu übersetzen. In Abbildung 8 ist dieser Vorgang schematisch dargestellt. Da das allumfassende Array gleichzeitig auch von Interrupt-Taktgebern genutzt werden kann, sind bei einfachen Szenarien nicht einmal mehrere Arrays notwendig.

2.4.4 Bus-Master

Bisher wurden nur Bus-Clients betrachtet. Das beschriebene Modell lässt sich aber ebensogut für Geräte nutzen, die einen Bus-Master implementieren, um selbst Transfers zu starten. Für die Anbindung des Bus-Masters wird ein dedizierter Kontrollfluss benötigt, welcher an den Ausgangssignalen des Geräts auf Übertragungsgesuche wartet. Zu diesem Zweck kann eine Klasse ähnlich dem Interrupt-Taktgeber genutzt werden. Die Übertragung wird dann – anders als beim Bus-Client – direkt auf Speicherzugriffe übersetzt. Handelt es sich um MMIO, ist

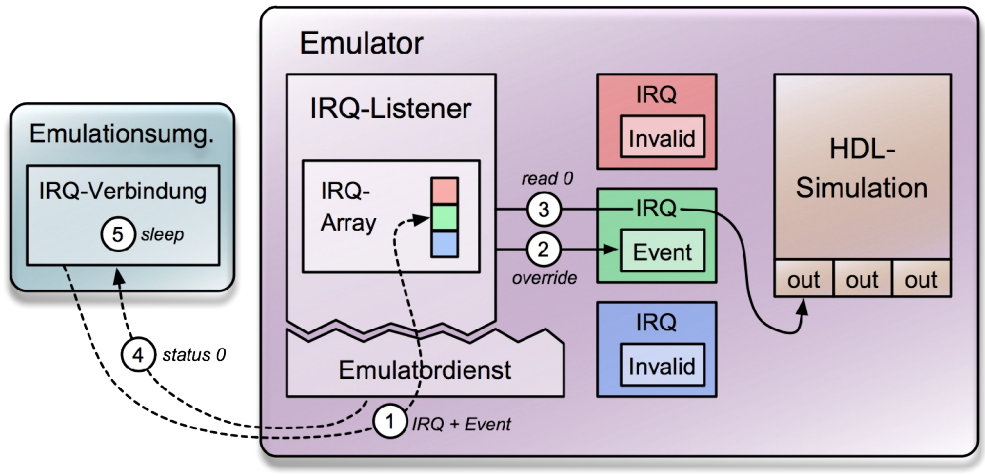


Abb. 8: Eine IRQ-Verbindung blockiert über den Listener auf den grünen IRQ

der Vorgang einfach. Das ist dem Umstand zu verdanken, dass in der Emulationsumgebung keine Annahme darüber gemacht wird, ob ein dienstsuchender Prozess ein Emulator ist oder nicht. Der Emulator kann den MMIO-Speicher, wenn er dazu berechtigt ist, ebenso wie jeder andere Prozess in seinen Adressraum einhängen. Handelt es sich um emuliertes MMIO, lenkt die Emulationsumgebung den Dienst wie gewohnt auf einen weiteren, passenden Emulator um. Das

Abhören der Geräteschnittstelle, das Beschaffen und Einhängen der Speicher-Capabilities, sowie den Zugriff entsprechend des HDL-Output, übernimmt eine neue Klasse – der Bus-Master. Er leitet vom Taktgeber des Bus-Interface ab.

Handelt es sich bei dem fraglichen Speicher nicht um MMIO, stellt die Übertragung einen Direct Memory Access – kurz DMA – auf den Speicher der Zielplattform dar. Bei DMA gestaltet sich der Zugriff etwas komplizierter. Der Emulator muss die Capability des Speicherbereichs anfordern um sie in seinen Adressraum einzuhängen. Er besitzt jedoch nur die physische Adresse des Speicherbereichs. Da das Betriebssystem bei normalem Arbeitsspeicher aber von physischen Adressen abstrahieren kann, ist eine so explizite Wahl der Capability nicht unbedingt möglich. Für einen DMA-Zugriff muss das Gerät die physische Adresse zuvor aber selbst vom Treiber erfahren haben. Es muss dem Treiber also möglich gewesen sein, die physische Adresse, einer einmal erlangten Speicher-Capability nachträglich zu ermitteln. Damit ergibt sich eine Lösung für das Adressenproblem. Die Emulationsumgebung kann bekanntlich sämtliche Dienste ihrer Kinder abhören. Es ist ihr also auch möglich, Buch über die Speicher-Capabilities zu führen, zu denen Kinder die physische Adresse ermittelt haben. Einmal im Besitz dieses Wissens, kann die Emulationsumgebung zumindest jene Capabilities anhand ihrer physischen Adresse finden, die für DMA in Frage kommen. Eben die Fähigkeit, auf die der Bus-Master bei DMA angewiesen ist. Für den Austausch von Adresse und Capability, kann die Emulationsumgebung den Emulatoren einen eigenen DMA-Dienst bieten. Sie kann aber auch einen vorhandenen Dienst erweitern. Der MMIO-Dienst würde sich dafür besonders eignen, da er etwa die gleiche Schnittstelle besitzt und die DMA-Adressen zwangsläufig noch nicht belegt sind. Doch die Vermittlung der DMA-Capabilities muss auch mit Bedacht geschehen, damit das Vertrauensmodell der realen Welt nicht unnötig ausgeweitet wird. Auf einen DMA-Bereich sollen deshalb nur solche Emulatoren zugreifen können, deren Eigentümer zuvor die physische Adresse erfragt haben und somit selbst die Capability, also das Zugriffsrecht besitzen.

3 Implementierung

Die beispielhafte Implementierung und Analyse des vorgestellten Modells soll auf einer möglichst aktuellen RISC-Plattform erfolgen, die eine gewisse Verbreitung aufweist und auch kostengünstig erworben werden kann. Software-seitig soll ein frei verfügbares Open-Source-Betriebssystem zum Einsatz kommen, das die Vorraussetzungen der Arbeit erfüllt. Außerdem sollen sowohl Plattform als auch Betriebsystem ein möglichst breites Spektrum von Anwendungsbereichen adressieren, insbesondere aber den eingebetteter Systeme.

3.1 Hardware-Umgebung

Als Plattform für die Implementierung kommt ein CORTEX-A9 zum Einsatz, der die ARMV7-Architektur umsetzt. Diese ist, wie alle ARM-Systeme eine RISC-Architektur mit zwei getrennten Bussen für Daten- und Befehlszugriffe. Ihr Befehlssatz folgt einer festen Befehlsbreite und gliedert sich in drei Kategorien. Die erste – die Load-Store-Kategorie – dient einzig den Speicherzugriffen. Befehle für den Speicherzugriff interferieren nicht mit anderen Operationen, wie den arithmetisch-logischen, oder denen zur Ausführungskontrolle [13, S. 24 (A1.1)]. Der CORTEX-A9 implementiert die VMSA-Variante der ARMV7-Spezifikation. Die Virtualisierung von Speicheradressräumen wird also durch eine MMU ermöglicht [16, S. 105 (6-2)]. Damit sind hardware-seitig alle Vorraussetzungen der Arbeit erfüllt.

Eine weitere Motivation zu dieser Wahl ist die Tatsache, dass ARM-Prozessoren mittlerweile in Kombination mit frei programmierbaren FPGAs existieren. Der XILINX ZYNQ-7000 zum Beispiel, ermöglicht einem eingebauten CORTEX-A9 über den Speicherbus Zugriff auf die programmierbare Logik [15]. Daraus ergibt sich in Zukunft, so die Hoffnung, weiteres Potential für die hier vorgestellte Codesign-Umgebung. Denn Komponenten könnten dann, soweit FPGA-Fläche verfügbar ist, nicht nur software-seitig, sondern auch ad-hoc durch Hardware emuliert werden. Die Wahl zwischen den beiden Emulationsarten wäre mit dem Modell dieser Arbeit vorraussichtlich transparent für den Entwickler. Beide Varianten setzen nur die HDL-Beschreibung voraus und könnten auf die selbe

Schnittstelle abbilden. Besonders zeitintensive Emulatoren würden so von den schnelleren FPGAs profitieren, während das System bei Platzmangel immer auf die Software-Variante zurückgegreifen kann.

3.2 Software-Umgebung

Die eingesetzte Software-Umgebung basiert, wie auch in der vorangegangenen Arbeit, auf dem GENODE OS-Framework [1]. Dieses Framework beschreibt den leicht portierbaren Mikrokernel eines capability-basierten Betriebssystems. Der Mikrokernel nennt sich in GENODE Core. Darauf aufbauend ermöglicht das Framework die Ausführung von Prozessen, welche durch Virtualisierung der Ressourcen voneinander isoliert sind. Prozesse können neue Prozesse erzeugen und stehen dann in einer Parent-Child-Beziehung zueinander, die eine Baumhierarchie beschreibt. Die Ressourcen, die ein Prozess initial benötigt, werden ihm von seinem Parent direkt bei der Erzeugung zugesprochen. Weitere Rechte werden von Server-Prozessen in Form virtueller Dienste angeboten. Diese Dienste basieren auf Capabilities und synchroner, nachrichtenbasierter Kommunikation. Und auch darüber hinaus folgen sie im wesentlichen dem Modell, welches in Kapitel 2.1 vorgestellt wurde. GENODE richtet sich nicht an einem speziellen Einsatzgebiet aus. Es stellt vielmehr ein flexibles, sicheres und leichtgewichtiges Framework für die individuelle Zusammenstellung eines Systems bereit. Außerdem handelt es sich um ein Open-Source-Projekt, was die zukünftigen Entwicklungs- und Einsatzmöglichkeiten begünstigt.

Der Core von GENODE bietet eine Menge grundlegender Dienste für die Anwendungsentwicklung. An dieser Stelle soll ein kurzer Überblick über diese API folgen, da sie für die Arbeit von großer Bedeutung ist. Die Basis eines Prozesses ist eine sogenannte Protection Domain – kurz PD – welche den virtuell isolierten Raum des Prozesses durch eine PD-Capability referenziert. Die Core-API ermöglicht die Erzeugung und Kontrolle solcher PDs über einen eigenen Dienst. Einer PD können mehrere parallele Kontrollflüsse – kurz Threads – angehören, welche – wie auch die PDs – jeweils durch eine Capability adressiert werden. Zur Erzeugung und Kontrolle dieser Threads bietet der Core ebenfalls einen eigenen Dienst. Jeder Speicherabschnitt, den die MMU virtuell bereitstellen kann, wird durch eine Capability adressiert. Die drei Dienste, die Speicher-Capabilities anbieten, unterscheiden sich in der Art des Speichers. So gibt es einen für Read-Only-Bereiche, einen für den üblichen Arbeitsspeicher – kurz RAM – und einen für MMIO-Bereiche. Nur MMIO-Bereiche werden über ihre physische Adresse angefordert. Read-Only-Bereiche sind durch fiktive Namen adressierbar, während RAM beim Allozieren nicht adressiert werden kann. Speicherreferenzen vermitteln jedoch noch keine Zugriffsmöglichkeit. Das bewerkstelligt erst der Dienst

der Adressraumverwaltung – kurz RM-Dienst, indem er den Speicher-Bereich der Capability lokal einhängt. Der RM-Dienst fängt auch die MMU-Fehler ab, die auf den eingehängten Bereichen ausgelöst werden.

Zur asynchronen Kommunikation ist ein Dienst mit zwei Capability-Arten nötig. Ein Empfänger wird durch die eine Art adressiert. Solch ein Empfänger kennt unterschiedliche asynchrone Signale, jeweils durch eine Capability des zweiten Typs dargestellt. Letztere kann ein Thread an Sender herausgeben und dann am Empfänger darauf blockieren. Er bekommt nun, sobald eine Sendung aussteht, die Signal Capability genannt. Außerdem wird ihm mitgeteilt, wie oft das Signal bis dahin ausgelöst wurde. Diesen Mechanismus setzt auch der RM-Dienst ein. Ein Client kann beim RM-Dienst vermerken, dass der Core nicht selbst versuchen soll die MMU-Fehler aufzulösen. Stattdessen bestimmt der Client ein asynchrones Signal, das bei jedem MMU-Fehler einmal ausgelöst wird. Ein Thread, der den Empfänger kennt und mit dem RM-Dienst verbunden ist, kann die Fehler so abhören und selbst auflösen. Von dieser Möglichkeit macht auch die Emulation von MMIO Gebrauch.

3.3 Portierung von Core

Der nicht direkt notwendige, aber dennoch hilfreiche erste Schritt der Implementierung, bestand darin, eine spezielle Portierung von GENODE's Core anzufertigen. Die folgenden Kapitel sollen die Beweggründe darstellen, die diese Arbeit zu einer Portierung lieferte und deren Konzept kurz umreißen.

3.3.1 Motivation

Für das Verständnis der Portierung ist ein kurzer Einblick in den typischen Startablauf von GENODE notwendig. Core ist immer der erste GENODE-Prozess, der gestartet wird, doch er setzt bereits auf ein Abstraktionslevel von Prozessen, Threads und anderen virtualisierten Ressourcen auf. Unterhalb dieses Levels arbeiten im Normalfall Kernel aus fremder Hand. Kernel wie LINUX, FIASCO.OC [5], OKL4 [8] oder NOVA [6], die so unterschiedliche Rollen wie Monolith, Mikrokernel und Hypervisor verfolgen. Core sorgt also eigentlich dafür, dass ein Anwendungsentwickler nicht zwischen diesen Kerneln und Konzepten zu unterscheiden braucht, wenn er auf die vereinheitlichende Mikrokerneloberfläche von GENODE setzt. Auf die individuellen Vorzüge kann aber, wo es von Nöten ist, dennoch zurückgriffen werden. Da Core in dieser Rolle sämtliche Ressourcen des Boards verwaltet, gehört er zwangsläufig zur Vertrauensbasis aller anderen GENODE-Prozesse. Für die Bereitstellung der Ressourcen benötigt er zudem relativ viel platformspezifischen Code, der auf keine konforme Schnittstelle aufsetzt. Darum liegt es im Interesse der GENODE-Entwickler, diesen Prozess so über-

schaubar wie möglich zu halten und schnellstmöglich in eine abstraktere Umgebung zu wechseln. Core startet also nur einen Prozess namens Init, dem er einfach alle ungenutzten Ressourcen vermacht. So einfach und offen, wie Core gehalten werden soll, so komplex und Fremdeingriffen gegenüber verschlossen sind mitunter die Kernel die ihm als Basis dienen. Da sich ihr Aufbau nicht an GENODE orientiert, enthalten sie, gemessen an diesem Einsatz zumeist überflüssigen Code und verquere Konzepte, die in Core trickreich umgemünzt werden müssen. Mit diesem Wissen führten drei Überlegungen zu dem Entschluss, eine Core-Variante zu schaffen die nicht auf einen Third-Party-Kernel angewiesen ist.

Zum einen ist die generische Core-Schnittstelle – bedingt durch die Kernel – nicht ganz ausreichend für die neue Emulationsumgebung. Wie schon erwähnt, muss es der Emulationsumgebung unter anderem möglich sein, lesend und schreibend auf den gesamten Ausführungskontext des Treibers zuzugreifen. Solche Zugriffsrechte müssen außerdem möglichst genau auf den fraglichen Code eingeschränkt werden, da sie kritisch für die Sicherheit des Treibers sind. Insbesondere andere Prozesse sollen keine Möglichkeit bekommen, die Rechte zu missbrauchen, und auch innerhalb der Emulationsumgebung soll ihr Einsatz überschaubar bleiben. Wie bei GENODE LABS bereits mit der Implementierung der GDB-Schnittstelle festgestellt wurde, bieten bisher nur wenige Kernel eine Lösung für diese Probleme [7]. In Betracht kommen vor allem OKL4 und FIASCO.OC. Letzterer basiert auf der L4.FIASCO Kernel-API, die ihrerseits ein Nachfolger der L4V2-API ist. Zur Thread-Registerkontrolle bietet sie, wie ihr Vorbild den System Call `l4_thread_ex_regs`. Über ihn lassen sich zwar Instruction Pointer und Stack Pointer eines Threads ermitteln und ändern, aber nicht die übrigen General Purpose Register [10]. OKL4 hingegen, erweitert die L4V2-API in diesem Bereich. Er ist damit der einzige, zum Einsatz kommende Kernel, der – durch den System Call `ExchangeRegisters` – den Zugriff auf alle Kontextregister – auch die ARM-spezifischen – ermöglicht [9, S. 29, 82ff, 158].

Die zweite Überlegung ergibt sich dadurch, dass die Emulationsumgebung Aspekte wie die Hardware-Virtualisierung des Speichers abgreift, deren Einfluss tief ins Betriebssystem hineinreicht. Für die Entwicklung der Umgebung ist es also von Vorteil, wenn der Kernel einen guten Einblick in Form von verständlichem Code und individuellem Debugging zulässt. Das Potential dazu variiert jedoch stark zwischen den derzeit eingesetzten Kerneln und ist durch die fremde Verwaltung des Codes mit Hürden verbunden.

Die dritte Überlegung geht auf eine ehemalige Core-Portierung zurück. Zu Beginn der Arbeit setzten alle Core-Versionen auf einen Third-Party-Kernel – mit Ausnahme der MICROBLAZE-Variante. Die MICROBLAZE-Variante adaptiert Core so, dass er direkt auf der Hardware betrieben werden kann, und nur das imple-

mentiert ist, was GENODE tatsächlich braucht. Sie ist jedoch konzeptionell sehr komplex und leicht instabil, wie sich bereits bei der Grundlagenarbeit zur Emulation herausstellte [4, S. 26ff]. Dennoch stellt der MICROBLAZE-Core Wissen um die niederen Betriebssystemfunktionen bereit und stammt aus eigener Hand. Die Überarbeitung und Weiterentwicklung dieses Konzepts liegt außerdem im Interesse von GENODE LABS. Deshalb – und wegen der ersten beiden Überlegungen – wurde der Entwicklung der Emulationsumgebung eine neue, eigenständige Core-Portierung vorangesetzt, die sich aus den Erkenntnissen der MICROBLAZE-Variante nährt. Der Flexibilität und dem Debugging-Potential dieser Portierung ist letztendlich zu verdanken, dass die Entwicklung der Emulationsumgebung so reibungslos vonstatten ging. Grundsätzlich ist der Einsatz auf anderen Kerneln aber ebenso möglich, wie erwünscht. Deshalb ist es sinnvoll, ein Verständnis dafür zu vermitteln, wie den Anforderungen der Emulationsumgebung auf Core-Ebene Folge geleistet wurde.

3.3.2 Konzept der Core-Portierung

Die neue, eigenständige Core-Variante besitzt grundlegende Gemeinsamkeiten mit der MICROBLAZE-Variante. Core ist zum Beispiel wieder zweigeteilt. Der Code der den privilegierten CPU-Modus benötigt, wurde auf ein Minimum reduziert und läuft als single-threaded Kernel, ohne den Bedarf einer dynamischen Speicherverwaltung. Er besitzt nur zwei Eintrittspfade. Der erste gilt der Initialisierung des Boards und der statischen Kernelobjekte. Mit den statischen Objekten wird auch der Core-Prozess, mit genau einem zugehörigen Thread erzeugt. Danach wird der Kernel nur noch über den zweiten Eintrittspfad, im Zuge einer Hardware-Exception betreten. Dieser sichert den unterbrochenen Ausführungskontext und die Exception-Parameter. Daraufhin wird im Kernel die geringstmögliche Behandlung der Exception durchgeführt und das Scheduling des nächsten Threads vorgenommen.

Auf den Kernel setzt der Anteil von Core auf, der im Unprivileged Mode der CPU läuft. Dieser Code bedient sich des Multithreadings durch den Kernel. Da ihm Anwendungen – abgesehen vom CPU-Mode – ebenso vertrauen müssen wie dem Kernel, teilen sich beide den selben, virtuellen Adressraum. Das vereinfacht die Core-interne Objekt-Kommunikation erheblich. Die Schnittstelle zwischen Kernel und Unprivileged Mode besteht aus einer relativ kleinen Anzahl von Syscalls, welche die Adressierung bestimmter Kernelfunktionalitäten durch Hardware-Exceptions ermöglichen. Der überwiegende Teil der Syscalls ist Core vorbehalten. Alle anderen Syscalls unterliegen zusätzlichen Maßnahmen zum Schutz der Kernelintegrität: Es wird prinzipiell davon ausgegangen, dass sie missbraucht werden.

3.3.3 Implementierung der Core-Schnittstelle

Für die Behandlung von MMU-Fehlern bietet der Kernel das Konzept von Pager-Threads. Jedem Thread kann durch Core ein anderer Thread als Pager zugeordnet werden. Wirft der Thread daraufhin einen MMU-Fehler, pausiert ihn der Kernel und sendet einen Fehlerbericht an den Pager. Dies erfolgt über nachrichtenbasierte IPC, die vom Kernel für jegliche Thread-Thread-Kommunikation angeboten wird. Einziger Unterschied ist in diesem Fall, dass der Sender kein Userland-Thread ist. Der Pager muss den Fehlerverursacher also explizit reaktivieren. Das funktioniert, wenn er dessen Thread-Capability besitzt. Nun bietet Core durch seinen RM-Dienst jedoch – wie in Kapitel 3.2 beschrieben – die asynchrone Signalisierung von MMU-Fehlern. Der Kernel ist aber auf die synchrone Weiterleitung angewiesen, damit ein dynamisches Datenaufkommen im privilegierten Modus vermieden werden kann. Core trägt deshalb ausschließlich eigene Threads als Pager ein, welche nur der asynchronen Weiterleitung zum eigentlichen Pager dienen. Das Reaktivieren des Treibers nach emulierten MMIO-Zugriffen obliegt also der Emulationsumgebung. Die Thread-Capability dazu besitzt sie, da die Thread-Erzeugung von ihr abgehört werden kann.

Ausführungskontexte können über Syscalls nicht nur erzeugt, ausgeführt und pausiert, sondern auch gelesen und modifiziert werden. Da all dies aber Core vorbehalten ist, vermittelt er die Funktionalität über den CPU-Dienst. Dabei wird der Zugriff auf einen Kontext von dem Besitz der entsprechenden Thread-Capability abhängig gemacht. Für das mehrfache Lesen und Schreiben wird eine Kopie des gesamten Kontextes kommuniziert. Es ist Aufgabe des Clients, die Kohärenz der Kopie sicherzustellen, solange sie nötig ist. Für die Entwicklungsumgebung bedeutet das, dass der Kontext des Treibers zwischenzeitlich nicht zur Ausführung kommen, und von keinem anderen Prozess modifiziert werden darf. Tatsächlich ist die Thread-Capability bei der Erzeugung eines Threads nur seiner Vertrauensbasis an Prozessen – auch Trusted Computing Base, kurz TCB – bekannt. Auch alle zukünftigen Besitzer der Capability müssen irgendwann durch den Treiber oder seine TCB ins Vertrauen geschlossen worden sein. Es obliegt also diesen Prozessen – und somit auch der Emulationsumgebung – gegebenenfalls eine Synchronisierung vorzunehmen.

Nun kommen manuelle Kontextmodifikationen auf Anwendungsebene recht selten vor. Für diese Arbeit ist nur ein Fall von Bedeutung: Theoretisch lassen sich Emulationsumgebungen auch ineinander verschachteln. In dieser Situation kommt der Implementierung zugute, dass Kontextmodifikationen nur für die Behandlung von MMU-Fehlern gebraucht werden. Der MMU-Fehler blockiert den betrachteten Kontext bis zum Abschluss der Emulation, so dass er keine weiteren Fehler erzeugen kann. Signalisiert wird der Fehler immer nur der Emulationsumgebung, welche dem Treiber am nächsten ist und den betroffenen MMIO-Bereich

behandelt. Überschneidungen bei der Kontextmodifikation kann es also, allein durch das Modell dieser Arbeit nicht geben. Deshalb enthält es auch keine entsprechende Synchronisation.

3.4 Adaption von Init

Als Vorlage für die neue Emulationsumgebung diente GENODE's Init-Prozess. Die folgenden Kapitel sollen einen Eindruck davon vermitteln, warum diese Vorlage genutzt wurde und wie aus ihr die neue Emulationsumgebung entstand. Dazu wird noch einmal die Einführung in GENODE's Startablauf, aus Kapitel 3.3.1 aufgegriffen.

3.4.1 Konzept der Init-Adaption

Für die meisten Szenarien, in denen GENODE zum Einsatz kommt, wird über die Abstraktion des Core hinaus noch ein statische Menge an weiteren Prozessen benötigt. Die Prozesse stehen in der Regel in einer wohldefinierten Beziehung zueinander und benötigen – zumindest initial – ein bekanntes Spektrum an Ressourcen. Diese szenarienabhängige, statische Konfiguration möglichst flexibel und automatisiert umzusetzen, ist das Ziel des sogenannten Init-Prozesses. Folgerichtig ist er im Normalfall das erste und einzige direkte Kind von Core. Um seinem Zweck gerecht zu werden, ist Init in der Lage, eine gegebene XML-Datei zu parsen und einige einfach verständliche Tags zu interpretieren. Da dieser Mechanismus in der Emulationsumgebung eine wichtige Rolle einnimmt, soll er genauer betrachtet werden.

Über die XML-Schnittstelle ist der Nutzer, unter anderem in der Lage, Kindprozesse aus Binärdateien zu erstellen und zu starten. Dazu setzt er einen sogenannten Start-Tag. In diesem Start-Tag kann er auch angeben, welche Ressourcen Init dem Kind überweisen soll und welche Dienste das Kind anbieten wird. Außerdem kann er dem Kind darin eine eigene XML-Konfiguration überreichen. Je Start-Tag erzeugt Init nun ein sogenanntes Child-Objekt. Das Child bildet, solange das Kind läuft, die Schnittstelle zwischen Init, dem Start-Tag und dem neuen Prozess. Das Child ist ein wichtiger Baustein für die neue Emulationsumgebung. Neben der Prozesserzeugung nimmt Init auch eine Art Vermittlerrolle ein, wenn eines seiner Kinder einen Dienst erfragt. Für diese Situation gibt es ein System von Routing-Tags in der XML-Konfiguration. Mit ihnen lassen sich Dienste verschiedenen Routings zuordnen. Für jedes mögliche Routing hält Init deshalb eine Dienstekartei – den Service Pool. So gibt es einen Pool für die Dienste, deren Anfragen an den Elternprozess – also Core – weitergeleitet werden sollen, und einen für alle Anfragen, deren nächstes Ziel ein bestimmtes Kind von Init ist.

Der Aufgabenbereich der neuen Emulationsumgebung weist viel Ähnlichkeit mit dem von Init auf. Die Emulationsumgebung startet üblicherweise ebenfalls ein statisches Set an Prozessen, welche den software-seitigen Entwicklungsstand verkörpern. Darunter finden sich die Treiber der zu validierenden Hardware-Komponenten. Im Allgemeinen sollen diese Prozesse, ebenso wie im Endsystem Dienste anbieten und anfragen können, womit auch hier ein Algorithmus zur Dienstevermittlung benötigt wird. Anders als in Init, soll dieses Routing aber eine Art Filter vor die nativen Ressourcen legen. Der Filter soll emulierte Adressbereiche überlagern und entsprechende Anfragen einer speziellen Routine unterziehen. Größe und Lage dieser besonderen Adressbreiche, soll über eine leicht verständliche Schnittstelle statisch konfigurierbar sein. Wie sich im Laufe des Kapitels zeigen wird, lässt sich Inits XML-basiertes Routing sehr gut um die Semantik dieses Filters erweiten. Sobald nun eine Dienstanfrage an einen emulierten Bereich vorliegt, muss mitunter ein passender Emulator als Kindprozess gestartet werden. Obwohl diese Prozesserzeugung dynamisch erfolgt, lassen sich Inits Werkzeuge zur statischen Prozesserzeugung als Basis verwenden. Jeder dynamisch erzeugte Emulator eines Gerätetyps nutzt nämlich das selbe, im Voraus bekannte Binary, mit der gleichen statischen Konfiguration initialer Ressourcenanforderungen und Dienstangebote. Es liegt also nahe, das Prinzip von Childs und Start-Tags zu adaptieren, um je Gerät einen Prototyp der Emulatoren anzugeben.

Neben all diesen funktionalen Eigenschaften, die Init als Vorlage für die Emulationsumgebung attraktiv machen, sind auch die geringen Vorraussetzungen für den Betrieb von Init praktisch. Lediglich die grundlegenden Dienste, welche Core anbietet, werden in Anspruch genommen. Dadurch bleibt eine hohe Flexibilität gewährleistet, was den Einhängepunkt des Software-Tests im Prozessbaum angeht. Rückblickend fällt außerdem auf, dass die Emulationsumgebung Kindern gegenüber die selbe Schnittstelle wie Init aufweist. Beide bieten ihren Kindern keine eigenen Dienste im klassischen Sinn, sondern vernetzen laufende Prozesse nur untereinander. Da die Adaption den Init-Prozess somit transparent ersetzen kann, wurde sie schlicht Virtualizing Init, kurz VINIT genannt.

3.4.2 Konfiguration und Initialisierung von VINIT

Der erste Aspekt, der bei VINIT in Angriff genommen wurde, waren die zusätzlichen Anforderungen bezüglich der Konfigurierbarkeit. Dabei ergaben sich drei neue Aspekte für den XML-Dialekt. Die Grundlage bildet die Beschreibung eines zu emulierenden Adressbereichs, durch die Einführung des Resource-Tags. Jeder Resource-Tag besitzt Parameter die den Ressourcentyp – also MMIO oder IRQ – die Basisadresse und die Größe des Bereichs definieren. Weiterhin sollen Emulatorentypen beschrieben werden, die beliebig oft instanziiert werden können. Dazu wurde der Emulator-Tag eingeführt. Dieser ist abgesehen von seinem

Namen vollkommen identisch zum Start-Tag von Init. Er bietet also die gleichen Parameter und Subtags wie die herkömmliche Prozesserzeugung. Anders als der Start-Tag definiert er aber keinen unmittelbar zu startenden Kindprozess, sondern liefert nur das Muster für spätere Prozesserzeugungen. Der dritte und letzte Aspekt besteht darin, die anfangs definierten Adressbereiche ihren Emulatorentypen zuzuordnen. Da ein Emulatorentyp mehrere verschiedene Ressourcen zugleich emulieren kann, liegt es nahe, die entsprechenden Resource-Tags zu gruppieren. Die Gruppierung und Zuordnung übernimmt der sogenannte Emulated-Tag, der mehrere Resource-Tags umschließen darf. Er adressiert genau einen Emulator-Tag anhand des Prozessnamens. Dabei ist es möglich, einem Emulatorentyp mehrere Ressourcengruppen zuzuordnen, aber nicht umgekehrt. Das macht Sinn, wenn man mehrere Instanzen eines Geräts an unterschiedlichen Adressen emulieren möchte. Die Ressourcengruppen weisen dann üblicherweise die gleiche Struktur auf, betreffen aber disjunkte Adressbereiche. Spätestens an dieser Stelle wird deutlich, dass ein emulierter Adressbereich, emulatorintern nicht zwangsläufig die selbe Adresse besitzen muss, wie im globalen Adressraum der XML-Konfiguration. Um VINIT eine Übersetzung zu ermöglichen, wurde der Resource-Tag um eine zusätzliche Angabe erweitert – die emulatorlokale Adresse des Bereichs. Neben den Resource-, Emulator- und Emulated-Tags bestehen natürlich sämtliche Konfigurationsmöglichkeiten von Init weiter. Dadurch lässt sich die software-seitige Entwicklungsumgebung wie ein normales GENODE-Szenario beschreiben.

Der nächste Schritt bestand darin, die Initialisierung von Init anzupassen. Dazu gehört unter anderem ein Interpreter für die neuen XML-Tags. Ziel dieses initial durchlaufenden Interpreters ist es, eine statische Map der emulierten Bereiche zu erstellen. Diese Map ermöglicht es VINIT später, angeforderte Adressen effizient aufzulösen. Die durchgezogenen Pfeile in Abbildung 9 verdeutlichen, welche Datenstrukturen vom Interpreter erstellt werden und wie diese später, über die gestrichelt dargestellten Pointer, zum passenden Emulator-Tag führen. Jede emulierte Ressourcengruppe wird als sogenannter Emulation Context vermerkt. Jeder Emulation Context merkt sich seinerseits den Emulator-Tag, der für seine Adressen zum Einsatz kommen soll. Dann wird jedes Resource-Tag interpretiert. Kann ein Resource-Tag einem Emulation Context zugeordnet werden, so wird eine Emulated Region erzeugt, die sich den Context merkt und einem adressindizierten AVL-Baum angehört. Für jeden emulierbaren Ressourcentypen – also Interrupts und MMIO – existiert ein solcher AVL-Baum, der wie eine Maske über den entsprechenden Adressbereich gelegt wird.

Ist dies getan, kommt VINIT – ebenso wie Init – zur Zusammenstellung des Diensteangebots. Ziel dieses Schritts ist es, die Service Pools zu befüllen, die bereits im Zusammenhang mit dem Routing erwähnt wurden. In diesem Zug sollen

auch die emulierenden Dienste und die Monitoring-Dienste verfügbar gemacht werden. Anders, als ein herkömmlichet Service werden die neuen Dienste jedoch von VINIT selbst angeboten. Die Routingentscheidung besteht also nicht in einer Weiterleitung der Anfrage, sondern darin, selbst tätig zu werden. Für diese Entscheidung werden zwei weitere Service Pools, namens Spy-Services und Emulated Services erzeugt. Die Trennung der beiden Dienstarten hat den Grund, dass Monitoring-Dienste immer und vorrangig angewandt werden sollen, während emulierende Dienste – sozusagen gleichauf mit den normalen Diensten – nur bei Anfrage eines bestimmten Adressbereichs zum Einsatz kommen. Den neuen Pools werden dann sogleich die beiden Monitoring-Dienste – RM und CPU – sowie die beiden emulierenden Dienste – MMIO und IRQ – hinzugefügt.

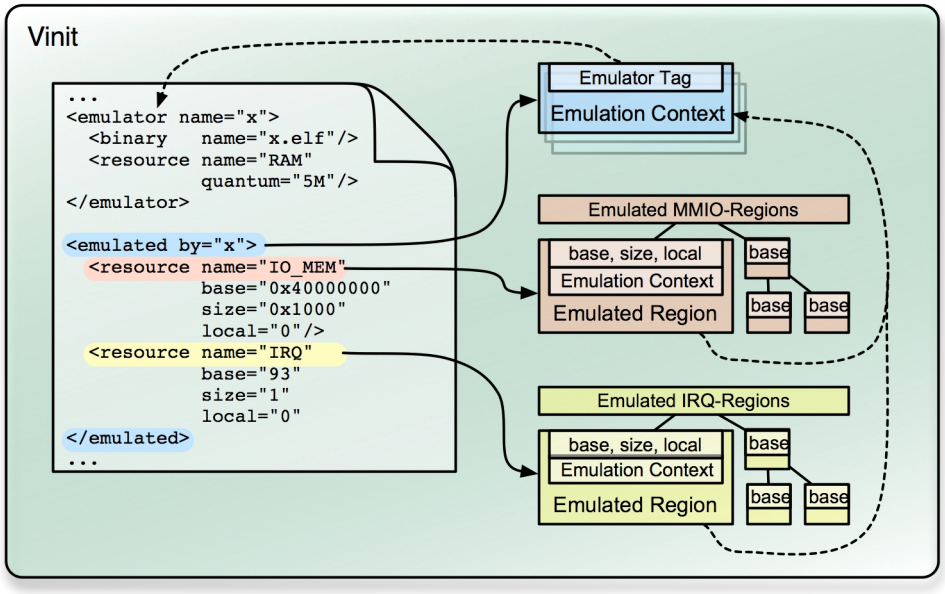


Abb. 9: Initiales Laden der Emulationseinstellungen in Vinit

An dieser Stelle soll ein kurzer Einblick in das Monitoring-Konzept unter GENODE erfolgen. In GENODE wird jede Verbindung zwischen einem Server und einem Client Server-seitig durch eine sogenannte Session repräsentiert. Sie implementiert die Schnittstelle des Dienstes und hält den Status dieser einen Verbindung. Das Monitoring wird in VINIT nun durch einen lokalen Dienst erledigt, welcher Client-seitig vorgibt, der erfragte Dienst zu sein. Zur Abarbeitung der eigentlichen Dienstfunktionalität, unterhält jede seiner Sessions aber wiederum eine Session zum tatsächlichen Server. Bei allen RPCs wird der eigentliche Server vom Monitor kontaktiert und dessen Ergebnis an den Client weitergeleitet. Bei

RPCs die des Monitorings bedürfen, wird das Ergebnis außerdem vom Monitor analysiert, bevor es weitergeleitet wird. Auf diese Weise überlagern die zwei Monitoring-Dienste in VINIT den nativen CPU- und RM-Dienst.

Das Monitoring des CPU-Dienstes ist in Abbildung 10 dargestellt. Es beschränkt sich auf die RPC-Methode `create_thread`, die neue Threads für einen Kindprozess erzeugt. Wurde der neue Thread erfolgreich vom Backend-Server erzeugt, vermerkt die Monitoring-Session einen sogenannten CPU-Client in einem AVL-Baum. Dieser eine AVL-Baum wird von allen Verbindungen des CPU-Monitors genutzt und dient somit als dienstweite Client-Datenbank. Sie ermöglicht es, jeden CPU-Client anhand der entsprechenden Thread-Capability ausfindig zu machen. Die CPU-Clients ihrerseits, merken sich die Monitoring-Session, über die ihr Thread erzeugt worden ist. So kann später, wenn der emulierende MMIO-Dienst am Zug ist, Zugriff auf die Ausführungskontexte der Threads genommen werden.

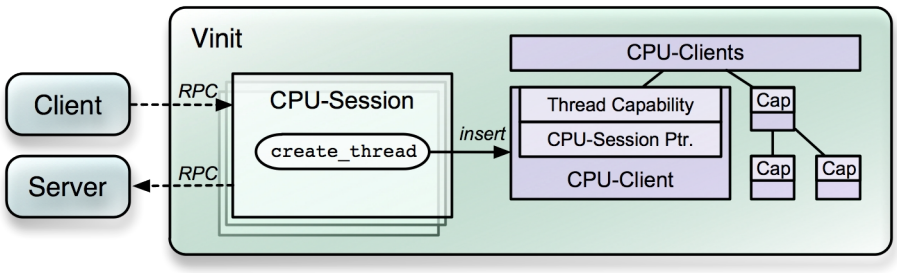


Abb. 10: Funktionsweise einer Session des CPU-Monitors

Anders als beim CPU-Monitor, müssen beim RM-Monitoring – wie Abbildung 11 zeigt – mehrere Funktionen überwacht werden. Wenn ein Kindprozess per `attach` ein Stück Speicher – Dataspace genannt – in seinen Adressraum einhängt, vermerkt sich die Monitoring-Session eine sogenannte Region. Sie kann später anhand der Adresse wieder aufgefunden werden, die der Speicher im Adressraum des Kindes besitzt. Dazu hält jede RM-Session einen AVL-Baum, der ihre Regions verwaltet. Die Region ihrerseits merkt sich die Capability des Dataspace. Dadurch lassen sich später, bei der Emulation, Client-lokale Adressen den global gültigen Capabilities zuordnen. Über diese Capabilities ist es VINIT dann möglich, jeden Speicherbereich des Clients auch lokal einzubinden. Ein Dataspace kann in GENODE jedoch, statt direkt auf Speicher zu verweisen, auch eine Indirektion zu einer weiteren RM-Session sein. Die Dataspaces dieser RM-Session verweisen dann auf den eigentlichen Speicher oder auf weitere RM-Sessions. Um also von einer Adresse des Clients tatsächlich auf den Speicher, und nicht eine weitere RM-Session zu gelangen, muss VINIT noch mehr in Erfahrung bringen, als nur die Dataspace Capability. Praktischerweise ist die RPC-Me-

thode `dataspace`, durch die Dataspace-Indirektionen einzig möglich werden, ebenfalls Teil des RM-Dienstes. Eine Monitoring-Session kann somit für jede neue Indirektion einen sogenannten Managed Dataspace vermerken. Nun prüft der AVL-Baum der Regions bei jedem gefundenen Dataspace einfach, ob ein Managed Dataspace dazu existiert. Ist dies der Fall, kann er die darin vermerkte RM-Session nach einem neuen, passenden Dataspace fragen. Dies geht so weiter bis ein Dataspace gefunden ist, zu dem kein Managed Dataspace existiert, der also tatsächlich Speicher adressiert.

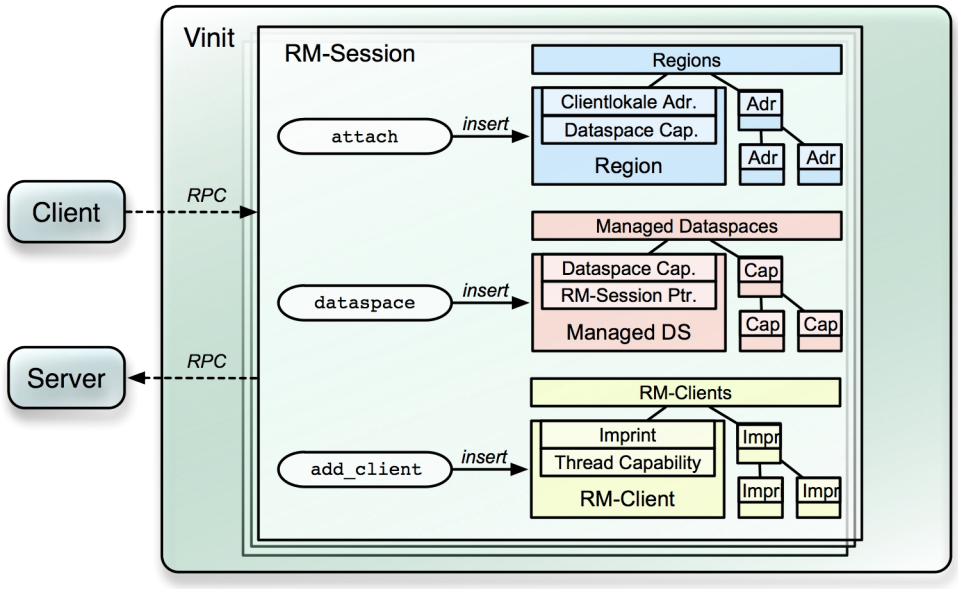


Abb. 11: Funktionsweise einer Session des RM-Monitors

Die dritte Methode, die in der RM-Session abgehört werden muss, ergibt sich dadurch, dass der verwaltete Adressraum zugleich von mehreren Threads genutzt werden kann. Threads müssen der RM-Session mittels `add_client` explizit gemeldet werden, wenn sie dem Adressraum beitreten wollen. Bei solchen Neulingen merkt sich der RM-Monitor die Thread-Capabilities in sogenannten RM-Clients. Jeder RM-Client erhält außerdem einen eindeutigen Namen vom Monitor – den sogenannten Imprint. Dann wird der eigentliche RM-Server gebeten, diesen Imprint bei jedem MMU-Fehler des Threads mitzuliefern. Auf diese Weise können MMU-Berichte später, bei der Emulation von MMIO, eindeutig zugeordnet werden. Der Imprint stellt in GENODE eine Erweiterung des üblichen RM-Diensts dar. Er ist jedoch – wie in Kapitel 2.2.3 beschrieben – unbedenklich für die Sicherheit des betrachteten Threads.

Die emulierenden Dienste sollen an dieser Stelle noch nicht vertieft werden, da ihre Funktionsweise im Kontext der Emulationsabläufe verständlicher ist. Ihre Initialisierung erfordert außerdem kein genaueres Verständnis. Nur bei der Initialisierung des emulierenden MMIO-Dienstes gibt es eine Besonderheit. Sessions dieses Dienstes müssen sich später beim RM-Monitor registrieren können, um von ihm die MMU-Fehler ihres Bereichs zugestellt zu bekommen. Deshalb merkt sich der Service für emuliertes MMIO schon jetzt den RM-Dienst von VINIT.

Damit steht der letzte Schritt der VINIT-Initialisierung an. Da der Emulationsfilter und die emulationsrelevanten Dienste bereit sind, kommt VINIT an den Punkt, an dem die gewünschten Kinder gestartet werden können. Dieser Schritt konnte von Init weitestgehend übernommen werden. Der einzige Unterschied ist, dass hier Emulated-Child- statt Child-Objekte kreiert werden. Diese erben zwar den Großteil ihrer Funktionalität von Child, merken sich aber zusätzlich die Service Pools der Monitore und der emulierenden Dienste. Außerdem ist es notwendig, dass sie bereits bei der Prozesserzeugung die neuen Dienste nutzen. Damit wird sicher gestellt, dass das Monitoring alle, auch die initialen Ressourcen der VINIT-Kinder erfasst. Nur so kann eine uneingeschränkte Emulation garantiert werden. Ansonsten folgt die Prozesserzeugung dem aus Init bekannten Schema, durch Interpretation der Start-Tags.

3.4.3 Session-Vermittlung über VINIT

Nun soll die Situation betrachtet werden, wenn ein Kind eine neue Session bei VINIT anfordert. Dazu übermittelt es dem Child in VINIT den Dienstnamen und dienstspezifische Parameter, wie MMIO-Adresse oder Interrupt-Nummer. Abbildung 12 zeigt solch eine Anfrage – ausgehend von einem Treiber – und die Reaktion von VINIT, die im folgenden genauer erklärt wird. Nachdem das Child in VINIT die Argumente geprüft hat, wird Emulated Child aktiv und wendet den Emulationsfilter an. Dieser ermittelt zuerst, ob es sich um einen emulierbaren Dienst handelt. Wenn es sich um den MMIO-Dienst handelt, wird der AVL-Baum emulierter Speicherbereiche herangezogen. Beim IRQ-Dienst hingegegen, wird der AVL-Baum emulierter Interrupts betrachtet. Wie bereits erwähnt, verwalten beide gleichermaßen Emulated Regions. Nur hat eine Interrupt-Region immer die Größe eins, da Interrupts immer einzeln vergeben werden. Für den weiteren Code ist der Ressourcentyp also unerheblich. Sollte sich in dem AVL-Baum nun eine Region befinden, die genau der angeforderten entspricht, so fährt der Filter fort. Gibt es hingegen eine partielle Übereinstimmung, so ist die Anfrage fehlerhaft. Ansonsten wird sie den normalen Diensten überlassen.

Gesetzt dem Fall, eine passende Region wurde gefunden, so muss das Emulated Child als nächstes herausfinden, ob es für die Region bereits einen Emulator besitzt. Zu diesem Zweck hält jedes Emulated Child einen AVL-Baum über Ob-

jekte vom Typ Emulator Child. Emulator Child ist eine Ableitungen von Emulated Child, die für die Prozesserzeugung, statt eines Start-Tags einen Emulator-Tag nutzt. Außerdem bekommen Emulated Childs ihre Ressourcenkontingente von VINIT spendiert. Die Kontingente eines Emulator Child hingegen, stammen aus dem Vorrat des Emulated Childs, von dem es gestartet wird. Ohne diese Vorkehrung würde ein Treiber zwar den üblichen Preis für den Dienst bezahlen, die zusätzlichen Kosten der Emulation müsste aber VINIT tragen. Eine weitere Besonderheit von Emulator Child liegt darin, dass es sich den Emulation Context merkt, für den es erzeugt wurde. Damit schließt sich der Kreis zur Session-Vermittlung wieder. Denn anhand des Emulation Context in der Emulated Region,

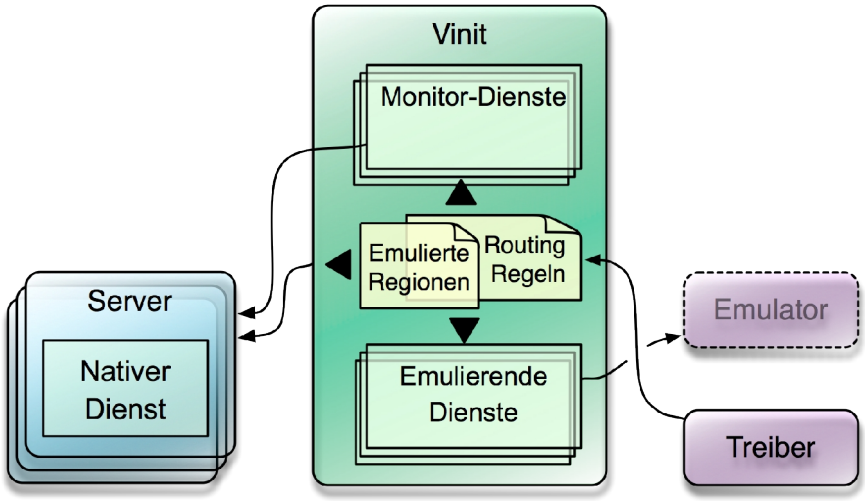


Abb. 12: Vermittlung von Sessions über Vinit

kann das Child nun seinen AVL-Baum nach einem passenden Emulator durchsuchen. Für den Fall, dass noch kein passendes Emulator Child existiert, hält der Emulation Context den richtigen Emulator-Tag parat. Mit ihm ist das Child in der Lage, eine neue Instanz des Emulators zu erzeugen. Diese wird dann sogleich im AVL-Baum des Emulated Child vermerkt. Zu beachten ist aber, dass die Dienste des Emulators betriebsbereit sein müssen, bevor er dem AVL-Baum beitritt. Deshalb wartet der Konstruktor des Emulator Child, bis der Prozess gestartet ist und seinen Dienst propagiert hat. Daraufhin stellt er eine Session zu dem Dienst her, bevor er den Emulator für Anfragen frei gibt. Alle weiteren Anweisungen an den Emulator werden seriell über diese eine Session geschickt. Das Emulated Child hat nun alles zusammen was es für eine Emulation bedarf.

Nun ist der Zeitpunkt gekommen an dem Child eine Routingstrategie für die Anfrage wählen würde, indem es die Service Pools nach einem passenden Service durchsucht. Doch vorher schaltet sich erneut Emulated Child ein. Bevor jedwede Routing-Strategie zum Einsatz kommt, stellt es fest, ob ein lokaler Monito-

ring-Service zu der Anfrage existiert. Ist dem so, wird zusätzlich ein nativer Service nach dem üblichen Verfahren gesucht. Ist auch dieser gefunden, so kann der Monitoring-Service eine Session erzeugen, die den normalen Service als Backend nutzt. Existiert kein Monitoring-Dienst, kommt der Emulated Service Pool zum Einsatz. Schlägt auch dies fehl, fällt Emulated Child zurück auf das übliche Routing-Verfahren von Child. Wird jedoch ein emulierender Service gefunden, so erzeugt dieser eine neue Session. Diese emulierende Session merkt sich die Emulator Session, die vom Konstruktor des Emulator Child erzeugt wurde. Erhält die emulierende Session nun Anweisungen des Treibers, übersetzt sie diese einfach auf die Emulator Session.

Handelt es sich um den MMIO-Service, wird der Treiber vorraussichtlich den Dataspace anfordern, um ihn bei sich einzuhängen. Der Dataspace, den die emulierende MMIO-Session im Gegenzug herausgibt, muss aber von besonderer Natur sein, sonst würde der Backend-Server – normalerweise Core – einfach selbst versuchen die MMU-Fehler aufzulösen, was natürlich vergebens ist. Stattdessen sollen Zugriffe auf dem eingehängten Dataspace immer wieder zu der emulierenden Session zurückführen. Deshalb erzeugt die emulierende MMIO-Session, statt eines normalen Dataspace, eine neue Session am RM-Service von VINIT. Eine solche Session hat eine besondere Eigenschaft: Sie kann – wie schon bei der Initialisierung von VINIT erwähnt – selbst als Dataspace benutzt werden. Mit dem Unterschied, dass für diesen Dataspace ein beliebiger Empfänger der MMU-Fehler eingestellt werden kann. Diese Weiterleitung erfolgt außerdem – wie in Kapitel 3.2 besprochen – asynchron. Um auf die MMU-Fehler zu warten, stellt die emulierende MMIO-Session nun einen dedizierten Thread bereit. Nachdem diese ganze Vorkehrung getroffen ist, kann die MMIO-Session die RM-Session, als Dataspace an den Treiber herausgeben.

Auch jede emulierende IRQ-Session startet bei ihrer Erzeugung einen Thread. Dieser besitzt jedoch eine andere Aufgabe. Während ein Dienst im Normalfall nur einen Thread bereithält, der die RPCs von allen Sessions seriell abarbeitet, reicht dieses Modell beim IRQ-Dienst nicht aus. Jeder IRQ-Session muss es möglich sein, auf ihren Interrupt zu blockieren, ohne dabei den gesamten Dienst lahmzulegen. Deshalb besitzt jede Session einen eigenen Thread für ihre RPCs.

Die folgenden Kapitel gehen darauf ein, wie die Kombination von Sessions, die VINIT vermittelt, die Emulation von Treiberanweisungen bewerkstelligt. Begleitend gibt Abbildung 13 einen Überblick über den Datenfluss emulierender Sessions, nativer Sessions und Monitor-Sessions.

3.4.4 Einsatz einer emulierenden MMIO-Session

Dieses Kapitel ist der Prozedur gewidmet, die ein Treiber anstößt, wenn er lesend oder schreibend auf den Dataspace einer emulierenden MMIO-Session zugreift. Der Zugriff führt unweigerlich zu einem MMU-Fehler, der vom RM-Server in Core abgefangen wird. Core versucht nun, realen Speicher zu dem Dataspace ausfindig zu machen. Dabei landet er jedoch zwangsläufig bei der leeren RM-Session der emulierenden MMIO-Session. Für ihn ist das eine Sackgasse. Deshalb speichert er den MMU-Bericht und meldet der MMIO-Session, dass ein Fehler aussteht. Dadurch wacht der Thread der MMIO-Session auf und fordert

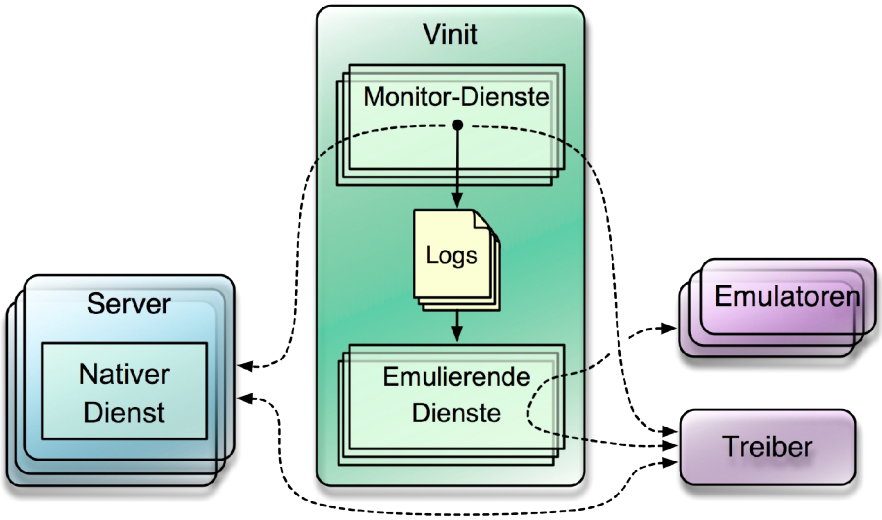


Abb. 13: Sessions eines Treibers unter Vinit

sogleich den Fehlerbericht über seine RM-Session an. Der klassische Fehlerbericht reicht jedoch nicht aus, um eine Emulation des Zugriffs durchzuführen. An dieser Stelle kommt der RM-Monitor zum Einsatz. Dieser Monitor hört nämlich nicht nur die Sessions der VINIT-Kinder ab, sondern auch die RM-Session, die die emulierende MMIO-Session nutzt. Durch das Abhören der Kinder besitzt er außerdem alle Informationen, die nötig sind, um einen klassischen MMU-Bericht für eine Emulation anzureichern. Er fängt also die Anfrage der emulierenden MMIO-Session ab, fragt Core nach dem Bericht und erweitert ihn, bevor er ihn an die Session herausgibt.

Im Zuge dessen muss der RM-Monitor als erstes in Erfahrung bringen, welcher RM-Client sich hinter dem Fehler verbirgt. Das gelingt ihm durch den eindeutigen Imprint, den Core im Bericht mitliefert, und den AVL-Baum in dem er sich alle RM-Clients gemerkt hat. Der RM-Client liefert dann die Thread-Capability des Fehlerverursachers. Damit lässt sich – über den AVL-Baum des CPU-Monitors – der passende CPU-Client ermitteln. Durch den CPU-Client wiederum,

kommt der RM-Monitor an die CPU-Session des Fehlerverursachers, und mit ihr schließlich an den Ausführungskontext. Dadurch ist er in der Lage, den treiberlokalen Befehlszeiger zu ermitteln. Doch der Befehlszeiger allein reicht nicht – der Monitor braucht den Befehl selbst. Deshalb fragt er seinen Region-AVL-Baum nach dem Dataspace, der sich hinter dem Befehlszeiger verbirgt. Nun braucht er den Dataspace nur noch über die RM-Session von VINIT einzuhängen, und der Befehlscode, der zum Fehler geführt hat, kann lokal ausgelesen werden. Der nächste Schritt besteht darin, den Befehlscode auf die Parameter zu übersetzen, die dem Emulator als Input dienen. Diese Aufgabe erledigt ein Befehlsdekoder – die einzige architekturspezifische Komponente von VINIT.

Die Schnittstelle des Dekoders ist im Grunde die selbe, wie schon in VDM. Sein Inneres wurde aber weitestgehend überarbeitet. Hier kommt nun eine template-basierte Lösung zum Einsatz. Dadurch wird der Aufbau des Dekoders nicht nur effizienter, sondern auch wiederverwendbarer und einfacher zu erweitern. Ist die Dekodierung erfolgreich, kann der erweiterte Fehlerbericht erstellt werden. In ihm werden die Ergebnisse des Dekoders und der klassische Bericht gespeichert. Sollte sich herausstellen, dass es sich um einen Schreibzugriff handelt, muss auch der zu schreibende Wert beigefügt werden. Laut den Vorraussetzungen, die an die Hardware gestellt wurden, muss sich dieser Wert in einem Register des Ausführungskontextes befinden. Welches Register das genau ist, lässt sich aus den Ergebnissen des Dekoders ablesen. Sollte es sich hingegen um einen Lesezugriff handeln, wird das Register erst nach der Emulation, beim Zurückschreiben benötigt. Der RM-Monitor kann den Registernamen also gleich im erweiterten Bericht vermerken, um sich später eine erneute Dekodierung zu sparen. Der erweiterte Fehlerbericht ist nun vollständig und kann dem Thread der emulierenden MMIO-Session übergeben werden.

Die MMIO-Session reagiert auf den Fehlerbericht, indem sie dem Emulator über die Emulator Session Anweisungen erteilt. Für MMIO existieren zwei Anweisungen namens `write_mmio` und `read_mmio`. Mit ihnen kann dem Emulator, unabhängig von der Plattform mitgeteilt werden, was der Treiber auf seinem MMIO beabsichtigt. Welche der beiden Anweisungen benötigt wird, verrät der Fehlerbericht des RM-Monitors. Als Argumente liefert der Bericht außerdem das Format und die Adresse des Zugriffs, sowie, wenn nötig, den Schreibwert. Die Adresse des Berichts muss die MMIO-Session aber noch umrechnen. Sie entspricht – wie schon im klassischen Bericht – nur dem Offset innerhalb des Dataspace. Die MMIO-Session addiert deshalb noch die emulatorlokale Adresse des Speichers, die sie sich aus ihrer Emulated Region gemerkt hat. Während `write_mmio` nur eine Art Hinweis an den Emulator ist, den Zugriff zur Kenntnis zu nehmen, wird von `read_mmio` ein Rückgabewert erwartet.

Hat der Emulator die Anweisung verarbeitet, bleibt nur noch der ungelöste MMU-Fehler des Treibers. Um den Treiber wieder lauffähig zu machen, greift die MMIO-Session ein zweites mal auf die Fähigkeiten des RM-Monitors zurück. Dieser kann während der Emulation schon ganz andere MMU-Fehler verarbeitet haben. Um einen alten Fehler dennoch wieder aufgreifen zu können, speichert der Monitor in den RM-Clients vorsorglich jeden Fehlerbericht und den dazugehörenden Ausführungskontext. Diese Notizen werden erst dann gelöscht, wenn der Fehler vollständig gelöst ist. Anhand des Imprints, den ihm nun die MMIO-Session liefert, sucht er den RM-Client erneut heraus. Auch den gelesenen Wert hat er – wenn nötig – von der MMIO-Session empfangen und schreibt ihn zurück in den Ausführungskontext. Nun wird nur noch der Befehlszeiger des Treibers um eine Befehlsbreite erhöht, und der so modifizierte Ausführungskontext kann zurückgeschrieben werden. Dies geschieht wieder über die CPU-Session des Treibers, die – wie schon einmal – über den RM-Client ermittelt wird. Abschließend weist der Monitor die CPU-Session noch an, den Thread des Treibers aufzuwecken, wodurch der neue Ausführungskontext zur Anwendung kommt. Dann wartet der Monitor wieder auf MMU-Fehler, die es zu emulieren gilt. Der Treiber hingegen, findet sich nach dem Erwachen so vor, als wäre sein letzter Zugriff tatsächlich ausgeführt worden.

3.4.5 Einsatz einer emulierenden IRQ-Session

Verglichen mit der emulierenden MMIO-Session gestaltet sich die Emulation einer IRQ-Session einfacher. Für jeden emulierten Interrupt, den ein Treiber in Besitz nimmt, wird eine eigene Session und ein zugehöriger Thread in VINIT erzeugt. Über die Session kann der Treiber vermelden, dass er auf dem Interrupt blockieren möchte. Im Zuge dieses RPCs gibt er den Kontrollfluss an den Thread der emulierenden IRQ-Session ab. Dieser nimmt daraufhin die Kommunikation mit dem Emulator auf. Die Emulator Session kennt nur eine Anweisung für die Interrupt-Emulation: Sie heißt `irq_handler` und übermittelt, neben dem emulatorlokalen Interrupt-Namen, einen Signal Context. Den emulatorlokalen Interrupt-Namen entnimmt die IRQ-Session ihrer Emulated Region. Der Signal Context hingegen ist ihr eigenes Produkt und stellt ein asynchrones Ereignis dar. Die Anweisung signalisiert dem Emulator also, dass er ab sofort bei einer Statusänderung des Interrupts den Signal Context auslösen soll. Als Rückgabewert von `irq_handler` liefert der Emulator den aktuellen Status des Interrupts. Sollte der Interrupt zu diesem Zeitpunkt noch nicht aktiv sein, blockiert der Thread der IRQ-Session auf den Signal Context. Ansonsten wird dieser Schritt ausgelassen. Abschließend ruft die Session noch einmal `irq_handler` auf. Dabei sendet sie dem Emulator den selben Interrupt-Namen, aber einen invaliden Signal Context. Das signalisiert ihm, dass der Interrupt-Status nun nicht mehr von Interesse ist,

oder leitet die asynchronen Ereignisse zumindest ins Leere. Die Metho-de `irq_handler` ist somit – je nach Validität des Signal Context – vergleichbar mit dem Demaskieren und Maskieren eines realen Interrupts am Interrupt Controller. Wenn der Thread der IRQ-Session nun den RPC beendet, deblockiert er den Treiber zugleich. Damit weiß der Treiber, dass der Interrupt ausgelöst wurde.

3.5 Emulatoren-Tools

Die Tools aus den Kapiteln 2.3 und 2.4, die das Codesign oberhalb der VINIT erleichtern sollen, können in GENODE eigenständig implementiert werden. Der Entwickler kann sie dann, je nach Gerätetyp und Szenario, in unterschiedlicher Kombination und Konfiguration einsetzen. Begleitend zu den folgenden Implementierungsdetails, gibt Abbildung 14 einen Überblick über das Zusammenspiel von Tools, Bibliotheken und der HDL-Simulation innerhalb eines Emulators.

3.5.1 Integration von HDL-Code

Wie in Kapitel 2.3.2 beschrieben, kommt bei der Erzeugung von Emulatoren eine angepasste Version des Third-Party-Programms VERILATOR zum Einsatz. Die Vorbereitung dieses Programms soll möglichst automatisiert im Voraus geschehen. Dazu wurde eine Besonderheit der GENODE-Toolchain genutzt. Sie bie-

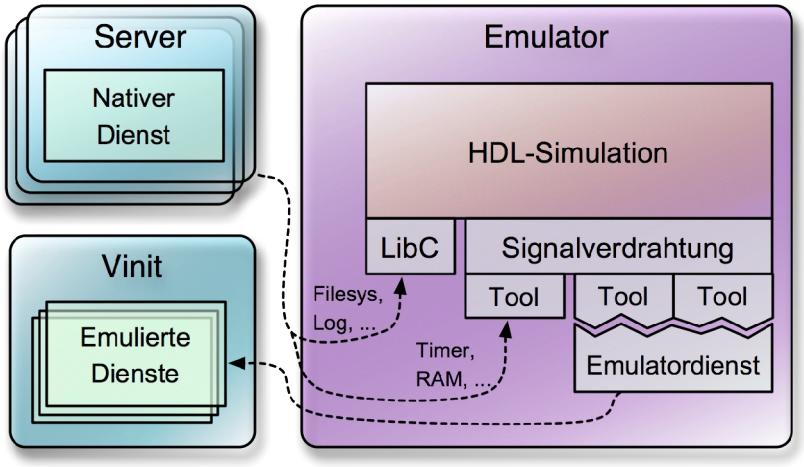


Abb. 14: Aufbau und Sessions eines Emulators

tet ein MAKE-Framework namens PREPARE an. Damit kann ein Anwender aus einer Zahl von Fremdpaketen wählen, welche er für seine Szenarien benötigt. Die gewählten Pakete arbeitet das Framework dann – durch downloaden, modifizieren und kompilieren – automatisch in die GENODE-Quellen ein. Nun soll darauf eingegangen werden, welche Modifikationen PREPARE am VERILATOR-Code vor-

nimmt. Die VERILATOR-Toolchain, die später die Emulatoren erzeugt, basiert – ebenso wie die von GENODE – auf MAKE. Dadurch konnte ihre Integration elegant gelöst werden. Zuerst stellt PREPARE sicher, dass die VERILATOR-Toolchain entscheidende Umgebungsvariablen, statt sie selbst zu definieren, als gegeben ansieht. Das heißt, die GENODE-Toolchain muss die Variablen definieren, ehe sie die VERILATOR-Toolchain invokiert. Viele dieser Variablen, wie den Crosscompiler-Präfix oder die Include-Pfade, benutzt die GENODE-Toolchain in gleicher Weise auch für ihre Zwecke. Sie sind also bereits passend gesetzt. Durch die Übernahme der Variablen, müssen jedoch all die Labels der VERILATOR-Toolchain umbenannt werden, welche der Redefinition von außen nicht zum Opfer fallen sollen.

Den Link-Prozess zu kappen, damit keine unnützen Abhängigkeiten und Abläufe übernommen werden, stellte sich als trickreich heraus, da VERILATOR ihn nicht direkt anstößt. Die Arbeit von VERILATOR teilt sich tatsächlich in zwei Phasen auf. Die erste ist generisch und erzeugt alle Objekt-Dateien aus den HDL-Quellen, sowie ein quellen-spezifisches Makefile. Dieses Makefile ist nur temporär für den aktuellen Übersetzungsvorgang vorhanden. Es kommt in der zweiten Phase an die Reihe und steuert unter anderem den Link-Prozess. Um den Link-Prozess also zu unterbinden, muss das temporäre Makefile während der Emulatorerzeugung modifiziert werden. Diese On-The-Fly-Modifikation übernimmt die GENODE-Toolchain, während sie die Toolchain von VERILATOR steuert. Die Test Bench, die VERILATOR dennoch zumindest kompilieren will, wird durch eine leere Quelldatei vertreten, die nach dem Vorgang gelöscht wird.

Ist VERILATOR nun modifiziert und kompiliert, kann ihn die GENODE-Toolchain zum Einsatz bringen. Anders als native GENODE-Programme, nutzen Emulatoren, die mit VERILATOR erzeugt wurden, externe Bibliotheken – darunter vor allem die LIBC. Damit werden HDL-fremde Funktionen, wie das Initialisieren von Speichereinheiten aus einer Binärdatei ermöglicht. Auch andere Portierungen für GENODE nutzen die LIBC. Es hat sich herausgestellt, dass diese Anwendungen meist nur wenige Funktionen der LIBC benötigen, welche in GENODE individuell, je nach Anwendung implementiert sein sollten. Deshalb bietet die GENODE-Variante der LIBC nur die LIBC-Schnittstelle, die erst einmal jede Funktion mit einem Fehler abfängt. Dahinter werden dann – je nach Bedarf – sogenannte Plugin-Bibliotheken geschaltet. Sie greifen selektiv Funktionsstümpfe der GENODE-LIBC auf und koppeln sie an eine echte Implementierung, die den Erwartungen der Anwendung entspricht. Dieser Mechanismus ermöglicht zum Beispiel die Dateizugriffe in Kapitel 4, obwohl die komplexe Anbindung eines Massenspeichers für das verwendete Board noch garnicht implementiert war. Das Plugin leitet die Zugriffe einfach auf ein RAM-Dateisystem um. Das RAM-Dateisystem seinerseits,

bezieht die fragliche Datei wiederum aus dem GENODE-Boot-Image. Abhängigkeiten der VERILOG-Simulation, wie die LIBC und ihre Plugins, werden in der Bibliothek `verilator_env` zusammengefasst. [41, S. 164ff][42]

Die Emulatorenerzeugung soll nun durch die Abhängigkeit eines Programms zu `verilator_env` ganz automatisch ausgelöst werden. Die Bibliothek selbst, kann diese Aufgabe aber nicht übernehmen, da sie an die Emulatoren nur gelinkt wird. Deshalb kommt eine weitere Funktionalität der GENODE-Toolchain zum Einsatz. Zu jeder Bibliothek kann eine sogenannte Import-Datei existieren. Sie enthält Toolchain-Anweisungen, die bei jedem Programm zum Einsatz kommen, das die Bibliothek nutzt. So gelangt die HDL-Erkennung und -Übersetzung automatisch in den Bauprozess potentieller Emulatoren. Findet sie dort keine passenden VERILOG-Quellen, bleibt die Erweiterung einfach inaktiv. Bei einem Treffer hingegen, blockiert sie sofort den Bauprozess aller anderen Quellen, damit das VERILOG-Design als erstes übersetzt werden kann. Für die Übersetzung erzeugt der Import-Code nun eine VERILATOR-Umgebung – dem Design entsprechend – und invokiert schließlich die VERILATOR-Toolchain. Danach liegen die Objektdateien der VERILOG-Simulation, sowie ein Header mit der Schnittstelle in einem temporären Ordner. Die Objektdateien werden vom Import-Code zur Linker-Liste, und der Header zu den Include-Pfaden des Emulators hinzugefügt. Damit ist der Import-Code fertig und deblockiert die anderen Bauvorhaben wieder. Diesen anderen Bauvorhaben – unter denen sich auch der Glue-Code zur Emulationsumgebung befinden muss – steht somit der fertige Simulator zur Verfügung.

3.5.2 Auswahl eines Bussystems

Um das Modell zur vereinfachten Emulatorenanbindung aus Kapitel 2.4.1 vorzuführen, musste ein geeignetes Bussystem gewählt werden. Bei der Wahl lag das Augenmerk auf mehreren Eigenschaften. Das Protokoll des Systems sollte möglichst erprobt und weithin verbreitet sein, wenn es um die Kommunikation zwischen CPU und Onboard-Peripherie geht. Gleichzeitig sollte es – zugunsten der Veranschaulichung – nicht allzu komplex sein. Im Hinblick auf Kapitel 4, sollten außerdem möglichst viele VERILOG-Designs aus Third-Party-Projekten existieren, die kompatibel zum gewählten Bus sind.

Die Wahl ist letztendlich auf den WISHBONE-Bus gefallen. Dieser Bus ist ein de-facto-Standard für Open-Source Hardware, der mittlerweile von der OPENCORES-Organisation verwaltet wird. Der Zweck dieser Organisation ist, nach eigenen Angaben die Entwicklung und Verbreitung von HDL-Designs als Open-Source [18]. Diesem Umstand ist zu verdanken, dass OPENCORES selbst ein großes Repertoire an Designs führt, die zu WISBONE kompatibel sind. Darunter finden sich nicht wenige, die in VERILOG geschrieben sind und auf Hardware-Emulatoren oder als synthetisierter ASIC getestet wurden. Da zudem – dank

Open-Source-Lizenz – sowohl die Busspezifikation als auch die erwähnten Designs frei zugänglich sind, bietet WISHBONE äußerlich die besten Voraussetzungen für die Analyse der vorgestellten Entwicklungsumgebung.

3.5.3 HDL-Anbindung als Bus-Slave

Zum Verständnis der WISHBONE-Tools folgt nun ein kurzer Einblick in das Busprotokoll anhand der Spezifikation [17]. In der Spezifikation wird der WISHBONE-Bus rein logisch beschrieben. Dadurch ist sichergestellt, dass die Bussimulation unabhängig von den Implementierungsumständen des Bussystems ist. Das heißt die konkrete Fertigungstechnologie – wie FPGA oder ASIC – sowie zum Einsatz kommende Synthese-Tools und Taktungen, müssen bei der Software-Variante nicht berücksichtigt werden. Auch der Umstand, dass der WISHBONE-Bus bei Transaktionen grundsätzlich ein synchrones Handshake-Protokoll einsetzt ist vorteilhaft für diese Arbeit. Der Einsatz komplexer Synchronisierungsmechanismen, zum zwangsläufig synchron arbeitenden Master hin, entfällt dadurch. Ohnehin ist anzunehmen, dass asynchrone Aspekte eines Designs über Interrupts gelöst werden können. Die Busspezifikation erlaubt desweiteren beliebig viele Master- und Slave-Teilnehmer, die sich durch verschiedene Arbitrierungsmethoden den Bus teilen. Dieser Mechanismus braucht bei der Software-Variante jedoch nicht beachtet werden. Verschiedene Transaktionen an einem Emulator werden bereits durch dessen RPC-Server serialisiert. Mehrere Verbindungen mit disjunkten Endpunkten hingegen, werden auch von verschiedenen Threads der Emulationsumgebung gehandhabt. Die Zeiteinteilung des virtuellen Bus-Backends, wird somit bereits durch den CPU-Scheduler erledigt. Der WISHBONE-Bus kann auch verschiedene Topologien zwischen den Teilnehmern umsetzten. Da die Busvergabe jedoch transparent für die Endpunkte ist, hat diese Entscheidung keinen Einfluss auf die Simulation. Mit den dedizierten Threads der Emulationsumgebung, entspricht das Konzept wohl am ehesten einerm Crossbar Switch, bei dem aber auch Verbindungen verschiedener Master-Prozesse zum selben Gerät hin nebenläufig sind.

Bei den Zugriffsmethoden selbst, bietet der WISHBONE-Bus ebenfalls verschiedene Modi. Die einfachste Variante ist das Read-Write-Protokoll für einzelne Datentransfers. Daneben gibt es Modi, die größere Datenmengen durch Zugriffsverkettungen optimieren. Die beispielhafte Implementierung der Tools soll jedoch einfach gehalten werden und vorerst nur die CPU-Peripherie-Kommunikation berücksichtigen. Diese kann, trotz Blockinstruktionen, laut ARMV7-Spezifikation [13, S. 151f (A3-45)] immer auf Einzelzugriffe übersetzt werden. Die Übersetzung solcher Blockinstruktionen, wird gegebenenfalls von der Emulationsumgebung vorgenommen. Gleiches gilt für die Zugriffe die breiter als 32 Bit sind [13, S. 33 (A2-3)], weshalb auch die Bitbreitenmodulierung des WISHBONE-

Bus nicht umgesetzt werden muss. Gut integrieren ließ sich hingegen der Rückgabewert der Transfer-Terminierung. Die möglichen Codes – `Normal`, `Retry` und `Error`, können direkt von den Tools interpretiert werden. Sie schließen dann den Transfer entsprechend ab, führen ihn erneut aus, oder halten sich selbst – und damit auch den Treiber – nach einer Fehlerausgabe an. Das beschriebene Verhalten wird nach dem Modell aus Kapitel 2.4.1 als Klasse umgesetzt. Ihre Objekte werden desweiteren einfach WISHBONE-Slaves genannt.

3.5.4 Emulatoren als Kommunikations-Master

Das Konzept der Master-Schnittstelle lässt sich im beschriebenenen Framework, durch GENODE's Konzept der Speicherdienste gut umsetzen. Generell erfolgt die Einbindung des Speichers am Emulator ebenso, wie an anderen Prozessen. Die explizite Adressierung von DMA-Speicher wird am einfachsten über den MMIO-Dienst erreicht. Da es sich bei dem Speicher aber nicht wirklich um MMIO handelt, den Core kennt, würde ein zwischengeschalteter MMIO-Monitor der Emulationsumgebung die Übersetzung auf Arbeitsspeicher vornehmen. Hier setzt – unter Zuhilfenahme der anderen Monitoring-Dienste – das Konzept aus Kapitel 2.4.4 an. Allerdings würde die Analyse der damit verbundenen Szenarien den Rahmen dieser Arbeit sprengen. Deshalb wurde die Anbindung einer Master-Schnittstelle vorerst nicht implementiert.

4 Analyse

Ein wichtiges Ziel dieser Arbeit ist es, die Integration von HDL-Code so einfach wie möglich zu gestalten. Um diesen Aspekt zu prüfen, soll zu Beginn der Analyse die Bedienung der Entwicklungsumgebung veranschaulicht werden, indem ein realer Einsatz durchgespielt wird. Diese Demonstration hilft auch, die Tests und Auswertungen in den folgenden Kapiteln nachzuvollziehen.

4.1 Bedienbarkeit

Zur Veranschaulichung soll ein Design eingebunden werden, dass die grundlegenden Anforderungen digitaler Peripherie abdeckt. Gut geeignet ist zum Beispiel ein Timer-Modul. Ein typischer Timer bietet MMIO zur Steuerung an, benutzt Interrupt-Signale und benötigt einen Taktgeber für zeitliche Aspekte. Ein passendes VERILOG-Design, namens PTC, war schnell gefunden. Es stammt von Damjan Lampret und beschreibt einen 32-Bit Timer, der in verschiedenen Modi, unter anderem als Counter und Impulsgeber genutzt werden kann [19]. Zur Steuerung der internen Funktionalität erlaubt das Modul über einen WISHBONE-Slave Zugriff auf vier verschieden breite Register. Zugriffe anderer Breite erzeugen einen Bus-Error. Der interne Zähler und der WISHBONE-Slave benutzen das selbe Taktsignal. Bestimmte interne Ereignisse signalisiert der Timer über einen Interrupt-Ausgang. Das Design des Timers ist auf mehrere VERILOG-Module in unterschiedlichen Quelldateien verteilt.

```
1 TARGET  = ptc_emulator
2 SRC_VLG = ptc.v
3 SRC_CC  = wiring.cc
4 LIBS    = verilator_env
```

Abb. 15: Makefile des PTC-Emulators

Der erste Schritt besteht darin, den Emulator zu erzeugen. Dazu werden alle VERILOG-Quellen des Designs unverändert in einen neuen Ordner kopiert. Zusätzlich wird in diesem Ordner ein Makefile für das Emulatorprogramm angelegt. Abbildung 15 zeigt den Inhalt dieser Datei. Dort wird angegeben dass die Datei `ptc.v` das Top-Modul des Designs enthält. Außerdem wird die Bibliothek `verilator_env` notiert. Dadurch wird beim Bauen des Emulators die

VERILOG-Übersetzung aktiv. Zu den Quelldateien gesellt sich dann noch eine C++-Datei – hier `wiring.cc` – die der Entwickler selbst schreiben muss. Sie erledigt die Verdrahtung von Geräteschnittstelle und Emulatorschnittstelle. Abbildung 16 zeigt den entscheidenden Teil von `wiring.cc`. Zuerst wird ein neues Interrupt-Array erstellt, das alle Interrupts des Geräts enthalten soll. Im Fall des Timers, enthält das Array – hier `irq` genannt – nur einen Interrupt. Nun wird ein Interrupt-Taktgeber namens `irq_clock` erzeugt. Ihm werden mehrere Werte übergeben: Das Taktsignal `clk_i`, die Taktflanke – hier "1" für die ansteigende Flanke, eine Frequenz von einhundert Takten je Millisekunde, ein Update-Intervall von zehn Millisekunden, das Lock für die Synchronisation zum Bus-Interface und abschließend das Interrupt-Array. Nun kann das Interrupt-Array dem Interrupt-Listener `irq_listener` übergeben werden. Als nächstes wird eine In-

```
 1 static Lock lock;
 2 static Irq irq(&ptc.irq_o);
 3 static Irq_clock clk(&ptc.clk_i, 1, 100, 10, &lock, &irq);
 4 static Irq_listener irq_listener(&irq);
 5
 6 struct Raw_wishbone_slave
 7 {
 8     void cycle() { return clk.cycle(); }
 9
11     uint8_t & rst_i() { return ptc.wb_rst_i; }
12     uint8_t & cyc_i() { return ptc.wb_cyc_i; }
13     uint8_t &  we_i() { return ptc.wb_we_i;  }
14     uint8_t & stb_i() { return ptc.wb_stb_i; }
15     uint8_t & ack_o() { return ptc.wb_ack_o; }
16     uint8_t & err_o() { return ptc.wb_err_o; }
17     uint8_t & rty_o() { static uint8_t dummy = 0; return dummy; }
18
19     void sel_i(uint8_t const    v) { ptc.wb_sel_i = v; }
20     void adr_i(uint32_t const   v) { ptc.wb_adr_i = (uint16_t)v; }
21     void dat_i(uint32_t const   v) { ptc.wb_dat_i = v; }
22     void dat_o(uint32_t * const v) { *v = ptc.wb_dat_o; }
23 };
24 static Sync_wishbone_slave<Raw_wishbone_slave, 10> wbs(&lock);
```

Abb. 16: Verdrahtung von Simulation und Emulator-Tools in `wiring.cc`

stanz des WISHBONE-Slave benötigt. Der Entwickler muss dem entsprechenden Tool mitteilen, welche VERILOG-Signale die WISHBONE-Signale darstellen und welche Daten- bzw. Adressbreite zum Einsatz kommt. Deshalb erstellt er das Backend `Raw_wishbone_slave`, das die passenden Akzessoren definiert und dem Tool in der letzten Zeile übergeben wird. Da das Taktsignal in `cycle()` das selbe ist, wie das des Interrupt-Taktgebers, wird `Sync_wishbone_slave`, die synchronisierte Variante der Busanbindung genutzt. Außerdem wird die Anbindung auf ein Transaktions-Timeout von zehn Millisekunden eingestellt.

Nun wird die GENODE-Toolchain einen funktionstüchtigen PTC-Emulator erzeugen. Der zweite Schritt besteht darin, VINIT mitzuteilen, dass sie den Emulator auch verfügbar machen soll. Dazu muss sie den neuen Typus von Emulator erst-

einmal als solchen kennen. Zu diesem Zweck wird der XML-Konfiguration ein Emulator-Tag hinzugefügt. Darin wird der Name des PTC-Binary vermerkt und wieviel Arbeitsspeicher jeder PTC-Emulator erhalten soll. Daraufhin kann der Entwickler beliebig viele Emulated-Tags beilegen die den Emulator nutzen. Jedes dieser Tags sollte einen MMIO-Bereich für die vier Register und einen Interrupt enthalten. Die treiberlokalen Adressen der entsprechenden Resource-Tags, können beliebig eingestellt werden, solange es keine Überschneidungen gibt. Die emulatorlokalen Adressen hingegen, müssen so eingestellt werden, dass sie den Erwartungen des PTC-Emulators entsprechen. Der PTC-Interrupt erhält also immer die Adresse "0", da er das einzige Element des Interrupt-Arrays ist. Abbildung 17 fasst die Einstellungen zusammen, die der VINIT-Konfiguration für die Emulation hinzugefügt wurden. Nun kann jeder Kindprozess von VINIT den PTC-Timer an Adresse `0x70000000` nutzen.

```
1 <emulator name="ptc_emulator">
2   <resource name="RAM" quantum="5M"/>
3 </emulator>
4
5 <emulated by="ptc_emulator">
6   <resource name="IO_MEM" base="0x70000000" size="0x1000" local="0x0"/>
7   <resource name="IRQ"    base="100"        size="1"      local="0"/>
8 </emulated>
```

Abb. 17: Die Bereitstellung des PTC-Emulators in Vinits XML-Konfiguration

Für ein vollständiges Codesign-Szenario darf der Einblick in einen beispielhaften Treiber nicht fehlen. Nach typischer GENODE-Manier wird angenommen, dass der Treiber die Fähigkeiten des PTC-Moduls als Server des sogenannten Timer-Dienstes feilbietet. Dieser Dienst stellt eine generische RPC-Schnittstelle zur Verfügung, über die sowohl einmalige, als auch periodische Zeitmessungen vorgenommen werden können [43]. Der Timer-Dienst ermöglicht es zudem, dass mehrere Clients gleichzeitig den selben Timer nutzen. Auch der PTC-Emulator selbst nimmt für seinen Taktgeber den Timer-Dienst eines anderen Treibers in Anspruch. Alle Timer-Treiber nutzen für ihren Dienst eine generische Timer-Server-Klasse, die wiederum eine Backend-Klasse namens `Platform_timer` verwendet, um die unmittelbare Steuerung des Timer-Geräts vorzunehmen. Jeder Treiber implementiert diese Backend-Klasse individuell entsprechend des Geräts das er nutzt. Abbildung 18 zeigt einen Ausschnitt der `Platform_timer`-Implementierung, wie sie im PTC-Treiber vorgenommen wird. Die wichtigsten Bestandteile des Backends sind die beiden Sessions `irq_session` und `mmio_session`. Sobald sie im Backend-Konstruktor in Zeile 11 und 12 initialisiert werden, senden sie jeweils eine Session-Anfrage an VINIT. VINIT erkennt, dass die mitgelieferten Adressen bezüglich Interrupt und MMIO-Bereich zum emulierten PTC-Modul gehören, startet einen Emulator und liefert dem Treiber zwei emulierte Sessions. Der Treiber lässt sich dann per RPC den Dataspace der

MMIO-Session zusenden, damit er ihn in Zeile 14 über seine RM-Session in den lokalen Adressraum einhängen kann. Nun kann er den MMIO-Bereich in der Methode `stop_and_sample` wie gewöhnlichen Speicher verwenden. Auf dem PTC-Interrupt blockiert er – wie in Zeile 19 zu sehen – über einen RPC der IRQ-Session.

```
1  class Platform_timer {
2
3    Genode::Irq_connection              irq_session;
4    Genode::Io_mem_connection           mmio_session;
5    Genode::Io_mem_dataspace_capability mmio_ds;
6    uint32_t volatile *                 mmio;
7
8    protected:
9
10     Platform_timer() :
11       irq_session(100),
12       mmio_session(0x71000000, 0x1000),
13       mmio_ds(mmio_session.dataspace()),
14       mmio(Genode::env()->rm_session()->attach(mmio_ds)) { }
15
16     uint32_t stop_and_sample() { mmio[3] = 0;      // PTC stoppen
17                                  return mmio[0]; } // PTC-Counter lesen
18
19     void wait_for_timeout() { irq_session.wait_for_irq(); }
20
21     ...
```

Abb. 18: Das emulierte Backend des Timer-Session-Servers (C++)

Angenommen, die Zielplattform bietet auch ein reales PTC-Modul, das – nur als Beispiel – den MMIO-Bereich bei `0x6ffff000` und den Interrupt "99" nutzt, so müsste man lediglich die beiden Adressen in Zeile 11 und 12 ändern und der gesamte Software-Test würde unverändert auf einem realen, statt einem emulierten Timer laufen. Abbildung 19 zeigt abschließend noch einmal alle Prozesse des PTC-Szenarios und wie sie miteinander interagieren.

4.2 Stabilität

Die vorgestellte Implementierung muss auch bei erheblicher Belastung und vorsätzlichem Fehlverhalten von Treibern und Emulatoren stabil laufen. Diese Grenzfälle auszuloten, ist Aufgabe der folgenden Kapitel.

4.2.1 Lasttests

Für die Lasttests wurde ein Szenario geschaffen, das die Emulation möglichst breitgefächert ausschöpft. Als Basis des Emulators, wurde jedoch ein zweckgebundenes, einfaches Design gewählt. Der Grund dafür ist, dass der Fokus ohnehin nur auf der Kommunikationstrecke zwischen Treiber und Emulator liegt und die Versuchsdauer in einem sinnvollen Rahmen gehalten werden muss. Das

VERILOG-Design akkumuliert – wie ein gewöhnlicher RAM-Baustein – geschriebene Werte auf einem großen MMIO-Bereich. Der aktuelle Wert einer Adresse kann außerdem jederzeit gelesen werden. Der initiale Speicherinhalt wird aus einer Hex-Datei geladen. Ein Teil der angebotenen Adressen ist jedoch schreibgeschützt, Schreibzugriffe an diesen Adressen werden einfach ignoriert. Wenn ein Treiber auf einen Interrupt des Designs blockiert, wird ihn das Design sofort auslösen, um den Treiber wieder zu deblockieren. Zu diesem Zweck wurde ein Hilfs-Input am Design eingerichtet, über den das Blockieren signalisiert werden kann.

Auf der Treiberseite kommt ein Programm mit zwei Threads zum Einsatz. Beide scannen je einen exklusiven Teil des MMIO Bereichs und testen, ob sie den Wert der aktuellen Adresse inkrementieren können. Sollte dies bei schreibgeschützten Bereichen gelingen oder bei beschreibbaren misslingen, so wird der Versuch erfolglos abgebrochen. Vor jeder Inkrementierung warten die Threads zudem auf einen der zwei Interrupts, die das Design bietet. Zwischen der Inkrementierung und der Überprüfung des Ergebnisses wird dann auf den jeweils anderen Interrupt gewartet. Der Treiber vermeldet Erfolg, sobald beide Threads ihren MMIO-Abschnitt erfolgreich gescannt haben. Danach springen beide Threads zurück zum Anfang und scannen erneut.

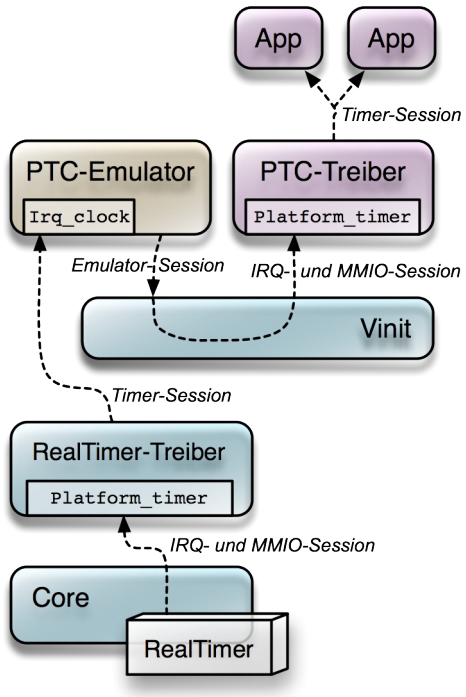


Abb. 19: Prozesse des PTC-Szenarios

Die Anzahl der Treiberprozesse wird nun, im Zuge des Lasttests variiert. Es konnte festgestellt werden, dass auch mit fünfzig Treiberinstanzen jeder einzelne Treiber erfolgreich ist. Rechnet man die Emulatorinstanzen hinzu, ergibt das hundert Prozesse, zwischen denen fortwährend bis zu hundert Emulationsvorgänge pseudo-parallel lauffen. Für den Test benötigt jeder Treiber mindestens elf Megabyte Arbeitsspeicher. Wie sich der Einsatzzweck dieser Ressource aufteilt zeigt Abbildung 20. Mit etwa acht Megabyte – durch die beiden blauen Flächen dargestellt – wird dem Treiber der Großteil seiner Zusicherung wieder entzogen, wenn VINIT die Emulatorinstanz startet. Einen Teil davon nutzt VINIT für die Erzeugung des Emulators und überweist diesem dann den Rest für seine eigene Tätigkeit. Das restliche Kontingent des Treibers

wird durch die beiden grünen Flächen dargestellt. Gut zwei Megabyte davon, also neunzehn Prozent des Gesamtverbrauchs fallen auf den Overhead, den die Nutzung der VINIT-Dienste verglichen mit nicht-emulierten Diensten verursacht. Die übrigen acht Prozent, oder knapp ein Megabyte sind somit die Kosten des Treibers, würde er das echte Gerät nutzen können. Bei dieser Betrachtung fällt der erhebliche Overhead des Emulators auf. Dies kann damit begründet werden, dass die HDL-Integration von `verilator_env` – auch bei simplen Designs – umfangreiche Bibliotheken, wie LIBC für Ausgabe und Dateisystem, LIBM und die C++ Standardbibliothek, sowie einen dynamischen Linker je Emulator benötigt.

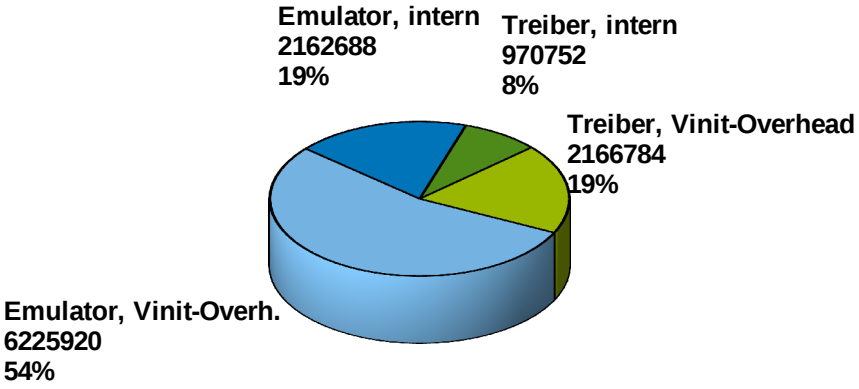


Abb. 20: Arbeitsspeicheraufteilung eines emulierten Treibers in Byte

Außerdem sind die grundlegenden Dienste des Emulators – durch den Umweg über VINIT – im gleichen Maß, wie die des Treibers kostenintensiver. Die hellblaue Fläche stellt diesen Anteil der Emulatorkosten dar, der allein durch das Monitoring von VINIT verursacht wird. Die reinen Emulatorkosten, von gut zwei Megabyte können dadurch, dass sie von Third-Party-Code bestimmt sind, wohl kaum gesenkt werden. Es wird hier aber ohnehin ersichtlich dass das größte Optimierungspotential bei VINIT und ihren Diensten liegt.

4.2.2 Fehlverhalten des Emulators

Ein Emulator soll für den Treiber möglichst die selben Eigenschaften besitzen wie das echte Gerät. Dementsprechend darf ein Fehlverhalten des Emulators auch nicht mehr Einfluss auf den Treiber haben, als ein fehlerhaftes Gerät ihn über seine Schnittstelle ausübt. Die Schnittstelle des Emulators gewährleitstet bereits einen sehr engen Rahmen für diesen Einfluss. Der Emulator kann neben dem Potential der Emulator-Schnittstelle nur seine eigenen Ressourcen korrumpieren. Denkbar wäre nun, dass der Emulator während eines Austauschs mit VINIT abstürzt oder anhält. In der VINIT selbst, stoppt dies aber nur den Thread, der ausschließlich den Treiber des Emulators bedient. Am Treiber werden also maximal

all die Threads angehalten, die den Emulator gerade nutzen. Dieser Effekt ist der gleiche, wie bei einem echten Gerät das eine MMIO-Transaktion nicht abschließt, oder die erwarteten Interrupts nie auslöst. Das gilt selbst dann, wenn man Bussysteme betrachtet, die sich in solchen Situationen durch Timeouts schützen, um zumindest die restlichen Teilnehmer weiter bedienen zu können.

Nun soll betrachtet werden, welches Potential dem Emulator durch die Emulatorschnittstelle zuteil wird. Möchte er die Schnittstelle zum Nachteil des restlichen Systems missbrauchen, bleiben ihm nur zwei Möglichkeiten. Er kann einen falschen Rückgabewert bei `read_mmio` bzw. `irq_handler` liefern, oder das Interrupt-Signal fälschlicherweise auslösen. Ein falsch gelesener MMIO-Wert gelangt durch VINIT immer nur in das Register, welches vom Treiber dafür vorgesehen wurde. Auch bei einem echten Gerät muss der Treiber davon ausgehen, dass dieses Register im Nachhinein einen ungünstigen Wert enthält. Nun könnte sich der Emulator die validen Interrupt-Signale, die ihm VINIT zuspielt auch dauerhaft merken. So könnte er sie auch weiterhin auslösen, obwohl VINIT bereits ein invalides Signal installiert hat. Dies hat jedoch keine negativen Auswirkungen. Jeder Interrupt-Thread in VINIT nutzt zwar ständig das gleiche valide Signal, aber er hört es nur ab, wenn der Treiber es verlangt. Wie oft das Signal ausgelöst wurde spielt dann keine Rolle. Das Auslösen veralteter Interrupt-Signale hat somit den gleichen Effekt, wie ein fälschlicherweise positiver Rückgabewert beim Aufruf von `irq_handler`. Der Treiber wird maximal zu früh deblockiert. Das wäre auch bei einem realen Gerät der Fall, wenn es seine Interrupt-Ausgänge falsch betreibt.

Überdies helfen die vorgestellten Tools, Fehler des HDL-Designs abzufangen und zu analysieren. Der WISHBONE-Slave kann Timeouts auf Transaktionen anwenden und ist stabil bei randomisierten Ausgangsbelegungen. Er wertet nur das Acknowledgment-Bit aus und reicht den Datenausgang durch. Betreibt ein Design fälschlicherweise Eingangssignale, hat dies keinen Effekt, da Eingangsbelegungen von den Tools nicht ausgewertet werden. Falsche Interrupt-Belegungen werden vom Interrupt-Taktgeber einfach auf die entsprechenden Signale übersetzt, die – wie besprochen – unkritisch sind. Über die Ein- und Ausgangsbelegungen hinaus, stehen dem HDL-Design nur Funktionen externer Bibliotheken, wie der LIBC zur Verfügung. gestattet der Entwickler dem Emulator jedoch nicht explizit entsprechende Plugins und Ressourcenrechte, so bleiben diese Funktionalitäten ein totes Ende und haben keinen Einfluss auf die Außenwelt.

4.2.3 Fehlverhalten des Treibers

Auch ein Fehlverhalten des Treibers an emulierten Bereichen, darf keinen Einfluss auf die eigenständige Funktion von VINIT oder anderen Prozessen haben. Zuerst soll solch ein Fehlverhalten bei der Anforderung der Ressourcen betrach-

tet werden. Deckt sich der angeforderte Bereich nur teilweise mit einem emulierten Bereich, verwehrt die Emulationsumgebung die Anfrage direkt. Dies führt zu einer ordentlichen Terminierung des RPCs, durch die Rückgabe eines Fehlercodes. Eine Ressource ist also entweder vollständig emuliert, oder vollständig nicht emuliert. Neben der Ressourcenart und den Bereichsgrenzen, werden keine weiteren Parameter der Anfrage ausgewertet. Nun soll angenommen werden, der Treiber habe einen emulierten MMIO-Bereich erfolgreich in Besitz genommen. Die damit verbundene Dataspace-Capability kann er nur weiterreichen oder lokal einbinden. Hat er den Bereich in den lokalen Adressraum eingebunden, so beschränkt sich die Interaktion mit der MMIO-Session auf MMU-Berichte, die dazugehörenden Befehlscodes und die Register, die von den Zugriffen genutzt werden. Sollte ein solcher Befehlscode ungültig sein oder ein ungültiges Register angeben, wird der Fehler bereits vom Compiler oder der CPU moniert werden. Unabhängig davon, werden solche Fehler aber auch vom Dekoder erkannt. Er würde dann in VINIT – wie schon der fehlerhafte Emulator – maximal den MMIO-Session-Thread blockieren, der ausschließlich den Treiber bedient. Den zu schreibenden oder gelesenen Wert hingegen, muss die Emulationsumgebung nicht auswerten. Er wird von ihr nur weitergeleitet. Am MMU-Bericht selbst, kann der Treiber indirekt nur die Zieladresse und das Zugriffsformat manipulieren. Ob ein Zugriff somit über den emulierten Bereich hinausragt prüft die Emulationsumgebung. Beschränkt sich der Zugriff auf einen Bereich, ist einzig der Emulator für die weitere Auswertung zuständig.

Hat der Treiber die Capability eines emulierten Interrupts erhalten, so kann er damit lediglich vermelden, dass er auf den Interrupt warten möchte. Dabei werden keine weiteren Werte kommuniziert. Das Vorgehen ist somit unkritisch. Auch Denial-Of-Service-Attacken sind nur eingeschränkt möglich. Da in VINIT jede emulierte MMIO-Verbindung ihren eigenen Thread besitzt – und somit auch jeder emulierte Interrupt – wird die CPU-Zeit der Emulationsumgebung fair auf alle Treiber aufgeteilt. Den Speicherverbrauch, der VINIT dadurch entsteht, dass sie ein Gerät emuliert, zieht sie dem Treiber von seinem Speicherquantum ab. Sollte das Speicherquantum nicht mehr ausreichen, wird maximal die entsprechende Session – wenn sie denn schon läuft – blockieren. Diese Blockade betrifft somit auch wieder nur einen VINIT-Thread, der ausschließlich dem Treiber dient. Sollte der Speicher allerdings schon beim Verbindungsaufbau ausgehen, so wird der RPC mit einer Fehlerrückgabe terminiert, da der zuständige VINIT-Thread alle Kinder bedient.

4.3 Skalierbarkeit

Um die Skalierbarkeit des Systems zu prüfen, wird die Dauer von emulierten Interruptzustellungen und die Dauer von emulierten Speicherzugriffen betrachtet. Zuerst wird die Anzahl der parallel zugreifenden Threads variiert, und im Anschluss auch die der zugreifenden Prozesse. Dadurch ergeben sich vier verschiedene Tests.

4.3.1 Die MMIO-Latenz über der Anzahl der Threads

Der erste Test zielt auf die Speicherzugriffe bei variierender Thread-Anzahl ab. Zu diesem Zweck kommt wieder das VERILOG-Design eines Speichermoduls, aus Kapitel 4.2.1 zum Einsatz. Alle Treiber-Threads laufen in einem Prozess, der über einen emulierten MMIO-Bereich mit dem Gerät kommuniziert. Jeder Thread verübt in einer Endlosschleife Scans über einen Teilbereich des MMIO, den nur er allein nutzt. Dabei wird fortwährend der Wert der aktuellen Adresse gelesen, inkrementiert und zurückgeschrieben. Danach wird erneut von der

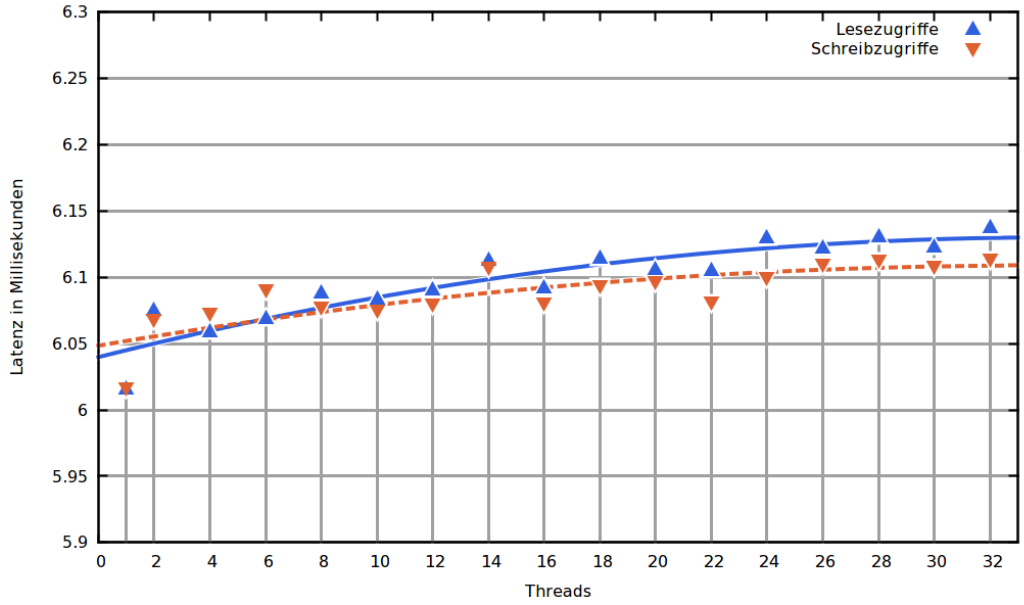


Abb. 21: Latenz emulierter MMIO-Zugriffe über der Anzahl der Threads

Adresse gelesen und das Ergebnis gegen die Schreibvorschriften des Designs geprüft. Der letzte Thread jedoch, wartet bis alle Initialisierungsarbeiten der anderen Threads abgeschlossen sind und startet dann selbst mit der gleichen Scan-Prozedur. Einziger Unterschied bei diesem Thread ist, dass die Dauer jedes MMIO-Zugriffs mittels eines realen Timers gemessen wird. Dass heißt, alle Threads erzeugen die gleiche, stetige Last an VINIT, aber der letzte Thread betreibt zusätzlich das Sampling. Der Sampling-Scan läuft über zweihundert aufeinanderfolgen-

de Adressen, wodurch vierhundert Lese- und zweihundert Schreiblatenzen gemessen werden. Über diesen Lese- und Schreiblatenzen wird dann jeweils das arithmetische Mittel gebildet. Die daraus resultierenden Werte werden an der Anzahl der Threads normiert, um die proportional abnehmende CPU-Leistung herauszufiltern. An dieser Stelle sei erwähnt, dass der CPU-Scheduler für die Tests so konfiguriert wurde, dass er eine faire Aufteilung ohne Prioritäten umsetzt. Die gemittelten Latenzen werden für die Normierung also einfach durch die Anzahl der Threads geteilt. Die Ergebnisse des Tests und entsprechende Approximationen sind in Abbildung 21 dargestellt. Bei niedriger Thread-Zahl steigen die normierten Latenzen noch merklich. Das ist vermutlich eine Folge dessen, dass das CPU-Scheduling in der genutzten Core-Variante eine Besonderheit aufweist: Um den Scheduling-Algorithmus einfach zu halten wird derzeit hingenommen, das ein Thread seine CPU-Zeit nicht an andere Threads weitergeben kann. Das hat zur Folge, dass ein Thread, der eine RPC-Anfrage an einen Server schickt, zugleich den Rest seiner aktuellen Zeitscheibe verliert und bei der Antwort wieder am Ende der Scheduling Queue eingegliedert wird. Dieser Zeitverlust hat bei geringen Zugriffslatenzen noch einen sichtbaren Einfluss auf die Werte. Mit steigender Thread-Zahl und somit steigenden Latenzen verliert er aber an Bedeutung. Je höher die Thread-Anzahl wird, desto deutlicher wird, dass die normierten Zugriffszeiten tatsächlich gegen einen konstanten Wert streben. Die Zugriffzeit skaliert also langfristig über der Anzahl der Treiber-Threads.

4.3.2 Die Interrupt-Latenz über der Anzahl der Threads

Im zweiten Test soll die Zustellungsdauer emulierter Interrupts betrachtet werden. Zu diesem Zweck muss die Anbindung des VERILOG-Designs angepasst werden. Jeder Interrupt soll – wie beim Lasttest – sofort ausgelöst werden, sobald jemand auf ihm blockiert. Bei einem bestimmten Interrupt jedoch, muss zusätzlich eine Messung der Zustellungsdauer stattfinden. Deshalb teilen sich Treiber und Emulator einen realen Timer. Der Emulator startet den Timer, sobald er den besagten Interrupt auslöst. Der Treiber hingegen, stoppt den Timer, unmittelbar nach dem Empfang des Interrupts. Unter den Treiber-Threads ist wieder der Thread mit der Messung beauftragt, der zuletzt startet. Er nutzt deshalb den speziellen Interrupt, während jeder andere Thread einen gewöhnlichen Interrupt des Emulators zugeteilt bekommt. Für jede Thread-Anzahl wird die Messung einhundertmal wiederholt. Die Ergebnisse werden – wie schon bei den MMIO-Zugriffen – gemittelt und an der Thread-Anzahl normiert.

Abbildung 22 zeigt die Ergebnisse des Tests und eine Approximation. Bei den Messungen mit wenigen Threads sieht man wieder – wie auch beim ersten Test – dass die Latenz erst stark zunimmt, um sich dann allmählich zu stabilisieren. Dieses Verhalten deutet erneut auf den Zeitverlust hin, der beim Scheduling von

RPC-Anfragen entsteht und dessen Anteil bei steigender Latenz abnimmt. Die Kurve ist aber weitläufiger als im ersten Test, was wohl darauf zurückzuführen ist, dass die gemessenen Latenzen viel geringer sind und der Zeitverlust somit stärker und länger ins Gewicht fällt. Doch auch wie im ersten Test gilt: Je mehr Threads parallel auf Interrupts warten, desto konstanter verhält sich die normierte Latenz. Die Dauer emulierter Interrupt-Zustellungen skaliert also langfristig über der Thread-Anzahl.

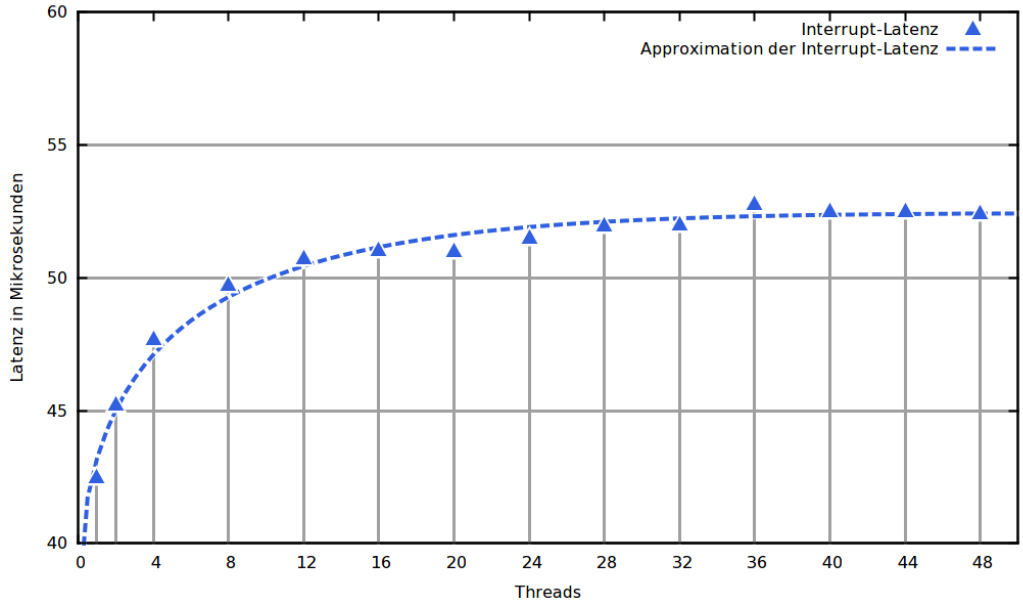


Abb. 22: Latenz emulierter Interrupts über der Anzahl der Threads

Der auffälligste Unterschied zwischen den beiden Testergebissen ist, dass kein Interrupt mehr als vierundfünzig Mikrosekunden bis zum Treiber braucht, während bei einem emulierten MMIO-Zugriff mindestens sechs Millisekunden verstreichen. Wenn man die beiden Vorgänge aus der Sicht von VINIT betrachtet, wird schnell ersichtlich, warum zwischen dem Durchsatz von MMIO- und Interrupt-Emulation Größenordnungen liegen. Bei einem emulierten MMIO-Zugriff muss Core zuerst den MMU-Bericht an VINIT schicken. VINIT lässt dann, über Core, den Befehlscode-Dataspace einhängen und den Thread Context lesen. Erst jetzt kann der Befehl dekodiert und vom Emulator ausgeführt werden. Ist dies getan, hängt VINIT den Befehlscode-Dataspace wieder aus, schreibt den Thread Context zurück und weckt den Treiber. Für jede dieser Aufgaben muss sie erneut auf Core zurückgreifen. Somit hat das System mindestens achtzehn Adressraumwechsel und hunderte Zeilen Code hinter sich, ehe der Treiber-Thread wieder erwacht. Die Interrupt-Zustellung hingegen, besteht nur darin, ein asynchrones Signal zu empfangen und den Treiber-Thread zu wecken. Dabei wechselt der Kontrollfluss lediglich dreimal den Adressraum.

4.3.3 Die MMIO-Latenz über der Anzahl der Treiber

Anders, als bei den zwei ersten Tests, soll nun auch die Anzahl der Treiberprozesse variiert werden. Dabei wird die Anzahl der Threads je Treiber auf einen Wert festgelegt, der einerseits – entsprechend des Multithreading-Tests – repräsentativ für die Zugriffsdauer ist und andererseits möglichst wenig Testzeit in Anspruch nimmt. Jeder Treiberprozess startet deshalb zwei Threads, die wieder jeweils auf einem exklusiven Teilbereich des MMIO arbeiten. Die Vorgehensweise der Threads ist die selbe, wie im ersten MMIO-Test. Alle Treiber nutzen den gleichen MMIO-Bereich, doch im Hintergrund erhält natürlich jeder seine eigene Emulatorinstanz von VINIT. Einer der Treiber-Threads wartet wieder, bis die Initialisierung aller anderen Prozesse und Threads abgeschlossen ist, und startet dann die Messung. Auch die Messung und Auswertung der Ergebnisse geschieht auf die gleiche Art und Weise, wie im ersten Test.

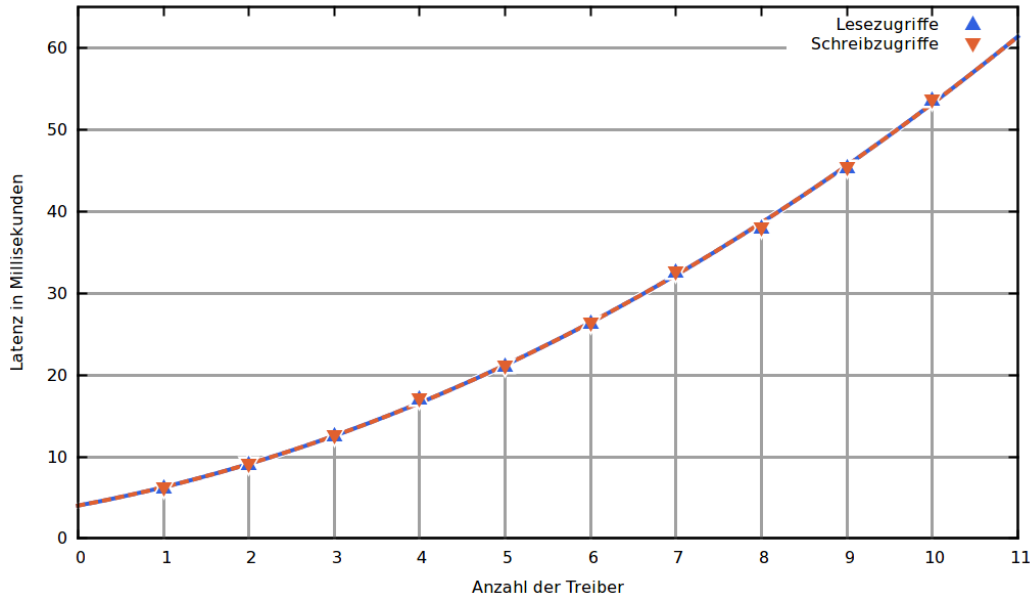


Abb. 23: Latenz emulierter MMIO-Zugriffe über der Anzahl der Treiber

Abbildung 23 zeigt das Resultat, normiert über der Anzahl aller Treiber-Threads. Bei einem einzelnen Treiber entspricht das Ergebnis von sechs Millisekunden erwartungsgemäß dem Wert, den auch der erste Test mit zwei Threads gemessen hat. Im Kontrast zum ersten Test steigt die normierte Latenz nun aber durchgängig. Für eine Erklärung muss beleuchtet werden, welche Folgen die Neuerungen in diesem Szenario nach sich ziehen. Durch den Zuwachs an Prozessen kommt keine weitere CPU-Last auf. Die Zahl pseudoparalleler Aktivitäten ist immernoch durch die Zahl der Treiber-Threads begrenzt, da in VINIT nur die entsprechenden Session-Threads laufen. Der Performance-Verlust muss also mit den zusätzlichen Adressräumen zusammenhängen. Tatsächlich zieht die Verwaltung

weiterer Adressräume Kosten nach sich. Gibt es wenige Adressräume, beschleunigen die MMU- und Speicher-Caches noch nahezu jeden Zugriff auf virtuellem Speicher. In diesem Zustand sind die Latenzen unabhängig von der Zahl der Adressräume. Nur die Zahl der Threads beeinflusst das Ergebnis. Die normierte Latenz ist deshalb anfangs – wie in der Abbildung zu sehen – noch recht stabil. Weitere Adressräume ziehen aber eine immer häufigere Verdrängung in den MMU- und Speicher-Caches mit sich, wodurch diese stetig an Effizienz verlie-

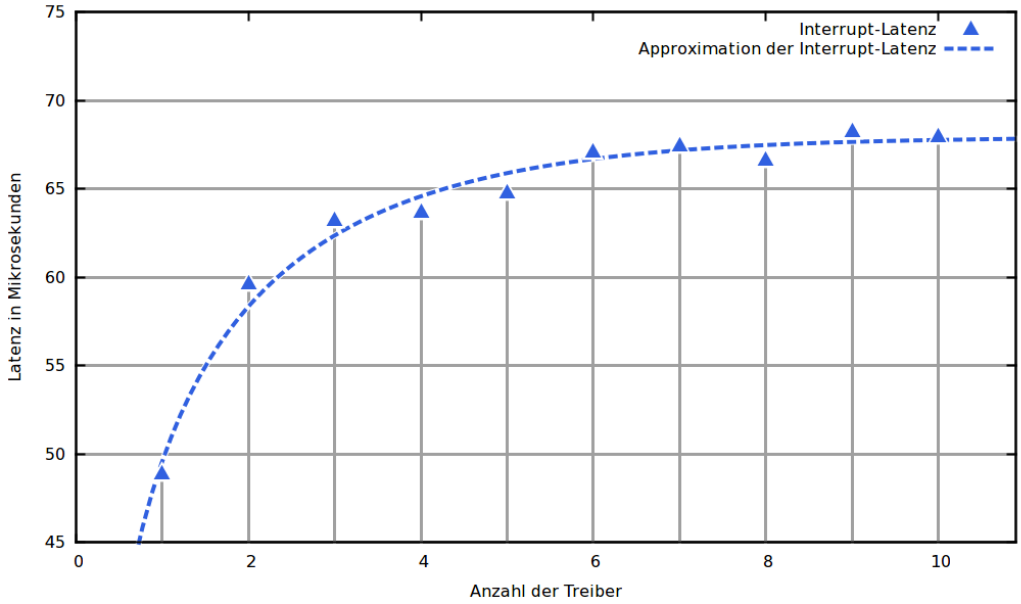


Abb. 24: Latenz emulierter Interrupts über der Anzahl der Treiber

ren. Der Effekt wird noch dadurch verstärkt, dass jeder Treiber auch einen weiteren Emulator mit sich bringt, also pro Treiber gleich zwei Adressräume hinzukommen. Diese Kosten werden durch die Normierung nicht gefiltert und es kommt allmählich zu einem Anstieg der Testwerte. Der beobachtete Performance-Verlust wird also aller Vorraussicht nach nicht durch die Codesign-Umgebung verursacht, sondern hängt mit den plattformspezifischen Beschränkungen der Speichervirtualisierung zusammen.

4.3.4 Die Interrupt-Latenz über der Anzahl der Treiber

Auch der Interrupt-Test wird noch einmal mit variabler Treiber-Anzahl durchgeführt. Statt zwei, kommen hier aber vier Threads je Treiber zum Einsatz, da sich die Ergebnisse des ersten Interrupt-Tests später einpegeln als die des ersten MMIO-Tests. Wieder werden einhundert Wiederholungen je Messung durchgeführt. Die Ergebnisse werden wie gewohnt gemittelt und über der Zahl aller Treiber-Threads normiert. Abbildung 24 zeigt das Resultat des Tests. Erneut ist eine Leistungseinbuße durch die zusätzlichen Adressräume zu erkennen: Die Latenz

pegelt sich nunmehr fünfzehn Mikrosekunden höher ein als beim ersten Interrupt-Test. Doch dass sie sich trotz zusätzlicher Adressräume überhaupt einpegelt, ist zugleich ein auffälliger Unterschied zu den MMIO-Tests. Offenbar scheint die Effizienz der MMU- und Speicher-Caches kaum unter dem Szenario zu leiden. Dieser Umstand ist wohl auf zwei Begebenheiten zurückzuführen. Einerseits zieht eine Interrupt-Zustellung selbst – wie schon erwähnt – nur drei, anstatt achtzehn Adressraumwechsel mit sich. Andererseits benötigen Interrupt-Zustellungen viel weniger Zeit als MMIO-Zugriffe. Dazu sei erwähnt, dass die Zeitscheibe des CPU-Schedulings bei allen Tests konstant auf einhundert Millisekunden eingestellt ist. Daran gemessen liegen zwischen den Ergebnissen des MMIO-Zugriffs und denen der Interrupt-Zustellung Größenordnungen. Die Chance einer zusätzlichen Unterbrechung der Zustellung – verbunden mit Adressraumwechseln und Cache-Verdrängungen – ist also viel geringer. Infolgedessen werden die Caches offenbar so geringfügig belastet, dass der Effizienzverlust über dem Definitionsbereich des Tests eher eine flache Gerade beschreibt, wodurch in der resultierenden Kurve der Eindruck einer konstanten Latenzsteigerung erweckt wird. Diese Erkenntnisse geben Anlaß dazu, insbesondere die Zahl der Adressraumwechsel in zukünftige Optimierungsvorhaben einzubeziehen.

4.4 Sicherheit

Die Grundlage für die Sicherheitsanforderungen eines emulierten Treibers bilden die Anforderungen, die im nicht-emulierten Fall verfolgt werden. Dort vertraut der Treiber all seinen Elternprozessen uneingeschränkt. Überdies erhalten Prozesse nur dann Informationen und Rechte, über die Ressourcen des Treibers oder seiner Eltern, wenn letztere es explizit erlauben. Betrachtet man den Prozessbaum für den nicht-emulierten Fall, beschränkt sich der Einschnitt, für die Emulation eines Treibers genau auf einen Prozess. Denn es wird nichts weiter getan, als den nächsthöheren Init-Prozess durch VINIT zu ersetzen. Der Sicherheitsvergleich zwischen emuliertem und nicht-emuliertem Fall, lässt sich also auf einen Vergleich zwischen Init und VINIT beschränken. Es muss sichergestellt sein, dass VINIT zwei Aspekte einhält. Sie darf nicht, über den Rahmen von Init hinaus zulassen, dass andere Prozesse Informationen mit dem Treiber in Verbindung bringen können. Außerdem dürfen andere Prozesse, durch den Einsatz von VINIT nicht mehr Einfluss auf den Treiber erlangen als durch Init. Letzteres wurde bereits in Kapitel 4.2 gezeigt. VINIT's Informationsfluss erweitert nun den von Init nur insofern, dass Zugriffsparameter von Core beschafft werden und ein Emulator Anweisungen erhält. Es bleibt also nur zu zeigen, dass es dem Emulator unmöglich ist, Anweisungen einem bestimmten Treiber zuzuordnen.

Ein Emulator bekommt bei seiner Erzeugung keine Information darüber, für welchen Prozess oder Emulationskontext er erzeugt wurde. VINIT stellt ihm lediglich sein Binary, seine meist leere XML-Konfiguration, seinen CPU-Kontext mit Stack- und Befehlszeiger, seinen virtuellen Adressraum, ausgenullten Arbeitsspeicher und eine unbestimmte Logausgabe zur Verfügung. Auch im weiteren Verlauf tritt nur VINIT mit ihm in Verbindung – immer dann, wenn der Treiber einer Emulation bedarf. Bei MMIO-Zugriffen wird die emulatorlokale Zieladresse übermittelt. Da die Umrechnung nach den treiberunabhängigen Angaben in der XML-Konfiguration von VINIT geschieht, ist es unmöglich, einen Rückschluss auf die Treiberadresse zu ziehen. Neben der Zugriffsadresse werden noch das Zugriffsformat und gegebenenfalls ein Schreibwert kommuniziert. Diese beiden Werte dienen nicht vielmehr zur Identifikation des Treibers, als sie es an einem echten Gerät täten. Auch beim Demaskieren eines Interrupts erhält der Emulator nur die emulatorlokale Adresse. Zusätzlich wird jedoch eine valide Signal Capability übertragen. Capabilities stellen aber im allgemeinen nur prozesslokale Informationen dar.

5 Schluss

In dieser Arbeit wurde ein System vorgestellt, das es ermöglicht, Software-Tests einfach, flexibel und sicher mit simulierten Geräteentwürfen anzureichern. Das System wurde im Hinblick auf die passgenaue Integration, in den Alltag des Hardware-Software-Codesigns entwickelt. Dort bildet es die Basis für eine vielseitige Systemanalyse. Dabei kommt keine proprietäre Software zum Einsatz. Das GENODE Betriebssystem-Framework, VINIT und sogar VERILATOR, der integrierte Third-Party-Simulator sind Open-Source-Software unter GPLV2 bzw. LGPL. Zudem wurden alle Komponenten, die im Zuge dieser Arbeit entstanden, umfangreich dokumentiert. Das System bietet also auch gute Vorraussetzungen für die eigenständige Einarbeitung, individuelle Adaptionen und eine gemeinschaftliche Weiterentwicklung.

5.1 Diskussion der Thesen

Für Treiber- und Geräteentwürfe ist die Emulationsumgebung transparent. Beispielhaft wird das an zwei typischen Kommunikationsmodellen gezeigt, die in Kapitel 3.5 implementiert werden. Die Emulation läuft sogar unabhängig davon ab, ob der Initiator ein Treiber oder selbst ein Emulator ist. Deshalb können Geräteentwürfe auch untereinander kommunizieren. Dabei weisen die Emulationsumgebung und die Emulatoren keine speziellen Hardware-Vorraussetzungen auf. Dass der Pagefault-Mechanismus einer MMU genutzt wird, ist eher der Tatsache geschuldet, dass die Speichervirtualisierung hardware-gestützt und nicht rein software-seitig abläuft. Allgemeiner gesehen, stellt das System ein Framework für die Implementierung verschiedenster Kommunikationsmodelle dar. So lässt sich – wie Kapitel 2.4.4 erklärt – auch eine DMA-Simulation hinter den nativen Schnittstellen umsetzen. Damit realisiert das System die grundlegende Idee dieser Arbeit, die in These 1.2.1 formuliert wurde.

Für die Synthese eines Emulators wird das unmodifizierte VERILOG-Design verwendet. Das Design wird wie ein normaler Software-Prozess in die Buildchain eingebunden und lediglich – wie Kapitel 3.5.1 erläutert – als emuliertes Gerät gekennzeichnet. In die Synthese der Software greift das System nicht ein. Die Forderungen bezüglich der Hardware- und Software-Synthese aus These 1.2.2 sind also erfüllt. Für die Konfiguration des Interrupt-Kommunikationsmodells müssen

– wie in Kapitel 2.4.3 gezeigt – Interrupt-Signal und Abtasttakt identifiziert werden. Diese Aufgabe entspricht bei einem realen Interrupt-Controller der Wahl eines Arbeitstaktes und dem Anlegen der Interrupt-Line. Durch das XML-Interface, aus Kapitel 3.4.2, erhält der Interrupt dann einen globalen Namen. Das sind die einzigen Einstellungen die ein Entwickler je Interrupt vornehmen muss. Für das WISHBONE-Kommunikationsmodell werden – wie in Kapitel 2.4.1 beschrieben – die Rollen einer Bus-Schnittstelle den entsprechenden Gerätesignalen zugeordnet. Dazu zählt auch wieder der Arbeitstakt. Darüber hinaus, werden dem Bus-Interface die gewünschten Einstellungen des Protokolls, wie Adressbreite und Timeouts genannt. Abschließend wird jedem MMIO-Bereich, den das Gerät anbietet, ein Speicherbereich des Treibers zugeordnet. Dafür kommt wieder das XML-Interface zum Einsatz. Die beiden Konzepte konnten also umgesetzt werden, ohne dass der Entwickler Einstellungen treffen müsste, die sich nicht direkt aus dem Kommunikationsmodell ergeben. Damit sind auch die Forderungen zur Schnittstellen-Synthese, aus These 1.2.2 erfüllt.

Derzeit erhält jeder Software-Prozess seinen eigenen Simulationskontext an einem emulierten Gerät. Diesen kann er mit den MMIO- und Interrupt-Capabilities des Geräts an andere Prozesse weitergeben. Dieses System ist ein Relikt der vorangegangenen Arbeit, in der die gemeinsame Nutzung von Geräten eine untergeordnete Rolle spielte. These 1.2.3 zur Konfigurierbarkeit der Exklusivität wird somit nur teilweise erfüllt, da die gemeinsame Nutzung einer Gerätesicht mitunter des Eingriffs in die Software bedarf. In Kapitel 5.2 wird jedoch gezeigt, dass die künstliche Einschränkung des Geltungsbereichs leicht zu einem intuitiven und transparenten Modell umgebaut werden kann.

Nun sollen die Kosten des Systems betrachtet werden. Das Upgrade von Init zu VINIT ermöglicht dem betroffenen Subsystem die Benutzung von Geräteentwürfen. VINIT besitzt bezüglich der Rechte und Ressourcen eine Vollmacht über das Subsystem, ebenso wie es bei Init der Fall ist. Zusätzliches Vertrauen kostet das Update also nicht. Dafür verlangt die dauerhafte Zwischenschaltung der Monitoring-Dienste aus Kapitel 2.2.3 mehr Speicher und CPU-Zeit. Diese Kosten schlagen sich – dank der Virtualisierung der Ressourcen – nur auf das Subsystem von VINIT nieder. Prozesse, die von den Geräteentwürfen unabhängig sind, und damit auch von Prozessen die diese Entwürfe nutzen, können unmodifiziert in ein anderes Subsystem verlagert werden. Die Kosten eines Simulationskontextes hingegen sind komplexer. Sie umfassen die Kosten des Emulators und der Verbindungen, die der Treiber zu VINIT unterhält, um mit dem Gerät kommunizieren zu können. Alle Ressourcen, die der Emulator benötigt, bezahlt der Treiber der ihn nutzen möchte. Dies geschieht – wie in Kapitel 3.4.1 beschrieben – sobald er bei VINIT die erste Verbindung zu dem Gerät beantragt. Wie aus Kapitel 4.4 hervorgeht, wird dem System, allein durch die Existenz des Emulators kein zusätzliches

Vertrauen abverlangt. Die Vertrauensbeziehungen, bei einer Invokation des Emulators, beschreibt dieses Kapitel ebenfalls. Tatsächlich lassen sie sich auf den Rahmen beschränken, den die Geräteschnittstelle vorgibt. CPU-Zeit und andere Ressourcen doniert der Treiber, nach der Erzeugung des Emulators nur explizit, über die Geräteschnittstelle. Damit entspricht die Kostenverteilung den Maßstäben, die in These 1.2.4 gesetzt wurden.

Auch das Ausmaß der Kosten wird den Anforderungen der Arbeit gerecht. Obwohl die Kosten der Emulationsumgebung teilweise noch recht hoch sind, konnte These 1.2.6 in den Kapiteln 4.1 und 4.3 bestätigt werden. Die CPU-Kosten einer Geräteanbindung steigen, über der Anzahl der pseudoparallel laufenden Anbindungen höchstens proportional. Wird jedoch zugleich die Zahl der Adressräume erhöht, so werden die Messwerte – offensichtlich durch den Effizienzverlust von MMU- und Speicher-Caches – verfälscht. Diesbezüglich sind detailliertere Tests vonnöten. Der Speicherverbrauch der emulierenden Verbindungen blieb während der Tests konstant. Das lässt sich darauf zurückführen, dass jede Session in VINIT ihre gesamten Speicherkosten selbst übernimmt und diese nicht davon abhängen, wieviele Sessions es insgesamt gibt.

Zur Beobachtung der Kommunikation mit emulierten Geräten, kann ein einfaches Modell herangezogen werden. VINIT ist aus Sicht des Entwicklers ein vertrauenswürdiger Prozess, der jedweden Austausch, zwischen einem emulierten Gerät und einem anderen Prozess im Klartext durchleitet. Aus Kapitel 2.2.3 geht hervor, dass dieses zentrale Organ zwischen den Kommunikationsendpunkten in beiden Richtungen auf synchronem RPC basiert. Deshalb obliegt es VINIT, wie lange der Kommunikationsfluss, und damit auch Sender und Empfänger blockiert bleiben. Die zu übermittelnden Daten kann VINIT auch abändern bevor sie weitergeleitet werden. Diese Fähigkeiten muss VINIT nun nur noch über einen eigenen Dienst bereitstellen, der den eigenen Kindern durch eine entsprechende XML-Konfiguration vorenthalten wird. Die Vermittlung durch einen prozessübergreifenden Dienst, begünstigt die eigenständige Entwicklung von Analyse-Tools in separaten Prozessen. Dieser Dienst macht jedoch nur dann Sinn, wenn der Beobachter Kommunikationsbeziehungen explizit adressieren kann. Zu diesem Zweck können die eindeutigen Namen herangezogen werden, die VINIT – ebenso wie schon Init – für alle direkten Kinder hält. Dem Entwickler, und damit auch den Analysetools sind die Namen aus der XML-Konfiguration bekannt. VINIT selbst, kann sie den emulierenden Sessions bei der Erzeugung mitgeben, damit im Fall eines Austauschs eine Zuordnung möglich ist. Damit ist auch These 1.2.5 bestätigt.

5.2 Ausblick

Das vorgestellte System eröffnet Möglichkeiten und Probleme, die es in der Zukunft wahrzunehmen bzw. zu beheben gilt. Zu den Problemen gehört vor allem die Tatsache, dass Performance-Optimierung nicht zu den Schwerpunkten dieser Arbeit zählt. Speicherverbrauch und Latenz können vorraussichtlich reduziert werden, wie bereits Kapitel 4 andeutet. Aus diesen Gründen sollen nun einige Anregungen zusammengestellt werden, die sich im Laufe der Arbeit ergeben haben.

Der Großteil des zusätzlichen Speicherverbrauchs kann auf die Datensätze der Monitoring-Dienste und die Emulatoren der emulierbaren Dienste zurückgeführt werden. Der erwartete Verbrauch einer solchen Verbindung wird momentan schon bei der Dienstanfrage eingefordert, obwohl z.B. die Intensität des Monitorings – auch unabhängig vom Emulationsaufkommen – stark variiert. Hinzu kommt, dass dieser Ressourcenvorschuss zwangsläufig für alle Verbindungen eines Dienstes gilt, egal, ob es sich um einen emulierten oder normalen Adressbereich handelt. So kommt es, dass VINIT z.B. bei jeder MMIO-Verbindung Kosten der Emulatorerzeugung berechnet, obwohl dies für reale MMIO-Bereiche garnicht nötig wäre. GENODE ermöglicht es aber mittlerweile, dass das Speicherressort eines Dienstes ad hoc vom Client angepasst wird. Dieser Mechanismus kann hier insofern Abhilfe schaffen, dass eine Session initial immer nur die Basiskosten berechnet, da sie Emulationskosten – wenn sie denn aufkommen – auch ad hoc in Rechnung stellen kann. Ein anderer Aspekt ist die Aufteilung des Speicherressorts, zwischen Monitoring- und Hintergrunddienst. Sie ist derzeit statisch und basiert nur auf einfachen Heuristiken. Auch hier könnte mit der dynamischen Zuweisung eine effizientere Speicherausnutzung erzielt werden. Weiteres Potential findet sich bei den Datenstrukturen mit denen die Monitoring-Dienste ihr Wissen verwalten. Sie wurden bisher nicht variiert, um eventuell eine effizientere Lösung zu finden.

Bei den Latenzen, die das System verursacht, fällt am deutlichsten die der emulierten MMIO-Kommunikation ins Gewicht. CPU-intensiv dürfte dabei vor allem die aufwändige Übersetzung des Pagefaults in Emulatoranweisungen sein. Bei jedem Zugriff erfordert dieser Schritt allein mehrere Adressraumwechsel, die unter Umständen eingespart werden könnten. Für gewöhnlich befindet sich, zum Beispiel der Befehl des Treibers in einem schreibgeschützen Dataspace. Solange sich also die Adresse des Dataspace nicht ändert, können Emulatoranweisungen – wenn sie einmal für einen bestimmten IP berechnet sind – für erneute Aufrufe gecached werden. Damit dürften vor allem Programmabschnitte beschleunigt werden, die sich häufig wiederholen. Die Größe dieses transparenten Caches könnte sich dynamisch daran bemessen, wieviel überschüssiger Speicher der Verbindung

gerade zur Verfügung steht. Somit läge es auch in der Hand des Treibers, wieviel er zusätzlich in die Geschwindigkeit der Verbindung investiert. Ein Anderer Ansatz bestände darin, den überschüssigen Speicher für Shared Memory zu nutzen. Für eine Emulation müssen immer wieder Daten aus anderen Adressräumen herangeschafft werden. Der Speicher der Treiberinstruktion wird lokal eingehängt. Der CPU-Kontext des Treibers wird über den Mikrokernel gelesen und geschrieben. Diese Umwege könnten eingespart werden, wenn die Daten beim ersten Zugriff als Shared Memory eingehängt würden. Für den Thread-Kontext erfordert das zwar eine Erweiterung der Kernel-Schnittstelle, die aber wie gewohnt durch Capabilities abgesichert wäre.

Ein anderer offener Punkt ist die eingeschränkte Hardware-Unterstützung, die momentan noch wenig Raum für den Einsatz des Systems lässt. Der aktuelle Dekoder ist aber so aufgebaut, dass das Grundgerüst nicht nur für ARMV7 herhält, sondern für Load-Store-Architekturen im Allgemeinen nutzbar ist. Dieses Gerüst ermöglicht es, die verschiedenen Spezifikationen eines Dekodersatzes, wie z.B. ARM, ARMV7 und CORTEX-A9 sauber auf verschiedene Dateien zu verteilen. So wird je nach Zielarchitektur ein Dekoder zusammengestellt, der nur die nötigen Funktionalitäten enthält, während Code-Redundanz vermieden wird. Das Modell des Dekoders kann, durch die Verwendung von Bitmaps oder Listen bei der Rückgabe, auch für SIMD-Zugriffe genutzt werden. Schwieriger wird die Umsetzung für Architekturen, in denen arithmetisch-logische Befehle, Kontrollflussbefehle und Speicherzugriffe vereint vorkommen. Das Modell von VINIT dürfte jedoch auch für diesen Umstand herhalten. So könnte VINIT die Speicherzugriffe für den Emulator herausfiltern, während sie die restlichen Anweisungen, am Kontext des Treibers und den Ergebnissen des Emulators selbst durchführt. Zieht diese Vorgehensweise zuviel Komplexität in VINIT nach sich, könnte Funktionalität, die nicht den Emulator betrifft und keine Sprünge enthält, auch als temporärer Codepatch in den Stack des Treiber-Threads injeziert werden. Der Stack-Bereich wird zuvor von VINIT gesichert. Nachdem der Treiber dann selbst die Funktionalität - unter Umständen mithilfe der Emulationsergebnisse - durchgeführt hat, lässt ihn der Patch zurück nach VINIT springen. Dort wird der Stack-Inhalt wiederhergestellt und der Thread zurück auf seine normale Bahn gebracht.

Weiterhin wäre es vorstellbar, die Beschränkung aufzuheben, dass das System nur Peripherie emuliert. Sorgt VINIT dafür, dass jeder Programm-Counter am Treiber einen Prefetch-MMU-Fehler auslöst, welcher dann – ebenso wie die Load-Store-MMU-Fehler der MMIO-Zugriffe – abgefangen wird, kann eine Instruction-Set-Simulation durchgeführt werden. Da der Dekoder damit ebenfalls zum Entwicklungsgegenstand und mitunter sehr komplex wird, würde er in ein emulatorähnliches Kind von VINIT ausgelagert. Er meldet VINIT dann, welche Effekte eine Instruktion auf CPU-Kontext und Arbeitspeicher des Treibers hat.

VINIT setzt diese Effekte an den Ressourcen des Treibers um und lässt ihn dann die nächste Instruktion laden. Effekte an emulierten Ressourcen kann VINIT gleich an die Peripherie-Emulatoren weiterleiten, so dass die restliche Emulation wie gewohnt vonstatten geht.

Ein Modell für die fehlende Implementierung virtueller DMA-Zugriffe wurde bereits in den Kapiteln 2.4.4 und 3.5.4 besprochen. Die überzogene Exklusivität von Simulationskontexten hingegen, wurde bisher nur angerissen. Erste Überlegungen führten zu dem Standpunkt, dass VINIT eher die Rolle einer kohärenten, virtuellen Umgebung einnehmen sollte, anstatt nur als Instrument zur Verwaltung und Anbindung von Emulatoren herzuhalten. Zu Beginn dieser Arbeit waren die einzelnen Aspekte der Emulationsumgebung noch stärker über Anwendungen, das Betriebssystem und den Kernel verteilt. Die Verwaltung der Emulationskontexte waren Aufgabe der VDM-Anwendung, während das Layout der Umgebung Teil des Betriebssystems und somit global gültig war. Nun sind beide Aspekte in VINIT vereint und das Konzept der eigenständigen Sicht eines Geräts kann in eine kohärente Sicht der gesamten Umgebung überführt werden. Unterschiedliche Entwürfe dieser emulierten Umgebung, könnten dennoch parallel getestet werden, indem mehrere VINIT-Subsysteme eingesetzt werden. Diese neue Sichtweise stellt auch eine wichtige Umstellung im Ressourcen-Accounting dar: Die Kosten einer Gerätesicht, also eines Emulators, können nicht länger einzelnen Kindern zugeordnet werden.

Letztendlich sollte der nächste größere Schritt, beim Ausbau der Codesign-Umgebung in der Anbindung von Analysetools bestehen. Ein erster Ansatz wäre die Visualisierung des Kommunikationsflusses, über die Schnittstelle, die Kapitel 5.1 zum Thema der Beobachtbarkeit vorschlägt. Nutzer sollten zu Beginn eines Tests Regeln für die Aktivierung des Monitors einrichten können. Erfüllt ein Kommunikationsvorgang in VINIT eine der Regeln, soll der Vorgang für eine Interaktion mit dem Monitor pausiert werden. Jede Regel kann mit einem individuellem Aufzeichnungsmuster verbunden werden. Dieses Muster kann Daten und Parameter der Kommunikation enthalten. Außerdem sollte der Nutzer die Möglichkeit haben, die Pausierung der Teilnehmer an Regeln zu binden. Tritt eine solche Regel dann ein, belässt der Monitor den Vorgang in VINIT blockiert, um dem Nutzer die Interaktion zu ermöglichen. Mit mehreren Monitoren könnten außerdem auch verschieden Aufzeichnungen, zugleich und an der selben VINIT durchgeführt werden. Eine weitere Idee wäre ein ähnlich gearteter Monitor, für die Ein- und Ausgangssignale von Geräteenwurfen. Dazu könnte man ein Plugin für Emulatoren anbieten, das den aktuellen Status wie gewohnt über einen Dienst bereitstellt.

Literaturverzeichnis

[1] Genode Labs, *Documentation of the Genode OS Framework*, verfügbar über *www.genode.org*, Zugriff am 15.10.2012

[2] F. Bellard, *QEMU a Fast and Portable Dynamic Translator*, 2005, verfügbar über static.usenix.org

[3] S. Bourdeauducq, *A performance-driven Soc architecture for video synthesis*, 2010, verfügbar über www.milkymist.org

[4] M. Stein, *Modell zur Transparenz der Verfügbarkeit von Hardware-Ressourcen auf der Systembusebene,* 2011, verfügbar über www.genode-labs.com

[5] Operating Systems Group of the TU Dresden, *Fiasco Microkernel Overview*, verfügbar über os.inf.tu-dresden.de, Zugriff am 16.10.2012

[6] U. Steinberg, B. Kauer, *NOVA: A Microhypervisor-Based Secure Virtualization Architecture*, 2010, verfügbar über os.inf.tu-dresden.de

[7] Genode Labs, *Current limitations and technical remarks of the User-level debugging on Genode via GDB*, verfügbar über www.genode.org, Zugriff am 17.10.2012

[8] G. Heiser, B. Leslie, *The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors*, 2010, verfügbar über dl.acm.org

[9] Open Kernel Labs, *OKL4 Microkernel Reference Manual, API Version 03, Rev. 12*, 2008, verfügbar über www.reds.ch

[10] Operating Systems Group of the TU Dresden, *Fiasco/L4 System Call C-Bindings Reference Manual – Other system calls*, verfügbar über os.inf.tu-dresden.de, Zugriff am 17.10.2012

[11] T. Gingold, *GHDL user guide*, 2006, verfügbar über attila.kinali.ch

[12] W. Snyder, D. Galbi, P. Wasson, *Verilator Documentation*, 2012 verfügbar über www.veripool.org

[13] ARM Limited, *ARM Architecture Reference Manual - ARMv7-A and ARMv7-R edition - Issue A*, 2007

[14] Xilinx Inc., *MicroBlaze Processor Reference Guide - v14.1*, 2012, verfügbar über www.xilinx.com

[15] Xilinx Inc., *Zynq-7000 All Programmable SoC Overview - v1.2*, 2012, verfügbar über www.xilinx.com

[16] ARM Limited, *Cortex-A9 Technical Reference Manual - Revision: r4p1*, 2012, verfügbar über infocenter.arm.com

[17] OpenCores, *Wishbone B4 - SoC-Interconnection Architecture for Portable IP Cores*, 2010, verfügbar über cdn.opencores.com

[18] OpenCores, *About OpenCores*, verfügbar über www.opencores.org, Zugriff am 07.11.2012

[19] D. Lampret, *PWM/Timer/Counter - IP Core Specification - Rev. 0.1*, 2001, verfügbar über opencores.org

[20] W. Wolf, *A Decade of Hardware-Software Codesign*, 2003, verfügbar über ieeexplore.ieee.org

[21] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*, verfügbar über s-space.snu.ac.kr

[22] R. K. Gupta, G. Micheli, *Hardware-Software Cosynthesis for digital Systems*, 1993, verfügbar über si2.epfl.ch

[23] Mentor Graphics Corporation, *Seamless Hardware/Software Integration Environment - Datasheet*, 2005, verfügbar über www.mentor.com, Zugriff am 16.01.2013

[24] S. Ha, S. Kim, C. Lee, Y.Yi, S. Kwon, Y. Joo, *PeaCE: A Hardware-Software Codesign Environment for Multimedia Embedded Systems*, 2007, verfügbar über www.cin.ufpe.br

[25] Luca Formaggio, Franco Fummi, Graziano Pravadelli, *A Timing-Accurate HW/SW Co-simulation of an ISS with SystemC*, 2004, verfügbar über www.cs.york.ac.uk

[26] V. Iivojnovit, S. Pees, H. Meyr, *LISA - Machine Description Language and Generic Machine Model for HW/ SW CO-Design*, 1996, verfügbar über www-inst.cs.berkeley.edu

[27] T. Yeh, M. Chiang, *On the interface between QEMU and SystemC for hardware modeling*, 2010, verfügbar über ieeexplore.ieee.org

[28] Xilinx Inc., *Documentation of the Virtex-II Pro*, verfügbar über www.xilinx.com, Zugriff am 28.01.2013

[29] F. Slomka, M Dorfel, R. Munzenberger, R. Hofmann, *Hardware/Software Codesign and Rapid Prototyping of Embedded Systems*, 2000, verfügbar über contech.suv.ac.kr

[30] L. Séméria, A. Ghosh, *Methodology for Hardware/Software Co-verification in C/C++*, 2000, verfügbar über citeseerx.ist.psu.edu

[31] G. Heiser, *Secure embedded systems need microkernels*, 2005, verfügbar über ssrg.nicta.com.au

[32] J. H. Saltzer, Protection and the Control of Information Sharing in Multics, 1974, verfügbar über ftp.multicians.org

[33] H. Kwok-Hay So, *BORPH: An Operating System for FPGA-Based Reconfigurable*, 2007, verfgbar über users.nik.uni-obuda.hu

[34] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou1, G. Chinya1, R. Plate, T. Mattner, F. Olbrich, P. Hammar, R. Singhal, J. Brayton, S. Steibl, H. Wang, *Intel R Nehalem Processor Core Made FPGA Synthesizable*, 2010, verfügbar über www-inst.eecs.berkeley.edu

[35] A. Kulmala, E. Salminen, T. D. Hämäläinen, *Evaluating Large System-on-Chip on Multi-FPGA Platform*, 2007, verfügbar über www.martes-itea.org

[36] M. B. Gokhale, J. M. Stone, *NAPA C: Compiling for a Hybrid RISC FPGA Architecture*, 1998, verfügbar über wiki.ittc.ku.edu

[37] L. W. Nagel, D. O. Pederson, *SPICE - Simulation Program with Integrated Circuit Emphasis*, 1973, verfügbar über www.eecs.berkeley.edu

[38] S. A. Hyder, D. S. Kanth, C. Chandrasekhar, E. Sammaiah, *Field Programmable Gate Array Implementation Technology*, 2012, verfügbar über www.ijeat.org

[39] Calypto Design Systems Inc., *Catapult C – Ptoduct Overview and White Papers*, verfügbar über www.calypto.com, Zugriff am 08.03.2013

[40] S. Williams, *iCarus Verilog project page,* verfügbar über iverilog.icarus.com, Zugriff am 08.03.2013

[41] ISO/IEC, *C Programming language – international standard ISO/IEC 9899:TC3*, 2007, verfügbar über www.open-std.org

[42] Genode Labs, *Genode Release Notes 10.02*, verfügbar über www.genode.org, Zugriff am 09.03.2013

[43] Genode Labs, *Genode Timer-Session Interface*, verfügbar über www.github.com, Zugriff am 12.03.2013

Abbildungsverzeichnis

Abkürzungsverzeichnis

API.....**A**pplication **P**rogramming **I**nterface
ASIC.....**A**pplication-**S**pecific **I**ntegrated **C**ircuit
CPU.....**C**entral **P**rocessing **U**nit
DMA.....**D**irect **M**emory **A**ccess
FPGA.....**F**ield **P**rogrammable **G**ate **A**rray
GPLv2.....GNU **G**eneral **P**ublic **L**icense **v**ersion 2
HDL.....**H**ardware **D**escription **L**anguage
HLS.....**H**igh-**L**evel **S**ynthesis
IOMMU.....**I**nput/**O**utput **M**emory **M**apping **U**nit
IC.....**I**ntegrated **C**ircuit
IP.....**I**nstruction **P**ointer
IRQ.....**I**nterrupt **Req**uest
ISS.....**I**nstruction **S**et **S**imulator
LGPL.....GNU **L**esser **G**eneral **P**ublic **L**icense
MMIO.....**M**emory **M**apped **I**nput / **O**utput
MMU.....**M**emory **M**anagement **U**nit
NCR.....**N**earest **C**ommon **R**oot
OS.....**O**perating **S**ystem
RAM.....**R**andom-**A**ccess **M**emory
RM.....**R**egion **M**anager
RPC.....**R**emote **P**rocedure **C**all
SoC.....**S**ystem-**o**n-a-**C**hip
SP.....**S**tack **P**ointer
VDM.....**V**irtual **D**evice **M**onitor
VM.....**V**irtual **M**achine
VMSA.....**V**irtual **M**emory **S**ystem **A**rchitecture
XML.....E**x**tensible **M**arkup **L**anguage

Thesenpapier zur Diplomarbeit

Ausbau einer Umgebung für das Codesign von Hardware und Software

eingereicht von Martin Stein am 15. März 2013
bei Prof. Dr.-Ing. habil. Armin Zimmermann

Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung, Institut für Technische Informatik und Ingenieurinformatik, Fachgebiet System- und Software-Engineering

Codesign im Zielsystem

1. Die virtuellen Umgebungen, in die moderne Betriebssysteme ihre Anwendungen einsperren, können für das Codesign von Hardware und Software genutzt werden.

2. Aus Datenflussmodellen lassen sich Anwendungen erzeugen, die der Software in ihrer virtuellen Umgebung wie echte Geräte erscheinen.

3. Software kann somit in der Umgebung getestet werden, die bereits real existiert und zugleich die Geräte nutzen, die sich noch in der Entwicklung befinden.

4. Dem Geräteentwurf ist es währenddessen möglich, auf andere reale oder virtuelle Geräte zuzugreifen, ganz so, als wenn er bereits ein Teil der echten Hardware wäre.

Bedienbarkeit

6. Ein Entwickler muss für die Erzeugung und Einbindung eines virtuellen Geräts nur Einstellungen vornehmen, die von der Funktionsweise des Geräts und des Treibers abhängig sind.

Exklusivität

7. Es lassen sich beliebig viele Zustände eines virtuellen Geräts verwalten, die von verschiedenen Anwendungen gleichzeitig genutzt werden können.

Kosten-Accounting

8. Wenn eine Anwendung ein virtuelles Gerät nutzt, hat dies keinen Effekt auf unbeteiligte Anwendungen.

9. Ein virtuelles Gerät und sein Benutzer erlangen nicht mehr Macht oder Wissen über den jeweils anderen, als es die Geräteschnittstelle zulässt.

Beobachtbarkeit

10. Die Kommunikation zwischen einem virtuellem Gerät und einer anderen Anwendung kann der Entwickler ohne großen Aufwand abhören lassen.

11. Immer wenn eine Anwendung und ein virtuelles Gerät Informationen austauschen, kann der Entwickler beide solange er mag anhalten und analysieren.

Skalierbarkeit

12. Die Kosten eines Zugriffs auf ein virtuelles Gerät skalieren über der Anzahl gleichzeitiger Zugriffe.

Martin Stein, Dresden, den 20.03.2013

Selbstständigkeitserklärung

Die vorliegende Arbeit habe ich selbständig und ohne Benutzung anderer als der angegebenen Quellen angefertigt. Alle Stellen, die wörtlich oder sinngemäß aus ver- öffentlichten und nicht veröffentlichten Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

_____

Martin Stein, Dresden, den 20.03.2013