Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Frühzeitige Ausführung von parallelize-mem

Bachelorarbeit von

Jonas Schwabe

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Sebastian Graf
Abgabedatum:	13. Juni 2019

Zusammenfassung

In dieser Bachelorarbeit wird die Reihenfolge der Optimierungsschritte des Compilers libFirm mit dem Ziel angepasst, dass eine Partitionierung der Speicherzugriffe im Firm-Graph zu einem früheren Zeitpunkt ausgeführt wird. Um dieses Ziel zu erreichen, werden jene Optimierungsschritte betrachtet, welche durch die Änderung der Ausführungsreihenfolge anschließend nach der Speicherpartitionierung ausgeführt werden. Durch die veränderte Graphstruktur kommt es zu fehlerhaften oder schlechteren Optimierungen in einigen dieser Schritte. Diese werden im Zuge dieser Bachelorarbeit untersucht und gelöst. Die angepasste Optimierungsreihenfolge sorgt dafür, dass bereits frühzeitig Abhängigkeiten zwischen Speicherzugriffen im Graph abgebildet werden. Hierdurch stehen Informationen zu Alias-Relationen zur Verfügung, die zuvor jeweils neu erzeugt werden mussten. Durch die vorgezogene Ausführung der *parallelize-mem* Operation lassen sich die Alias-Relationen früher und einfacher auch in anderen Optimierungsschritten verwenden.

Inhaltsverzeichnis

1. Einleitung

Compiler werden verwendet, um Programmcodes, die in einer höheren Programmiersprache entwickelt wurden, in Maschinencodes zu übersetzen. Die Entwickler*innen vertrauen hierbei darauf, dass das erzeugte Kompilat die Semantik des Quellprogramms erhält, das Programm also die im Code definierten Instruktion korrekt und in der richtigen Reihenfolge ausführt. Ein Compiler sollte unter dieser Maßgabe außerdem möglichst schnelle Maschinencodes erzeugen. Insbesondere sollten Instruktionen, die nicht von äußeren Einflüssen abhängen, bereits bei der Kompilierung, und damit schon vor Ausführung des Programms, durchgeführt werden. Daneben können strukturelle Änderungen vorgenommen werden, insofern die Semantik des Programmes hierdurch nicht beeinflusst wird.

libFirm ist ein Compiler, der auf einer Graphrepräsentation des Quellcodes operiert [1]. Dieser Graph fungiert als Zwischensprache und erlaubt die Verwendung von Firm mit unterschiedlichen Programmiersprachen, insofern diese in einen passenden Graphen überführt werden können. Ein solcher Graph wird durch Firm auf verschiedene Arten optimiert. Bei der Anwendung dieser Optimierungsschritte muss sichergestellt werden, dass das im Graph codierte Programm vor und nach der Optimierung die gleiche Semantik besitzt.

Ziel der vorliegenden Bachelorarbeit ist die Anpassung der Ausführungsreihenfolge dieser Optimierungsschritte. Die Optimierung *parallelize-mem* soll dazu zu einem möglichst frühen Zeitpunkt ausgeführt werden. *parallelize-mem* stattet den Graph mit zusätzlichen Informationen über das Verhältnis verschiedener Schreib- und Leseoperationen aus, die in anderen Optimierungsschritten verwendet werden können.

Die in einem Firm-Graphen als Knoten dargestellten Speicheroperationen sind vor der Ausführung von *parallelize-mem* basierend auf dem Quelltext der Eingabe total geordnet. Ein Programm, welches mehrere Speicherbereiche schreibt, würde dies also ohne *parallelize-mem* in genau der von den Programierer*innen vorgesehenen Reihenfolge tun. Nach der Ausführung der Optimierung können die Speicherzugriffe je nach Verhältnis der Speicherbereiche untereinander neu angeordnet werden. So ist es für die Ausführung des folgenden C Codes unerheblich, ob zunächst `a` oder `b` beschrieben wird, solange beide Zuweisungen vor der Addition durchgeführt wurden.

Listing 1.1: Beispiel für Zuweisungsreihenfolge von a und b

```
int a, b;
int main(void) {
    a = 5;
    b = 10;
    return a + b;
}
```

Wie im Folgenden gezeigt wird, sind Daten über das Verhältnis von Speicheroperationen für verschiedene Optimierungsschritte essenziell. Die Informationen werden außerdem beim Erzeugen des Maschinencodes verwendet, um eine hinsichtlich der Performance optimale Zugriffsreihenfolge auf Speicherbereiche zu finden.

In der vorliegenden Bachelorarbeit werden die folgenden Themen behandelt:

- Die Grundlagen von libFirm und den behandelten Optimierungsschritten werden erläutert.
- Die Vorgehensweise bei der Umstellung der Optimierungsschritte wird eingeführt.
- Die Probleme bei der Behandlung von Speicherpartitionen mit der Load-Store-Optimierung werden evaluiert und behoben, dazu wird insbesondere erläutert wie die Struktur der Speicherpartitionen bestimmt werden und während der Optimierung korrekt ausgelesen werden kann.
- Die in der lokalen Optimierung entstehenden Probleme werden erläutert und gelöst.
- Die Ergebnisse werden evaluiert.
- Ausblicke auf mögliche Anwendungen werden eingebracht.

2. Grundlagen

Im folgenden werden die theoretischen Grundlagen und Begriffe, die zum Verständnis der vorliegenden Bachelorarbeit nötig sind, erläutert.

2.1. libFirm

libFirm ist eine Bibliothek zur Generierung, Optimierung und Kompilierung einer graphenbasierten Repräsentation höherer Programmiersprachen. Die Bibliothek besteht aus Algorithmen zur Erzeugung und Veränderung von Graphen, die zur Entwicklung eines programmiersprachenspezifischen Frontends verwendet werden können. Zusätzlich umfasst sie verschiedene Backends zur Generierung von Assemblercode für spezielle Computerarchitekturen sowie Algorithmen zur Optimierung der Graphen. Anpassungen an einigen dieser Optimierungen zur Sicherstellung der korrekten Funktion nach Umstellung der Optimierungsreihenfolge, sind Gegenstand der vorliegenden Arbeit. Zum aktuellen Zeitpunkt kann libFirm Programme für IA32, SPARC und mit experimenteller Implementierung für MIPS, ARM und AM64 erzeugen. [1]

2.2. cparser

cparser ist ein Frontend für libFirm mit dem C99 Quellcode kompiliert werden kann. Es besteht aus Präprozessor, Lexer sowie einem parser. *cparser* ist in der Lage einen abstrakten Syntaxbaum zu erzeugen und erste semantische Analysen durchzuführen, bevor der durch Verwendung von libFirm erzeugte Repräsentationsgraph an die Optimierungsroutine von *Firm* übergeben wird [2]. Durch die in libFirm implementierten Backends ist es mithilfe von *cparser* möglich, Programme für verschiedene Plattformen zu erzeugen.

In dieser Bachelorarbeit wird *cparser* zur Verifikation der Funktionalität von libFirm nach etwaigen Änderungen ausgeführt. Dazu wird *cparser* entweder durch die Test-

bibliothek Firm Testsuite aufgerufen, oder mit Testcode verwendet, welcher speziell zu diesem Zwecke implementiert wurde.

2.3. Firm-Graphen

Ein Firm-Graph repräsentiert einen Programmteil, der in der Regel in einer höheren Programmiersprache implementiert wurde. Ein Programm, welches aus mehreren unabhängigen Teilen besteht, wird in mehrere Firm-Graphen übersetzt die unabhängig voneinander behandelt werden können. So wird ein C Programm das aus unterschiedlichen Funktionen besteht, in die gleiche Anzahl Graphen umgewandelt. Diese können anschließend jeweils separat optimiert werden.

Jeder Firm-Graph besteht aus mehreren *Knoten* verschiedenen Types. Diese sind in *Blöcken* organisiert, welche durch *Kontrollfluss-* sowie *Speicherflusskanten* verbunden sind. Wird ein solcher Block durch eine Kontrollflusskante betreten, wird jede zugehörige Instruktion ausgeführt. Auf Details zu den einzelnen Knoten- und Kantentypen wird in der nachfolgenden Ausführung näher eingegangen.

2.4. Firm Testsuite

Um die Funktionalität der libFirm Bibliothek effizient überprüfen zu können, wurde die Firm Testsuite [3] entwickelt. Diese besteht zu einem Großteil aus Regressionstests.

2.5. Knoten

Firm nutzt unterschiedliche Knotentypen, um verschiedene Programme abzubilden.

Speicherknoten Speicherknoten interagieren mit dem Speicher des Computers, oder bilden einen speziellen Zustand ab. Die für die Aufgabenstellung relevanten Speicherknoten sind:

- **Load**: Liest einen Wert aus dem Speicher an einer angegebenen Adresse.

- **Store**: Schreibt einen Wert in den Speicher an einer angegebenen Adresse.
- **Sync**: Führt mehrere Speicherpartitionen zusammen. Diese Speicherpartitionen wurden zuvor durch die Ausführung von *parallelize-mem* erzeugt.
- **Proj**: Gibt spezielle Werte aus einem Tupel zurück.

Kontrollflussknoten Kontrollflussknoten verbinden Knoten mit im Anschluss auszuführenden Blöcken. Beispielsweise wird der bei einer Fallunterscheidung auszuführende Code in separate Blöcke ausgelagert, welche über eine Kontrollflusskante nach Evaluation der Bedingung ausgeführt werden können.

Konstantenknoten Konstantenknoten enthalten konstante Werte wie Programmkonstanten oder Adressen.

2.5.1. Blöcke

Blöcke sind spezielle Knoten, die mehrere Knoten zu einer Menge zusammenfassen. In einer grafischen Darstellung umschließen Blöcke die zugehörigen Knoten wie unter anderem in Abbildung 2.1 zu sehen ist. Für die so generierten Konstrukte ist die von Vorbedingungen abhängige Ausführung von Teilen des Graphen grundlegend (so beispielsweise bei der Verwendung von Bedingungen oder Schleifen). Strukturell muss hierbei eine Kontrollflusskante zwischen den Blöcken existieren. Dabei verweist ein Kontrollflussknoten innerhalb eines Blocks auf einen Block. Erreicht der Kontrollfluss den entsprechenden Knoten, wird zu dem jeweiligen Block gesprungen. Blöcke in libFirm sind als Grundblöcke organisiert. Das bedeutet, dass nie einzelne Knoten innerhalb eines Blocks ausgeführt werden. Wird ein Block betreten, so werden immer alle enthaltenen Instruktionen ausgeführt [4].

2.5.2. Kontrollfluss

Der Kontrollfluss wird über Kanten zwischen Kontrollflussknoten und Blöcken dargestellt. Dies wird durch die zuvor beschriebene Eigenschaft der Blöcke möglich. Wird ein Block erreicht, so wird jede zugehörige Instruktion ausgeführt. Ein Block enthält somit immer einen zusammenhängenden Programmteil, der sequenziell und ohne Unterbrechung ausgeführt wird.

Eine `if`-Abfrage wird beispielsweise in mindestens zwei Blöcke zerlegt. Ist die Ausgabe der Abfrage `Wahr`, wird der Kontrollflusskante in den konditionalen Block gefolgt. Gibt die Abfrage `Falsch` zurück, wird in einen Block gesprungen, der den nachfolgenden Programmcode enthält. Nach der Ausführung der Abfrage laufen beide Kontrollflüsse wieder zusammen.

In den Abbildungen werden Kontrollflusskanten rot dargestellt. Abbildung 2.1 zeigt eine Methode mit `if`-Abfrage. Ist die Abfrage `wahr`, wird 1 zurückgegeben. Gibt die Abfrage `Falsch` zurück wird 0 zurückgegeben.

2.5.3. Speicherkanten

Speicherkanten verbinden Knoten, die mit dem Arbeitsspeicher interagieren. Der Subgraph bestehend aus Speicherkanten und mit diesen inzidenten Knoten geht vom Programmende aus und führt über Speicherknoten bis zum Startknoten. Ein einfacher Graph welcher ein Programm repräsentiert, in dem eine einfache Zuweisung stattfindet, ist in Abbildung 2.2 zu sehen. In den Abbildungen werden Speicherkanten blau dargestellt.

Durch die Verbindung von Knoten mit Speicherkanten entsteht eine totale Ordnung der Speicherzugriffe im repräsentierten Programm. Jeder Vorgänger im Graph operiert auf dem durch seine Nachfolger veränderten Speicher. Ein Knoten, der auf Speicher zugreift, kann so den Speicher Subgraph traversieren, um für die Operation relevanten Knoten in der korrekten Reihenfolge zu finden.

2.6. Static Single Assignment

Firm-Graphen befinden sich in Static Single Assignment (SSA) Form [5]. Das bedeutet, dass jede in der Graphrepräsentation deklarierte Variable nur einmal geschrieben wird und ihren Wert anschließend bis zum Ende ihrer Sichtbarkeit behält. Wird eine Variable im Quellprogramm erneut beschrieben, so wird in der SSA Form eine weitere Variable deklariert, die für darauf folgende Zugriffe bis zu einer weiteren Zuweisung verwendet wird. In Listing 2.1 und 2.2 ist die Umwandlung einfacher Zuweisungen dargestellt.

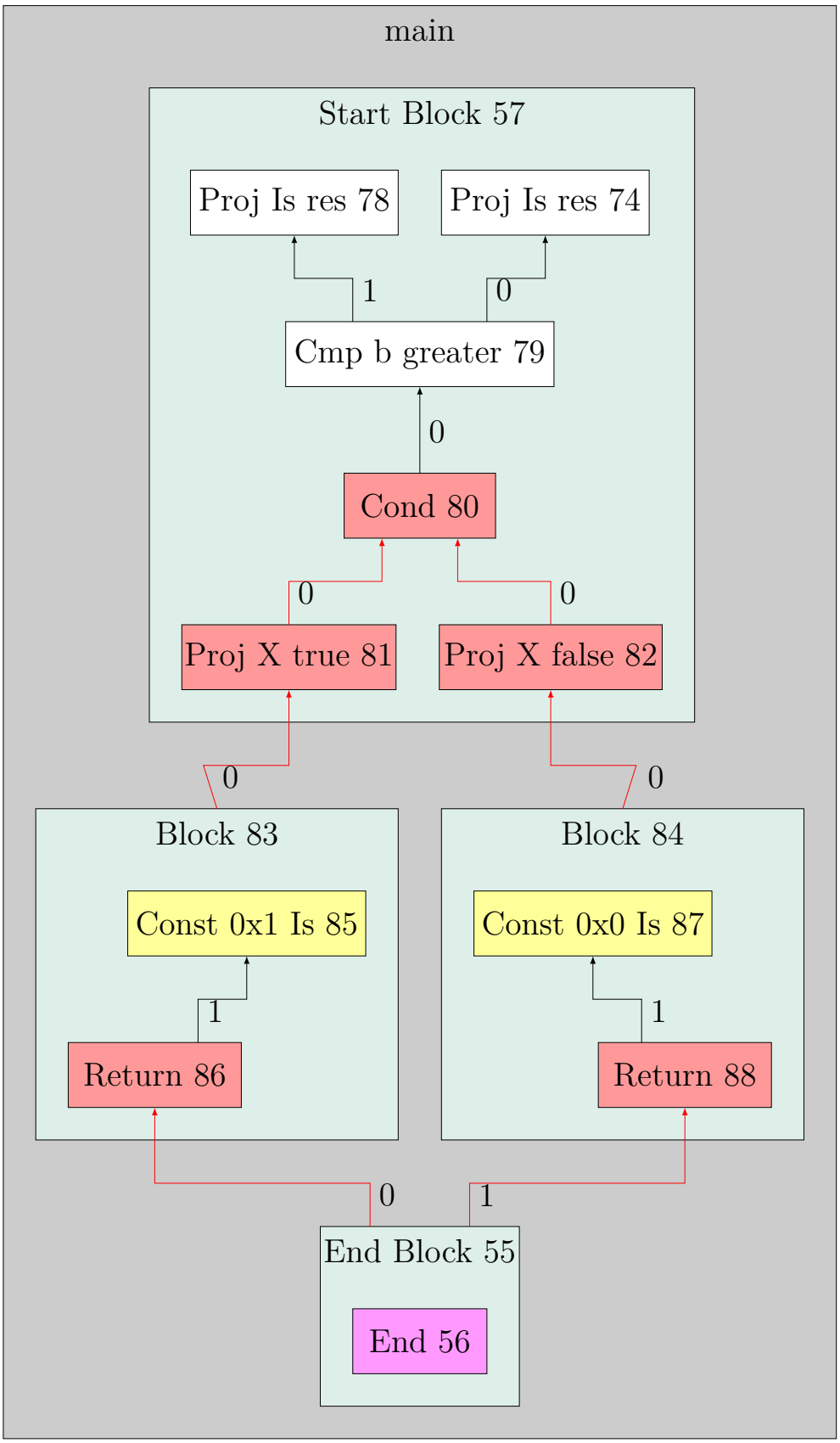

Abbildung 2.1.: Beispielgraph Kontrollfluss

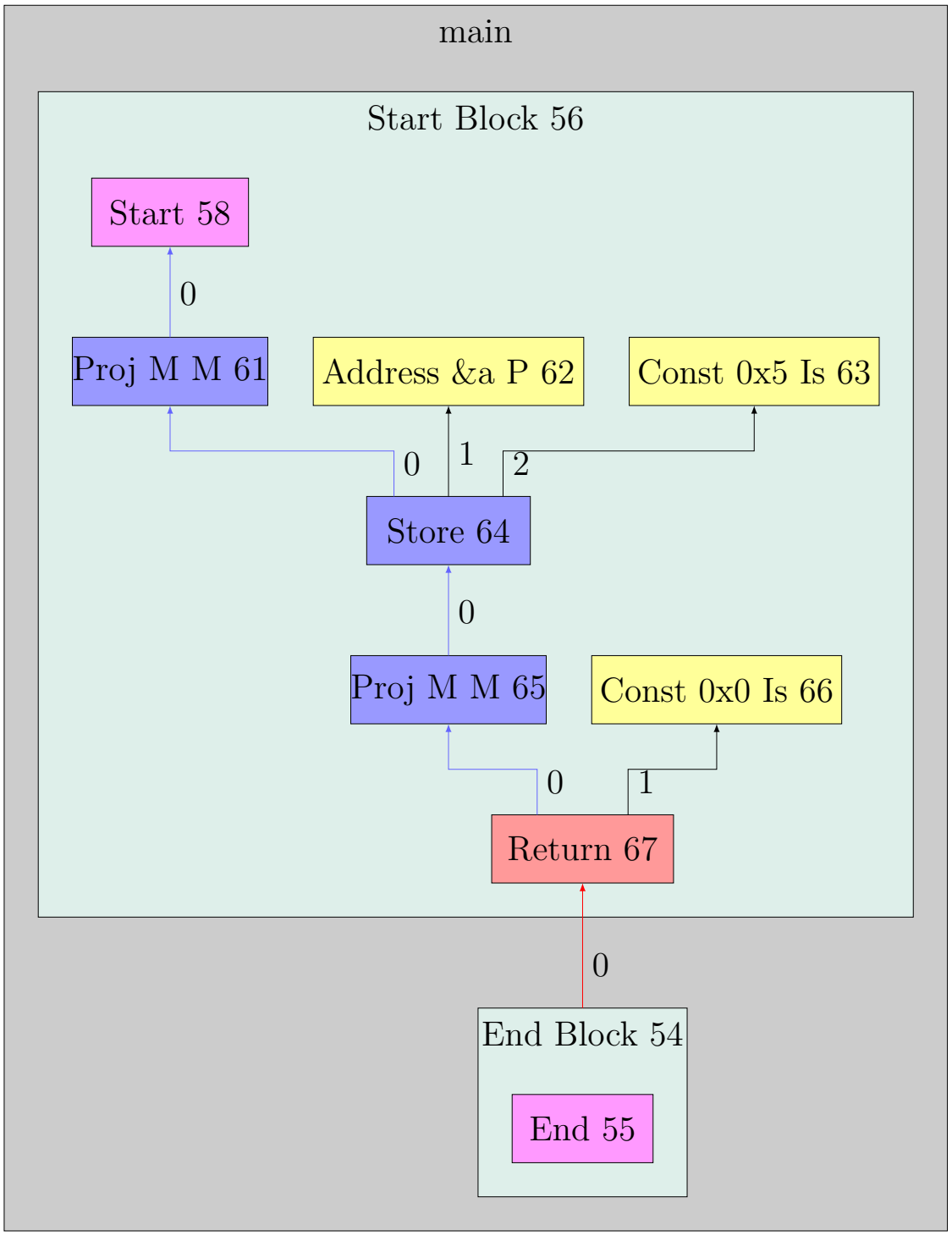

Abbildung 2.2.: Beispielgraph Speicherkanten

Listing 2.1: Nicht SSA Form

$$A \leftarrow 4$$
$$A \leftarrow 6$$
$$B \leftarrow A + 6$$

Listing 2.2: SSA Repräsentation

$$A_1 \leftarrow 4$$
$$A_2 \leftarrow 6$$
$$B_1 \leftarrow A_2 + 6$$

Falls eine Zuweisung in einer Schleife ausgeführt wird oder konditional ist, wird eine Funktion zugewiesen, die bei der Evaluation unterschiedliche Werte annehmen kann. Diese Funktion wird als ϕ-Funktion bezeichnet.

Lokale Variablen aus dem Quellprogramm werden in der `Firm`-Graphstruktur direkt aufgelöst, sodass die durch die folgenden Instruktionen konsumierten Inhalte der ursprünglichen Variablen direkt referenziert werden. Durch die Entfernung lokaler Variablen ist eine Umbenennung der Variablen bei folgenden Schreibvorgängen wie oben erläutert nicht nötig. Für globale Variablen und den entsprechend in `Firm` referenzierten Speicherzellen gelten durch die Einführung von *Load*- und *Store*-Knoten analoge Regeln. Eine Instruktion, die mit dem entsprechendem Speicher interagiert, referenziert den aktuellen *Load*-Knoten. Zugriffe auf eine Variable referenzieren über Speicherkanten die nächstgelegene Zuweisung einer Variable. Ein `Firm`-Graph ist somit in SSA Form. ϕ-Funktionen werden im Graph als ϕ-Knoten codiert, welche abhängig von der verwendeten Eingangskante in einen Block zu einem anderen Wert evaluieren.

Die SSA Form ist insbesondere in der im Folgenden beschriebenen Load-Store-Optimierung von großem Nutzen, da die Zugriffe auf einen konkreten Zustand einer Speicherstelle ohne weitere Analysen voneinander unterschieden werden können.

2.7. Load-Store-Optimierung

In vielen Fällen können Lese- oder Schreibzugriffe auf den Arbeitsspeicher entfernt oder ersetzt werden, ohne die Funktionsweise des Programms zu verändern. Beispielsweise muss eine gerade geschriebene Speicherzelle nicht erneut gelesen werden, wenn der Wert in einem Register vorgehalten wird. Der vorherige Wert kann in diesem Fall direkt weiterverwendet werden. Über die Art der Speicherung wird erst beim Erzeugen des Maschinencodes im Firm Backend entschieden. Die Grundlage dieser Entscheidung wird dabei bereits in den vorherigen Optimierungen gelegt.

In libFirm existieren die im Folgenden ausgeführten Load-Store-Optimierungen:

- **Load-after-Load**: Falls der zu ladende Speicherbereich Teil eines zuvor gela-

denen Speicherbereichs ist, so kann der zweite *Load*-Knoten entfernt werden. Ist ein vorheriger *Load* Teil des aktuellen Loads, so kann der erste *Load*-Knoten erweitert und der zweite *Load*-Knoten entfernt werden.

- **Load-after-Store**: Falls der zu lesende Speicherbereich zuvor geschrieben wurde, so kann der geschriebene Wert verwendet werden. Ein *Load* ist nicht nötig.

- **Store-after-Store**: Falls die zweite *Store*-Instruktion den Speicherbereich des vorherigen *Stores* überlagert, so kann der vorherige *Store*-Knoten entfernt werden.

Durch die totale Ordnung der Speicheroperationen im `Firm`-Graph, enthält der vorherige Schreibzugriff auf eine Speicherzelle genau dann den zuletzt geschriebenen Wert, wenn auf dem Pfad zwischen dem betrachteten Knoten und dem Schreibzugriff kein weiterer Knoten liegt, der schreibend auf denselben Speicherbereich zugreift.

Um herauszufinden, ob ein anderer Schreibzugriff den betrachteten Speicher beeinflusst wird eine Alias-Analyse durchgeführt.

2.7.1. Alias-Analyse

Bei der Betrachtung von zwei Speicherbereichen kann auf Speicherebene zwischen drei Relationen unterschiedenen werden:

Sure-Alias: Beide Speicherbereiche greifen sicher auf den gleichen Speicherbereich zu. Dies ist beispielsweise der Fall, wenn zwei Pointer auf den gleichen Speicherbereich zeigen oder einer der Bereiche vom anderen subsumiert wird.

No-Alias: Beide Speicherbereiche greifen sicher nicht auf den gleichen Speicherbereich zu. Die Zugriffe können in diesem Fall in beliebiger Reihenfolge ausgeführt werden, ohne den Programmfluss zu ändern.

May-Alias: Es kann nicht mit Sicherheit festgestellt werden, wie sich zwei Speicherbereiche zueinander verhalten. Dies geschieht unter anderem, wenn der Wert zweier Pointer zur Kompilierzeit unbekannt ist. Zur Erläuterung dient das Listing 2.3.

Listing 2.3: May-Alias-Relation

```
int mayAlias(int *a, int *b) {
  *a = 3;
  *b = 4;
  return *a + *b;
}
```

In diesem einfachen Beispiel könnten `a` und `b` denselben Speicherbereich referenzieren. Der Rückgabewert des Beispielcodes wäre 8. Würde *a* und *b* nicht auf die gleichen Speicherbereiche referenzieren, so wäre der Rückgabewert 7. Eine Optimierung, die die Summe zur Kompilierzeit berechnet, kann somit allein auf Grundlage der Betrachtung der Funktion *mayAlias* nicht durchgeführt werden. Bettet man den Funktionsaufruf in einen größeren Kontext ein, in dem die entsprechenden Parameter fest definiert sind, so wäre eine Optimierung möglich.

2.8. Speicherpartitionierung

Die Optimierung *parallelize-mem* führt eine Speicherpartitionierung aus. Dabei werden Speicherzugriffe, welche sich innerhalb des selben Blocks befinden und deren zugehörige Speicherknoten paarweise in einer *No-Alias*-Relation stehen, in der Graphrepräsentation aus ihrer totalen Ordnung gelöst und auf eine Ebene verschoben. Semantisch greifen Speicherpartitionen damit auf disjunkte Speicherbereiche zu. Diese werden anschließend mit einem *Sync*-Knoten zusammengeführt. Abbildung 2.3 zeigt in zwei Visualisierungen von Firm-Graphen das Schreiben auf zwei Adressbereiche in sequenzieller und unbestimmter Reihenfolge nach der Speicherpartitionierung. Befinden sich zwei Speicherbereiche in einer May-Alias-Relation so dürfen diese nicht in unterschiedliche Partitionen verschoben werden.

Da die Partitionen voneinander unabhängig sind, müssen die Speicherzugriffe ausschließlich innerhalb ihrer Partition in der ursprünglichen Reihenfolge ausgeführt werden. Bei der Erzeugung des Assemblercodes kann daher hinsichtlich der Performance eine optimale Ausführungsreihenfolge gewählt werden, beispielsweise indem mehrere lesende Zugriffe unmittelbar nacheinander ausgeführt werden, um die Summe der Latenzen zu minimieren.

Ab dem Zeitpunkt der Optimierung sind die unterschiedlichen Alias-Informationen direkt im Graph codiert. Damit ergibt sich bei Betrachtung zweier Knoten kein Alias, falls diese Teil verschiedener Partitionen sind. Über zwei Knoten welche innerhalb derselben Partition liegen lassen sich keine weiteren Aussagen treffen.

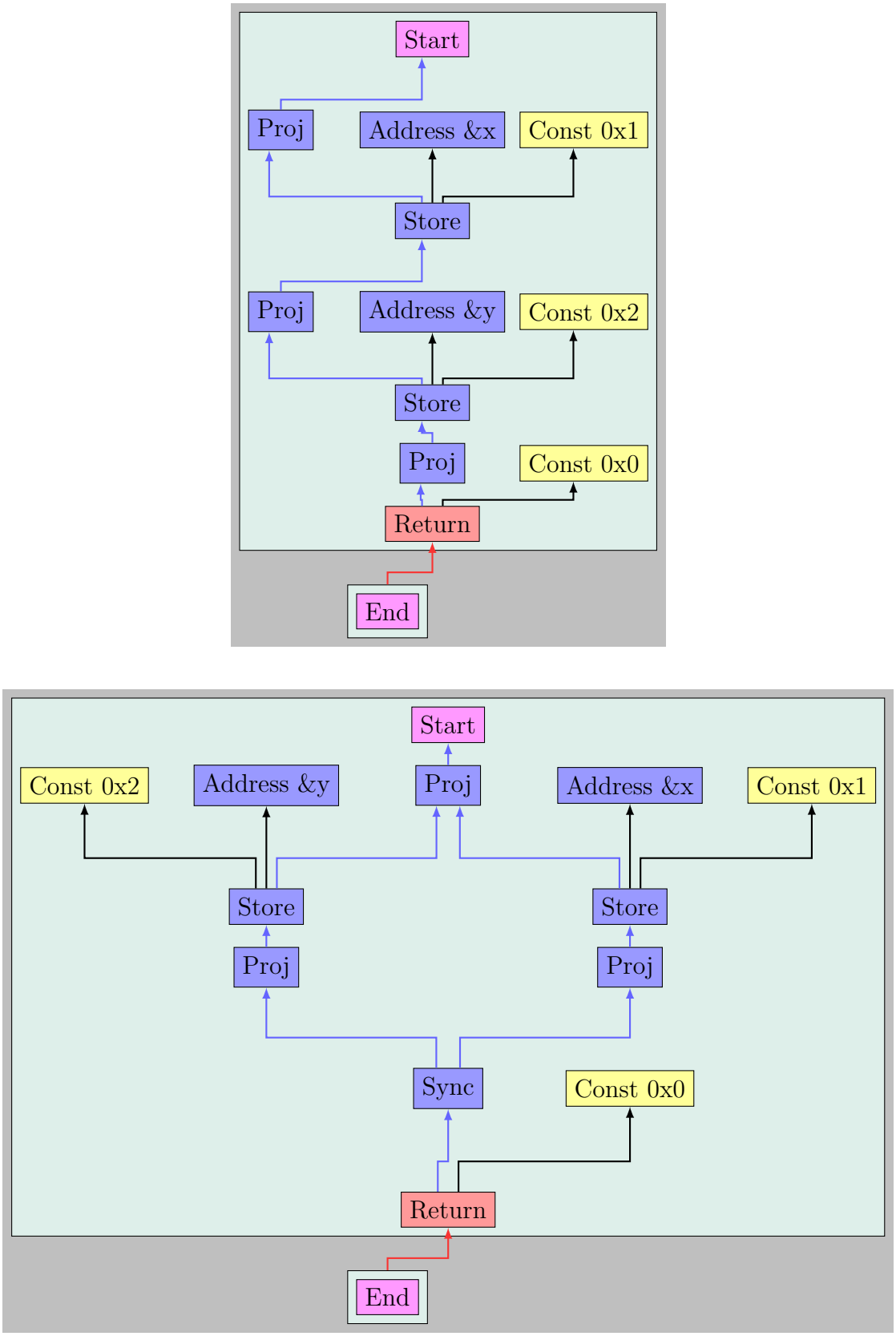

Abbildung 2.3.: Speicherpartitionierung in einem einfachen Beispiel. Oben: Schreiben von x und y in fester Reihenfolge. Unten: Schreiben von x und y in unbestimmter Reihenfolge nach Speicherpartitionierung.

3. Entwurf und Implementierung

3.1. Vorbereitung

3.1.1. Korrektheit

Die *parallelize-mem* Optimierung ist in der Ausgangsstruktur des Kompiliervorganges einer der letzten Schritte. Das Ziel ist, die Optimierung mindestens vor der Load-Store-Optimierung auszuführen. Dazu muss die Speicherpartitionierung deutlich früher durchgeführt werden. Hierzu wird in einem ersten Schritt die Optimierung an den Anfang des Vorgangs verschoben. In der entstandenen Konfiguration wird das Programm mithilfe der Firm Testsuite auf seine Korrektheit geprüft. Diese ist dann gegeben, wenn folgende Optimierungen so implementiert sind, dass die Korrektheitsinvariante der Optimierungen auch bei vorhandenen *Sync*-Knoten und damit gegebenen Speicherpartionen erfüllt ist.

Die Ausführung der Firm Testsuite mit angepasster Ausführungsreihenfolge zeigt, dass mindestens eine Optimierung die frühzeitig hinzugefügten *Sync*-Knoten nicht korrekt behandelt. Auf Grundlage des fehlgeschlagenen Test `ack/test.c` lässt sich ein Beispiel konstruieren, welches bei frühzeitiger Ausführung von *parallelize-mem* ein fehlerhaftes Ergebnis ausgibt. Dieser Fehler ist auf eine fehlerhafte Behandlung von *Sync*-Knoten in der Load-Store-Optimierung zurückführen.

Es zeigten sich Probleme durch die Kontraktion mehrerer Blöcke, wenn *Sync*-Knoten verbunden wurden. Zusätzlich wurde deutlich, dass die lokale Version der Load-Store-Optimierung bisher keinerlei Ansatzpunkte zur Verarbeitung von *Sync*-Knoten besaß.

Die Ursachen sowie Lösungsansätze zu diesen Problemstellungen werden im Folgenden Kapitel erörtert.

3.1.2. Optimierungsergebnis

Zusätzlich zur Korrektheit des Programmes ist es zentral, dass die neue Implementierung Programme erzeugt, die mindestens genauso gut optimiert sind wie die ursprünglichen Versionen. Um die Vergleiche durchzuführen, wurde eine Software in Python entwickelte, die den C-Quelltext mit verschiedenen Compilern kompiliert und den ausgegebenen Assemblercode vergleicht.

Eine erste Prüfung erfolgte hierbei durch Vergleich des erzeugten Assemblercodes. In vielen Fällen war hier die Gleichheit der erzeugten Programme direkt abzulesen. Daneben kam es in einigen Fällen zu anderen Reihenfolgen bei der Ausführung verschiedener Graphpartitionen, die durch die frühere Optimierung ausgelöst wurden. Da in diesen Fällen keine für diese Arbeit relevanten Unterschiede zu den Ergebnissen der ursprünglichen Compilerversion existieren, ist eine weitere Analyse dieser Testfälle nicht nötig.

Die Prüfung der verbleibenden Fälle zeigt Probleme auf, die sich auf falsche oder fehlende Behandlung von *Sync*-Knoten in den speziellen Optimierungen zurückführen lassen. Diese Problematik wird im Folgenden näher untersucht.

3.2. Implementation

Aufbauend auf den Beobachtungen, sollen nun konkrete Probleme identifiziert und behoben werden. Dazu wird die *parallelize-mem* Optimierung an den Anfang der Optimierungskette verschoben, um zu prüfen, in welchen folgenden Optimierungsschritten die ersten Fehler auftreten. Falls vor Ausführung der Load-Store-Optimierung eine Optimierung fehlgeschlagen wäre, so hätte die Speicherpartitionierung in der Abfolge der Optimierungsschritte weiter nach hinten verschoben werden können. Als Ergebnis der Untersuchungen zeigte sich, dass die Load-Store-Optimierung die erste problematische ist. Auf Grundlage dieser Beobachtung wurde die Optimierungsreihenfolge angepasst und *parallelize-mem* an den Anfang der Kette gesetzt. Daraus resultierende Fehler weredn im weiteren Verlauf untersucht und behoben.

In der nachfolgenden Ausführung werden die aufgezeigten Probleme näher betrachtet sowie entsprechende Lösungsansätze erarbeitet.

3.2.1. *Sync*-Knoten in der Load-Store-Optimierung

Die Load-Store-Optimierung liefert bei vorhandenen *Sync*-Knoten nicht in allen Fällen Ergebnisse mit gleicher Semantik wie das Quellprogramm.

Der Fehler kann durch einen einfachen Codeabschnitt ausgelöst werden und wird in der Firm Testsuite abgedeckt. In Listing 3.1 ist der zum Verständnis des Problems relevante Pseudocode dargestellt.

Listing 3.1: Load-Store-Optimierung Fehler Beispiel

```
a, b, c: int
a ← 5
b ← 10
c ← 15
// a, b und c verwenden
a ← 42
b ← 42
c ← 42
result ← a = c && a = b
```

Obwohl `a`, `b` und `c` vor dem Vergleich am Ende des Listings mit demselben Wert beschrieben werden, kann es unter gewissen Umständen zur Rückgabe `false` kommen.

Um den Fehler zu erfassen, ist ein tieferes Verständnis für die Funktionsweise der Optimierung grundlegend.

Funktionsweise der Load-Store-Optimierung

Die Load-Store-Optimierung iteriert über alle im Graph vorhandenen Knoten. Die tatsächlich durchgeführte Optimierung hängt vom entsprechenden Knotentyp ab. Der betrachtete Fehler tritt bei der Behandlung von *Store*- oder *Load*-Knoten auf.

In beiden Fällen wird ein ähnlicher Algorithmus ausgeführt, welcher den zum untersuchten Knoten nächstgelegenden, passenden Knoten findet, um unnötige *Load*- oder *Store*-Instruktionen zu entfernen.

Zunächst wird der zu optimierende Knoten hinsichtlich Optimierbarkeit geprüft. So dürfen beispielsweise als *volatile* deklarierte Variablen nicht optimiert werden.

Im nächsten Schritt wird auf dem Firm-Graph ausgehend vom zu optimierenden Knoten eine Tiefensuche ausgeführt, die entweder an einem Knoten mit bestehender Alias-Relation terminiert, einen Knoten zur Optimierung erreicht oder den gesamten Graphen abläuft. Trifft die Tiefensuche auf einen Knoten mit Alias-Relation, kann nicht sichergestellt werden, dass die zu optimierende Speicherzelle in der Zwischenzeit nicht geändert wurde. Eine weitere Suche nach der letzten Instruktion, die auf denselben Speicher zugegriffen hat, ist in diesem Fall nicht mehr sinnvoll, da eine Optimierung mit diesem Knoten ein Programm mit anderer Semantik erzeugen könnte. Bricht die Tiefensuche durch vollständiges Ablaufen des Graphen ab, so ist kein Knoten vorhanden mit dem, unter den gegebenen Voraussetzungen optimiert werden kann.

Bisherige Behandlung von *Sync*-Knoten Die oben aufgeführte Funktionsweise basiert auf der totalen Ordnung der Instruktionen im Graph. Diese ist nötig, da zur Optimierung eine einfache Tiefensuche durchgeführt wird. Bei der Iteration des Graphen muss sichergestellt werden, dass sämtliche für eine Optimierung wichtige Knoten in der ursprünglichen Reihenfolge besucht werden.

Ein *Sync*-Knoten teilt den Speichergraph in mehrere Partitionen, welche jeweils Knoten mit Alias-Relation, Knoten zur Optimierung oder keinen relevanten Knoten enthalten können. Bei einer einfachen Tiefensuche wird zunächst eine Partition, anschließend der Rest des Graphen und erst am Schluss die anderen Partitionen traversiert. Das Problem tritt auf, wenn sich ein Knoten, der entweder durch eine Alias-Relation zu einem Abbruch der Optimierung führen würde oder Ziel der Optimierung sein könnte, in einer Partition befindet, die nicht als erstes betrachtet wird. In diesem Fall würde der Bereich vor dem partitionierten Bereich untersucht, bevor die relevante Partition untersucht wird. Wird außerhalb des partitionierten Bereichs eine Optimierungsmöglichkeit gefunden, so wird die mögliche Optimierung durchgeführt, obwohl zwischen den beiden Knoten ein weiterer Knoten liegt, welcher diese Operation verhindert hätte. Eine Korrektheit des Programms kann nach dieser Operation im Allgemeinen nicht mehr gewährleistet werden.

Wenn während einer *Load*-Optimierung beispielsweise eine Partition einen *Store*-Knoten mit Alias-Relation enthält, diese Partition aber nicht zufällig als erstes untersucht wird, und der Graph im weiteren Verlauf noch weitere Knoten für eine mögliche Optimierung enthält, so wird die Optimierung fehlerhafterweise nicht abgebrochen und der *Load*-Knoten infolgedessen mit einem veralteten Wert optimiert. Im Beispiellisting wird im Kompilat nicht gewährleistet, dass `a, b, c = 42` korrekt ausgeführt wurde, bevor der Vergleich durchgeführt wird. In Abbildung 3.1 ist das Problem als Graph dargestellt.

Im Beispiel wird Knoten *Load[Ls] 179* optimiert. Der rot markierte Pfad zeigt den

durch die Tiefensuche gewählten Pfad. Dieser führt am grünen Pfad vorbei zum Knoten *Load[Ls] 110* mit dem eine *Load-Load*-Optimierung ausgeführt wird. Der zuvor geladene Wert wird also am zu optimierenden *Load* wiederverwendet. Der grün markierte Pfad führt zu dem gewünschten Ergebnis. Innerhalb des durch den Knoten *Sync 251* partitionierten Bereiches, wird die Variable `&li` beschrieben. Der zuvor geladene Wert ist dementsprechend nach dem partitionierten Bereich nicht mehr korrekt. Der an Knoten *Store 143* gespeicherte Wert kann allerdings für eine *Load-after-Store* Optimierung verwendet werden.

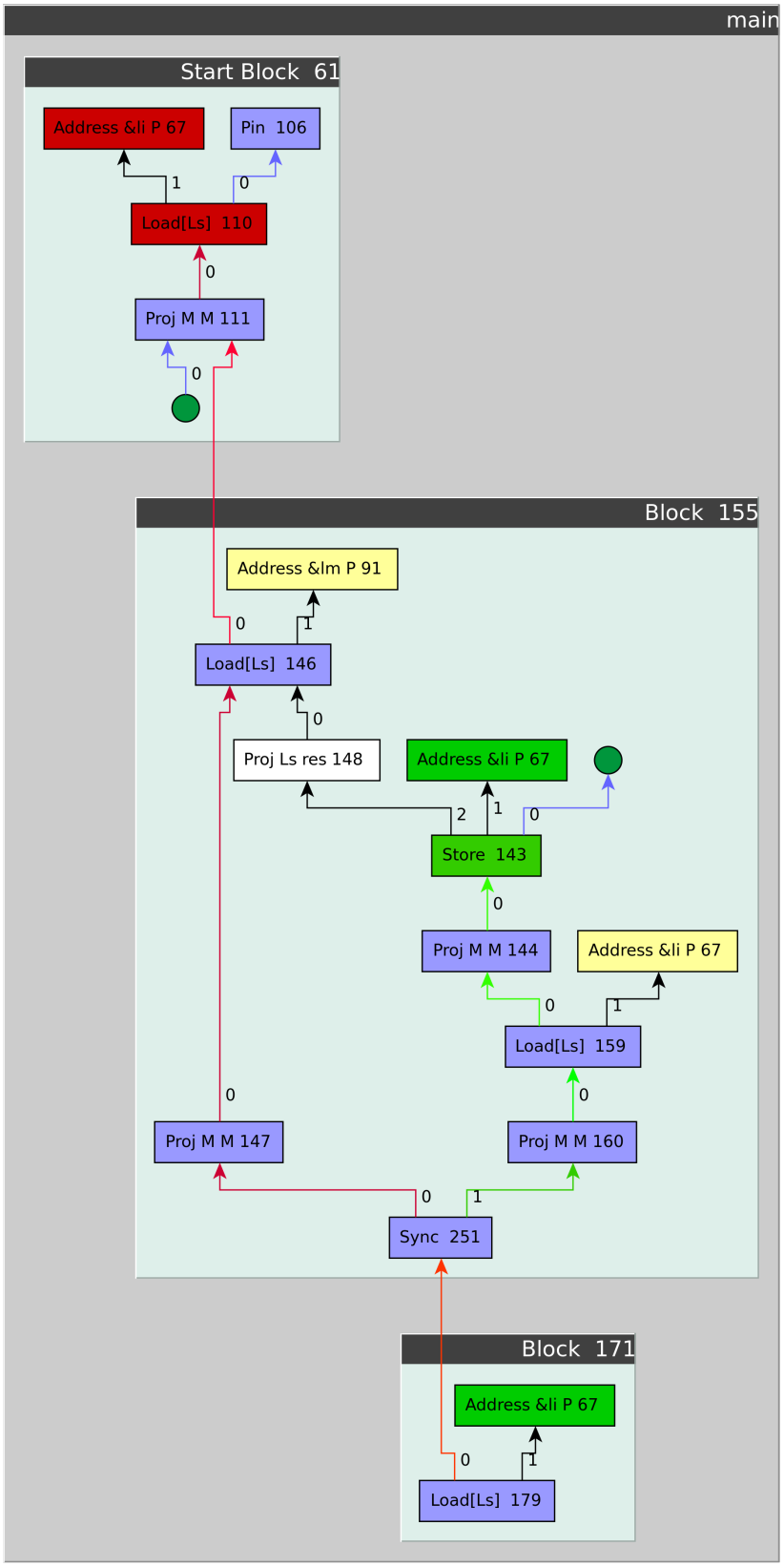

Abbildung 3.1.: Optimierung von *Load[Ls] 179* würde an *Load[Ls] 110* (rot) terminieren, anstatt *Store 143* (grün) zu erreichen.

Es ist ersichtlich, dass trotz Speicherpartitionierung die ursprüngliche Ordnung in ihrer Grundstruktur beibehalten werden muss. Das Überspringen von Knoten wie im Beispiel gezeigt muss verhindert werden, um eine fehlerfreie Optimierung mit *Sync*-Knoten zu gewährleisten.

Korrekte Optimierung mit *Sync*-Knoten

Die Betrachtungsreihenfolge der zu einem *Sync*-Knoten zugehörigen Partitionen ist aufgrund der Konstruktion der Partitionen für die Load-Store-Optimierung nicht relevant.

Folgende Fälle können auftreten:

Eine Partition beinhaltet einen zur Optimierung geeigneten Knoten und eine andere Partition enthält einen Knoten, der zu einem Abbruch der Optimierung führen würde: Da die Partitionen im Kompilat beliebig angeordnet sein dürfen, lässt sich die Partition, in der der Algorithmus abbricht, in einer totalen Ordnung vor der Partition ablegen, die den zu optimierenden Knoten enthält. Die Tiefensuche erreicht in dem Fall den Abbruch-Knoten nicht. Die Optimierung kann durchgeführt werden, wie in Abbildung 3.2 dargestellt.

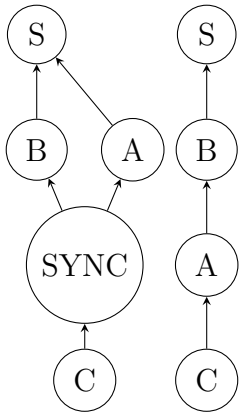

Abbildung 3.2.: DFS Test mit Knotengraden und innerer Sync Partition. A ist für eine Optimierung geeignet, B steht in Alias-Relation zu C.

Es existieren zwar Knoten mit Alias-Relation, aber es existiert kein erreichbarer, optimierbarer Knoten in den vorliegenden Partitionen: In diesem Fall muss der Algorithmus abbrechen. Es ist nicht möglich eine Anordnung der Partitionen zu finden, in der eine Optimierung vor dem Abbruch durchgeführt werden kann.

Innerhalb der Partitionen existiert kein Knoten, der die Optimierung beeinflusst: Der partitionierte Bereich kann in diesem Fall übersprungen werden. Auch hier lässt sich mit einer beliebigen Anordnung der Partitionen argumentieren - ohne Alias-Relation läuft die Tiefensuche in sequenzieller Anordnung über den betrachteten Bereich hinaus weiter.

Mehrere Partitionen enthalten Knoten, die für eine Optimierung geeignet wären: Dieser Fall tritt ein, falls Knoten aus mehreren Partitionen in einer Alias-Relation stehen. Durch Konstruktion der Partitionen lässt sich dieser Fall ausschließen. Zudem können die referenzierten Adressen aus verschiedenen Richtungen an die zu optimierende Adresse angrenzen. Es kann eine der Optimierungen ausgeführt werden, ohne dass der andere Zugriff beeinflusst wird. Die beiden Speicherbereiche grenzen anschließend aneinander an, überlappen aber nicht.

Mithilfe dieser Vorüberlegungen lässt sich ein Algorithmus konstruieren, mit dem *Sync*-Knoten in der Load-Store-Optimierung korrekt behandelt werden können. Kern der Überlegung ist es, innerhalb der Partitionen eine neue Abbruchbedingung für die Tiefensuche einzuführen. Diese verhindert, dass außerhalb des partitionierten Bereichs gesucht wird, obwohl die Gesamtstruktur der Partitionen noch nicht untersucht wurde. Zentral für die Funktionsweise eines solchen Algorithmus sind Informationen über die Struktur der *Sync*-Partitionen. Hier existiert auf der einen Seite der *Sync*-Knoten, welcher einen entsprechenden Bereich abgrenzt. Am anderen Ende des partitionierten Bereichs fehlt diese Information allerdings zunächst. Es ist somit nicht bekannt, über welche Knoten sich eine solche Partition erstreckt. Da Partitionen außerdem verschachtelt sein können, ist es nicht möglich, diese Informationen verlässlich anhand der Knotengrade herauszufinden.

Ist die Ausdehnung der Partitionen bekannt, so lässt sich das Problem jedoch lösen. Wenn ein neuer Knoten gefunden wurde und folgender Code wahr zurückgibt, darf der Knoten noch nicht behandelt werden. Wird ein *Sync*-Knoten besucht, so muss der Endknoten des *Sync*-Bereichs markiert werden indem `Knoten.Markiert = wahr` gesetzt wird.

Listing 3.2: Load-Store-Optimierung mit *Sync*-Knoten

```
// Vor dem nächsten DFS Schritt für "Knoten" ausführen:
FALLS Knoten.Ausgangsgrad > 1
UND Knoten.Markiert:
  für alle Nachfolger:
    falls !Nachfolger.besucht:
      result ← wahr
```

3.2.2. Bestimmen der Partitionsstruktur

Um den in Listing 3.2 dargestellten Algorithmus korrekt auszuführen, muss das Ende des partitionierten Bereichs bestimmt werden. Der *Sync*-Knoten kann als Beginn

verwendet werden. Methoden zum Finden des zu markierenden Endknoten werden im Folgenden betrachtet.

Knotengrade

Der nahe liegende Ansatz ist zunächst, die erreichbaren Knoten und sich daraus ergebene Knotengrade zu zählen. Ein Knoten darf in dieser Implementierung nur dann traversiert werden, wenn die Zahl der markierten Vorgänger mit der Zahl der erreichbaren Eingangskanten übereinstimmt. In diesem Fall ist sicher, dass jeder zum *Sync*-Knoten adjazente Pfad vollständig traversiert ist.

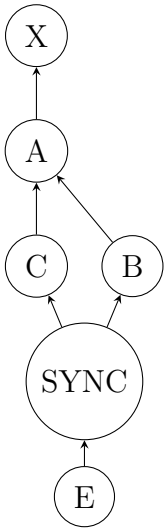

Abbildung 3.3.: DFS Test mit Knotengraden

Um diesen Ansatz zu implementieren, wird vor der Optimierung eine Tiefensuche durchgeführt und die erreichbaren Knoten samt Knotengrad gespeichert. Zur Prüfung konkreter Knoten wird dabei jeder besuchte Vorgänger gezählt. Stimmt die Zahl der besuchten Vorgänger mit dem zuvor bestimmten Grad überein, darf die Tiefensuche ausgehend vom aktuellen Knoten fortgesetzt werden.

Abbildung 3.3 zeigt einen entsprechenden Graphen. Die Knoten `B` und `C` liegen in unterschiedlichen Partitionen innerhalb des durch die Knoten `SYNC` und `A` eingeschlossenen Bereiches. Der Eingangsgrad von `A` ist 2, `A` darf daher nur betreten werden, wenn `B` und `C` bereits als besucht markiert wurden.

Dieses Vorgehen funktioniert allerdings nur, solange der zu untersuchende Knoten nicht innerhalb einer Partition liegt. Ist dies der Fall, können die anderen Partitionen durch Tiefensuche nicht erreicht werden und die Partition somit nicht verlassen werden. Die Semantik des Programms bleibt in diesem Fall bestehen, das Optimierungsergebnis wird allerdings im Allgemeinen schlechter. Abbildung 3.4 zeigt diese Problematik. Der Knoten `A` hat den Eingangsgrad drei, eine Suche ausgehend von `B`, `C`, `D`, `F`, `Z` oder `Y` kann `A` daher nie traversieren, da die mögliche Zahl der Pfade von einem dieser Knoten zu `A` höchstens zwei ist.

Befindet sich der Startknoten innerhalb einer Partition, so kann gefahrlos über den partitionierten Bereich hinaus gesucht werden, da die nicht erreichbaren Partitionen per Definition in einer *No-Alias*-Relation stehen.

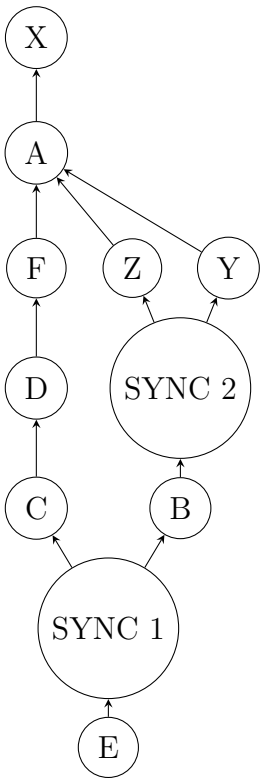

Abbildung 3.4.: DFS Test mit Knotengraden und innerer Sync Partition

Der Ansatz lässt sich anpassen, indem der eben erläuterte Erreichbarkeitstest jeweils ausgehend vom untersuchten Knoten gestartet wird. Der Endknoten der Partition hat bei Start innerhalb einer Partition Grad eins und könnte traversiert werden.

Eine solche Anpassung führt allerdings zu einer quadratischen Laufzeit, da pro Knoten eine Tiefensuche durchgeführt werden muss.

Dominanzanalyse

Zwischen zwei Knoten A, B und einem Ausgangsknoten R lässt sich eine Dominanzrelation definieren:

Definition 3.2.1. *A dominiert $B \Leftrightarrow$ sämtliche Pfade von R nach B führen über A.*

Insbesondere lässt sich der **direkte** Dominator definieren:

Definition 3.2.2. *A ist direkter Dominator von $B \Leftrightarrow A$ dominiert B und A dominiert keinen Knoten X, welcher auch B dominiert.*

Definition 3.2.3. *Analog lässt sich die (direkte) Postdominanz definieren: A post-dominiert B $\Leftrightarrow$ sämtliche Pfade von B nach R führen über A.*

Definition 3.2.4. *A (post-)dominiert B strikt $\Leftrightarrow$ A (post-)dominiert B und $A \neq B$*

In Abbildung 3.5 ist die Relation exemplarisch dargestellt. Der Startknoten ist X, der Endknoten Z.

- X dominiert direkt A. Außerdem dominiert X die Knoten M, N, B und Z.
- A dominiert direkt M, N, B und Z.

Im Folgenden wird der Graphen betrachtet, der entsteht, wenn ausschließlich Speicherkanten und dazu inzidente Knoten beibehalten werden. Dominanzrelationen beziehen sich auf die Pfade vom End- zum Startknoten.

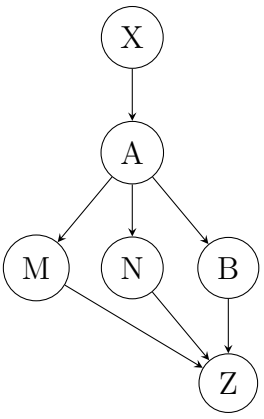

Abbildung 3.5.: Beispielgraph zur Dominanzrelation

Satz 3.2.1. *Ein Graph-Segment in welchem der Speicherfluss auf mehrere Graphpartitionen aufgeteilt ist, ist durch einen Sync-Knoten und dem zugehörigen direkten Postdominator minimal beschränkt. Es kann kein anderer Knoten gefunden werden, welcher die vorhandenen Partitionen mit dem Sync-Knoten einschränkt, wobei die Zahl der eingeschlossenen Knoten geringer ist.*

Beweis. durch Widerspruch. Nachfolgend wird der Knoten X betrachtet, der den partitionierten Bereich zusammen mit dem *Sync*-Knoten S beschränkt:

Fall 1: *X postdominiert den Sync-Knoten nicht.*

a) Falls $X \overset{+}{\nrightarrow} S$ gilt (vgl. Abbildung 3.6): X kann den partitionierten Bereich nicht beschränken.

b) Falls X Nachfolger von S ist, aber ein weiterer Knoten Y existiert, der Nachfolger von X und S ist und S postdominiert (vgl. Abbildung 3.7): Der Pfad zwischen S und Y, der nicht durch X verläuft, ist eine Partition, die außerhalb des durch X und S begrenzten Bereiches liegt. Damit ist X nicht der gesuchte Knoten.

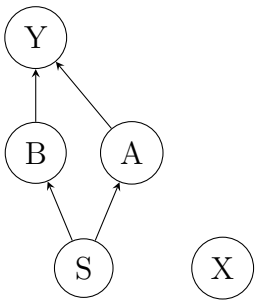

Abbildung 3.6.: X ist kein Nachfolger von S, postdominiert S also nicht. Y beschränkt den partitionierten Bereich und postdominiert S direkt.

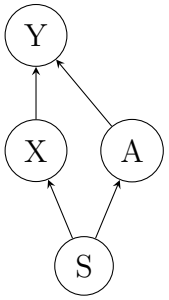

Abbildung 3.7.: X ist Nachfolger von S, postdominiert S aber nicht. Y beschränkt den partitionierten Bereich und postdominiert S direkt.

Fall 2: *X ist nicht **direkter** Postdominator von S.*
Es gibt einen Vorgänger Y von X der S postdominiert. Y beschränkt daher wie auch X sämtliche Partitionen. Der eingeschlossene Bereich ist aber kleiner und der ursprüngliche Bereich somit nicht minimal. X ist daher nicht der gesuchte Knoten wie in Abbildung 3.8 dargestellt.

□

Die Dominanzrelationen können in einem Dominanzbaum als transitive Reduktion der Dominanzrelation gespeichert werden. Der Dominanzbaum kann einmalig erzeugt werden und lässt dadurch Rückschlüsse über die Struktur aller im Graph vorhandenen Speicherpartitionen zu. Mithilfe des Algorithmus von Lengauer und Tarjan [6] kann der Dominanzbaum erzeugt werden.

Die in der Load-Store-Optimierung durchgeführte Änderungen am Graphen, können dazu führen, dass der Dominanzbaum die Dominanzrelationen im `Firm`-Graphen repräsentiert. Da fehlerhafte Postdominanzrelationen zu den zuvor genannten Fehlern führen können, muss der Dominanzbaum nach einer durchgeführten Graphänderung in der Load-Store-Optimierung korrigiert werden. In der für diese Bachelorarbeit entstandenen Implementierung wird zu diesem Zweck der Lengauer-Tarjan Algorithmus

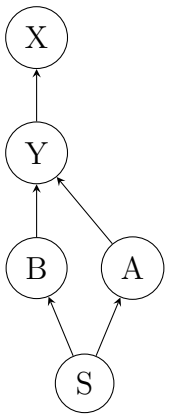

Abbildung 3.8.: X beschränkt den partitionierten Bereich zwischen S und Y nicht minimal. X ist daher nicht der gesuchte Knoten

nach erfolgter Änderung erneut aufgerufen. Die mehrfache Erzeugung des gesamten Dominanzbaums ist hierbei im Hinblick auf die Laufzeit nicht wünschenswert. Weitere Ansätze werden daher im Ausblick diskutiert.

3.2.3. Dominanzanalyse in parallelize-mem

Analog zu der zuvor beschriebenen Problematik in der Load-Store-Optimierung existiert eine fehlerhafte Behandlung von *Sync*-Knoten in *parallelize-mem*.

Diese Optimierung traversiert den Graphen ausgehend von einem Startknoten bis ein Knoten erreicht wird, welcher mit dem Startknoten in einer Alias-Relation steht. Sämtliche Knoten, die auf dem Graphen zwischen diesen liegen, können in separate Partitionen aufgeteilt werden. Ähnlich wie in der Load-Store-Optimierung wird hierbei eine Tiefensuche verwendet. Dabei werden alle aus einem *Sync*-Knoten ausgehenden Teilgraphen bis zur Abbruchbedingung traversiert. Enthält allerdings eine aus einem *Sync*-Knoten ausgehende Partition einen Knoten, der in Alias-Relation mit dem Startknoten steht, so muss diese Relation für den gesamten zwischen *Sync*-Knoten und direktem Postdominator eingeschlossenen Bereich angenommen werden.

Um das Problem zu beheben, wird in der neu entstanden Implementierung eine Prüfung vor der Traversierung eines *Sync*-Knotens ausgeführt. Dabei werden sämtliche aus dem *Sync*-Knoten ausgehende Kanten bis zum Postdominator verfolgt. Jeder auf dem Pfad liegende Knoten wird hinsichtlich seiner Alias-Relation untersucht. Wird der Postdominator über jede Kante erreicht, ohne dass auf einem der Pfade eine Alias-Relation gefunden wurde, so darf der Parallelisierungsalgorithmus fortgesetzt werden.

3.2.4. Lokale Optimierung

Durch die veränderte Ausführungsreihenfolge muss auch die lokale Optimierung korrekt mit partitioniertem Speicher umgehen. Obwohl die lokale Optimierung bereits nach *parallelize-mem* erneut ausgeführt wurde, war die Kompatibilität nicht vollständig gegeben. Die im Folgenden beschriebene Inkompatibilität trat allerdings nicht auf, da die problematischen Knoten zuvor in der Load-Store-Optimierung korrekt optimiert wurden. Eine Behandlung in der lokalen Optimierung war somit nicht mehr möglich. Erst durch die Ausführung der lokalen Optimierung zwischen *parallelize-mem* und der ersten Ausführung der Load-Store-Optimierung wird der vorliegende Fehler ausgelöst.

Load-Store-Optimierung in lokaler Optimierung

In der lokalen Optimierung gibt es eine einfache Form der Load-Store-Optimierung. Im Gegensatz zur *echten Load-Store*-Optimierung wird bei der lokalen Optimierung nur ein lokaler Teil des Graphen betrachtet. An dieser Stelle wird in der Optimierung immer nur der direkte Vorgänger betrachtet. Auf die Tiefensuche, die bei fehlender Alias-Relation in der Load-Store-Optimierung durchgeführt wird, wird hier verzichtet.

Der direkte Vorgänger eines *Load- / Store*-Knotens kann nun durch die Partitionierung ein *Sync*-Knoten sein, der bisher bei der Optimierung nicht behandelt wurde. Im Falle eines *Sync*-Knotens darf der jeweils erste Knoten jedes Pfades untersucht werden. Da beide Partitionen disjunkt sind, kann zwischen den ersten Knoten keine Alias-Relation existieren. Es kann also ausschließlich durch einen Test auf Knotenidentität festgestellt werden, ob eine Optimierung erlaubt ist oder nicht.

Ohne Durchführung einer Alias-Analyse dürfen die Pfade nicht weiter traversiert werden. Verwendet der erste Knoten nicht den gleichen Speicherbereich wie der zu optimierende Knoten, kann er dennoch ein möglicher Alias-Kandidat sein. Die vorherige Speicherpartitionierung bringt bei dieser Optimierung einen praktischen Vorteil. Wurde zuvor nur ein Knoten geprüft, so können nun auch mehrere Knoten untersucht werden, ohne die Laufzeit der lokalen Analyse signifikant zu erhöhen. Dadurch werden zuvor nicht optimierbare Fälle abgedeckt, bei denen Variablenzugriffe unterschiedlich geordnet sind.

In Listing 3.3 wird das Problem an einem Beispiel illustriert.

Listing 3.3: Beispielcode ordnungsabhängige Optimierung in der lokalen Load-Store-Optimierung

```
int a,b;
int main(void) {
    a = 0;
    b = 1000;
    while(a < 1) {
        a = 10;
        b = 10000;
    }
    return a + b;
}
```

Abhängig von der Reihenfolge der Initialisierung oder des Zugriffs ändert sich die Sortierung der Knoten im Graph. Hierbei sind die in Abbildung 3.9 und 3.10 dargestellten Varianten denkbar. Falls der direkte Vorgänger des *Loads* die Variable schreibt, kann eine Optimierung durchgeführt werden. Ansonsten lässt sich die Optimierung ohne Alias-Analyse nicht sicher fortsetzen. Die Optimierung terminiert entsprechend nach Vergleich der Adressen `&a` und `&b`. Nach einer Speicherpartitionierung befinden sich die *Store*-Knoten in verschiedenen Partitionen. Eine Optimierung ist somit in beiden Fällen möglich.

Obwohl diese Optimierung später während der Load-Store-Optimierung durchgeführt worden wäre, ergibt sich aufgrund der frühen Ausführung ein Vorteil für folgende Optimierungen. Es handelt sich hierbei um ein sogenanntes Phasenproblem, also ein je nach Ausführungsreihenfolge auftretender Fehler. So können teilweise Vergleiche und hierdurch ganze Codepfade aus dem Programm entfernt werden.

3.2.5. Sync-Sync Kanten

Durch das Ausführen bestimmter Optimierungen kann die Situation entstehen, dass zwei *Sync*-Knoten durch eine Kante direkt verbunden sind. So sorgt beispielsweise die *control-flow*-Optimierung dafür, dass mehrere Blöcke zusammengeführt werden. Falls beide Blöcke an entsprechender Stelle einen *Sync*-Knoten besitzen, werden in der Ausgabe der Optimierung zwei *Sync*-Knoten verbunden. Da für die Behandlung von *Sync*-Knoten häufig spezielle Implementierungen existieren, muss entweder sichergestellt sein, dass entsprechende Kanten korrekt behandelt werden können oder vor der Ausführung problematischer Optimierungen wieder entfernt werden.

Häufig geht damit einher, dass die aufeinander folgenden partitionierten Bereiche auf die gleichen Speicherbereiche zugreifen. Die so entstehenden Partitionen sind

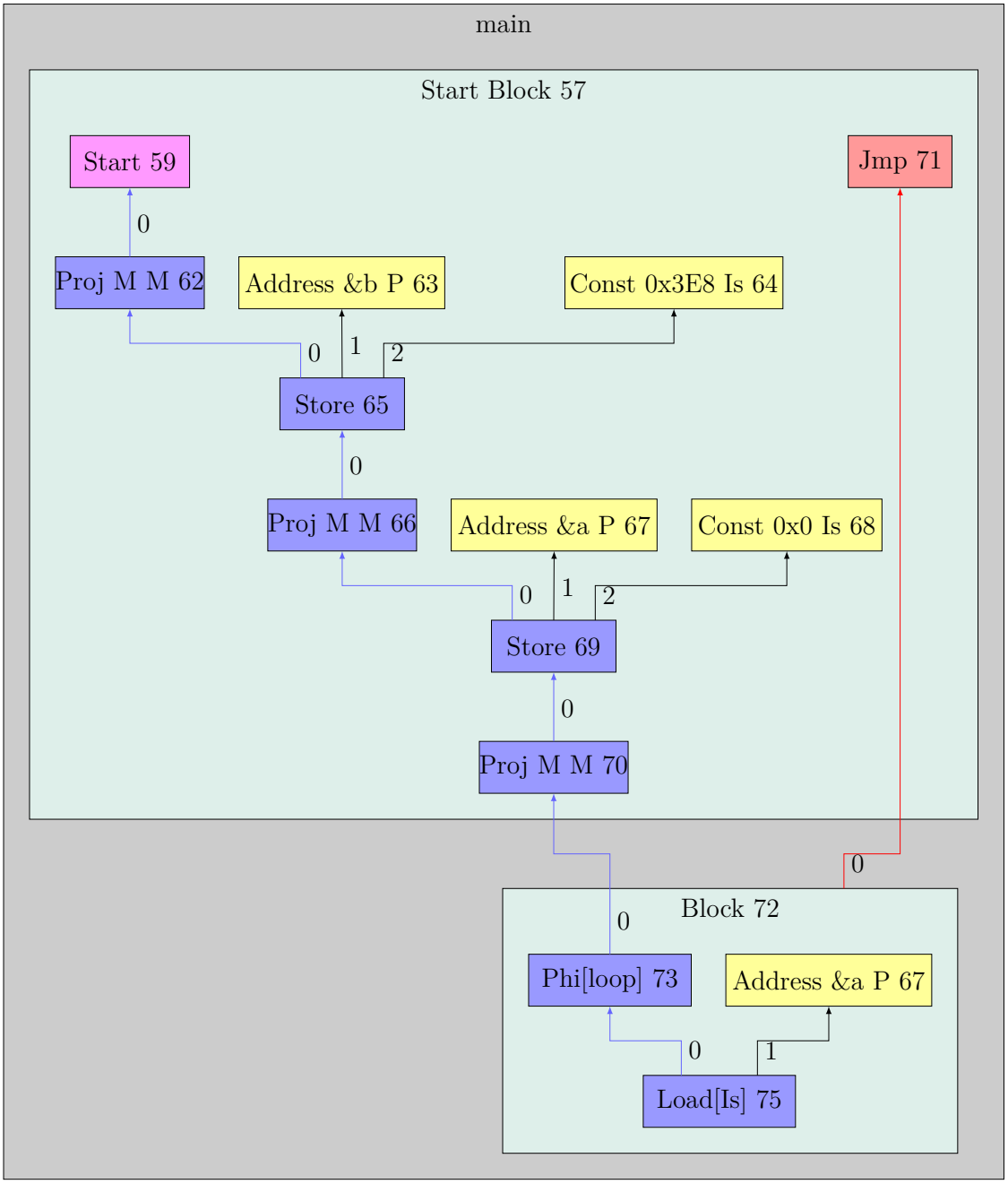

Abbildung 3.9.: Die lokale Optimierung kann bei Load 75 mit Store 69 optimieren.

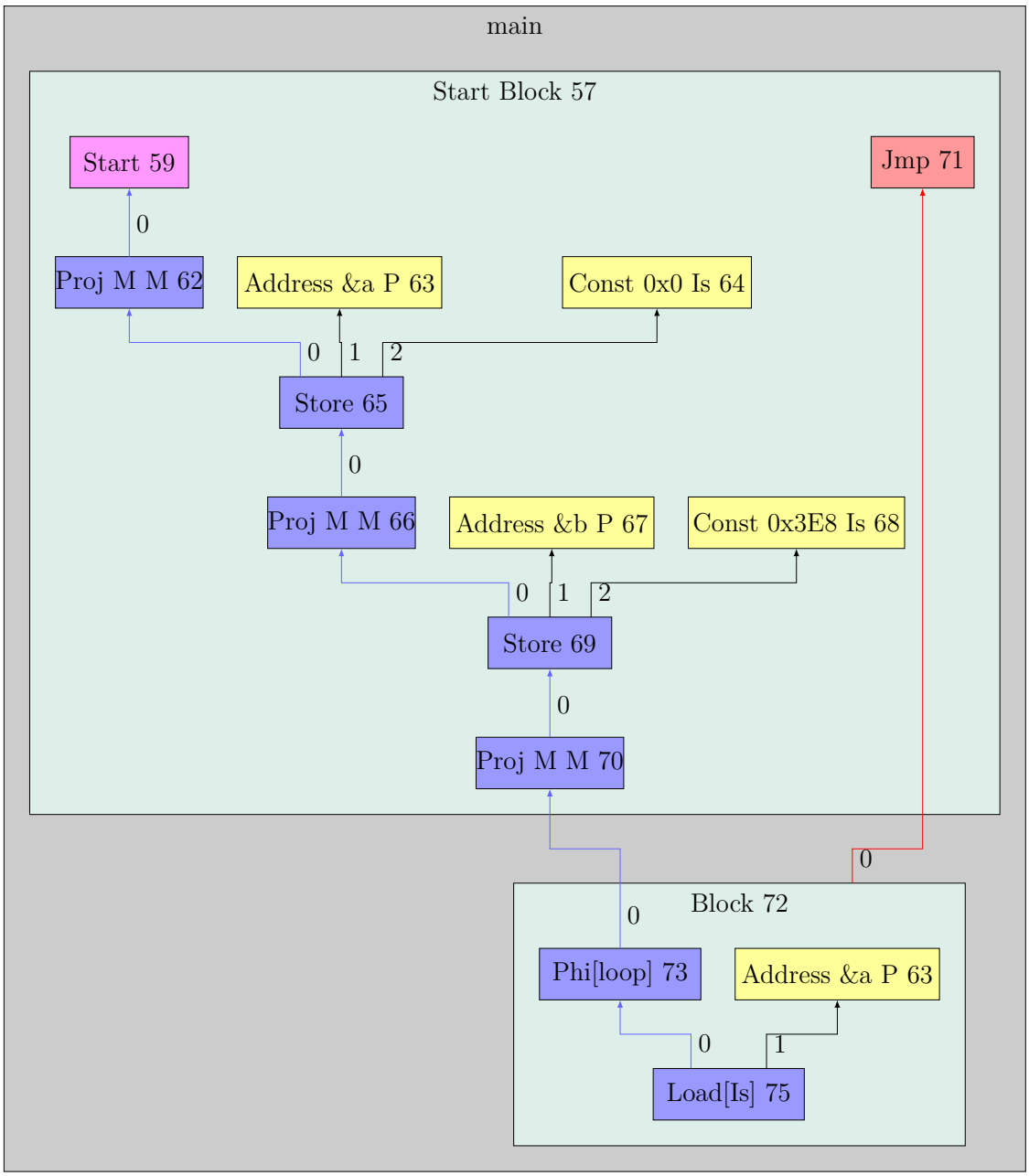

Abbildung 3.10.: Die lokale Optimierung kann Load 75 nicht mit Store 65 optimieren, da die Optimierung bereits bei Store 69 abbricht.

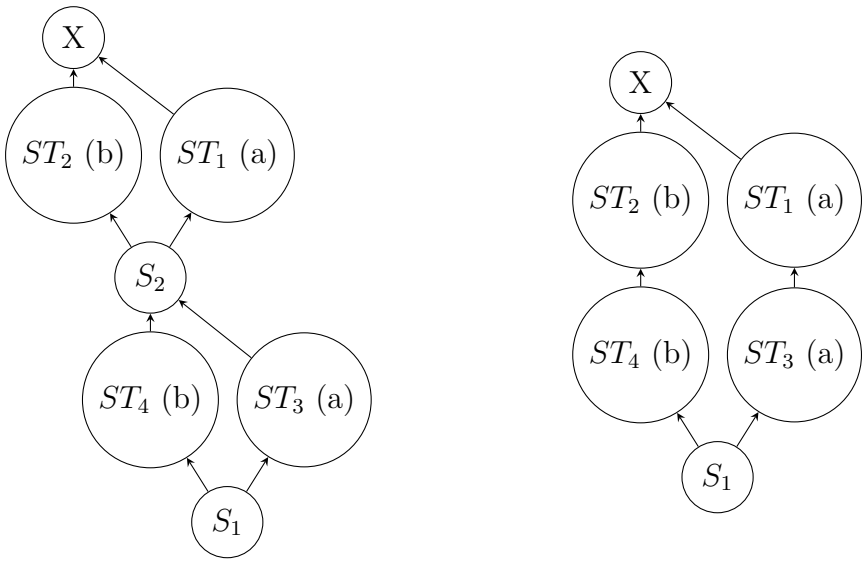

Abbildung 3.11.: Pfade aufeinanderfolgender Speicherpartitionen können vereinigt werden, wenn Speicherzugriffe einer Partition mit genau einer Partition der anderen Speicherpartition in einer Alias-Relation steht.

somit nicht maximal. Das bedeutet, dass vor oder nach dem partitionierten Bereich Knoten liegen, welche auch Teil einer Partition sein könnten. Betrachtet man in Abbildung 3.11 den ersten *Sync*-Knoten S_1, den zweiten *Sync*-Knoten S_2 sowie den ersten gemeinsamen strikten Postdominator X, so existiert auf genau einem Pfad zwischen S_2 und X ein Knoten, der mit einem Knoten auf einem beliebigen Pfad von S_1 nach S_2 in einer Alias-Relation steht. In diesem Fall können die beiden Pfade zusammengeführt werden. Existiert kein Knoten zwischen S_1 und S_2m der mit einem Knoten auf dem betrachteten Pfad in einer Alias-Relation steht (vgl. Abbildung 3.12), so kann der gesamte Pfad von $S_2 \rightarrow X$ nach $S_1 \rightarrow S_2$ verschoben werden. Anschließend kann die Kante zwischen S_1 und S_2 entfernt werden.

Eine entsprechende Umformung wird in *parallelize-mem* durchgeführt. Um nicht nach jedem Optimierungsschritt, welcher *Sync*-Knoten vereinigen könnte, eine vollständige Graphpartitionierung durchführen zu müssen, wurde die entsprechende Funktionalität in die neue Optimierung *eliminate-sync-edges* ausgelagert. Diese korrigiert ausschließlich aufeinander folgende Sync Partitionen. Eine durch Konkatenation entstandene nicht-maximal partitionierte Struktur kann somit durch die Ausführung dieser Optimierung vor kritischen Operationen repariert werden.

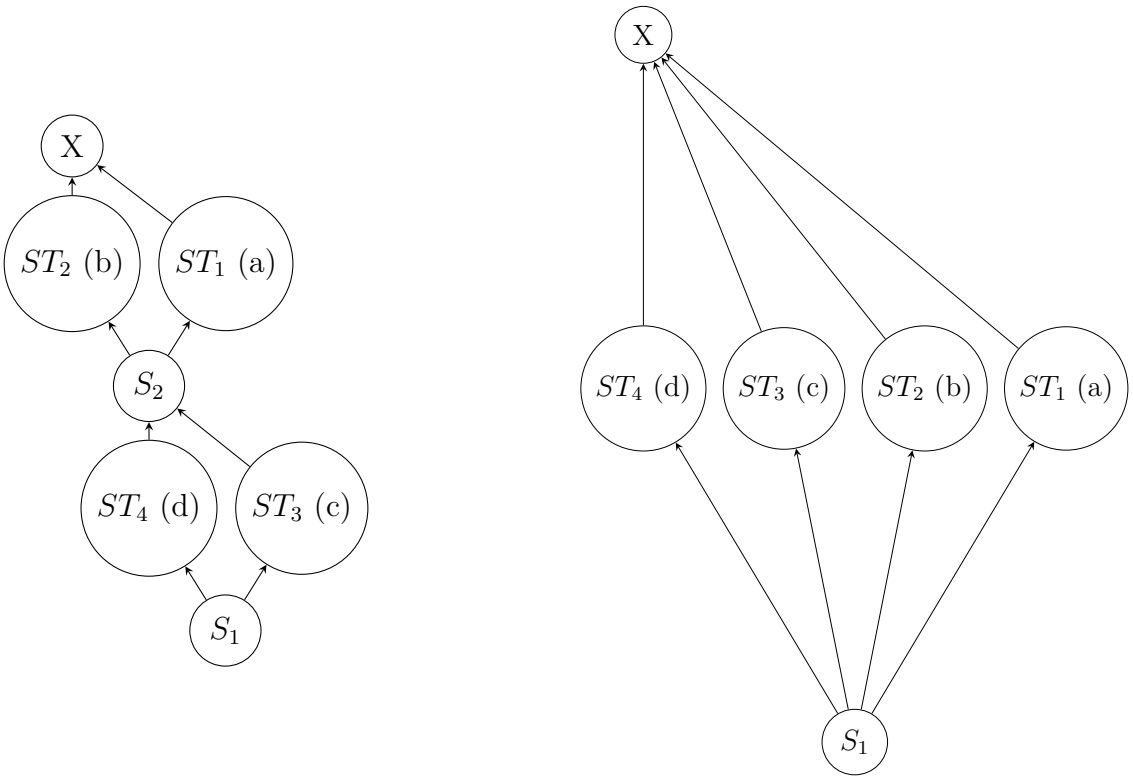

Abbildung 3.12.: Aufeinanderfolgende Speicherpartitionen können zusammengeführt werden, wenn die verwendeten Speicherzellen paarweise verschieden sind.

Lokale Sync-Sync Optimierung

Eine solche Optimierung ist die lokale Optimierung, in der aufeinander folgende *Sync*-Knoten zusammengefasst werden. In diesem Schritt wird allerdings davon ausgegangen, dass sämtliche Partitionen der beiden betroffenen *Sync*-Knoten disjunkt sind. Hierzu werden alle Kanten des zweiten *Sync*-Knotens an den ersten angehängt und der zweite *Sync*-Knoten wird anschließend entfernt. Stehen Pfade hierbei in einer Alias-Relation, so ist anschließend die Ausführungsreihenfolge der Speicherzugriffe nicht mehr sichergestellt. Gleichzeitig können andere Optimierungen, die *Sync*-Knoten traversieren, nicht mehr korrekt ausgeführt werden. Dies kann dazu führen, dass Folgefehler beispielsweise in der Load-Store-Optimierung auftreten (vgl. Abbildung 3.13).

Durch das Ausführen von *parallelize-mem* zwischen Load-Store-Optimierung oder *local*, nachdem zuvor control-flow ausgeführt wird, können die Probleme gelöst werden.

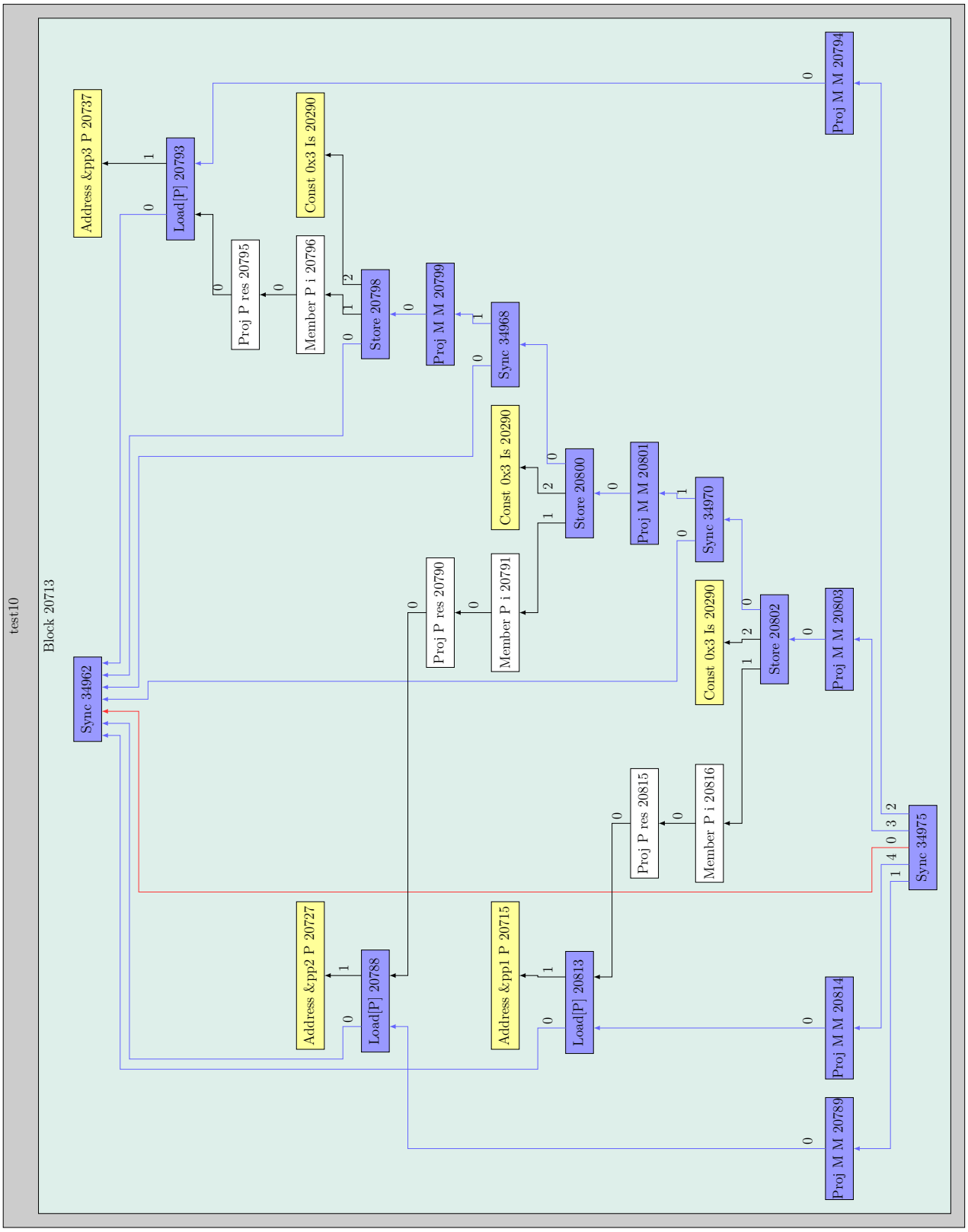

Abbildung 3.13.: Abkürzungen über leere Speicherpartitionen

4. Evaluation

Die frühzeitige Ausführung von *parallelize-mem* lässt sich durch die oben genannten Anpassungen durchführen. Die Semantik der erzeugten Kompilate bleibt in allen getesteten Fällen, einschließlich der Firm Testsuite, erhalten.

Es gilt nun noch die Performance der erzeugten Programme zu überprüfen. Zu diesem Zweck werden die C Benchmarks von SPEC CPU2000 [4] ausgeführt und die Ergebnisse miteinander verglichen. Sämtliche Benchmarks wurden mit den durch spec.py generierten Standardeinstellungen ausgeführt. Jeder Benchmark wurde insgesamt 120 mal ausgeführt.

Die gemessenen Ergebnisse sind in Tabelle 4.1 aufgeführt. Die Zeitangaben in den Spalten "Original" und "Neu" sind, basierend auf den Angaben der Bachelorarbeit "Besser Benchmarken" [7], die geometrischen Mittelwerte der Messergebnisse der jeweiligen Benchmark Suite. Die maximale Standardabweichung gibt die größere der beiden auf dem geometrischen Mittel basierenden Standardabweichungen an. Der Variationskoeffizient wird durch $|\frac{\text{Differenz}}{\text{Max Standardabweichung}}|$ berechnet.

Die Mehrzahl der Ergebnisse liegt demnach nah beieinander. Bei einigen Benchmark Suites wie "175.vpr", "197.parser", oder "254.gap" lohnt sich möglicherweise ein Blick auf die erzeugten Programme. Die Änderung ist hier am größten, sodass die Wahrscheinlichkeit eine Optimierung zu identifizieren, die mit der neuen Ausführungsreihenfolge nicht korrekt funktioniert am größten ist. Weiterhin ist interessant, dass einige Suites wie "253.perlbmk", "177.mesa" oder "179.art" deutlich schneller ausgeführt werden. Auch hier wäre eine Betrachtung der erzeugten Programme interessant, um mögliche GrÜnde für diese Effizienzsteigerung zu identifizieren.

Es lässt sich zusammenfassend feststelle, dass die Umstellung der Optimierungsreihenfolge technisch realisiert werden kann. Es konnten im Rahmen der Bachelorarbeit keine Probleme identifiziert werden, welche zu semantisch inkorrekten Kompilaten führen. Vor einer Integration sollte allerdings untersucht werden, aus welchem Grund einige Benchmarks bei geänderter Ausführungsreihenfolge langsamer sind. Möglicherweise führt eine Anpassung der hierbei identifizierten Optimierungsprobleme sogar zu einer verbesserten Performance wie beispielsweise in "253.perlbmk", "177.mesa" oder "179.art" beobachtet werden kann.

Name	Original (s)	Neu (s)	Differenz (s)	Änderung	Max Standardabweichung	Variationskoeffizient (Differenz)
164.gzip	70,53	70,31	0,21	-0,30%	0,0046	46,1
175.vpr	50,39	50,63	-0,24	0,48%	0,0035	69,5
176.gcc	20,94	20,98	-0,04	0,18%	0,0015	25,1
181.mcf	24,18	24,21	-0,03	0,14%	0,0057	5,8
186.crafty	27,47	27,49	-0,02	0,08%	0,0015	13,9
197.parser	62,96	63,36	-0,4	0,63%	0,0018	217,2
253.perlbmk	50,95	49,29	1,66	-3,36%	0,0019	891,6
254.gap	25,1	25,32	-0,22	0,88%	0,0063	35,2
255.vortex	37,62	37,4	0,22	-0,59%	0,0034	65,0
256.bzip2	49,34	49,43	-0,09	0,18%	0,0028	32,3
300.twolf	70,23	69,98	0,26	-0,37%	0,0035	73,4
177.mesa	49,2	48,36	0,84	-1,74%	0,0099	85,0
179.art	24,24	23,78	0,47	-1,96%	0,0405	11,5
183.equake	22,8	22,74	0,06	-0,28%	0,0224	2,8
188.ammp	87,26	86,46	0,8	-0,92%	0,0282	28,2

Tabelle 4.1.: Ausführungszeit von CPU2000 Benchmarks mit und ohne Anpassung der Optimierungsreihenfolge

5. Fazit und Ausblick

5.1. Kontextabhängige Anpassung des Dominanzbaums

Um eine korrekte Ausführung der Load-Store-Optimierung mit Speicherpartitionen sicherzustellen, werden Informationen zu Dominanzrelationen zwischen Speicherknoten im `Firm`-Graph benötigt. Operationen an der Graphstruktur ändern allerdings auch den Dominanzbaum, der im Zweifel komplett neu erstellt werden muss. In der Load-Store-Optimierung gelten allerdings Einschränkungen bezüglich der auf dem Graph durchgeführten Operationen, welche möglicherweise verwendet werden können, um den Dominanzbaum zu *reparieren*. Graphänderungen während der Optimierung nutzen die *exchange(X, Y)* Funktion, welche die ausgehenden Kanten eines Knotens Y auf einen Knoten X überträgt. In der Load-Store-Optimierung ist Y aufgrund der zum Auffinden dieses Knotens verwendeten Tiefensuche ein Nachfolger von X. Dadurch lässt sich möglicherweise eine Änderung der Postdominatoren aller Nachfolger von Y ausschließen. Außerdem lässt sich der Bereich, auf dem neue Dominanzrelationen gesucht werden müssen, anhand des alten Dominanzbaums einschränken. Ein erster Ansatz zur Implementierung eines Algorithmus, welcher diese Einschränkungen einbezieht, ist im Anhang zu finden.

5.2. Verwendung der Alias-Informationen

Die im Rahmen dieser Bachelorarbeit entwickelte Änderung der Optimierungsreihenfolge sorgt dafür, dass bereits frühzeitig Abhängigkeiten zwischen Speicherzugriffen ermittelt werden können. Hierdurch stehen Informationen zu Alias-Relationen zur Verfügung, die zuvor jeweils neu erzeugt werden mussten. Durch die vorgezogene Ausführung der *parallelize-mem* Operation lassen sich die Alias-Relationen früher und einfacher auch in anderen Optimierungsschritten verwenden.

Beispielsweise lassen sich unter bestimmten Bedingungen gesammelte Alias-Informationen für ganze Partitionen berechnen. In der Load-Store-Optimierung kann zunächst

geprüft werden, ob eine Partition von vornherein ausgeschlossen werden kann. Gilt das für alle Partitionen, so kann die Suche direkt beim Postdominator fortgesetzt werden. Hierdurch lassen sich Alias-Tests während der Tiefensuche vermeiden. Zu prüfen wäre allerdings, ob die Zahl der Fälle, in denen eine Aussage über die Alias-Relation zwischen einem Knoten und einer Menge von Knoten ausreichend ist, um die nötigen Vorberechnungen zu rechtfertigen.

5.3. Blockübergreifende Speicherpartitionierung

Aktuell werden Speicherpartitionen nur innerhalb von Blöcken erzeugt. Eine Speicherpartition kann sich nicht über mehrere Blöcke erstrecken, obwohl Blöcke in der Praxis mit disjunkten Speicherbereichen interagieren können. Durch das Aufheben dieser Grenze wäre es gegebenenfalls möglich, die Anordnung der Speicherzugriffe im Assembler zu optimieren. Ein testweises Aufheben der Beschränkung führt zwar zu keinen offensichtlichen Fehlern im erzeugten Graph, enthält allerdings viele verschachtelte Partitionen, die zumindest im Rahmen einer ersten Untersuchung gegenüber dem ursprünglichen Ergebnis keinen Vorteil brachten. Die entstandene Struktur schränkt jedoch die Lesbarkeit des Graphen ein. Nichtsdestotrotz ließe sich möglicherweise, beispielsweise mithilfe eines anderen Partitionierungsalgorithmus, ein gutes Optimierungsergebnis erzeugen.

Literaturverzeichnis

[1] G. Lindenmaier, "libFIRM – a library for compiler optimization research implementing FIRM," Tech. Rep. 2002-5, Sept. 2002.

[2] "C99 parser and frontend for libfirm." `https://github.com/libfirm/cparser`.

[3] "libfirm regression testsuite." `https://github.com/libfirm/firm-testsuite`.

[4] "Sprachtechnologie und compiler - transformation." `https://pp.ipd.kit.edu/lehre/SS2018/compiler/intern/04-Transformation.pdf`.

[5] 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.

[6] T. Lengauer and R. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, pp. 121–141, 01 1979.

[7] J. Bechberger, "Besser benchmarken." `http://pp.ipd.kit.edu/publication.php?id=bechberger16bachelorarbeit`, Apr. 2016.

Erklärung

Hiermit erkläre ich, Jonas Schwabe, 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

Danke

Ich danke Lilly und Karl-Heinz für mehrmaliges Korrekturlesen, sowie Sebastian Graf für die Betreuung der Arbeit.

A. Anhang

A.1. Kontextabhängige Dominanzrelationen reparieren

https://gitlab.jnugh.de/jonas/dom-fix

```
import networkx as nx
import matplotlib.pyplot as plt
from functools import reduce

def exchangeAndFix(graph: nx.DiGraph, domInfo: dict, a, b):
    assert isSuccessorOf(graph, b, a)
    predecessors = list(graph.predecessors(a))
    exchange(graph, a, b)
    #for predecessor in predecessors:
    fixDominance(graph, domInfo, None, b)
    removeUnreachable(graph, 'END')

def fixDominance(graph: nx.DiGraph, domInfo: dict, a, b):
    # find a common postdominator of END and b
    nodeA = 'END'
    nodeB = b
    markedNodes = {}
    markedNodes[nodeA] = True
    markedNodes[nodeB] = True
    commonPostDom = 'START'
    if nodeB == 'END':
        commonPostDom = 'END'
    else:
        while True:
            if nodeA != domInfo[nodeA]:
                nodeA = domInfo[nodeA]
                if nodeA in markedNodes:
                    commonPostDom = nodeA
                    break
            if nodeB != domInfo[nodeB]:
                nodeB = domInfo[nodeB]
                if nodeB in markedNodes:
                    commonPostDom = nodeB
                    break
            if nodeA == domInfo[nodeA]
            and nodeB == domInfo[nodeB]:
                break

    originalSuccessors = list(
        graph.successors(commonPostDom)
    )
    for successor in originalSuccessors:
        graph.remove_edge(commonPostDom, successor)

    # Search between common postdominator and END
    newIDoms = nx.immediate_dominators(
        nx.reverse(graph),
        commonPostDom
    )

    for node, postdom in newIDoms.items():
        if node != commonPostDom:
            domInfo[node] = postdom

    for successor in originalSuccessors:
        graph.add_edge(commonPostDom, successor)

# Tests is a successor of b
def isSuccessorOf(graph: nx.DiGraph, a, b):
```

```
    successors = reduce(
        lambda carry, x: carry + x,
        nx.dfs_successors(graph, b).values(),
        []
    )
    return a in successors

# Just like Firm does it
def exchange(graph: nx.DiGraph, a, b):
    predecessors = graph.predecessors(a)
    for predecessor in list(predecessors):
        graph.add_edge(predecessor, b)
        graph.remove_edge(predecessor, a)

# Just like firm does it
def removeUnreachable(graph: nx.DiGraph, start):
    reachable = reduce(
        lambda carry, x: carry + x,
        nx.dfs_successors(graph, start).values(),
        []
    ) + [start]
    for node in list(graph.nodes):
        if node not in reachable:
            graph.remove_node(node)
```

A.2. cparser-diff und weitere Debugging Tools

Die im Rahmen diese Bachelorarbeit entstanden Debug Tools welche nicht direkt mit dem Inhalt dieser Arbeit in Zusammenhang stehen sind unter: https://git-lab.jnugh.de/jonas/firm-tools zu finden.