Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

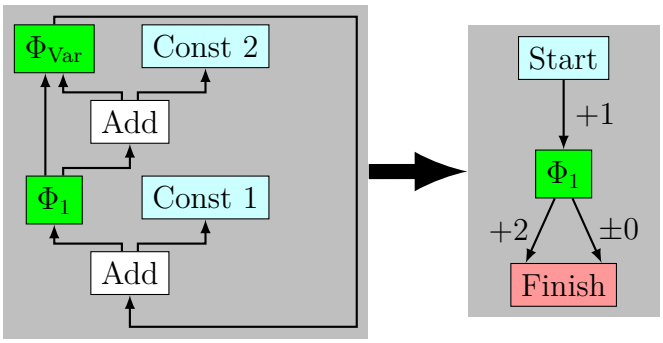
Lehrstuhl Prof. Dr.-Ing. Snelting

Compilerunterstützte automatisierte Seitenmigration auf MPSoCs

Bachelorarbeit von

Leon Bentrup

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M. Sc. Andreas Fried
Abgabedatum:	30. Juni 2020

Zusammenfassung

Auf MPSoCs kommen häufig nicht uniforme Speicherarchitekturen zum Einsatz. Es kann daher sinnvoll sein, vor gehäuften Zugriffen auf einen Speicherbereich, diesen in einen lokalen Cache zu laden. Wir implementieren eine Compileroptimierung und zugehörige Laufzeitbibliothek, die ohne Änderungen am Quellcode Speicherseiten in einen Software-Managed Cache lädt. Dazu werden in einer Zwischensprache Schleifen analysiert und Speicherbereiche identifiziert, auf die in ihnen zugegriffen wird. Wir evaluieren dieses Verfahren mit einigen einfachen Algorithmen. Dabei beobachten wir eine Laufzeitverkürzung um 25% beim Quadrieren einer 600×600 Matrix.

MPSoCs often have non-uniform memory access. Because of this, migrating memory pages into a local cache when they are accessed frequently is often benefitial. We implement a compiler optimization and accompanying runtime library that automatically migrates pages into a software managed cache. The optimization uses an intermediate representation of the program to analyze loops and identify memory regions that are accessed in them. We evaluate this system with simple algorithms. We observed a 25% shorter execution time when squaring a 600×600 matrix.

Inhaltsverzeichnis

1 Einführung

Moderne Mikroprozessorsysteme erhalten ihre Leistungssteigerung schon lange nicht mehr durch höhere Taktraten, sondern durch die erhöhte Zahl an Prozessorkernen. Um jedoch Resourcen auf Systemen mit potenziell tausenden von Prozessorkernen effizient zu verteilen sind andere Konzepte als klassiches Scheduling notwendig.

Ein solches Konzept ist das Invasive Rechnen. Resourcen werden nicht mehr zentral vergeben, sondern von den Programmen selbst angefordert und belegt, wenn diese benötigt werden. Das InvasIC-Projekt entwickelt Software und Hardware für dieses Konzept.

Ein weiteres Problem bei dieser großen Zahl von Prozessoren betrifft den Hauptspeicher: Ein gleichzeitiges Zugreifen von tausenden von Kernen auf Arbeitsspeicher und gleichzeitiges Beibehalten von Cache-Koherenz ist nicht praktikabel.

Das InvasIC-Projekt geht daher einen anderen Weg: Eine Zahl von Prozessorkernen (z. B. 5) wird zusammen mit etwas RAM zu einer Kachel zusammengefasst. Die Kachel agiert im Inneren wie ein normales System. Mehrere dieser Kacheln werden mit einem Network on Chip verbunden. Über Kachelgrenzen hinweg muss Speicher über das Netzwerk aufgerufen werden. Dies ist natürlich viel langsamer als ein Zugriff auf Speicher in der eigenen Kachel. Da der Großteil des Speichers als externe DDR-Module nur an wenige Kacheln des Systems angebunden ist, sind solche Speicherzugriffe auf andere Kacheln in vielen Programmen notwendig.

Um diesen Nachteil auszugleichen, bietet InvasIC einen Mechanismus, um einen Teil des Speichers der Kachel als Cache zu verwenden. Der Cache muss von der Anwendung selbst verwaltet werden. Dies erfordert Änderungen am Programmcode.

In dieser Arbeit wird eine Best-Effort-Optimierung entwickelt, die diesen Cache automatisch verwaltet. Bestehende Programme sollen mit unverändertem Quelltext den Cache verwenden und so schneller laufen.

2 Grundlagen und Verwandte Arbeiten

In diesem Kapitel werden Grundlagen erläutert, auf die in dieser Arbeit aufgebaut wird. Anschließend wird auf ähnliche Arbeiten zur Vorliegenden eingegangen.

2.1 FIRM

Moderne Compiler lassen sich in drei Teile gliedern: Das Frontend liest den Quelltext (z. B. C) und überführt ihn dabei in eine Repräsentation in einer Zwischensprache (Intermediate Representation). Das Middle-End führt auf dieser Repräsentation Analysen und Optimierungen durch. Das Backend erzeugt aus dieser optimierten Repräsentation Maschinenbefehle.

libFirm ist eine Programmbibliothek, die ein Middle- und Backend bereitstellt. [1] Als Intermediate Representation kommt dabei FIRM zum Einsatz. Programme werden dabei als Graphen repräsentiert. Der FIRM-Graph wird in Abschnitt 2.1.2 näher erläutert.

2.1.1 Static Single Assignment

Single Static Assignment (SSA) ist eine Eigenschaft von Zwischensprachen, die bedeutet, dass jede Variable nur ein einziges Mal zugewiesen wird und, dass jede Variable zugewiesen wird, bevor sie verwendet wird. Um eine Veränderung eines Wertes zur Programmlaufzeit darzustellen, muss er einer neuen Variable zugewiesen werden, die dann fortan benutzt wird. [2]

Nach Verzweigungen im Kontrollfluss hat eine Variable oft einen Wert, der davon abhängt, welchem Zweig gefolgt wurde. Dieser Umstand wird mit Φ-Funktionen abgebildet. Sie erhalten als Parameter mehrere Variablen und wählen für ihren Rückgabewert die zum gewählten Zweig passende Variable aus. In Abbildung 2.1

$a \leftarrow 1$
$a \leftarrow a + 2$
if $a < 1$ **then**
  $a \leftarrow 4$
else
  $a \leftarrow 8$
end if
$b \leftarrow a$

$a_0 \leftarrow 1$
$a_1 \leftarrow a_0 + 2$
if $a_1 < 1$ **then**
  $a_2 \leftarrow 4$
else
  $a_3 \leftarrow 8$
end if
$b_0 \leftarrow \Phi(a_2, a_3)$

Abbildung 2.1: Beispielprogramm mit seiner SSA-Form

wird die Φ-Funktion den Wert von a_3 annehmen, da zuvor dem Else-Zweig gefolgt wurde.

2.1.2 FIRM-Graph

In FIRM werden Programme in SSA-Form als Graphen repräsentiert. Verschiedene Operationen (Addition, Vergleich, Φ, ...) werden als Node im Graphen repräsentiert. Statt das Ergebnis einer Operation wie in Abbildung 2.1 in eine Variable zu speichern, die dann später als Operand benutzt wird, wird im FIRM-Graph eine Kante eingefügt. Sie zeigt von der Operation, die einen Wert als Operand erhält zur Operation, die diesen Wert erzeugt. Somit ist in einem FIRM-Graph keine strikte Ordnung von Operationen gegeben, sondern ein „Netz“ aus Abhängigkeiten. [1]

In Abbildung 2.2 verwendet die Operation `Add` die Werte der Operationen `Const 20` und `Const 22` als Operanden.

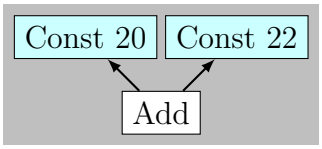


Abbildung 2.2: FIRM-Graph einer Addition von zwei Konstanten

Speicherzugriffe

Bei Speicher- und Ladeoperationen ist die Ausführungsreihenfolge allerdings relevant. Ein Verändern der Reihenfolge der Speicheroperationen beim Kompilieren kann dazu führen, dass falsche Daten geschrieben und gelesen werden. Um die Ordnung der Speicheroperationen im FIRM-Graph festzulegen verwendet FIRM *Memory-Kanten* (meist blau dargestellt). Jede Operation mit Speicherzugriff besitzt einen Eingang

für den Speicherzustand, in dem sie ausgeführt wird und einen Ausgang für den Speicherzustand nach ihrer Ausführung. So wird eine „Kette“ von Speicheroperationen gebildet. Jede Speicheroperation zeigt auf eine andere Speicheroperation, die vorher ausgeführt wird. Um Verzweigungen des Kontrollfluss abzubilden werden auch hier Φ-Nodes eingefügt. Sie wählen aus mehreren Speicherzuständen denjenigen aus, der zum gewählten Zweig passt.

Kontrollfluss

Der Kontrollfluss in einem FIRM-Graphen wird mit Blöcken abgebildet. Ein Block ist eine spezielle Node im Graphen. Jede andere Node besitzt genau eine zusätzliche Kante, die auf den zugehörigen Block zeigt. Diese Kante wird typischerweise nicht als solche dargestellt. Stattdessen werden lle Nodes, die zu einem Block gehören, in einem Rechteck gruppiert.

Jeder Block hat einen oder mehrere Vorgänger. Diese Vorgänger sind *Jmp-Nodes* oder *True-* bzw. *False-Nodes*, je nachdem, ob es sich um einen unbedingten oder bedingten Sprung handelt. Sprünge können also nur zu einem anderen Block führen, innerhalb eines Blocks gibt es keine Verzweigungen im Kontrollfluss.

Hat ein Block mehrere Vorgänger, kann eine Φ-Node in diesem Block wie eine Φ-Funktion je nach tatsächlichem Vorgänger zur Laufzeit einen Wert auswählen. Die Φ-Node besitzt so viele Eingänge wie der Block, in dem sie sich befindet. Der n-te Eingang der Φ-Node entspricht dabei dem n-ten Eingang des Blocks.

Schleifen

Eine Schleife in FIRM besteht aus einem oder mehreren Blöcken. Einer dieser Blöcke besitzt Kontrollflussvorgänger außerhalb der Schleife. Diesen Block nennen wir *Schleifenkopf*. Neben den Vorgängern außerhalb der Schleife besitzt der Schleifenkopf noch mindestens einen Vorgänger in der Schleife selbst (den *Rücksprung*). Außer trivialen Schleifen, die entweder keine Abbruchbedingung besitzen (Endlosschleife) oder Schleifen, die in jeder Iteration keine Operation ausführen, benötigt eine Schleife in FIRM mindestens zwei Blöcke: Einen Block, der die Operationen enthält, die in jeder Iteration ausgeführt werden, und einen Block, der die Schleifenbedingung prüft. Abbildung 2.3 zeigt eine typische Schleife als FIRM-Graph.

Nicht jede Schleife terminiert. Entweder sie hat gar keinen Nachfolger im Kontrollfluss oder er existiert, aber die Abbruchbedingung wird nie erfüllt. In der Regel können in einem FIRM-Graphen Nodes, auf die keine Kante zeigt (deren Ergebnis also

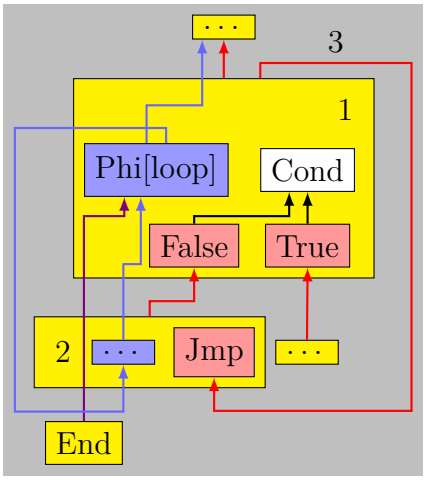


Abbildung 2.3: Vereinfachte Darstellung einer Schleife in FIRM (Der Inhalt der Blöcke würde für die Übersicht teilweise ausgelassen) mit Schleifenkopf **1**, Schleifenrumpf **2** und Rücksprungkante **3**

nie verwendet wird) problemlos aus dem Graphen entfernt werden. Dies ist bei Endlosschleifen aber nicht der Fall. Würde die Schleife entfernt werden, würde das Programm nun auf einmal doch terminieren. Damit hat sich das Laufzeitverhalten verändert. Aus diesem Grund wird in jede Schleife eine Φ_{loop}-Node eingefügt, auf die eine *Keepalive-Kante* vom End-Node zeigt. Das Φ_{loop} ist ebenfalls dafür verantwortlich, den Speicherzustand vor Betreten der Schleife und den Speicherzustand aus der letzten Schleifeniteration zu vereinen.

libFirm berechnet für jeden Graphen einen *Schleifenbaum*. Jedem Block ist ein `ir_loop`-Struct zugewiesen, das die Schleife identifiziert, zu der der Block gehört. Über das `ir_loop` ist es ebenfalls möglich die äußeren und inneren Schleifen zur Schleife zu finden. Jeder Graph besitzt ebenfalls ein `ir_loop`. Es stellt keine echte Schleife dar, ermöglicht aber das Abfragen aller Schleifen in einem Graphen. Die verschachtelten `ir_loop` bilden den Schleifenbaum des Graphen.

Confirm-Nodes

Mit *Confirm-Nodes* kann in einem Graphen angemerkt werden, wenn eine obere oder untere Grenze für einen Wert bekannt ist. Die Confirm-Node erhält als Eingabe einen Wert und einen *Bound*. Als Ausgabe wird der Eingabewert unverändert weitergegeben. Für alle Nodes, die auf die Confirm-Node zeigen, gilt die Grenze.

In Abbildung 2.4 sagt die Confirm-Node aus, dass der Operand von `Operation` (`Value`) kleiner als 42 ist.

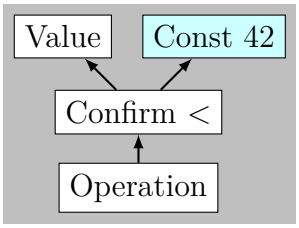


Abbildung 2.4: Confirm-Node in einem FIRM-Graphen

Eine Optimierung in libFirm kann automatisch solche Grenzen erkennen, und mit entsprechenden Confirm-Nodes im Graphen markieren. Dabei sind Confirm-Nodes nur als Anmerkung zu verstehen. Es findet in libFirm keine Prüfung statt, ob die durch das Confirm angegebene Bedingung auch tatsächlich stimmt.

2.2 NUMA-Architektur

In klassischen Rechnerarchitekturen mit mehreren Prozessoren ist der Arbeitsspeicher von jedem Prozessorkern aus „gleich“ zu erreichen. Das bedeutet, die Zugriffszeit auf einen Wert im Arbeitsspeicher ist von jedem Prozessorkern aus gleich. Umgekehrt kann von einem Prozessorkern aus auf jede Adresse im Arbeitsspeicher gleich schnell zugegriffen werden.

In vielen moderneren Architekturen ist das nicht mehr der Fall. Eine Rechnerarchitektur mit `Non-Uniform Memory Access` (NUMA) besitzt zwar für den gesamten Arbeitsspeicher einen globalen Adressraum, allerdings sind verschiedene Teile des Arbeitsspeichers an verschiedene Prozessoren angeschlossen. Man unterscheidet dann für jeden Prozessor lokalen Arbeitsspeicher mit vergleichsweise kurzen Zugriffszeiten und entfernten Arbeitsspeicher mit vergleichsweise langen Zugriffszeiten. [3]

2.3 Invasives Rechnen

Invasives Rechnen ist gekennzeichnet durch die Fähigkeit eines Programms, das auf einem Parallelprozessor ausgeführt wird, selbst Resourcen (wie Prozessorkerne, Speicher oder Kommunikationsschnittstellen) zeitweise zu reservieren, zu nutzen und anschließend wieder freizugeben. [4] Es gibt keine zentrale Verwaltung, die den laufenden Programmen Resourcen zuweist.

Invasive Programme belegen Resourcen in 3 Phasen. In der *invade*-Phase werden Resourcen (Prozessor-Tiles) angefragt. Das Programm erhält einen *Claim* auf die

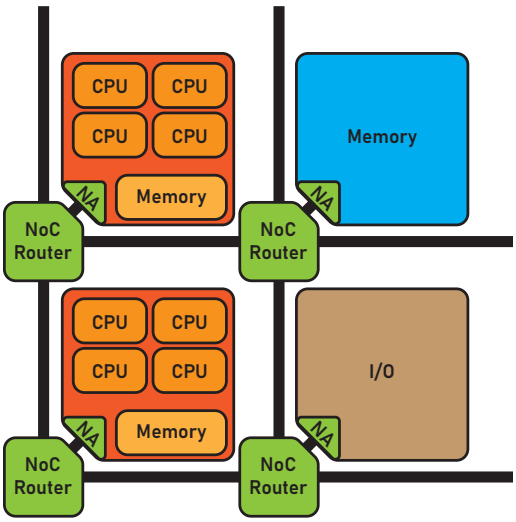


Abbildung 2.5: Schematische Darstellung der Invasic-Hardware [4]

zugeteilten Resourcen. In der *infect*-Phase wird die Ausführung auf den Resourcen gestartet. In der *retreat*-Phase wird die Ausführung beendet und die Resourcen wieder freigegeben.

2.3.1 OctoPOS

Das Betriebssystem *OctoPOS* und die Middleware *iRTSS* (invasive Run-Time Support System) bieten eine Platform für den praktischen Einsatz invasiven Rechnens. [5] OctoPOS ist ein Library Operating System, wird also mit der Anwendung (bzw. mehreren Anwendungen) zu einem Executable gelinkt.

Neben der x86-64-Architektur, die vor allem zum Testen mit QEMU verwendet wird, unterstützt OctoPOS eine eigens für invasives Rechnen entwickelte Hardware auf Basis von LEON3-Prozessoren.

2.3.2 InvasIC-Hardware

Die InvasIC-Hardware verwendet eine gekachelte Architektur. Eine einzelne Kachel ist dabei ähnlich aufgebaut wie ein klassisches Mehrprozessorsystem: Mehrere LEON3 Prozessorkerne und ein DRAM von 12 MiB sind mit einem AMBA-Bus verbunden. Ebenfalls an diesen Bus angeschlossen ist ein Netzwerkadapter (NA), der die Kachel über ein *Network on Chip* (NoC) mit den anderen Kacheln auf dem Chip verbindet. Abbildung 2.5 zeigt den logischen Aufbau der Hardware.

Manche der Kacheln erfüllen je nach Konfiguration spezielle Funktionen. Eine IO-Kachel enthält Hardware für die Kommunikation außerhalb des Chips, z.B. einen Ethernet-Adapter. Eine Memory-Kachel ist über ihren internen Bus mit DDR-Speicher verbunden.

2.3.3 Speicher in OctoPOS

Die verschiedenen Speicher des Systems sind wie folgt im Adressraum angeordnet:

DDR-Speicher der Memory-Tile
TLM der Kachel 0
TLM der Kachel 1
...
TLM der Kachel N
TLM der lokalen Kachel

Das bedeutet, dass jedes Element im TLM einer Kachel über zwei Adressen angesprochen werden kann, nämlich über eine globale Adresse im vorderen Teil, und über eine lokale Adresse im letzten Teil des Adressraums. Die globale Adresse zeigt immer auf den gleichen Speicher, während die lokale Adresse je nachdem, von welcher Kachel der Zugriff erfolgt, immer auf einen anderen Speicher (den der zugreifenden Kachel) zeigt.

Programme können auf Speicher im TLM anderer Kacheln oder den Speicher der Memory-Tile zugreifen. Zugriffe auf den Speicherbereich anderer Kacheln werden vom NA über das NoC an die richtige Kachel weitergeleitet. Das geschieht ohne Zutun der Anwendung, die auf den Speicher zugreift.

Damit ist die InvasIC-Hardware eine NUMA-Architektur (siehe Abschnitt 2.2). Die Zugriffszeit auf entfernten Speicher ist deutlich größer als für den TLM der eigenen Kachel: Ein Zugriff auf den TLM der lokalen Kachel benötigt 20 Prozessorzyklen. Ein Zugriff auf den Speicher der Memory-Tile benötigt je nach Auslastung des NoC etwa 90 Prozessorzyklen. [6]

Wegen der deutlich längeren Zugriffszeit auf den Speicher der Memory-Tile bietet OctoPOS einen *Pagecache*, einen reservierten Teil des TLM jeder Kachel, der als Cache verwendet werden kann. OctoPOS stellt einen Syscall bereit, mit dem Seiten des Speichers anderer Kacheln in den Pagecache kopiert werden können. Die Memory Map der lokalen MMU wird so angepasst, dass Speicherzugriffe nicht mehr an den

NA, sondern an den lokalen Pagecache gerichtet werden. Mit einem weiteren Syscall kann dieses Mapping wieder aufgehoben werden. Änderungen, die währenddessen im Pagecache erfolgt sind, werden dabei zurückgeschrieben.

2.4 Verwandte Arbeiten

Ähnlich zur hier behandelten Seitenmigration ist das Prefetching. Dabei wird mit einer CPU-Instruktion eine Cache-Zeile aus dem Arbeitsspeicher in den Cache des Prozessors geladen. Während des Arbeitsspeicherzugriffs kann die CPU bereits weitere Befehle ausführen. Moderne Compiler wie GCC oder LLVM verfügen über Optimierungen, diese Instruktionen automatisch in das Ausgabeprogramm einzufügen. [7] Das erfordert ebenfalls eine Analyse von Speicherzugriffen in Schleifen. Allerdings ist der CPU-Cache deutlich kleiner als der Pagecache in OctoPOS, und auch eine Cache-Zeile ist viel kleiner als eine Speicherseite. Deshalb unterscheidet sich die Art der Analyse sehr von dieser Arbeit.

Die vorliegende Arbeit ist stark von einer Arbeit von Piccoli und Santos [8] inspiriert. Dort wird eine Compileroptimierung entwickelt, die vor Schleifen automatisiert Anweisungen einfügt, die den Linux-Kernel anweisen, Seiten aus anderen NUMA-Domänen zu migrieren. Allerdings wird diese Optimierung nicht auf der Intermediate Representation, sondern anhand des Syntaxbaumes durchgeführt. Sie lässt sich somit nicht einfach auf andere Programmiersprachen übertragen.

Diener und Cruz [9] erweitern die vorhergehende Arbeit von Piccoli und Santos um eine Betriebssystemkomponente, die mehrere gleichzeitig ausgeführte Programme bei der Seitenmigration in Betracht zieht, um eine gleichmäßige Auslastung des Systems zu erreichen. Ebenfalls wird dort auch der Thread, der einen Speicherzugriff ausführt, bei Bedarf auf einen anderen Prozessor verschoben, um Zugriffszeiten zu verkürzen. Lezteres lässt sich nicht ohne Weiteres auf die dezentrale Resourcenverwaltung invasiven Rechnens übertragen.

In einer Arbeit von Seo und Lee [10] wird ein Software Managed Cache für Cell BE Prozessoren entwickelt. Statt einer Analyse beim Kompilieren werden die Programme mit einer speziellen OpenMP-Runtime ausgeführt, die die Verwaltung des Caches selbst übernimmt. Die Runtime versucht zur Laufzeit die Größe der Cache-Zeilen und die Ersetzungsstrategie zu optimieren. Vorhersagen über zukünftige Speicherzugriffe werden hier jedoch nicht getroffen.

3 Entwurf und Implementierung

Wir implementieren eine Compileroptimierung „Pagecache-Optimierung“, die beim Kompilieren Speicherzugriffe in Schleifen analysiert. Die Ergebnisse werden an eine Programmbilbiothek „LibVSM“ weitergeleitet, welche die VSM-Schnittstelle von OctoPOS aufruft, um Seiten in den TLM zu kopieren.

Im Nachfolgenden werden wir zunächst betrachten, wie die Compileroptimierung mit libFirm eine Schleife analysiert. Danach betrachten wir die Analyse der Speicherzugriffe. Zum Schluss erläutern wir, wie die Programmbibliothek entscheidet, ob eine Seite in den Cache geladen werden soll, und wie sie OctoPOS dazu anweist.

3.1 Compileroptimierung

Wir erwarten den größten Nutzen des Pagecache bei aufeinanderfolgenden Zugriffen auf zusammenhängenden Speicher. Dies ist regelmäßig bei Array-Zugriffen in Schleifen der Fall.

Deshalb versucht unsere Optimierung für jede Schreib-/Lese-Operation in einer Schleife eine Abschätzung für die untere und obere Grenze des Speicherbereichs zu finden, auf den während der Laufzeit durch die Operation zugegriffen wird.

Dazu ist es zunächst notwendig Grenzen für die Schleifenvariablen zu bestimmen, also die Variablen, die sich bei jeder Operation ändern. In FIRM bietet es sich dazu an, die entsprechenden Φ-Nodes, die den Wert der Variablen vor Betreten der Schleife und den Wert der vorigen Schleifeniteration zusammenführen, zu betrachten.

Danach setzt die Optimierung die gefundenen Grenzen in den Teilgraph ein, der die Adresse der Speicheroperation berechnet. So ergibt sich eine Abschätzung für die obere und untere Grenze des Speicherbereichs, auf den die Operation zugreift.

Zum Schluss werden für eine Speicheroperation zwei Funktionsaufrufe an unsere Programmbibliothek in den Graphen eingefügt, jeweils vor und nach der Schleife. Als Argumente erhalten sie die gefunden Grenzen des Speicherbereichs.

3.1.1 Vereinfachen des Kontrollflusses

Bevor die eigentliche Analyse der Schleifen beginnt, verändert die Optimierung zunächst den FIRM-Graphen so, dass jede Schleife nur einen Vorgängerblock, den *Preheader* besitzt. In diesen Preheader werden später die Funktionsaufrufe an die LibVSM eingefügt.

Um den Preheader zu erzeugen, fügt die Optimierung für jede Schleife einen neuen Block ein, der zunächst nur eine Jmp-Node enthält. Danach betrachtet die Optimierung die Kontrollflussvorgänger des Schleifenkopfes, und teilt diese ein in Vorgänger außerhalb der Schleife und Vorgänger innerhalb der Schleife (Rückwärtskanten).

Dann ersetzt die Optimierung das Sprungziel der Vorgänger außerhalb der Schleife durch den Preheader. Am Schleifenkopf entfernt sie diese Vorgänger und fügt die Jmp-Node im Preheader als neuen, einzigen Vorgänger außerhalb der Schleife ein. Die Rückwärtskanten bleiben unverändert.

Analog zum Kontrollfluss passt die Optimierung die Φ-Nodes im Schleifenkopf an. Für die Eingänge außerhalb der Schleife fügt sie im Preheader eine neue Φ-Node ein, die diese vor der Schleife zusammenführt. Abbildung 3.1 zeigt eine Schleife vor und nach Einfügen des Preheaders.

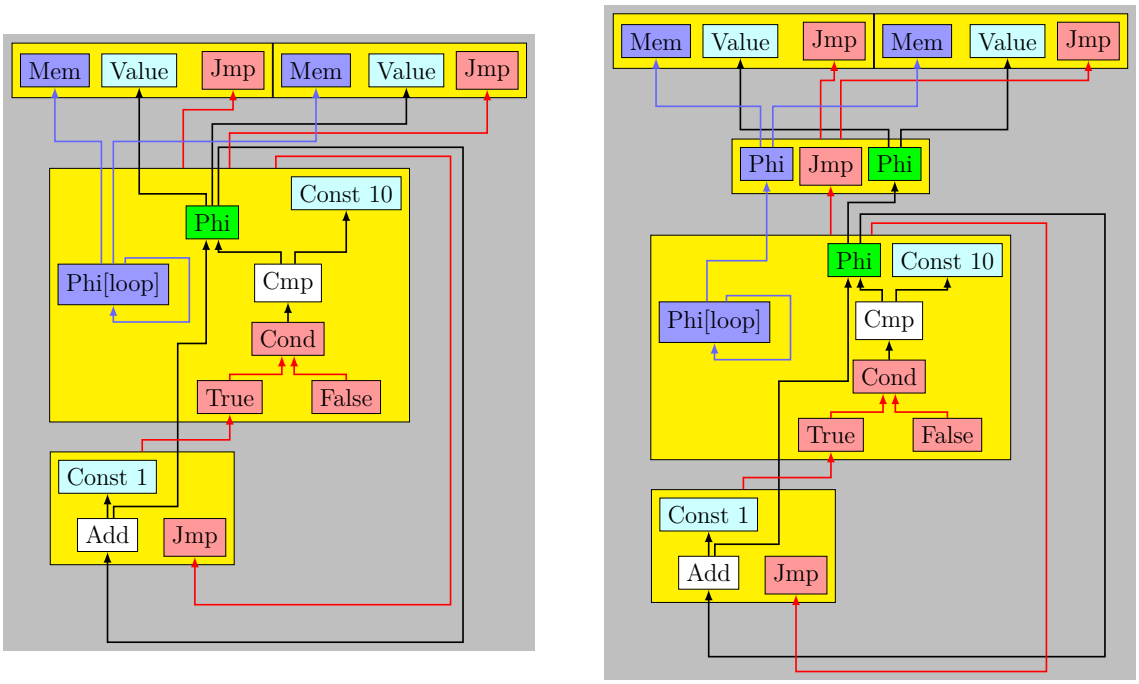


Abbildung 3.1: Ein Firm-Graph einer Schleife mit zwei Vorgängern.
Links die ursprüngliche Form, Rechts mit eingefügtem Preheader.

3.1.2 Analyse der Schleifenvariable

Unsere Optimierung versucht für jede Schleife alle Schleifenvariablen zu bestimmen. Dazu werden für jede Φ-Node im Schleifenkopf folgende Bedingungen geprüft: 1. Es existiert ein Pfad von der Φ-Node zu sich selbst. 2. Alle Nodes entlang des Pfades gehören zur Schleife oder zu einer inneren Schleife. Sind diese Bedingungen erfüllt, hängt der Wert der Variable von der vorhergehenden Iteration der Schleife ab und es handelt sich um eine Schleifenvariable. Die Optimierung prüft diese Bedingungen mit einer Tiefensuche ausgehend von der Φ-Node.

Als nächstes versucht die Optimierung, Grenzen für den Wert der Schleifenvariable zu finden. Dazu nehmen wir an, dass es sich um eine monotone Schleifenvariable handelt. Dann sind die Werte, die die Variable in der ersten und der letzten Iteration der Schleife annimmt, die gesuchten Grenzen. Wir nennen den Wert in der ersten Iteration *Init* und den Wert in der letzten Iteration *Limit*. Ist die Schleifenvariable nicht monoton, wird der Wertebereich eventuell zu klein eingeschätzt. Dann wird später möglicherweise nur ein Teil des Speicherbereichs auf den zugegriffen wird in den Pagecache geladen.

Der Init-Wert ist ausgehend vom Φ im Schleifenkopf trivial zu bestimmen: Da die Schleife immer über den Preheader betreten wird, und sich dort ein Φ befindet, welches alle (ehemaligen) Vorgänger der Schleife zusammenführt, ist der Wert dieses Φ-Nodes der Init-Wert.

Für den Limit-Wert wird auf die Confirm-Funktionalität von libFirm zurückgegriffen. Ausgehend von der Φ-Node der Variablen im Schleifenkopf wird ein Confirm-Node gesucht, der auf das Φ zeigt. Der Bound-Wert dieses Confirms ist der Limit-Wert.

Die Analyse der Schleifenvariablen kann in manchen Fällen fehlschlagen: Wenn die Optimierung keine Φ-Node im Schleifenkopf findet, für die auch eine Confirm-Node existiert, schlägt die Analyse für diese Schleife fehl. Dies ist insbesondere bei Endlosschleifen der Fall. Wenn die Analyse für eine Schleife fehlschlägt, werden die inneren Schleifen dennoch betrachtet. Siehe dazu Abschnitt 3.1.7.

3.1.3 Anzahl der Schleifeniterationen

Zusätzlich zu den Grenzen der Schleifenvariable wollen wir herausfinden, wie oft eine Schleife durchlaufen wird.

Dazu implementieren wir zunächst eine Heuristik die den Inkrement der Schleifenvariable in jeder Iteration abschätzt. Wir nennen dieses Inkrement den *Step*-Wert der

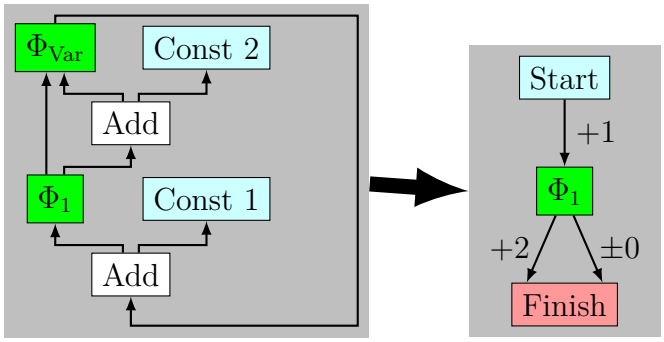


Abbildung 3.2: FIRM-Graph und resultierender Dijkstra-Graph

Schleifenvariable.

Die Optimierung erzeugt aus dem FIRM-Graphen einen vereinfachten Graphen mit Kantengewichten. Auf diesem Graphen wird wie in der Arbeit von Piccoli und Santos [8] der Dijkstra-Algorithmus ausgeführt.

Um den vereinfachten Graphen zu konstruieren beginnt die Optimierung an der Φ_{Var}-Node im Schleifenkopf, die zur Schleifenvariable gehört, mit einer Tiefensuche. Jede Φ-Node, die gefunden wird, wird als Node in den neuen Graphen eingefügt, und mit einer zuvor eingefügten Node verbunden, wenn im FIRM-Graph einen Pfad zwischen den entsprechenden Φ-Nodes existiert. Alle Additionen und Subtraktionen mit Konstanten entlang dieses Pfades werden aufsummiert und als Kantengewicht dieser Kante festgelegt. Die Tiefensuche endet, wenn sie wieder den Startpunkt oder eine Node außerhalb der Schleife erreicht. Dabei wird die Φ-Node, die zur Schleifenvariable gehört in eine Start- und Ziel-Node im Dijkstra-Graphen umgewandelt. In Abbildung 3.2 sehen wir links einen Ausschnitt des urpsrünglichen FIRM-Graphen. Rechts sehen wir den resultierenden Dijkstra-Graphen. Die Node Φ_{Var} wurde in Start und Finish umgewandelt, Φ_1 wurde übernommen und die Add-Nodes wurden durch Kantengewichte ersetzt.

Mit dem Dijkstra-Algorithmus bestimmt die Optimierung dann den kürzesten Pfad mit größtmöglichen Kantengewichten als obere Abschätzung für den Inkrement der Schleifenvariable. Der normale Dijkstra-Algorithmus berechnet den kürzesten Pfad mit geringsten Kantengewichten. Wir kehren die Relaxationsbedingung um, damit der Pfad mit größten Kantengewichten ausgewählt wird.

Ausgehend vom gefundenen Init-, Limit-, und Step-Wert berechnet die Optimierung eine Schätzung für die Zahl der Schleifendurchläufe:

$$n = \frac{\text{limit} - \text{init}}{\text{step}}$$

Für innere Schleifen wird dieser Wert mit dem Wert der äußeren Schleife multipliziert.

So lässt sich später für jeden Speicherzugriff schätzen, wie oft er ausgeführt wird, je nachdem in welcher Schleife er sich befindet.

3.1.4 Analyse der Speicherzugriffe

Wie in Abschnitt 2.1.2 beschrieben, besitzt jede Schleife im Schleifenkopf ein Φ_{loop}, welches in eine „Speicherkette“ eingebunden ist. Alle Speicheroperationen einer Schleife befinden sich an einem Pfad von diesem Φ_{loop} zu sich selbst.

Die Compileroptimierung benutzt diese Eigenschaft, um mittels Tiefensuche alle Speicheroperationen einer Schleife zu finden. Entlang dieses Pfades befinden sich ebenfalls alle Speicheroperationen aller inneren Schleifen der gegebenen Schleife.

Die Optimierung sammelt alle gefundenen Speicheroperationen in einer Liste. Dabei wird jeweils ein Pointer auf die `ir_node` des Speicherzugriffs selbst und der Adresse, sowie der Datentyp der Operation gespeichert.

```
  int a[10];
  int b[10];
  int i = 0;
loop:
  if (i > 9) goto end;

  a[i] = i;
  b[9-i] = a[i];

  i++;
  goto loop;
end:
```

Abbildung 3.3: Ein Programm in GOTO-Form, welches a die Zahlen von 0-9 aufsteigend, und b absteigend zuweist.

Betrachten wir das Programm in Abbildung 3.3: Die Optimierung findet die Speicherzugriffe `Store a[i]`, `Store b[9-i]` und `Load a[i]`.

3.1.5 Bestimmen des Speicherbereichs

Für die im vorherigen Schritt gefundenen Speicheroperationen schätzt die Optimerung den Speicherbereich, auf den die Operationen während der Ausführung der gesamten

Schleife zugreifen.

Dazu bestimmt die Optimierung basierend auf den Init- und Limit-Grenzen der Schleifenvariablen eine Init- und Limit-Grenze des Speicherbereichs. Die Optimierung dupliziert rekursiv alle Knoten, die die Adresse der Speicheroperation angeben und sich in der Schleife befinden, in den Preheader. Stößt sie dabei auf eine Φ-Node, welche zu einer Schleifenvariable gehört, wird es durch die gefundene Init- bzw. Limit-Grenze der Schleifenvariable ersetzt. Diese Operation wird zwei mal durchgeführt. Einmal wird durch die Init-Grenze und einmal durch die Limit-Grenze ersetzt.

In der Regel ist der Term, der die Adresse angibt, eine Kombination der vier Grundrechenarten $(+, -, \cdot, \div)$. Bei einer Kombination aus Addition und Multiplikation schätzt diese Methode den korrekten Speicherbereich, auch wenn mehrere Schleifenvariablen enthalten sind. Bei der Subtraktion und der Division ist dies jedoch im Allgemeinen nicht der Fall, denn ein höherer rechter Operand führt zu einem niedrigeren Ergebnis. Aus diesem Grund vertauscht die Optimierung die Init- und Limit-Grenze, wenn sie den rechten Operanden einer Subtraktion oder Division verfolgt.

Wir betrachten erneut das Beispielprogramm in Abbildung 3.3. Bei der Analyse der Schleifenvariable (Abschnitt 3.1.2) wurde bereits die Init-Grenze 0, und die Limit-Grenze 9 gefunden. Bei der Analyse der Speicherzugriffe (Abschnitt 3.1.4) wurden die Speicherzugriffe `Store a[i]`, `Store b[9-i]` und `Load a[i]` gefunden.

Für die Adresse `a[i]` wird i durch die Init- und Limit-Grenze ersetzt. Dadurch wird der Speicherbereich `a[0]-a[9]` gefunden.

Für die Adresse `b[9-i]` stößt die Optimierung auf eine Subtraktion. Im Rechten Operanden wird dann Init und Limit vertauscht. Als Init-Grenze berechnet die Optimierung `b[9-9]`, als Limit-Grenze `b[9-0]`. Insgesamt wird der Speicherbereich `b[0]-b[9]` gefunden.

Um die Notwendigkeit, bei Subtraktion und Division Init- und Limit-Wert zu vertauschen, zu verdeutlichen betrachten wir als weiteres Beispiel den BubbleSort-Algorithmus in Abbildung 3.4. Die Optimierung hat hier bereits die Schleifenvariablen `i` von `1` bis `9` und `j` von `0` bis `10-i` erkannt.

Für den Speicherzugriff auf `a[j]` ersetzt die Pagecache-Optimierung für die Limit-Grenze `j` durch `10-i`, und erhält `a[10-i]`. Als nächstes ersetzt die Optimierung `i` durch den Init-Wert `1`, da es auf der rechten Seite einer Subtraktion steht. Die Limit-Grenze ist also `a[9]`. Für die Init-Grenze erhalten wir durch ersetzen von `j` durch `0` dann `a[0]`.

Ohne die Vertauschung von Init- und Limit-Wert würden wir die falschen Grenzen

```
int a[10];
int i = 1;
while (i < 10) {
  int j = 0;
  while (j < 10 - i) {
    compareAndSwap(a[j], a[j+1]);
    j++;
  }
i++;
}
```

Abbildung 3.4: Der BubbleSort-Algorithmus

`a[0]` und `a[0]` erhalten: Beim Berechnen der Limit-Grenze ersetzt die Optimierung zunächst (richtig) `a[j]` zu `a[10-i]`. Danach ersetzt sie jedoch `a[10-i]` zu `a[10-10]`.

Bei einer Schleife, die „rückwärts“ läuft, ist die Init-Grenze größer als die Limit-Grenze. Dieser Fall wird von der LibVSM erkannt, und die Init- und Limit-Grenze werden vertauscht.

3.1.6 Einfügen der Funktionsaufrufe

Über jeden in dieser Liste vorhandenen Speicherzugriff wollen wir unsere Programmbibliothek zur Laufzeit vor Betreten der Schleife informieren. Die Programmbibliothek besitzt dazu zwei Funktionen: `vsm_prefetch` und `vsm_free`. Durch Aufrufen von `vsm_prefetch` informiert das Programm zur Laufzeit die Programmbibliothek darüber, dass der angegebene Speicherbereich in der folgenden Schleife wahrscheinlich benötigt wird. Ein Aufruf von `vsm_free` signalisiert, dass der angegebene Speicherbereich (für den zuvor bereits `vsm_prefetch` aufgerufen wurde) nicht mehr benötigt wird. Die Implementierung dieser Funktionen wird in Abschnitt 3.2 erläutert.

Unsere Optimierung fügt für jeden Speicherzugriff einen Aufruf an `vsm_prefetch` in den Preheader und einen Aufruf an `vsm_free` in jeden Nachfolger der Schleife ein.

Da Funktionsaufrufe potenziell den Speicherzustand verändern können, müssen sie in die Speicherkette des FIRM-Graphen eingefügt werden. Dazu „trennt“ die Optimierung die Kanten Φ_{loop} zum Φ im Preheader, und alle Kanten von außerhalb der Schleife zum Φ_{loop}, auf und fügt dazwischen den Funktionsaufruf ein.

Als Argumente werden dabei übergeben:

- Die Init-Grenze des Speicherbereichs
- Die Limit-Grenze des Speicherbereichs
- Die Größe des Datentyps der Speicheroperation (z.B. 4 Byte bei einem `int`-Array)
- Die geschätzte Zahl der Ausführungen der Speicheroperation

3.1.7 Verschachtelte Schleifen

Die Optimierung betrachtet den Schleifenbaum des Graphen und betrachtet jede äußere Schleife des Graphen für sich. Nachdem unsere Optimierung in dieser Schleife die Variablen analysiert hat, führt sie diesen Schritt für alle Kinder der Schleife erneut durch. Wenn dies abgeschlossen ist, analysiert die Optimierung die Speicherzugriffe. Dabei betrachtet sie auch die Speicherzugriffe in den inneren Schleifen.

Schlägt die Analyse der Schleifenvariable in einer Schleife fehl, fährt die Optimierung mit den inneren Schleifen dieser Schleife fort, so als wären sie eine äußere Schleife des Graphen.

Betrachten wir den Schleifenbaum in Abbildung 3.5 als Beispiel: Zunächst werden die Variablen analysiert. Die Optimierung beginnt mit Schleife 1. Sie war erfolgreich und fährt mit Schleife 3 fort. Damit ist die Analyse dieses Teilbaums abgeschlossen. Für die Speicheroperationen in Schleife 1 und 3 werden Aufrufe an die Programmbibliothek vor und nach Schleife 1 eingefügt.

Die Optimierung betrachtet dann Schleife 2. Dort schlägt die Analyse der Schleifenvariablen fehl. Sie betrachtet nun Schleife 4 und 5 als eigenständige Schleifen. Für die Speicheroperationen in Schleife 4 werden Funktionsaufrufe vor und nach Schleife 4 eingefügt, nicht vor und nach Schleife 2. Ebenso für Schleife 5.

3.2 Programmbibliothek

Zusätzlich zur Compileroptimierung implementieren wir eine Programmbibliothek *LibVSM*, die alle zur Programmlaufzeit benötigten Funktionen bereitstellt. Sie wird vom Programm durch die von der Optimierung eingefügten Befehle aufgerufen.

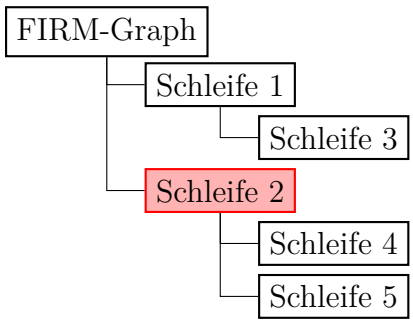


Abbildung 3.5: Ein Schleifenbaum eines FIRM-Graphen

LibVSM enthält die zwei Funktionen `vsm_prefetch` und `vsm_free`.

3.2.1 Die Funktion `vsm_prefetch`

Die Funktion `vsm_prefetch` entscheidet, ob ein von der Pagecache-Optimierung gefundener Speicherbereich in den Pagecache geladen werden soll.

Dazu prüft sie zunächst, ob sich der Speicherbereich im Speicher einer entfernten Kachel befindet. Befindet sich der Speicher nicht in einer entfernten Kachel, wird er nicht in den Pagecache geladen. In diesem Fall ist der Pagecache genau so schnell wie der direkte Aufruf, da er nicht über das Network on Chip erfolgt.

Als Nächstes prüft die Funktion, ob die Größe des Speicherbereichs größer ist als eine Zeile des Level 2-Caches. Falls nicht, wird der Speicherbereich ebenfalls nicht in den Pagecache geladen.

LibVSM hält im TLM der Kachel eine Liste der Speicherbereiche, die sich zur Zeit im Pagecache befinden. Überschneidet sich der angefragte Speicherbereich mit einem Speicherbereich, der sich bereits in der Liste befindet, wird er eventuell verkleinert, oder verworfen, falls sich der vollständige Bereich bereits im Pagecache befindet.

Abschließend wird der ggf. verkleinerte Bereich in einem `acquire`-Aufruf an das Betriebssystem weitergegeben, und ebenfalls in die Liste eingetragen.

Die Funktion gibt den Index des Speicherbereichs in der Liste zurück.

Die aktuelle Implementierung der LibVSM benutzt die Information, wie oft eine Speicheroperation ausgeführt wird (Abschnitt 3.1.3), noch nicht. Die Pagecache-Implementierung in OctoPOS ist noch nicht vollständig, und das Kopieren einer Seite in den Pagecache benötigt deutlich länger, als es zukünftig zu erwarten ist.

Daher lässt sich zur Zeit noch keine gute Heuristik entwickeln, die abschätzt, ob das Kopieren eines Speicherbereichs in den Pagecache schneller ist als der direkte Zugriff über das NoC. Siehe dazu auch Abschnitt 5.1.

3.2.2 Die Funktion `vsm_free`

`vsm_free` erhält als Parameter den von `vsm_prefetch` zurückgegebenen Index. Die Funktion entfernt den Speicherbereich aus der Liste und übergibt ihn in einem `release`-Aufruf an das Betriebssystem.

4 Evaluation

Im Folgenden wird die automatische Seitenmigration evaluiert. Als Vergleichswert wird dabei der CParser mit deaktivierter Pagecache-Optimierung verwendet.

Da es für die OctoPOS-Platform keine standardisierten Benchmark-Suites gibt und sich diese nicht ohne weiteres auf das invasive Ausführungsmodell portieren lassen, wählen wir selbst verschiedene Algorithmen auf Datenstrukturen aus und implementieren diese selbst.

Wir kompilieren das Library-Betriebssystem sowie die LibVSM mit GCC 8.2.0, unser Testprogramm mit CParser 1.22. Jedes Programm wird in zwei Versionen kompiliert: Eine Version mit eingeschalteter Pagecache-Optimierung im CParser, eine Version mit augeschalteter Pagecache-Optimierung. Für die Compileroptionen von GCC verwenden wir außer der Pagecache-Optimierung die Standardoptionen des Compilers.

Jedes so kompilierte Programm führen wir anschließend auf der Hardware-Platform des InvasIC-Projekts aus, die mit einem FPGA synthetisiert wird. [11] Die Taktfrequenz des Systems beträgt $f = 50\text{MHz}$. Das System besteht aus 4 Kacheln, als 2x2-Quadrat angeordnet. Der DDR-Speicher ist an Kachel 3 angeschlossen, die Algorithmen werden auf Kachel 0 ausgeführt. Für einen Zugriff auf den DDR-Speicher sind also 2 Hops über das NoC notwendig. Der Aufbau des Systems ist in Abbildung 4.1 dargestellt.

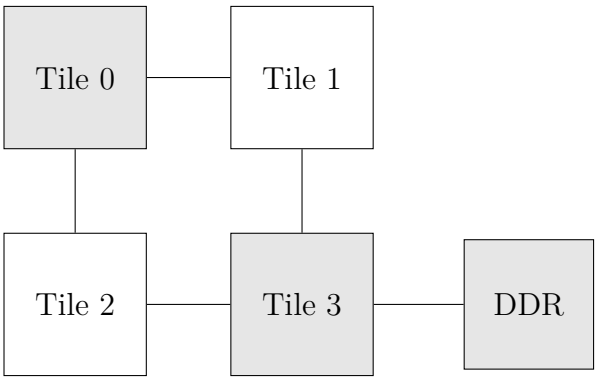


Abbildung 4.1: Logische Anordnung der Kacheln im Testsystem

Wir testen jeden Algorithmus mit verschiedenen Eingabegrößen N. Jeder Algorithmus wird mit einer Eingabegröße 10 mal ausgeführt. Dabei wird jeweils die Laufzeit als Wall-Clock-Time anhand der Taktzyklen bestimmt. Aus den 10 Testläufen berechnen wir das arithetische Mittel (Gleichung (4.1)), die empirische Standardabweichung (Gleichung (4.2)) und den Stichproben-Variationskoeffizient (Gleichung (4.3)):

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{4.1}$$

$$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2} \tag{4.2}$$

$$v = \frac{s}{\bar{x}} \tag{4.3}$$

Dabei bezeichnet n den Stichprobenumfang, in unserem Fall ist $n = 10$.

Um die Verbesserung oder Verschlechterung der Optimierung zu beurteilen, berechnen wir für jedes Programm und jede Eingabegröße die relative Änderung (Gleichung (4.4)) von deaktivierter zu aktivierter Pagecache-Optimierung.

$$r = \frac{\bar{x}_e - \bar{x}_d}{\bar{x}_d} \tag{4.4}$$

Dabei steht $\bar{x}_e$ für den Mittelwert der Laufzeit bei eingeschalteter Pagecache-Optimierung, $\bar{x}_d$ für den Mittelwert bei ausgeschalteter Optimierung.

Für die Evaluation wählen wir die folgenden Algorithmen aus:

- *DotProduct* - berechnet das Skalarprodukt zweier Vektoren der Größe N,
- *NSieve* - berechnet mit dem Sieb des Eratosthenes alle Primzahlen von 2 bis N,
- *BubbleSort* - sortiert einen umgekehrt sortierten Array der Größe N,
- *MatSquare* - berechnet das Produkt einer $N \times N$-Matrix mit sich selbst.

In Abbildung 4.2 ist der Aufwand der Implementierung der ausgewählten Algorithmen im O-Kalkül angegeben.

Zum Zeitpunkt dieser Arbeit ist die VSM-Funktionalität in OctoPOS noch nicht vollständig implementiert: Das Kopieren von Speicherseiten in den Tile-Local Memory erfolgt nicht durch die DMA-Einheit des Systems, sondern druch einen der CPU-Kerne. Das Kopieren durch die CPU nimmt deutlich mehr Zeit in Anspruch

Algorithmus	Aufwand	Speicherbedarf
DotProduct	$O(n)$	$O(n)$
NSieve	$O(n \log(\log(n)))$	$O(n)$
BubbleSort	$O(n^2)$	$O(n)$
MatSquare	$O(n^3)$	$O(n^2)$

Abbildung 4.2: Laufzeiteigenschaften der ausgewählten Algorithmen

als das Kopieren durch die DMA-Einheit. Deshalb dauert das Aufrufen der `acquire`- und `release`-Syscalls deutlich länger als mit der fertigen Implementierung zu erwarten ist. Aus diesem Grund messen wir zusätzlich die Laufzeit dieser Syscalls und geben zusätzlich zur gesamten Laufzeit des Algotihmus die Laufzeit abzüglich der gemessenen Zeit für die Syscalls an. Jegliche andere Berechnungen in der LibVSM, z.B. die Abfrage, ob ein Speicherbereich bereits im Pagecache ist, wird von der bereinigten Laufzeit dennoch gemessen. Aus den bereinigten Laufzeiten wird erneut das arithmetische Mittel und die relative Änderung im Vergleich zur unoptimierten Laufzeit berechnet.

4.1 Auswertung

Unter allen ausgeführten Testläufen mit jeweils 10 Wiederholungen betrug der maximale beobachtete Stichproben-Variationskoeffizient $\max v = 3.9\%$. Wir verzichten daher zur Übersicht auf die Angabe der Standardabweichung und des Stichproben-Variationskoeffizients.

4.1.1 Vergleich der Algorithmen

Betrachten wir die Veränderung durch die Pagecache-Optimierung in Abbildung 4.3 sehen wir, dass der DotProd-Benchmark in den getesteten Eingabegrößen nicht von der Pagecache-Optimierung profitiert. Der NSieve-Benchmark profitiert zumindest bei großen Eingabegrößen und die BubbleSort- und MatSquare-Benchmarks profitieren bei so gut wie allen Eingabegrößen.

Beim DotProduct-Algorithmus wird nur ein einziges Mal auf jedes Element der beiden Eingabe-Arrays zugegriffen. (Der Akkumulator befindet sich auf dem Stack, also im TLM.) Dadurch führt der zusätzliche Kopiervorgang in den Pagecache zu deutlichem Overhead. Dieser Kopiervorgang benötigt etwa genau so viel Zeit, wie ein regulärer Zugriff über das NoC. Der zusätzliche Aufwand durch die LibVSM

fällt im Vergleich zur insgesamt sehr geringen Aufwand des eigentlichen Algorithmus sehr stark ins Gewicht. Das führt dazu, dass auch die bereinigte Laufzeit keine Verbesserung zeigt.

Beim BubbleSort-Algorithmus wird im Gegenzug viel öfter auf das Array zugegriffen: Jede Vergleichsoperation benötigt 2 Arrayzugriffe, jede Tauschoperation 4 (2 lesende und 2 schreibende). Wir haben den Algorithmus auf den ungünstigsten Fall eines umgekehrt sortierten Arrays angewendet. In diesem Fall sind $\frac{1}{2}(N^2 - N)$ Vergleiche und Vertauschungen notwendig. Dies führt im Schnitt zu deutlich mehr als einem Speicherzugriff je Arrayelement. Der Pagecache bringt dann eine deutliche Verbesserung. Bei hinreichend großer Eingabegröße wird sogar der Nachteil des Kopierens ohne DMA ausgeglichen.

Insgesamt stellen wir fest, dass sich die Pagecache-Optimierung umso mehr lohnt, je öfter auf ein Speicherelement, das in den Pagecache kopiert wurde, zugegriffen wird.

4.1.2 Analyse nach Eingabegröße

Vergleichen wir die Veränderung der Laufzeiten durch die Pagecache-Optimierung für verschiedene Eingabegrößen des gleichen Algorithmus (Abbildung 4.4) sehen wir, dass die meisten Algorithmen bei größeren Eingaben stärker vom Pagecache profitieren.

Die Pagecache-Optimierung fügt vor Beginn und nach Ende der äußersten Schleife jeder Funktion neue Befehle (die Berechnung des Speicherbereichs und die LibVSM-Aufrufe) ein. Eine Ausnahme dazu bildet der Fall, dass eine Schleife nicht analysiert werden konnte, siehe dazu Abschnitt 3.1.7.

Bei den gewählten Algorithmen konnten alle Schleifen analysiert werden, die Befehle wurden also an die äußersten Schleifen der Funktion eingefügt. Das bedeutet, dass die Anzahl der Aufrufe an die LibVSM und die Zahl der zu berechnenden Speicherbereiche konstant über verschiedene Eingabegrößen ist.

Der Aufwand für das Filtern eines Speicherbereichs in der `vsm_prefetch`-Funktion in der LibVSM steigt linear mit der Zahl der Speicherbereiche, die bisher im Pagecache sind (lineare Suche). Diese Zahl ist in den getesteten Fällen ebenfalls konstant über verschiedene Eingabegrößen.

Das Kopieren eines Speicherbereichs in den Pagecache ist abhängig von der Anzahl der Seiten die kopiert werden müssen. Der Aufwand steigt linear mit der Anzahl der Seiten, also auch linear mit der Größe des Speicherbereichs. Diese ist in den

ausgewählten Algorithmen abhängig von der Eingabegröße N. Dieser Zusammenhang ist in Abbildung 4.2 als *Speicherbedarf* angegeben.

Bei der Ausführung sorgt jede Speicheroperation für eine etwa gleich große Verkürzung der Laufzeit bei aktiviertem Pagecache.

Bei Erhöhen der Eingabegröße steigt die Anzahl der Speicheroperationen und die Größe des Speicherbereichs, der in den Pagecache kopiert werden muss und damit die Laufzeit des Kopiervorgangs. Der genaue Zusammenhang hängt vom Algorithmus ab. Der Aufwand für Berechnen des Speicherbereichs und Tests in der LibVSM bleibt unverändert.

Somit sehen wir für die nicht-bereinigte Veränderung in fast allen Tests eine Verbesserung der relativen Änderung r. (Siehe Abbildung 4.4). Beim DotProd-Algorithmus ist sowohl die Zahl der Speichervorgänge als auch der Speicherbedarf linear. Deshalb erfährt er bei Erhöhen der Eingabegröße keine bis eine sehr geringe Laufzeitverbesserung.

In der bereinigten Laufzeit, bei der der das Kopieren des Speichers in den Pagecache nicht gezält wird, erfährt auch der DotProd-Algorithmus eine Verbesserung bei Erhöhen der Eingabegröße.

4.1.3 Korrektheit der Pagecache-Optimierung

Insgesamt sehen wir, dass alle Schleifen in den Algorithmen korrekt analysiert wurden. Ebenfalls wurden alle Speicheroperationen auf die Eingabearrays der Algorithmen erkannt.

Bei den evaluierten einfachen Algorithmen erzeugte die Heuristik, die den Wirkungsbereich eine Speicheroperation abschätzt, sogar exakte Ergebnisse. Das Verhalten bei Ausnahmefällen wie nicht-monotonen Schleifenvariablen, Schleifen die nicht auf jedes Arrayelement zugreifen und Endlosschleifen betrachten wir jedoch in dieser Arbeit nicht.

Im Folgenden wollen wir einige potenziell unerwünschte Verhaltensweisen durch die Pagecache-Optimierung betrachten, die bei der Evaluation aufgefallen sind.

Beinahe überlappende Speicherbereiche

Beim Ausführen des BubbleSort-Algorithmus beobachteten wir vor Betreten der äußeren Schleife folgende Aufrufe an die LibVSM:

`vsm_prefetch`(a[0], a[N-1])
`vsm_prefetch`(a[1], a[N])

Diese entstehen dadurch, dass im Bubblesort-Algorithmus sowohl auf `a[i]` als auch `a[i+1]` zugegriffen wird. Durch die zwei `vsm_prefetch`-Aufrufen löst die LibVSM zwei acquire-Syscalls aus: `acquire`(a[0], a[N-1]) und `acquire`(a[N-1], a[N]).

Da Syscalls mit einigem Aufwand verbunden sind, wäre es wünschenswert, dieses Verhalten zu unterbinden. In der Regel führt jedoch der zweite acquire-Aufruf nicht zu einem erneuten Kopieren einer Speicherseite, da sich die entsprechende Seite bereits durch den ersten Aufruf im Pagecache befindet.

Wiederholtes Kopieren des gleichen Speicherbereichs

Der NSieve-Agorithmus bildet das Streichen von Zahlen im Sieb des Erastothenes mit einem Array ab, der für jede Zahl entweder eine 1 oder eine 0 enthält je nachdem, ob sie durchgestrichen wurde.

Zu Beginn des Algorithmus wird jedes Element dieses Arrays in einer Schleife mit 0 initialisiert. Vor Betreten der Schleife wird dieser Array in den Pagecache kopiert, und nach Verlassen wieder freigegeben. Gleich danach wird der Speicherbereich erneut in den Pagecache kopiert, da für den eigentlichen Algorithmus ebenfalls darauf zugegriffen wird.

DotProd					
N	ohne PC	mit PC	bereinigt	Veränderung	bereinigt
1000	$0.00057s$	$0.00687s$	$0.00073s$	$+1100.88\%$	$+28.39\%$
10000	$0.00315s$	$0.03213s$	$0.00332s$	$+918.42\%$	$+5.17\%$
100000	$0.02831s$	$0.31613s$	$0.02848s$	$+1016.65\%$	$+0.60s$

NSieve					
N	ohne PC	mit PC	bereinigt	Veränderung	bereinigt
100	$0.00030s$	$0.00959s$	$0.00058s$	$+3108.95\%$	$+95.22\%$
1000	$0.00126s$	$0.01054s$	$0.00138s$	$+739.37\%$	$+9.86\%$
10000	$0.01384s$	$0.02794s$	$0.01116s$	$+101.89\%$	-19.41%
100000	$0.23764s$	$0.27652s$	$0.17416s$	$+16.36\%$	-26.72%
1000000	$5.82569s$	$3.52204s$	$2.21174s$	-39.64%	-62.03%

BubbleSort					
N	ohne PC	mit PC	bereinigt	Veränderung	bereinigt
100	$0.00246s$	$0.00961s$	$0.00254s$	$+290.90\%$	$+3.58\%$
1000	$0.21838s$	$0.21689s$	$0.20950s$	-0.68%	-4.06%
10000	$29.5251s$	$25.1507s$	$25.1207s$	-14.82%	-14.92%
100000	$3013.54s$	$2548.14s$	$2547.86s$	-15.44%	-15.45%

MatSquare					
N	ohne PC	mit PC	bereinigt	Veränderung	bereinigt
100	$0.57441s$	$0.53190s$	$0.50179s$	-7.40%	-12.64%
200	$6.75411s$	$6.04691s$	$5.93690s$	-10.47%	-12.10%
400	$129.183s$	$92.2485s$	$91.7413s$	-28.59%	-28.98%
600	$527.070s$	$391.945s$	$390.375s$	-25.64%	-25.93%

Abbildung 4.3: Ergebnisse der Testläufe. Mit Eingabegröße N, durchschnittlicher Ausführungszeit ohne Pagecache, durchschnittlicher Ausführungszeit mit Pagecache, durchschnittlicher bereinigter Ausführungszeit mit Pagecache, sowie Veränderung r von Ausführungszeit bzw. bereinigter Ausführungszeit mit Pagecache zu Ausführungszeit ohne Pagecache.

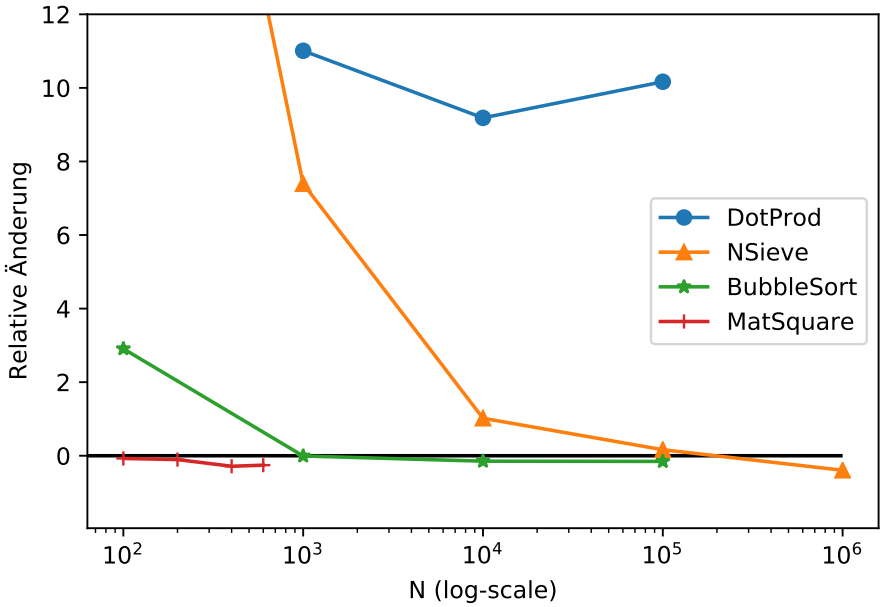


Abbildung 4.4: Plot der relativen Änderung der nicht-bereinigten Ausführungszeit über die Eingabegröße

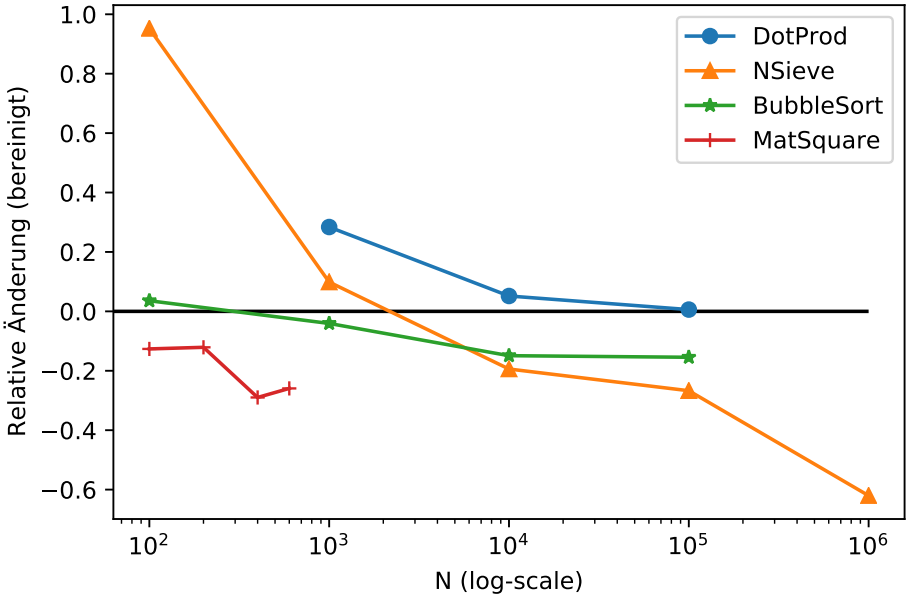


Abbildung 4.5: Plot der relativen Änderung der bereinigten Ausführungszeit über die Eingabegröße

5 Fazit und Ausblick

Die Auswertung hat gezeigt, dass mit Hilfe des Pagecache in einigen Fällen deutliche Verbesserungen der Laufzeit zu erzielen sind. Ob eine Verbesserung zu erzielen ist und wie groß diese Ausfällt hängt sowohl vom Algorithmus ab, der optimiert wird als auch von dessen Eingabegröße.

Die Pagecache-Optimierung hatte bei den ausgewählten Algorithmen keine Probleme die Schleifen zu analysieren und die Speicherbereiche zu ermitteln. Es bleibt für zukünftige Arbeiten dies mit komplexeren Programmen zu validieren und zu untersuchen, welchen Einfluss auf die Effektivität der Optimierung Schleifen haben, die nicht analysiert werden können.

Die naive Implementierung der LibVSM, bei der jeder Speicherbereich ungeachtet der Anzahl Schleifeniterationen und Anzahl der Speicheroperationen je Iteration, in den Pagecache geladen wird, ist nicht ausreichend. Sie führt in vielen der getesteten Fälle zu einer Verschlechterung der Programmlaufzeit.

Wir glauben, dass die hier vorgestellte Optimierung ein guter Anfang ist, der mit einigen Erweiterungen und Abstimmen auf die Hardware Performance-Steigerungen ohne zusätzlichen Programmieraufwand bringen kann.

Bevor die Optimierung eingesetzt werden kann, muss jedoch eine geeignete Heuristik gefunden werden, die entscheidet, ob ein Speicherbereich in den Pagecache geladen wird. Die Ergebnisse der Evaluation lassen uns vermuten, dass die bereits von der Optimierung bereitgestellte Information, wie oft eine Speicheroperation ausgeführt wird, ein guter Ansatzpunkt für diese Entscheidung ist.

5.1 Künftige Arbeit

In der Pagecache-Optimierung wurde bereits in dieser Arbeit eine Heuristik dafür implementiert, wie häufig eine Speicheroperation in der gesamten Laufzeit einer (äußeren) Schleife aufgerufen wird. Diese Information wurde in der LibVSM jedoch ignoriert. Es wäre möglich zur Laufzeit am Beginn einer Schleife zunächst alle

Informationen über die Speicheroperationen in der Schleife zu sammeln, und eine Art „Heatmap“ für den Speicher zu erstellen, die für jeden Speicherbereich angibt, wie oft darauf zugegriffen wird. Anhand dieser Informationen kann der Pagecache dann effizient befüllt werden.

Die aktuelle Implementierung ignoriert Speicherbereiche, die größer sind als der Pagecache der Kachel. Hier könnte bei weiterführender Analyse der Schleife der Speicherbereich auf die Schleifeniterationen aufgeteilt werden: Nach einem Teil der Schleifeniterationen wird der Speicherbereich im Pagecache durch den Nächsten ausgetauscht.

Ebenfalls wurde in dieser Arbeit kein Multithreading betrachtet. Eine gleichzeitige Ausführung mehrerer iLets auf der gleichen Kachel könnte dazu führen, dass der Pagecache ausgelastet ist und ineffizient zwischen den iLets aufgeteilt wird.

Großes Optimierungspotential sehen wir in aufeinanderfolgenden Schleifen, die auf gleiche Speicherbereiche zugreifen, was zu wiederholtem Kopieren von Seiten in den Pagecache führt. Würden diese Wiederholungen erkannt, ist bei komplexeren Programmen mit einer deutlichen Reduktion der Kopiervorgänge zu rechnen.

Die Pagecache-Optimierung wurde vollständig im Middle-End der libFirm implementiert. Dies erlaubt es, sie auch mit anderen Frontends als dem in dieser Arbeit verwendeten CParser zu verwenden. So existiert für die Programmiersprache X10 bereits ein Compiler auf Basis von libFirm [12], welcher im InvasIC-Projekt schon länger benutzt wird.

Literaturverzeichnis

[1] M. Braun, S. Buchwald, and A. Zwinkau, "FIRM-A Graph-Based Intermediate Representation," tech. rep.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.

[3] C. Lameter, "Numa (non-uniform memory access): An overview," *Queue*, vol. 11, no. 7, pp. 40–51, 2013.

[4] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting, "Invasive computing: An overview," *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, pp. 241–268, 2011.

[5] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, W. Schröder-preikschat, and F. Alexander, "OctoPOS: A Parallel Operating System for Invasive Computing," *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys*, pp. 9–14, 2011.

[6] S. Rheindt, A. Fried, O. Lenke, L. Nolte, T. Wild, and A. Herkersdorf, "NEMESYS," in *Proceedings of the International Symposium on Memory Systems - MEMSYS '19*, (New York, New York, USA), pp. 3–18, ACM Press, 2019.

[7] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, vol. 27, pp. 62–73, 1992.

[8] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira, "Compiler support for selective page migration in NUMA architectures," in *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pp. 369–380, Institute of Electrical and Electronics Engineers Inc., 2014.

[9] M. Diener, E. H. Cruz, M. A. Alves, E. Borin, and P. O. Navaux, "Optimizing

memory affinity with a hybrid compiler/OS approach," in *ACM International Conference on Computing Frontiers 2017, CF 2017*, pp. 221–229, Association for Computing Machinery, Inc, may 2017.

[10] S. Seoy, J. Lee, and Z. Suraz, "Design and implementation of software-managed caches for multicores with local memory," *Proceedings - International Symposium on High-Performance Computer Architecture*, pp. 55–66, 2009.

[11] J. Becker, S. Friederich, J. Heisswolf, R. Koenig, and D. May, "Hardware prototyping of novel invasive multicore architectures," in *17th Asia and South Pacific Design Automation Conference*, pp. 201–206, 2012.

[12] M. Braun, S. Buchwald, M. Mohr, and A. Zwinkau, "An x10 compiler for invasive architectures," Tech. Rep. 9, Karlsruhe Institute of Technology, 2012.

Erklärung

Hiermit erkläre ich, Leon Bentrup, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift