Logiksynthese für In-Memory-Computing mit resistiven Speichern[1]

Saeideh Shirinzadeh[2]

Abstract: Die wachsende Notwendigkeit, das Problem des Speicherengpasses in den gegenwärtigen Computerarchitekturen zu umgehen, hat zu einer großen Aufmerksamkeit für das In-Memory-Computing geführt, welches durch aufkommende Speichertechnologien wie RRAM (Resistive Random Access Memory) ermöglicht wird. Diese Dissertation untersucht In-Memory-Computing aus zwei Perspektiven, nämlich "customized" und befehlsbasiert. Der customized-Ansatz nutzt logische Repräsentationen, um In-Memory-Computing-Schaltungen zu realisieren. Der Ansatz schlägt Designansätze und Optimierungsalgorithmen für jede Repräsentation in Bezug auf Fläche und Latenz bei der Realisierung ihrer logischen Grundelemente vor. Beim befehlsbasierten Ansatz wird ein automatischer Compiler eingesetzt, um Logic-in-Memory-Computerarchitektur zu verwenden und resultierende Programme zu optimieren. Die experimentellen Ergebnisse für beide Ansätze zeigen erhebliche Verbesserungen gegenüber dem Stand der Technik.

1 Einführung

Die Entwicklungsfortschritte bei den Prozessoren moderner Computer übertreffen die des Arbeitsspeichers in Bezug auf Zugriffzeit bei Weitem. Die wesentlich höhere Latenzzeit des Speichers im Vergleich zu einem Prozessor einerseits und die Kommunikation zwischen diesen beiden in der von-Neumann-Architektur andererseits schränken die Gesamtleistung aktueller Computersysteme ein, was als *memory wall* bezeichnet wird. Neue Anwendungen wie *Internet der Dinge* (*engl*. Internet of Things: IoT) und *big data*, die mit großen Datenmengen umgehen, stellen hohe Anforderungen, was zu intensiver Forschung führt [So17]. Unter den möglichen Ansätzen erscheint *in-memory computing* als sehr vielversprechend. Die Integration der Speicher und Rechenparadigmen erlaubt es, die "memory wall" zu überwinden und die Leistung um ein Vielfaches zu steigern.

Eine der Kerntechnologien für In-Memory-Computing, ist RRAM (Resistive Random Access Memory). RRAM ist eine vielversprechende nicht-volatile Speichertechnologie mit hoher Skalierbarkeit und ohne Energieverlust im Standby, bei der der Innenwiderstand zwischen zwei Zuständen (hoch/niedrig) umgeschaltet werden kann. Es wurden verschiedene Ansätze vorgeschlagen, die diese resistive Schalteigenschaft nutzen, um logische Operationen innerhalb von RRAMs auszuführen. Der Gebrauch von *Material Implication* (IMP) wurde in großem Umfang für In-Memory Computing verwendet [Bo10]. In [Kv14] wurde eine Klasse logischer Operationen namens MAGIC vorgeschlagen, mit der Boolesche Funktionen mit NOR- und NOT-Gattern realisiert werden können. In [Ga16] wurde gezeigt, dass sich eine Majority-basierte Logik für In-Memory-Computing verwenden lässt, da RRAM Majority (MAJ) nativ implementiert werden kann und es somit

[1] Synthesis and Optimization for Logic-in-Memory Computing using Memristive Devices

[2] Universität Bremen, Fachbereich Mathematik und Informatik / Cyber-Physical Systems, DFKI GmbH Bibliothekstr. 5, 28359 Bremen, Deutschland

möglich ist, Mehrheitsvergleiche als Logikbausteine für In-Memory-Computing zu verwenden wird.

In dieser Dissertation untersuchen wir die Synthese für In-Memory-Computing aus zwei verschiedenen Perspektiven, d.h. (i) basierend auf einem „customized" Ansatz auf Gatterebene, die logische Repräsentationen verwendet, und (ii) einem befehlsorientierten Ansatz für die effiziente Kompilierung und Ausführung von Programmen auf einer Logic-in-Memory-Architektur. Der vorgestellte customized-Syntheseansatz verwendet IMP und MAJ als grundlegende Operationen für RRAM-Arrays, während der befehlsbasierte Ansatz MAJ nur für die Ausführung von Programmen verwendet.

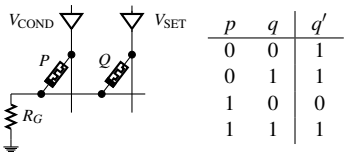
Der customized-Syntheseansatz beginnt mit der Suche nach effizienten Realisierungen für die logischen Grundelemente der verwendeten Repräsentationen, d.h. *Binary Decision Diagrams* (BDDs), *AND-Inverter-Graphen* (AIGs) und *Majority-Inverter-Graphen* (MIGs). Der Ansatz bietet Optimierungsalgorithmen und eine umfassende Entwurfsmethodik, um jede Darstellung in äquivalente Sequenzen von Operationen und RRAMs abzubilden, die auf einem resistiven Speicherarray ausgeführt werden sollen. Die Ergebnisse des vorgestellten customized-Ansatzes zeigen eine deutliche Verbesserung gegenüber herkömmlichen Ansätzen.

Durch den befehlsbasierten Ansatz automatisieren und optimieren wir vollständig eine vorhandene Computerarchitektur. Darüber hinaus befassen wir uns mit dem Problem der geringeren Schreibdauer von RRAM-Schaltern und schlagen Techniken zur Verschleißminderung vor, um die Lebensdauer der Architekturen zu erhöhen. Experimente, die mit großen Arithmetik- und Steuerfunktionen durchgeführt wurden, zeigen eine beträchtliche Verbesserung der Verteilung von Schreibvorgängen im gesamten Speicherfeld. Auch die Anzahl der Zyklen und der RRAM-Vorrichtungen, die die Latenz und den Bereich der resultierenden Implementierungen darstellen, konnte reduziert werden.

2 Logikoperationen innerhalb von RRAM

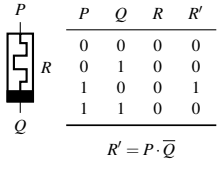
In [Bo10] wurde gezeigt, dass die Material Implication (IMP), d.h. $q' \leftarrow p \,\mathrm{IMP}\, q = \overline{p} + q$, aus der Interaktion zweier RRAM-Schalter ausgeführt werden kann; unter bestimmten Spannungspegeln, die durch V_{SET} und V_{COND} in Abb. 1 (a) angegeben sind. Die logischen Zustände der resistiven Schalter können auch einfach zwischen logischer 1 oder 0, d.h. FALSE-Betrieb, umgeschaltet werden, wenn sie an geeignete Spannungsimpulse angelegt werden. IMP und FALSE bilden zusammen einen universellen Satz von Logikoperationen, die ausreichen, um alle Boolesche Funktionen auf resistiven Arrays auszuführen.

In [Ga16] wurde eine von RRAM-Schaltern aktivierte intrinsische Majoritätsoperation eingeführt, die ausreicht, um jede Boolesche Funktion zu berechnen. Bezeichnen wir die obere und die untere Elektrode eines RRAM-Schalters mit P und Q (siehe Abb. 1 (b)). Angenommen, der aktuelle resistive Zustand des Schalters (R) kann durch Anlegen eines positiven oder negativen Spannungspegels V_{PQ} auf 1 oder 0 umgeschaltet werden, dann ändert sich der nächste Zustand des RRAMs (R')) basierend auf den in Abb. 1 (b) gezeigten Wahrheitstabellen. Durch Erweitern der Booleschen Relation in den Tabellen kann einfach gezeigt werden, dass der nächste Widerstandszustand des Schalters



p	q	q'
0	0	1
0	1	1
1	0	0
1	1	1

(a)



P	Q	R	R'
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	0

$R' = P \cdot \overline{Q}$

P	Q	R	R'
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	1

$R' = P + \overline{Q}$

(b)

Abb. 1: Die Booleschen Operationen, die innerhalb von RRAM ausgeführt werden können. (a) Implementierung des IMPs und seine Wahrheitstabelle [Bo10]. (b) Die intrinsische Majority Operation innerhalb eines RRAM-Schalters (MAJ) [Ga16].

$R' = M(P, \overline{Q}, R) = P \cdot \overline{Q} + P \cdot R + \overline{Q} \cdot R$ ist, d.h. das Ergebnis der Majoritätsfunktion mit drei Eingängen, die in diesem Dokument mit MAJ bezeichnet wird.

3 Customized Syntheseansatz

Der customized Syntheseansatz umfasst drei Stufen: (i) Finden effizienter Realisierungen mit RRAM-Schaltern für das logische Grundelement jeder Repräsentation BDD, AIG und MIG, (ii) Definieren der Entwurfsmethodik zum Abbilden der Repräsentationen auf das RRAM-Array (iii) und schließlich Optimieren der Repräsentationen mit Bezug auf die Anzahl der RRAMs und Rechenoperationen, die durch die Entwurfsmethodik bestimmt werden. Im Folgenden erläutern wir kurz den vorgeschlagenen customized Ansatz für die Logikdarstellung MIG und geben ein Implementierungsbeispiel. Aus Platzgründen verweisen wir Leserinnen und Leser auf [Sh18], wo Einzelheiten zu den Optimierungsalgorithmen für jede Darstellung beschrieben sind.

Die IMP-basierte Realisierung [Sh16, Sh18] für das Majority-Gatter ist im Folgenden dargestellt. Die Realisierung erfordert sechs RRAMs, Eingabeschalter X, Y und Z und zusätzliche Schalter A, B und C, die zum Negieren oder Speichern der Operationsausgänge erforderlich sind. Die Majority-Funktion wird nach zehn Operationen ausgeführt. Bei der ersten Operation werden die erforderlichen Schalter mit den Eingabevariablen und Null geladen, und die restlichen Schritte umfassen die Operationen IMP und FALSE.

01: $X = x, Y = y, Z = z$ $A = 0, B = 0, C = 0$	**06:** $c \leftarrow y \,\text{IMP}\, c = \overline{x + y}$
02: $a \leftarrow x \,\text{IMP}\, a = \bar{x}$	**07:** $c \leftarrow z \,\text{IMP}\, c = \overline{x \cdot z + y \cdot z}$
03: $b \leftarrow y \,\text{IMP}\, b = \bar{y}$	**08:** $a = 0$
04: $y \leftarrow a \,\text{IMP}\, y = x + y$	**09:** $a \leftarrow b \,\text{IMP}\, a = x \cdot y$
05: $b \leftarrow x \,\text{IMP}\, b = \bar{x} + \bar{y}$	**10:** $a \leftarrow c \,\text{IMP}\, a = x \cdot y + y \cdot z + x \cdot z.$

Es ist offensichtlich, dass die MAJ-basierte Realisierung für die MIG-basierte Synthese durch die Nutzung der nativ implementierten Majority-Funktion in RRAM-Schaltern effizienter realisiert werden kann. Wie im Folgenden gezeigt, erfordert die Realisierung eines Majority-Gatters in diesem Fall maximal vier Schalter und drei Schritte.

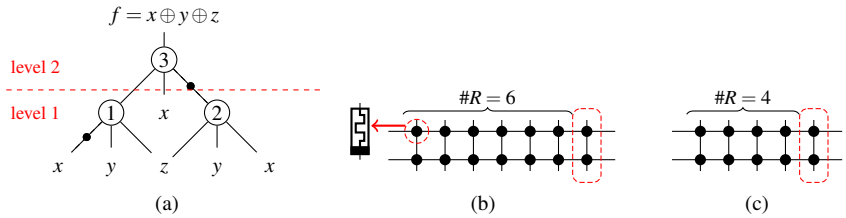


Abb. 2: (a) MIG, der ein Drei-Bit-XOR-Gatter darstellt, und eine obere Grenze der RRAM Crossbar für deren Implementierung mit (b) IMP-basierter and (c) MAJ-basierter Realisierung.

$$\textbf{1: } X = x, Y = y, Z = z, A = 0$$
$$\textbf{2: } P_A = 1, Q_A = y, R_A = 0 \Rightarrow R'_A = \bar{y}$$
$$\textbf{3: } P_Z = x, Q_Z = \bar{y}, R_Z = z \Rightarrow R'_Z = M(x, y, z).$$

Der MIG wird zunächst hinsichtlich der Anzahl der RRAMs und der Rechenschritte optimiert. Wir schlagen drei verschiedene Optimierungsalgorithmen im Bezug auf eine oder beide Kostenmetriken vor. Weitere Informationen können unter Optimierungsalgorithmen in [SSD16, Sh18] gefunden werden. Anschließend können sie auf RRAM-Arrays gemäß einer Level-by-Level-Entwurfsmethodik implementiert werden. Dies bedeutet, dass ausgehend vom unteren Rand des Diagramms alle Knoten in jeder MIG-Ebene gleichzeitig berechnet werden. Nach der Berechnung der einzelnen Ebenen werden die RRAM-Schalter freigegeben und können als Eingabeschalter für die Berechnung der nächsten Ebene verwendet werden. Diese Prozedur wird fortgesetzt, bis die Wurzelfunktion berechnet ist.

Zur Berechnung jedes Levels umfasst die Anzahl der erforderlichen RRAMs das Sechsfache (mit der IMP-basierte Realisierung) oder das Vierfache (mit der MAJ-basierte Realisierung) der Anzahl der Knoten in der Ebene. Für jede ergänzte Kante wird auch ein zusätzlicher RRAM-Schalter berücksichtigt. Da die RRAMs wiederverwendet werden, entspricht die Anzahl der Schalter, die zur Berechnung eines MIGs erforderlich sind, der maximalen Anzahl der erforderlichen Schalter über alle Ebenen. Die Anzahl der Berechnungsschritte zum Berechnen eines MIGs beträgt mindestens zehn oder dreimal die Anzahl der BDD-Stufen für IMP- bzw. MAJ-basierte Realisierungen. Dieser Wert muss jedoch zu der Anzahl der Ebenen hinzugefügt werden, die über komplementierte Kanten verfügen, da ihre Negation zusätzliche Operationen benötigt [Sh18].

Abbildung 2 (a) zeigt einen MIG, der ein XOR-Gatter mit drei Eingängen darstellt. Hier zeigen wir, wie dieses MIG auf einem RRAM-Array mit Schaltern implementiert werden kann, die durch R_{ij} dargestellt werden, wobei i und j die Indizes der Zeile bzw. der Spalte bezeichnen.

Synthese mit IMP-basierter Realisierung. Die vorgestellte Entwurfsmethodik berechnet alle Knoten gleichzeitig auf einer Ebene und weist zu diesem Zweck jedem Knoten eine Zeile zu. Dies bedeutet, dass in jeder Zeile eine einzelne Operation pro Zyklus ausgeführt wird. Da das Beispiel für MIGs eine maximale Ebenengröße von zwei hat, benötigt die erforderliche RRAM crossbar mindestens zwei Reihen mit mindestens sechs Schaltern (siehe Abbildung 2 (b)). Ein zusätzlicher Schalter am Ende jeder Reihe wird für ergänzte

Kanten verwendet. Wie erwartet, ist die Anzahl der erforderlichen Schritte 22, d.h. das 10-fache der Tiefe 2 plus zwei zusätzliche Schritte für die ergänzten Kanten auf beiden Ebenen.

Initialisierung	$R_{ij} = 0;$
1: Laden für Ebene 1	$R_{11} = x, R_{12} = y, R_{13} = z;$ $R_{21} = x, R_{22} = y, R_{23} = z;$
2: Negation für Knoten 1	$R_{17} \leftarrow x \,\mathrm{IMP}\, R_{17} : R_{17} = \bar{x};$
3-11: Berechnung der Ebene 1	Knoten 1: $R_{14} = M(\bar{x}, y, z);$ Knoten 2: $R_{24} : M(x, y, z);$
12: Laden für Ebene 2	$R_{11} = x, R_{12} = M(x, y, z), R_{13} = M(\bar{x}, y, z)$ $R_{14} = R_{15} = R_{16} = R_{17} = 0;$
13: Negation für Knoten 3	$R_{17} \leftarrow R_{12} \,\mathrm{IMP}\, R_{17}$: $R_{17} = \overline{R_{12}} = \overline{M}(x, y, z);$
14-22: Berechnung der Ebene 2	$R_{14} = M(M(\bar{x}, y, z), x, \overline{M(x, y, z)});$

Synthese mit MAJ**-basierter Realisierung.** Die Schritte für die MAJ-basierte Implementierung des MIGs, dargestellt in Abb. 2 (a), werden im Folgenden gezeigt. Wie die Schritte zeigen, wird die XOR-Funktion mit nur drei RRAM Schaltern und innerhalb von nur vier Schritten ausgeführt, trotz der oberen Grenze von 8, die für ein MIG mit zwei Ebenen mit ergänzten Kanten erwartet werden. In der Tat können die komplementierten Kanten an den Knoten 1 und 3 direkt als zweite Eingabe von MAJ verwendet werden, ohne invertiert zu werden. Darüber hinaus können die aktualisierten RRAMs als Eingaben für den nächsten Zyklus verwendet werden, sodass der Schritt des Ladens nicht erforderlich ist. Es sollte beachtet werden, dass das Anwenden von Signalen auf die Zeilen und Spalten während der MAJ-basierten Implementierung eine Datenverzerrung vermeiden soll, indem zuvor berechnete Ergebnisse und die gleichzeitigen Operationen in anderen Zeilen beibehalten werden [Sh18]. Zum Beispiel werden in Schritt 2 die Werte von R_{11} und R_{21} beibehalten, indem die logischen Zustände ihrer Terminale angeglichen werden, wenn eine MAJ-Operation innerhalb von R_{25} ausgeführt wird.

Initialisierung	$R_{ij} = 0 : Q_{ij} = 1, P_{ij} = 0;$
1: Laden	$Q_1 = Q_2 = 0, P_1 = P_2 = z;$ $R_{11} : RM_3(z, 0, 0) = M(z, 1, 0) = z;$ $R_{21} : RM_3(z, 0, 0) = M(z, 1, 0) = z;$
2: Negation für Knoten 2	$Q_1 = Q_2 = x, P_1 = x, P_2 = 1;$ $R_{25} : RM_3(1, x, 0) = M(1, \bar{x}, 0) = \bar{x};$
3: Berechnung der Ebene 1	Knoten 1: $P_1 = y, Q_1 = x, R_{11} = z$ $R_{11} : RM_3(y, x, z) = M(y, \bar{x}, z);$ Knoten 2: $P_1 = y, Q_2 = \bar{x}\ (@R_{25})\ , R_{21} = z;$ $R_{21} : RM_3(y, \bar{x}, z) = M(y, x, z);$

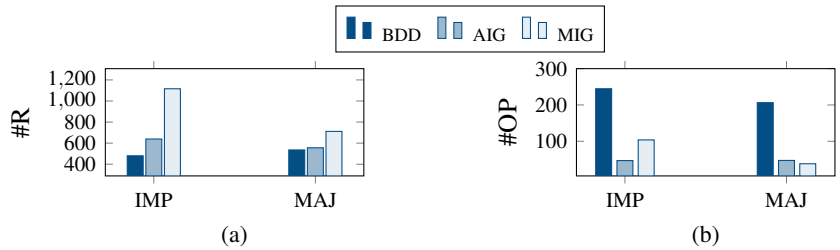


Abb. 3: Vergleich der Syntheseergebnisse anhand von logischen Darstellungen für In-Memory-Computing. (a) Die durchschnittliche Anzahl der RRAM-Schalter, (b) die durchschnittliche Anzahl der Operationen.

4: Berechnung der Ebene 2 $P_1 = x, Q_1 = @R_{21}, R_{11} = M(\bar{x}, y, z);$
$R_{11} : RM_3(x, @R_{21}, @R_{11}) = M(x, \overline{@R_{21}}, @R_{11}) :$
$M(M(\bar{x}, y, z), x, \overline{M}(x, y, z));$

Abb. 3 vergleicht die Syntheseergebnisse auf der Grundlage der drei Darstellungen, wobei sowohl IMP als auch MAJ für die Implementierung verwendet werden. Wie die Abbildung zeigt, benötigt der BDD-basierte Ansatz die geringste Anzahl von Schaltern, führt jedoch zu einer hohen Latenz. Andererseits erfordern Synthesemethoden, die auf AIG und MIG basieren, mehr RRAMs, verringern jedoch die Operationsdauer. Insbesondere der MIG-basierte Ansatz unter Verwendung von MAJ führt zu Implementierungen mit der geringsten Latenzzeit.

4 Befehlsbasierter Syntheseansatz

In [Ga16] wurde eine *Programmable Logic-in-Memory* (PLiM) Computerarchitektur vorgeschlagen, die es erlaubt, logische Operationen an einem regulären RRAM-Array durchzuführen. PLiM verfügt über einen Controller, der aus einer einfachen Zustandsmaschine und einigen Arbeitsregistern besteht, um MAJ-Operationen auszuführen. Es ist nur eine einzige MAJ-Anweisung pro Zyklus zulässig. Eine Anweisung hat das Format $M(P, \bar{Q}, R)$ mit drei zuzuweisenden Operanden. Der erste Operand P ist das an die obere Elektrode der RRAM-Vorrichtung angelegte Signal, d.h. der Zeilentreiber, und der zweite Operand Q ist das an die untere Elektrode angelegte Signal, d.h. der Spaltentreiber. Der dritte Operand R ist der aktuelle Status des zu berechnenden Schalters, der automatisch aktualisiert wird, wenn die Anweisung ausgeführt wird.

Der PLiM-Computer leitet den Befehlssatz zum Berechnen einer Booleschen Funktion aus seiner MIG-Darstellung ab. Die Eigenschaften eines MIGs, dessen Knotenreihenfolge für Berechnungen und die resultierende Zuweisung der Operanden zu jeder Instruktion beeinflussen die Anzahl der benötigten RRAMs und Instruktionen, welche sich wiederum auf die Fläche und die Verzögerungszeit der resultierenden PLiM-Implementierungen auswirken [So17, So16, Sh17b]. In dieser Arbeit schlagen wir einen automatischen Compiler vor, der PLiM-Programme zur Berechnung beliebiger Funktionen generiert und dabei die oben genannten Kostenmetriken berücksichtigt.

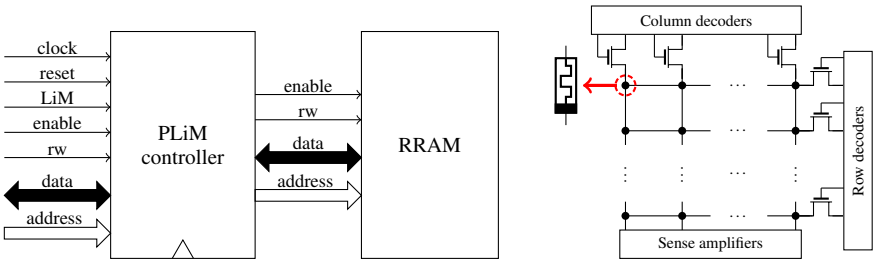
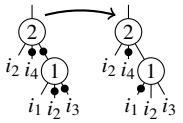


Abb. 4: (a) Die PLiM-Computerarchitektur [Ga16]. (b) Eine Speicherbank eines RRAM Crossbar-Arrays, auf dem ein PLiM-Programm ausgeführt wird.

Da PLiM Berechnungen vollständig seriell durchführt, ist die Anzahl der Knoten in einem MIG, d.h. die Größe des MIGs, ein bestimmender Faktor für die Länge der resultierenden Sequenz von Befehlen. Daher kann das Optimieren der MIGs in Bezug auf die Anzahl von Knoten die Latenz von PLiM-Implementierungen erheblich verbessern. Bei der MIG-Optimierung sollte jedoch nicht nur die Anzahl der Knoten berücksichtigt werden. MAJ benötigt im Idealfall eine komplementäre Kante, um einen Knoten innerhalb einer einzelnen Anweisung zu berechnen. Andernfalls sind ein zusätzlicher Schalter und eine Anweisung erforderlich, um einen Operanden zu invertieren, bevor der Knoten berechnet wird, wodurch die Anzahl und Position der komplementierten Kanten beeinflusst wird. Abb. 5 zeigt einen Beispiel-MIG mit zwei Knoten vor und nach der Optimierung, bei dem der Graph nur bezüglich der komplementierten Kanten geändert wurde. Wie die Abbildung zeigt, reduziert die MIG-Optimierung sowohl die Anzahl der RRAMs als auch die Anzahl der Anweisungen. Wir verweisen Leserinnen und Leser auf [So16] für den MIG-Optimierungsalgorithmus für PLiM.



Vor Optimierung		**Nach Optimierung**	
1: 0, 1, $@R_1$	$R_1 \leftarrow 0$	**1:** 0, 1, $@R_1$	$R_1 \leftarrow 0$
2: 1, i_3, $@R_1$	$R_1 \leftarrow \bar{i}_3$	**2:** i_3, 0, $@R_1$	$R_1 \leftarrow i_3$
3: i_1, i_2, $@R_1$	$R_1 \leftarrow N_1$	**3:** i_2, i_1, $@R_1$	$R_1 \leftarrow N_1$
4: 0, 1, $@R_2$	$R_2 \leftarrow 0$	**4:** i_4, i_2, $@R_1$	$R_1 \leftarrow N_2$
5: 1, $@R_1$, $@R_2$	$R_2 \leftarrow \overline{N}_1$		
6: i_2, i_4, $@R_2$	$R_2 \leftarrow N_2$		

Abb. 5: Die Auswirkungen der MIG-Optimierung auf die Anzahl der RRAMs und Anweisungen.

Unser vorgeschlagener Compiler findet zuerst eine effiziente Reihenfolge von Kandidatenknoten für die Berechnung und übersetzt dann die Knoten in einen MAJ-Befehl, indem den Operanden die kleinste Anzahl von RRAM-Schaltern und -Befehlen zugewiesen wird. Der Compiler berücksichtigt auch die Ursachen für ungleichmäßigen Schreibverkehr, der einige Schalter langfristig viel früher als andere verschleißen lässt, was die Lebensdauer erhöht, indem die Schreibvorgänge über den gesamten Arbeitsspeicher verteilt werden [Sh17a]. Dies ist besonders wichtig, da RRAM-Schalter eine begrenzte Schreibdauer haben, die im Entwurfsprozess berücksichtigt werden sollte.

Abb. 6 zeigt die Anzahl der erforderlichen RRAMs und Anweisungen des vorgeschlagenen befehlsbasierten Ansatzes für EPFL-Benchmarks[1]. Die Ergebnisse sind für naive PLiM-Implementierungen und Implementierungen nach nur der MIG-Optimierung und Kompilierung gezeigt. Die Ergebnisse zeigen Verbesserungen von 19,95% bzw. 61,4% in Bezug auf die Anzahl der Befehle und der RRAM-Schalter. Die Standardabweichung der Schreibvorgänge über die RRAM-Schalter ist in Abb. 7 dargestellt. Der Vergleich der Ergebnisse mit den naiven Implementierungen zeigt eine Verbesserung von 72,17% gegenüber allen Benchmarks.

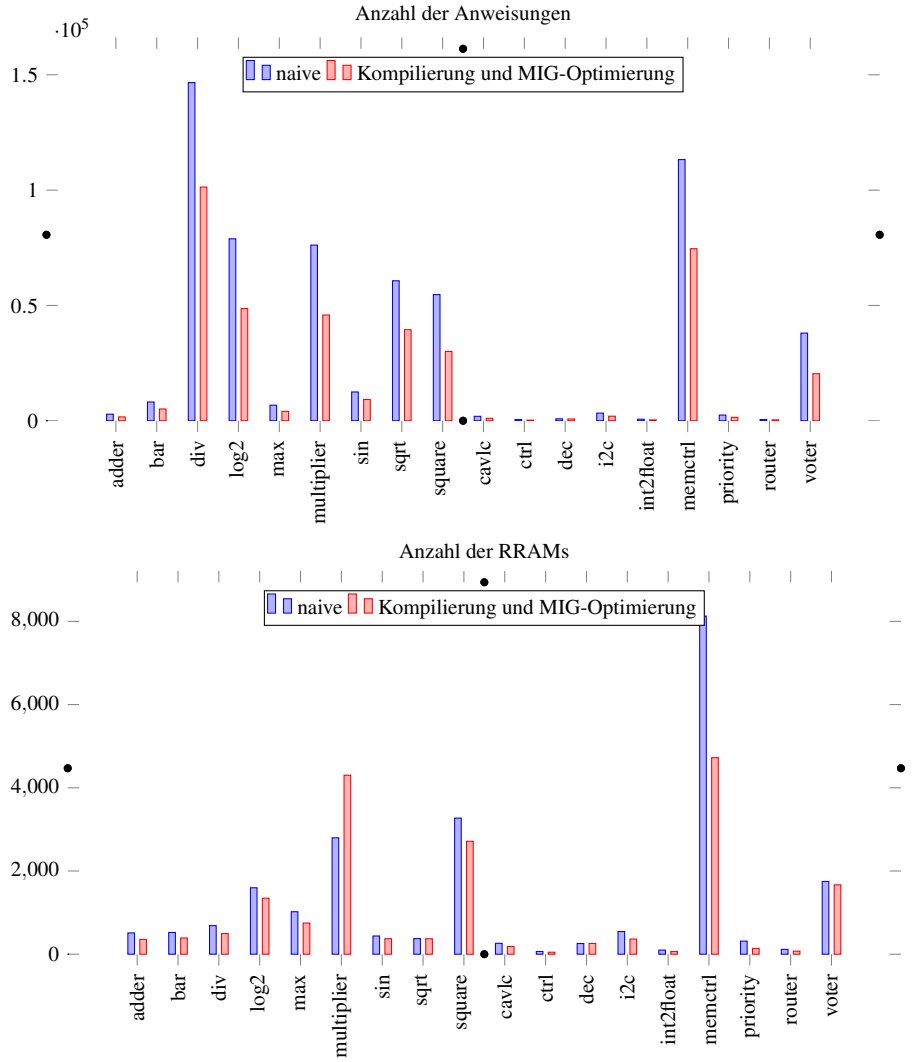


Abb. 6: Die Anzahl der Anweisungen und RRAMs, die von der PLiM-Computerarchitektur benötigt werden.

[1] http://lsi.epfl.ch/benchmarks

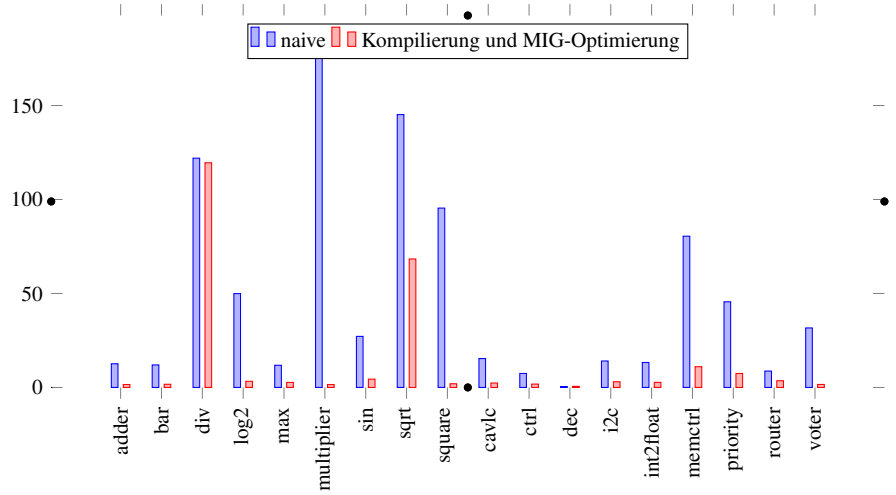


Abb. 7: Standardabweichung der Schreibvorgänge für die PLiM-Architektur.

5 Zusammenfassung

Diese Dissertation präsentiert einen umfassenden Ansatz für die Synthese von Logic-in-Memory-Schaltungen unter Verwendung der logischen Repräsentationen BDD, AIG und MIG. Der vorgestellte Ansatz führt die Realisierung der Logik-Grundelemente mit zwei grundlegenden Operationen ein, die durch RRAM-Schalter ermöglicht werden, und stellt Optimierungsalgorithmen und Entwurfsmethodiken für die Crossbar-Implementierung bereit. Die Arbeit schlägt auch einen automatischen Compiler für eine reguläre Logic-in-Memory-Computerarchitektur vor und verbessert die Ausführungskosten in Bezug auf Latenz, Fläche und Schreibgleichgewicht erheblich. Die Beiträge dieser Dissertation entwickeln einen umfassenden Rahmen für das vielversprechende Gebiet des Logic-in-Memory-Computing. Damit bietet sie eine solide Grundlage für dieses aufstrebende Computer-Paradigma, welches verschiedene Anwendungen ermöglicht, und schafft Möglichkeiten für die weitere Forschung und Entwicklung in den verschiedenen Bereichen programmierbarer Hardware und Architekturen von resistiven Speichern.

Danksagung

Diese Arbeit entstand im Graduiertenkolleg System Design (SyDe) der Universität Bremen, das im Rahmen der Exzellenzinitiative finanziert wird.

Literatur

[Bo10] Borghetti, J.; Snider, G.S.; Kuekes, P.J.; Yang, J.J.; Stewart, D.R.; Williams, R.S.: Memristive switches enable stateful logic operations via material implication. Nature, 464(7290):873–876, 2010.

[Ga16] Gaillardon, Pierre-Emmanuel; Amarù, Luca Gaetano; Siemon, Anne; Linn, Eike; Waser, Rainer; Chattopadhyay, Anupam; De Micheli, Giovanni: The Programmable Logic-in-Memory (PLiM) computer. In: Design, Automation & Test in Europe. S. 427–432, 2016.

[Kv14] Kvatinsky, S.; Belousov, D.; Liman, S.; Satat, G.; Wald, N.; Friedman, E.G.; Kolodny, A.; Weiser, U.C.: MAGIC – Memristor-Aided Logic. IEEE Trans. Circuits Syst. II, 61(11):895–899, 2014.

[Sh16] Shirinzadeh, Saeideh; Soeken, Mathias; Gaillardon, P.-E.; Drechsler, Rolf: Fast logic synthesis for RRAM-based in-memory computing using Majority-Inverter Graphs. In: Design, Automation & Test in Europe. S. 948–953, 2016.

[Sh17a] Shirinzadeh, Saeideh; Soeken, Mathias; Gaillardon, P.-E.; De Micheli, Giovanni; Drechsler, Rolf: Endurance management for resistive Logic-In-Memory computing architectures. In: Design, Automation & Test in Europ. S. 1092–1097, 2017.

[Sh17b] Shirinzadeh, Saeideh; Soeken, Mathias; Gaillardon, Pierre-Emmanuel; Drechsler, Rolf: Logic Synthesis for Majority Based In-Memory Computing. In (Vaidyanathan, Sundarapandian; Volos, Christos, Hrsg.): Advances in Memristors, Memristive Devices and Systems, S. 425–448. Springer International Publishing, 2017.

[Sh18] Shirinzadeh, S.; Soeken, M.; Gaillardon, P.-E.; Drechsler, R.: Logic Synthesis for RRAM-based In-Memory Computing. IEEE Trans. on CAD of Integrated Circuits and Systems, 37(7):1422–1435, 2018.

[So16] Soeken, Mathias; Shirinzadeh, Saeideh; Gaillardon, P.-E.; Amarù, Luca Gaetano; Drechsler, Rolf; De Micheli, Giovanni: An MIG-based compiler for programmable logic-in-memory architectures. In: Design Automation Conference. S. 117:1–117:6, 2016.

[So17] Soeken, Mathias; Gaillardon, P.-E.; Shirinzadeh, Saeideh; Drechsler, Rolf; De Micheli, Giovanni: A PLiM Computer for the Internet of Things. IEEE Computer, 50(6):35–40, 2017.

[SSD16] Shirinzadeh, Saeideh; Soeken, Mathias; Drechsler, Rolf: Multi-objective BDD optimization for RRAM based circuit design. In: IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems. S. 46–51, 2016.

Saeideh Shirinzadeh erhielt ihren B.Sc. (2010) und M.Sc. (2012) in Elektrotechnik an der University of Guilan, Iran. Nach einer Tätigkeit als Dozentin war sie von 2014 bis 2018 Doktorandin an der Universität Bremen, Fachbereich Mathematik und Informatik, in der Arbeitsgruppe Rechnerarchitektur (Leitung: Prof. Dr. Rolf Drechsler). Sie verteidigte ihre Doktorarbeit mit dem Prädikat *summa cum laude* im Oktober 2018. Seit 2019 arbeitet sie als Postdoc-Forscherin am Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI). Ihre Forschungsinteressen konzentrieren sich auf Logiksynthese, In-Memory-Computing, Mehrziel-Optimierung und evolutionäre Berechnungen.