Maximilian Eibl, Martin Gaedke. (Hrsg.): INFORMATIK 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 493

Modellierung anwendungsspezifischer Hardware und deren Einbettung in die DBT-basierte Prozessor-Verhaltenssimulation

Steffen Köhler[1] and Rainer G. Spallek[1]

Abstract: Zur High-Performance Simulation von Prozessorarchitekturen und deren Modellierung auf Befehlssatzebene kommen häufig Verfahren zur Anwendung, die den auszuführenden Zielcode mittels Dynamic Binary Translation (DBT) direkt auf den Prozessor des Simulationshosts abbilden. Während für Standardoperationen eine gute Abbildbarkeit des Befehlssatzmodells auf den Prozessor des Simulationshosts gegeben ist, lassen sich Modelle komplexer Spezialbefehle oder Peripherie nur ineffizient durch die Befehlssatzarchitektur des Simulationshosts nachbilden.
In unserem Beitrag stellen wir die Modellintegration und Co-Simulation dieser anwendungsspezifischen Komponenten auf Basis von FPGA-Technologie vor. Als Hardwareplattform anwendungsspezifischer Funktionalität fungiert dabei ein FPGA-Accelerator, der mittels PCIe-Schnittstelle an das von uns entwickelte RUBICS-Simulationsframework (Retargetable Universal Binary Instruction Conversion Simulator) angebunden wird. RUBICS stellt ein flexibles Werkzeug zur Modellierung und Evaluation kompletter, eingebetteter Prozessorarchitekturen (SoC) auf einem hohen Abstraktionsniveau (Befehlsebene, IO-Verhalten) basierend auf der ECMA CLR-Plattform dar. Ausgehend von einem in einer dedizierten Architekturbeschreibungssprache (ADL) vorliegenden ARMv7 Struktur- und Verhaltensmodell wird die exemplarische Modellierung eines FPGA-basierten FFT-Prozessors als anwendungsspezifisches Peripheriegerät der ARMv7-Architektur aufgezeigt und analysiert.

Keywords: Processor Simulation; Dynamic Binary Translation; FPGA Accelerator

1 Einleitung

Die Möglichkeit zur Simulation von eingebetteten Prozessorarchitekturen bildet einen wichtigen Grundstein für den erfolgreichen Entwurf von SoC-Architekturen. Dabei spielt vor allem die Analyse von Wechselwirkungen zwischen Prozessorkernen und peripheren Komponenten eine entscheidende Rolle. Die aktuell häufig vorliegende Komplexität von Hardware- und Softwarekomponenten erfordert eine Analyse des Gesamtsystems zu einem möglichst frühen Zeitpunkt des Entwurfsprozesses. Im Gegensatz zur Emulation, bei der nur das peripher beobachtbare Verhalten von Softwarekomponenten einer Zielarchitektur von Interesse ist, ermöglicht die Simulation das Inspizieren des gesamten Modellzustandes. Zu den Anforderungen an einen geeigneten Simulator gehören vor allem eine einfache Architekturmodellierung auf einem hohen Abstraktionsniveau (Befehlssatz, Befehlsverhalten,

[1] Technische Universität Dresden, Institut für Technische Informatik, 01062 Dresden, Germany
{steffen.koehler,rainer.spallek}@tu-dresden.de

doi:10.18420/in2017_45

IO-Verhalten), eine detaillierte Beobachtbarkeit der internen Abläufe (Register, Datenpfade) sowie eine hohe Simulationsperformance.
Neben der Anwendung in der Anfangsphase des Entwurfsprozesses kann die Simulation darüber hinaus auch bei bereits produzierten Prozessoren genutzt werden, indem die Ausführung einer Softwarekomponente im simulierten und physischen Prozessor protokolliert (Trace) und miteinander verglichen wird.
Zur Simulation einer Prozessorarchitektur wird eine Modellbeschreibung benötigt, nach welcher der Simulator die Befehle der zu simulierenden Softwarekomponente der Zielarchitektur auf den Simulationshost abbilden und dort abarbeiten kann. Die Qualität der Übersetzung auf die Befehlssatzarchitektur des Simulationshosts bestimmt dabei maßgeblich die Simulationsperformance. Eine vielfach verwendete Methode zur schnellen Prozessorsimulation auf Verhaltensebene ist die Binary Translation unter Verwendung eines Just-In-Time-Compilers (JIT), welcher Befehlssequenzen der Zielarchitektur zur Laufzeit auf den Simulationshost abbildet, optimiert und ausführt.
Verschiedene Ansätze JIT-basierter Simulatoren und Emulatoren nutzen dabei verbreitete Laufzeitumgebungen, wie beispielsweise Java [Kau+11], Common Language Runtime (CLR) [Wri14] oder PyPy [LIB15], aber auch dedizierte Compiler wie Tiny Code Generator [Bel16]. Die erreichbare Qualität der Befehlsabbildung des simulierten Prozessormodells auf den Prozessor des Simulationshosts hängt dabei entscheidend vom Grad der Ähnlichkeit beider Architekturen ab. Neben der Befehlsabbildung ist auch die Modellabbildung von Prozessorperipherie von diesem Problem betroffen. Beide Teilprobleme lassen sich effizient nur durch Bereitstellung einer flexibleren Abbildungsplattform auf dem Simulationshost lösen. Ein möglicher Ansatz ist dabei die Integration von FPGA-Hardware, verbunden mit einer Partitionierung des Simulationsmodells.
Die günstigsten Voraussetzungen im Bezug auf einfache Modellierbarkeit und Erweiterbarkeit bietet dabei das Simulationsframework RUBICS [Koe+16]. Sowohl die Verfügbarkeit einer dedizierten Architekturbeschreibungssprache (ADL) als auch die Erweiterungsmöglichkeiten der verwendeten CLR-Plattform mittels dynamischer Laufzeitbibliotheken (DLL) erleichtern die Einbindung von FPGA-basierten Modellpartitionen in den Simulationsprozess.

2 Simulationsplattfrom

Die Basisplattform zur Untersuchung der Integration anwendungsspezifischer Hardware in die High-Performance Prozessor-Verhaltenssimulation bildet das bereits erwähnte RUBICS-Simulationsframework. Dieses stellt sowohl Beschreibungsmittel zur Struktur- und Verhaltensmodellierung in Form einer Architekturbeschreibungssprache (ADL), als auch Kommunikationsschnittstellen zur Einbindung externer Kern- und Peripheriemodelle sowie die Steuerung des Simulationsablaufes sowie Test- und Debug-Funktionalität bereit. Das Framework ist modular aufgebaut und besteht aus einer Kernkomponente, die durch dynamische Bibliotheken nahezu beliebig erweiterbar ist. Einzige Bedingung ist dabei die Unterstützung durch die im ECMA-335 Standard beschriebene Common Language

Runtime (CLR) [ECM06]. Neben der Verhaltensbeschreibung mittels einer durch die CLR unterstützten Sprache (z.B. C#) ist darüber hinaus auch ein direkter Zugriff auf native Systembibliotheken und damit auf nahezu alle relevanten Systemressourcen möglich. Dies erleichtert die Einbindung anwendungsspezifischer Hardware in den Simulationsprozess erheblich. Die CLR stellt die Basis des gesamten Frameworks dar und fungiert über den Umweg der Zwischendarstellung im CIL-Bytecode (Common Intermediate Language) auch als Backend für die Binary Translation. Die prinzipielle Struktur des RUBICS-Simulationsframeworks ist in Abbildung 1 dargestellt.

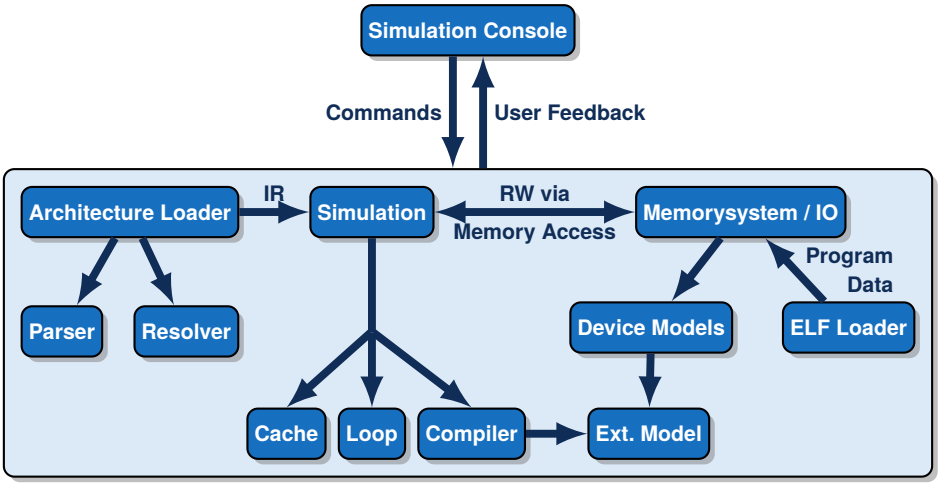


Abb. 1: Übersicht der Simulatorkomponenten

Zur Integration von FPGA-Hardware mit dem Ziel einer Steigerung der Simulationsperformance anwendungsspezifischer Modellkomponenten bietet die Simulationsumgebung in Anlehnung an die Erfordernisse das Prozessormodells folgende Möglichkeiten:

- Lose Kopplung durch generischen Bus-Schnittstelle für Peripherie- und Co-Prozessormodelle
- Enge Kopplung durch Plugin-Schnittstelle zur ADL-Modellinstrumentierung als Teil der Verhaltensbeschreibung

Obwohl sich beide Varianten prinzipiell für eine FPGA-Kopplung eignen, wurde die Plugin-Variante nicht näher untersucht. Als Hauptgrund für diese Herangehensweise muss die im praktischen Einsatz nicht zu vermeidende Kommunikationslatenz zwischen Simulationshost und FPGA unter Anwendung einer PCIe Schnittstelle angeführt werden.

3 Architekturmodellierung

Bereits in [Kau+11] wurde für die Verhaltensmodellierung von Prozessorarchitekturen eine Architekturbeschreibungssprache (ADL) vorgeschlagen. Dieses Konzept bietet gegenüber der Modellierung in einer Programmier- oder Hardware-Beschreibungssprache den Vorteil, dass der grundlegend identische strukturelle Grundaufbau einer jeden Prozessorarchitektur bereits vorgegeben wird. Die Modellierung beschränkt sich damit auf Prozessorzustand, Befehlssatz, Befehlsverhalten und Peripherieverhalten.
Die Architekturbeschreibung besteht dabei aus folgenden Elementen:

- `import <id>` Ausdrücke, welche externe Modelle wie Busgeräte (Devices) und Plugins importieren,
- `bus <id> {}` Schnittstellen für Speicher und Memory-Mapped Peripherie,
- `plugins {}` Plugin Schnittstellen für die Modellinstrumentierung,
- `context {}`, in welche der Maschinenkontext mit Hilfe von getypten Variablen deklariert wird,
- `decoder {}`, in welcher das Dekoderverhalten mittels Operationen definiert wird,
- `behavior {}`, in welcher der Verhaltensteil der dekodierten Befehle mittels Operationen definiert wird,
- `interrupts {}`, in welcher die Operationen definiert werden, die durch Interrupts initiiert werden.

In Abbildung 2 ist ein einfaches Architekturmodell mit sowohl eng und lose gekoppelter FPGA-Hardware beispielhaft dargestellt. Das Befehlsverhalten beschränkt sich auf die Inkrementierung des Befehlszählers und den Aufruf der Plugin-Funktion `ExecFunc()`.
Die Beschreibung beinhaltet sowohl Struktur als auch Verhalten der Zielarchitektur. Nach der Deklaration eines Speicherbusses `mem` mit eingebundenem FPGA-Gerätemodell `fpga_dev` können die unter `access` definierten Zugriffsarten in der weiteren Architekturbeschreibung genutzt werden, so zum Beispiel der Ausdruck `mem.byte[<address>]`. Die Angabe `unit` spezifiziert dabei die kleinste adressierbare Datenwortbreite des Speichers. Über die `fetch`-Umgebung wird festgelegt, von welchem Speicherbus (`mem`) und an welcher Adresse (`pc`) der nächste Befehl zu holen ist. Die Dekoderbeschreibung kann optional einen eigenen `context` enthalten, der die Modellierung komplexer Befehlssätze erleichtert.
Die ADL stellt die Standarddatentypen der CLR für Dekoder- und Verhaltensmodellierung zur Verfügung. Typekonvertierungen werden, falls diese eindeutig sind, automatisch durchgeführt. Zusatzfunktionen erlauben darüber hinaus eine manuelle Typkonvertierung sowie verschiedene Bit-Level- und Gleitkommaoperationen. Auf eine Typisierung von Festkomma-Literalen wurde bewusst verzichtet, da diese mittels Bereichsüberprüfung direkt in jeden geforderten Festkomma-Zieltyp konvertiert werden können.

Die Architekturbeschreibung wird nach dem Laden und Auflösen der Ausdrücke in eine interne Darstellung Intermediate Representation (IR) überführt, welche die Basis für den Simulationsablauf sowie die JIT-Compilierung darstellt.

```
1  bus mem {
2      unit = 8;
3      endian = LITTLE;
4      access {
5          byte = 1;
6          short = 2;
7          word = 4;
8      }
9      devices {
10         ram mem {
11             base = 0;
12             size = 0x40000000;
13         }
14         fpga fpga_dev {
15             base = 0x40000000;
16             size = 0x10000000;
17         }
18     }
19 }
20 plugins {
21     plugin.fpga fpga_plug;
22 }
23 context {
24     uint pc;
25 }
26 decoder {
27     fetch {
28         bus = mem;
29         pc;
30     }
31     context {
32         uint instruction_word;
33     }
34     operation oper {
35         instruction_word = fetch.word;
36         IPC;
37     }
38 }
39 behavior {
40     operation IPC {
41         fpga_plug.ExecFunc();
42         pc += 4;
43     }
```

Abb. 2: Architekturmodellierung mittels ADL

4 Simulationsablauf und Binary Translation

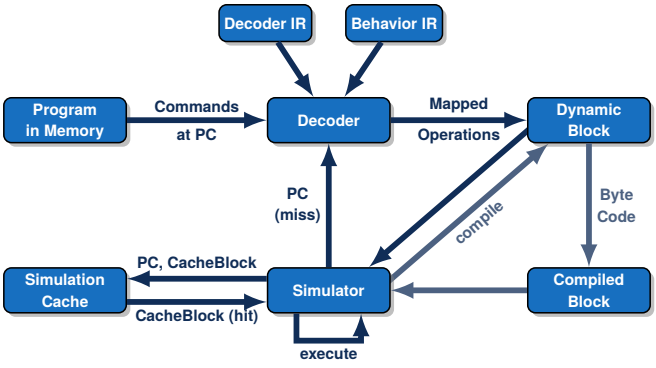


Abb. 3: Prinzipieller Simulationsablauf

Nachdem die Zielarchitektur sowie das zu simulierende Programm geladen und der Program Counter (PC) gesetzt sind, kann die Simulation gestartet werden. Mithilfe der aus der Architekturbeschreibung erstellten IR kann das Zielprogramm dekodiert und solange ausgeführt werden, bis der Wert des PC eine Abbruchbedingung (gesetzter Breakpoint) erreicht hat. Der schematische Ablauf der Simulation wird in Abbildung 3 gezeigt.

Der Ausführung des Befehlsverhaltens muss eine blockweise Dekodierung der auszuführenden Befehlssequenz vorausgehen, da der Simulationscache zunächst noch keine gültigen Einträge aufweist (Cache-Miss). Im weiteren Simulationsverlauf können dekodierte Befehlssequenzen bei einem Cache-Hit wiederverwendet und redundante Dekodieroperationen vermieden werden. Die Größe der dekodierten Befehlsblöcke (Dynamic Block) richtet sich nach der Länge des linearen Kontrollflusses im Zielcode. Ein Verlassen des linearen Kontrollflusses wird durch das Schlüsselwort `exit` in der Architekturbeschreibung beschrieben (Ende des Basisblocks).
Der Prozess der Binary Translaten erstellt in einem ersten Schritt eine Sequenz befehlsspezifischer Kopien der Verhaltensbeschreibung, welche anschließend einem mehrstufigen Optimierungsprozess zugeführt wird. Dies hat sich als vorteilhaft herausgestellt, da die CLR redundante Operationen sowie Mehrfachzugriffe auf Variablen nicht ausreichend optimieren kann. Die Amortisation des Zusatzaufwands führt lediglich bei sehr kurzen Simulationsläufen zu einem signifikanten Mehraufwand. Der direkten Übersetzung des Architekturverhaltens werden zusätzlich Schnittestellenaufrufe zu nativen Modellkomponenten hinzugefügt, welche direkt als CLR-Komponente eingebunden werden. Ein Lookup-basierter Memory-Delegator stellt Funktionen für eine effiziente Abbildung von Speicheradressen auf Peripheriemodellinstanzen bereit. Speicherblöcke des Zielsystemmodells werden unter Verwendung von Betriebssystemfunktionen direkt auf den virtuellen Adressraum des Simulationshosts abgebildet.
Im Verlauf der Simulation werden die dekodierten Basisblöcke zunächst interpretierend ausgeführt und nach Erreichen einer festgelegten Compile-Schwelle durch transformierten Bytecode der CLR ersetzt (Hot-Spot-Technik). Die weitere Umsetzung in native Instruktionen der Host-Architektur wird vom JIT-Compiler der CLR automatisiert durchgeführt.

5 Anwendungsspezifische Hardware

Neben der Beschreibung des Architekturmodells mittels ADL werden für die vollständige Nachbildung des Prozessorverhaltens entsprechende Peripherie- und Pluginmodelle benötigt. Eine einfache Integration in den Simulationsprozess ist dabei durch die Nutzungsmöglichkeit einer durch die CLR unterstützten Programmiersprache (C#, VB) gegeben. In Abhängigkeit des Kommunikationsverhaltens zwischen den einzelnen Modellkomponenten ist eine Abbildung von Peripherie- und Pluginmodellen oder deren Teilkomponenten auf FPGA-Hardware zur Simulationsbeschleunigung möglich. Voraussetzung ist eine Modellpartitionierung mit dem Ziel einer synthesefähigen Hardwarespezifikation der betreffenden Partition. Da die FPGA-Hardware nur mittels einer verfügbaren Schnittstelle in den Simulationshost integriert werden kann, ist die Auswahl der Partitionierung an die der verfügbaren Ressourcen zur Inter-Partitionskommunikation anzupassen. Für die überwiegende Mehrzahl relaventer Host-Plattformen kann externe FPGA-Hardware auf einfache Weise nur mittels PCIe-Schnittstelle genutzt werden, woduch eine effiziente Inter-Partitionskommunikation nur auf Basis eines Streaming-Ansatzes möglich ist.

Abbildung 4 zeigt die prinzipielle Einbettung von anwendungsspezifischen Modellen in den Simulationsprozess auf der Basis von FPGA-Hardware.

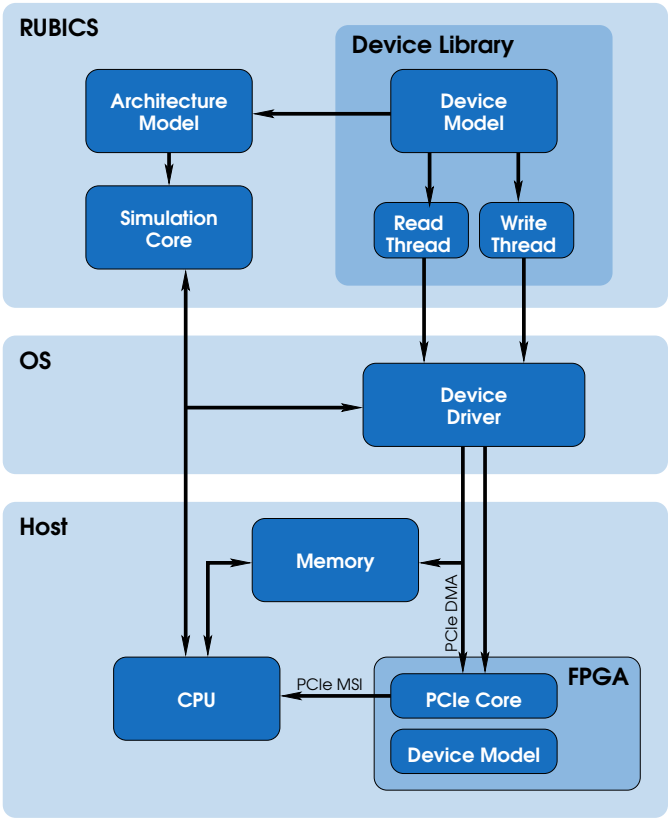
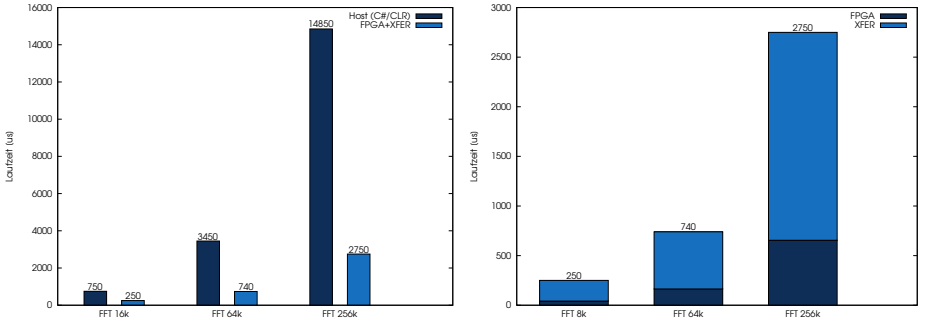


Abb. 4: Simulationsumgebung mit FPGA-Hardware

6 Leistungsbewertung

Zur Evaluation der Simulationsperformance von anwendungsspezifischer Hardwaremodellen auf Basis von FPGA-Hardware kam ein Hostssystem mit Intel Core-i5 6500 (SkyLake) und 16GB DDR4-Speicher zum Einsatz. Dieses wurde durch ein über vier PCIe-Lanes gekoppeltes FPGA-Board vom Typ Terasic DE5-Net (Altera/Intel Stratix V FPGA) [Ter17] ergänzt. Die Systemumgebung bildete Debian Linux/Kernel v4.10.1. Alle Performance-Messungen wurden mit einem ARMv7 Thumb2-Befehlssatzmodell [ARM10] unter RUBICS v0.5 und Mono CLR v4.8 durchgeführt. Als Benchmark fungierte ein exemplarisches, anwendungsspezifisches Decimation-In-Frequency FFT-Modell [Coo65] mit drei verschiedenen Blocklängen, welches sowohl als C#-Implementierung direkt in der CLR-Umgebung als auch auf dem gekoppelten FPGA simuliert wurde. Die Beschreibung der FPGA-Partition

erfolgte sowohl in Verilog HDL (RTL) als auch strukturell automatisiert durch das Generatortool QSys der Entwurfssoftware Quartus Prime v15.1. Die Kopplung zwischen FPGA und Host übernahme ein XILLYBUS PCIe-Streaming-Controller IP-Core [Xil17] inklusive Linux Kernel-Treiber. Obwohl das vorgegebene Optimierungsziel performanceorientiert war, konnte die mögliche FFT-Pipeline-Tiefe bedingt durch die vom XILLYBUS-Core vorgegebene Streaming-Taktfrequenz von 250MHz nicht komplett genutzt werden.
Abbildung 5 veranschaulicht die erreichten Simulationslaufzeiten mit und ohne FPGA unter Verwendung verschiedener FFT-Blockgrößen sowie die Aufteilung in Rechen- und Transferzeit. Die gemessenen Zeiten stellen Mittelwerte über 1000 FFT-Durchläufe dar, um Messungenauigkeiten aufgrund von JIT-Amortisationseffekten vorzubeugen. Der erreichte Beschleunigungsfaktor unter Berücksictigung des Transfer-Overheads liegt dabei zwischen 3 (FFT 16k) und 5,4 (FFT 256k).



(a) Host vs. FPGA (b) FPGA vs. Datentransfer

Abb. 5: Simulationslaufzeit in Abhängigkeit der FFT-Blocklänge

7 Zusammenfassung und Ausblick

In unserem Beitrag konnte wir Möglichkeiten der Modellintegration anwendungsspezifischer Hardware in des RUBICS-Simulationsframework und deren beschleunigte Simulation durch den Einsatz eines FPGAs im Simulationshost aufzeigen. Diese Heransgehensweise ist insbesondere sinnvoll für komplexe Verhaltensmodelle peripherer Komponenten in Prozessor-basierten SoC. Weiterhin wurde die Unterstützung externer Modelle durch die verwendete Architekturbeschreibungssprache (ADL) vorgestellt, die eine flexible Kommunikationsschnittstellen für eng (Plugins) und lose (Peripherie) gekoppelte anwendungsspezifischer Hardware bereitstellt. Im Rahmen der durchgeführten Untersuchungen konnte die Möglichkeit einer Simulationsbeschleunigung durch die Migration einer anwendungsspezifischen Modellpartition (FFT) auf einen im Simulationshost integrierten und über PCEe gekoppelten FPGA gezeigt werden. Der erreichbare Laufzeitvorteil ist dabei umso größer, je größer der Berechnungsvorteil des FPGA gegenüber dem Host-Prozessor und je kleiner der Transfer-Overhead ist. Das gewählte Beispiel der FFT erfüllt diese Voraussetzung nur

bedingt, sodass der einzelte Performance-Gewinn relativ niedrig ausfällt. Dies ist vor allem auf die hohe Kommunikationslatenz des auf Datendurchsatz optimierten XILLYBUS IP-Cores zurückzuführen. Die Erstellung entsprechender Partitionierungsvorgaben erscheint daher sinnvoll und notwendig. Ebenso wäre die Untersuchung eines auf geringe Kommunikationslatenz ausgerichteten PCIe-Core von Interesse, um eine enge Modellkopplung effizient realisieren zu können.

Literaturverzeichnis

[ARM10] ARM Ltd. ARMv7-M Architecture Reference Manual. ARM DDI 0403D, ID021310, 2010. http://www.arm.com

[Bel16] Fabrice Bellard. Tiny Code Generator (TCG README), 2016. http://wiki.qemu.org/Documentation/TCG/.

[Coo65] J. W. Cooley, J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, April 1965.

[ECM06] ECMA International. *ECMA-335: Common Language Infrastructure (CLI)*. Fourth edition, 2006.

[Kau+11] M. Kaufmann, M. Häsing, T. Preußer, and R. G. Spallek. The Java Virtual Machine in Retargetable, High-performance Instruction Set Simulation. In *Proc. 9th Int'l Conf. Principles and Practice of Programming in Java (PPPJ '11)*. ACM, 2011.

[Koe+16] S. Köhler, T. Frank, M. Häsing, and R. G. Spallek. RUBICS: Ein retargierbares Framework zur Modellierung von Prozessorarchitekturen auf der Basis von .NET CLR. In *Proc. Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS 2016)*. Fraunhofer Verlag, 2016.

[LIB15] D. Lockhart, B. Ilbeyi, and C. Batten. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing JIT compilers. In *IEEE Int'l. Symp. Performance Analysis of Systems and Software (ISPASS)*, March 2015.

[Ter17] Terasic Inc. DE5-Net User Manual V1.04. March 21. 2017. http://www.terasic.com.tw

[Wri14] Patrick Andrew Wright. Dynamic Binary Translation on the .NET Platform. Master's thesis, 2014.

[Xil17] Xillybus Ltd. An FPGA IP core for easy DMA over PCIe, Haifa, Israel, 2017. http://xillybus.com