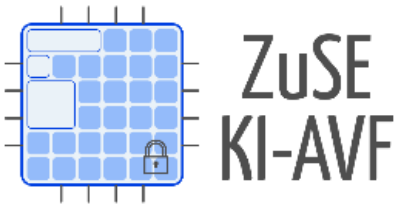

ZuSE-KI-AVF

Anwendungsspezifischer KI-Prozessor für die intelligente Sensorsignalverarbeitung im autonomen Fahren

Abschlussbericht

Cadence Design Systems GmbH

GEFÖRDERT VOM

Zuwendungsempfänger: Cadence Design Systems GmbH
Förderkennzeichen: 16ME0060K
Vorhabenbezeichnung: Anwendungsspezifischer KI-Prozessor für die intelligente Sensorsignalverarbeitung im autonomen Fahren – ZuSE-KI-AVF
Laufzeit des Vorhabens: 01.10.2020 – 30.06.2024

Inhalt

I. Kurzdarstellung

1. Aufgabenstellung

Im Rahmen dieses Projektes sollte eine konfigurierbare, massiv-parallele Prozessorarchitektur als Open-Source IP-Core Komponente erarbeitet werden, die sich für Anwendungen der künstlichen Intelligenz im Bereich Advanced Driver Assistance Systems (ADAS) eignet. Hieraus ergibt sich eine Flexibilität, die Prozessorarchitektur sowohl in ASIC- als auch in FPGA-basierten System-on-Chips (SoCs) als Embedded IP-Core Komponente einzusetzen. Eine optimale Abbildung der Architektur auf verschiedenen Plattformen ist dabei von besonderer Bedeutung, um die durch die Technologie bereitgestellte maximale Taktfrequenz und damit den höchsten Datendurchsatz zu erreichen.

Zahlreiche in ADAS-Anwendungen genutzte Sensorsysteme wie RADAR, LiDAR oder Kameras liefern hochauflösende, mehrdimensionale Messdaten, deren Verarbeitung anwendungsbedingt in Echtzeit erfolgen muss. Um in einem Fahrszenario aus dieser Datenmenge komplexe und sinnvolle Schlussfolgerungen zu extrahieren, haben Methoden des Deep Learning (DL) große Erfolge erzielt und sind somit in modernen Fahrerassistenzsystemen unerlässlich. Dabei stellen einerseits hohe Datenmengen aus einer Vielzahl von Sensoren und andererseits die massive Rechenleistungsanforderung speziell bei der Ausführung von KI-Algorithmen (z.B. Convolutional Neural Networks - CNNs) große Herausforderungen an die zugrundeliegende Hard- und Software dar. Weitere anwendungsbezogene Aspekte bilden außerdem die Energieeffizienz, Robustheit, Flexibilität, die Bereitstellung von funktionaler Sicherheit sowie die IP-Security. Demzufolge bedarf es eines konfigurierbaren KI-Prozessors, um diese Anforderungen zu erfüllen.

Die Klasse der KI-Anwendungen bietet ein hohes Maß an Datenparallelität und besteht in Bezug auf z.B. CNNs aus einer extrem hohen Anzahl an Faltungsoperationen. Die zu erstellende Prozessorarchitektur basiert auf einer in vorangegangen Arbeiten entwickelten Vektorprozessor-Architektur ([6]). Die ideale Eignung für sowohl Computer Vision (CV) als auch speziell KI-Anwendungen und das Erreichen von hohen Taktfrequenzen auf FPGAs konnte dabei schon gezeigt werden (ca. 400 MHz auf einem Xilinx Virtex-6 FPGA). Die Vektorprozessorarchitektur hebt sich in besonderem Maße durch die auf vertikalen Vektoren basierte Datenverarbeitung und einen komplexen Adressierungsmodus, mit dem eine 2D-Faltung in nur einer Instruktion kodiert werden kann, von bestehenden Architekturen wie z.B. mobilen GPUs ab. Wesentlich für einen hohen Datendurchsatz bei komplexen Anwendungen ist die Anbindung an den Speicher über einen effizienten Speichercontroller (DMA), welcher innerhalb des Projektes erarbeitet wurde.

Durch die beiden Aspekte Skalierbarkeit und Konfigurierbarkeit kann die Prozessorarchitektur auf eine Anwendung und eine zur Verfügung stehende Hardware optimiert werden. Dadurch wird der flexible Einsatz als Embedded IP-Core in kommerziellen SoCs ermöglicht.

2. Voraussetzungen, unter denen das Vorhaben durchgeführt wurde

Eine der wichtigsten Voraussetzungen für das autonome Fahren ist die Interpretation und das Verstehen der aktuellen Umgebung. Aufgaben wie semantische Segmentierung der Umgebung (Einteilung in die Klassen Straße, Himmel, Auto, Fußgänger etc.) oder Objektklassifizierung sind dabei fundamental. Dabei steht am Anfang zunächst das Wahrnehmen der Umgebung. Um funktionale Sicherheit zu gewährleisten, gilt es, eine Vielfalt an Sensoren bereitzustellen und durch Redundanz eine möglichst große Ausfallsicherheit des Systems zu garantieren. Neben passiven Sensoren wie Kameras am Auto kommen

aktive Sensoren wie RADAR und LiDAR zum Einsatz, die ihrerseits Signale in die Umgebung aussenden und die zurückgestreuten Reflektionen aufnehmen. Somit erhält man eine große Menge an mehrdimensionalen Messdaten, aus denen dann im autonom fahrenden Auto Schlussfolgerungen über die Fahrzeugumgebung und die vorherrschende Verkehrssituation gezogen werden, worauf basierend dann z.B. Steuerungsentscheidungen oder andere Fahrzeugaktionen zu treffen sind.

Durch die Verwendung von künstlichen neuronalen Netzen (z.B. CNNs) sind in den letzten Jahren große algorithmische Fortschritte im Bereich der Interpretation der Umgebungsdaten erzielt worden, sodass sie zunehmend in der Forschung und Vorentwicklung realer Fahrerassistenzsysteme zum Einsatz kommen. Diese Klasse von Algorithmen erfordert eine sehr hohe, auch in Zukunft immer weiter steigende Rechenleistung, da die verwendeten CNNs für sinkende Fehlerraten in ihrer Topologie immer komplexer werden (d.h. mehr Schichten und Neuronen erforderlich sind). Das Verlustleistungsbudget für einen dafür verwendeten System-on-Chip (SoC) ist im Auto sehr begrenzt (wenige Watt). Daher müssen neuartige Hardwarearchitekturen entwickelt werden, die in der Lage sind, die geforderte Rechenleistung bei gleichzeitig niedriger Verlustleistung bereitzustellen.

Neben der reinen Rechenleistung und Energieeffizienz spielt auch die Flexibilität der Hardware-Architektur eine große Rolle. Zum einen haben unterschiedliche Sensoren verschiedene Arten von Nachverarbeitungscharakteristiken zur Folge. Zum anderen unterliegen die verwendeten CNNs einer stetigen Weiterentwicklung. Im Bereich der Kamerasignalverarbeitung muss mit Störeinflüssen wie Bildrauschen bei Dunkelheit, Überbelichtung bei Gegenlicht oder Verzerrungen bei Regen umgegangen werden. Ein robustes CNN für derartige Aufgaben ist Teil des algorithmischen Entwicklungsprozesses. Dabei kann ein großes Netz in seiner Struktur (Topologie, Bitbreite der Berechnungen) unter Verringerung des Rechenleistungs- und Energiebedarfs verkleinert werden, bis die entsprechende Genauigkeit bzw. Fehlerrate des Netzes ein akzeptables unteres Limit gerade noch erreicht (Abtausch Energie/Rechenleistung/Fehlerrate). Ein zur Verfügung stehendes SoC Konzept muss daher flexibel sein und fortlaufend die sich in neuen Algorithmen ergebenden Veränderungen abbilden können.

In diesem Projekt wurde eine konfigurierbare massiv-parallele Vektorprozessorarchitektur erarbeitet, die in der Lage ist, hohe Rechenleistung als Embedded IP-Core-Komponente für FPGA- und ASIC-basierte Systeme bereitzustellen. Dabei diente eine in vorangegangen Arbeiten entwickelte, vektorbasierte Prozessorarchitektur als Grundlage.

Expertise / Vorarbeiten

Cadence Design Systems GmbH (CDN) in Feldkirchen bei München ist die deutsche Tochter der Cadence Design Systems, Inc., einem weltweit führenden Anbieter von Software für Electronic Design Automation. Cadence Deutschland beschäftigt ca. 160 Mitarbeiter, davon etwa 80 Ingenieure. Innerhalb des Gesamtkonzerns trägt Cadence Deutschland zur Entwicklung neuer Methodiken, Werkzeuge und Dienstleistungen bei. Diese Rolle ermöglicht nicht nur enge Kooperationen in Forschungsprojekten und den anschließenden Transfer von Ergebnissen, sie bietet auch die Möglichkeit für firmen-, technologie- oder anwendungsspezifische Erweiterungen zum Stand der Technik. In Erwartung einer steigenden Bedeutung von System-Level-Design in der Industrie engagiert sich Cadence in der Entwicklung neuartiger Technologien, Methodiken und Softwarewerkzeuge. In den vergangenen Jahren hat Cadence Deutschland aktiv an mehreren Forschungsprojekten teilgenommen. In diesen Projekten lag der Fokus auf Methodik-Themen im Allgemeinen, sowie im Bereich Design-Flow, Tool und Technologie-Setup und der Anwendung von neuartigen Lösungen, u.a. auch im Bereich Virtual Prototyping. In diesem Projekt hat CDN den Virtuellen Prototypen für die Projektpartner entwickelt. Gegenüber einem herkömmlichen Virtuellen

Prototyp sollte der in diesem Projekt entwickelte Virtuelle Prototyp um Funktionen erweitert werden, die ein effizientes “SW Hardening” ermöglichen (AP 4).

CDN war zuletzt u.a. in den Verbundprojekten SmartHeaP und MORFEUS tätig und ist derzeit noch in die Projekte EVENTS und Diva-IC involviert. In allen Projekten entwickelt CDN Virtuelle Prototypen, die den jeweiligen Projektpartnern für die Durchführung ihrer Arbeiten zur Verfügung gestellt werden. Während die Abbildung des jeweiligen zu entwickelnden Systems in den Virtuellen Prototypen im Wesentlichen eine notwendige Dienstleistung für die Projektpartner darstellt, werden und wurden in den oben genannten Verbundprojekten unterschiedliche Aspekte der Entwicklung von Virtuellen Prototypen untersucht, die sich von den für dieses Projekt vorgeschlagenen Aspekten unterscheiden. Im SmartHeaP Projekt wurde der Zusammenhang zwischen der Simulationsgeschwindigkeit und der Genauigkeit der erzielten Ergebnisse, z.B. in Hinblick auf das zeitliche Verhalten oder den abgeschätzten Energiebedarf, näher untersucht, so dass in Zukunft zu Beginn der Modellierungsarbeiten gezielter ein geeigneter Modellierungsansatz und Detailgrad gewählt werden können. Im Verbundprojekt MORFEUS wurden Techniken zur Fehlerinjektion in Virtuellen Prototypen und zur verteilen Simulation von Virtuellen Prototypen entwickelt.

3. Planung und Ablauf des Vorhabens

In diesem Abschnitt wird der Arbeitsplan, der im Rahmen des ZuSE-KI-AVF Projekts bearbeitet wurde, vorgestellt. Das Projektvorhaben wurde vom Institut für Mikroelektronische Systeme (IMS) koordiniert. Der Plan besteht aus 8 Arbeitspaketen, denen jeweils ein verantwortlicher AP-Leiter, sowie ein Stellvertreter zugewiesen wurde. Die geplante Gesamtlaufzeit betrug 3 Jahre und wurde kostenneutral um ein weiteres Jahr verlängert. Im Folgenden werden die Arbeiten der Partner aufgezeigt. In Abbildung 1 ist die Arbeitspaketstruktur mit den Abhängigkeiten und ihrem zeitlichen Ablauf dargestellt. Im Rahmen des Projekts fanden regelmäßige Workshops mit allen Projektpartnern statt, um den Informationsaustausch untereinander zu gewährleisten. Die Treffen dienten dem Austausch der Partner und sollten sich an den Meilensteinen orientieren. Zusätzlich zu diesen AP-übergreifenden Treffen gab es mit den jeweils beteiligten Projektpartnern eines Arbeitspakets regelmäßige Treffen, um die Bearbeitung zu synchronisieren. In der finalen Phase des Projekts (2021/2022) wurden die Ergebnisse zu “Trends in modern ASIC design” öffentlich präsentiert und z.B. auf dem Tensilica Day, der einmal Jahr am Institut für Mikroelektronische Systeme veranstaltet wird, vorgestellt.

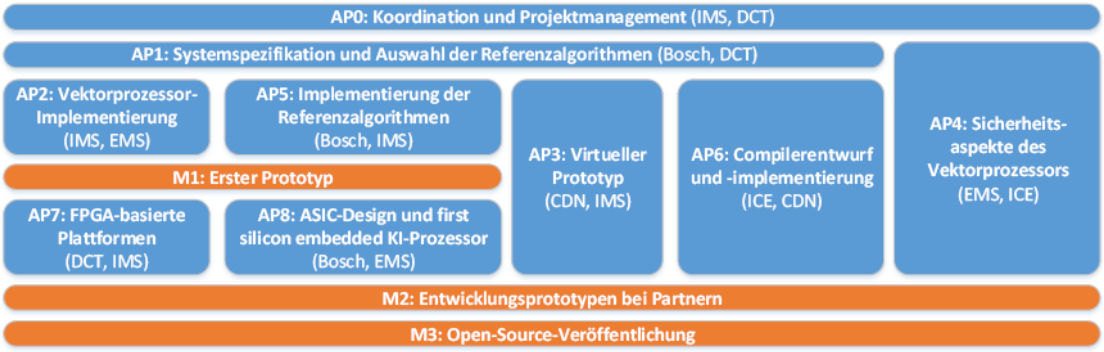

Abbildung 1: Arbeitspaketstruktur und Meilensteine im zeitlichen Ablauf

Arbeitspakete

AP 0: Koordination und Projektmanagement: Koordination des Projektes und alle notwendigen Maßnahmen zur Erreichung der Ziele.
AP 1: Systemspezifikation und Auswahl der Referenzalgorithmen: Definition der Anforderungen an die Zielplattform, z.B. Speicherbedarf und Übertragungsbandbreite, Rechenleistung (Operationen pro Sekunde), etc., auf Basis ausgewählter KI-Referenzalgorithmen und dedizierter Hardware-Akzeleratoren für die semantische Segmentierung von Kamerabildern und RADAR- bzw. LiDAR-Punktwolken.
AP 2: Vektorprozessor-Implementierung: Implementierung der spezifizierten Vektorarchitektur in der Hardware- Beschreibungssprache VHDL (IMS) sowie Implementierung der benötigten Speicherinterfaces und -controller (EMS).
AP 3: Virtueller Prototyp: Aufbau eines virtuellen Prototyping-Frameworks mit Prozessor-, Speicher- und Busmodellen sowie typischer Peripherie in SystemC/TLM (CDN). Der Virtuelle Prototyp ist das zentrale Werkzeug, um die Leistungsfähigkeit der Vektorprozessorarchitektur im Systemkontext zu vermessen und in Zusammenarbeit mit AP1 und AP5 zu optimieren.
AP 4: Sicherheitsaspekte des Vektorprozessors: Analyse und Implementierung von Verschlüsselungstechniken in der Vektorprozessorarchitektur und die Fehlererkennung auf Softwareebene: (4.a) symmetrische Verschlüsselungen im Speichercontroller, (4.b) Logikverschlüsselungen auf Netzlistenebene und (4.c) Software-Hardening.
AP 5: Implementierung der Referenzalgorithmen: Implementierung der Algorithmen als Referenzcode (z.B. C-Code, MATLAB oder Python) und Training der Modelle. Die SaFAD Richtlinien bilden bei der funktionalen Sicherheit (ASIL) und nominalen Systemleistung (SOTIF) den einfassenden Rahmen zur Verifizierung und Validierung
AP 6: Compilerentwurf und -implementierung: Zur effizienten Ausnutzung des Instruktionssatzes der Vektorprozessorarchitektur wird eine Programmiersprache konzipiert, die es ermöglicht, Algorithmen direkt auf Architektureigenschaften, wie z.B. komplexe Adressierungsmodi und Quantisierungseigenschaften abzubilden.
AP 7: FPGA-basierte Plattformen: Effiziente Abbildung der Prozessorimplementierung aus AP2 auf mehreren FPGA- Architekturen (Xilinx, Altera/Intel) durch IMS. Für das resultierende Vektorprozessor-IP (Intellectual Property) wird zusammen mit einer Bus-Infrastruktur, z.B. AXI, der Speichercontroller (EMS) auf FPGA-basierten Plattformen integriert.
AP 8: ASIC-Design und „first-silicon-embedded" KI-Prozessor: Entwurf eines ASIC-Designs aus dem vollständigen und optimierten Vektorprozessor-IP, welches in einer 22nm FDSOI Technologie (GlobalFoundries) gefertigt wird.

Zeitplanung und Meilensteine

Im Projekt wurden zwölf Meilensteine definiert Daraus ergab sich die in der folgenden Tabelle dargestellte Meilensteinplanung über die Projektlaufzeit:

Monat	AP	Meilenstein
15	2/5	M1: Mit dem Ende von AP 2 & AP 5 sind die Anwendungsfälle für Kamera, LiDAR und RADAR Anwendungen definiert, Referenzalgorithmen, Bewertungskriterien und Implementierungskonzepte für die gewählten Use Cases fertiggestellt und Anforderungen an die Zielplattform definiert. Ein Systemprototyp (FPGA) ist vorhanden und zeigt die Bearbeitung der implementierten Algorithmen in einer Basis-Konfiguration.
30	5/7	M2: Im Anschluss an AP 5 und AP7 ist eine FPGA-basierte Plattformen bei den Partnern DCT, Bosch und IMS implementiert, die den Vektorprozessor skalierbar realisiert und mit Verwendung geeigneter Peripherie die Algorithmen ausführt.
33	alle	M3: Im Anschluss an AP 5 und AP7 ist eine FPGA-basierte Plattformen bei den Partnern DCT, Bosch und IMS implementiert, die den Vektorprozessor skalierbar realisiert und mit Verwendung geeigneter Peripherie die Algorithmen ausführt.

Um den zeitlichen Verlauf einzuhalten wurde der folgende Projektablauf eingeplant:

	Arbeitspakete (AP)	1. Jahr				2. Jahr				3. Jahr				PM
		I	II	III	IV	I	II	III	IV	I	II	III	IV	ges.
0	Koordination und Projektmanagement													0
1	Systemspezifikation und Auswahl der Referenzalgorithmen													35
2	Vektorprozessor-Implementierung													49
3	Virtueller Prototyp													18
4	Sicherheitsaspekte des Vektorprozessors													45
5	Implementierung der Referenzalgorithmen													77
6	Compilerentwurf und -implementierung													49
7	FPGA-basierte Plattformen													194
8	ASIC-Design und first silicon embedded KI-Prozessor													103
	Meilensteine					M1						M2	M3	

Tabelle 1: Balkendiagramm des ursprünglichen Projektes

Teilvorhaben: Erstellung eines Virtuellen Prototyps des Vektorprozessorsystems für KI-basierte ADAS-Anwendungen und Entwicklung einer Methode zur Erkennung von Speicherfehlern in Software mit Hilfe des Virtuellen Prototyps.

In den folgenden Abschnitten sind die Beschreibungen der Arbeitspakete aus der Teilvorhabensbeschreibung noch einmal als Referenz eingefügt.

AP 1 - Systemspezifikation und Auswahl der Referenzalgorithmen:

Das System des Vektorprozessors wird spezifiziert und der Basisfunktionsumfang der Architektur wird definiert. Der Ressourcenbedarf auf FPGAs soll skalierbar auf verschiedene FPGA-Plattformen abgebildet werden. Die Spezifikation geeigneter Plattformen zur Implementierung findet dabei unter Berücksichtigung der verfügbaren Logikelemente und dedizierten Komponenten statt. CDN wird hierbei seine Erfahrungen aus dem Entwurf komplexer System-on-Chips (SoCs) einbringen sowie ggf. Abschätzungen für verschiedene Architekturalternativen mit Hilfe erster Teile des Virtuellen Protoyps durchführen.

AP 3 - Virtueller Prototyp:

In diesem Arbeitspaket wird ein Virtueller Prototyp entwickelt, der das gesamte System inklusive Prozessor-, Speicher- und Busmodellen sowie typischer Peripherie in SystemC/TLM modelliert. Der Virtuelle Prototyp erlaubt eine frühe Softwareentwicklung und die Abschätzung der Systemperformanz. Der Virtuelle Prototyp ist das zentrale Werkzeug, um die Leistungsfähigkeit der Vektorprozessorarchitektur im Systemkontext zu vermessen und in Zusammenarbeit mit AP1 und AP5 zu optimieren. CDN kooperiert dabei mit IMS bei der Integration des Modells der in AP 2 entwickelten Vektorprozessorarchitektur in den Virtuellen Prototypen. Das Modell des Vektorprozessors wird von IMS in enger Abstimmung mit CDN ebenfalls in AP 2 entwickelt. Das Simulationsmodell für den Speichercontroller wird von EMS entwickelt und geliefert. Bevor der Virtuelle Prototyp integriert werden kann, muss die Systemspezifikation einen gewissen Reifegrad erreicht haben, um zusätzliche Arbeiten durch eine übermäßige Anzahl an Änderungen zu vermeiden und die Infrastrukturelemente gezielt auswählen bzw. entwickeln zu können. Hierbei ist keine finale Systemspezifikation notwendig, da der Virtuelle Prototyp auch dazu genutzt werden soll, die Systemarchitektur weiter zu optimieren. Dieses Arbeitspaket unterteilt sich aus Sicht von CDN in zwei Teile. In einem ersten Teil wird ein funktionaler Virtueller Prototoyp entwickelt und im Projektverlauf gewartet, während in einem zweiten Teil ein zeitgenauer Virtueller Prototyp entwickelt wird. Für den funktionalen Virtuellen Prototyp soll die sogenannte “Loosely-Timed Transaction Level Modelling” (LT TLM) Methode angewendet werden. Diese Methode hat für das Projekt zwei entscheidende Vorteile. Da bei dieser Methode weitgehend vom zeitlichen Verhalten abstrahiert wird, sind die entsprechenden Modelle einfacher zu erstellen und somit liegt zu einem frühen Zeitpunkt bereits ein Virtueller Prototyp vor, der alle wichtigen Funktionen modelliert. Dieser Virtuelle Prototyp kann dann bereits in einer frühen Projektphase für die Softwareentwicklung verwendet werden. Aufgrund der vereinfachten Modelle erreicht ein mit LT TLM implementierter Virtueller Prototyp sehr hohe Simulationsgeschwindigkeiten, so dass auch komplexe Software auf dieser Art eines Virtuellen Prototyps entwickelt und getestet werden kann. Ein besonderes Augenmerk wird in dieser Phase auf die vom Virtuellen Prototyp zur Verfügung gestellten Debugging-Möglichkeiten des Vektorprozessors gelegt. Hierzu ist eine enge Zusammenarbeit mit IMS (Simulationsmodell des Vektorprozessors) geplant. Der in diesem Arbeitspaket entwickelte Virtuelle Prototypen soll über den Projektverlauf hin gepflegt werden, d.h. evtl. Änderungen oder Optimierungen im System sollen im Virtuellen Prototyp widergespiegelt werden.

In einer zweiten Phase der Entwicklung des Virtuellen Prototyps soll die “Approximately-Timed TLM” (AT TLM) Methode zum Einsatz kommen. Hierbei wird das zeitliche Verhalten der verschiedenen Module im System auf Transaktionsebene zeitlich genau modelliert. Mit einem AT modelliertem Virtuellen Prototyp ist es möglich, die Performanz des Systems genau zu untersuchen und zu optimieren. Da die AT-Modellierung im Vergleich zur LT-Modellierung deutlich aufwändiger ist, wird der AT basierte Virtuelle Prototyp erst nach dem LT Virtuellen Prototyp zur Verfügung stehen und auch eine geringere Simulationsgeschwindigkeit haben. Von daher macht es Sinn, in dem Projekt beide Modellierungsarten zu unterstützen. Auch der AT basierte Virtuelle Prototyp soll nach seiner Fertigstellung über die Projektlaufzeit hinweg gepflegt werden.

AP 5 Sicherheitsaspekte des Vektorprozessors:

Im Teilarbeitspaket des Software-Hardenings sollen Verfahren zur Erkennung von Fehlern bei der Speichernutzung und Speicherverwaltung in Software so implementiert werden, dass die Software nicht verändert werden muss und damit ihr zeitliches Verhalten unverändert bleibt. Es sollen Fehler erkannt werden wie Speicherlecks, das Lesen von nicht initialisiertem Speicher sowie das Schreiben auf nicht allozierten Speicher. Die anzuwendenden Verfahren, um diese Art von Fehlern zu erkennen, sind bekannt und z.B. in den Werkzeugen Valgrind [4], MemorySanitizer [5], AddressSanitizer [3] und LeakSanitizer implementiert und dokumentiert. Nach dem Stand der Technik werden die Analyseverfahren entweder direkt in den auszuführenden Code eingebaut oder die notwendigen Analyseschritte werden während der Ausführung in den Programmablauf injiziert (Valgrind), so dass die Ausführung des zu untersuchenden Programms verändert wird. In diesem Teilarbeitspaket soll der Virtuelle Prototyp so erweitert werden, dass er die Analyseverfahren auf dem Host, auf dem der Virtuelle Prototyp läuft, in der Regel ein leistungsfähiger Rechner, ausführen kann, während die zu analysierende Software, die innerhalb des Virtuellen Prototyps ausgeführt wird, 9 unverändert bleibt. Dies ist wichtig aus drei Gründen: Erstens wird erwartet, dass auf dem zu entwickelnden KI-Chip in erster Linie echtzeitfähige Software laufen soll. Hierbei ist es hilfreich, das zeitliche Verhalten nicht durch Analysewerkzeuge zu verändern. Zweitens sind in eingebetteten Systemen, wie sie in ADAS-Anwendungen zu finden sind, die zur Verfügung stehenden Ressourcen (Rechenleistung, Speicher) begrenzt, so dass auf diesen Systemen häufig die oben genannten Werkzeuge nur beschränkt oder gar nicht eingesetzt werden können. Drittens ist zu erwarten, dass die Analysewerkzeuge außerhalb der Simulation erheblich schneller laufen als innerhalb. Da der Virtuelle Prototyp prinzipiell Zugriff auf alle Werte innerhalb seiner Simulation erlaubt, sollten alle für die Analysewerkzeuge notwendigen Daten zugreifbar sein. Die Verfahren sollen in einem zweiten Schritt bezüglich ihrer Ausführungseffizienz untersucht und ggf. optimiert werden. Der Virtuelle Prototyp mit den Erweiterungen soll den Projektpartnern zur Verfügung gestellt werden, um die im Rahmen dieses Projekts zu entwickelnde bzw. einzusetzende Software auf die oben genannten Fehler überprüfen lassen zu können.

AP 6 – Compilerentwurf und Implementierung:

Die Spezifikation der Debug Information wird in Zusammenarbeit mit AP3 (Virtueller Prototyp) und AP4 (Sicherheitsaspekte des Vektorprozessors) durchgeführt, um eine optimale Nutzung des Virtuellen Prototyps zu ermöglichen. Es sollen Debugging Informationen erzeugt werden, die sowohl im Virtuellen Prototypen als auch auf der späteren Hardware eine bestmögliche Unterstützung beim Debuggen der Software liefern. Daneben soll untersucht werden, inwieweit Wissen des Compilers zur Optimierung des Software Hardenings genutzt werden kann und ob und wie dieses Wissen in die Debugging Informationen gepackt werden sollte.

	Arbeitspakete (AP)	1. Jahr				2. Jahr				3. Jahr				4 Jahr			PM (CDS)
		I	II	III	IV	I	II	III	IV	I	II	III	IV	I	II	III	ges.
0	Koordination und Projektmanagement																
1	Systemspezifikation und Auswahl der Referenzalgorithmen																2
2	Vektorprozessor-Implementierung																
3	Virtueller Prototyp																16
4	Security- und Safety-Aspekte des Vektorprozessors																15
5	Implementierung der Referenzalgorithmen																
6	Compilerentwurf und -implementierung																3
7	FPGA-basierte Plattformen																
8	ASIC-Design und first silicon embedded KI-Prozessor																
	Meilensteine					M 1						M 2	M 3				36

Tabelle 3: Balkendiagramm der APs mit Arbeitsaufwand von Cadence, orangener Bereich beschreibt die Verlängerung

Patentlage

Eine Patentrecherche nach relevanten Patenten, die das Vorhaben gefährden könnten, war ergebnislos. Dem ZuSE-KI-AVF Konsortium sind auch keine anderen dem Vorhaben entgegenstehenden Patente bekannt, so dass das Konsortium davon ausging, dass das Vorhaben durchgeführt werden konnte. Alle innovativen Ideen wurden im Laufe des Projekts sorgfältig auf Patentierbarkeit geprüft. Die entwickelte Software ist durch Urheberrechte geschützt und steht daher für die kommerzielle Verwertung durch die Projektpartner zur Verfügung.

4. Zusammenarbeit mit Anderen

Zur Erreichung der Ziele des Projektes war eine intensive Zusammenarbeit in jeder Phase des Projektes mit den Projektpartnern notwendig. Besonders wichtig ist die stete Synchronisation mit allen Partnern, die für den Aufbau und die Spezifikation des Gesamtsystems verantwortlich sind. Hieraus bildet sich im späteren Verlauf der Virtuelle Prototyp ab, der ein Ebenbild des späteren Systems ist. Gerade durch die Wiederverwendung des bereits existierenden ISS des Vektorprozessors, ist hier eine gute Abstimmung notwendig.

II. Eingehende Darstellung

1. Verwendung der Zuwendung und der erzielten Ergebnisse

Für die Durchführung des Projektes und die Erreichung der Ziele wurden, insbesondere Personalmittel benötigt. Darüber hinaus wurden im Rahmen des Projektes Konferenz-Veröffentlichungen geschrieben, sodass zusätzlich zu den Reisekosten für regelmäßige, projektinterne Workshops auch Reisen zu Fachkonferenzen aus Projektmitteln bezahlt wurden. Im Einzelnen wurden von den aus dem Projekt bezahlten Mitarbeitern die folgenden Projektergebnisse erzielt:

2. Ergebnisse in Kooperation mit den Projektpartnern

AP1 - Systemspezifikation und Auswahl der Referenzalgorithmen

Im Zuge des Arbeitspaketes 1 hat Cadence zum Beginn des Projektes bei der Erarbeitung einer vorläufigen Systemspezifikation mitgewirkt. Eine solche Spezifikation ist gerade in Hinsicht auf die Entwicklung eines Virtuellen Prototypen besonders wichtig, da dieser zeitlich weit vor der eigentlichen Hardware erstellt werden soll. Konkrete Fragen zur Systemarchitektur konnten bereits zu Beginn geklärt werden. Um einen Überblick über das Gesamtsystem zu ermöglichen, wurden Blockschaltbilder erstellt und in Telefonkonferenzen mit den Projektpartnern diskutiert. Dadurch wurde sichergestellt, dass alle Projektpartner über dasselbe Verständnis der Systemarchitektur verfügen. Das erste Blockschaltbild des Gesamtsystems ist in **Error! Reference source not found.** dargestellt wobei der Virtuelle Prototyp nur den eingerahmten Teil repräsentieren soll. Zur Vertiefung dieser Diskussionen wurde eine separate zweiwöchentliche Telefonkonferenz initiiert, die sich speziell mit den Belangen der Hardware sowie des Virtuellen Prototypen auseinandersetzt. Diese zweiwöchentliche Telefonkonferenz wurde ein wichtiger Bestandteil und Treiber von Diskussionen unter den Projektpartnern bezüglich der Architektur und Hardware bezogenen Themen im Allgemeinen.

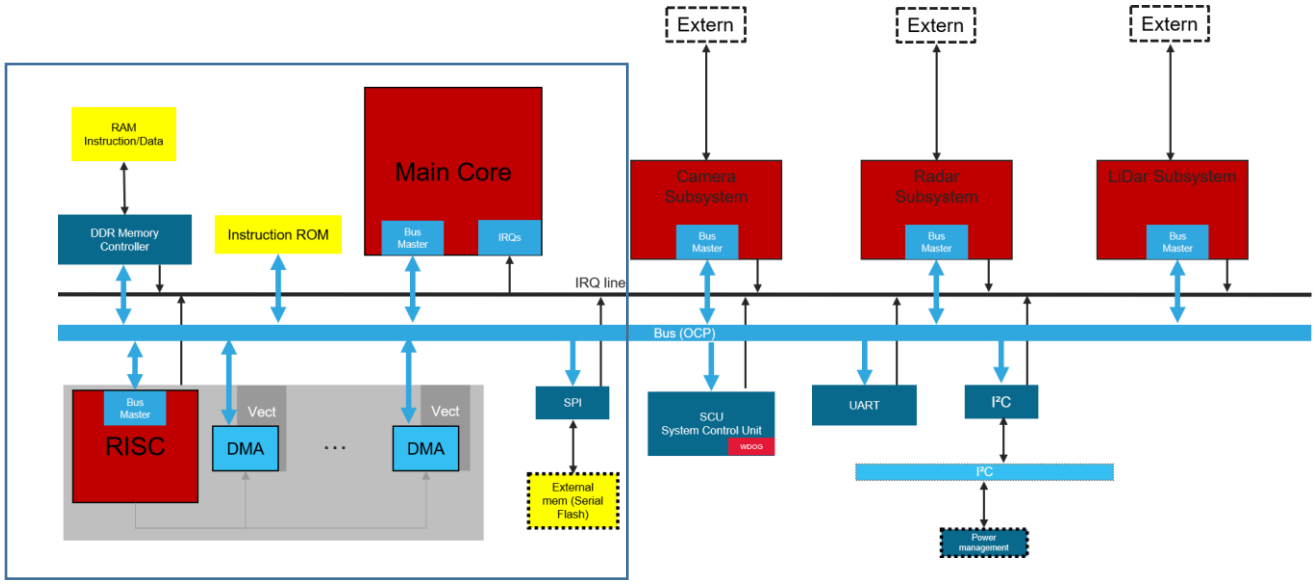

Abbildung 2: Blockschaltbild des Gesamtsystems

AP6 - Compilerenwurf und Implementierung

Die zweiwöchentliche Telefonkonferenz wird weiterhin für die stetige Synchronisation der verantwortlichen Partner des ICE (RWTH Aachen) und der Cadence Design Systems GmbH genutzt. Derzeit arbeitet der Virtuelle Prototyp mit Applikationen, die mit GNU RISC-V Toolchain kompiliert wurden und ist zudem so detailreich modelliert, dass Applikationen, die für die FPGA-Implementierung erstellt worden sind, ebenfalls auf dem Virtuellen Prototypen ausführbar sind. Mit diesem Aufbau ist es somit möglich die Funktionalität des neu entwickelten Compilers auf beiden Plattformen zu analysieren und zu bestätigen. Im weiteren Verlauf des Projektes haben Gespräche stattgefunden, um herauszufinden, wie und ob der neu entwickelte Compiler die angestrebten Techniken bezüglich SW-Hardening unterstützen kann. Der Compiler und die damit generierten Anwendungen können vollständig am Virtuellen Prototypen evaluiert werden.

3. Ergebnisse der Cadence Design Systems GmbH

AP3 - Virtueller Prototyp

Während des ersten Jahres wurden strukturelle und organisatorische Vorkehrungen getroffen, die die Entwicklung und den Austausch des Virtuellen Prototypen im Laufe des Projektes erleichtern. Dazu gehört auch die Auflistung aller benötigter Lizenzen, Toolversionen und dergleichen. Die bereits im Abschnitt zu Arbeitspaket 1 angesprochene Telefonkonferenz zu Hardware-Themen wurde dazu genutzt, die Anforderungen und Erwartungen an den Virtuellen Prototypen zwischen allen Beteiligten zu synchronisieren. Abbildung 3 zeigt die initiale Architektur des Virtuellen Prototypen mit den für das Projekt relevanten Komponenten. Hieraus ergibt sich zum Beispiel die Abstraktion der Subsysteme für Kamera, Lidar und Radar durch einen File-basierten Ansatz. Auch technische Themen wie das Einbinden von Instruktionssatzsimulatoren für verschiedene Prozessoren wurden diskutiert und beschlossen. Aufgrund der Tatsache, dass das Thema Virtueller Prototyp für viele Projektpartner neu war, wurde seitens Cadence eine Einleitung in das Thema gegeben, die Vorteile erläutert und mit Hilfe von Blockschaltbildern der Aufbau und der Umfang Virtueller Prototypen erklärt und aufgezeigt.

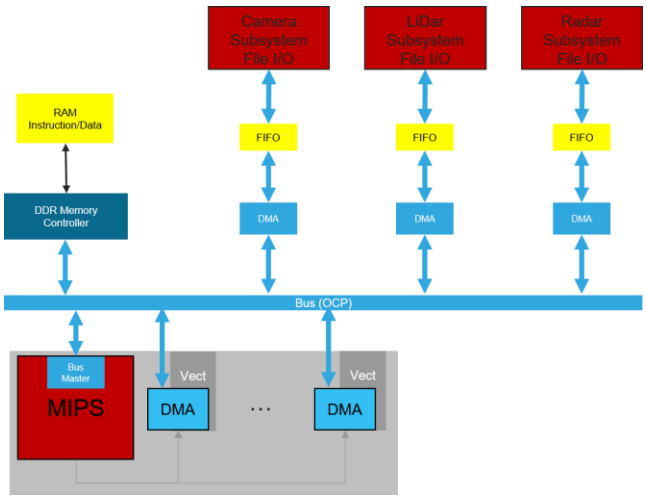

Abbildung 3: Initiale Architektur des Virtuellen Prototypen

Im weiteren Verlauf wurde der Virtuelle Prototyp iterativ erweitert, verfeinert und zuerst eine so genannte „Pipecleaner“-Version des Virtuellen Prototypen entwickelt und an die jeweiligen Projektpartner herausgegeben. Die Aufgabe dieser „Pipecleaner“-Version ist es, die getroffenen strukturellen und organisatorischen Vorkehrungen bezüglich Lizenzen, Toolversionen und dergleichen zu verifizieren. Zudem dient er als frühzeitig verfügbares Testvehikel, mit dem die Struktur und die Benutzung eines Virtuellen Prototypen durch die Partner analysiert und evaluiert werden kann. Der Aufbau des „Pipecleaners“, bestehend aus einer MIPS CPU, Bus, Speicher und einigen weiteren Peripherien, ist in Abbildung 4**Error! Reference source not found.** dargestellt.

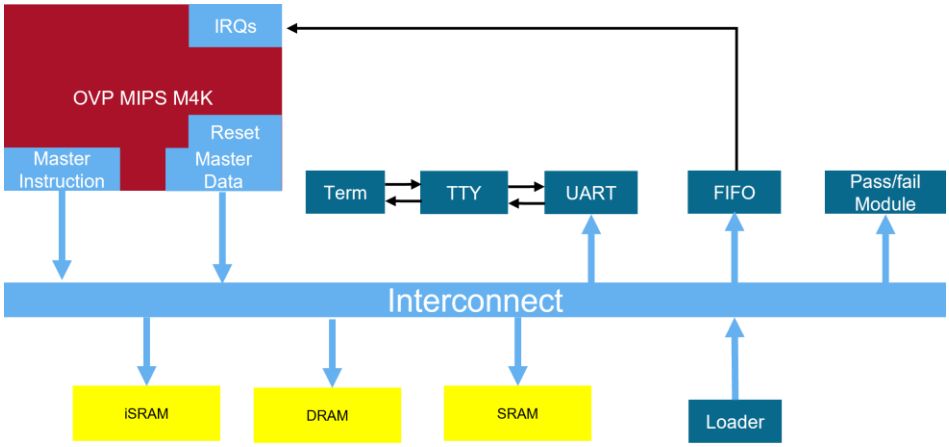

Abbildung 4: Aufbau des Pipecleaners

Im weiteren Verlauf wurden mit Hilfe der zweiwöchentlichen Hardwaretelefonkonferenz sowie mit vielen weiteren Telefonaten und Absprachen die weiteren Rahmenbedingungen für den ersten Virtuellen Prototypen festgelegt. Es wurde sich darauf geeinigt, dadurch, dass bereits für den MIPS entwickelte Applikationen zur Verfügung standen, die Funktionalität des Virtuellen Prototypen anhand dieser Applikationen effizient zu verifizieren. Im Verlauf des Projektes wurde der MIPS-Prozessor durch einen RISC-V-Prozessor ersetzt.

Die erste Version des Virtuellen Prototypen besteht aus dem Kontrollprozessor, dem Instruktionssatzsimulator (ISS) des Vektorprozessors, sowie einem Interconnect, Speicher und einem Loader. Der Loader dient der Vorinitialisierung des Speichers mit Daten oder Programmen. Der Virtuelle Prototyp basiert auf der Hardwarebeschreibungssprache SystemC. Der vom Partner IMS zur Verfügung gestellte Instruktionssatzsimulator des Vektorprozessors basiert auf einer Implementierung aus C++ und dem Entwicklungsframework Qt5 und wurde für den „Standalone“-Betrieb entwickelt und enthält einige Abstraktionen, die die Genauigkeit der Simulationsergebnisse beeinflussen. Die Applikation zum Beispiel wird in der „Standalone“-Version nicht für einen eingebetteten Prozessor kompiliert, sondern auf dem Host ausgeführt. Speicherzugriffe wurden auf ein internes Array abgebildet und damit Zugriffslatenzen maskiert.

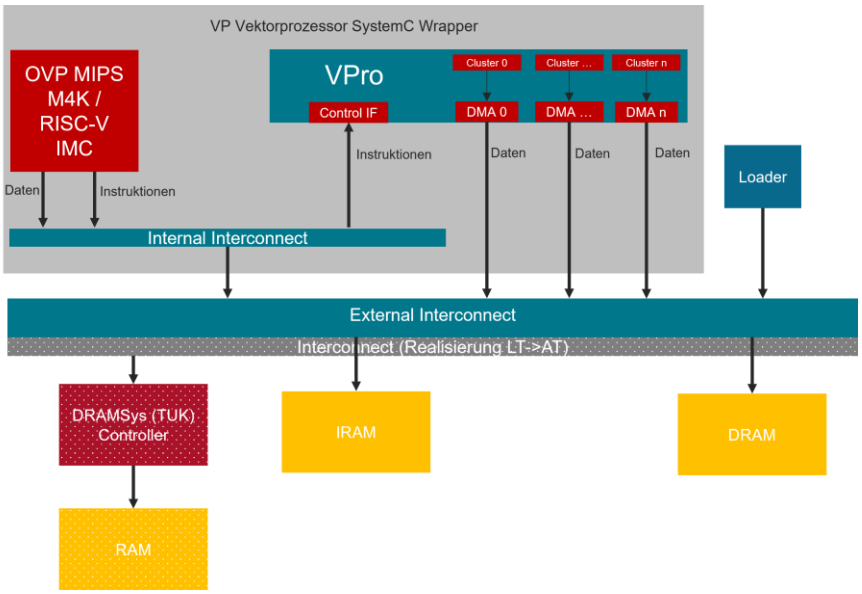

Abbildung 5: Erste Version des Virtuellen Prototyps

Mit Hilfe eines Virtuellen Prototypen können diese Abstraktionen detailreicher implementiert und somit aussagekräftigere Ergebnisse erzielt werden. Um den gelieferten Simulator nun in die SystemC-Welt einbinden zu können war die Entwicklung eines SystemC-Wrappers für den ISS notwendig. Wichtige und aufwändige Aspekte bei der Entwicklung dieses Wrappers war die Schaffung einer gemeinsamen Zeitbasis und Synchronisation sowie das Herausführen von Speicherzugriffen aus dem ISS in die externen Modelle von Prozessor und Speicher. Die erste Version des Virtuellen Prototypen ist in Abbildung 5 dargestellt. Zu sehen ist der Kontrollprozessor (erst MIPS, später RISC-V), der über ein Kontrollinterface auf den Vektorprozessor zugreifen und ihn damit programmieren kann, sowie verschiedene DMAs des Vektorprozessors, die über ein Interconnect auf externen Speicher zugreifen können. Diese Version des Virtuellen Prototypen wurde den Projektpartnern im Projektjahr zur Verfügung gestellt und spiegelt die funktionale Version des Virtuellen Prototypen wider, die mit Meilenstein 3.1 bereitgestellt werden sollte. Dieser Meilenstein wurde also rechtzeitig und erfolgreich erzielt.

```
const int16_t kernel_outline[9] = {-1, -1, -1, -1, 9, -1, -1, -1, -1};
const int16_t kernel_sharpen[9] = {0, -1, 0, -1, 5, -1, 0, -1, 0};
const int16_t kernel_laplace[9] = {0, 1, 0, 1, -4, 1, 0, 1, 0};
const int16_t kernel_sobel_x[9] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
const int16_t kernel_sobel_y[9] = {-1, -2, -1, 0, 0, 0, 1, 2, 1};
dma_ext1D_to_loc1D(0, (uint64_t) ((intptr_t) (&kernel_laplace)), LM_BASE_VU(0) + 300, 9);
dma_wait_to_finish(0xFFFFFFFF);

// kernel LM to LS
__vpro(LS, NONBLOCKING, IS_CHAIN, FUNC_LOADS, NO_FLAG_UPDATE,
                                DST_ADDR(300, 1, 3),
                                SRC1_ADDR(300, 1, 3), SRC2_IMM(0), 2, 2);



// kernel LS to RF
__vpro(L0, NONBLOCKING, NO_CHAIN,
                                FUNC_ADD, FLAG_UPDATE, DST_ADDR(300, 1, 3),
                                SRC1_LS, SRC2_IMM(0),
                                2, 2);
```

Abbildung 6: Programmierung des Vektorprozessors

In Abbildung 6**Error! Reference source not found.** ist eine beispielhafte Applikation, kompiliert für den MIPS-Kontrollprozessor, zu sehen. Im blau umrandeten Bereich wird der DMA des Vektorprozessors für einen Transfer programmiert. Der rote Bereich startet eine Bearbeitung der Daten auf dem Vektorprozessor. Der Vektorprozessors wurde so integriert, dass die Debugging-Umgebung des ISS parallel

zu der SystemC-Simulation läuft. Abbildung 7 zeigt z. B., wie die Daten nach der Programmierung, wie in Abbildung 6 gezeigt, im lokalen Speicher und im Registerfile des Vektorprozessors liegen. Der Transfer der Daten kann parallel in der Debugging-Umgebung der SystemC-Simulation (SimVision) verfolgt werden, wie in Abbildung 8**Error! Reference source not found.** angedeutet. Die Instruktionspipeline des Vektorprozessors ist ebenfalls in der Konsole sichtbar, wie in Abbildung 9**Error! Reference source not found.** dargestellt. Um die Arbeit der Programmierer zu vereinfachen, wurde zusätzlich eine Semihosting-Funktionalität implementiert, die es erlaubt, Printausgaben aus der eingebetteten Software auf der Konsole auszugeben. Außerdem kann man innerhalb der Simvision Debug-Umgebung der SystemC-Simulation den gesamten Speicherbereich der Plattform live sehen und somit die Funktionalität der Applikation verifizieren.

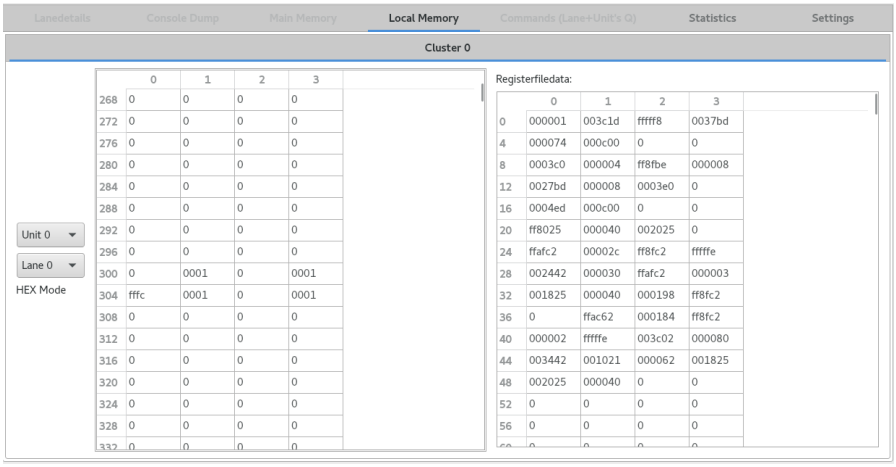

Abbildung 7: Speicherinhalt des Vektorprozessors

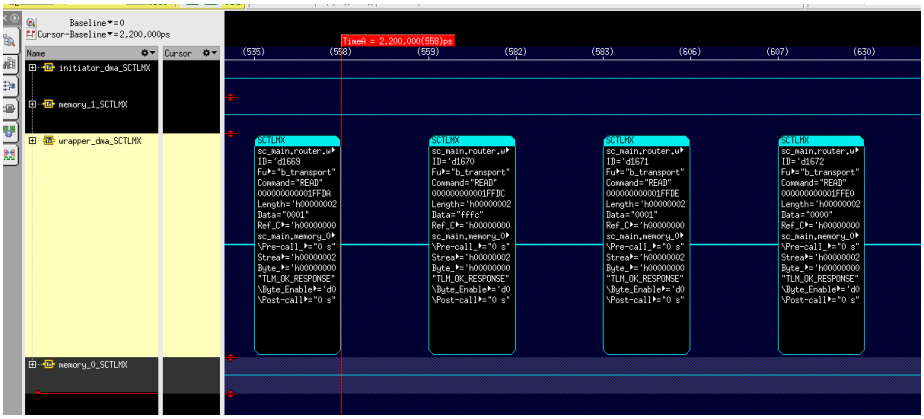

Abbildung 8: Transport der Daten über den Bus

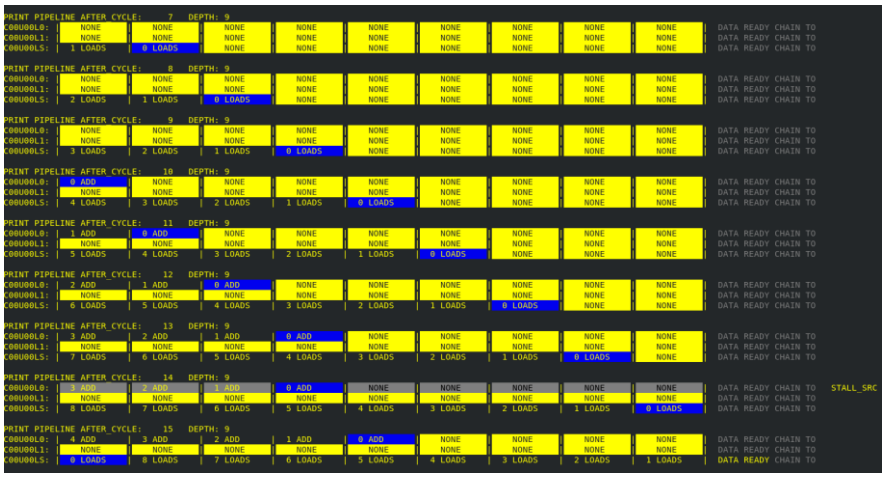

Abbildung 9: Bearbeitungspipeline des Vektorprozessors

Die zuvor beschriebene erste Version des Virtuellen Prototypen integrierte bereits den Instruktionssatzsimulator (ISS) des Vektorprozessors, den ISS des Kontrollprozessors, ein Interconnect sowie beispielhafte Speichermodelle für Daten und Instruktionen. Ein zusätzlicher Loader wurde verwendet, um den Speicher mit der gewünschten Applikation zu initialisieren. Wie vom Konsortium beschlossen, wurde der MIPS-Kontrollprozessor endgültig durch einen RISC-V (RV32IMC) basierten Prozessor ersetzt, der mit den Erweiterungen I:32 Bit Register, M: Integer Multiplikation und Division, C: komprimierte Instruktionen ausgestattet ist. Im abgelaufenen Projektjahr wurde dieser Virtuelle Prototyp um alle notwendigen und vorgesehenen Funktionen und Komponenten erweitert und den Projektpartnern zur Verfügung gestellt. Das Blockschaltbild des erweiterten Virtuellen Prototypen ist in Abbildung 10 dargestellt. Ein wichtiger Bestandteil dieser Komplementierung war das Einbinden des DRAM-Controllers (DRAMSys), der durch die Universität Kaiserslautern bereitgestellt wurde. Um diese Integration zu realisieren, musste eigens für dieses Projekt ein Interconnect entwickelt werden, der es erlaubt, die verschiedenen Abstraktionslevel der vorhandenen Komponenten miteinander zu verbinden. Durch Support, Bugfixes und Weiterentwicklungen kommt es weiterhin zu Aufwänden in diesem Arbeitspaket.

Der DRAM-Controller wurde im sogenannten AT (Approximately Timed) Modellierungsstil beschrieben. Hierbei sind Transporte zwischen verschiedenen Modellen nicht blockierend und folgen einem 4-phasigen Protokoll (Begin Request, End Request, Begin Response, End Response). Dieser Stil erlaubt eine detailreichere Darstellung der Transporte und beispielsweise Realisierung von Pipelining. Dieser Stil ist für die Modellierung des DDR-Controllers notwendig, da dadurch so genannter „Back-Pressure“ aufgebaut werden kann. Das heißt, dass viele Transporte schnell hintereinander auftreten können ohne das vorherige abgeschlossen sind und somit die wirkliche Auslastung des Speicherinterfaces analysiert werden kann. Diese detailreiche Modellierung verringert jedoch im Gegenzug die Simulationsperformanz.

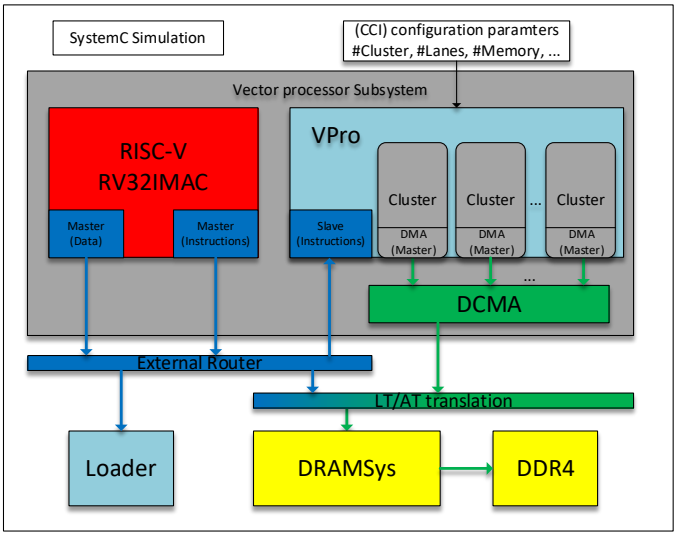

Abbildung 10: Blockschaltbild des erweiterten Virtuellen Prototypen

Der in diesem Projekt verwendete ISS von Imperas zur Modellierung des RISC-V verwendet blockierende Transporte und realisiert somit den LT (Loosely timed) Modellierungsstil, der einen wesentlich geringeren Detailgrad aufweist, jedoch die Simulationsperformanz enorm erhöht. Um nun ein Gesamtsystem mit den benannten Einzelkomponenten aufbauen zu können mussten einige Randbedingungen erfüllt werden.

Zum Beispiel musste der ISS des Vektorprozessors, beziehungsweise das Interface zur Programmierung und die Dateninterfaces der variierenden Anzahl von DMAs einem der beiden zuvor beschriebenen Modellierungsarten zugeschrieben werden. Da die Programmierung des Vektorprozessors über den RISC-V Core realisiert wird, und dieser wie beschrieben mit blockierenden Transporten arbeitet und zudem der Detailgrad im LT-Stil bezüglich dieser Schnittstelle ausreichend ist, wurde das Kontrollinterface als LT definiert. Anders sieht es jedoch mit den Dateninterfaces der DMAs aus. Diese sind verantwortlich für die Hauptlast am DRAM-Controller und wie bereits beschrieben, ist der Detailgrad bezüglich „Back-Pressure“ und Pipelining für ein solches Modell von hoher Wichtigkeit. Daher wurden diese Interfaces so implementiert, dass sie dem AT-Modellierungsstil folgen.

Um stetige Weiterentwicklungen am Vektorprozessor und dem Virtuellen Prototypen zu unterstützen, ohne, dass die Integration der beiden Komponenten davon beeinflusst wird und zudem der ISS des Vektorprozessors noch im „Standalone“ Betrieb, also ohne Systemkontext betrieben werden kann, wurde zusammen mit den Partnern des EIS aus Braunschweig eine API (Applikation Programming Interface) entwickelt, die genau definiert, welche Funktionen implementiert werden müssen, um Datentransfers aus dem Vektorprozessor zu initiieren. Diese API ist in nachfolgender Liste dargestellt.

- bool request_read_transfer_cb (intptr_t dst_addr_ptr, const uint32_t burst_length, const uint32_t initiator_id);
- void dbg_write_cb(intptr_t dst_addr, uint8_t* data_ptr);
- void dbg_read_cb(intptr_t dst_addr, uint8_t* data_ptr);
- bool request_write_transfer_cb(intptr_t dst_addr_ptr, const uint8_t *data_ptr, const uint32_t burst_length, const uint32_t initiator_id) ;
- bool is_read_data_available_cb (const uint32_t initiator_id);
- bool is_write_data_ready_cb (const uint32_t initiator_id);
- bool read_data_cb (uint8_t * data_ptr, const uint32_t initiator_id);

Einer der wichtigsten Aspekte war anschließend die Implementierung eines Interconnects, welches in der Lage ist, die zuvor beschriebenen, verschieden abstrahierten Transporte gegenseitig zu übersetzen und somit die Verbindung der verschiedenen Komponenten zu einem Gesamtsystem zu ermöglichen. Der entwickelte Interconnect ist in der Lage, die blockierenden Transporte des RISC-V (Daten und Instruktionen) entgegenzunehmen und in die Phasen des AT-Modellierungsstils zu übersetzen. Der DDR-Controller kann diese Transporte verarbeiten und Daten sowie Instruktionen zurückliefern. Der Interconnect ist in Abbildung 10**Error! Reference source not found.** mit der Aufschrift „LT/AT translation“ zu sehen. Blaue Interfaces beschrieben in dieser Abbildung LT-basierende Interfaces und grüne Interfaces eine AT-basierende Implementierung. Der farbliche Übergang von Blau nach Grün im Interconnect deutet die Übersetzung an.

Abbildung 11**Error! Reference source not found.** zeigt ein so genanntes „Transaction Stripe Chart“, mit dem es möglich ist, die Transporte, die sich durch das System ausbreiten, zu verfolgen. Hier in diesem Beispiel beobachten wir einen Read Transport vom neu entwickelten Interconnect Richtung DDR-Controller. In der Spalte „Pre-Call phase“ kann man die verschiedenen, bereits beschriebenen Phasen des AT-Protokolls beobachten: Begin-Request -> End-Request -> Begin-Response -> End-Response.

Time	Stream_name	Label	Initiator	Id	Hop	Function	Command	Pre-call_phase	Pre-call_time	Streaming_width	Byte_enable
2000ps	initor_socket_SCTLMX	SCTLMX	zuse_platform.at_router_ext.initor_socket	33047	0	nb_transport_fw	READ	BEGIN_REQ	0 s	00000002	0000000
2499ps	initor_socket_SCTLMX	SCTLMX	zuse_platform.DDR.arbiter.multi_passthrough_initiator_socket_0	33047	1	nb_transport_fw	READ	BEGIN_REQ	0 s	00000002	0000000
2499ps	initor_socket_SCTLMX	SCTLMX	zuse_platform.DDR.controller0.tSocket	33047	2	nb_transport_bw	READ	END_REQ	0 s	00000002	0000000
2499ps	initor_socket_SCTLMX	SCTLMX	zuse_platform.DDR.controller0.iSocket	33047	3	nb_transport_fw	READ	0 s	00000002	00000000	0000000
2499ps	initor_socket_SCTLMX	SCTLMX	zuse_platform.DDR.DRAMSys_tSocket	33047	4	nb_transport_bw	READ	END_REQ	0 s	00000002	0000000
16660ps	initor_socket_SCTLMX	SCTLMX	zuse_platform.DDR.controller0.iSocket	33047	5	nb_transport_fw	READ	0 s	00000002	00000000	0000000
34153ps	initor_socket_SCTLMX	SCTLMX	zuse_platform.DDR.controller0.tSocket	33047	6	nb_transport_bw	READ	BEGIN_RESP	0 s	00000002	0000000
34153ps	initor_socket_SCTLMX	SCTLMX	zuse_platform.DDR.DRAMSys_tSocket	33047	7	nb_transport_bw	READ	BEGIN_RESP	0 s	00000002	0000000
35153ps	initor_socket_SCTLMX	SCTLMX	zuse_platform.at_router_ext.initor_socket	33047	8	nb_transport_fw	READ	END_RESP	0 s	00000002	0000000
35819ps	initor_socket_SCTLMX	SCTLMX	zuse_platform.DDR.arbiter.multi_passthrough_initiator_socket_0	33047	9	nb_transport_fw	READ	END_RESP	0 s	00000002	0000000

Abbildung 11: Transaction Stripe Chart

Im obigen Text wurde die erweiterte Version des Virtuellen Prototypen, die alle wesentlichen notwendigen Komponenten integriert, beschrieben. Dazu gehören der RISC-V (RV32IMC) basierte Prozessor, der mit den Erweiterungen I: 32 Bit Register, M: Integer Multiplikation und Division, C: komprimierte Instruktionen ausgestattet ist, sowie der DDR-Controller (DRAMSys 4.0) des Partners TU Kaiserslautern und dem Instruktionssatzsimulator des Vektor Prozessors (VPro) der TU Braunschweig.

Im weiteren Verlauf wurde der Virtuelle Prototyp kontinuierlich weiterentwickelt und an die Veränderungen der RTL/FPGA Referenzimplementierung angepasst, sowie Bug-Fixes und etwaige Verbesserungen eingepflegt. Der VP (Abbildung 12) reflektiert hier das aktuell bestehende Gesamtsystem und somit auch den endgültigen fertigen Virtuellen Prototypen. Dieser kann zur Analyse und Softwareentwicklung von den Projektpartnern bei Bedarf eingesetzt werden.

Erweiterungen am Virtuellen Prototypen

Eine wesentliche Neuentwicklung war die Implementierung von Cache Modellen für die Instruktionen sowie Daten des RISC-V Prozessors. Die Cache Modelle sind in Abbildung 12 in Rot an den beiden Master-Ports des RISC-V Kontrollprozessors dargestellt. Die Modelle wurden so implementiert, dass sie, wie der Vektorprozessor, auf einfache Art und Weise von außen per CCI (Control, Configuration and Inspection) Parameter bezüglich der „Line Size“, „Anzahl an Lines“ und „Assoziativität“ konfigurierbar sind. Der Instruktions-Cache ist z.B. in der Grundversion wie folgt konfiguriert:

- icache.ln_line_size 9 ; 512 Bytes
- icache.ln_nb_cache_lines 4 ; 16 lines
- icache.ln_nb_ways 0 ; 1 ways, 16 sets

Die eingestellten Werte werden als Potenz zur Basis 2 gesehen. Der Cache hat somit 2^9 = 512 Bytes pro Line und 2^4 = 16 Lines. In der Summe ergibt dies eine Cachegröße von 8192 Bytes. Zusätzlich gibt der Anzahl der Wege die Assoziativität des Cache an. In dem Falle ist ein „direct mapped“ Cache dargestellt.

Eine weitere wesentliche Erweiterung war die Implementierung einer dedizierten programmierbaren „Finite State Maschine“ (FSM), die dazu benutzt wird, automatisch „Direct Memory Access“ (DMA) Instruktionen aus dem Daten Cache des RISC-V Prozessors in die DMA-Instruktion-Queue des Vektorprozessors zu übertragen.

Im Blockdiagramm in Abbildung 12**Error! Reference source not found.** ist diese als „Command DMA“ bezeichnet. Um die Einbindung zu ermöglichen, wurde der SystemC Wrapper des Vektorprozessors um einen weiteren Slave-Port erweitert, über den auf die DMA-Instruktions-Queue zugegriffen werden kann. Das Daten Cache Modell wurde ebenfalls um einen Slave-Port erweitert, aus dem die FSM aktiv die Daten herauslesen kann. Die Memory-Map wurde so angepasst, dass die Programmierschnittstelle der FSM im selben IO-Bereich liegt, wie die des Vektorprozessors.

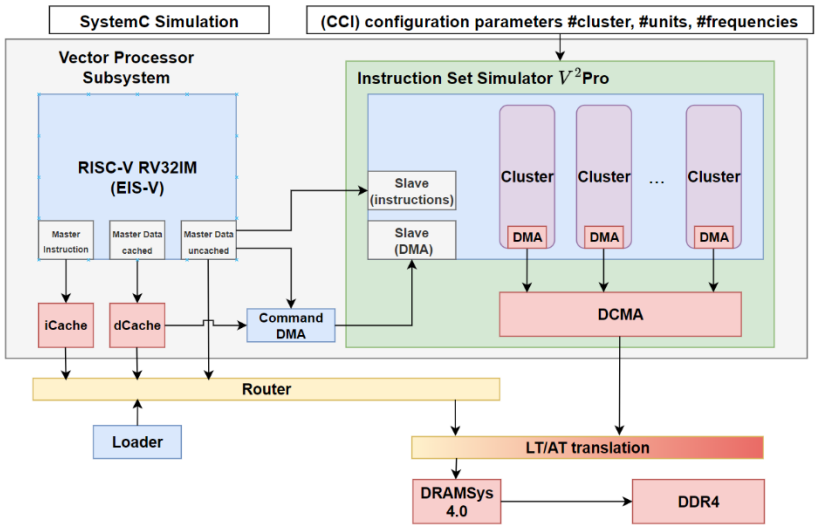

Abbildung 12: Blockdiagramm des vollständigen Virtuellen Prototypen

Experimente und Vergleiche

In Zusammenarbeit mit der TU Braunschweig wurden im weiteren Verlauf des Projektes die drei Convolutional Neural Networks (CNNs) VGG-16, MobilenetV2 und Yolo-Lite auf dem Virtuellen Prototypen implementiert und mit den anderem im Projekt benutzen Prototyping-Methoden (FPGA und ISS-Standalone) bezüglich der Genauigkeit, Ausführungszeiten sowie erzeugbarer Metriken verglichen.

Im folgenden Abschnitt werden die Ergebnisse dieser Arbeit kurz zusammengefasst. Eine ausführliche Abhandlung wurde in Form einer Veröffentlichung bei der „35th IEEE International Conference on Application-specific Systems, Architectures and Processors“ (ASAP2024) unter dem Titel „Multi-Level Prototyping of a Vertical Vector AI Processing System veröffentlicht [5].

Vergleich von Genauigkeiten

Der erste Vergleich der verschiedenen Prototypen-Techniken bezieht sich auf die Genauigkeit der Ergebnisse in Bezug auf die gemessene Ausführungszeit des „Inference-Path“ der verschiedenen Netzwerke. Die FPGA-Implementierung wurde hierbei als Referenz angenommen, und mit dieser der relative Fehler von ISS-Standalone sowie Virtuellem Prototypen zum FPGA errechnet. Jedes Netzwerk wurde in 64 verschiedenen Konfigurationen des Vektorprozessors hinsichtlich der Anzahl von Clustern und Units pro Cluster untersucht, während der Rest der Konfiguration im Rahmen dieser Arbeit statisch gehalten wurde. Die Ergebnisse sind links in Abbildung 13 dargestellt. Das Diagramm auf der linken Seite zeigt die relative Abweichung für das MobileNetV2 Netzwerk. Die Y-Achse repräsentiert die relative Abweichung der Laufzeit im Vergleich zum FPGA in Prozent. Die grünen Graphen beziehen sich auf die Differenz ISS-FPGA, die sich zwischen 16,25% für die 1c8u Konfiguration und 45,15% für die 7c8u Konfiguration bewegt. Die blauen Plots beziehen sich auf die Differenz zwischen VP und FPGA und bewegen sich zwischen 3,9% für die 1c8u und 29,8% für die 7c8u Konfiguration. Im Mittel erreichen wir mit dem ISS eine Abweichung von 29% und für den VP eine Abweichung von 14% und erreichen somit eine 2,07x bessere Genauigkeit mit dem VP.

Die Ergebnisse für das VGG-16 Netzwerk sind rechts in Abbildung 13 dargestellt. Die ebenfalls grünen Graphen für die ISS-FPGA Differenz zeigen eine Abweichung zwischen 4,49% und 5,05% während die blauen für die VP-FPGA Different eine Abweichung zwischen 1,45% und 2,07% zeigen. Im Mittel erreicht der ISS eine Abweichung von 4.66% und der VP von 1,62%, also eine 2,9x bessere Genauigkeit.

Vergleich von Laufzeit und DDR-Bandbreite.

Der zweite Vergleich bezieht sich auf die Analyse weiterer Metriken, die nur über den Virtuellen Prototypen und den dadurch gegebenen Systemkontext zur Verfügung stehen sowie auf Konfigurationsgrößen, die durch den FPGA aufgrund seiner Ressourcenlimitierung nicht darstellbar sind. Abbildung 14, links, stellt weitere 18 Konfigurationen für das MobileNetV2 Netzwerk dar, bis hin zur Größe von 32 Cluster und 32 Units pro Cluster.

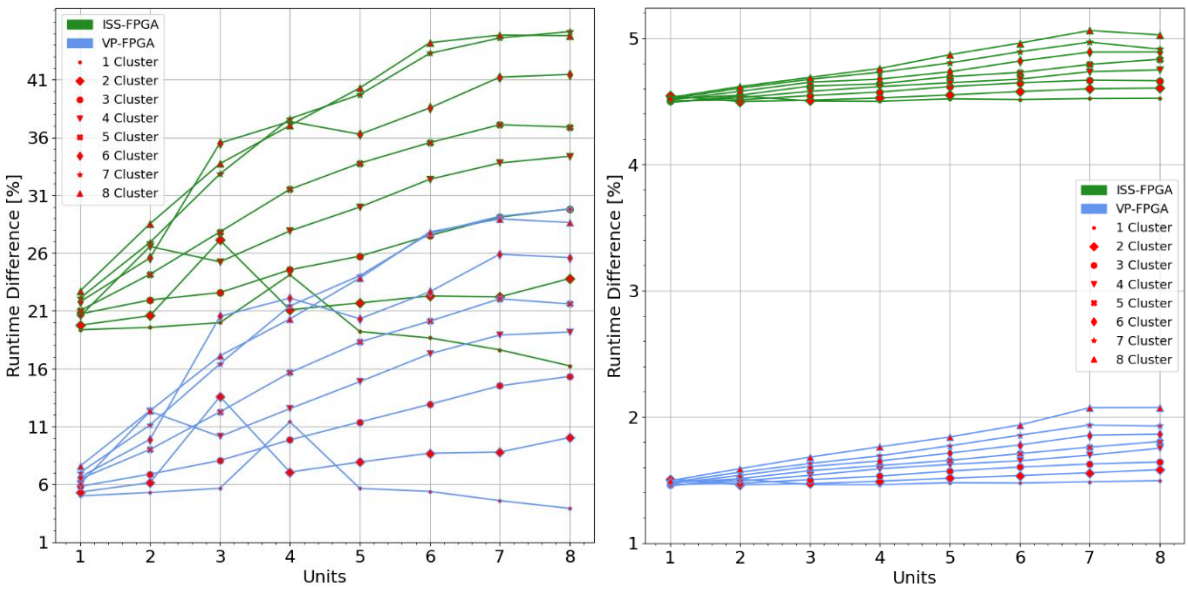

Abbildung 13: Vergleich der Genauigkeit zwischen VP und ISS (links: MobilenetV2; rechts: VGG-16)

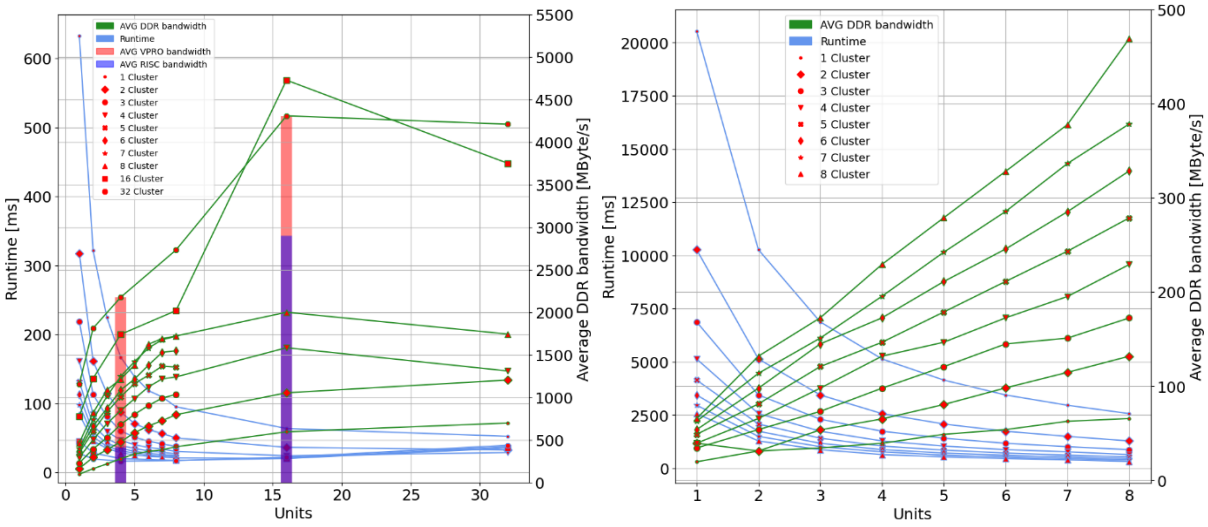

Abbildung 14: Laufzeit und benutzte DDR-Bandbreite (links: MobileNetV2; rechts VGG-16)

An den blauen Graphen ist ersichtlich, dass die Laufzeit für das Neuronale Netzwerk (linke Y-Achse in ms) zwischen 632ms für die kleinste Konfiguration (1c1c) und 16ms für die Konfiguration 32c4u liegt. Die größte Konfiguration (32c32u) führt offensichtlich nicht zur höchsten Performanz. Die grünen Graphen beziehen sich auf die durchschnittliche am DDR-Controller genutzte Bandbreite (rechte Y-Achse in Mbyte/s). Interessant an den Ergebnissen ist die Tatsache, dass bei Konfigurationen mit mehr als 32c4u die durchschnittliche Datenbandbreite hauptsächlich durch den RISC-V und die Cache-Lasten bestimmt wird, die VPro Befehle liefern. Wie im Balkendiagramm links in Abbildung 14 zu sehen ist, setzt sich die durchschnittliche Bandbreite von 2.176 MByte/s für die 32c4u-Konfiguration aus 1.762 MByte/s Datentransfer vom Speicher direkt zu VPro (roter Balken) und 414 MByte/s zwischen Speicher und RISC-V und Caches (violetter Balken) zusammen. Das Verhältnis kehrt sich bei der 32c16u-Konfiguration um, die im rechten Balkendiagramm dargestellt ist. Diese Konfiguration benötigt eine durchschnittliche

Bandbreite von 4.308 MByte/s, von denen 2.901 MByte/s (violetter Balken) durch RISC-V und Caches und nur 1.406 MByte/s durch den VPro (roter Balken) erzeugt werden.

Ein ähnliches Diagramm für das neuronale Netz VGG-16 ist rechts in Abbildung 14 dargestellt. Das neuronale Netz selbst ist viel größer als MobileNetV2, was zu einer längeren Gesamtlaufzeit führt, die zwischen 20.542 ms für die 1c1u-Konfiguration und 321 ms für die größte getestete Konfiguration (8c8u) liegt. Die verwendete Speicherbandbreite steigt stetig an, beginnend bei 19.66 MByte/s für die kleinste Konfiguration (1c1u) und erreicht 469 MByte/s für die 8c8u-Konfiguration.

Exploration von Cache Parametern

Im Folgenden wird eine Untersuchung zusätzlicher Konfigurationen der Befehlscaches vorgestellt. Auf dem VP ist dies durch Änderung der Konfigurationsparameter, wie zuvor beschrieben, der Cachemodelle und eine anschließende parallele Simulation leicht und sehr schnell zu erreichen. In dieser kleinen Untersuchung wird die Größe der Cache-Zeilen und damit die Gesamtgröße des Befehls-Caches des RISC-V um eine Zweierpotenz von 16 Zeilen mit 2^9 = 512 Bytes pro Zeile auf 2^1= 2 Bytes pro Lane reduziert. Die Messungen reichen also von Cache-Größen zwischen 32 und 8.192 Bytes. Die in Tabelle 2 gezeigten Ergebnisse wurden für die 2c2u-Vektorprozessorkonfiguration erstellt, auf der die Anwendung MobileNetV2 läuft.

Die Gesamtlaufzeit des Inferenzpfads verringerte sich erheblich mit höheren Cache-Größen. Bei einer Cache-Größe von 1.024 Byte sind etwa 99% aller Befehlszugriffe bereits Cache-Hits. Die Spalte „control BW“ gibt an, welcher Anteil der Gesamtbandbreite auf RISC-V-Befehle und Cache-Zugriffe entfällt, während VPro die vom Vektorprozessor selbst erzeugten Anteile angibt. Mit zunehmender Cache-Größe steigt die vom VPro erzeugte Nutzung von nur 16% auf 39%.

Tabelle 2: Cache-Parameter Exploration

Size [Bytes]	Read Hit [%]	DDR BW [Mb/s]	Control BW [%]	VPro BW [%]	Runtime [ms]
32	0,29	127,79	83,05	16,94	1494,40
64	63,90	253,87	79,90	20,09	634,40
128	87,94	433,75	76,04	23,95	311,44
256	94,99	561,89	73,34	25,65	244,50
512	97,56	648,07	74,11	25,88	192,90
1024	99,10	653,36	71,66	28,33	174,80
2048	99,50	696,99	72,50	27,49	168,91
4096	99,96	538,33	63,13	36,86	163,10
8192	99,99	509,76	60,90	39,09	162,40

AP 4 – Sicherheitsaspekte des Vektorprozessors

Aufgrund fehlenden Personals, zum Teil auch begründet durch die damals globale Coronavirus-Pandemie, hatten sich die Arbeiten an diesem Arbeitspaket zeitlich nach hinten verschoben. Der Zeitverzug hat sich sogar als sinnvoll herausgestellt, da eine existierende erste Version des Virtuellen Prototypen für die Arbeiten an den Sicherheitsaspekten von großem Vorteil war. Im Bereich der Sicherheitsaspekte des Vektorprozessors geht es darum, die eingebettete Software möglichst frühzeitig auf Fehler zu überprüfen und somit eine stets sichere Ausführung zu gewährleisten. Dazu wurde analysiert, welche typischen Fehler auftreten können und welche eine besondere Wichtigkeit für das Projekt annehmen.

Typische Fehler sind zum Beispiel die Verwendung nicht initialisierter Speicher („read before write“), Zugriffe außerhalb zugewiesenen oder vorhandenen Speichers, Verwendung nicht initialisierter Pointer („use after delete“) oder Speicherlecks. Speicherlecks entstehen beispielsweise durch das Vergessen der Freigabe nicht mehr verwendeten Speichers. Diese Fehler machen sich meist erst später zur Laufzeit bemerkbar und sind daher wichtig zu finden. Für dieses Projekt wird die Überprüfung auf Verwendung von nicht initialisiertem Speicher als erster zu findender Fehler festgelegt, da Vektorprozessor Anwendung sehr datenzentrisch und diese Fehler ohne entsprechende Hilfe schwierig zu finden sind.

Neben der Fehler an sich, wurden bereits existierende Lösungen analysiert. Namhafte Kandidaten sind hier Werkzeuge wie Valgrind[4], Memory Sanitizer,[5] Mudflap oder gperftools. All diese Werkzeuge haben den Nachteil, dass sie die Laufzeit des ausgeführten Programms verändern, was bei echtzeitfähigen eingebetteten Systemen zu großen Problemen führen kann. Durch das Einbetten dieser Verfahren in die SystemC- Simulation soll dies verhindert werden. Die analysierten Werkzeuge können in zwei Gruppen eingeteilt werden. Zum einen gibt es solche, die das bereits kompilierte Programm, das so genannte Binary, instrumentieren. Das hat den Vorteil, dass Programme nicht neu kompiliert werden müssen, um auf bestimmte Fehler zu testen. Die andere Gruppe beinhaltet Werkzeuge, die den Quelltext instrumentieren. Diese sind meist schneller als die Werkzeuge der anderen Gruppe, benötigen aber unter Umständen eine neue Übersetzung.

Es wurde ein erstes Konzept erstellt, wie eine solche Fehleranalyse in eine SystemC-Simulation eingebunden werden kann und welche Komponenten einer solchen Simulation, (ISS, SystemC, Host) welche Aufgaben übernehmen könnten. Eine Übersicht ist in Abbildung 15 dargestellt. Zu sehen ist hier auf der linken Seite der Kontrollprozessor, der die beschriebene Applikation ausführt. Jeder Aufruf einer Funktion zum Anfragen von Speicher (z. B. malloc) innerhalb des ISS soll über ein Analysemodul mitgeschnitten und in eine externe Datenbank geschrieben werden. Das Modul zur Zugriffsanalyse in der Mitte der Übersicht schneidet alle Zugriffe auf Adressen im externen Speicher mit und schreibt diese ebenfalls in die externe Datenbank. Ein Live Analyse Tool überwacht mit Hilfe der generierten Datenbank zur Laufzeit, ob die Zugriffe auf den Speicher auf vorher angefragte Regionen gehen, und generiert ein Logfile sowie auch Warnungen, falls hier Probleme entstehen. Dieses Konzept wurde im weiteren Projektverlauf anhand des Virtuellen Prototypen implementiert und getestet.

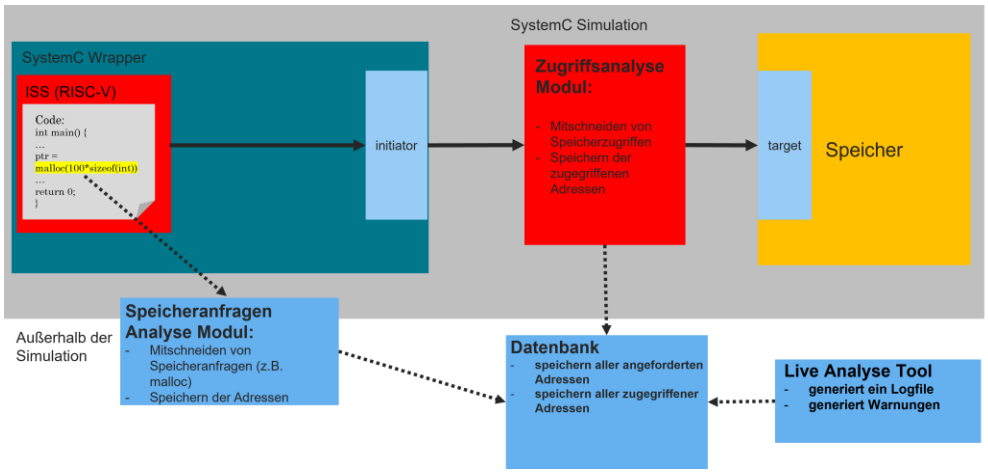

Abbildung 15: Übersicht zum Konzept des Softwarehardening

Im Rahmen von Arbeitspaket 4 wurde eine Liste mit konkret zu testenden Fehlerfällen aufgestellt, die im Rahmen von kleinen Experimenten untersucht wurden, so wie das vorgestellte Konzept zur Erkennung von Fehlern mit Hilfe einer SystemC Simulation verfeinert und erste Implementierungen vorgenommen.

Folgende Fehlerfälle wurden als relevant eingestuft und sollten im Verlauf des Projektes mit Hilfe des Virtuellen Prototypen analysierbar sein.

1. Heap Overflow (Überschreiten von laufzeitallokiertem Speicherbereich)
2. Stack Overflow (Überschreiten von Speicherbereich reserviert für Stack)
3. User after free (Benutzung von dynamischer Variablen, nachdem sie freigegeben wurden)
4. Use after return (Verwendung von Variablen außerhalb, der Funktion in der sie definiert wurden)
5. Uninitialized Memory reads (Verwenden von nicht initialisiertem Speicher mit nicht definierbaren Werten)
6. Memory Leaks (Nicht verwendeter Speicher wurde nicht freigegeben)
7. Stack Underflow (Unterschreiten des Speicherbereichs, der für den Stack reserviert ist)

Die Arbeiten befassten sich mit der Erkennung von Heap basierten Speicherfehlern wie Overflow oder auch Verwendung von nicht initialisiertem Speicher. Wie im vorher beschriebenen Konzept vorgestellt, ist es dafür ausgelegt die Befehle zur dynamischen Speicherbereitstellung (malloc/free) in der laufenden Applikation zu finden und die wichtigen Übergabe so wie Return-Parameter abzufangen, ohne dabei die Laufzeit oder das Speicherlayout der eigentlichen Applikation zu verändern. Um dies Umzusetzen, müssen Veränderungen am ISS des Prozessors vorgenommen werden, der die eingebettete Software ausführt. Im Falle dieses Projektes also dem von Imperas entwickelten Simulator des RISC-V Cores. Imperas stellt eine API zur Verfügung, um eine so genannte Intercept Library zu implementieren, die erlaubt bestimmte Symbole oder auch Adressen, die vom Simulator ausgeführt werden zu unterbrechen bzw. mit weiterer Funktionalität zu erweitern.

In der Software (hier C) wird ein dynamischer Speicherbereich über die Funktion „void* malloc(uint32_t size)“ angefordert, die in der C-Standardlibrary implementiert ist. Das Funktionsargument bezeichnet hierbei die gewünschte Größe des Speicherbereichs. Der Return-Wert von „malloc“ ist ein sogenannter void-Pointer (void*) der auf den Start des neu reservierten Speicherbereichs zeigt.

Um diese Werte abfangen zu können müssen also verschiedene Dinge erfolgen (am Beispiel für malloc())

1. Erkennen, wenn die Funktion malloc() aufgerufen wird
2. Erkennen mit welchem Argument diese Funktion aufgerufen wurde
3. Erkennen, wann der Aufruf von malloc() zu Ende ist
4. Erkennen, welchen Wert der Aufruf zurückgeliefert hat

Zu 1:

Der Aufruf von malloc() wurde dadurch erkannt, indem ein Callback auf das Debug Symbol „malloc“ welches im Binärfile der Applikation vorhanden ist im Simulator zu registrieren. Bei Imperas Simulatoren wird dazu ein neuer String in die Intercept-Liste eingetragen. Die rote Box in Abbildung 16 zeigt die Einträge die für free() und malloc(). Dieser Eintrag enthält einen so genannten Callback, der angibt was

passieren soll, wenn dieses Symbol gefunden wird. Wenn also das Symbol malloc() gefunden wird, wird innerhalb des Simulators die Funktion mallocCB() aufgerufen die im Folgenden beschrieben wird.

```
.intercepts = {
    // -------------------   ----------- ------ ----------------
    // Name                 Address     Opaque Callback
    // -------------------   ----------- ------ ----------------
    //
    // Intercept main for argc, argv, env
    //

    { "malloc",                  0,      OSIA_NONE , mallocCB  },
    { "free",                    0,      OSIA_NONE , freeCB  },
    {  NULL,                     0xfbc,  OSIA_NONE , addrCB  },
    {0}
  }
};
```

Abbildung 16: Intercept Liste

Zu 2:

Nachdem der Aufruf der Funktion gefunden wurde, musste sichergestellt werden, dass das Funktionsargument korrekt abgefangen wird. Hierbei muss man beachten, um welchen Prozessortypen es sich handelt. Der Ort für Übergabeparameter an Funktionen ist in der so genannten „Calling Convention“ Prozessorabhängig beschrieben. Für das hier verwendete Modell eines RISC-V Prozessors werden beim Aufruf einer Funktion die notwendigen Werte vom Aufrufenden in den Registern x10-11 des Prozessors abgelegt. Prozessorintern auch als a0-1 bezeichnet (siehe Abbildung 17)

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Abbildung 17: RISC-V Calling Convention[1]

[1] https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

In der Implementierung des Callbacks muss also der Wert des Registers a0 gespeichert werden. Dies ist im roten Rahmen in Abbildung 18 angedeutet.

```
static VMIOS_INTERCEPT_FN(mallocCB) {
    // vmiMessage("I", "EXC" "_FW", "Intercept 'mallocCB'.");
    Uns64 arg[8];
    getArg(processor, object, 0, &arg[0]);
    global_malloc_size = arg[0];
    //vmiMessage("I", "EXC" "_FW", "Intercept 'mallocCB' Malloc Size %d",(int)(arg[0]));
    printf("Saved Malloc Size: 0x%08x \n",global_malloc_size);
}
```

Abbildung 18: Malloc Callback

Zu 3/4:

Das Erkennen des Endes eines Funktionsaufrufs ist recht komplex, da je nach Parametern oder auch Erfolg und Fehlschlag eine variable Funktionslänge vorliegt. In erster Linie kann man das Binärfile der Applikation bezüglich der malloc() Funktion analysieren und feststellen, dass es nur einen Returnpfad aus dem Funktionsaufruf gibt und zwar in diesem Fall an Adresse 0xfbc (rot markiert in Abbildung 19).

```
1  0000e18 <_malloc_r>:
2  e18: 7179                addi sp,sp,-48
3  e1a: ce4e                sw s3,28(sp)
4  e1c: d606                sw ra,44(sp)
5  ....
6  fb6: 4c92                lw s9,4(sp)
7  fb8: 4d02                lw s10,0(sp)
8  fba: 6145                addi sp,sp,48
9  fbc: 8082                ret
10
```

Abbildung 19: Assembler Code von malloc()

An diese Adresse wurde nun ebenfalls ein Callback im Simulator registriert. Wie bei Funktionsargumenten musste auch hier in der „Calling Convention“ nachgeschlagen werden, in welchem Register der Return-Wert von der Aufgerufenen Funktion abgelegt wird. In diesem besonderen Fall ebenfalls in den Registern a0-1. Die Callback Implementierung ist in Abbildung 20 dargestellt.

```
static VMIOS_INTERCEPT_FN(addrCB) {
    Uns64 arg[8];
    getArg(processor, object, 0, &arg[0]);
    global_malloc_return = arg[0];
    //vmiMessage("I", "EXC" "_FW", "Intercept 'AddrCB' Val %d",(int)(arg[0]));
    printf("Saved Malloc Addr: 0x%08x \n",global_malloc_return);
}
```

Abbildung 20: Adressen Callback

Das beschriebene Verfahren wurde ebenfalls für die Funktion free() angewandt um neben der Bereitstellung ebenfalls die Freigabe des Speichers dokumentieren zu können. Das Ergebnis der Erweiterungen kann man nun z.B. über die Konsole ausgeben lassen und verifizieren. Abbildung 21 zeigt den C-Code der Beispielapplikation. In Zeile 6 wird ein Speicherbereich der Größe „0xDEAD“ angefordert und anschließend die Adresse in der Software ausgegeben. Ähnliches geschieht in Zeile 9 und 10 für einen zweiten Pointer.

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  printf("Start Test\n");

  int*ptr  = (int*)malloc(0xDEAD);
  printf("ESW: Address of ptr: 0x%08x \n", ptr);
  *ptr = 0x99;
  int *ptr2  = (int*)malloc(0xCAFFE);
  printf("ESW: Address of ptr2: 0x%08x \n", ptr2);
  *ptr2 = 0x99;

  free(ptr);
  free(ptr2);

  printf("End Test\n");
return 0;
}
```

Abbildung 21: Beispiel Applikation

```
xcelium> run

Start Test
Saved Malloc Size: 0x0000dead
Saved Malloc Addr: 0x00010620
  ESW Address of ptr: 0x00010620
Saved Malloc Size: 0x000caffe
Saved Malloc Addr: 0x0001e4d8
  ESW  Address of ptr2: 0x0001e4d8
Freed Malloc Addr: 0x00010620
Freed Malloc Addr: 0x0001e4d8

End Test
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

Abbildung 22: Simulationsergebnis

Das Ergebnis der Simulation ist in Abbildung 22 zu sehen. Die Ausgaben aus der eigentlichen Software sind etwas eingerückt dargestellt. Die geschriebene Intercept-Library speichert eine Allokation der Größe 0xdead und den Pointer zum Start des Adressbereichs 0x10620. Die zusammengehörenden Ausschnitte von Software und Simulation sind mit einer grünen Box markiert. Dasselbe gilt für die zweite Allokation (rote Box). Am Ende der Simulation sieht man ebenfalls die Ergebnisse für das Instrumentierte free() im Simulator. Hier werden die zuvor beschriebenen Adressbereiche wieder freigegeben. Das im letzten Jahr vorgestellte Konzept wurde weiter verfeinert und stützt sich in einigen Aspekten auf die Ideen aus dem Research-Paper zum AdressSanitizer von Google [3] und ist in Abbildung 23 dargestellt. Die aus der Interception Library gewonnen Informationen werden über eine Backdoor in einen sogenannten Shadow Memory geschrieben. Dieser Shadow Memory hat zusätzlich die Fähigkeit alle auftretenden Transfers innerhalb der Plattform zu Analysieren und kann je nach Implementierter Logik bestimmte Fehler finden.

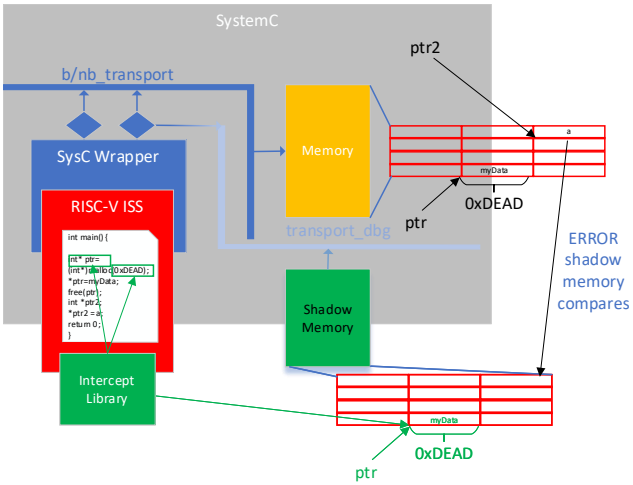

Abbildung 23: Konzept SW-Hardening

Im weiteren Verlauf des Projektes wurden noch zwei offene Punkte bearbeitet. Zum einen das Erkennen des zurückgelieferten Wertes der malloc-Funktion, sowie das Übertragen dieser Werte in die SystemC-Welt. Bisher wurden diese Werte lediglich erkannt und innerhalb des ISS des RISC-V Prozessors auf die Konsole umgeleitet. Zuvor wurde die Adresse der Rückgabeinstruktion der malloc-Funktion konkret in die Intercept Library eingetragen wie in Abbildung 24 zu sehen, würde also die Intercept Library bei der Adresse 0xFBC eingreifen und den Wert aus den Prozessor Registern lesen.

```
0000e18 <_malloc_r>:
e18: 7179                addi sp,sp,-48
e1a: ce4e                sw s3,28(sp)
e1c: d606                sw ra,44(sp)
....
fb6: 4c92                lw s9,4(sp)
fb8: 4d02                lw s10,0(sp)
fba: 6145                addi sp,sp,48
fbc: 8082                ret
```

Abbildung 24: Adresse der "return" Instruktion

Das hat jedoch den Nachteil, dass das Ändern der Applikation und ein anschließendes Kompilieren dazu führen kann, und auch meistens wird, dass die „ret“ Instruktion nicht mehr an derselben Adresse wie vorher liegt. Daher müsste man die Intercept Library nach jedem Ändern der Applikation ebenfalls ändern und neu kompilieren. Um das zu verhindern, wurde diese Erkennung nun zweistufig vorgenommen.

```
static VMIOS_INTERCEPT_FN(malloc_r_in_CB) {
    // Creating Callback to the End of the _malloc_r function
    //printf("ADR: 0x%08x\n",_malloc_r_addr+_malloc_r_return_offset);
    vmirtAddPCCallback(processor, _malloc_r_addr+_malloc_r_return_offset, malloc_r_out_CB, object);
}
```

Abbildung 25: Malloc_r Symbol und Callback

```
// Find the Address of _malloc_r symbol
vmiSymbolCP symbol_constr = vmirtGetSymbolByName(processor, "_malloc_r");
//printf("Malloc Addr= 0x%08lx\n", vmirtGetSymbolAddr(symbol_constr));
_malloc_r_addr = vmirtGetSymbolAddr(symbol_constr);
```

Abbildung 26: Berechnung der Return-Adresse von malloc_r

Das Symbol „malloc_r“ ist der benötigte Einsprungspunkt, der einfach innerhalb der Applikation gefunden werden kann (Abbildung 25), was im Vergleich mit einer einfachen Assembly-Instruktion in dem Zusammenhang nicht funktioniert. Innerhalb dieser Funktion können wir die Adresse des Symbols ermitteln, wissen also die Startadresse. Der Offset der Return Funktion innerhalb der malloc-Funktion ist ein konstanter Wert. Daher können wir innerhalb des Callbacks eine weitere Callback-Funktion registrieren, die sensitiv auf die Adresse „malloc_r_addr+malloc_r_return_offset“ ist. Abbildung 26 zeigt das Auslesen der Adresse vom „malloc_r“ Einsprung. Die Registrierung des neuen Callbacks ist in Abbildung 25 gezeigt.

```
int global_malloc_return;
static VMI_PC_WATCH_FN(malloc_r_out_CB) {
    // Callback for the End of the _malloc_r function
    // userData consists poitner to object
    Uns64 arg[8];
    getArg(processor, userData, 0, &arg[0]);
    global_malloc_return = arg[0];
    print_malloc_returned_address(global_iss_value,global_malloc_return);
    //printf("    IRC:2345 Intercepted Malloc Return Addr: 0x%08x \n", global_malloc_return);

}
```

Abbildung 27: Neuer Callback für das Ende der malloc_r Funktion

In den soeben beschriebenen Callbacks ist es uns dann wieder möglich anhand der RISC-V „calling convention“, wie bereits im Bericht zum letzten Jahr beschrieben, die generierte Adresse über die Register auszulesen und weiterzuleiten (Abbildung 27).

Die Übertragung der gewonnenen Daten in die SystemC-Welt ist über einen globalen Pointer und mehrere externe Funktionen geregelt. Die Funktion *print_malloc_returned_address(global_iss_value, global_malloc_return),* wie in Abbildung 27 dargestellt, liegt extern innerhalb der Observer Klasse in SystemC. Mit Hilfe der Adresse kann dann innerhalb der Observer Klasse jegliche Member Variable oder Methode aufgerufen werden.

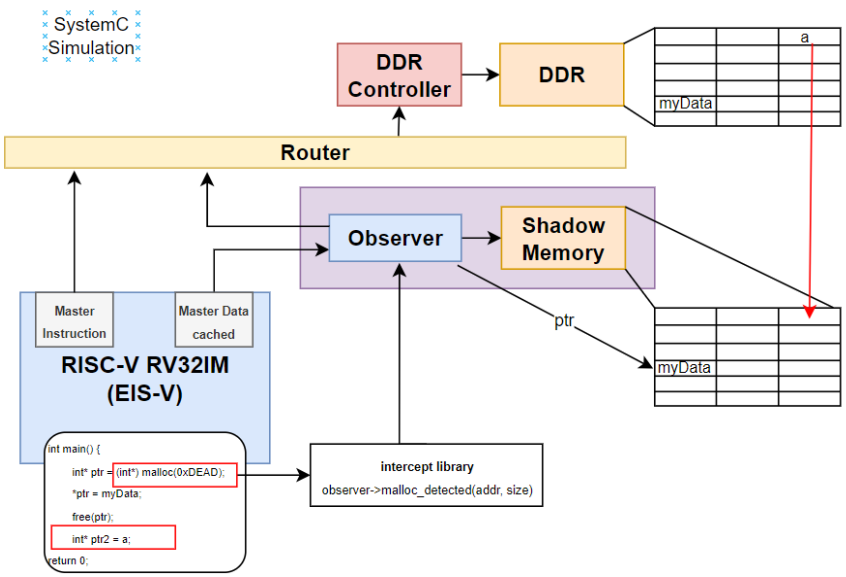

Abbildung 28: Aufbau des SW-Hardening Frameworks

Abbildung 28 zeigt den Aufbau des entwickelten SW-Hardening Frameworks. Ein *malloc()* Befehl wird beispielsweise von der Intercept Library abgefangen, und alle relevanten Daten per Funktionsaufruf an die Observer-Klasse übergeben. Innerhalb des Observers wird dann z.B. in einer Art Shadow Memory festgehalten, welcher Bereich durch das *malloc()* reserviert wurde und somit ein Zugriff auf diese Speicherstelle grundsätzlich erlaubt ist. Zudem kann auch zum Erkennen von Memory Leaks gezählt werden, ob es die gleiche Anzahl m*alloc*()s wie *free()*s gab. Der Observer liest alle Zugriffe, die über den Master Datenport des RISC-V gehen mit und analysiert, ob die Zugriffe erlaubt sind. Hier ist ein Vorteil, dass nicht nur die Befehle des RISC-V kontrolliert werden können. Ebenfalls wäre es möglich zu analysieren, ob zum Beispiel ein falsch konfigurierter DMA auf nicht erlaubte Speicherstellen zugreift. Dazu müssten lediglich zwei weitere Ports am Observer instanziiert werden, über die die DMA-Zugriffe mitgelesen und analysiert werden. Das Schreiben des Wertes „a“ auf die Adresse von ptr2 zum Beispiel, wird vom Observer mitgelesen, und es wird festgestellt, dass an der Adresse kein Speicher reserviert wurde, denn der Pointer ptr2 wurde niemals initialisiert und zeigt auf eine zufällige Adresse im Speicher des Gesamtsystems. Somit wird ein „read from non initialized memory“ erkannt, ohne die Laufzeit der Applikation zu ändern.

```
C-Code malloc if size 0x8
     Shadow Reporter: registered a malloc of size 0x8
     Shadow Reporter: registered malloc returned addr 0x10410
C-Code got Addr: 0x00010410
C-Code malloc if size 0x24
     Shadow Reporter: registered a malloc of size 0x24
     Shadow Reporter: registered malloc returned addr 0x10420
C-Code got Addr: 0x00010420
C-Code malloc if size 0xDEAD
     Shadow Reporter: registered a malloc of size 0xdead
     Shadow Reporter: registered malloc returned addr 0x10448
C-Code got Addr: 0x00010448
C-Code malloc if size 0xBEEF
     Shadow Reporter: registered a malloc of size 0xbeef
     Shadow Reporter: registered malloc returned addr 0x1e300
C-Code got Addr: 0x0001e300
Free:
     Shadow Reporter: registered a free of addr 0x10410
     Shadow Reporter: registered a free of addr 0x10420
     Shadow Reporter: registered a free of addr 0x10448
```

Abbildung 29: Testapplikation

Die Beispielapplikation in Abbildung 29 zeigt die Allokation von 4 Memory-Regionen der Größen 0x8, 0x24, 0xDEAD, 0xBEEF und zeigt die abgefangenen Adressen im Observer sowie die Adressen, die im C-Code verwendet werden, an. Am Ende kann man sehen, dass dreimal die *free()* Funktion aufgerufen wird, also einmal weniger als Speicher reserviert wurde. Daher wird am Ende der Simulation ein Memory-Leak angezeigt (Abbildung 30). Diese Analysen sind auch während der Laufzeit möglich.

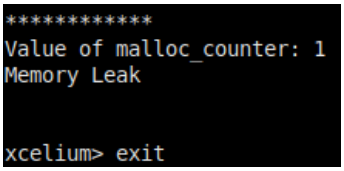

Abbildung 30: Grobes Analyseergebnis der Applikation

4. Wichtige Positionen des zahlenmäßigen Nachweises

5. Notwendigkeit und Angemessenheit der geleisteten Arbeit

Aus Sicht der Cadence Design Systems GmbH war jegliche im Projektkontext geleistete Arbeit notwendig und trug wesentlich zum Erfolg des Projektes bei.

6. Verwertbarkeit der Ergebnisse

Wirtschaftlich

Cadence bietet mit der "Helium Virtual and Hybrid Studio" ein Softwarepaket zum Entwickeln und Nutzen von Virtuellen Prototypen bzw. Plattformen an. Dieses Softwarepaket ermöglicht das Debuggen von Virtuellen Prototypen auf SystemC-Ebene. Weiter unterstützt es das Debuggen der auf dem Virtuellen Prototypen auszuführenden Software. Um das Softwarepaket langfristig attraktiv zu erhalten, ist die fortlaufende Integration von innovativen Funktionen notwendig. Es ist geplant, die in diesem Projekt entwickelten Funktionen zum "SW Hardening" bei entsprechendem Erfolg in das Helium-Softwarepaket zu integrieren. Es wird erwartet, dass sich durch die Funktionen zum "SW Hardening" eine deutlich verbesserte Qualität der auf dem Virtuellen Prototypen entwickelten Software und damit verbunden eine Verkürzung der Projektlaufzeiten erzielen lassen. Damit sollte diese Funktion für unsere Kunden attraktiv erscheinen. Cadence würde dabei von den erzielten Lizenzeinnahmen profitieren.

Wissenschaftlich

Die Ergebnisse aus diesem Projekt sind veröffentlicht worden, soweit hierdurch die wirtschaftliche Verwertbarkeit nicht eingeschränkt wurde.
Nach der oben beschriebenen Integration der entwickelten Methoden für das Software Hardening mit einem Virtuellen Prototypen in das Helium Softwarepaket, kann jeder Entwickler, der das Helium Softwarepaket nutzt, ebenfalls die neuen Methoden zum Software Hardening verwenden. Darüber hinaus wurde gezeigt, welchen Nutzen der Virtuelle Prototyp für die Analyse des vorhandenen Entwurfsraumes hat. In der Veröffentlichung [5] wurde gezeigt, dass der VP enorme Vorteile bezüglich Genauigkeit, Performance und Skalierbarkeit bietet, mit denen man die Qualität zu entwickelnder Systeme in Zukunft massiv verbessern kann.

7. Fortschritte auf dem Gebiet des Vorhabens bei anderen Stellen

Der Cadence Design Systems GmbH sind keine Fortschritte auf dem Gebiet des Vorhabens bei anderen Stellen bekannt.

8. Erfolgte und geplante Veröffentlichungen der Ergebnisse

[1] G. B. Thieu et al., "ZuSE Ki-Avf: Application-Specific AI Processor for Intelligent Sensor Signal Processing in Autonomous Driving," 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 2023, pp. 1-6, doi: 10.23919/DATE56975.2023.10136978.

[2] F. Kautz et al., "Multi-Level Prototyping of a Vertical Vector AI Processing System", 2024 35th IEEE International Conference on Application-specific Systems (ASAP), Architectures and Processors, Hongkong

III. Literaturverzeichnis

[3] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12). USENIX Association, USA, 28.

[4] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42, 6 (June 2007), 89–100. https://doi.org/10.1145/1273442.1250746

[5] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15). IEEE Computer Society, USA, 46–55.

[6] S. Nolting, F. Giesemann, J. Hartig, A. Schmider, G. Paya-Vaya: Application-specific soft-core vector processor for advanced driver assistance systems, IEEE 27th International Conference on Field Programmable Logic and Applications (FPL), 2017

Erfolgskontrolle

Wird als separates Dokument eingereicht.

Berichtsblatt

Wird als separates Dokument eingereicht.

ZuSE-KI-AVF

Anwendungsspezifischer KI-Prozessor für die intelligente Sensorsignalverarbeitung im autonomen Fahren

Kurzbericht

Cadence Design Systems GmbH

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

Zuwendungsempfänger: Cadence Design Systems GmbH
Förderkennzeichen: 16ME0060K
Vorhabenbezeichnung: Anwendungsspezifischer KI-Prozessor für die intelligente Sensorsignalverarbeitung im autonomen Fahren – ZuSE-KI-AVF
Laufzeit des Vorhabens: 01.10.2020 – 30.06.2024

Im Rahmen dieses Projektes sollte eine konfigurierbare, massiv-parallele Prozessorarchitektur als Open-Source IP-Core Komponente erarbeitet werden, die sich für Anwendungen der künstlichen Intelligenz im Bereich Advanced Driver Assistance Systems (ADAS) eignet. Hieraus ergibt sich eine Flexibilität, die Prozessorarchitektur sowohl in ASIC- als auch in FPGA-basierten System-on-Chips (SoCs) als Embedded IP-Core Komponente einzusetzen. Eine optimale Abbildung der Architektur auf verschiedenen Plattformen ist dabei von besonderer Bedeutung, um die durch die Technologie bereitgestellte maximale Taktfrequenz und damit den höchsten Datendurchsatz zu erreichen.

Zahlreiche in ADAS-Anwendungen genutzte Sensorsysteme wie RADAR, LiDAR oder Kameras liefern hochauflösende, mehrdimensionale Messdaten, deren Verarbeitung anwendungsbedingt in Echtzeit erfolgen muss. Um in einem Fahrszenario aus dieser Datenmenge komplexe und sinnvolle Schlussfolgerungen zu extrahieren, haben Methoden des Deep Learning (DL) große Erfolge erzielt und sind somit in modernen Fahrerassistenzsystemen unerlässlich. Dabei stellen einerseits hohe Datenmengen aus einer Vielzahl von Sensoren und andererseits die massive Rechenleistungsanforderung speziell bei der Ausführung von KI-Algorithmen (z.B. Convolutional Neural Networks - CNNs) große Herausforderungen an die zugrundeliegende Hard- und Software dar. Weitere anwendungsbezogene Aspekte bilden außerdem die Energieeffizienz, Robustheit, Flexibilität, die Bereitstellung von funktionaler Sicherheit sowie die IP-Security. Demzufolge bedarf es eines konfigurierbaren KI-Prozessors, um diese Anforderungen zu erfüllen.

Die Aufgabengebiete seitens der Cadence Design Systems GmbH war zum einen das Verfeinern sowie die Verbreitung einer Methodik zur Entwurfsraumexploration (Design Space Exploration oder DSE) basierend auf Virtuellen Prototypen (VP). In Zusammenarbeit mit den Projektpartnern wurde für den in ZusE-KI-AVF entwickelten SoC ein VP des Gesamtsystems erstellt. Anhand des Prototypen konnten viele wichtige Fragen zur Architektur bereits frühzeitig erörtert und beantwortet werden, so dass wesentliche Änderungen noch vor Produktionsstart des eigentlichen Chips möglich waren.

Der Virtuelle Prototyp wurde verwendet, um den Entwurfsraum jenseits von auf einem FPGA realisierbarer Konfigurationen zu explorieren, und dabei zusätzliche Informationen bezüglich Cache Hit/Miss-Raten, Speicher- sowie Interconnect-Auslastung zu generieren. Ein weiterer Aspekt war die Implementierung einer minimalinvasiven Technik zur Verbesserung der Qualität eingebetteter Software. Mit Hilfe des entwickelten Frameworks und des Virtuellen Prototypen kann entwickelte Software auf vorhandene Fehler (z.B. Zugriff auf nicht initialisierten Speicher) analysiert werden, ohne dabei einen Einfluss auf die Ausführung der Software oder die Verfügbarkeit von Ressourcen auf einem eingebetteten System zu nehmen.