Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Planarisierung von Datenflussgraphen bei der Codegenerierung

Bachelor's Thesis

by

Yannic Schopfer

September 17, 2025

University of Kaiserslautern-Landau
Department of Computer Science
67663 Kaiserslautern
Germany

Examiner: Prof.Dr. Klaus Schneider
M.Sc Nadine Kercher

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Planarisierung von Datenflussgraphen bei der Codegenerierung “ selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 17.9.2025

Yannic Schopfer

Abstract

Buffered Exposed Datapath (BED) architectures such as SCAD move data transport, scheduling, and processing-unit (PU) assignment to the compiler. In this setting, FIFO buffer states and edge crossings in dataflow graphs (DFGs) cause conflicts that are commonly addressed with copy nodes (for FIFO-conform leveling) and swap nodes (for critical crossings), at the cost of additional compute and memory. This thesis asks how to minimize the number of newly inserted copy and swap nodes under a *fixed* node order and *fixed* PU allocation.

A two-stage heuristic is proposed: (i) a deterministic queue simulation with dynamic lookahead and planned give-back records exactly those rotations that are truly required and derives a minimal set of copy nodes; (ii) remaining per-channel conflicts are detected as adjacent inversions and resolved by locally inserted swap nodes. Both stages preserve FIFO semantics, the original node order, and the PU assignment of existing nodes.

Zusammenfassung

Buffered-Exposed-Datapath-(BED)-Architekturen wie SCAD verlagern Datentransport, Scheduling und PU-Zuordnung in die Codegenerierung. Dabei führen FIFO-Zustände und Kantenkreuzungen in Datenflussgraphen (DFGs) zu Konflikten, die üblicherweise durch Copy-Knoten (für FIFO-konformes Leveling) und Swap-Knoten (für kritische Kreuzungen) entschärft werden – jedoch mit zusätzlichem Rechen- und Speicheraufwand. Diese Arbeit untersucht, wie sich die Anzahl neu einzufügender Copy- und Swap-Knoten bei *fester* Ausführungsreihenfolge (Node-Order) und *fester* PU-Allokation minimieren lässt.
Eine zwei-stufige Heuristik wird vorgeschlagen: (i) Eine deterministische Simulation der Pufferdynamik mit dynamischer Vorausschau und geplanter Rückgabe identifiziert genau jene Rotationen, die tatsächlich notwendig sind, und leitet daraus eine minimale Menge an Copy-Knoten ab. (ii) Verbleibende Kanal-Konflikte werden als benachbarte Inversionen erkannt und durch lokal eingefügte Swap-Knoten aufgelöst. Beide Phasen wahren FIFO-Semantik, die ursprüngliche Node-Order und die PU-Allokation der vorhandenen Knoten.

Inhaltsverzeichnis

1 Einleitung

Buffered Exposed Datapath (BED) Architekturen wie SCAD verlagern Aufgaben, die klassische Prozessoren zur Laufzeit erledigen, in die Codegenerierung: Der Compiler steuert Datentransport, Scheduling und die Zuordnung von Operationen auf Verarbeitungseinheiten (Processing Units, PUs) mit FIFO-Puffern. Damit rücken Datenflüsse, ihre Reihenfolgen und Pufferzustände in den Fokus der Korrektheit und Effizienz des erzeugten Codes. In der Praxis werden sequentielle Programme zunächst in Datenflussgraphen (DFGs) übersetzt, deren Knoten Operationen und deren Kanten Transportbeziehungen modellieren. Aus diesen DFGs entsteht anschließend der Move Code für die Zielarchitektur. Falsche Werte an den Köpfen der FIFO-Puffer sowie Kantenkreuzungen führen dabei zu lokalen und strukturellen Konflikten, die ohne Gegenmaßnahmen die Ausführbarkeit behindern oder blockieren. Copy-Knoten (für FIFO-konformes Leveling) und Swap-Knoten (für das Auflösen kritischer Kreuzungen) sind etablierte Bausteine, bringen aber zusätzlichen Rechen- und Speicheraufwand mit sich. [SBR22a; SBR22b]

Diese Thesis adressiert die Frage, wie sich die Zahl zusätzlich benötigter Copy- und Swap-Knoten unter festen Rahmenbedingungen minimieren lässt: Die vorgegebene Knotenausführungsreihenfolge (Node-Order) sowie eine feste PU-Allokation bleiben für alle ursprünglichen Knoten erhalten. Neue Knoten dürfen nur an markierten Konfliktpositionen eingefügt werden und werden stets auf derjenigen PU platziert, auf der der Konflikt sichtbar wird. Damit bleiben sowohl Reihenfolge als auch Allokation invariant und sind für die Optimierung als Nebenbedingungen zu respektieren.

Der vorgeschlagene Ansatz arbeitet zweistufig. Zunächst simuliert die Heuristik die Node-Order entlang mit einer dynamischen Vorausschau und geplanter Rückgabe. Hierbei wird die minimale Anzahl der benötigten Copy-Knoten ermittelt. In der zweiten Phase werden verbliebene critical crossings kanalweise als benachbarte Inversionen erkannt und durch gezielte Swaps auf derselben PU aufgelöst. Beide Phasen wahren eine feste Reihenfolge, feste Allokation und korrekte FIFO-Semantik. .

2 Hintergrund

2.1 Buffered Exposed Datapath Architekturen

Eine Buffered Exposed Datapath Architektur ist eine spezielle Prozessorarchitektur, bei der die internen Datenpfade und Verarbeitungseinheiten (Processing Units, PUs) sichtbar gemacht werden. Das heißt, dem Compiler werden sowohl die Art der PUs als auch ihre Anzahl und der Datenspeicher verfügbar gemacht. Bei Buffered Exposed Datapath Architekturen (BED Architekturen) steuert der Compiler nicht nur den Datentransport, sondern kümmert sich auch statisch um das Scheduling der Instruktionen. Hier kümmert sich der Compiler um die Kommunikation zwischen den PUs und um die Allokation jeder Instruktion für jede PU.
Das Scheduling wird mit FIFO Puffer umgesetzt. Davon kann es pro PU mehrere geben. Mit dieser Art des Schedulings und des Zwischenspeicherns der Daten wird bei BED Architekturen auf globale Register verzichtet. [SBR22a]
Bei dieser Architektur werden die Instruktionen im Program Memory gespeichert und die Daten, die abgerufen werden, im Data Memory. Der Zugriff auf die Instruktionen erfolgt über eine Control Unit (CU) und der Zugriff auf die Daten über eine Load/Store Unit (LSU). Die PUs erhalten Werte, führen Berechnungen mit diesen Werten aus und geben die Ergebnisse aus. Instruktionen geben an, welche Berechnung eine PU durchführt und woher die Werte kommen. Die Kommunikation zwischen PUs, CU, LSU und den Puffern findet über Interconnection Networks statt. Wie eine BED Architektur im Allgemeinen aussieht, ist dargestellt in Abbildung 2.1.

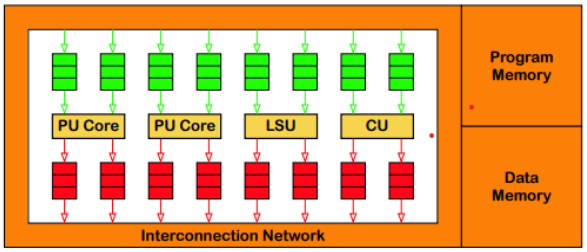

Abbildung 2.1: *Allgemeines Template einer BED Architekur [SBR22a]*

2.2 SCAD

Die SCAD(Synchrounous Control Asynchronous Dataflow) Architektur ist eine BED Architektur, die zur Ausführung von Programmen den asynchronen Datenfluss und die synchrone Steuerung kombiniert. [Bha21; Ker23]

SCAD ist so aufgebaut, dass es mehrere PUs gibt, die jeweils 2 Input Puffer, 2 Output Puffer und einen Input Puffer für den opcode haben. Der opcode ist eine Anweisung, die festlegt, welche Operation die PU ausführen soll. Die Puffer sind mit den Interconnection Networks verbunden. Zum einen gibt es den Move Instruction Bus (MIB), der die Werte synchron von der Control Unit zu den PUs schickt, und zum anderen gibt es das Data Transport Network (DTN), das Werte zwischen den PUs, der LSU und der CU hin und her sendet, sollten diese Werte verfügbar sein.

2.2.1 FIFO Puffer

Jeweils 2 Input/Output Puffer einer PU speichern Paare von Einträgen in der Form (adr, val). Für einen Input Puffer ist adr die Adresse des Output Puffers der PU, die den Wert val produziert hat oder noch produzieren wird. Für Output Puffer ist adr die Adresse des Input Puffers der PU, die den Wert val konsumieren wird.

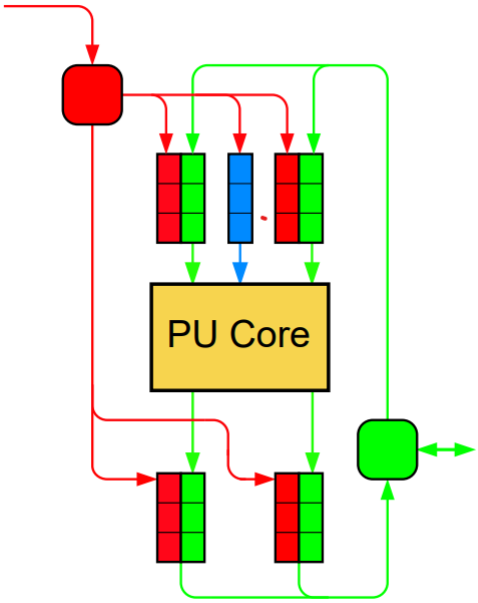

Abbildung 2.2: *Struktur einer PU mit dazugehörigen FIFO Puffern [Ker23]*

Die Struktur der Input/Output Puffer und einer PU sind in Abbildung 2.2 dargestellt. Hier sieht man den PU Core, der in Gelb dargestellt ist, und die Input Puffer oberhalb der PU sowie die Output Puffer unterhalb der PU. Diese sind jeweils grün-rot dargestellt. Die Pfeile dazwischen und das rote und grüne Quadrat stellen die Interconnection Networks mit ihren Controllern dar. Das rote Quadrat ist der MIB Controller und das grüne Quadrat ist der DTN Controller. Auf diese wird im weiteren Verlauf der Thesis noch näher eingegangen. Falls ein Wert val von einem Input Puffer erwartet wird, aber nicht ankommt, weil er von dem Output Puffer noch nicht produziert oder noch nicht gesendet wurde, wird im Input Puffer ein Eintrag gespeichert der Form $(adr, \perp)$. Das

Symbol $\perp$ ist hierbei ein Platzhalter für *val*. Der Eintrag $(adr, \perp)$ wird ebenfalls im Output Puffer gespeichert, wenn der Wert *val* erst noch von der PU produziert werden muss. *adr* hat jeweils einen unterschiedlichen Wert für die Puffer. Im Output Puffer wird als *adr* das *tgt* gespeichert und im Input Puffer die *src*.
Der dritte Input Puffer einer PU speichert den opcode *op*. Dieser enthält die Operation, die eine PU bei einer Berechnung verwendet. Er ist in Blau dargestellt in der Abbildung 2.2.

2.2.2 Move Code

SCAD Programme bestehen aus einer Sequenz von Move Instructions. Diese haben drei Formen:

- $src \rightarrow tgt$: Hierbei ist *src* der Output Puffer einer PU und *tgt* der Input Puffer einer PU. Das heißt, eine Move Instruction sagt aus, dass der Wert *val* vom Output Puffer *src* zum Input Puffer *tgt* geschickt wird.
- $imm \rightarrow tgt$: Hierbei ist *imm* ein immediate value und *tgt* der Input Puffer einer PU. Das heißt, ein immediate value wird zum Input Puffer *tgt* geschickt.
- $op \rightarrow tgt$: Hierbei ist *op* der opcode und *tgt* der Input Puffer einer PU. Das bedeutet, der opcode wird zum Input Puffer *tgt* geschickt.

2.2.3 Interconnection Networks, Control Unit und Load/Store Unit

Bei Ausführung eines Programmes, also des entsprechenden Move Codes, wird anhand des Program Counters immer die nächste Move Instruction von der CU genommen und an die PUs über den MIB gesendet. In Abbildung 2.2 ist der MIB Controller dargestellt als rotes Quadrat und der MIB sind die roten Pfeile. Handelt es sich um eine Move Instruction der Form $src \rightarrow tgt$ speichert der Input Puffer mit der Adresse *tgt* den Eintrag $(src, \perp)$ am Ende des Puffers. Der Output Puffer *src* speichert den Eintrag $(tgt, \perp)$ ebenfalls am Ende des Puffers. Bei einer Move Instruction der Form $op \rightarrow tgt$ speichert der Input Puffer *tgt* den opcode *op* im op Input Puffer. Sollte entweder der Output oder Input Puffer bereits voll sein, wird keiner der Einträge, also auch nicht der für den entsprechenden anderen Puffer, gespeichert. Die CU enthält dann ein Feedback und wartet mit der Ausführung und verschickt es erneut mit dem nächsten Schritt des Programms.
Hier fällt auf, dass die Adressen *adr* strikt synchron nach Move Code und Program Counter entsprechend versendet und gespeichert werden. Die eigentlichen Daten jedoch werden asynchron versendet und zueinander geschickt, und zwar erst dann, wenn sie bereitstehen.
Diese Daten werden über das DTN versendet. Der Transport der Daten zwischen PUs wird mit Nachrichten realisiert, die von den Output Puffern gesendet werden. Diese Nachrichten haben die Form (src, tgt, val) und bestehen aus

der Adresse des Output Puffers *src* von dem aus der Wert geschickt wird, der Adresse des Input Puffers zu dem der Wert gesendet wird *tgt* und dem Wert *val* selbst. Die Nachricht (src, tgt, val) wird vom Output Puffer erstellt, sollte der Eintrag am Kopf des Puffers von der Form (tgt, val) sein. Die Nachricht wird entsprechend an den Input Puffer *tgt* weiter geleitet und der Eintrag $(src, \perp)$, der am nächsten zum Kopf des Puffers liegt, wird ersetzt durch den Eintrag (src, val). Die Input Puffer beobachten entsprechend auch die Interconnection Networks. Die Reihenfolge, in der die Einträge im Input Puffer bereits bereitstehen, ist komplett abhängig vom Move Code und Program Counter. Das heißt, dass Werte, die von verschiedenen Output Puffer an einen Input Puffer gesendet werden, entsprechend in der Reihenfolge angeordnet werden, die vorgegeben ist, durch die bisherigen Einträge im Puffer (sprich dem Move Code und der program order).
Die LSU liest oder schreibt Werte von oder auf den Data Memory. Diese Werte werden dann über das DTN gesendet. In Abbildung 2.2 ist der DTN Cntroller dargestellt als grünes Quadrat und das DTN als grüne Pfeile. Es gibt Move Instructions der Form $imm \rightarrow tgt$. Das bedeutet, dass immediate values zu einem Input Puffer *tgt* gesendet werden. Der speichert am Ende des Puffers den Eintrag $(\top, imm)$, wobei $\top$ als Platzhalter dient. [Bha21]

2.2.4 Verarbeitungseinheit(PU)

Sollte eine Verarbeitungseinheit einen Eintrag der Form (adr, x) mit $x \neq \perp$ am Kopf eines ihrer Input Puffer finden, dann wird dieser Eintrag konsumiert und resultierende Werte y werden produziert. Dabei ist $y = f(x_1...x_m)$, wenn es Einträge mit $x_1...x_m$ gibt. Bei Werten $y_1...y_m$ wird jeder Wert im Eintrag des Output Puffers i der Form $(tgt, \perp)$ gespeichert, und zwar in diejenigen Einträge, die am nächsten am Kopf liegen. Dabei ersetzt der Wert den Platzhalter $\perp$ und wird in dem Eintrag gespeichert.

2.2.5 Datenflussgraphen

Die Codegenerierung eines strukturierten sequentiellen Programmes zum letztendlichen Move Code hat als Zwischenschritt die Erstellung eines Datenflussgraphen (DFG). Dieser ist ein Data Process Networks (DPN) mit einer beschränkten Menge an Knoten und Kanten. Die Knoten vom DFG sind die Prozessknoten und die Kanten stellen das Senden und Empfangen von Output Puffern zu Input Puffern dar. Ein strukturiertes Programm heißt in diesem Fall ein Programm ohne "*goto*" oder "*break*" Anweisung. Das strukturierte sequentielle Programm wird also zunächst in ein DFG übersetzt und dann werden die Knoten dieses DFGs auf die PUs des Prozessors gemappt. Die Edges des DFGs werden gemappt auf die Input und Output Puffer des Prozessors. Je nachdem, welcher Output Puffer Werte zu welchem Input Puffer sendet ändert sich die Form und die Farbe der Pfeile im DGF. Das führe ich später aus. [Sch21; SBR22b]

Grundsätzlich ist ein DPN ein gerichteter Graph, wobei die Knoten Prozessknoten sind und die Kanten das Senden und Empfangen der Werte zwischen

syntax	semantics
token control	
$(y) := \mathsf{C}(x)$	$t = \mathsf{get}(x)$ $\mathsf{push}(t, y)$
$(y_0, y_1) := \mathsf{D}(x)$	$t = \mathsf{get}(x)$ $\mathsf{push}(t, y_0)$ $\mathsf{push}(t, y_1)$
$(y_0, y_1) := \mathsf{S}(x_0, x_1)$	$t_0 = \mathsf{get}(x_0)$ $t_1 = \mathsf{get}(x_1)$ $\mathsf{push}(t_1, y_0)$ $\mathsf{push}(t_0, y_1)$
$(y) := \mathsf{J}(x_0, x_1)$	$t_0 = \mathsf{get}(x_0)$ $t_1 = \mathsf{get}(x_1)$ $\mathsf{push}(t_0, y)$
$() := \mathsf{K}(x)$	$t = \mathsf{get}(x)$
control flow	
$(y) = \mathsf{SEL}(c, x_1, x_0)$	$t_c = \mathsf{get}(c)$ $t_d = \mathsf{if}(t_c)\ \mathsf{then}\ \mathsf{get}(x_1)\ \mathsf{else}\ \mathsf{get}(x_0)$ $\mathsf{push}(t_d, y)$
$(y_1, y_0) = \mathsf{SWT}(c, x)$	$t_c = \mathsf{get}(c)$ $t_x = \mathsf{get}(x)$ $\mathsf{if}(t_c)\ \mathsf{then}\ \mathsf{push}(t_x, y_1)\ \mathsf{else}\ \mathsf{push}(t_x, y_0)$
memory access	
$(tk_{out}, y) = \mathsf{LD}_a(adr, tk_{in})$	$t_a = \mathsf{get}(adr)$ $t_m = \mathsf{get}(tk_{in})$ $\mathsf{push}(\mathsf{mem}[a, t_a], y)$ $\mathsf{push}(t_m, tk_{out})$
$(tk_{out}) = \mathsf{ST}_a(adr, tk_{in}, x)$	$t_a = \mathsf{get}(adr)$ $t_m = \mathsf{get}(tk_{in})$ $t_x = \mathsf{get}(x)$ $\mathsf{mem}[a, t_a] := t_x$ $\mathsf{push}(t_m, tk_{out})$
data operations	
$(y) = \mathsf{Const}(c)$	$\mathsf{push}(c, y)$
$(y) = \mathsf{MonOp}(f, x)$	$t_x = \mathsf{get}(x)$ $\mathsf{push}(f(t_x), y)$
$(y) = \mathsf{BinOp}(\odot, x_0, x_1)$	$t_0 = \mathsf{get}(x_0)$ $t_1 = \mathsf{get}(x_1)$ $\mathsf{push}(t_0 \odot t_1, y)$
$(y) = \mathsf{ITE}(c, x_1, x_0)$	$t_c = \mathsf{get}(c)$ $t_1 = \mathsf{get}(x_1)$ $t_0 = \mathsf{get}(x_0)$ $\mathsf{if}(t_c)\ \mathsf{then}\ \mathsf{push}(t_1, y)\ \mathsf{else}\ \mathsf{push}(t_0, y)$

Abbildung 2.3: *Syntax und Semantik der Prozessknoten von DPNs [Sch21]*

PUs darstellen [SBR22b]. In Abbildung 2.3 sieht man Syntax und Semantik aller Prozessknoten der DFGs. Die Lese und Schreiboperationen funktionieren folgendermaßen:

- $get(x)$ konsumiert den Kopf des Input Puffers für einen nicht leeren Puffer $x = [x_0, ..., x_{n-1}]$. Der Kopfwert des Puffers x_0 wird ausgegeben und vom Puffer entfernt.
- $push(x, y)$ fügt den Wert x am Ende des Puffers y hinzu.

So lassen sich die Prozessknoten erklären. In Abbildung 2.3 zeigt sich die Funktionsweise von Copy (C), Duplicate (D), Swap (S), Join (J), Kill (K), Select (SEL), Switch (SWT), Load Memoryα(LD_α), Store Memoryα(ST_α) und c (Const). Der Index α bezeichnet hierbei eine bestimmte Speicherinstanz.
Wie man in Abbildung 2.4 sieht, gibt es eine Graphendarstellung in der verschiedene Operationen der Prozessknoten und verschiedene Arten der Knoten

und Kanten zeigen, wie Puffer und Prozessknoten miteinander interagieren und verbunden sind. Ein schwarzer Pfeil steht für den linken Input, ein blauer gestrichelter Pfeil steht für den rechten Input, eine gefüllte Pfeilspitze steht für einen linken Output und eine leere Pfeilspitze steht für einen rechten Output. Der Token Input durch LD und ST wird in Orange dargestellt.

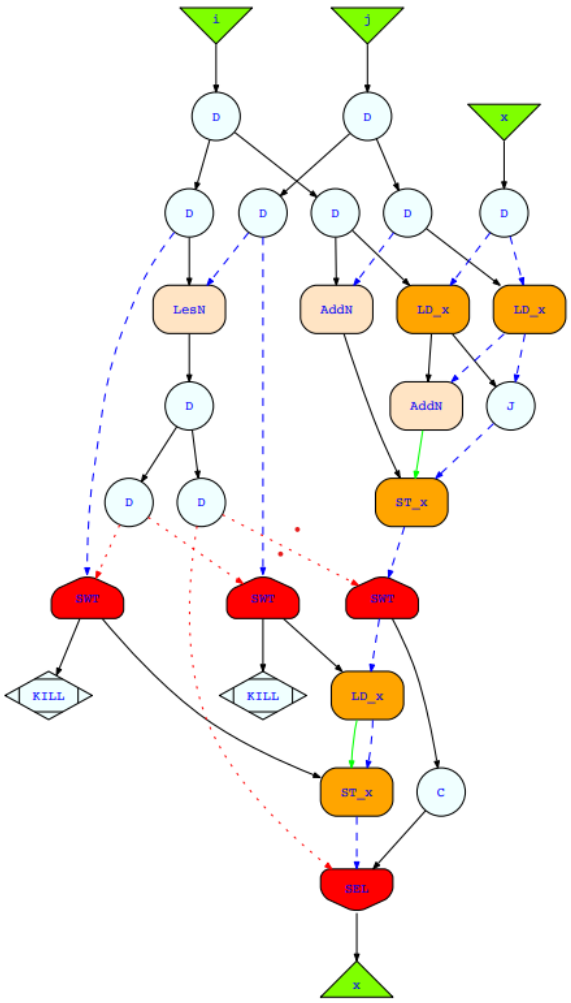

Abbildung 2.4: *Beispiel eines Datenflussgraphen [Sch21]*

2.3 Mögliche Konflikte und Kreuzungen im Datenverkehr

Im nächsten Schritt werden die Prozessknoten auf PUs und die Puffer des DFGs auf die Puffer der PUs gemappt. Wie DFGs gemappt werden, wird in [SBR22b] genauer beschrieben, lässt sich aber auch aus Abbildung 2.5 herauslesen.

Hierbei ist *opc* der opcode, also die Operation die von der PU ausgeführt

node	inL	inR	opc	outL	outR
$(y) := \mathsf{C}(x)$	x	–	C	y	–
$(y_0, y_1) := \mathsf{D}(x)$	x	–	D	y_0	y_1
$(y_0, y_1) := \mathsf{S}(x_0, x_1)$	x_0	x_1	S	y_0	y_1
$(y) := \mathsf{J}(x_0, x_1)$	x_0	x_1	J	y	–
$() := \mathsf{K}(x)$	x	–	K	–	–
$(tk_{out}, y) = \mathsf{LD}_a(adr, tk_{in})$	adr	–	LD	y	–
$(tk_{out}) = \mathsf{ST}_a(adr, tk_{in}, x)$	adr	x	ST	–	–
$(y) = \mathsf{Const}(c)$	–	–	CS(c)	y	–
$(y) = \mathsf{MonOp}(f, x)$	x	–	f	y	–
$(y) = \mathsf{BinOp}(\odot, x_0, x_1)$	x_0	x_1	$\odot$	y	–

Abbildung 2.5: *Code Generation Mapping von Datenflussgraphen zu Move Code [SBR22b]*

wird. inL, inR sind der linke und rechte Input Puffer einer PU und $outL, outR$ sind der linke und rechte Output Puffer einer PU.
Bei einem DFG können sich Kanten kreuzen. Das heißt, die Kreuzungen entstehen bei der Übertragung von Output Puffern zu Input Puffern. Wenn diese Kreuzungen kritisch sind, sorgt das dafür, dass der Move Code auf dem Prozessor nicht ausführbar ist, es sei denn, man verwendet Swap-Knoten. Grundsätzlich können vier Fälle auftreten bei 2x2 PUs wie in Abbildung 2.6 zu sehen ist [SBR22b; SB23]. Wir gehen von 2 Output Puffer src_1, src_2 und 2 Input Puffer tgt_1, tgt_2 bei einem gelevelten DFG mit den Instruktionen $src_1 \rightarrow tgt_1, src_2 \rightarrow tgt_2$ aus:

- Sollte $src_1 \neq src_2$ und $tgt_1 \neq tgt_2$ sein, dann entsteht keine Kreuzung und jede Reihenfolge von $src_1 \rightarrow tgt_1$ und $src_2 \rightarrow tgt_2$ im Move Code ist richtig.
- Sollte $src_1 \neq src_2$ und $tgt_1 = tgt_2$ sein, dann muss eingehalten werden, dass x_1 vor x_2 konsumiert wird. Die Reihenfolge, in der x_1 und x_2 produziert werden, ist egal.
- Sollte $src_1 = src_2$ und $tgt_1 \neq tgt_2$ sein, dann muss eingehalten werden, dass x_1 vor x_2 produziert wird. Die Reihenfolge, in der x_1 und x_2 konsumiert werden, ist egal.
- Sollte $src_1 = src_2$ und $tgt_1 = tgt_2$ sein, dann kann eine kritische Kreuzung, ohne die Anwendung von Swap-Knoten, entstehen. Wie in Abbildung 2.6 gezeigt, entsteht die Kreuzung, wenn x_1 vor x_2 produziert wird, aber x_2 erst vor x_1 konsumiert werden sollte.

2.3.1 Leveln des Datenflussgraphen

Um spätere Wartezeiten und Blockierungen des Move Codes zu verhindern, levelt man den DFG. Diese Wartezeiten und Blockierungen können entstehen, weil die FIFO Puffer mit dem FIFO (First In First Out) Prinzip arbeiten. Also

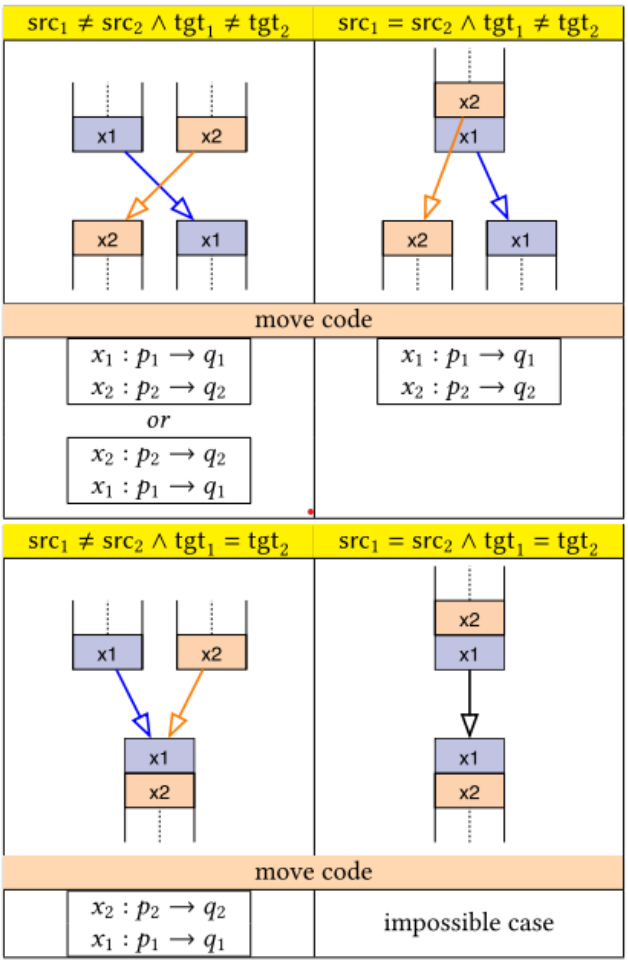

Abbildung 2.6: *Darstellung der möglichen Kreuzungen in einem DFG [SB23]*

wird bei dem Zugriff auf den Puffer erst der Kopf des Puffers weggenommen und sollte ein Wert hinzugefügt werden, wird er am Ende des Puffers hinzugefügt. Dementsprechend kann die Berechnung der PUs nur durchgeführt werden, wenn die für die Berechnung benötigten Werte am Kopf ihrer Puffer gespeichert sind. Sollte das nicht der Fall sein, kann man mit Copy-Knoten diese Werte an den Kopf des Puffers bringen. Das funktioniert so, dass ein token von der PU konsumiert wird und ein token mit demselben Wert von dieser PU produziert wird. Ein gelevelter Datenflussgraph ist ein Datenflussgraph, dem eine Levelstruktur $\ell : V \rightarrow N$ zugeordnet ist, so dass für jede Kante $(u, v) \in E$ gilt $\ell(u) < \ell(v)$ und zwischen zwei abhängigen Knoten keine Levels übersprungen werden. Copy-Knoten werden genutzt, um ungelevelte DFGs zu leveln. [SBR22b]

2.3.2 Planarisierung des Datenflussgraphen

Wie schon in Kapitel 2.3 erwähnt gibt es in einem gelevelten DFG kritische Kreuzungen, die auftreten können. Auch Erwähnung fand, dass diese durch Swap-Knoten aufgelöst werden können. Bezogen auf Abbildung 2.6 kann man die Werte mit Swap-Knoten vertauschen bis x_1 nach x_2 produziert und auch erst danach konsumiert wird. Mit Hilfe dieser Swap-Knoten lassen sich die kritischen Kreuzungen eines DFGs auflösen und er wird planar.

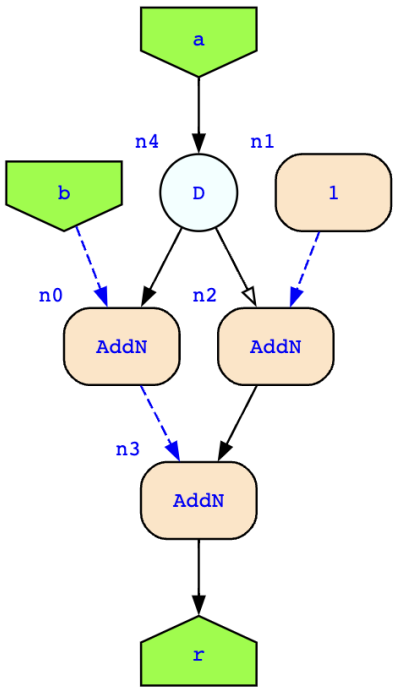

Abbildung 2.7: *Datenflussgraph eines MiniC Programm das eine kritische Kreuzung enthält, bei Ausführung auf einer PU*

In Abbildung 2.7 ist ein Graph mit einer solchen kritischen Kreuzung dargestellt, unter der Annahme, dass der sequenzielle Code auf einer PU des SCAD-Prozessors ausgeführt wird. Zusätzlich entsteht die kritische Kreuzung nur unter einer bestimmten Reihenfolge, in der die Knoten ausgeführt werden, der sogenannten Node-Order. Der sequenzielle Code, auf dem der Graph basiert, ist in MiniC geschrieben und wird in dem Code 2.1 dargestellt.

Mit diesem Code entsteht die kritische Kreuzung unter der Voraussetzung, dass er auf einer PU ausgeführt wird. Die Node-Order ist: n4, n1, n2, n0, n3. In Abbildung 2.7sind zu den Knoten die jeweiligen Benennungen zu sehen. Diese stehen immer links oben zu den Knoten. n4 ist zum Beispiel der Dup Knoten D. Als Erstes wird n4 ausgeführt und a wird dupliziert, womit 2 tokens entstehen. Die tokens bf0 und bf3 haben denselben Wert wie a. bf0 befindet sich jetzt im linken Output Puffer der PU. Durch die Ausführung des Knotens n2 wird bf3 mit dem token das durch den Knoten n1 entsteht addiert und ergibt das

Code 2.1 MiniC Programm das eine Kreuzung verursacht, vorausgesetzt, das Programm läuft auf einer PU

```
nat a, b;
nat r;
thread crossingSwap {
    nat x1, x2;
    x1 = a + b;
    x2 = a + 1;
    r = x2 + x1;
}
```

token bf5. bf5 ist jetzt ebenfalls im linken Output Puffer. Als nächstes müsste n0 ausgeführt werden. Hierfür wird bf0 im linken Input Puffer der PU benötigt und b im rechten. Das geht aber nicht weil bf5 nach wie vor im Output Puffer ist und aufgrund des FIFO Prinzips zuerst zum Linken Input Puffer bewegt werden müsste. Das ist eine kritische Kreuzung. In der Abbildung 2.8 ist eine Lösung dieser kritischen Kreuzung mit Swap-Knoten dargestellt.
Hier wird ein Swap-Knoten nach dem Knoten n4 eingefügt und dementsprechend die tokens bf3 und bf0 vertauscht wodurch die Kreuzung verhindert wird. bf0 wird nämlich jetzt zuerst gebraucht und mit dem Knoten n0 verarbeitet und danach entsteht mit der Addition bf3 und Const(1) der token bf5 der dann ohne das bf0 erwartet wird und für die letztendliche Addition durch den Knoten n3 hergenommen werden kann.

2.3.3 Notwendigkeit der Copy und Swap-Knoten

Gelevelte Graphen durch Copy-Knoten haben den Vorteil des FIFO konformes Verhalten ohne Risiko von Deadlocks. Der Nachteil ist jedoch, dass mehr Rechenaufwand und Speicheraufwand vorhanden sind. Bei der Berechnung durch die PUs wird mehr Rechenleistung benötigt für Operationen, die eigentlich irrelevant für die Berechnung der Werte sind. Durch das Weglassen ausgewählter Copy-Knoten ist der DFG zwar nicht mehr gelevelt, aber Ressourcen und Speicher können eingespart werden. Man muss nur darauf achten, dass man nur Copy-Knoten entfernt, die keine Deadlocks auflösen.
Ähnlich ist es bei den Swap-Knoten. Auf manche dieser kann verzichtet werden, was dieselben Vorteile bringt. Planarität heißt in diesem Fall, dass die kritischen Kreuzungen durch Swap-Knoten verhindert werden [SBR22b]. Das heißt, auf Swap-Knoten zu verzichten, kommt immer mit dem Nachteil daher, dass man bestimmte Constraints einhalten muss oder zusätzliche PUs benötigt, was wiederum auch mehr Ressourcen verbraucht.
Constraints bedeutet in diesem Fall, dass man das Auftreten kritischer Kreuzungen gar nicht erst erlaubt. Das hat den entscheidenden Nachteil, dass viele PUs notwendig sein können. Also können hier auch viele PUs und Ressourcen notwendig sein, nur um Crossings zu umgehen. Wie und wie viel Copy und Swap-Knoten man einspart, ist ein Abwägen der genannten Vor- und Nachtei-

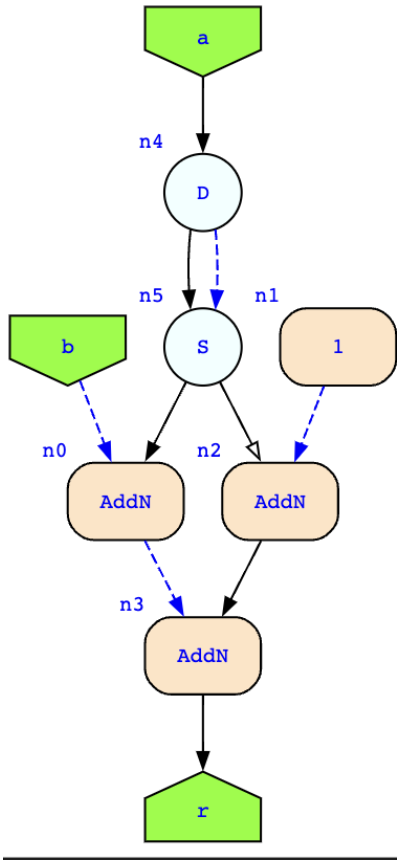

Abbildung 2.8: *Datenflussgraph eines MiniC Programm das eine kritische Kreuzung, die durch einen Swap-Knoten gelöst wurde, enthält, bei Ausführung auf einer PU*

le.
Auf jeden Fall benötigt es, um aus einem DFG generisch einen level-planaren DFG zu machen, viel Speicher und Ressourcen. Das kann mit Weglassen besagter Knoten optimiert werden.

3 Heuristik zur Minimierung von Copy- und Swap-Knoten

3.1 Voraussetzungen und Rahmenbedingungen

3.1.1 Ausgangslage

Es gibt eine feste Node-Order π und eine feste PU-Allokation μ, die vom Algorithmus vorausgesetzt werden. Der Algorithmus erwartet als Eingabe einen DPN und liefert als Ausgabe einen DPN mit minimaler Knotenzahl sowie die entsprechend angepasste Node-Order und die angepasste PU-Allokation.

3.1.2 Node-Order

Die Ausführungsreihenfolge der Knoten wird festgelegt und bleibt während und nach Anwendung des Algorithmus unverändert, abgesehen von neu eingefügten Knoten. Das wird realisiert, indem der Algorithmus die Positionen markiert, an denen ein Copy- oder Swap-Knoten erforderlich ist, und diese Knoten später genau an diesen Stellen in die Node-Order einfügt.

3.1.3 PU-Allokation

Jedem existierenden Knoten wird eine PU zugeordnet. Wie viele PUs es gibt und welcher Knoten welcher PU zugeordnet wird, hängt von der PU-Allokation ab. Formal wird eine PU-Allokation als Abbildung $\mu : V \to \{0, \ldots, k-1\}$ betrachtet, die jedem DPN-Knoten $v \in V$ eine PU zuweist, wobei k die verfügbare PU-Kapazität ist.
Auch die PU-Allokation bleibt, abgesehen von neu eingefügten Knoten, unverändert. Die Allokation der eingefügten Knoten erfolgt stets so, dass die Tokens, die zuvor vom betroffenen Knoten konsumiert wurden, auch vom neu eingefügten Knoten konsumiert werden. Das heißt: Neue Knoten werden immer auf derselben PU eingefügt, auf der der Konflikt entstanden ist.

3.1.4 Zu minimierende Knotentypen

Copy-Knoten

Copy-Knoten haben folgendes Verhalten: $(y) := Copy(x)$. Beim Leveln des DPN entstehen Copy-Knoten (falls der DPN nicht bereits gelevelt ist). Diese sind aber nicht alle notwendig, um konfliktfrei zu bleiben. Der Algorithmus behält nur einen Teil der Copy-Knoten des gelevelten DPN und fügt sie in die Node-Order ein, um mit weniger Copy-Knoten dennoch konfliktfrei zu bleiben.

Swap-Knoten

Swap-Knoten haben folgendes Verhalten: $y_1, y_2 := Swap(x_1, x_2)$ und können notwendig sein, falls nach dem Einfügen minimaler Copy-Knoten weiterhin Konflikte bestehen. Dies betrifft critical crossings, wie in 2.3 beschrieben.

3.1.5 Allgemeiner Ansatz

Der Algorithmus funktioniert so, dass zunächst nach einer möglichst minimalen Anzahl an Copy-Knoten gesucht wird. Sollten dennoch k Konflikte vorhanden sein, wird weiterhin nach der kleinsten möglichen Anzahl an Copy-Knoten gesucht, bei gleichbleibendem k. Anschließend werden die verbleibenden critical crossings mit einer minimalen Anzahl an Swap-Knoten gelöst.

3.1.6 DPN-Modell und Pufferarchitektur

Ein DPN ist ein gerichteter Graph $G = (V, E)$. Jeder Knoten $v \in V$ besitzt eine Operation *op* mit Signatur $y_1, \ldots, y_m := op(x_1, \ldots, x_n)$. Kanten führen Werte von y_i zu x_j.
Jede PU hat zwei Eingabepuffer, die als zwei FIFO-Queues modelliert werden: L und R. Ein Knoten kann in einem Schritt nur zwei Kopfwerte h_1, h_2 konsumieren und nur dann, wenn sie den erwarteten Werten entsprechen: ls (links erwarteter Wert) und rs (rechts erwarteter Wert) des nächsten Knotens in π. Die Werte der Ausgabepuffer werden für den Algorithmus sofort über eine Konsumenten-Abbildung in die Ziel-Queue eingeordnet (Konsumenten-Abbildung: $Var \to (PU, Side)$, wobei $Side \in \{L, R\}$).
Jeder Wert lv (linker Ausgabewert) und rv (rechter Ausgabewert) wird unmittelbar in einen Eingabepuffer geschrieben. Die Ziel-PU und die Wahl des Puffers erfolgen wie folgt:

- **PU:** Der Wert v wird auf die PU des ersten Konsumenten von v verschoben.
- **Puffer:** Falls $v = ls$, wird v nach L verschoben; falls $v = rs$, nach R. Es kann vorkommen, dass zwei Werte nach L und R verschoben werden, dass kein Wert verschoben wird, oder, falls es nur einen Wert gibt, dieser je nach Übereinstimmung mit ls bzw. rs nach L oder R.

3.1.7 Konfliktarten

Seien v_1 und v_2 zwei Werte. Falls v_1 bzw. v_2 ungleich ls oder rs sind, entspricht der erwartete Wert nicht dem am Kopf des jeweiligen Eingabepuffers. Es entsteht ein Konflikt, der mit einem Copy-Knoten gelöst werden muss. Der Copy-Knoten erzeugt ein neues Token. Im Fall von diesem Algorithmus wird jedoch derselbe Wert an das Ende des Puffers angehängt, dieses Verhalten ist eine Rotation. Wird derselbe Wert zweimal rotiert, wären zwei Copy-Knoten notwendig. In diesem Fall würde der zweite Copy-Knoten den durch den ersten entstandenen Token kopieren. Da jedoch kein neuer Token durch dieses Verfahren entsteht, wird die zweite Rotation desselben Werts registriert und als

zweiter Copy-Knoten gewertet, der den ersten Copy-Knoten des ursprünglichen Werts kopiert. Das wird so analog weitergeführt sollte ein Wert mehrfach kopiert werden müssen. Für jeden blockierenden Wert v wird geprüft, ob im gelevelten DPN ein Copy-Knoten existiert, der v kopiert, d. h. ein Copy-Knoten $v_r := \text{Copy}(v)$.

3.2 Heuristische Vorgehensweise zur Minimierung von Copy- und Swap-Knoten

In diesem Abschnitt wird die Heuristik beschrieben. Sie arbeitet unter den in Abschnitt 3.1 umrissenen Rahmenbedingungen: Eine feste Node-Order π und eine feste PU-Allokation μ sind vorgegeben und bleiben für alle ursprünglichen Knoten erhalten, nur neu eingefügte Knoten (Copy, Swap) werden an genau markierten Positionen in π ergänzt. Neue Knoten werden stets so allokiert, dass sie auf derselben PU liegen wie der Konsument, dessen Erwartung sie erfüllen, d. h. ein eingefügter Knoten erscheint auf derjenigen PU, auf der der Konflikt sichtbar wird. Auf jeder PU modellieren wir zwei FIFO-Eingabepuffer L und R, und pro Schritt darf der nächste Knoten in π höchstens zwei Kopfwerte konsumieren, die exakt den links bzw. rechts erwarteten Werten entsprechen. Produzierte Werte werden sofort gemäß einer Konsumenten-Abbildung in die Zielpuffer einsortiert.

Ziel und Grundidee: Ziel ist es, bei fixer Ausführungsreihenfolge und fixer PU-Zuordnung die Anzahl zusätzlich benötigter Knoten zu minimieren. Dazu zerlegt die Heuristik die Aufgabe in zwei Phasen:

- (i) Zunächst werden nur Copy-Knoten betrachtet. Wir simulieren die Pufferdynamik, protokollieren dabei, wann eine Rotation stattfindet, und leiten aus diesen Rotationen die kleinste notwendige Menge an Copy-Knoten ab.
- (ii) In einem zweiten, nachgelagerten Schritt werden verbleibende Konflikte auf demselben Kanal, die critical crossings, durch gezielte Swap-Knoten aufgelöst.

Pufferdynamik, Erwartung und Simulation

Ausgangspunkt ist die gegebene Node-Order π. Wir durchlaufen π in Produzent-Konsument-Schritten. Während jedem Schritt, werden die Ausgabewerte sofort dem Eingabepuffer derjenigen PU und Seite zugeführt, auf der der erste Konsument den Wert erwartet. Damit ist zu jedem Zeitpunkt die „nächste Erwartung“ des kommenden Konsumentenknotens eindeutig: (ls, rs) für die linke bzw. rechte Seite. Stimmen die Köpfe der beiden Puffer (h_L, h_R) mit (ls, rs) überein, kann konfliktfrei konsumiert werden.

Treffen Erwartung und Kopfwerte nicht zusammen, bietet der Algorithmus zwei Mechanismen:

1. **Rotation zum erwarteten Kopf:** Das erwartete Token wird innerhalb des entsprechenden Puffers nach vorne rotiert. Jede Rotation hängt das vordere Token an, an das Pufferende. Wichtig ist, dass dieses Ereignis protokolliert wird: Für jeden rotierten Wert v wird vermerkt, dass v an dieser Stelle an das Pufferende angehängt wurde. Im Endergebnis wird jede dieser beobachteten Rotationen als Copy-Bedarf für den Wert v gezählt, vorausgesetzt, dass v einem Input eines vorhandenen Copy-Knotens des gelevelten DPN entspricht.

2. **Vorausschauende Verschiebung (rechts nach links) mit Rückgabe:** Da der Einsatz von Copy-Knoten dazu führen kann, dass Werte nicht im rechten, sondern im linken Puffer erwartet werden, verwendet der Algorithmus einen kleinen Lookahead K. Wenn auf derselben PU innerhalb der nächsten K Konsumentenschritte klar ist, dass ein aktuell rechts anstehender Wert dort als rechter Kopf benötigt werden wird, jedoch durch davorliegende rechte Tokens blockiert wäre, wird dieses blockierende rechte Token vorzeitig nach links umgehängt (ans Ende von L) und als „später wieder zurückzugeben“ markiert. Genau in dem Schritt, in dem es rechts tatsächlich benötigt wird, wird das Token von L am Kopf genommen und zurück nach R gelegt (die „Rückgabe“). Diese Rückgabe ist deterministisch geplant.

Beide Maßnahmen sind vorläufige Puffer-Operationen in der Simulation: Dabei werden noch keine realen Copy- oder Swap-Knoten in den DFG eingefügt. Stattdessen sammeln wir ausschließlich Evidenz dafür, wo Rotationen auftreten. In beiden Fällen sind Copy-Knoten erforderlich, die jeweils den zu rotierenden oder zu verschiebenden Wert kopieren. Dabei werden ausschließlich Copy-Knoten verwendet, die im gelevelten DPN vorhanden sind.

Ableitung der minimalen Copy-Menge

Nach dem vollständigen Simulationslauf über π liegt eine Menge von Rotationsereignissen vor, die wir auf Inputnamen von Copy-Knoten abbilden. Intuitiv: Wenn in der Simulation ein Wert v auf einer PU mehrfach rotiert werden musste, um die linke oder rechte Erwartung zu erfüllen, dann wäre im realen System eine entsprechende Anzahl an Kopien von v nötig gewesen, um diese Rotationen ohne Änderung der Node-Order zu ermöglichen. Daraus konstruieren wir eine minimale Copy-Menge:

- Wir betrachten zunächst den gelevelten DPN mit den Copy-Knoten.
- Von diesen Copy-Knoten behalten wir nur diejenigen, deren Inputvariable in der protokollierten Rotationsmenge vorkommt.
- Alle anderen Copy-Knoten werden entfernt, indem deren Output im DPN systematisch durch ihren Input ersetzt wird, ohne die beobachtete Semantik der Simulation zu verletzen.

Die Node-Order π bleibt dabei erhalten. Neu hinzukommende Copy-Knoten werden exakt an jene Positionen eingefügt, an denen die Rotation protokolliert wurde. Ihre PU-Allokation ist die des jeweiligen Konsumentenkonflikts (d.h. identisch zur PU des Knotens, der die Erwartung an dieser Stelle bildet).

Critical Crossings und Swaps

Auch wenn einige Rotationsbedarfe durch Kopien eliminiert sind, können auf demselben Kanal (gleiche PU und gleiche Seite) noch critical crossings verbleiben: Die global produzierte Reihenfolge $(\ldots, a, b, \ldots)$ auf diesem Kanal steht dann in Widerspruch zur vom Konsumenten erwarteten Reihenfolge $(\ldots, b, a, \ldots)$. Hier setzt Phase (ii) an: Wir erkennen diese Inversionen kanalweise, indem wir für jede (PU,Seite)-Kombination die produzierte Sequenz und die erwartete Sequenz projizieren und benachbarte invertierte Paare erkennen. Für jedes invertierte Nachbarpaar wird ein Swap-Knoten eingefügt, der die beiden Werte auf diesem Kanal vertauscht. Der Swap wird auf derselben PU platziert, damit keine zusätzliche Kanal- oder PU-Kommunikation erforderlich ist.

Wahrung der Invarianten

Drei Invarianten sind zentral:

1. **Feste Reihenfolge:** Die ursprüngliche Node-Order π bleibt für alle vorhandenen Knoten unverändert. Neue Knoten werden ausschließlich an protokollierten Konfliktpositionen eingefügt.

2. **Feste Allokation:** Die PU-Allokation μ bleibt für alle vorhandenen Knoten unverändert. Neue Knoten werden auf der PU eingefügt, auf der der jeweilige Konflikt sichtbar ist. Bei Kopien ist das die PU des Zielkonsumenten des kopierten Wertes, bei Swaps die PU des betroffenen Kanals.

3. **Lokale Korrektheit der Puffersemantik:** Vorausschauende Verschiebungen (rechts nach links) werden nur dann durchgeführt, wenn eine spätere Rückgabe (links nach rechts) exakt zum Bedarfszeitpunkt möglich ist. Dieser Schritt ist deterministisch geplant und ersetzt eine ansonsten fällige Rotation. Er verändert die semantische Reihenfolge des Konsums nicht.

Komplexität, Parameter und Grenzen

Komplexität: Pro Entscheidung ist die Laufzeit im Worst Case linear in der „Distanz bis zum Verbrauch“ des betrachteten Tokens (Anzahl der dazwischenliegenden Consumer auf derselben PU). Über die gesamte Simulation ergibt sich damit eine Laufzeit, die in der Praxis quasi-linear in der Knotenzahl bleibt, solange typische Distanzen klein sind. Es gibt keinen globalen, einstellbaren Lookahead-Parameter mehr. Die Vorausschau ist datengetrieben.

Grenzen: Der Algorithmus garantiert keine globale Optimalität über alle DPNs: Entscheidungen für einzelne Tokens können interagieren, und die spätere Einfügung von Swap-Knoten beeinflusst ggf. nachfolgende Konfliktlagen. Lokal ist die Strategie jedoch sparsam: Ein Token wird nur dann verschoben, wenn dies eine reale Rotation (und damit einen zu zählenden Kopierbedarf) sicher verhindert. Empirisch führt dies bei fester Reihenfolge π und fester Allokation μ zu kleinen Copy-Mengen. Verbleibende Inversionen werden durch Swaps aufgelöst.

4 Implementation

Dieses Kapitel beschreibt die konkrete Implementierung des Algorithmus zur Minimierung von Copy- und Swap-Knoten unter der Nebenbedingung einer festen Node-Order π (Node-Order) und einer festen PU-Allokation μ. Die Implementierung ist in F# auf .NET realisiert und nutzt das Framework Averest. Zunächst werden die verwendeten Datenstrukturen und Averest-Funktionen skizziert. Anschließend folgen die internen Repräsentationen, gefolgt von den Kernfunktionen: (i) Konsumentenzuordnung, (ii) Simulation mit Vorausschau und geplanter Rückgabe (L→R), (iii) Extraktion der wirklich nötigen Copy-Knoten und stabile Projektion in Node-Order und PU-Map, sowie (iv) Erkennung und Behebung verbleibender Konflikte durch Swap-Knoten. Abschließend werden Komplexität und Implementierungsentscheidungen diskutiert.

4.1 Verwendete Averest-Strukturen und -Funktionen

Die Implementierung verwendet folgende, von Averest bereitgestellte Strukturen und Funktionen:

`DataflowProcessNetwork` DPN/DFG-Repräsentation eines MiniC-Programms.

`MiniC2DPN` Übersetzt ein MiniC-Programm in ein DPN.

`LevelizeDPN` Erzeugt einen gelevelten DPN.

`AllocProcUnitsByRandomMap` Erzeugt eine PU-Map μ als Ausgangspunkt für die Simulation.

`PrintDPN`, `WriteDPN2Dotfile` Ausgabe bzw. Visualisierung von DPNs.

Der Algorithmus arbeitet auf den von Averest gelieferten Graphen und Maps.

4.2 Interne Repräsentation und Invarianten

Graph, Reihenfolge, Allokation

Als Eingabe erwartet die Implementierung

1. einen DPN $G = (V, E)$,
2. eine feste Node-Order π (Permutation der Knotenindizes),
3. eine feste PU-Allokation $\mu : V \to \{0, \ldots, k-1\}$.

Die beiden Invarianten „feste Reihenfolge“ und „feste Allokation“ werden für alle vorhandenen Knoten strikt gewahrt. Neue, von der Heuristik eingefügte Knoten Copy/Swap werden an protokollierten Positionen in π ergänzt und immer auf genau der PU eingefügt, auf der der zugehörige Konflikt auftritt. Bei Kopien ist dies die PU des ersten relevanten Consumers, bei Swaps die PU des betroffenen Kanals (L oder R).

Pufferarchitektur

Für jede PU verwaltet die Implementierung zwei FIFO-Eingangspuffer, L und R. Ein Knoten konsumiert im Schritt höchstens die zwei Kopfwerte (h_L, h_R), sofern sie den erwarteten Eingängen (ℓ, r) seiner Operation entsprechen. Neu produzierte Werte werden über eine Konsumentenzuordnung (siehe unten) sofort in den Zielpuffer (L/R) der richtigen PU eingeordnet.

Konsumentenzuordnung (Var→(PU, Side))

Für jede Variable v wird bestimmt, auf welcher PU und auf welcher Seite (L/R) sie zum ersten Mal konsumiert wird. Diese Map wird zur Laufzeit genutzt, um produzierte Tokens in die korrekte Zielqueue einzureihen. Sie ist entscheidend, damit neu eingefügte Copy-Knoten automatisch die PU und Side des Knotens erben, dessen Wert kopiert werden soll, wodurch Fehler vermieden werden.

4.3 Zentrale Bausteine der Implementierung

4.3.1 Erkennung von Copy-Knoten und Rewriting

Zur Erkennung von Kopien genügt eine syntaktische Prüfung der Knotensignatur $(y) := \mathrm{Copy}(x)$ (bzw. Operatorname enthält `Copy`). Für das gezielte Entfernen einzelner Copy-Knoten wird ein lokales Rewriting genutzt: alle Verwendungen von `y` auf der rechten Seite werden durch `x` ersetzt: Der Copy-Knoten selbst wird aus dem DPN entfernt. Das Rewriting ist so implementiert, dass Copy-Ketten iterativ von hinten sicher abgebaut werden.

4.3.2 Simulation mit dynamischer Vorausschau und geplanter Rückgabe

Der Kern der Heuristik ist ein deterministischer Simulationslauf entlang π, der Rotationen für einzufügende Kopien zählt. Es können vorausschauende Verschiebungen notwendig sein:

- **Vorausschau:** Für die rechte Queue R einer PU wird ab der aktuellen Position in π so weit auf derselben PU K Schritte vorausgescannt. Blockiert ein Wert einen anderen, wird der blockierende Wert nach links verschoben.
- **Geplante Verschiebung (R→L) nur im Blockadefall:** Wird vor dem (virtuellen) Verbrauch ein Blockierer b gefunden (d. h. der aktuelle

R-Kopf passt nicht zur nächsten rechten Erwartung auf derselben PU), wird sofort genau dieser Token b aus R nach L verschoben und eine exakte Rückgabe (L→R) zum Fälligkeitsknoten[1] deterministisch geplant.

- **Geplante Rückgabe (L→R) exakt zum Bedarf:** Wenn der Fälligkeitsknoten erreicht ist, rotiert die Implementierung die linke Queue L minimal bis b am Kopf steht, verschiebt b zurück nach R und matcht (ℓ, r) ohne zusätzliche Rotation auf R. Jede dabei nötige Rotation auf L wird als Copy-Bedarf gezählt (und vermerkt).
- **Rotationen als Messgröße:** Klassische FIFO-Rotationen (ohne Vorausschau) würden Copies „erzwingen". Die Strategie verschiebt Tokens nur dann vorzeitig, wenn eine echte Blockade droht. Wo immer möglich wird nicht rotiert. Wo Rückgabe (L→R) zwingend eine L-Rotation erfordert, wird diese gezählt, damit die später tatsächlich einzufügenden Copy-Knoten exakt an den richtigen Stellen landen.

Die Simulation führt zusätzlich ein EreignisLog (Wert, nach welchem Knoten eingefügt), um die spätere Projektion in die Node-Order zu steuern.

Korrektheit des Pufferverhaltens: Die geplanten Verschiebungen und Rückgaben verändern nicht die Konsumtionsreihenfolge. Ein Token wird nur proaktiv umgehängt, wenn seine spätere rechte Verwendung eindeutig ist und die Rückgabe exakt dort stattfindet. Damit ersetzen wir reine FIFO-Rotationen durch Verschiebungen, ohne die Reihenfolge der Verbrauchsereignisse zu verändern.

4.3.3 Extraktion und Einfügung der minimal nötigen Copy-Knoten

Nach der Simulation liegt die Menge $\mathcal{C}_{\min}$ der Tokens vor, die (durch Rotationen auf L oder durch geplante Rückgaben) tatsächlich eine Kopie erfordern. Ausgehend vom gelevelten DPN werden alle Copy-Knoten gelöscht, deren kopierte Werte nicht in $\mathcal{C}_{\min}$ liegt. Die verbleibenden Kopien werden in die ursprüngliche Node-Order π an genau den vermerkten Konfliktpositionen eingefügt. Dabei wird

1. die Projektionsfunktion verwendet, um die alte Reihenfolge robust auf den modifizierten Graphen zu übertragen (exakte Schlüssel *op/lhs/rhs* mit fallbacks),
2. die *PU-Allokation* der eingefügten Kopien auf die PU des ersten relevanten Consumers der kopierten Variable gesetzt (damit wird der zuvor simulierte Konsumort getroffen).

Diese Schritte stellen sicher, dass die Copy-Knoten weder die ursprüngliche Allokation vorhandener Knoten ändern noch auf einer falschen PU landen.

[1] Dem Knoten, an dem b später tatsächlich als rechter Eingang r gebraucht wird.

4.3.4 Erkennung und Behebung verbleibender Konflikte durch Swaps

Nach der Einfügung der minimalen Kopien können auf einzelnen Kanälen ($PU \times \{L, R\}$) noch kritische Inversionen verbleiben: ein Paar (a, b) erscheint in der Produktionssequenz benachbart als (a, b), wird aber in der Erwartungssequenz benachbart als (b, a) konsumiert. Die Implementierung entdeckt solche inversen Nachbarschaften kanalweise und fügt Swap-Knoten $y_1, y_2 := \text{Swap}(x_1, x_2)$ ein, wobei Konsumenten von a und b auf die neuen Ausgänge umgeschrieben werden ($a \mapsto y_2$, $b \mapsto y_1$). Jeder Swap sollte auf der PU des betroffenen Kanals eingefügt werden.
Hierbei traten in der Implementierung jedoch Probleme auf, wodurch nicht alle Swaps erkannt wurden und die Zählungen von Konflikten und Rotationen verfälscht waren. Vermutlich lag dies an der fehlenden bzw. unzureichenden Simulation der Swap-Knoten und der zugrunde liegenden Swap-Dynamik.

4.4 Wesentliche Funktionen

Im Folgenden werden die wichtigsten implementierten Funktionen nach Aufgabe geordnet beschrieben:

`BuildConsumerMap` Bestimmt für jede Variable v das Paar $(\text{PU}, \text{Seite})$ des *ersten relevanten Consumers*.

`SimulationWithRotation1 & SimulationWithRotation2` Simulation entlang π mit zwei FIFO-Queues (L/R) je PU. Zählt Konflikte, Rotationen und Crossings. Kern ist die dynamische Vorausschau auf R : Virtueller R-Head wird über die Kette kommender rechter Erwartungen derselben PU vorwärts konsumiert, bis entweder der aktuelle Head verbraucht ist (kein Eingriff nötig) oder ein Blockierer entdeckt wird. Im Blockadefall: vorzeitige Verschiebung des Blockierers R→L und geplante Rückgabe L→R zum Fälligkeitsknoten. Die Rückgabe dreht ggf. L minimal vor (zählt als Copy-Bedarf) und stellt danach den rechten Match ohne zusätzliche Rotation her. Alle bewegten Werte werden ereignisbasiert protokolliert (`(value, node_after)`).

`remove_copies_func` Erzeugt aus dem Ereignis- und Rotationsprotokoll die Menge $\mathcal{C}_{\text{min}}$ der tatsächlich benötigten Kopien (Tokens, die mindestens einmal für Korrektheit rotiert bzw. zurückgegeben werden mussten). Auf Basis eines gelevelten DPN werden alle anderen Copy-Knoten entfernt.

`transferPUMapPreserveBase_LHSFirst` & `reassignCopyPUsToValueOrigin` Überträgt die alte PU-Map auf den modifizierten DPN, ohne existierende Zuordnungen zu verändern. Für Copy-Knoten wird die PU explizit auf die des ersten relevanten Konsumentenknotens gesetzt (damit wird die Konsumstelle aus der Simulation respektiert).

`DetectCriticalCrossings` Baut kanalweise Produzenten- und Erwartungssequenzen auf und findet invertierte benachbarte Paare. Fügt pro Befund einen Swap auf der betroffenen PU ein und rewired die Konsumenten auf die durch den Swap erzeugten Ausgänge. Hier gab es bei der Implementierung Probleme.

4.5 Idee des Algorithmus

1. **Vorbereitung:** MiniC→DPN, optional Levelisierung; PU-Map μ initialisieren (Averest). Node-Order π ist gegeben.
2. **Konsumentenzuordnung:** `BuildConsumerMap`.
3. **Simulation:** `Simulation` mit dynamischer Vorausschau auf allen PUs. Zählen/Loggen von Rotationen und Rückgaben.
4. **Copy-Extraktion:** `ExtractMinimalCopies` und Entfernen überflüssiger Kopien aus dem gelevelten DPN; Einfügen der verbliebenen Kopien in π an den protokollierten Stellen; PU der Kopien = PU des ersten relevanten Consumers der kopierten Variable.
5. **Swap-Phase:** `DetectCriticalCrossings`

4.6 Probleme bei der Implementierung

Wie bereits erwähnt funktioniert die Erkennung und Anwendung der minimal erforderlichen Swap-Knoten derzeit nicht zuverlässig. Dadurch lässt sich in den Experimenten wie auch im Algorithmus nur bestimmen, ob Swap-Knoten grundsätzlich erforderlich sind, nicht jedoch, wie viele davon minimal benötigt werden. Entsprechend berichten die Experimente lediglich die Notwendigkeit von Swap-Knoten, nicht deren Anzahl.

5 Experimente

5.1 Zielsetzung

Dieser Abschnitt untersucht, wie die implementierte Konflikterkennung und -behebung auf unterschiedlichen MiniC-Programmen wirkt. Dokumentiert werden ausschließlich strukturelle Effekte (benötigte Kopien, benötigte Swaps) unter Variation der PU-Anzahl und der Node-Order. Die Ergebnisse basieren auf dem Algorithmus.

5.2 Versuchsaufbau

Beispiele Fünf exemplarische MiniC-Programme mit unterschiedlicher Struktur und Größe: `BinaryTreeScl32`, `FastFourierTransform`, `HornerPoly8`, `min_copy_needed`, `first_example`. Die ersten beiden Beispielprogramme stammen aus dem Ordner UnrolledRAM der Averest-Website. Die drei weiteren Beispielprogramme wurden im Rahmen dieser Arbeit eigens erstellt.

Erlaubte Korrekturen.

- **Rotation** nur, wenn der blockierende Kopfwert durch einen Copy-Knoten im gelevelten DPN gedeckt ist und dessen einmaliges Budget noch nicht verbraucht wurde.

- **Swap**, wenn (a) keine Rotation möglich ist, dann werden die 2 Werte am Kopf des Puffers vertauscht und als Swap gezählt.

Variationen. PU-Anzahl (1–3), Knotenordnung (fix/spezifisch vs. "klug" gewählt).

Metriken. Anzahl benötigter Copy-Knoten und Anzahl benötigter Swaps in der Simulation.

5.3 Zusammenfassung der Ergebnisse

Die Tabelle 5.1 fasst die Resultate zusammen. Für Swaps bedeutet „Ja“, dass sie benötigt werden. # PU bezeichnet stets die Anzahl der PUs.

Programm	# PU	Copies	Swaps	Bemerkungen
`BinaryTreeScl32`	1	0	nein	Referenzfall ohne Maßnahmen.
`FastFourierTransform`	1	1	ja	Mit kluger Order ungelevelt auf 1 ausführbar.
`FastFourierTransform`	2	0	ja	Copies entfallen, Swaps weiterhin nötig.
`HornerPoly8`	1	12	ja	Kopien nehmen mit # PU ab (unten).
`HornerPoly8`	2	9	ja	
`HornerPoly8`	3	5	ja	
`min_copy_needed`	1	3	nein	Untere Schranke für Copy-Bedarf.
`min_copy_needed`	2	2	nein	
`min_copy_needed`	3	2	nein	
`first_example`	1	0	nein	

Tabelle 5.1: *Überblick: benötigte Kopien/Swaps je Benchmark und Konfiguration.*

Erkenntnisse

- Ohne Interferenz (z.,B. BinaryTreeScl32, first_example) sind weder Kopien noch Swaps nötig.
- FastFourierTransform: Auf 1 PU werden 1 Copy und Swaps benötigt. Auf 2 PUs entfallen Kopien, Swaps bleiben. Eine kluge Node-Order kann den 1 PU-Fall zusätzlich entschärfen.
- HornerPoly8: Mehr PUs reduzieren Copies stark (12→9→5).
- `min_copy_needed`: zeigt eine Konstellation mit mindestens 2 Kopien (bei ≥2 s) und 3 Kopien (bei 1 PU).

6 Fazit

Diese Arbeit hat eine zweistufige Heuristik vorgestellt, die unter fixierter Node-Order und PU-Allokation die Anzahl zusätzlich benötigter Knoten minimiert: (i) eine simulationsbasierte Ableitung einer minimalen Copy-Menge über Rotationsereignisse mit dynamischer Vorausschau und geplanter Rückgabe sowie (ii) eine anschließende, kanalweise Erkennung und Behebung verbleibender Paarinversionen mittels Swap-Knoten. Die Invarianten feste Reihenfolge, feste Allokation und FIFO-Korrektheit werden durchgängig gewahrt.

Es zeigt sich: Mehr PUs reduzieren vor allem den Copy-Bedarf (z. B. `HornerPoly8`: $12 \rightarrow 9 \rightarrow 5$ Kopien bei #PU $1 \rightarrow 2 \rightarrow 3$).

Grenzen und Ausblick: Die Heuristik ist bewusst lokal und garantiert keine globale Optimalität. Entscheidungen für einzelne Tokens können interagieren. Die Laufzeit bleibt in der Praxis quasi-linear in der Knotenzahl, da die Vorausschau datengetrieben und auf die Konsumentenfolge pro PU beschränkt ist. Künftige Arbeiten könnten (a) die gemeinsame Optimierung von Node-Order und PU-Allokation untersuchen, (b) kostenbewusste Varianten (Zeit/Energie/Belegung) integrieren, (c) formale Optimalitätsaussagen für Teilklassen von DPNs ableiten und (d) die Methodik auf breitere Benchmarks und komplexere Speicherhierarchien übertragen. Auch das systematische Abwägen zwischen Constraints zur Kreuzungsvermeidung und zusätzlicher Ressourcen (PUs, Puffer) verspricht neue Einsichten.

Abbildungsverzeichnis

Literatur

[Bha21] Anoop Bhagyanath. "Code Generation for Synchronous Control Asynchronous Dataflow Architectures". doctoralthesis. Technische Universität Kaiserslautern, 2021, S. XVII, 126. DOI: 10.26204/KLUEDO/6242. URL: https://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-62425.

[Ker23] Nadine Kercher. "Code Generation for Buffered Exposed Datapath Architectures". Examiner: Prof. Dr. Klaus Schneider, Dr.-Ing. Anoop Bhagyanath. Master's thesis. Kaiserslautern, Germany: Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, Department of Computer Science, Juli 2023.

[SB23] Klaus Schneider und Anoop Bhagyanath. "Consistency Constraints for Mapping Dataflow Graphs to Hybrid Dataflow/von Neumann Architectures". In: *ACM Trans. Embed. Comput. Syst.* 22.5 (Sep. 2023). ISSN: 1539-9087. DOI: 10.1145/3607869. URL: https://doi.org/10.1145/3607869.

[SBR22a] K. Schneider, A. Bhagyanath und J. Roob. "Virtual Buffers for Exposed Datapath Architectures". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Hrsg. von J. Brandt. Bd. 302. ITG-Fachbericht. Virtual Event: VDE, 2022, S. 45–55.

[SBR22b] Klaus Schneider, Anoop Bhagyanath und Julius Roob. "Code generation criteria for buffered exposed datapath architectures from dataflow graphs". In: *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, S. 133–145. ISBN: 9781450392662. DOI: 10.1145/3519941.3535076. URL: https://doi.org/10.1145/3519941.3535076.

[Sch21] Klaus Schneider. "Translating structured sequential programs to dataflow graphs". In: *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE '21. Virtual Event, China: Association for Computing Machinery, 2021, S. 66–77. ISBN: 9781450391276. DOI: 10.1145/3487212.3487343. URL: https://doi.org/10.1145/3487212.3487343.