Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

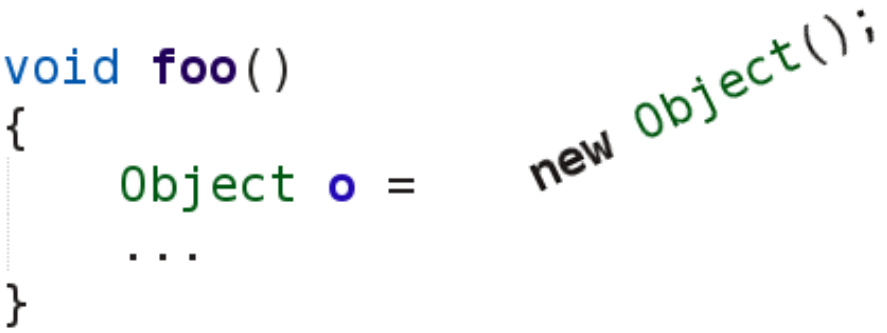
Lehrstuhl Prof. Dr.-Ing. Snelting

Stack-Allokation mittels Escape-Analyse

Bachelorarbeit von

Bernhard Scheirle

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. Jörg Henkel
Betreuender Mitarbeiter:	Dipl.-Inform. Manuel Mohr

Bearbeitungszeit: 1. Juli 2014 – 30. Oktober 2014

Zusammenfassung

Moderne objektorientierte Sprachen wie X10 und JAVA erstellen viele bzw. alle Objekte im Heap-Speicher. Allokationen im Heap-Speicher sind im Vergleich zu Allokationen im Stack-Speicher langsam.

Ziel der Escape-Analyse ist es, eine Aussage darüber zu treffen, ob nach vollständigem Abarbeiten einer Methode weiterhin Zugriffe auf Objekte, welche innerhalb dieser Methode angelegt wurden, möglich sind.

Das Analyseergebnis kann dazu verwendet werden, Heap-Allokationen in Stack-Allokationen umzuwandeln.

Modern object-oriented languages like X10 and JAVA create many objects on the heap. Allocations on the heap are slow compared to allocations on the stack.

The aim of the escape analysis is to determine if an object, which is created in a function, can still be accessed after the execution of this function.

The analysis result can be used to transform heap allocations into stack allocations.

Inhaltsverzeichnis

1. Einführung

Heap-Allokationen sind im Vergleich zu Stack-Allokationen teuer. Trotzdem erstellen moderne objektorientierte Sprachen wie Java und X10 viele Objekte im Heap-Speicher. Dieses Verhalten kann der Programmierer nur sehr schwer bis gar nicht beeinflussen.

In Programmausschnitt 1.1 werden Beispielsweise immer 1000 Heap-Allokationen durchgeführt, obwohl eventuell eine Stack-Allokation ausreichen würde. Um dieses Verhalten zu verbessern, ist eine Compiler-Analyse und Optimierung nötig.

Programmausschnitt 1.1: Objekt Erzeugung innerhalb einer Schleife

```
for( int i = 0; i < 1000; ++i )
{
  Object o = new Object();
  ...
}
```

Diese Arbeit befasst sich mit der Escape-Analyse und ihren Optimierungen, welche unter anderem Heap-Allokationen in Stack-Allokationen umwandeln können. In diesem Rahmen wurde eine Escape-Analyse sowie Optimierungen für Firm Implementiert.

Hierfür wurde eine praxisnahe Herangehensweise gewählt, insbesondere sollen X10-Schleifen optimiert werden.

Ein typisches Beispiel ist in Programmausschnitt 1.2 zu sehen. Es wird eine Region mit 100 Punkten angelegt und über diese wird iteriert. X10 erstellt nun in jeder Iteration ein neues Punkt-Objekt im Heap-Speicher.

Die implementierte Optimierung ersetzt diese Heap-Allokationen durch Stack-Allokationen und beschleunigt Programme um ∼10%.

Programmausschnitt 1.2: Gängige X10-Schleife

```
val region = (0..10)*(0..10);
for( p in region )
{  ...
}
```

2. Grundlagen und Verwandte Arbeiten

In diesem Kapitel werden einige grundlegende Begriffe und Konzepte vorgestellt. Zunächst wird in Abschnitt 2.1 die Escape-Analyse und die zugehörigen Optimierungen beschrieben. In Abschnitt 2.2 folgt die Beschreibung der Compilerzwischensprache Firm. Abschließend wird in Abschnitt 2.3 auf die Firmerweiterung liboo für objektorientierte Sprachen eingegangen.

2.1. Escape-Analyse

Bei der Escape-Analyse handelt es sich um eine statische fluss-insensitive Programmanalyse. Ihre Aufgabe ist es, Objekte zu verfolgen, welche während der Ausführung einer Methode angelegt werden. Diese Objekte wurden entweder direkt in der Methode angelegt, oder in einer von der Methode aufgerufenen Methode.

Ziel ist es, eine Aussage darüber zu treffen, ob nach vollständigem Abarbeiten einer Methode weiterhin Zugriffe auf Objekte, welche innerhalb dieser Methode angelegt wurden, möglich sind. Weitere Zugriffe sind nur dann möglich, wenn eine Referenz auf das Objekt die Methode verlässt und nach vollständigem Abarbeiten der Methode gültig bleibt. Objekte, auf welche weiterhin zugegriffen werden kann, werden von der Analyse als Escape markiert. Sollte hingegen keine gültige Referenz auf ein Objekt mehr existieren, wird dieses als NoEscape markiert.

Wie bei vielen Programmanalysen muss auch bei der Escape-Analyse konservativ approximiert werden, damit die Korrektheit garantiert werden kann. Bei einer zu konservativen Herangehensweise werden beispielsweise alle Objekte als Escape markiert, was eine spätere Optimierung unmöglich macht. Eine hingegen zu offene Herangehensweise würde zu falschen NoEscape-Markierungen führen. Diese würden durch die Optimierung zu einem fehlerhaften Programm führen.

```
void myFunction( Object p );
void escape_function_arg() {
  Object o = new Object();
  myFunction( o );
}
```

Programmausschnitt 2.1: Beispiel zur konservativen Approximation.

Beispiel konservative Approximation

Im Programmausschnitt 2.1 wird in der Methode *escape_function_arg* ein neues Objekt *o* angelegt und an die Methode *myFunction* übergeben. Ohne genauere Betrachtung von *myFunction* kann keine Aussage darüber getroffen werden, ob nach Abarbeiten von *escape_function_arg* weitere Zugriffe auf *o* möglich sind. Um die Korrektheit zu gewährleisten, muss also *o* als ESCAPE markiert werden. Durch diese Markierung ist aber eine Optimierung unmöglich.

Geschickter wäre es, sich den Parameter der *myFunction*-Methode genauer anzuschauen. Denn wenn dieser mit NOESCAPE markiert ist, existiert nach Abarbeiten von *myFunction* keine gültige Referenz mehr auf den Parameter. Daraus folgt, dass die gültigen Referenzen auf *o* sich vor und nach dem Aufruf von *myFunction* nicht unterscheiden. Somit könnte in diesem Fall *o* mit NOESCAPE markiert werden und eine Optimierung wäre an dieser Stelle möglich.

2.1.1. Stack-Allokationen

Die NOESCAPE-Markierung der Escape-Analyse gibt an, welche Objekte direkt auf dem Stack- anstatt Heap-Speicher angelegt werden können. Jedoch gilt folgendes zu beachten:

Objekte welche innerhalb einer Schleife angelegt werden, dürfen nur dann auf dem Stack alloziert werden, wenn die Objekte der verschiedenen Schleifeniterationen überschneidungsfreie Lebenszeiten haben. Dies ist nötig, da nur dann der Speicherbereich in den folgenden Iterationen wiederverwendet werden kann.

Manche Methoden erstellen ein Objekt und geben eine Referenz auf dieses zurück. In diesem Fall kann das Objekt nicht im Stack-Speicher der Methode, welche das Objekt erstellt, alloziert werden. Hingegen könnte aber Speicher im Stack der aufrufenden Methode reserviert werden und ein Zeiger auf diesen an eine spezialisierte Version der Objekt-erstellenden Methode übergeben werden. Die Initialisierung des Objekts würde in der spezialisierten Methode stattfinden.

2.1.2. Related Work

[11] beschreibt einen Escape-Analyse-Algorithmus, welcher auf Verbindungsgraphen arbeitet. Verbindungsgraphen stellen die Beziehungen zwischen Objekten und Referenz dar. Dies ermöglicht z.B. Aussagen über die einzelnen Felder der Objekte. Speziell wird zudem auch auf Java-Threads und Synchronisation eingegangen.

In [13] wird eine interprozedurale Escape-Analyse für objektorientierte Programme vorgestellt. Insbesondere werden Methoden, welche Objekte erzeugen und zurückgeben betrachtet.

Die Escape-Analyse in [8] arbeitet auf dem Java-Bytecode und analysiert besonders Zuweisungen genau. Hierfür wird ein spezielles Verfahren verwendet, dessen Korrektheit bewiesen wird.

In [18] wird ein Algorithmus vorgestellt, welche eine verbundene Points-To- und Escape-Analyse durchführt. Der Algorithmus arbeitet dabei auf Parallelen-Interaktions-Graphen, mit denen sich insbesondere Threads genau analysieren lassen.

Eine Partielle Escape-Analyse wird in [16] vorgestellt. Besonders ist, dass die Analyse fluss-sensitiv arbeitet und gegebenenfalls Optimierungen nur in Teilzweigen anwendet.

2.2. Firm

Ein Compiler besteht meist aus einem Front-End, welches den Quellcode liest und einem Back-End, welches den Maschinencode erzeugt. Zwischen Front- und Back-End wird das Programm durch eine Zwischendarstellung (Englisch: intermediate representation (IR)) beschrieben. Diese Zwischendarstellung ermöglicht das Kombinieren verschiedener Quell- und Zielsprachen, sowie das Schreiben von quellsprachunabhängigen Optimierungen.

Fiasco's Intermediate Representation Mesh (Firm) ist eine solche Graphen-basierte Zwischendarstellung, welche 1996 für den namensgebenden Sather-K-Compiler Fiasco entwickelt wurde. Firm baut auf der Arbeit von C. Click [12] auf, in welcher er eine einfache Graphen-basierte Zwischendarstellung beschreibt und wurde insbesondere von M. Trapp in [17] erweitert. In [9] ist eine aktuelle Beschreibung der Zwischendarstellung sowie deren Konstruktion beschrieben.

Ursprünglich wurde FIRM für imperative Sprachen entwickelt, mit Hilfe von LIBOO (Abschnitt 2.3) können aber auch objektorientierte Sprachen einfach abgebildet werden. FIRM basiert wie bei modernen Zwischendarstellung üblich auf der *Static-Single-Assignment Form* (SSA-Form). Die SSA-Form entstand aus der Arbeit von B. K. Rosen, M. N. Wegman und F. K. Zadeck [15].

Die SSA-Form schreibt vor, dass jede Variable genau eine Definition besitzt und verbietet das erneute Zuweisen einer Variable. Dies hat zur Folge, dass jede Variable immer genau einen Wert repräsentiert.

Ein Programm wird in SSA-Form gebracht, indem bei jeder Zuweisung eine neue Variable der Form *Ursprünglicher Variablenname* + *Index* eingeführt wird. Beim Zugriff auf eine Variable wird nun auf die passende indexierte Variable zugegriffen. Die passende indexierte Variable zu finden ist nicht immer statisch möglich. Zum Beispiel wird im Programmausschnitt 2.2 der Steuerfluss durch das *if* geteilt und erst zur Laufzeit ist erkennbar welche Zuweisung ausgeführt wird. Dieses Programm kann trotzdem in SSA-Form gebracht werden. Hierfür benötigt man die ϕ-Funktion, welche zur Laufzeit die passende Variable auswählt. Die Auswahl erfolgt durch den zuvor ausgeführten Block im Steuerflussdiagramm.

Die Funktion *foo* befindet sich in Programmausschnitt 2.2 nicht in SSA-Form, da die Variable *a* verschiedene Werte annimmt. In Programmausschnitt 2.3 befindet sie sich hingegen in SSA-Form, da die Variablen *a1* bis *a5* immer genau einen Wert annehmen.

Programmausschnitt 2.2:

```
void foo(bool input) {
  int a = 5; // a ist 5
  a = a + 3; // a ist 8
  if ( input ) {
    a = 10;
  } else {
    a = 0
  }
  print(a);
}
```

Abbildung 2.1.: Programm befindet sich nicht in SSA-Form.

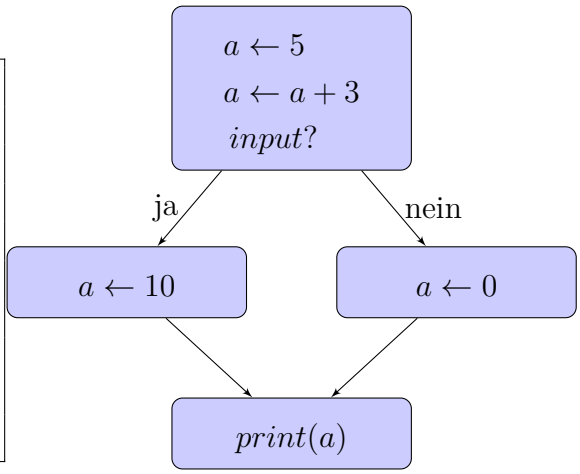
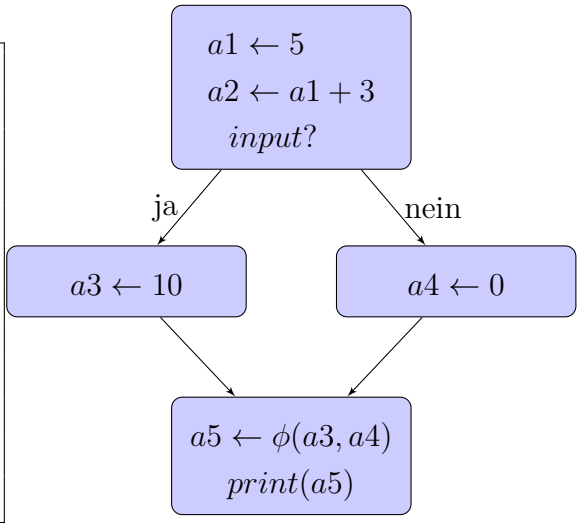


Abbildung 2.2.: Programm befindet sich in SSA-Form.

Programmausschnitt 2.3:

```
void foo(bool input) {
  int a1 = 5; // a1 ist 5
  int a2 = a1 + 3; // a2 ist 8
  int a3, a4;
  if ( input ) {
    a3 = 10;
  } else {
    a4 = 0;
  }
  int a5 = φ(a3,a4);
  print(a5);
}
```



Da durch die SSA-Form jede Variable genau einen Wert repräsentiert, hängen Operation in Firm nicht von einer Variablen ab, sondern von der Operation, die den Wert der Variablen erzeugt. Somit werden Variablen nicht mehr benötigt und es müssen nur Werte repräsentiert werden. Auf diese Weise werden Abhängigkeiten zwischen Operanden explizit dargestellt.

2.2.1. Firm-Graphen

Jede Funktion im Eingabeprogramm wird durch das Front-End in einen Firm-Graphen übersetzt. Firm-Graphen befinden sich immer in SSA-Form und sind explizite Abhängigkeitsgraphen, welche von M. Trapp [17] wie folgt definiert worden sind:

Definition. *Ein expliziter Abhängigkeitsgraph (EAG) ist ein gerichteter, markierter Graph. Die Knoten sind mit Funktionssymbolen aus Σ_{EAG} markiert. Knoten besitzen geordnete Eingänge und Ausgänge. Die Anzahl der Ein- und Ausgänge der Knoten und ihre Ordnung ist identisch mit der der Parameter und Ergebnisse des zugehörigen Terms aus Σ_{EAG}. Die Ein- und Ausgänge sind mit Typen aus T_{EAG} markiert. Kanten verbinden Ausgänge mit Eingängen desselben Typs.*
T_{EAG} und Σ_{EAG} sind in Tabelle A.1 und Tabelle A.2 definiert.

Jede Operation (Tabelle A.2) wird durch einen Knoten dargestellt. Die Kanten zwischen den Knoten repräsentieren Abhängigkeiten, z.B. Daten- oder Steuerflussabhängigkeiten. Jeder Knoten hängt von einem Grundblock ab, dies wird graphisch nicht durch eine Kante, sondern durch Ineinanderzeichnen realisiert.

Ein sehr wichtiger Knoten ist der *Proj*-Knoten, dieser extrahiert ein Wert aus einem Tupel. Da viele Operation in FIRM ihr Ergebnis als Tupel zurückgeben, werden für die weitere Verwendung *Proj*-Knoten benötigt. Im Folgenden werden *Proj*-Knoten durch *Proj a b* beschrieben, wobei *a* der Typ (siehe Tabelle A.1) des zu extrahierenden Wertes ist und *b* den zu extrahierenden Wert beschreibt.

In Abbildung 2.3 ist ein FIRM-Graph des Programmausschnitt 2.4 abgebildet. Wie in diesem Graph besitzt jeder FIRM-Graph einen *Start*- und *End*-Knoten, sowie die umschließenden gleichnamigen Grundblöcke. Die *Start*-Operation liefert einerseits den initialen Steuerfluss (*Proj X X_initial_exec*) und Speicherzustand (*Proj M M*), andererseits aber auch die Funktionsparameter (*Proj T T_args*) und den Funktionsstack (*Proj P P_frame_base*). Die einzelnen Funktionsparameter sind wiederum durch ein Tupel gebündelt und müssen mithilfe der *Proj*-Operation (*Proj Is Arg 0*) extrahiert werden. Im Funktionsstack, welcher in FIRM als ein Verbundstyp modelliert ist, befinden sich lokale Variablen der Funktion, diese können mit der *Member*-Operation (*Member local*) ausgewählt werden. Nicht verwendete Knoten, wie zum Beispiel der zweite Parameter der Main-Methode (*argv*), werden gelöscht.

```
typedef struct {
  int x;
  ...
} ComplexType;

int main(int argc, char **argv) {
  ComplexType local;
  local.x = argc;
  return local.x;
}
```

Programmausschnitt 2.4: Beispiel C-Programm zur Illustration eines Firm-Graphen.

Der mittlere Block repräsentiert in diesem Fall den Methodenrumpf. Zunächst wird mit *Member local* die sich auf dem Stack befindende lokale Variable *local* ausgewählt. Da in das Element *x* von *local* geschrieben werden soll, muss dieses wiederum zuerst mit *Member x* ausgewählt werden. Nun kann man mit der *Store*-Operation den nullten Parameter, also *argc*, in *x* speichern. Weiterhin soll *local.x* zurückgeben werden. Da *local.x* jedoch immer den gleichen Wert wie der nullte Parameter beinhaltet, geben wir mit *Return* direkt den Wert des Parameters zurück. Zuletzt wird die Methode mit dem *End*-Knoten abgeschlossen.

2.2.2. Speicherzustand

In vielen Quellsprachen können beim Methodenaufruf Seiteneffekte auftreten, deshalb ist die Reihenfolge, in der Methoden ausgeführt werden, enorm wichtig. Da FIRM aber nur Daten- und Steuerflussabhängigkeiten darstellt, könnten Methoden, welche auf verschiedene Daten arbeiten, in der Reihenfolge vertauscht werden. Um die richtige Reihenfolge zu gewährleisten, werden einige Operationen, insbesondere Methodenaufrufe, künstlich in Abhängigkeit gebracht. Die Abhängigkeit wird durch einen neu eingeführten SSA-Wert, den Speicherzustand, ermöglicht. Dieser wird zusätzlich an die Operation übergeben, welche wiederum als Ausgabe einen neuen Speicherzustand generiert.

2.2.3. libFirm

LIBFIRM [3] ist eine Open-Source-Implementierung der FIRM Zwischensprache in C. Für LIBFIRM gibt es Front-Ends für unter anderem Java-Bytecode [1], C [2] und X10 [6]. Auch ein Back-End für die IA-32-Architektur ist bereits in LIBFIRM integriert. Eine ältere Einführung in LIBFIRM befindet sich in [14].

2.3. liboo

LIBOO [4] ist eine Bibliothek, welche FIRM erlaubt, objektorientierte Konstrukte darzustellen. Hierfür fügt sie den in Tabelle A.2 beschriebenen Knoten vier weitere hinzu (siehe Tabelle 2.1). Zudem ermöglicht LIBOO die Darstellung der Klassenhierarchie und fügt Methoden zusätzliche Informationen hinzu. LIBOO ermöglicht somit z.B. Abfragen, ob eine Methode *abstrakt* oder *final* ist, von einer Oberklasse ererbt wurde oder wie Klassen in Beziehungen zueinander stehen.

Tabelle 2.1.: LIBOO Funktionssymbole

Name	Typ	Beschreibung
Arraylength	$B \times M \times P \to I$	Gibt die Größe eines Arrays an.
InstanceOf	$B \times M \times P \to b$	Prüft, ob ein Objekt eine Instanz einer bestimmten Klasse ist.
MethodSel	$B \times M \times P \to P$	Gibt das konkrete Ziel eines dynamisch gebundenen Methodenaufrufs an.
VptrIsSet	$B \times M \times P \to P$	Garantiert, dass der Vptr für das Objekt gesetzt ist.

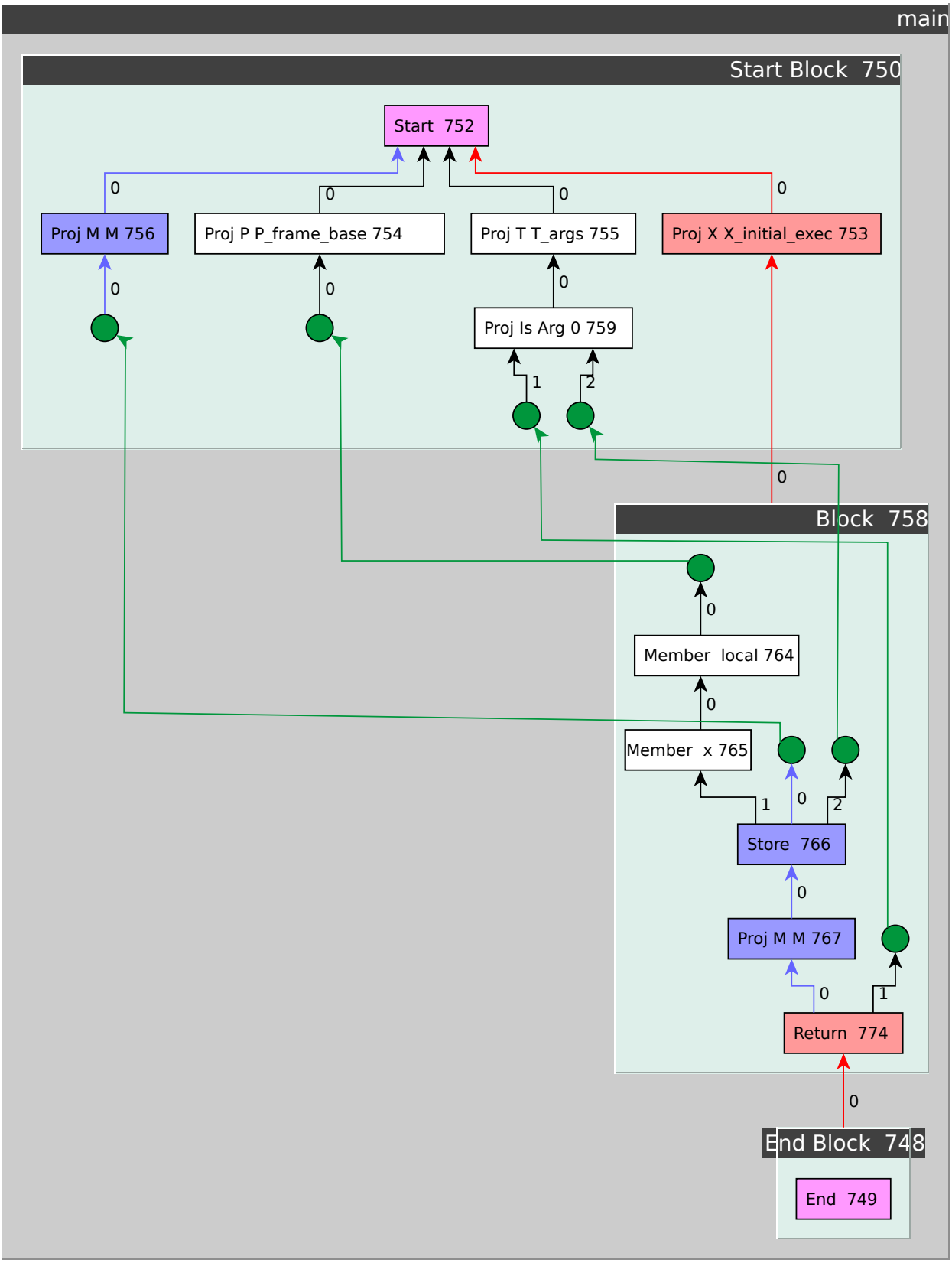


Abbildung 2.3.: Firm-Graph der Main-Funktion aus Programmausschnitt 2.4.

3. Entwurf und Implementierung

Ziel dieser Arbeit ist es, die Escape-Analyse und die dazugehörigen Optimierungen zu libFirm hinzuzufügen. Wie üblich wurde die Analyse und Optimierung in verschiedene Phasen aufgeteilt. In Abschnitt 3.1 wird die Analysephase genauer beleuchtet und insbesondere auf die verschiedenen Subanalysen eingegangen. Die Optimierungsphase wird in Abschnitt 3.2 erläutert.

Die Analysephase und Optimierungsphase werden für jede Methode im Eingabeprogramm nacheinander ausgeführt.

3.1. Analysephase

Die wichtigste Aufgabe der Analysephase ist es, Heap-Allokationen zu erkennen und zu überprüfen, ob diese in Stack-Allokationen umgewandelt werden dürfen. Der Firm-Knoten, welcher das neu angelegte Objekt repräsentiert, wird entsprechend des Analyseergebnisses markiert. Gültige Markierungen sind in Tabelle 3.1 angegeben. Hierbei ist zu beachten, dass sobald ein Objekt markiert ist, es nur noch Markierungen höheren Ranges (weiter oben in der Tabelle) annehmen kann.

Tabelle 3.1.: Gültige Markieren für zu analysierende Objekte.

Markierung	Beschreibung
Escape	Nach Abarbeiten der Methode sind weiter Zugriffe auf das Objekt möglich.
EscapeReturn	Wie Escape, jedoch nur weil es mittels *return* zurückgegeben wird.
NoEscape	Nach Abarbeiten der Methode sind keine weiteren Zugriffe auf das Objekt möglich.
Unknown	Das Objekt wurde noch nicht analysiert.

3.1.1. Fresh-Analyse

Wie in Unterabschnitt 2.1.1 beschrieben, gibt es Methoden, welche auf jedem Pfad im Steuerflussgraph ein Objekt erzeugen und Referenzen auf diese zurückgeben. Die Fresh-Analyse überprüft jede Methode, ob sie dieses Verhalten aufweist. Ist dies der Fall, wird die Methode *Fresh*-Methode genannt. Im Falle einer *Fresh*-Methode, gibt die Analyse zusätzlich noch die Größe des größten zurückgegebenen Objektes zurück.

Die Fresh-Analyse gestaltet sich sehr einfach, sie iteriert über alle *Return*-Knoten des Firm-Graphen und überprüft, ob die zurückgegebenen Objekte den gleichen Typ haben und mit EscapeReturn markiert sind.

3.1.2. Heap-Allokationen erkennen

Um überhaupt Heap-Allokationen analysieren zu können, müssen diese zunächst auch als solche erkannt werden.

In Programmausschnitt 3.1 ist eine Funktion abgebildet, welche mittels *malloc* eine Heap-Allokation durchführt und dabei acht Byte Heap-Speicher reserviert. Zuletzt wird noch eine Eins in den Speicher geschrieben. Abbildung 3.2 zeigt den dazugehörigen Firm-Graphen, in welchem der Funktionsaufruf der *malloc*-Funktion deutlich zu erkennen ist. Abbildung 3.1 hebt alle für die Analyse wichtigen Knoten einer Heap-Allokation nochmals hervor.

```
int main() {
  int* a = malloc(8);
  *a = 1;
  return 0;
}
```

Programmausschnitt 3.1: C-Programm, welches eine Heap-Allokation mittels *malloc* durchführt.

Um alle Heap-Allokationen eines Graphen zu finden, läuft die Analyse einmal über den gesamten Graph und überprüft bei jedem *Proj P*-Knoten, ob er sich in einem Konstrukt wie in Abbildung 3.1 befindet, dabei darf der *malloc* Aufruf auch durch ein Aufruf einer *Fresh*-Methode ersetzt werden. Ist dies der Fall, wird der *Proj P*-Knoten mit Unknown markiert und mit der Größe des zu allozierenden Speichers in eine Hashmap gespeichert. Die Größe ergibt sich aus dem nullten Parameter des *Call*-Knoten oder im Falle einer *Fresh*-Methode aus dem berechneten Wert. Anschließend wird die eigentliche Escape-Analyse ausgeführt.

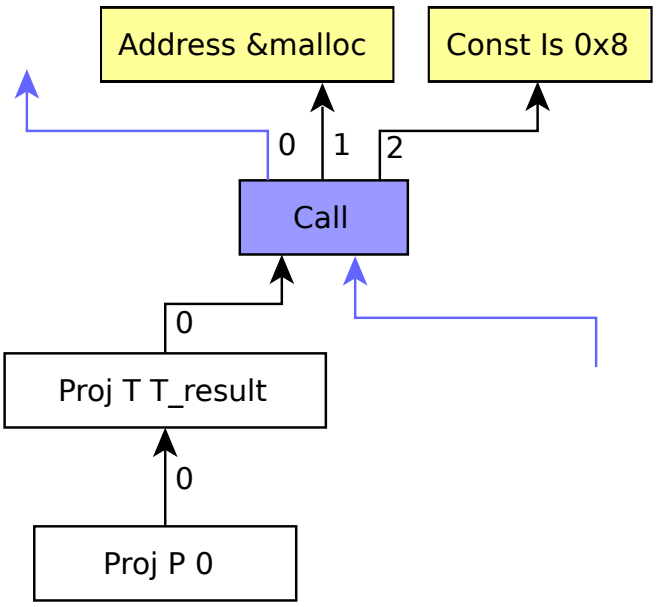


Abbildung 3.1.: Heap-Allokation in FIRM

Um die Optimierung einfach zu halten, werden nur Heap-Allokationen mit konstanter Größe weiterverfolgt. Das Einschränken auf Heap-Allokationen konstanter Größe ist bei objektorientierten Sprachen keine allzu starke Einschränkung, denn alle Objekte haben eine konstante Größe, ausgenommen Arrays dynamischer Größe. Genauer thematisiert wird dies in Unterabschnitt 3.2.1.

Flexiblere Erkennung

Das Erkennen von Heap-Allokationen wie zuvor beschrieben ist sehr starr und setzt voraus, dass die *malloc*-Funktion eine bestimmte Methodensignatur hat, damit die Größe des zu allozierenden Speichers ausgelesen werden kann. Durch diese Voraussetzung hängt die Analyse von der Quellsprache ab.

Ein flexiblerer Ansatz wäre mittels Callback ins Front-End möglich. Dabei würde im Front-End ein Callback definiert werden, welcher zu einer gegebenen Methode zurück gibt, ob es sich um eine Methode handelt, welche eine Heap-Allokation durchführt. In diesem Falle, würde zusätzlich noch die Größe des zu allozierenden Speichers zurückgegeben. Der Callback würde somit eine komplett quellsprachenunabhängige Analyse ermöglichen.

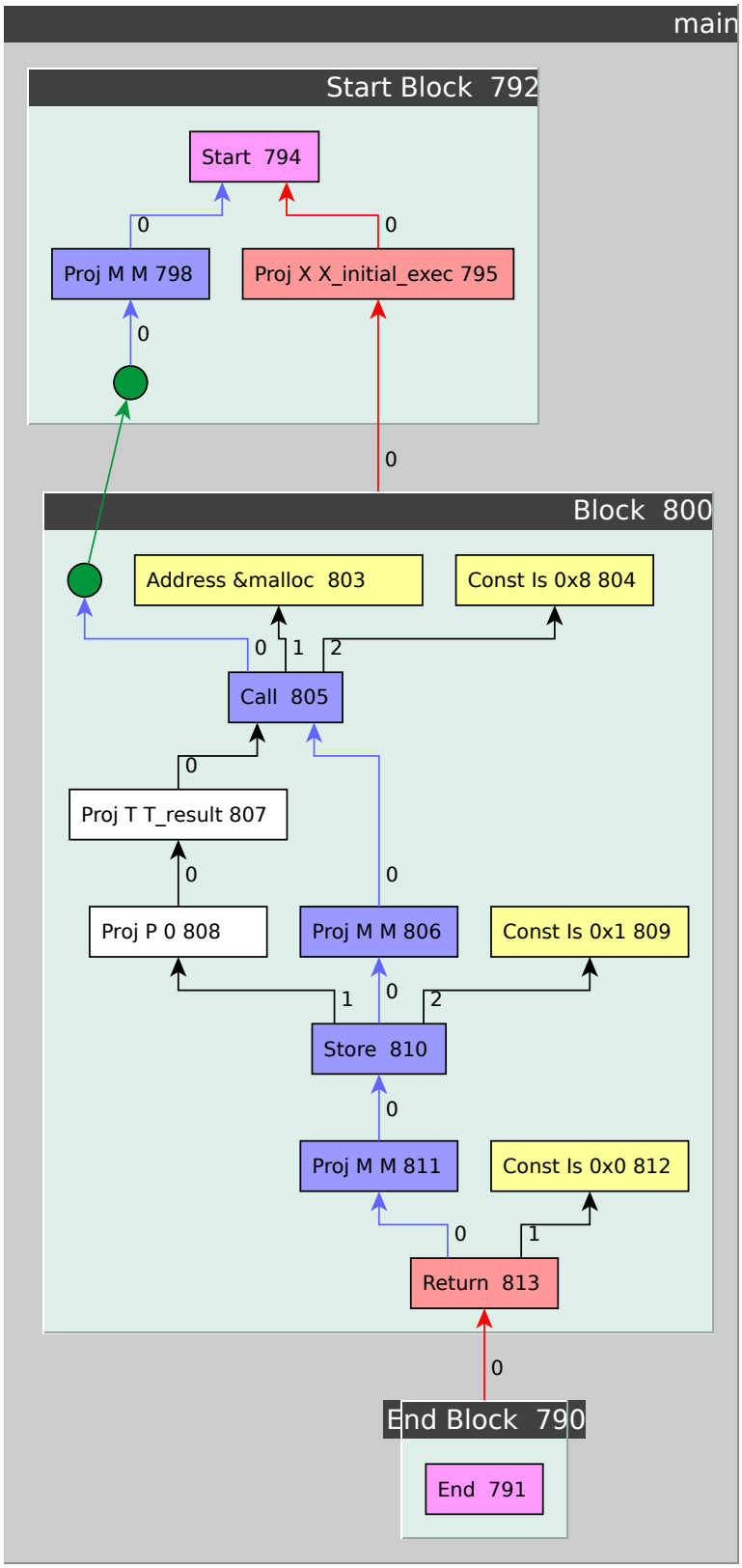


Abbildung 3.2.: Firm-Graph der Main-Funktion aus Programmausschnitt 3.1.

3.1.3. Escape-Analyse

Die Escape-Analyse selbst schaut sich alle Verwender des *Proj P*-Knotens rekursiv an. Für jeden Verwender wird entschieden, ob die Markierung (Tabelle 3.1) des Objekts aktualisiert werden muss und ob dessen Verwender zusätzlich auch noch überprüft werden müssen.

Ist die Analyse für einen Verwender nicht explizit implementiert, wird das Objekt mit Escape markiert. In einigen Fällen kann das Objekt aber auch direkt markiert werden. Sollte zum Beispiel der Verwender ein *Return*-Knoten sein, wird das Objekt direkt mit EscapeReturn markiert.

Die Verwender, welche den größten Analyseaufwand erfordern werden im Folgenden genauer betrachtet.

3.1.4. Escape-Analyse: Load-Knoten

Load-Knoten laden Daten aus einem Objekt. Auch dies kann die Markierung des Objektes verändern. Wenn ein Pointer aus einem Objekt geladen wird und dieser nicht mit NoEscape markiert ist, darf das Objekt auch nicht mit NoEscape markiert werden. Sollte das Objekt nämlich mit NoEscape markiert werden und dadurch im Stack-Speicher alloziert werden, könnte der Destructor des Objektes, die Daten auf die der geladene Pointer zeigt, löschen, obwohl diese noch in Verwendung sind.

Deshalb muss im Falle eines geladenen Pointers dieser selbst wieder analysiert werden. Die daraus folgende Markierung überträgt sich auf das Objekt.

3.1.5. Escape-Analyse: Store-Knoten

Store-Knoten speichern einen Wert an eine beliebige Adresse. Sollte es sich bei dem Wert um ein Objekt handeln, darf dieses nur dann mit NoEscape markiert werden, wenn garantiert werden kann, dass die Adresse nach Abarbeiten der Methode nicht mehr gültig ist.

Diese Analyse wählt eine striktere Bedingung für die NoEscape-Markierung. Es wird überprüft, ob die Adresse, welche durch Firm-Knoten repräsentiert wird, *lokal* ist. Eine Adresse ist *lokal*, wenn sie in den Stack-Frame der Methode zeigt. Hierfür werden alle Knoten, welche zur Adressberechnung gehören, rekursiv abgelaufen und auf *Lokalität* geprüft. Sollte wiederum ein Knoten nicht explizit implementiert sein, wird angenommen,

dass die Adresse nicht *lokal* ist. Die *Proj*- und *Phi*-Knoten werden genauer betrachtet.

Im Falle eines *Proj*-Knotens muss dessen Vorgänger *lokal* sein um das Speichern in nicht-*lokale* Objekte zu vermeiden. Zusätzlich muss jedoch der *Proj*-Knoten selbst mit NoEscape markiert sein, damit das zu speichernde Objekt nicht an anderer Stelle die Methode verlässt. Zur Berechnung dieser Markierung wird die Escape-Analyse für diesen *Proj*-Knoten extra ausgeführt.

Im Falle eines *Phi*-Knotens müssen auch alle Vorgänger *lokal* sein. Besonders muss jedoch darauf geachtet werden, dass *Phi*-Knoten Vorgänger von sich selbst sein können wenn sie im Rumpf einer Schleife stehen.

Schleifen

Eine weitere Einschränkung gilt für *Store*-Knoten, welche sich innerhalb von Schleifen befinden. Wie in Unterabschnitt 2.1.1 erläutert, dürfen Objekte in Schleifen nur dann optimiert werden, wenn sich ihre Lebenszeiten nicht überschneiden. Da dies sehr aufwendig zu überprüfen ist, wird an dieser Stelle sehr konservativ approximiert. Es wird angenommen, dass sich die Lebenszeiten von Objekten überschneiden, sobald diese den Schleifenkörper verlassen. Dementsprechend markiert die Analyse das Objekt mit Escape, wenn sich die Adressknoten nicht in derselben oder inneren Schleifen wie der *Store*-Knoten befinden. Um dies zu überprüfen, wird der Schleifenbaum[7] verwendet.

Programmausschnitt 3.2 zeigt die Grenzen dieser Approximation. *b* könnte in diesem Fall mit NoEscape markiert werden, wird aber hingegen mit Escape markiert, da in der letzten Iteration *b* den Schleifenkörper verlässt, indem es in *a* gespeichert wird.

```
int main(int argc) {
  int *a[1];
  for( int i = 0; i <= argc; i++) {
    int* b = malloc(8);
    if ( i == argc )
      a[0] = b;
  }
}
```

Programmausschnitt 3.2: Grenzen der Store-Knoten-Analyse

3.1.6. Escape-Analyse: Call-Knoten

Call-Knoten repräsentieren Methodenaufrufe und sind nur dann Verwender eines Objektes, wenn sie entweder einen aus dem Objekt geladenen Funktionspointer aufrufen oder das Objekt als Argument entgegen nehmen.

Im Folgenden wird der Fall betrachtet, bei dem das Objekt als Parameter übergeben wird. Ein einfacher Ansatz würde das Objekt mit ESCAPE markieren, da keine Aussage über das Verhalten der aufgerufen Methode gemacht werden kann. Dieser Ansatz führt jedoch bei objektorientierten Sprachen zu sehr wenig bis keinen NOESCAPE-Markierungen.

Parameter-Analyse

Wie in Abschnitt 2.1 beschrieben, hilft es, die Parameter der Methode vorab zu analysieren. Die Parameter-Analyse führt für jeden Pointer-Parameter die Escape-Analyse aus. Die Markierungen der einzelnen Parameter wird für jede Methode in einer Hashmap zwischengespeichert.

Wenn die aufzurufende Methode statisch gebunden ist, kann die zwischengespeicherte Markierung des entsprechenden Parameters direkt auf das Objekt übertragen werden.

Dynamische Bindung

Eine weitere Herausforderung sind dynamisch gebundene Methodenaufrufe. Da die aufzurufende Methode erst zur Laufzeit ermittelt wird, ist die Parameter-Analyse nur begrenzt anwendbar.

Um trotzdem eine Aussage in diesen Fällen treffen zu können, müssen alle möglichen Aufrufziele analysiert werden. Hierzu wird die Klassenhierarchie einmal abgelaufen und nach passenden Methoden gesucht. Damit alle Methoden gefunden werden können, muss das gesamte Programm vorliegen. Durch eine *Whole-World*-Annahme wird dies garantiert. Nur wenn alle Aufrufziele den Parameter mit NOESCAPE markiert haben, übernimmt auch das Objekt diese Markierung.

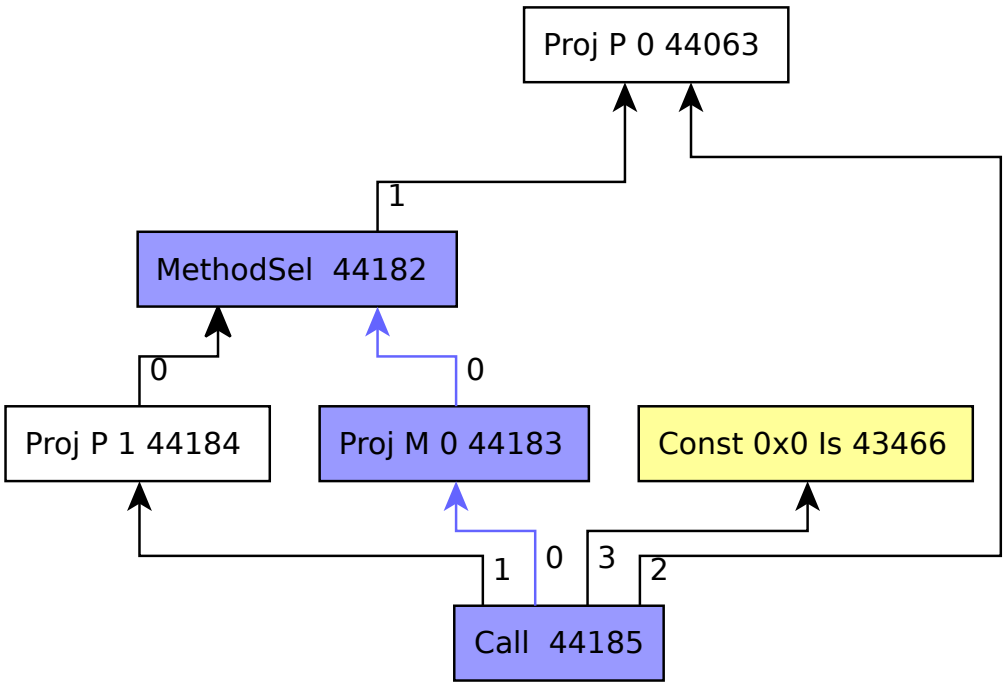


Abbildung 3.3.: FIRM-Graph des Programmausschnitt 3.3. *Proj P 0 44063* ist der Punkt *p*, welcher einerseits die aufzurufende Methode beschreibt (*MethodSel*), andererseits als *this*-Pointer an die Methode übergeben wird. Der nullte Parameter (der *this*-Pointer) aller passenden Indexmethoden, muss mit NOESCAPE markiert sein, damit dieser Punkt ebenfalls mit NOESCAPE markiert werden darf.

Programmausschnitt 3.3 und Abbildung 3.3 zeigen anhand des Indexoperators von Punkten in X10, dynamisch gebundene Aufrufe.

```
var p:Point{rank==2} = ...;
p(0); //Zugriff auf das erste Element des Punktes p
```

Programmausschnitt 3.3: In X10 ist der Indexoperator für Punkte dynamisch gebunden.

3.2. Optimierungsphase

In der Optimierungsphase wird eine einzelne Methode entsprechend der Ergebnisse der Analysephase optimiert. Hierbei wird unterschieden zwischen der einfachen Stack-Allokation (Unterabschnitt 3.2.1) und der etwas aufwendigeren Stack-Allokation bei *Fresh*-Methoden (Unterabschnitt 3.2.2).

3.2.1. Stack-Allokation

Die einfache Stack-Allokation konzentriert sich nur auf die NoEscape-Markierungen der Analyse und ersetzt alle Zugriffe, auf Objekt mit dieser Markierung, durch Stack-Zugriffe. Die dazugehörigen Heap-Allokationen werden entfernt und es wird statisch Platz auf dem Stack reserviert.

Die statische Stack-Allokation hat den Vorteil, dass der Speicher nicht manuell wieder freigegeben werden muss. Sie ist jedoch nur möglich, da schon die Analyse Heap-Allokationen dynamischer Größe nicht analysiert. Bei einer dynamischen Allokation müsste der Speicher auch wieder manuell freigegeben werden. Die Herausforderung beim Freigegeben von Speicher ist, den richtigen Punkt im Programm zu finden. Dies ist insbesondere innerhalb von Schleifen aufwendig.

In Abbildung 3.4 ist der optimierte Firm-Graph von Programmausschnitt 3.1 abgebildet (vgl. nicht optimierter Graph Abbildung 3.2). Auffallend und wie zu erwarten wurde die Heap-Allokation komplett entfernt und Abhängigkeiten auf den *Proj P*-Knoten durch Abhängigkeiten auf einen neun *Member*-Knoten ersetzt. Dieser *Member*-Knoten repräsentiert das sich nun auf dem Stack befindende Objekt.

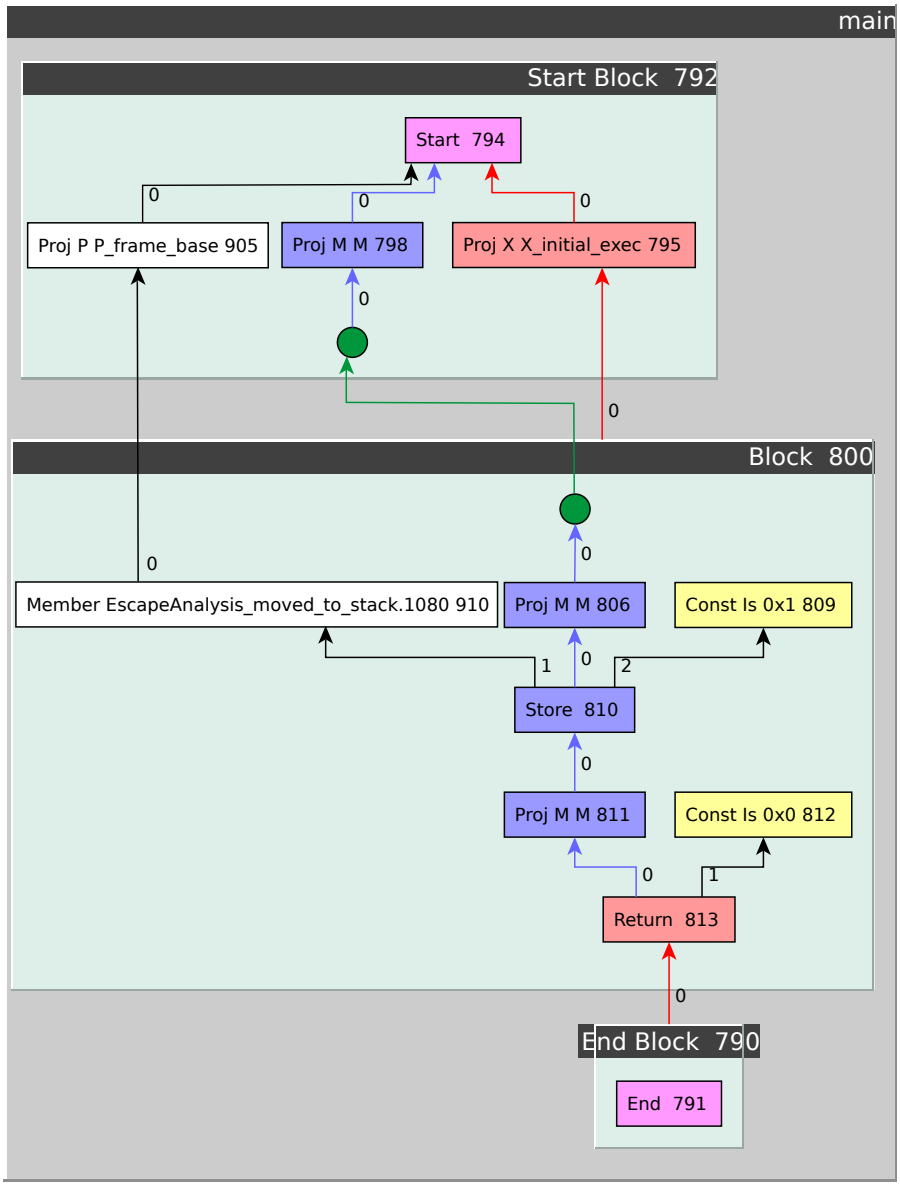


Abbildung 3.4.: Optimierter Firm-Graph der Funktion aus Programmausschnitt 3.1.

3.2.2. Fresh-Methoden

Das Ziel bei *Fresh*-Methoden ist, dass nach der Optimierung das Objekt im Stack-Speicher des Aufrufers liegt.

Um dies zu erreichen wird die *Fresh*-Methode durch eine spezialisierte Methode ersetzt. Die spezialisierte Methode ist zu beginn eine Kopie der *Fresh*-Methode. Sie wird aber um einen Parameter erweitert. Zudem werden die Heap-Allokationen entfernt und alle Zugriffe auf die *Proj P*-Knoten, welche das Objekt repräsentieren, werden durch Zugriffe auf den hinzuzufügen Parameter ersetzt. In der aufrufenden Methode wird Stack-Speicher reserviert und der Aufruf an die originale *Fresh*-Methode wird durch einen Aufruf an die spezialisierte Methode ersetzt, welche zusätzlich einen Zeiger auf den reservierten Stack-Speicher entgegen nimmt.

In Abbildung 3.5 ist der Teilgraph vor der Optimierung eins Aufrufes an eine *Fresh*-Methode abgebildet. Abbildung 3.6 zeigt den selben Teilgraph nach der Optimierung.

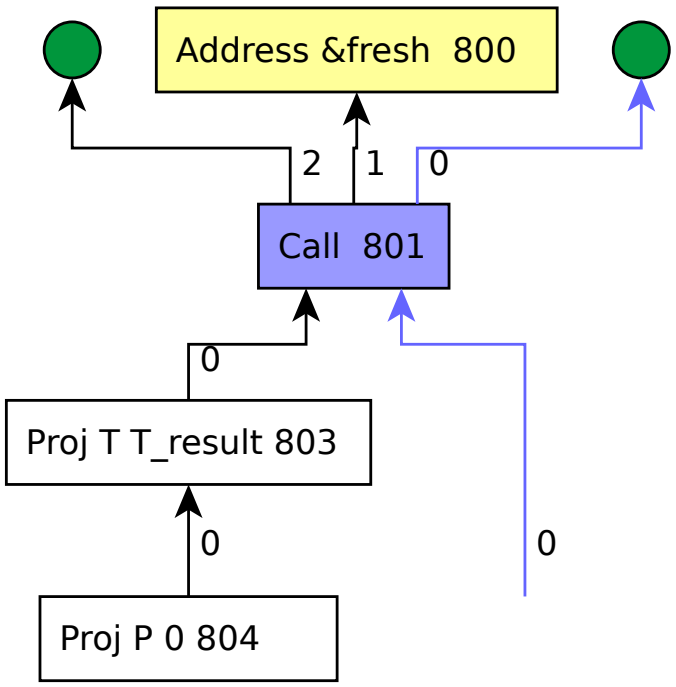


Abbildung 3.5.: Teilgraph vor der Optimierung eines *Fresh*-Methoden Aufrufes.

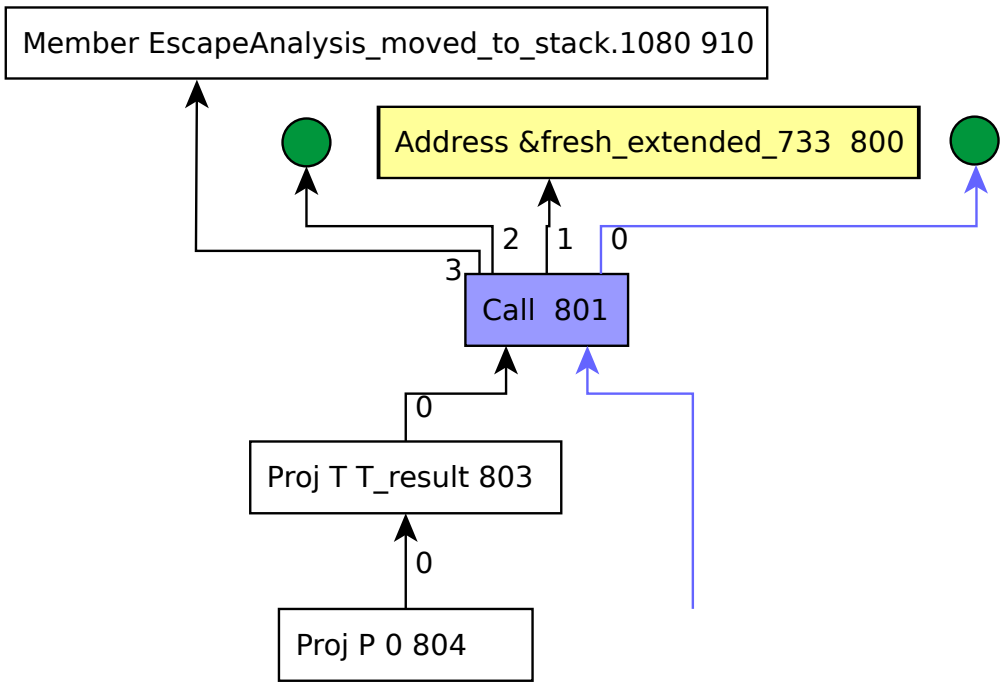


Abbildung 3.6.: Teilgraph nach der Optimierung eines *Fresh*-Methoden Aufrufes.

4. Datenflussanalyse

Datenflussanalysen [5] sind statische Programmanalysen, die gewöhnlich auf den Grundblöcken B_i des Steuerflussgraphen arbeiten.

Definition. *Ein Verband* $(L, \leq, \sqcap, \sqcup)$ *ist eine Halbordnung, für die gilt, dass für je zwei Elemente* $x, y \in L$ *stets* $x \sqcap y$ *und* $x \sqcup y$ *existieren. Ferner muss ein kleinstes* ($\bot$) *und größtes* ($\top$) *Element bezüglich* $\sqcap$ *und* $\sqcup$ *existieren.*

In Verbänden gilt: $x \leq y \iff x \sqcup y = y \iff x \sqcap y = x$.

Der Effekt jedes Grundblocks B_i wird durch eine Transferfunktion $f_{B_i} : L \to L$ beschrieben. Der aktuelle Ein- und Ausgangsstatus jedes Grundblocks wird durch $in : B \to L$ und $out : B \to L$ beschrieben. Alle Nachfolgerblöcke bzw. Vorgängerblöcke eines Blockes im Steuerflussgraph werden durch $succ : B \to 2^B$ bzw. $pred : B \to 2^B$ beschrieben.

Im Falle der Escape-Analyse wird die Datenflussanalyse als Rückwärtsanalyse implementiert und

$$L = \{\bot, NoEscape, EscapeReturn, Escape, \top\}, mit$$
$$\bot \leq NoEscape \leq EscapeReturn \leq Escape \leq \top$$

$$in(B_i) = f_{B_i}(out(B_i))$$
$$out(B_i) = \bigsqcup_{B_j \in succ(B_i)} in(B_j)$$

gewählt.

Der Initialwert jedes Grundblocks ist NoEscape bzw. für *Return*-Knoten EscapeReturn. Die Transferfunktion f_{B_i} muss für jeden Firm-Knoten implementiert werden. Für nicht explizit angegebene Transferfunktionen wird $f_{B_e} = Escape$ gewählt. Die Implementierung der Transferfunktionen erfolgt ähnlich der in Kapitel 3 beschriebenen Implementierung.

Eine Ausnahme ist jedoch der *Store*-Knoten, welcher für die Überprüfung der *Lokalität* eine weitere Datenflussanalyse verwenden könnte. In diesem Fall wird die Datenflussanalyse als Vorwärtsanalyse implementiert und

$$L = \{\bot, Local, NonLocal, \top\}, mit$$
$$\bot \leq Local \leq NonLocal \leq \top$$

$$in(B_i) = \bigsqcup_{B_j \in pred(B_i)} out(B_j)$$
$$out(B_i) = f_{B_i}(in(B_i))$$

gewählt.

Der Initialwert jedes Grundblocks ist LOCAL bzw. für *Address*-Knoten NONLOCAL. Die Transferfunktion f_{B_i} muss wiederum für jeden FIRM-Knoten implementiert werden. Für nicht explizit angegebene Transferfunktionen wird $f_{B_e} = NonLocal$ gewählt.

Mithilfe der Fixpunktiteration, z.B. in Form eines *Work-list*-Algorithmuses, kann der FIRM-Graph analysiert werden und alle Knoten werden entsprechend markiert. Es sind aber zwei Fixpunktiterationen nötig um beide Datenflussanalysen zu bearbeiten.

5. Evaluation

In diesem Kapitel wird die Implementierung der Analyse und Optimierungen evaluiert, insbesondere werden die Ausführungszeiten mit und ohne Optimierung betrachtet. In Abschnitt 5.1 wird zunächst ein einfacher Benchmark betrachtet. Für die *Real-World*-Evaluation wird Multigrid [10] herangezogen.

Bei dem Testsystem handelt es sich um ein 32-bit System, mit vier GB Arbeitsspeicher und einem Pentium Dual-Core Prozessor mit je 2.6 GHz pro Kern.

5.1. Einfacher Benchmark

In Programmausschnitt 5.1 ist ein X10-Programm, welches als Benchmark verwendet wird. Das Programm erstellt eine zweidimensionale rechteckige Region, welche 100 Millionen Punkte beinhaltet. Mithilfe einer *for-each*-Schleife wird über diese Punkte iteriert und die Koordinaten addiert. In jeder Iteration wird ein neuer Punkt im Heap-Speicher angelegt.

Dies ist nicht durch Programmausschnitt 5.1 ersichtlich, aber das Front-End wandelt die *for-each*-Schleife in zwei *for*-Schleifen um. In Programmausschnitt 5.2 ist die umgewandelte Version abgebildet. In diesem Ausschnitt ist zu erkennen, das ein Punkt durch die *Point.make*-Methode angelegt wird.

```
public class SimpleBenchmark {
  public static def main(Array[String]) {
    val size:Int = 10000;
    val region = (0..size) * (0..size);
    var sum:Int = 0;
    for (p in region)
      sum += p(0)+p(1);
    Console.OUT.println(sum);
  }
}
```

Programmausschnitt 5.1: Einfacher x10 Benchmark

```
for (var i:Int=0; i <= size; ++i) {
  s = ...;
  s(0) = i;
  for (var j:Int=0; j <= size; ++j) {
    s(1) = j
    var p:Point = Point.make(s);
    sum += p(0)+p(1);
  }
}
```

Programmausschnitt 5.2: Umwandlung der for-each-Schleife in zwei for-Schleifen.

Programmausschnitt 5.3 zeigt die Implementierung der *Point.make*-Methode. Diese alloziert auf jedem Steuerflusspfad ein neues Punkt-Objekt und gibt dieses zurück. Die *Point.make*-Methode wird von der Analyse korrekt als *Fresh*-Methode erkannt und die eigentliche Escape-Analyse wird ausgeführt.

```
public static def make[T](r:Array[T](1)){T<:Int}:Point(r.size) {
  switch(r.size) {
    case 1: return new Point(r(0));
    case 2: return new Point(r(0), r(1));
    case 3: return new Point(r(0), r(1), r(2));
    case 4: return new Point(r(0), r(1), r(2), r(3));
    default: return new Point(...);
  }
}
```

Programmausschnitt 5.3: Implementierung der Point.make-Methode.

Im ursprünglichen Benchmark sind die Verwender des Punktes Hauptsächlich *Call*-Knoten, welche den dynamisch gebundene Indexoperator aufrufen. Ein Beispiel zum dynamisch gebunden Indexoperator wurde in Abschnitt 3.1.6 vorgestellt.

Im folgenden werden vier Varianten des Benchmarks verglichen.

Tabelle 5.1.: Varianten des Einfachen Benchmarks

Variante	Escape-Analyse	Sonstige Optimierungen
SimpleBenchmark	inaktiv	Standard[1]
SimpleBenchmark + Escape	aktiv	Standard
SimpleBenchmark + O3	inaktiv	Erweitert[2]
SimpleBenchmark + O3 + Escape	aktiv	Erweitert

Übersetzungsdauer

Zur Messung des Übersetzungsvorganges wurden die oben genannten Varianten des Benchmarks einzeln übersetzt. Wie Tabelle 5.2 zu entnehmen ist, sind unter 3% der Gesamtübersetzungsdauer der Escape-Analyse zuzuschreiben.

Tabelle 5.2.: Übersetzungsdauer der verschiedenen Varianten von SimpleBenchmark.

Variante	Insgesamt	Nur Escape-Analyse	
SimpleBenchmark	216.07s	-	-
SimpleBenchmark + Escape	224.87s	4.72s	2%
SimpleBenchmark + O3	421.85s	-	-
SimpleBenchmark + O3 + Escape	428.36s	4.69s	1%

Laufzeit

Zur Laufzeitmessung wurde für jede Variante 10 Messungen vorgenommen. Tabelle 5.3 listet die durchschnittliche Laufzeit sowie den durchschnittlichen Speedup der Escape Varianten auf.

Tabelle 5.3.: Laufzeit der verschiedenen Varianten von SimpleBenchmark.

Variante	Laufzeit in Sekunden	Speedup
SimpleBenchmark	24.233	-
SimpleBenchmark + Escape	11.418	2.1
SimpleBenchmark + O3	13.905	-
SimpleBenchmark + O3 + Escape	2.367	5.9

[1] Die Standard-Optimierungen umfassen: remove-unused, opt-tail-rec, control-flow, local, lower-const, scalar-replace, dead, remove-phi-cycles, frame, lower-oo, lower-sels, target-lowering, opt-load-store

[2] Die Erweiterten-Optimierungen umfassen die Standard-Optimierungen sowie: reassociation, place, parallelize-mem, inline, thread-jumps

5.2. Multigrid

Multigrid ist ein in X10 geschriebene Simulation. Sie simuliert mit dem Mehrgitterverfahren, die Wärmeentwicklung auf einer Metallplatte, welche mit einem Laser beschrieben wird.

5.2.1. Dynamisch gebundene Fresh-Methoden

Multigrid verwendet *for-each*-Schleifen, welche über *Point-Regions* iterieren. Diese *Point-Regions* werden durch *DistArrays* generiert, hierbei geht jedoch die Information verloren, dass es sich um rechteckige *Point-Regions* handelt. Das Front-End ersetzt diese *for-each*-Schleifen deshalb durch normale *for*-Schleifen, welche ein Iterator dazu verwenden um über die einzelnen Punkte zu iterieren. Würde die Information, dass es sich um rechteckige *Point-Regions* handelt nicht verlogen gehen, würde das Front-End sich nicht für *for*-Schleifen mit Iterator entscheiden. Hingegen würde es sich wie beim einfachen Benchmark für zwei ineinander geschachtelte *for*-Schleifen ohne Iterator entscheiden.

Programmausschnitt 5.4 zeigt eine solche *for-each*-Schleife. Diese wird durch das Front-End durch eine *for*-Schleife mit Iterator wie sie in Programmausschnitt 5.5 abgebildet ist ersetzt. Das Problem dieser *for*-Schleife ist, dass die Analyse an dieser Stelle keine Heap-Allokation erkennen kann. Die *Iterator::next()*-Methode ist zwar durchaus *fresh*, aber dynamisch gebunden. Somit ist statisch nicht ermittelbar, welches Objekt an dieser Stelle auf dem Stack erstellt werden soll. Da keine Heap-Allokation erkannt wird, wird der Punkt weder analysiert noch optimiert.

```
private static def doParallelCopy(
  distArrayData_dst: DistArray[T], // distributed array
  distArrayData_src: DistArray[T]  // distributed array
) {
  val dist = distArrayData_src.dist;
  for (p in dist | here)
    distArrayData_dst(p) = distArrayData_src(p);
}
```

Programmausschnitt 5.4: for-each-Schleife in Multigrid.

```
private static def doParallelCopy(
  distArrayData_dst: DistArray[T], // distributed array
  distArrayData_src: DistArray[T]  // distributed array
) {
  val dist = distArrayData_src.dist;
  val reg = dist.get(here);
  for ( var i = reg.iterator(); i.hasNext(); ) {
    var p:Point = i.next()
    distArrayData_dst(p) = distArrayData_src(p);
  }
}
```

Programmausschnitt 5.5: for-each-Schleife wurde durch eine for-Schleife mit Iterator ersetzt.

Dies ist ein sehr ernüchterndes Ergebnis, deshalb werden im Folgenden einige Änderungen an Multigrid vorgenommen um diese *for*-Schleifen zu vermeiden. Generell muss erreicht werden, dass die Information, dass es sich um rechteckige *Point-Region* handelt, nicht verloren geht. Dies wird durch einen expliziten Cast der *Point-Region* in eine *RectRegion* erreicht. Damit dieser überhaupt möglich ist, muss zudem in der Standardbibliothek die Klasse *RectRegion* auf *public* gesetzt werden. Programmausschnitt 5.6 zeigt die so modifizierte Version von Programmausschnitt 5.4.

```
private static def doParallelCopy(
  distArrayData_dst: DistArray[T], // distributed array
  distArrayData_src: DistArray[T]  // distributed array
) {
  val dist = distArrayData_src.dist;
  val reg = dist.get(here) as RectRegion {rank==2,rect==true};
  for (p : Point{rank==2} in reg)
    distArrayData_dst(p) = distArrayData_src(p);
}
```

Programmausschnitt 5.6: Cast der Point-Region in eine RectRegion.

Programmausschnitt 5.6 wird durch das Front-End jetzt wiederum in zwei *for*-Schleifen ohne Iterator übersetzt, wie es schon beim Einfachen Benchmark der Fall war (vgl. Programmausschnitt 5.2). Da nun der dynamisch gebundene *Fresh*-Methodenaufruf (*Iterator::next()*) durch ein statisch gebundenen *Fresh*-Methodenaufruf (*Point.make(...)*) ersetzt wurde, kann die Analyse an dieser Stelle wie gewohnt fortfahren.

5.2.2. Benchmark

Im Folgenden werden vier Varianten des Multigrid-Benchmarks verglichen. Die Varianten, bei denen die Escape-Analyse aktiv ist, wurden zusätzlich wie in Unterabschnitt 5.2.1 beschrieben modifiziert.

Tabelle 5.4.: Varianten des Multigrid Benchmarks

Variante	Escape-Analyse	Sonstige Optimierungen
Multigrid	inaktiv	Standard
Multigrid + Escape	aktiv	Standard
Multigrid + O3	inaktiv	Erweitert
Multigrid + O3 + Escape	aktiv	Erweitert

Übersetzungsdauer

Zur Messung des Übersetzungsvorganges wurden die oben genannten Varianten des Benchmarks einzeln übersetzt. Wie Tabelle 5.5 zu entnehmen ist, sind unter 3% der Gesamtübersetzungsdauer der Escape-Analyse zuzuschreiben.

Tabelle 5.5.: Übersetzungsdauer der verschiedenen Varianten von Multigrid.

Variante	Insgesamt	Nur Escape-Analyse	
Multigrid	260.54s	-	-
Multigrid + Escape	264.71s	6.7s	2.5%
Multigrid + O3	482.64s	-	-
Multigrid + O3 + Escape	504.58s	6.8s	1.3%

Laufzeit

Zur Laufzeitmessung wurde für jede Variante 10 Messungen vorgenommen. Bei jeder Messung wurden 100 Iterationen der Simulation berechnet. Tabelle 5.6 listet die durchschnittliche Laufzeit sowie den durchschnittlichen Speedup der Escape Varianten auf. Abbildung 5.1 und Abbildung 5.2 stellen alle Messungen in Boxplot-Diagrammen dar.

Tabelle 5.6.: Laufzeit der verschiedenen Varianten von Multigrid in Millisekunden.

Variante	Laufzeit in Millisekunden	Speedup
Multigrid	15617	-
Multigrid + Escape	13148	1.188
Multigrid + O3	9163	-
Multigrid + O3 + Escape	8363	1.096

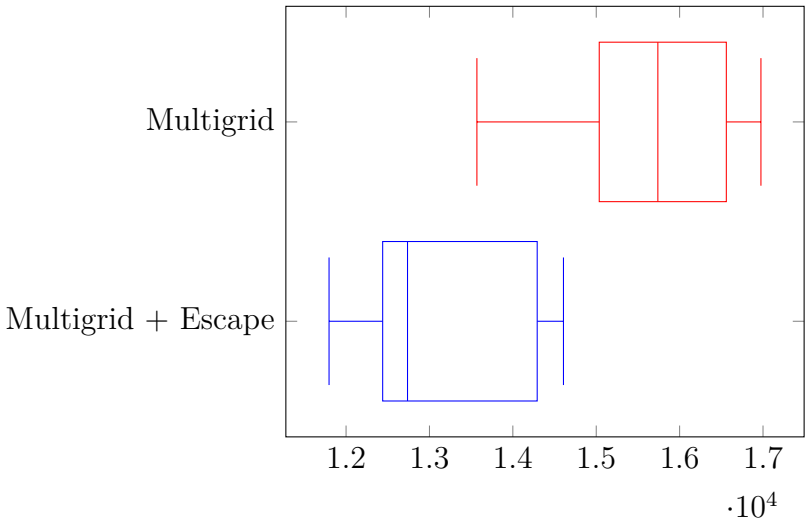


Abbildung 5.1.: Boxplot der Laufzeit von Multigrid in Millisekunden.

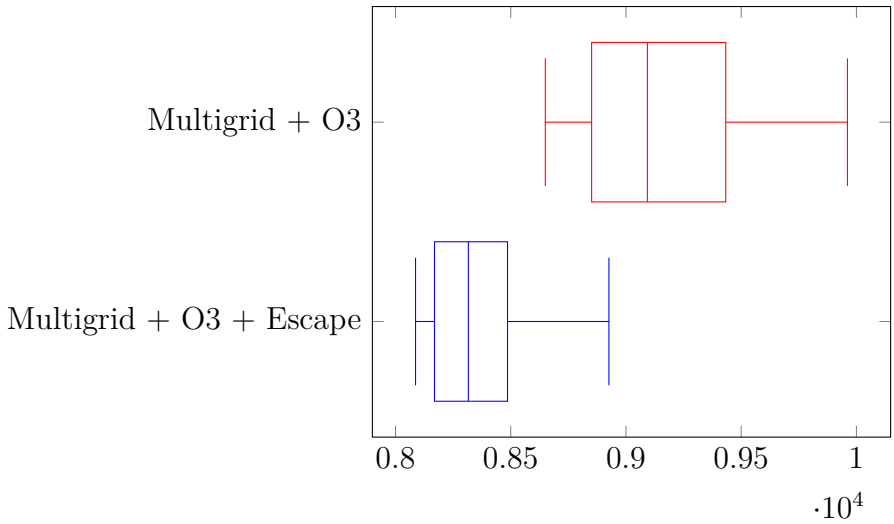


Abbildung 5.2.: Boxplot der Laufzeit von Multigrid in Millisekunden.

Sonstige Statistiken

Die Escape-Analyse und Optimierung betrachtete bei der vierten Variante (Multigrid + O3 + Escape) 5365 Firm-Graphen und brauchte im Schnitt 1,3 Millisekunden pro Graph. Die Optimierung spezialisierte 40 *Fresh*-Methoden und wandelte 317 Heap-Allokationen in Stack-Allokationen um.

6. Fazit und Ausblick

In Kapitel 2 wurden grundlegende Konzepte und verwandte Arbeiten vorgestellt. Insbesondere wurde die Analyse motiviert und Firm vorgestellt. In Kapitel 3 wurde die konkrete Implementierung beschrieben. Hervorzuheben ist das Trennungsprinzip zwischen Analyse und Optimierung, die *Call*-Knoten-Analyse, welche trotz dynamisch gebundene Methoden nicht zu konservativ approximiert, sowie die Fresh-Analyse, welche es ermöglicht ein Objekt im Stack-Speicher des Aufrufers anzulegen. In Kapitel 4 wird die Escape-Analyse als Datenflussanalyse beschrieben. Die Evaluierung der Escape-Analyse und deren Optimierungen wird in Kapitel 5 behandelt. Aus der Evaluierung folgt, dass Programme um $\sim$10% beschleunigt werden.

Es gibt an einigen Stellen noch Verbesserungsmöglichkeiten. Zum Beispiel bei der Erkennung von Heap-Allokationen, könnte der schon in Abschnitt 3.1.2 beschriebene flexiblerer Ansatz, eine komplett quellsprachenunabhängige Analyse ermöglichen. Aber auch die momentan sehr konservative Analyse der *Store*-Knoten, könnte durch eine genauere Analyse ersetzt werden. Diese sollte sich z.B. genauer mit dem Speichern von Objekten in Objekte befassen. Wie in Kapitel 4 beschrieben, lässt sich die Escape-Analyse auch als Datenflussanalyse implementieren. Dies sollte nicht allzu großem Aufwand erfordern, da viel der jetzigen Analyse wiederverwendet werden kann.

Essenziell für den produktiven Einsatz ist jedoch ein besserer Umgang mit dynamisch gebundenen *Fresh*-Methoden. Dies ist aber nicht Aufgabe der Escape-Analyse. Eine eigene Optimierung, welche sofern möglich z.B. dynamisch gebundene Aufrufe durch statische ersetzt, würde dieses Problem lösen.

Literaturverzeichnis

[1] Bytecode2firm website. `https://github.com/MatzeB/bytecode2firm`. Accessed: 2014-09-01.

[2] Cparser website. `https://github.com/MatzeB/cparser`. Accessed: 2014-09-01.

[3] Libfirm website. `http://pp.ipd.kit.edu/firm/`. Accessed: 2014-09-01.

[4] Liboo website. `https://github.com/MatzeB/liboo/`. Accessed: 2014-09-01.

[5] Sprachtechnologie und compiler vorlesungsfolien. Lehrstuhl Programmierparadigmen - IPD Snelting, Karlsruher Institut für Technologie. Sommersemester 2014.

[6] X10i website. `http://pp.ipd.kit.edu/git/X10i/`. Accessed: 2014-09-01.

[7] On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5):455–490, Sept. 2002.

[8] B. Blanchet. Escape analysis for object-oriented languages: Application to java. *SIGPLAN Not.*, 34(10):20–34, Oct. 1999.

[9] M. Braun, S. Buchwald, and A. Zwinkau. Firm—a graph-based intermediate representation. Technical Report 35, Karlsruhe Institute of Technology, 2011.

[10] H.-J. Bungartz, C. Riesinger, M. Schreiber, G. Snelting, and A. Zwinkau. Invasive computing in hpc with x10. In *Proceedings of the third ACM SIGPLAN X10 Workshop*, X10 '13, pages 12–19, New York, NY, USA, 2013. ACM.

[11] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, Nov. 2003.

[12] C. Click and M. Paleczny. A simple graph-based intermediate representation. *SIGPLAN Not.*, 30(3):35–49, Mar. 1995.

[13] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In D. Watt, editor, *Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 82–93. Springer Berlin Heidelberg, 2000.

[14] G. Lindenmaier. libFIRM – a library for compiler optimization research implementing FIRM. Technical Report 2002-5, Sept. 2002.

[15] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27. ACM, 1988.

[16] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165:165–165:174, New York, NY, USA, 2014. ACM.

[17] M. Trapp. *Optimierung objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen.* PhD thesis, University of Karlsruhe, Faculty of Informatik, Oct. 2001.

[18] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 187–206, New York, NY, USA, 1999. ACM.

Erklärung

Hiermit erkläre ich, Bernhard Scheirle, 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

A. Anhang

Tabelle A.1.: FIRM Knoten-Typen T_{EAG} (Entnommen aus [9]).

B	basic block	**I**	integer (32-bit)	**D**	floating point (64-bit)
X	control flow	**S**	integer (16-bit)	**E**	floating point (80-bit)
M	memory	**B**	integer (8-bit)	**b**	boolean
P	pointer	**F**	floating point (32-bit)	**T**	tuple
Any	T_{EAG}	**Num**	P, I, S, B, F, D, E		

Tabelle A.2.: FIRM Funktionssymbole Σ_{EAG} (Entnommen aus [9]).

Name	Typ	Beschreibung
Add	$B \times Num \times Num \to Num$	Addition
Alloc	$B \times M \times Num \to M \times P$	Allocate memory on the stack
And	$B \times Num \times Num \to Num$	Bitwise AND
ASM	$B \times variable \to variable$	Inline assembler
Bad	$\to Any$	Value of unreachable calculation
Block	$X_0 \times \cdots \times X_n \to\to B$	Basic block
Call	$B \times Num_0 \times \cdots \times Num_k \times M \to$ $Num_0 \times \cdots \times Num_k \times M$	function call
Cmp	$B \times Num \times Num \to b0 \times \cdots \times b15$	Binary compare
Cond	$B \times b \to X \times X$	Conditional branch
Const	$\to Num$	Constant value
Conv	$B \times Num \to Num$	Type conversion
Div	$B \times Num \times Num \to Num$	Division
End	$B \times X \to$	End of function
Eor	$B \times Num \times Num \to Num$	Bitwise XOR
Free	$B \to M \times P \times Num$	Release memory on the stack
Jmp	$B \to X$	Unconditional jump
Load	$B \times P \times M \to Num \times M$	Load from memory
Minus	$B \times Num \to Num$	Negate number
Mod	$B \times Num \times Num \to Num$	Modulo
Mul	$B \times Num \times Num \to Num$	Multiplication
Mux	$B \times b \times Num \times Num \to Num$	Select value depending on boolean
NoMem	$\to M$	Empty subset of memory
Not	$B \times Num \to Num$	Bitwise NOT
Or	$B \times Num \times Num \to Num$	Bitwise OR
Phi	$B \times Num_0 \times \cdots \times Num_n \to Num$	ϕ-function
Proj	$B \times T \to Num$	Projection node
Return	$B \times M \times Num_0 \times \cdots \times Num_n \to X$	Return
Rotl	$B \times Num \times Num \to Num$	Rotate left
Sel	$B \times M \times P \to M \times P$	Select field from structure
Shl	$B \times Num \times Num \to Num$	Shift left
Shr	$B \times Num \times Num \to Num$	Shift right zero extended
Shrs	$B \times Num \times Num \to Num$	Shift right sign extended
Start	$B \to X \times M \times P \times P \times T$	Start of function
Store	$B \times M \times P \times Num \to M$	Write to memory
Sub	$B \times Num \times Num \to Num$	Subtraction
SymConst	$B \times P$	Symbolical constant
Sync	$B \times M_0 \times \cdots \times M_n \to M$	Memory barrier
Unknown	$\to Any$	Undefined value