Karlsruhe Institute of Technology

Verbesserung der libFirm Inline-Optimierung

Bachelorarbeit
von

Tobias Rapp

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation (IPD)

Gutachter:	Prof. Dr.-Ing. Gregor Snelting
Betreuender Mitarbeiter:	Dipl.-Inform. Andreas Zwinkau

Bearbeitungszeit: 13. März 2013 – 22. April 2013

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, 22. April 2013

.....

(**Tobias Rapp**)

Zusammenfassung

Die Inlining Compiler Optimierung ersetzt Funktionsaufrufe durch die Funktion selbst, so dass der Aufwand für die Aufrufe vermieden wird und zusätzlich die Möglichkeit für weitere Optimierungen vorhanden ist.
Diese Arbeit beschreibt die Verbesserung der bestehenden Inline Optimierung in libFirm, welche eine Implementation der graphbasierten Zwischensprache Firm ist.
Die eingeschränkte Funktionalität der bestehenden libFirm Inline Ersetzung wurde erweitert und vervollständigt. Zusätzlich wurde das heuristische Verfahren, welches die Inline-Entscheidung und Reihenfolge der Funktionsaufrufe für die eigentliche Ersetzung bestimmt, überarbeitet.
In einer abschließender Evaluierung wird gezeigt, dass die Inline Optimierung in mehrerer Hinsicht verbessert wurde. So konnte eine Verbesserung von 4,25% mit den SPEC CPU2000 Benchmarks gemessen werden.

Abstract

The inlining compiler optimization replaces function calls with the body of the called function, thereby removing the overhead of the function call and enlarging the scope for further optimizations.
This thesis describes improvements in the existing inline optimization in libFirm, which is an implementation of the fully graph based intermediate representation Firm.
The limited functionality of the existing libFirm inliner has been enhanced and completed. Further improvements were made by replacing the inline heuristic, which determines the inline decision and order for replacing a set of function calls.
This thesis concludes, that the inline optimization has been improved in several ways. As a result, the runtime of the SPEC CPU2000 benchmark specification could be improved by 4,25%.

Inhaltsverzeichnis

1. Einleitung

Eine der wichtigsten Optimierungen die ein Compiler ausführen kann ist die Inline Ersetzung. Ziel dieser Optimierung ist es Funktionsaufrufe von Unterprogrammen zu vermeiden, indem der Aufruf durch den Funktionsrumpf ersetzt wird. Das so genannte Inlining vermeidet nicht nur den Aufwand eines Funktionsaufrufes, sondern ermöglicht dem Compiler die Ausführung weiterer intraprozeduraler Optimierungen. So können die Kopien einer Funktion an verschiedenen Stellen unterschiedlich, entsprechend dem spezifischen Kontext, optimiert werden.

Allerdings hat das Inlining den Nachteil, dass es die Größe des Programmes erhöhen und dadurch zu einem Performanz-Verlust führen kann. Es muss daher eine Entscheidung getroffen werden, ob es sinnvoll ist einen Funktionsaufruf zu ersetzen.

libFirm ist eine Implementierung der graphbasierten Zwischensprache Firm, welche es ermöglicht sprachen- und maschinenunabhängige Optimierung auszuführen. Für die Inline Optimierung bedeutet dies, dass es genügt die Optimierung für die Firm Zwischensprache zu entwickeln, da diese dann mit beliebigen Quell- und Zielsprachen verwendet werden kann.

Ziel dieser Arbeit ist es, ausgehend von der bisherigen Implementation der Inline Optimierung in libFirm, diese zu verbessern und zu erweitern.

Von Bedeutung sind vor allem die im Folgenden vorgestellten Erweiterungen der Inline Ersetzung eines Funktionsaufrufes. Diese verbessern die Möglichkeit zum Inlining in einigen von libFirm bisher nicht unterstützen Fällen. Wird als Quellsprache X10 verwendet, so sind dies keineswegs Spezialfälle. Durch die Verbesserung der Inline Ersetzung soll für die Verwendung von X10 mit libFirm im Zusammenhang mit dem „X10 Compiler for Invasive Architectures“ (vgl. [BBMZ12]) eine erhebliche Performanzsteigerung möglich sein.

Unabhängig von der Inline Ersetzung eines einzelnen Funktionsaufrufes, muss bei der Inline Optimierung für jeden Funktionsaufruf zuerst die Entscheidung getroffen werden, ob es sinnvoll ist diesen zu ersetzen. Gleichzeitig muss eine Reihenfolge festgelegt werden, in der die Inline Ersetzungen ausgeführt werden sollen. All dies wird durch ein heuristisches

Verfahren erledigt, welches damit von entscheidender Bedeutung für die Qualität der Inline Optimierung ist. Durch ein neues, verbessertes Verfahren soll nicht nur eine Performanzsteigerung möglich sein, sondern auch eine Verkleinerung der entstehenden Programmgröße.

Im Folgenden werden in Kapitel 2 werden alle benötigten Grundlagen und Begriffe, wie die Inline Optimierung und Firm erläutert. Kapitel 3 enthält alle Erweiterungen der Inline Ersetzung eines Funktionsaufrufes. Im Anschluss daran wird in Kapitel 4 die neue Inline-Heuristik für die Inline Optimierungsphase von libFirm vorgestellt. In Kapitel 5 werden die erbrachten Erweiterungen und Veränderungen evaluiert. Abschließend wird in Kapitel 6 ein Fazit gezogen.

2. Grundlagen

Dieses Kapitel stellt einige grundlegende Begriffe und Konzepte vor. Als erstes wird in Abschnitt 2.1 die Inline Optimierung vorgestellt. In Abschnitt 2.2 erfolgt die Beschreibung der Zwischensprache Firm. Darauf aufbauend wird in Abschnitt 2.3 libFirm, sowie die bestehende Implementation der Inline Ersetzung in libFirm erläutert.

2.1. Inline Optimierung

Nachfolgend ist beispielhaft die Funktion *max* dargestellt.

```
int max(int a, int b) {
    if (a >= b) {
        return a;
    }
    return b;
}
```

Zu erwarten ist, dass der Aufwand für den Aufruf dieser Funktion und das Zurückgeben des Ergebnisses deutlich höher ist als die eigentliche Ausführung des Funktionsrumpfes. Dieser Aufwand könnte vermieden werden, indem der Aufruf durch den Funktionsrumpf ersetzt wird. Während dieser Ersetzung müssen die Parameter und der Rückgabewert entsprechend angepasst werden. Die Summe derartiger Ersetzungen innerhalb eines Programms wird als Inline Optimierung bezeichnet.

Ferner ermöglicht das Ersetzen eines Aufrufes, abhängig vom konkreten Kontext, die Anwendung weiterer Optimierungen. Wären beispielsweise in einem Aufruf der obigen *max* Funktion die Parameter konstant, so könnte der Funktionsrumpf komplett wegfallen. Die Inline Optimierung ist eine interprozedurale Optimierung, die über das ganze Programm hinweg arbeitet und damit alle Funktionen und ihre Aufrufbeziehungen betrachtet. Sie ermöglicht Optimierungen, die nicht durch intraprozedurale Verfahren erfolgen können. Daraus folgt aber auch eine höhere Komplexität dieser Optimierung.

Eine Konsequenz der Inline Optimierung ist, dass dies zu einer Veränderung der Programmgröße führt. Ob es sich dabei um eine Vergrößerung oder Verkleinerung handelt, ist im Allgemeinen nicht vorhersehbar. Meist resultiert die Inline Optimierung in einer erheblichen Vergrößerung des Programms. Zu große Programme führen, vor allem aufgrund

von Cache-Effekten, zu einem Performanzverlust. Allerdings kann trotz einer erheblicher Vergrößerung des Programms die Inline Optimierung immer noch zu einer besseren Performanz führen. Insofern muss für jeden Funktionsaufruf entschieden werden, ob es sinnvoll ist die Inline Ersetzung auszuführen.

Diese Abwägung ist von zentraler Bedeutung für die Qualität der Inline Optimierung und sollte anhand einer heuristischen Funktion getroffen werden.

2.2. Firm

Üblicherweise besitzt ein Compiler ein Front-End, welches das Quellprogramm analysiert und daraus eine Zwischendarstellung, im Englischen intermediate representation (IR) genannt, erzeugt. Aus dieser Zwischendarstellung kann das Back-End den eigentlichen Zielcode erzeugen. Die Zwischendarstellung trennt nicht nur Front-End und Back-End, so dass diese idealerweise beliebig kombiniert werden können, sondern ermöglicht auch das Erstellen von Quell- und Zielsprachen unabhängigen Optimierungen.

Firm ist eine solche Zwischendarstellung und entstand 1996 am Karlsruher Institut für Technologie für den Sather-K Compiler Fiasco, woher auch der Name stammt: Fiasco's Intermediate Representation Mesh. Das grundlegende Prinzip stützt sich vor allem auf die Arbeit von C. Click (vgl. [CP95]) und wurde später insbesondere durch M. Trapp in [Tra01] erweitert. Eine aktuelle Beschreibung der Zwischendarstellung und seiner Konstruktion ist in [BBZ11] zu finden.

Bei Firm handelt es sich um eine graphbasierte Zwischendarstellung für imperative und objektorientierte Sprachen. Wie bei aktuellen Zwischendarstellungen üblich, basiert Firm auf *Static-Single-Assignment Form* (SSA-Form). Die SSA-Form entstand aus der Arbeit von B. K. Rosen, M. N. Wegman und F. K. Zadeck [RWZ88], eine Einführung befindet sich in [ALSU08].

Aus der SSA-Form folgt, dass jede Variable genau eine Definition besitzt. Dies ermöglicht es, dass in Firm eine Operation nicht von einer Variablen abhängt, sondern von der Operation, die den Wert der Variablen erzeugt. Variablen werden damit nicht benötigt und es müssen nur Werte repräsentiert werden. Auf diese Weise werden Abhängigkeiten zwischen Operanden explizit dargestellt.

2.2.1. Graph Struktur

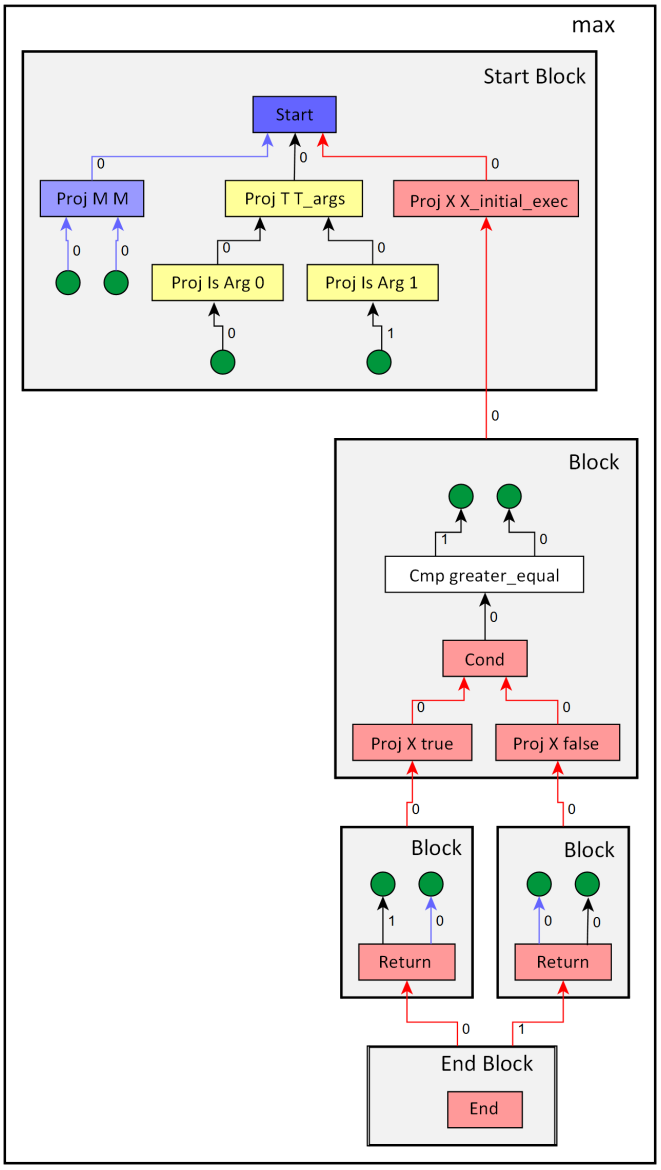
Firm Graphen sind explizite Abhängigkeitsgraphen, die in [Tra01] wie folgt definiert worden sind:

Definition 1 *(Expliziter Abhängigkeitsgraph, EAG)*
Ein expliziter Abhängigkeitsgraph (EAG) ist ein gerichteter, markierter Graph. Die Ecken sind mit Funktionssymbolen aus Σ_{EAG} markiert. Ecken besitzen geordnete Eingänge und Ausgänge. Die Anzahl der Ein- und Ausgänge der Ecken 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.2 dargestellt.

In dem FIRM Graph einer Funktion werden Operationen als Knoten und Kanten als explizite Abhängigkeiten, wie Daten- und Steuerfluss, dargestellt. Der erste Vorgänger jedes Knotens ist sein zugehöriger Grundblock, welcher als Grundblock-Knoten realisiert ist. Oft wird diese Zugehörigkeit graphisch dadurch dargestellt, dass der Block den Knoten umschließt.

Als Beispiel ist der FIRM Graph der *max* Funktion in Abbildung 2.1 abgebildet. Jeder

Abbildung 2.1.: FIRM Graph der *max* Funktion



FIRM Graph besitzt zwei ausgezeichnete Knoten *Start* und *End*, die jeweils gleichnamigen Blöcken zugeordnet sind. Die *Start*-Operation liefert ein Tupel (X, M, T) aller Informationen, die am Beginn einer Funktion benötigt werden. Dabei ist X der initiale Steuerfluss, M der Speicherzustand und T ein Tupel das alle Parameter enthält. Mit der *Proj*-Operation können einzelne Elemente aus einem Tupel entnommen werden. Durch *Proj Is Arg i*, i = 0,1 wird auf den Parameter a bzw. b aus dem Tupel T zugegriffen.

Der nächste Block enthält eine Vergleichs-Operation *Cmp*, die überprüft ob a größer gleich b ist. Der bedingte Sprung wird durch eine Cond-Operation repräsentiert. Das Ergebnis

ist ein Tupel, welches den Steuerfluss in Abhängigkeit der Vergleichskondition enthält.

Die nächsten beiden Blöcke enthalten je eine *Return*-Operation, die den jeweiligen Rückgabewert der Funktion darstellt. Der Speicherzustand wird dabei unverändert übergeben. Durch die *End*-Operation endet der Graph.

2.2.2. Darstellung von Speicherzugriffen

Bei Datenstrukturen, die sich auf dem Heap befinden, und Variablen, bei denen Aliasing-Effekte auftreten können, ist es nicht sinnvoll diese in SSA-Form zu bringen. Deshalb wird jeder Lese- und Schreibzugriff auf diese Variablen durch explizite Load- und Store-Operationen realisiert. Load- oder Store-Knoten benötigen einen SSA-Wert Speicherzustand und geben einen neuen Speicherzustand zurück.

Dies bildet eine Verkettung von Speicherzugriffen, welche aber nicht immer sinnvoll ist. Wird beispielsweise nicht mit Aliasing-Effekten gerechnet oder wird der Speicher nur gelesen, so kann mehreren dieser Operationen der gleiche Speicherzustand übergeben und die Ergebnisse mit einem Sync-Knoten zusammengeführt werden.

2.2.3. Komplexe Datentypen

Jedes Firm Programm besteht aus einer Menge von *Entitäten*. Dies sind typisierte Objekte, die sich zur Laufzeit im Speicher des Programms befinden und folglich adressierbar sind. Entitäten stellen Elemente komplexer Typen dar, wie beispielsweise Felder und Methoden von Klassen. Gleichzeitig repräsentieren diese aber auch Klassen, globale und lokale Variablen, die dem universellen Typ *Global-Type* zugewiesen werden.

Entitäten erfassen weiterhin Informationen über das Speicherlayout, wie beispielsweise Alignment, eine relative Adresse oder einen Initialisierungswert.

Das Typsystem von Firm definiert Typen wie *Compound types*, *Primitive types*, *Pointer types* und *Method types*. Bedeutend im Rahmen dieser Arbeit sind die *Compound types* und *Array types*. *Compound types* sind ein Oberbegriff für komplexe Typen, welche aus mehreren Typen zusammengesetzt sind. Dazu zählen *Struct types*, eine Menge von Feldern mit beliebigen, unterschiedlichen Typen, *Union types*, die verschiedene Interpretationen eines Speicherbereichs beschreiben und *Class types*, eine partielle Menge von Feldern und Methoden. *Array types* sind eine Sammlung von Feldern gleichen Typs. Im Folgenden werden all diese Typen als komplexe Datentypen bezeichnet.

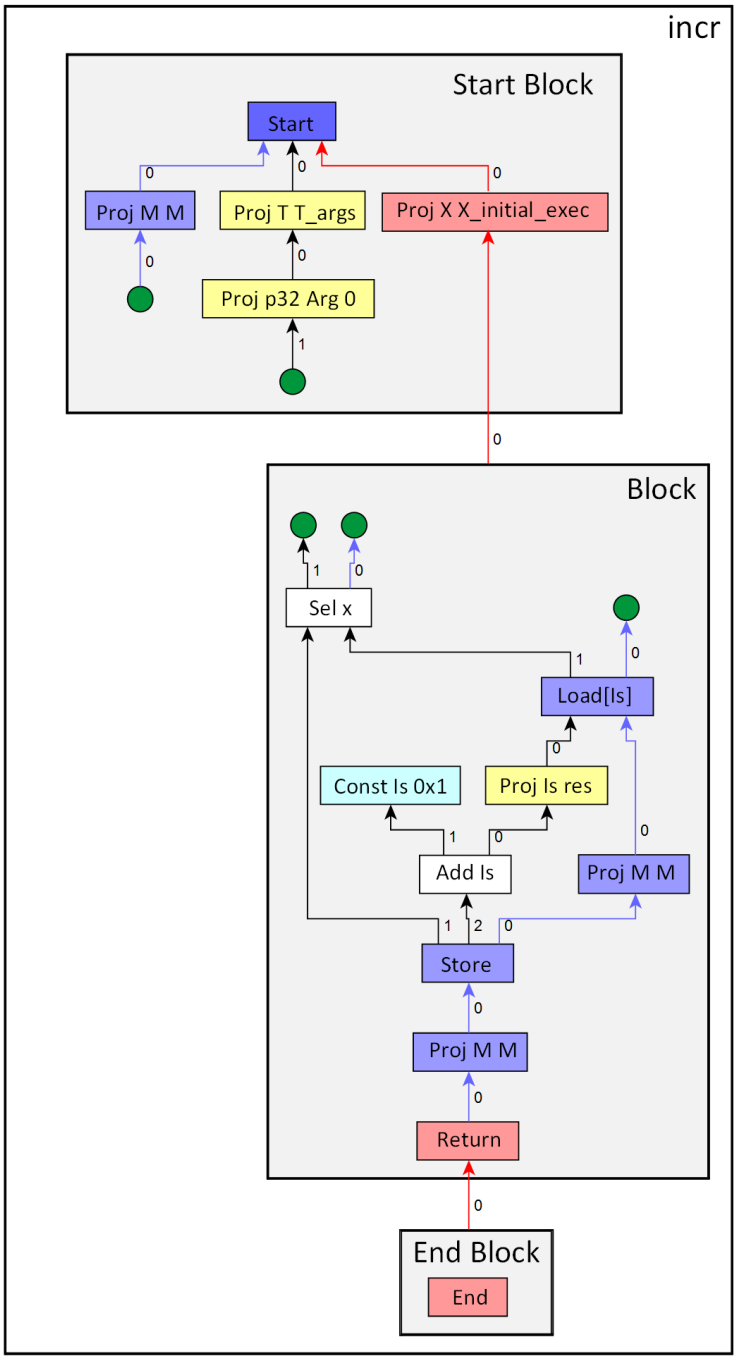
Firm stellt eine wichtige Operation zum Umgang mit diesen komplexen Datentypen bereit, die Sel-Operation. Diese abstrahiert den Zugriff auf einzelne Elemente eines komplexen Typs. Dazu hat ein Sel-Knoten eine Adresse als Operand und eine Entität als Attribut, welches das auszuwählende Element angibt. Das Ergebnis der Sel-Operation ist die Adresse des Elements.

Ein Beispiel für den Umgang mit komplexen Datentypen und der Darstellung von Speicherzugriffen ist die nachfolgende Funktion *incr*.

```
struct comp {
    int x;
};
```

```
void incr(struct comp* p) {
    ++p->x;
}
```

Abbildung 2.2.: FIRM Graph der *incr* Funktion



Der zugehörige FIRM Graph befindet sich in Abbildung 2.2. Nachdem im Start-Block alle Parameter aus dem Start-Knoten entommen wurden, wird im nächsten Block zuerst die Adresse des Elements x durch Verwendung einer Sel-Operation in dem Struct ermittelt. Anschließend wird diese Adresse durch eine Load-Operation explizit geladen. Das Speichern des Elements erfolgt durch eine Store-Operationen, die den Speicherzustand des Load-Knotens verwendet. Das Ergebnis dieser Store-Operation ist wiederum ein Speicherzustand, den der Return-Knoten benötigt.

2.3. Inline Ersetzung in libFirm

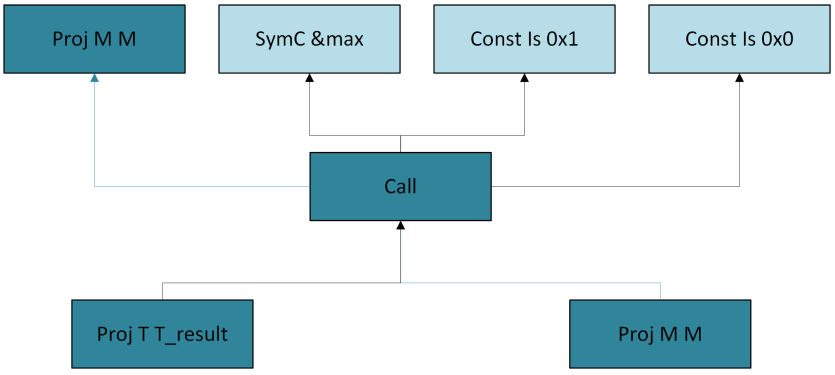
libFirm (siehe [lib]) ist eine Open Source Implementierung der Firm Zwischensprache in C. Für libFirm existieren Open Source Front-Ends für Java (vgl. [b2f]) und C (siehe [cpa]). Auch ein Back-End für die IA-32 Architektur ist bereits in libFirm integriert. Eine ältere Einführung in libFirm befindet sich in [Lin02].

Mit Beginn dieser Arbeit wurde die Inline Optimierungsphase von libFirm überarbeitet und bietet nun zwei öffentliche Funktionen an: *inline_method*, welche die Inline Ersetzung für einen Funktionsaufruf ausführt, und *inline_functions*, eine heuristische Funktion zur Bewertung von Funktionsaufrufen im ganzen Programm.

Kapitel 4 beschreibt die bestehende heuristische Funktion und die Entwicklung des neuen heuristischen Verfahrens. Nachfolgend wird die Vorgehensweise der bestehenden Inline Ersetzung kurz beschrieben. Die mit dieser Arbeit durchgeführten Erweiterungen sind in Kapitel 3 ausführlich beschrieben.

Ein vereinfachter Ausschnitt eines Firm Graphen, der einen Funktionsaufruf an die *max* Funktion enthält, ist in Abbildung 2.3 dargestellt. Anhand dieses Beispiels wird die Inline Ersetzung des *max* Call-Knoten erläutert.

Abbildung 2.3.: Vereinfachter Firm Graph welcher ein Aufruf zu *max* enthält



Die Vorgänger des Call-Knotens sind der Speicherzustand *Proj M*, eine symbolische Konstante des Funktionsnamens, und die beiden Parameter die jeweils die konstanten Werte 0 und 1 sind. Das Resultat des Funktionsaufrufes ist wiederum ein Speicherzustand und ein Tupel *Proj T*, das beliebig viele Ergebnisse beinhalten kann. Im Fall der *max* Funktion ist dies ein konstanter Wert, der durch ein Proj-Knoten aus dem Tupel entnommen wird.

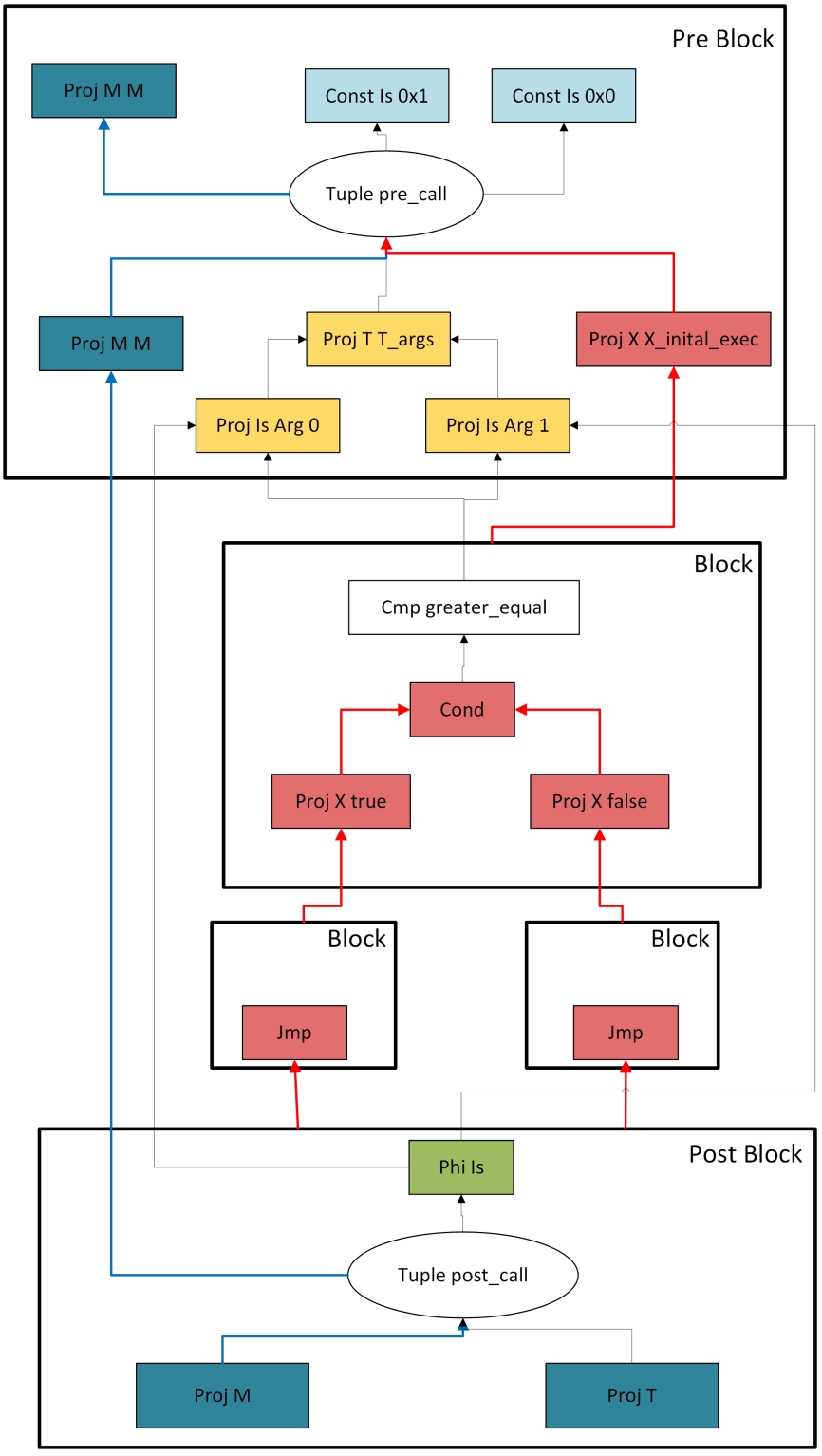
Das Ziel der Inline Ersetzung ist das Ersetzen des Funktionsaufrufes durch die aufgerufene Funktion. Dabei müssen die übergebenen Parameter in die Funktion eingesetzt und auch die Rückgabewerte angepasst werden. Firm erzwingt dies bereits dadurch, dass Start- und End-Block nicht kopiert werden dürfen. Deshalb wird anfänglich ein neues Tupel erzeugt. Diesem werden die gleichen Vorgänger wie dem Call-Knoten zugewiesen. Nach Bedarf müssen allerdings Conv-Operationen eingefügt werden, um die Typ-Korrektheit zu garantieren. Dieses Tupel wird während des Kopiervorgangs als neuer Start-Knoten benutzt, um alle initialen Werte bereit zu stellen.

Daraufhin wird der aufgerufene Graph kopiert. Hierbei ist zu beachten, dass alle Abhängigkeiten gültig und korrekt bleiben und der Start-Knoten durch das erstellte Tupel ersetzt

wird. Nach dem Kopiervorgang entsteht aus dem Call-Knoten, dem nicht mehr benötigte Funktionsaufruf, ein weiteres Tupel. Dieses soll die Ergebnisse des Funktionsaufrufes beinhalten, also einen Speicherzustand und ein Tupel mit den Ergebnissen. Dabei werden alle kopierten Return-Knoten in ein Jmp-Knoten umgewandelt und die Ergebnisse werden durch eine Phi-Operation zusammengeführt.

Dies ist der abstrakte Vorgang der Inline Ersetzung. Verschiedene Details, wie beispielsweise die Behandlung von Exceptions, werden hierbei ausgelassen. Der entstehende Graph ist in Abbildung 2.4 vereinfacht abgebildet.

Abbildung 2.4.: Vereinfachter FIRM Graph nach der Inline Ersetzung von *max*



3. Erweiterung der Inline Ersetzung

Dieses Kapitel umfasst die Verbesserung an der Inline Ersetzung in libFirm. Diese basieren größtenteils auf der Behandlung von Funktionen mit Parameter Entitäten, welche in Abschnitt 3.1 vorgestellt werden.

Funktionen, die Parameter mit komplexen Datentypen besitzen, sind wichtige Sonderfälle von Funktionen mit Parameter Entitäten und werden gesondert behandelt. Die Bedeutung und Vorgehensweise der erweiterten Inline Ersetzung bezüglich dieser Funktionen wird in Abschnitt 3.2 erläutert. Abschnitt 3.3 behandelt Funktionen mit komplexen Datentypen als Rückgabewerte. Abschließend werden in Abschnitt 3.4 die Funktionen betrachtet, die Alloca Aufrufe beinhalten.

3.1. Parameter Entitäten

Eine Parameter Entität ist eine Entität, welche die Adresse eines Parameters repräsentiert. Diese existiert genau dann, wenn innerhalb einer Funktion auf die Adresse eines Parameters zugegriffen wird. Die Funktion *foo* stellt ein solches Beispiel dar:

```
int bar(int *ptr) {
        return *ptr;
}

int foo(int param) {
        return bar(&param);
}
```

Funktionen, die ein oder mehrere Parameter Entitäten enthalten, können aufgrund der folgenden Erweiterung nun ersetzt werden.

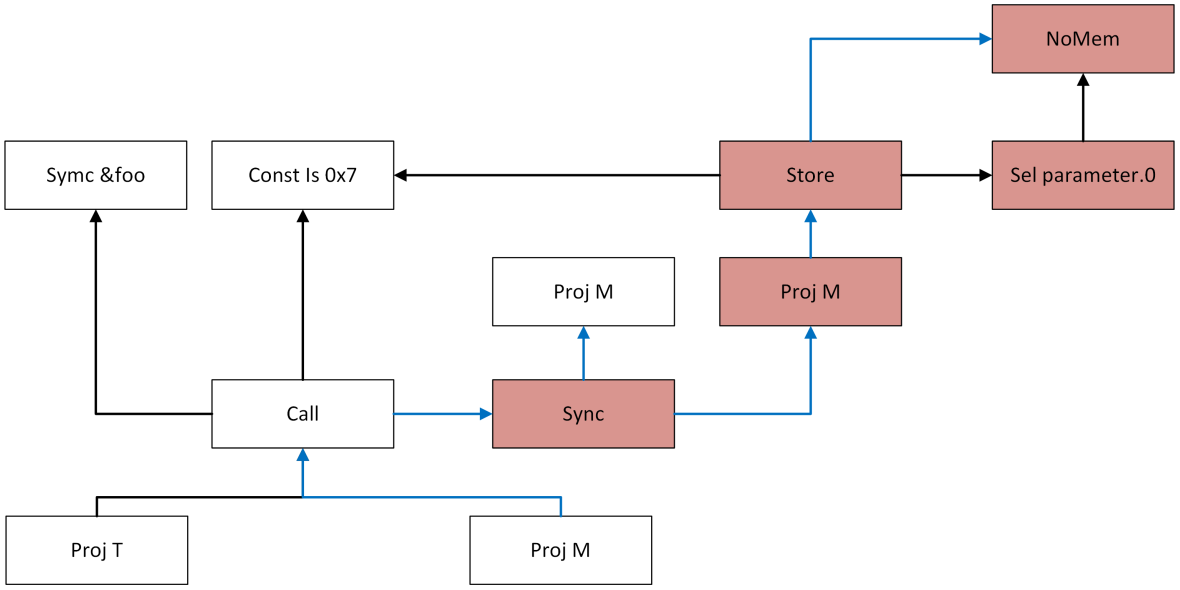
Wird ein Funktionsaufruf durch eine Funktion mit Parameter Entitäten ersetzt, so muss die Adresse der übergebenen Parameter in dem aufrufenden Firm Graphen vorhanden sein. Dies ist aber nicht immer der Fall, es können beispielsweise auch konstante Werte als Parameter übergeben werden.

Für jeden Parameter mit komplexem Datentyp muss wie in Abschnitt 3.2 beschrieben vorgegangen werden. Ansonsten kann der Graph mit dem Funktionsaufruf wie folgt verändert werden, damit die Inline Ersetzung korrekt ablaufen kann.

Damit die Adresse aller betroffenen Parameter verwendbar ist, müssen diese auf den Stack Frame der Funktions kopiert werden, die den Funktionsaufruf enthält. In Firm wird dazu für jede Parameter Entität eine neue Entität mit gleichem Typ erzeugt, welche dem aufrufenden Graph zugewiesen wird. Diese Entität soll die Parameter Entität ersetzen, welche nicht kopiert werden darf. Anschließend wird eine Sel-Operation hinzugefügt, die auf die neue Entität zugreift. Da die neue Entität auf dem Stack Frame liegt, benötigt die Sel-Operation keinen Speicherzustand. Ihr wird deshalb ein NoMem-Knoten übergeben. Nun wird eine Store-Operation erstellt, welche die Sel-Operation nutzt, um den betroffenen Parameter zu speichern. Auch der Store-Knoten ist abhängig von einem NoMem-Knoten und gibt einen neuen Speicherzustand zurück, der dem Call-Knoten zugeführt werden muss. Hierfür werden alle benötigten Speicherzustände mit einer Sync-Operation zusammengeführt.

Abbildung 3.1 zeigt die neuen Operationen, die vor den Call-Knoten eingefügt werden.

Abbildung 3.1.: Ausschnitt eines Firm Graphen, in dem die Inline Ersetzung der Funktion *foo* ermöglicht wird. Die rötlich gefärbten Knoten sind neu hinzugefügt.



Es gibt zwei Fälle, in denen die Inline Ersetzung auf diese Weise nicht durchgeführt werden kann. Dies betrifft Funktionen mit einer variablen Anzahl von Parametern und Funktionen die innere Funktionsdefinitionen (engl. nested functions) enthalten.

3.2. Parameter mit komplexem Typ

Besitzt eine Funktion ein oder mehrere komplexe Datentypen als Parameter, so kann die ursprüngliche Inline Ersetzung keinen Aufruf dieser Funktion ersetzen. Als Beispiel dient die folgende Funktion:

```
typedef struct comp {
    int a, b;
} comp;
```

```
int max_comp(comp param) {
    if (param.a >= param.b) {
        return param.a;
    }
    return param.b;
}
```

Der Parameter ist ein *struct*, also ein *compound type*, deshalb ist die Inline Ersetzung dieser Funktion bislang nicht möglich. Da die Funktion sehr klein ist und wahrscheinlich durch weitere Optimierungen komplett wegfallen würde, wäre dies aber wünschenswert.

In jedem Aufruf der *max_comp* Funktion muss für den übergebenen Parameter eine lokale Kopie erzeugt werden. Dies begründet nicht nur die Schwierigkeit bei der Inline Ersetzung, sondern ist in der Sprache C meist unerwünscht, da dort üblicherweise Zeiger auf komplexe Datentypen übergeben werden.

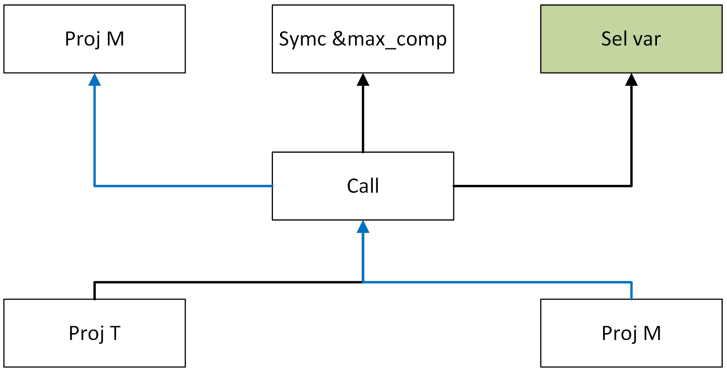
libFirm unterstützt aber weitere Sprachen, beispielsweise X10, wo dies von Bedeutung ist. So kann in X10 bereits die folgende triviale Funktion von der ursprünglichen Inline Ersetzung nicht behandelt werden:

```
public def foo(x : int) : int {
    return x * 42;
}
```

Der Grund dafür ist, dass in X10 primitive Typen als Objekte repräsentiert werden.

Ein Ausschnitt eines Firm Graphen, der *max_comp* aufruft, ist in Abbildung 3.2 gegeben.

Abbildung 3.2.: Ausschnitt eines Firm Graphs, der *max_comp* aufruft. Der übergebene Parameter ist in grün dargestellt.



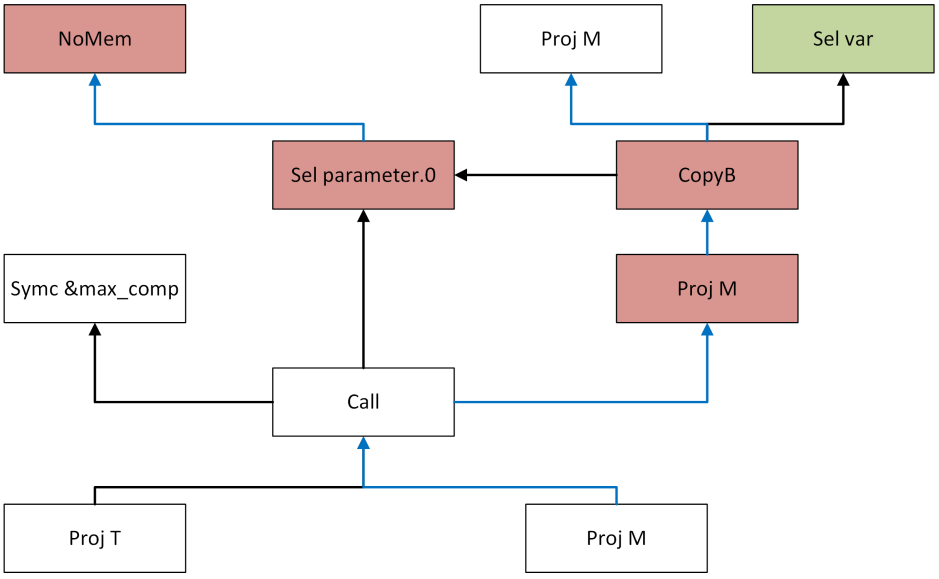
Damit der Aufruf korrekt ersetzt werden kann, müssen alle Parameter mit *Compound* oder *Array type* angepasst werden. Dies ergibt sich daraus, dass jeder dieser Parameter eine zugehörige Parameter Entität besitzt, die nicht wie in Abschnitt 3.1 beschrieben behandelt werden darf.

Da die jeweils zugehörige Parameter Entität nicht kopiert werden kann, muss auf dem Stack Frame der aufrufenden Funktion ein neue Entität mit gleichem Typ erzeugt werden. Für den Zugriff auf diese neue Entität muss eine zugehörige Sel-Operation erstellt

werden. Da diese Entität auf dem Stack Frame liegt, wird der Speicherzustand durch ein NoMem-Knoten dargestellt. Der eigentliche Kopiervorgang des Parameters wird durch eine CopyB-Operation repräsentiert. Diese benötigt nicht nur die alte und neue Sel-Operation, sondern auch den vorherigen Speicherzustand des Call-Knotens. Das Ergebnis ist ein neuer Speicherzustand, der dem Call-Knoten übergeben wird. Gibt es mehrere Parameter Entitäten, müssen die neuen Speicherzustände mit einer Sync-Operation zusammengeführt werden.

Der veränderte Ausschnitt des Firm Graphen ist in Abbildung 3.3 abgebildet. Nach der Veränderung des Paramters kann der Funktionsaufruf wie gewohnt ersetzt werden.

Abbildung 3.3.: Veränderter Graph bei dem der Parameter angepasst wurde und die Inline Ersetzung möglich ist



3.3. Komplexe Datentypen als Rückgabewerte

Funktionen mit einem komplexen Datentyp als Rückgabewert werden von der ursprünglichen Inline Ersetzung ausgenommen. Dies liegt daran, dass der Typ überprüft wird, um eventuelle Typumwandlungen durchzuführen. Bei komplexen Datentypen ist die Überprüfung und Umwandlung nicht möglich und führt infolgedessen zu Fehlern.

Das Weglassen der Typüberprüfung bei komplexen Datentypen ist ausreichend, um die korrekte Inline Ersetzung zu ermöglichen. Auch die Semantik bleibt dahin gehend korrekt, so ist beispielsweise das Anlegen einer Kopie nicht erforderlich.

3.4. Alloca Aufrufe

Ein Alloca Aufruf alloziert Speicher auf dem Stack Frame der aufrufenden Funktion. In Firm werden solche Aufrufe durch Alloc-Operationen repräsentiert, die den Stack Frame adressieren. Eine Unterscheidung von Alloca Aufrufen und variable-length arrays (VLA) ist nicht möglich.

Da der allozierte Speicher eines Alloca Aufrufes erst am Ende der Funktion freigegeben wird, kann das Inlining von Funktionen, die solche Aufrufe enthalten, den Speicherverbrauch eines Programms drastisch erhöhen. Dies ist abhängig von der Anzahl der Alloca Aufrufe und der Aufrufhäufigkeit der zu ersetzenden Funktion. Funktionen welche variable-length arrays verwenden sind hiervon nicht betroffen und können somit ohne Bedenken ersetzt werden. Bislang werden aber alle Funktionen, die Alloca Aufrufe oder variable-length arrays enthalten, von der ursprünglichen Inline Ersetzung ausgenommen.

Das Inlining dieser Funktionen ist jedoch kein besonderer Fall und sollte bei geeigneten Funktionen durchgeführt werden. Deshalb wird der Einfluss von Alloca Aufrufen in die Inline-Heuristik aufgenommen, wo eine konservative Entscheidung zum Inlining getroffen wird. Dadurch kann bei häufig auftretenden Funktionsaufrufen, beispielsweise in einer Schleife, das Inlining untersagt werden. Gleichzeitig kann bei geeigneten Funktionsaufrufen die Inline Ersetzung durchgeführt werden.

4. Die Inline-Heuristik

Ausgehend von der Inline Ersetzung eines einzelnen Funktionsaufrufes, beschreibt dieses Kapitel die Inline Optimierung einer ganzen Übersetzungseinheit. Dabei muss für alle vorhandenen Call-Knoten die Entscheidung getroffen werden, ob die Inline Ersetzung ausgeführt werden soll oder nicht. Gleichzeitig werden alle ausgewählten Funktionsaufrufe ersetzt.

Die Entscheidung, ob die Inline Ersetzung für einen Funktionsaufruf durchgeführt wird, lässt sich auf praktikable Weise nur durch eine Heuristik treffen. Dieses Kapitel beschreibt die ursprüngliche Implementierung der Inline-Heuristik von libFirm sowie die neue Inline-Heuristik, welche im Laufe dieser Arbeit entstanden ist.

Abschnitt 4.1 fasst das Vorgehen der ursprünglich in libFirm implementierten Heuristik zusammen und zeigt dessen Vor- und Nachteile auf. Das grundlegende Verfahren der verbesserten Inline-Heuristik wird in Abschnitt 4.2 erläutert. Die Betrachtung der eigentlichen Berechnung der Priorität der einzelnen Funktionsaufrufe erfolgt anschließend in Abschnitt 4.3. Abschnitt 4.4 analysiert die adressierbare Prioritätsliste, die im Rahmen dieser Arbeit entstand und die Grundlage für das heuristische Verfahren bildet. Abschließend wird in Abschnitt 4.5 die ursprüngliche und die neue Inline-Heuristik analysiert, um die algorithmischen Komplexitäten abzuschätzen.

4.1. Die ursprüngliche Inline-Heuristik

Die ursprüngliche Inline-Heuristik von libFirm beruht auf einem Bottom-Up Verfahren, welches in Abbildung 4.1 als UML Aktivitätsdiagramm dargestellt ist.

Es wird zuerst der Callgraph berechnet, alle Graphen daraus entnommen und sequentiell in umgekehrter Richtung als Array gespeichert. In der nächsten Aktivität wird jeder Graph in dem Array durchlaufen, um all seine Funktionsaufrufe zu finden. Dabei werden gleichzeitig weitere Informationen wie beispielsweise die Graphgröße und die Anzahl Blöcke berechnet. Im nächsten Schritt werden wieder alle Graphen nacheinander abgearbeitet. Für alle Call-Knoten jedes Graphen wird dabei die Priorität berechnet, um diese in eine Prioritätsliste einzufügen, falls die Priorität hoch genug ist. Dies wird durch ein Threshold-Wert entschieden, welcher der Inline Optimierung als Parameter übergeben wird. Aus der Prioritätsliste

wird solange das Element mit der höchsten Priorität entnommen, bis alle Call-Knoten aller Graphen abgearbeitet sind. Hierbei wird für jeden entnommenen Call-Knoten versucht die Inline Ersetzung auszuführen. Ist dies erfolgreich, so müssen die Informationen der jeweiligen aufrufenden Graphen aktualisiert werden, da sich diese durch die Inline Ersetzung verändern können. Eventuell neu eingefügte Call-Operationen aus den kopierten Graphen mit entsprechend hoher Priorität werden in die Prioritätsliste eingefügt. Sind alle Graphen abgearbeitet, kann auf alle geänderten Graphen eine abschließende Optimierung durchgeführt werden.

Dieses Vorgehen ist unkompliziert, da jeder Graph einzeln betrachtet wird. Die gewählte Reihenfolge hat den Vorteil, dass Funktionen die Blätter im Callgraph sind eine hohe Inline-Wahrscheinlichkeit besitzen.

Abbildung 4.2 zeigt einen beispielhaften Callgraphen. Es sind A, B, C Funktionen, wobei A und B Funktionsaufrufe F_A und F_B mit Prioritäten $\rho(F_A) \leq \rho(F_B)$ enthalten. Bedingt durch die Prioritäten folgt, dass zuerst F_B und dann F_A ersetzt werden sollte. Die Heuristik betrachtet die Funktionen nacheinander in umgekehrter Reihenfolge des Callgraph, also zuerst den Graph C, dann B und zuletzt A. Damit wird implizit, unabhängig von der Voraussetzung $\rho(F_A) \leq \rho(F_B)$, zuerst F_B und dann F_A ersetzt. Auch wenn die Prioritäten nicht verglichen werden, so ist dieses Vorgehen optimal.

Seien nun die Prioritäten durch $\rho(F_A) > \rho(F_B)$ gegeben. Im Idealfall müsste also zuerst F_A, dann F_B ersetzt werden. Die Heuristik geht jedoch gleich wie in dem vorherigen Fall vor. Es wird zuerst F_B und dann F_A ersetzt, was nicht optimal ist. Daraus könnte resultieren, dass die Inline Ersetzung von F_A nicht erfolgt, da durch die vorherige Ersetzung von F_B die Priorität von $\rho(F_A)$ unter den Threshold-Wert fällt oder die Funktion B über die maximale Größe steigt. Der Grund für dieses Verhalten ist, dass die Prioritäten nur lokal in den Graphen beachtet werden und damit die Reihenfolge der Inline Ersetzungen von Call-Knoten in unterschiedlichen Graphen durch die Aufrufreihenfolge gegeben ist.

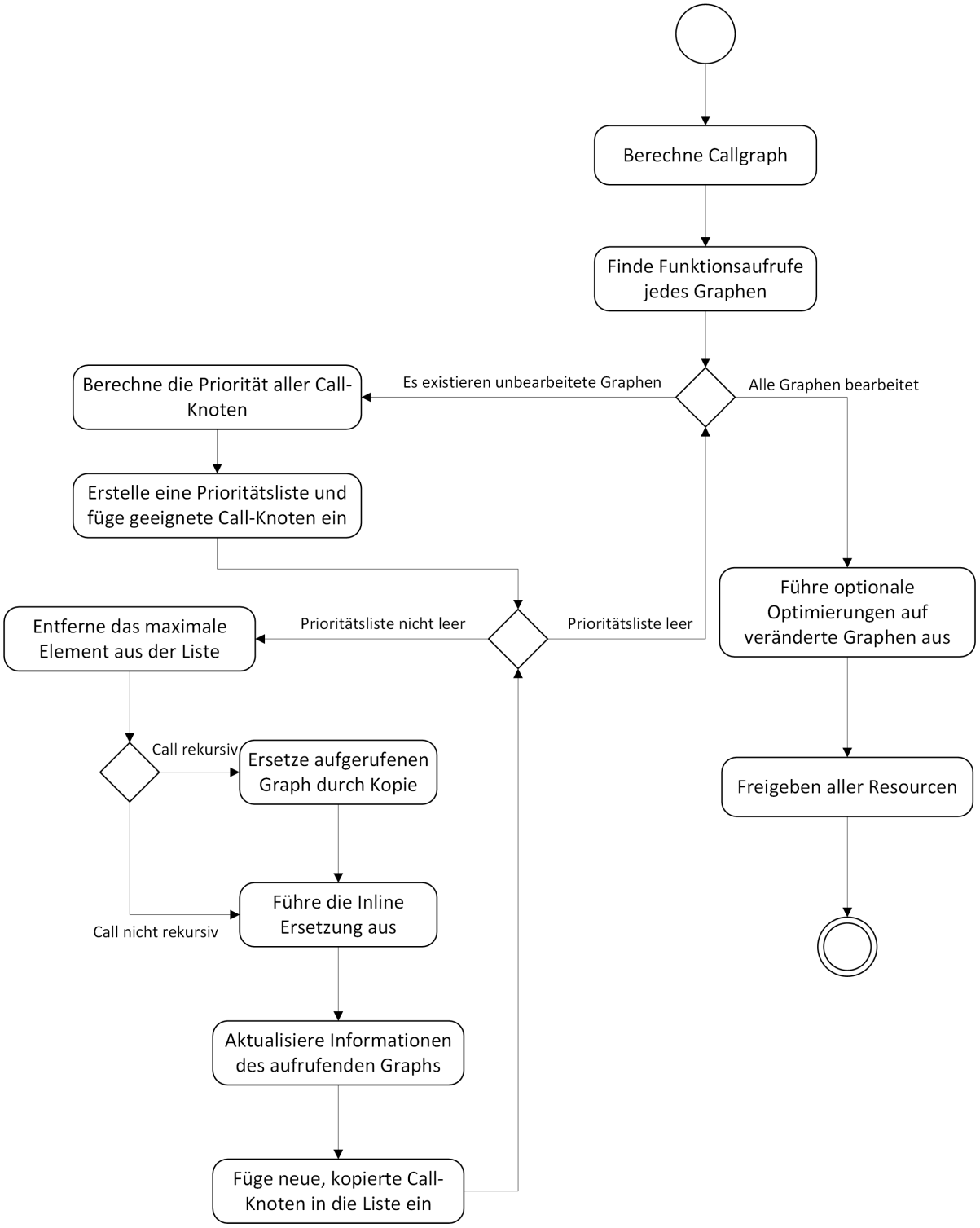
Das Verfahren unterstützt auch rekursive Funktionsaufrufe, indem vor der Inline Ersetzung eine Kopie des Graphen angelegt wird. Bei der Inline Ersetzung eines rekursiven Funktionsaufrufes werden alle neu entstehenden Call-Knoten in die Prioritätsliste aufgenommen. Auf diese Weise führen rekursive Funktionsaufrufe eines Graphen zu einer Folge von Inline Ersetzungen, die erst aufhören, wenn die maximale Graphgröße erreicht ist. Besser wäre es, wenn die Prioritäten der jeweils neuen rekursiven Call-Knoten im Laufe des Inlinings entsprechend angepasst würden. Auf diese Weise könnte man die Vorteile des Inlinings beibehalten und gleichzeitig die Graphen nicht zu sehr vergrößern.

4.2. Die Inline-Heuristik mit globaler Priorisierung

Die grundlegende Schwäche der ursprünglichen Inline-Heuristik ist, dass die Graphen einzeln und in einer festgelegten Reihenfolge betrachtet werden. Im Gegensatz dazu berechnet die neue, in dieser Arbeit entstandene, Inline-Heuristik eine globale Priorität für jeden Funktionsaufruf. Unabhängig von der Zugehörigkeit eines Funktionsaufrufes zu einem Graphen werden die globalen Prioritäten verglichen.

Jede Inline Ersetzung eines Funktionsaufrufes in einen Graphen bewirkt eine Änderung der berechneten Prioritäten von allen Call-Knoten, welche diesen Graphen wiederum ersetzen könnten. Infolgedessen müssen die Prioritäten all dieser betroffener Call-Knoten nach jeder Inline Ersetzung aktualisiert werden. Durch die dynamische Anpassung der einzelnen

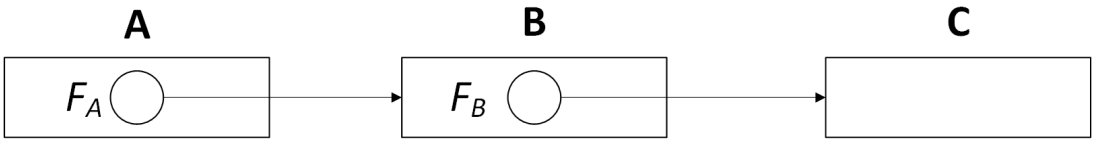
Abbildung 4.1.: Ein UML-Aktivitätsdiagramm, das den Ablauf des ursprünglichen Inline Verfahrens darstellt.



Prioritäten nach jeder Inline Ersetzung kann die Heuristik bessere Entscheidungen treffen, allerdings wird dazu eine Prioritätsliste benötigt, die diese Operation unterstützt. Abbildung 4.3 stellt das Vorgehen der Heuristik in einem UML-Aktivitätsdiagramm dar.

Dieses Verfahren ist nicht von der Aufrufreihenfolge der einzelnen Graphen abhängig, deshalb muss der Callgraph nicht berechnet werden. Die Graphen werden in beliebiger Reihen-

Abbildung 4.2.: Einfaches Beispiel eines Callgraph, der die Graphen A, B, C und zwei Funktionsaufrufe F_A und F_B enthält



folge durchlaufen, um alle Call-Knoten zu finden und die für die Berechnung der Priorität benötigten Informationen über die Graphen zu sammeln. Für jeden Call-Knoten wird anschließend die globale Priorität berechnet. Liegt diese Priorität über dem Threshold-Wert der Inline-Optimierung, so wird der Call-Knoten in eine Prioritätsliste eingefügt.

Sind alle notwendigen Vorbereitungen getroffen und enthält die Prioritätsliste alle Call-Knoten die ersetzt werden könnten, so wird diese abgearbeitet. In jedem Durchlauf wird dabei das Element mit der höchsten Priorität entnommen und einzeln betrachtet. Ist der aktuelle Call-Knoten ein rekursiver Funktionsaufruf, so kann der Graph nicht in sich selber ersetzt werden. Für die Inline Ersetzung von rekursiven Funktionsaufrufen ist es notwendig eine Kopie des Graphen zu erzeugen. Um nicht immer wieder eine neue Kopie zu erstellen, werden diese gespeichert und nach Bedarf wieder verwendet. Daraufhin kann für den Call-Knoten die Inline Ersetzung aufgerufen werden. Ist das Inlining des Funktionsaufrufes erfolgreich, so ist es möglich und sogar wahrscheinlich, dass in den aufrufenden Graph während der Inline Ersetzung neue Call-Knoten kopiert wurden. Deshalb werden diese neuen Funktionsaufrufe, falls sie eine geeignete Priorität haben, in die Prioritätsliste aufgenommen.

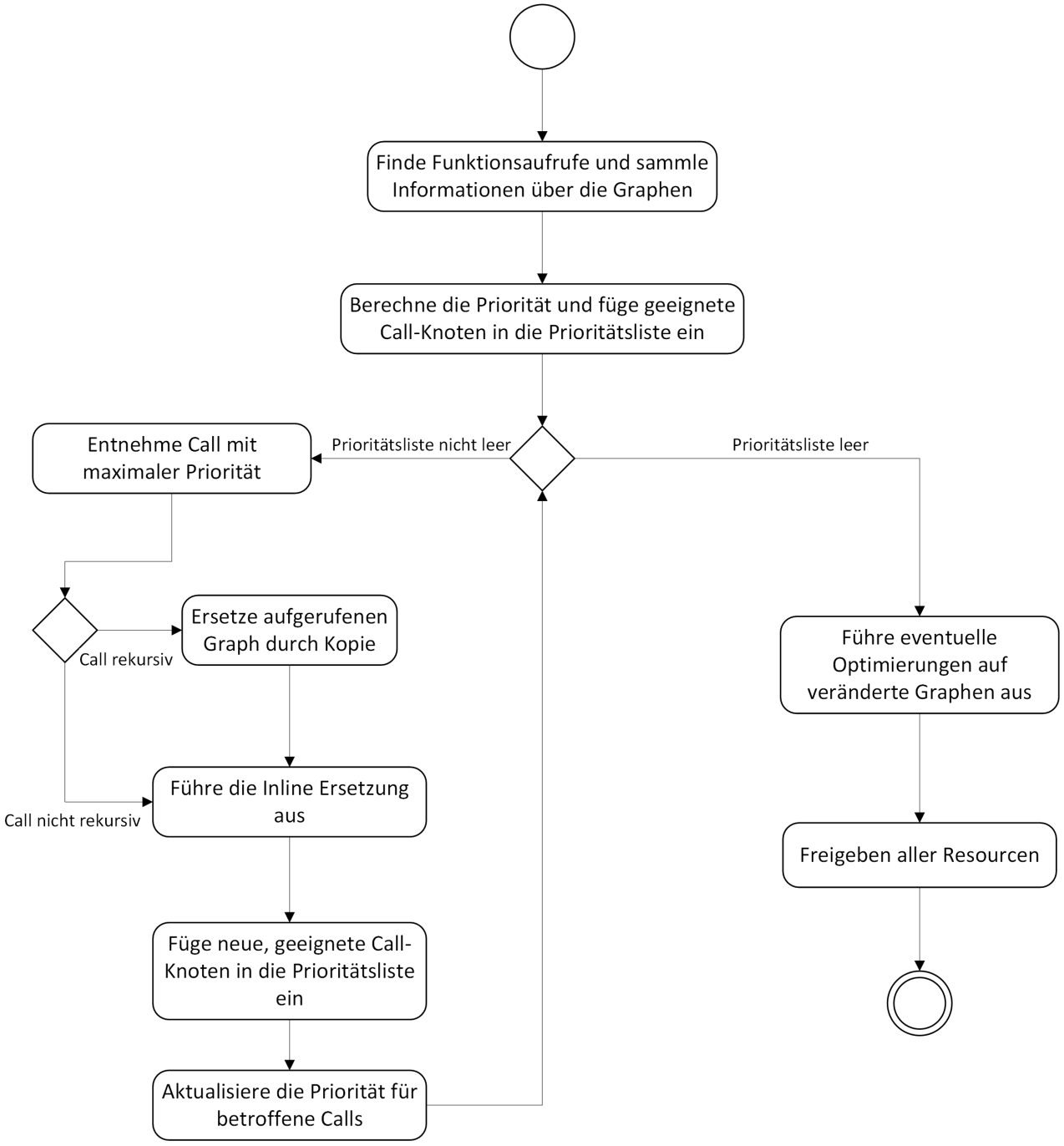
Während der Inline Ersetzung verändert sich der Graph in dem ein Funktionsaufruf ersetzt wird. Aufgrund dessen wird für alle Funktionsaufrufe, die diesen veränderten Graphen ersetzen könnten, die Priorität aktualisiert. Dies wird dadurch realisiert, dass jeder Graph eine Liste von Call-Knoten besitzt, dessen Prioritäten von diesem Graphen abhängig sind. Nicht alle Call-Knoten, die sich anfangs in der Prioritätsliste befinden, müssen zwangsweise ersetzt werden. So kann die maximale Graphgröße erreicht werden oder durch die Veränderung der Prioritäten können Call-Knoten unter den Threshold-Wert fallen und damit aus der Prioritätsliste entfernt werden. Enthält die Prioritätslise schließlich keine Elemente mehr, so wird für jeden veränderten Graphen eine Optimierung aufgerufen, falls diese der Inline-Heuristik übergeben wird.

4.3. Anpassung der Prioritätsberechnung

Die Priorität ρ eines Call-Knotens c setzt sich aus drei verschiedenen Werten zusammen: Der Aufrufhäufigkeit $h(c)$, dem statischen Gewicht $stat(c)$ und dem dynamischen Gewicht $dyn(c)$. Die genaue Berechnung erfolgt nach der Gleichung $\rho(c) = h(c) * (stat(c) + dyn(c))$.

Die Aufrufhäufigkeit wird aus dem *execfreq* Modul von libFirm gewonnen. Sie gibt an, wie oft ein Basis Block voraussichtlich ausgeführt wird. Dieser Wert kann eine Schätzung anhand des Kontrollflussgraphen sein oder aus Profile-Informationen gewonnen werden. Das statische Gewicht wird für jeden Call-Knoten genau einmal berechnet. Alle Eigenschaften des Funktionsaufrufes, die sich nicht während der Inline Optimierungsphase ändern, werden an dieser Stelle einbezogen. Im Gegensatz dazu wird das dynamische Gewicht mehrfach

Abbildung 4.3.: Ein UML-Aktivitätsdiagramm, dass den Ablauf der Inline-Heuristik mit globaler Prioritisierung darstellt.



berechnet und zwar immer dann, wenn sich der zu ersetzende Graph verändert. Alle Eigenschaften, die von dieser Veränderung betroffen sind, werden hierbei ausgewertet. Eine Tabelle mit den statischen und dynamischen Gewichtungen befindet sich in Tabelle A.1.

4.4. Die adressierbare Prioritätsliste

Bei der Inline-Heuristik mit globaler Priorisierung steht als zentrale Datenstruktur eine Prioritätsliste im Mittelpunkt. Hierin werden alle Call-Knoten mit ihrer berechneten Priorität eingefügt, damit im darauffolgenden Schritt der Heuristik das Element mit der höchsten Priorität entnommen werden kann.

Einer der wichtigsten Vorteile der neuen Inline-Heuristik ist, dass sich die Priorität von Call-Knoten im Laufe des Inline-Prozesses verändern können. Daraus lassen sich zusätzliche Anforderung an eine Prioritätsliste ableiten: Das Verändern der Priorität und Entfernen von Elementen, die sich bereits in der Datenstruktur befinden.

libFirm stellt eine Prioritätsliste *pqueue* zur Verfügung. Diese ist mittels eines binären Heaps realisiert, welches eine einfache und schnelle Implementation darstellt. Die genannten zusätzlichen Anforderungen lassen sich mit dieser Prioritätsliste jedoch nicht ausführen, da einzelne Elemente nicht adressierbar und damit nach dem Einfügen auch nicht mehr veränderbar sind.

Deshalb ist im Laufe dieser Arbeit die adressierbare Prioritätsliste *apqueue* erstellt worden, die als Pairing Heap (vgl. [FSST86]) implementiert ist. Die Operationen der Prioritätsliste mit jeweiliger Komplexität sind in Tabelle 4.1 dargestellt. Es ist zu beachten, dass die genaue algorithmische Komplexität von *apqueue_change_priority* unbekannt ist (vgl. [Fre99] und [Pet05]). Auch wenn eine vollständige theoretische Analyse des Pairing Heaps immer noch ein offenes Problem ist, so sind diese effizient in der Praxis. J. Stasko und J. Vitter [SV87], als auch B. Moret und H. Shapiro [MS91] belegen diese Annahme und zeigen das Pairing Heaps genauso schnell, oft auch schneller als Binäre Heaps oder andere vergleichbare Datenstrukturen sind.

Tabelle 4.1.: Operationen der adressierbaren Prioritätsliste *apqueue*. Dabei bezeichnet n die Länge, mit (*) gekennzeichnete Komplexitäten sind amortisiert.

Komplexität	**Operation**	**Beschreibung**
$O(1)$	new_apqueue	Erstellen der Datenstruktur
$O(1)$	del_apqueue	Entfernen und Freigeben aller Ressourcen
$O(1)$	apqueue_put	Einfügen eines Elements
$O(\log n)$*	apqueue_pop_front	Entfernt und gibt das kleinste Element zurück
$O(1)$	apqueue_length	Die Größe der Prioritätsliste
$O(1)$	apqueue_empty	Prüft, ob die Prioritätsliste leer ist
$O(1)$	apqueue_contains	Prüft, ob ein Element enthalten ist
$O(1)$	apqueue_get_priority	Gibt die Priorität eines Elements zurück
$O(\log n)$*	apqueue_remove	Entfernt ein Element aus der Prioritätsliste
$2^{O(\sqrt{\log \log n})}$*	apqueue_change_priority	Ändert die Priorität eines Elements

4.5. Abschätzung der algorithmischen Komplexität

In diesem Abschnitt wird die Komplexität der ursprünglichen und der neuen Inline-Heuristik abgeschätzt. Dazu wird zwischen dem *worst-case* und *average-case* unterschieden.

Für den *average-case* wird angenommen, dass der Callgraph azyklisch ist und keine rekursiven Funktionsaufrufe enthält. Der *worst-case* liegt genau dann vor, wenn der Callgraph zyklisch ist und jeder darin enthaltene Funktionsgraph einen Aufruf zu jeder Funktion besitzt. Jede Funktion enthält damit auch einen rekursiven Aufruf zu sich selbst. Der *worst-case* ist sehr unwahrscheinlich und ist daher weniger aussagekräftig, gibt aber dennoch Aufschluss über die generelle Komplexität.

Es sei n_{ops} die Anzahl an FIRM Operationen im Programm und n_g die Anzahl an Funktionsgraphen. Mit $i = 0, .., n_g$ sei weiterhin n_i die Anzahl von Operationen des Graphen G_i und c_i die Anzahl von Funktionsaufrufen innerhalb von G_i. Dadurch wird $c_{max} := max(c_i)$ und $n_{max} := max(n_i)$ definiert.

Im Folgenden wird von einem linearen Aufwand der Inline Ersetzung ausgegangen.

4.5.1. Die ursprüngliche Inline-Heuristik

Das Verfahren benötigt den Callgraph, weshalb dieser immer berechnet werden muss. Der Aufwand dieser Berechnung sei $O(ccg)$. Das Finden aller Call-Knoten hingegen liegt in $O(n_{ops})$. Ebenso ist die Inline Ersetzung aller Funktionsaufrufe im Programm durch $O(n_{ops})$ beschränkt. Für jeden Graphen wird eine Prioritätsliste mit all seinen Call-Knoten angelegt. Da diese Prioritätsliste komplett durchlaufen wird und das Entfernen des Elements mit der höchsten Priorität logarithmischen Aufwand besitzt, ist dieses Vorgehen durch $O(c_{max} * log(c_{max}))$ beschränkt. Insgesamt ergibt sich im *average-case* für das Verfahren eine Komplexität von $O(ccg + n_{ops} * c_{max} * log(c_{max}))$.

Bei der Inline Ersetzung können durch zyklische und rekursive Funktionsaufrufe neue Call-Knoten entstehen. Diese werden in die Prioritätsliste eingefügt, was bei dem Binären Heap von LIBFIRM jeweils logarithmischen Aufwand benötigt. Die maximale Anzahl an neuen Knoten ist c_{max}. Für den *worst-case* ergibt sich damit eine Komplexität von $O(ccg + n_{ops} * c_{max}^2 * log(c_{max}))$.

4.5.2. Die verbesserte Inline-Heuristik

Im Gegensatz zu der ursprünglichen Inline-Heuristik muss bei der neuen Inline-Heuristik der Callgraph nicht berechnet werden. Dafür muss die Aufrufhäufigkeit jedes Graphen durch das *execfreq* Module bestimmt werden. Dies soll im Folgenden durch $O(exec)$ beschränkt sein. Weiterhin müssen wieder alle Funktionsaufrufe gefunden werden, was zu einer Komplexität von $O(n_{ops})$ führt. Da die Prioritätsliste nun global ist und vollständig abgearbeitet werden muss, impliziert dies eine amortisierte algorithmische Komplexität von $O(\chi * log(\chi))$, wobei χ die Anzahl aller Call-Knoten im Programm repräsentiert.

Die Komplexität der verbesserten Inline-Heuristik ist, abgesehen von hohen konstanten Faktoren, identisch im *average-* und *worst-case*. Dies liegt an der dynamischen Veränderung oder dem Entfernen von Elementen der Prioritätsliste, was eine amortisierte Komplexität von $O(log(\chi))$ besitzt und bis zu χ mal ausgeführt werden könnte. Die Inline Ersetzung ist linear, kann also durch $O(n_{max})$ beschränkt werden. Die Komplexität im *average-* und *worst-case* ist damit amortisiert $O(n_g * exec + n_{ops} + n_{max} * \chi^2 * log(\chi))$.

Die Komplexität liegt deutlich über dem *average-case* der ursprünglichen Heuristik. Inwiefern sich diese erhöhte Komplexität auf die Laufzeit der verbesserten Inline Optimierung auswirkt, wird in dem nächsten Kapitel betrachtet.

5. Evaluation

Im Folgenden werden die Verbesserungen der Inline Optimierung bewertet. Das zentrale Kriterium ist dabei die Laufzeit optimierter Programme. Ferner wird die Codegröße des erzeugten Programms betrachtet, da auch dies Aufschluss über die Qualität der Inline Optimierung gibt.

Die Anpassung der Inline Optimierung für die Verwendung von X10 war die hauptsächliche Motivation dieser Arbeit. Eine Bewertung erfolgt in Abschnitt 5.1. Daraufhin werden in Abschnitt 5.2 die Verbesserungen für die Sprache C anhand der SPEC CPU2000 Spezifikation evaluiert. Abschließend befindet sich in Abschnitt 5.3 ein Vergleich der Laufzeiten der ursprünglichen und der verbesserten Inline Optimierungsphase in libFirm.

Die Testplattform besteht aus einem Intel Core i5-750 Prozessor mit 2,67GHz, sowie 4GB DDR3-Ram. Als Betriebssystem wird Ubuntu 12.10 eingesetzt.

5.1. Evaluation der Inline Optimierung mit X10

Bei *ArrayBenchmark* handelt es sich um das folgende Programm:

```
import x10.util.IndexedMemoryChunk;

class Benchmark {
    public static def main(args : Array[String]) : void {
        val size = 100000000;
        val a = IndexedMemoryChunk.allocateZeroed[int](size);
        val b = IndexedMemoryChunk.allocateZeroed[int](size);
        val c = IndexedMemoryChunk.allocateZeroed[int](size);

        for (i in 0..(size-1)) {
            c(i) = a(i) * b(i);
        }
    }
}
```

Bei der Ausführung von *Array Benchmark* werden zuerst drei Arrays angelegt und auf Null gesetzt. Danach werden alle Element der Arrays durchlaufen, um das jeweilige Element

Tabelle 5.1.: Die Messergebnisse von ArrayBenchmark mit verschiedenen Inline Optimierungen

	Ohne	**Ursprünglich**	**Erweitert**	**Neue Heuristik**
Laufzeit	5,313 s	3,475 s	2,657 s	2,573 s
Lines-Of-Code	214454	329521	348609	233247

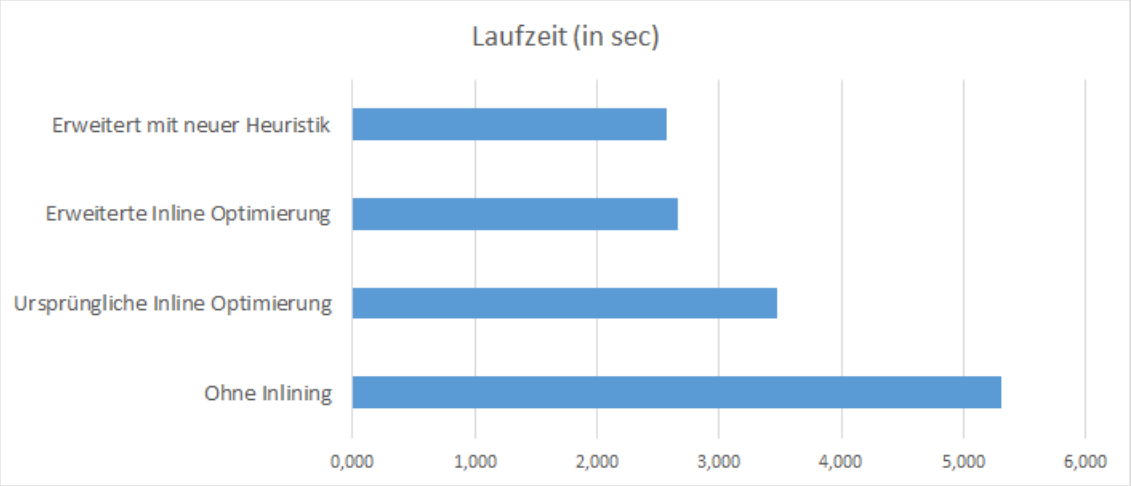
von *a* mit dem von *b* zu multiplizieren und in *c* zu speichern. In X10 ist die Multiplikation in der Schleife ein Funktionsaufruf mit den Objekten *a(i)* und *b(i)* als Parameter. Auch der Rückgabewert ist wiederum ein Objekt.

Aufgrund des Typs der Parameter und des Rückgabewertes ist die Ersetzung eines solchen Funktionsaufrufes mit der ursprünglichen Inline Ersetzung von libFirm nicht möglich. Die Erweiterungen aus Kapitel 3 ermöglichen es, solche Funktionsaufrufe zu ersetzen.

Wird mit der verbesserten Inline Optimierung *Array Benchmark* in Maschinencode übersetzt, so werden alle Funktionsaufrufe in der Schleife ersetzt. Übrig bleiben nur die notwendigen Multiplikationen und Load-/Store-Operationen.

Bei insgesamt 30 Messungen auf der Testplattform wurde eine minimale Laufzeit von 5,313 s für *ArrayBenchmark* ohne Inline Optimierung gemessen. Die Verwendung der ursprünglichen Inline Optimierung ergab eine Verbesserung von 34,59%. Durch die erweiterte Inline Ersetzung konnte dies auf 49,99% erhöht werden und bei zusätzlicher Verwendung der neuen Inline-Heuristik waren dies sogar 51,57%. Die Laufzeiten sind in Tabelle 5.1 und in Abbildung 5.1 dargestellt.

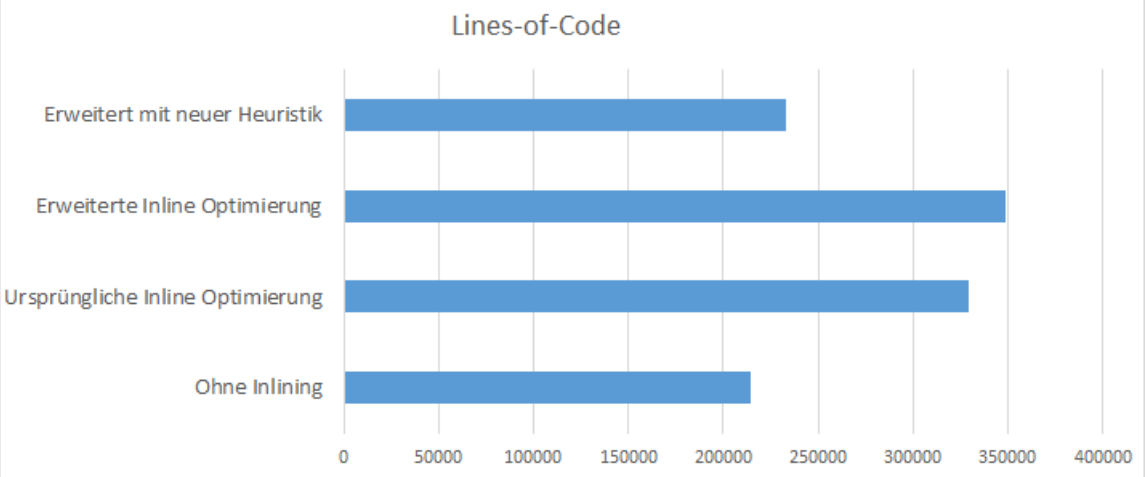
Abbildung 5.1.: Graphische Darstellung der Laufzeit von ArrayBenchmark



Bei der Übersetzung eines X10 Programms mit dem ”X10 Compiler for Invasive Architectures” wird das gesamte Programm, inklusive der Standardbibliothek, in eine Datei übersetzt. Dadurch entsteht bereits ohne Inline Optimierung eine Ausgabe in X86 Assembler mit 214454 Zeilen Code. Die ursprüngliche Inline Optimierung führt zu einer Vergrößerung um 53,66%, mit der erweiterten Inline Ersetzung sind dies 62,56%. Allerdings kann bei zusätzlicher Verwendung der neuen Inline-Heuristik die Vergrößerung auf nur 8,76% reduziert werden. Die jeweilige Anzahl an Lines-Of-Code ist in Tabelle 5.1 und in Abbil-

dung 5.2 zu finden.

Abbildung 5.2.: Graphische Darstellung der erzeugte Lines-Of-Code von ArrayBenchmark



Die Messungen belegen, dass im Vergleich mit der ursprünglichen Inline Optimierung nicht nur die Laufzeit um 16,98% erhöht, sondern gleichzeitig auch eine Verkleinerung von 44,89% der entstehenden Codegröße erreicht werden konnte.

5.2. Evaluation anhand der SPEC CPU2000 Spezifikation

Die Standard Performance Evaluation Corporation (SPEC) (vgl. [SPE]) erstellt und verwaltet eine Menge von standardisierten Benchmarks. CPU2000 dient zur Evaluation der CPU Geschwindigkeit. Dabei wird zwischen Ganzzahl- und Gleitkomma-Benchmarks unterschieden, welche jeweils eine Reihe von Programmen enthalten. Eine Beschreibung von CPU2000 befindet sich in [Hen00].

Um die Inline Optimierung von libFirm mit SPEC CPU2000 zu evaluieren, wird das *cparser* Front-End verwendet. Bei allen Messungen werden identische Versionen von *cparser* und libFirm verwendet, nur die Inline Optimierung wird ausgetauscht. Jedes Benchmark wurde dabei in jeder Einstellung genau 20 Mal auf der Testplattform ausgeführt, wobei die SPEC Evaluierungssoftware bereits drei Läufe für jeden Aufruf ausführt.

Die folgenden Messergebnisse bezeichnen die *Base Runtime* des besten Durchlaufs eines Benchmarks. Bei den angegebenen Mittelwerten handelt es sich um das arithmetische Mittel bei den Laufzeiten oder um das geometrische Mittel bei den prozentualen Unterschieden.

Gemessen wurde zuerst mit den üblichen Einstellungen, im Anschluss daran wurden Messungen mit *Profile-guided optimization* durchgeführt. Abschließend wurden die Größen der erstellten Benchmark Programme verglichen.

5.2.1. SPEC CPU2000 Messungen

Die Ergebnisse der Evaluation mit SPEC CPU2000 sind in Tabelle 5.2 aufgelistet.

Obwohl in Einzelfällen auch eine längere Laufzeit gemessen wurde, zeigen die Ergebnisse eine durchschnittliche Verbesserung von 4,25%.

Tabelle 5.2.: Ergebnisse von SPEC CPU2000. Die Werte geben die *Base Runtime* und den prozentualen Unterschied an

Benchmark	**Ursprünglich**	**Verbessert**	**Unterschied**
177.mesa	86,6	86,1	0,58%
179.art	47,0	41,5	10,84%
183.equake	50,4	50,0	0,80%
188.ammp	130	129	0,78%
164.gzip	80,9	82,3	-1,70%
175.vpr	76,9	71,2	8,01%
176.gcc	42,7	41,5	2,89%
181.mcf	37,6	34,9	7,74%
186.crafty	37,4	37,5	-0,27%
197.parser	109,0	106,0	2,83%
253.perlbmk	91,1	90,0	1,22%
254.gap	49,8	45,8	8,73%
255.vortex	80,8	70,9	12,55%
256.bzip2	92,0	86,2	6,73%
300.twolf	124,0	120,0	3,33%
Mittelwert	75,6	72,9	4,25%

Die Messungen belegen damit, dass die durchschnittliche Laufzeit der verbesserten Inline Optimierungen deutlich kürzer ist.

5.2.2. SPEC CPU2000 Messungen mit Profile Informationen

libFirm besitzt die Option zur Verwendung von *Profile-guided optimization*. Dazu wird ein Übersetzungsvorgang durchgeführt, bei welchem Instrumentierung, das Einfügen von Analysecode, ausgeführt wird. Anschließend muss das Programm mit mehreren Testläufen ausgeführt werden, bei welchem Profile Informationen gesammelt werden. Bei einer weiteren Übersetzung werden diese Informationen verwendet.

Solange die gesammelten Informationen für die spätere Benutzung des Programms repräsentativ sind, können Entscheidungen beim Übersetzen besser getroffen werden. Dies gilt auch für die neue Inline-Heuristik, welche durch die Verwendung des *execfreq* Moduls diese Profile Informationen nutzt.

SPEC CPU2000 bietet Unterstützung für diese Art der Optimierung. Dabei sind eigene Datensätze für die Durchführung von Testläufen vorhanden. Tabelle 5.3 enthält die Messergebnisse bei Verwendung von *Profile-guided optimization*.

Die Ausführung von 176.gcc war, unabhängig von der Inline Optimierung, auf Grund eines Fehlers beim Übersetzen nicht möglich. Die durchschnittliche Verbesserung der neuen Inline Optimierung beträgt 4,64%. Dabei gab es nur bei 164.gzip eine Verschlechterung um -0,49%, ansonsten wurden Verbesserungen bis zu 14,07% erreicht.

Im Vergleich zu den Messwerten aus Tabelle 5.2 ist eine generelle Verbesserung zu erkennen. Da dies sowohl bei der ursprünglichen, als auch bei der verbesserten Inline Optimierung auftritt, ist diese Veränderung hauptsächlich auf andere Optimierungen zurückzuführen. Die verbesserte Inline Optimierung kann durch Verwendung von *Profile-guided optimization* eine zusätzliche Verbesserung von 0,39% erreichen.

Tabelle 5.3.: Ergebnisse von SPEC CPU2000 mit *Profile-guided optimization*. Die Werte geben die *Base Runtime*, sowie den prozentualen Unterschied an

Benchmark	**Ursprünglich**	**Verbessert**	**Unterschied**
177.mesa	69,5	68,3	1,76%
179.art	41,5	41,4	0,24%
183.equake	43,3	39,9	10,18%
188.ammp	116	114	1,75%
164.gzip	81,5	81,9	-0,49%
175.vpr	77	67,5	14,07%
176.gcc	-	-	-
181.mcf	37,2	35	6,29%
186.crafty	39,5	37,2	6,18%
197.parser	99,4	97	2,47%
253.perlbmk	88,9	88	1,02%
254.gap	50,6	50,3	0,60%
255.vortex	78,2	65,9	18,66%
256.bzip2	78,1	74,9	4,27%
300.twolf	105	105	0,00%
Mittelwert	71,8	69,0	4,64%

5.2.3. Vergleich der Größe der erzeugten Programme

In Tabelle 5.4 sind alle getesteten SPEC CPU2000 Benchmarks mit ihrer jeweiligen Programmgröße in Bytes dargestellt. Für alle Benchmarks gilt, dass sie durch Verwendung der verbesserten Inline Optimierung nicht größer geworden sind. Im Durchschnitt wird eine Verringerung von 12,72% erreicht.

Tabelle 5.4.: Größe der erstellten SPEC CPU2000 Benchmark Programme in Bytes

Benchmark	**Ursprünglich**	**Verbessert**	**Unterschied**
177.mesa	789099	678710	16,26%
179.art	25668	25668	0%
183.equake	29803	25707	15,93%
188.ammp	165978	157786	5,19%
164.gzip	61146	57095	7,10%
175.vpr	201909	190238	6,13%
176.gcc	2396075	2040862	17,41%
181.mcf	21144	17048	124,03%
186.crafty	254578	250482	1,64%
197.parser	235299	198435	18,58%
253.perlbmk	952713	740161	28,72%
254.gap	693726	558505	24,21%
255.vortex	890375	726535	22,55%
256.bzip2	63907	59811	6,85%
300.twolf	242068	237972	1,72%
Mitelwert	468233	397668	12,72%

Inwiefern die Verbesserung der Laufzeiten und die geringere Programmgrößen korrelieren ist nicht bekannt. Es ist aber wahrscheinlich, dass ein Großteil der verbesserten Laufzeiten

ein Resultat der kleineren Programmgrößen ist.

5.3. Laufzeit der Inline Optimierungsphase

Auf Grund der gesteigerten Komplexität der neuen Inline-Heuristik ist mit einem Anstieg der Laufzeit der Inline Optimierungsphase von libFirm zu rechnen. In Tabelle 5.5 sind diese Laufzeiten bei der Kompilierung mehrerer Übersetzungseinheiten aufgelistet. Für jede Übersetzungseinheit ist die Laufzeit der Inline Optimierungsphase und die Laufzeit aller Optimierungen in libFirm jeweils für die ursprüngliche und verbesserte Inline Optimierung angegeben.

Tabelle 5.5.: Vergleich der Laufzeiten (in ms) der Inline Optimierungsphase von libFirm

	Ursprünglich	**Gesamt**	**Verbessert**	**Gesamt**
bzip2				
blocksort.c	8,38	673,7	28,34	635,05
huffmann.c	0,97	99,74	2,90	101,41
compress.c	9,32	971,65	48,90	971,74
decompress.c	8,56	2016,26	440,82	2450,90
bzlib.c	37,83	1623,52	40,31	1346,4
gzip				
gzip.c	19,8	970,38	54,94	860
deflate.c	3,09	199,15	8,51	205,53
trees.c	8,63	457,03	11,09	430,7
bits.c	0,95	74,53	3,39	69,39
unzip.c	1,07	105,04	1,90	106,01
inflate.c	10,07	530,82	16,24	412,17
util.c	2,53	138,08	3,41	133,16
unpack.c	1,52	100,45	2,43	101,81
unlzh.c	10,34	395,91	16,97	401,25

Ein beträchtlicher Anstieg der Laufzeiten der verbesserten Inline Optimierung ist sofort zu erkennen. Insbesondere bei *decompress.c* ist die Laufzeit dramatisch angestiegen.

Bei der ursprünglichen Inline-Heuristik ist der Anteil der Inline Optimierung des gesamten Optimierungsvorgangs 1,41%, bei der verbesserten Inline Optimierung sind dies 3,93%.

Allerdings führt die neue Inline Optimierung auch oft zu einer insgesamt kürzeren Laufzeit aller Optimierungen. So gibt es eine Verbesserung der gesamten Laufzeit um durchschnittlich 4,00%. Möglich ist, dass diese Verbesserung aus den verkleinerten Codegrößen resultieren. Denn eine geringere Codegröße impliziert weniger Firm Graphen und Operationen, wodurch ein geringerer Arbeitsaufwand für weitere Optimierungen folgt.

Die Laufzeit der Inline Optimierungsphase ist wie erwartet angestiegen. Überraschenderweise führt dies im Mittel aber zu einer Verbesserung der Laufzeit aller Optimierungen in libFirm.

6. Fazit

Mehrere Erweiterungen der Inline Ersetzung von libFirm ermöglichen das Ersetzen von zuvor nicht unterstützten Funktionsaufrufen. Auf Grund dessen konnte bei der Evaluation von X10 Programmen eine erhebliche Performanzsteigerung gezeigt werden. Allerdings führte dies gleichzeitig zu einer weiteren Vergrößerung der erzeugten Programme, was bei der bereits sehr großen Ausgabe des „X10 Compilers for Invasive Architectures“ durchaus relevant ist.

Die Entwicklung der neuen Inline-Heuristik ermöglicht es die verbesserte Inline Ersetzung zu verwenden, da diese weitaus kleinere Programme erzeugt. Durch die verbesserte Inline Optimierung werden die erzeugten Programme nicht nur kleiner, sondern auch schneller. So erreicht die verbesserte Inline Optimierung eine um 16,98% bessere Laufzeit und gleichzeitig eine Verkleinerung der Programmgröße um 44,89% bei dem *ArrayBenchmark* X10 Programm.

Aber auch die Optimierung von C Programmen liefert bessere Resultate. Die erreichten Verbesserungen bei den SPEC CPU2000 Benchmarks befinden sich bei 4,25%, was die Erwartungen übertroffen hat.

Die Verbesserungen führen zu einem Anstieg der Laufzeit der Inline Optimierungsphase von libFirm. Zu gleich verringert die verbesserte Inline Optimierung aber im Mittel die Laufzeit aller Optimierungen von libFirm, wodurch dieser Anstieg gerechtfertigt ist.

Weitere Erweiterungen der Inline Ersetzung sind zwar vorstellbar, würden sich jedoch auf Ausnahmefälle beschränken. Signifikante Performanzsteigerungen sind deshalb nicht zu erwarten. Im Gegensatz dazu lässt ein heuristisches Verfahren immer noch weitere Verbesserungen zu. Vorausgesetzt das vorgestellte Verfahren wird nicht grundlegend verändert, sind wesentliche qualitative Verbesserungen allerdings eher unwahrscheinlich. Daraus lässt sich folgern, dass die Inline Optimierung von libFirm erfolgreich verbessert und vervollständigt wurde.

Literaturverzeichnis

[ALSU08] Alfred V. Aho, Monica S. Lam, Ravi Sethi und Jeffrey D. Ullman: *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium, 2. aktualisierte Auflage, 2008.

[b2f] *Bytecode2Firm*. `https://github.com/MatzeB/bytecode2firm`.

[BBMZ12] Matthias Braun, Sebastian Buchwald, Manuel Mohr und Andreas Zwinkau: *An X10 Compiler for Invasive Architectures*. Technischer Bericht 9, Karlsruhe Institute of Technology, 2012.

[BBZ11] Matthias Braun, Sebastian Buchwald und Andreas Zwinkau: *Firm—A Graph-Based Intermediate Representation*. Technischer Bericht 35, Karlsruhe Institute of Technology, 2011.

[CP95] Cliff Click und Michael Paleczny: *A simple graph-based intermediate representation*. SIGPLAN Not., 30(3):35–49, März 1995, ISSN 0362-1340.

[cpa] *cparser C-Frontend*. `https://github.com/MatzeB/cparser`.

[Fre99] Michael L. Fredman: *On the efficiency of pairing heaps and related data structures*. J. ACM, 46(4):473–501, Juli 1999, ISSN 0004-5411.

[FSST86] Michael Fredman, Robert Sedgewick, Daniel Sleator und Robert Tarjan: *The pairing heap: A new form of self-adjusting heap*. Algorithmica, 1(1):111–129, November 1986.

[Hen00] John L. Henning: *SPEC CPU2000: Measuring CPU Performance in the New Millennium*. Computer, 33(7):28–35, Juli 2000, ISSN 0018-9162.

[lib] *The libFirm library*. `http://www.libfirm.org`.

[Lin02] Götz Lindenmaier: *libFIRM – A Library for Compiler Optimization Research Implementing FIRM*. Technischer Bericht, 2002.

[MS91] Bernard M.E. Moret und Henry D. Shapiro: *An empirical analysis of algorithms for constructing a minimum spanning tree*. In: *Algorithms and Data Structures*, Band 519 der Reihe *Lecture Notes in Computer Science*, Seiten 400–411. Springer Berlin Heidelberg, 1991, ISBN 978-3-540-54343-5.

[Pet05] S. Pettie: *Towards a final analysis of pairing heaps*. In: *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, Seiten 174 – 183, oct. 2005.

[RWZ88] B. K. Rosen, M. N. Wegman und 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, Seiten 12–27, New York, NY, USA, 1988. ACM, ISBN 0-89791-252-7.

[SPE] *Standard Performance Evaluation Corporation*. `http://www.spec.org/`.

[SV87] John T. Stasko und Jeffrey Scott Vitter: *Pairing heaps: experiments and analysis.* Commun. ACM, 30(3):234–249, März 1987, ISSN 0001-0782.

[Tra01] Martin Trapp: *Optimierung objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen.* Dissertation, Oct. 2001.

Anhang

A. Tabellen

Name	Wert	Beschreibung
Summanden		
load_store_param_weight	12	Load/Store Parameter einer lokalen Adresse
const_sel_param_weight	6	Konstante Sel Parameter einer lokalen Adresse
all_const_param_weight	250	Alle Parameter sind konstant
one_call_to_static_weight	750	Die Funktion wird nur einmal aufgerufen
recursive_call_weight	-150	Rekursiver Funktionsaufruf
leaf_function_weight	800	Die Funktion enthält keine weiteren Aufrufe
one_block_function_weight	50	Die Funktion besteht aus einem Block
small_function_weight	1000	Die Funktion enthält wenig Operationen
Faktoren		
register_param_weight	2	Für die Anzahl an Parameter in Registern
stack_param_weight	5	Für die Anzahl an Parameter im Stack
inner_loop_depth	1000	Für die Schleifentiefe
block_function_weight	-4	Anzahl an Blöcken
large_function_weight	-3	Anzahl an Operationen
alloca_function_weight	-350	Für jede Alloca-Operation

Tabelle A.1.: Statische und dynamische Gewichte der neuen Inline-Heuristik in libFirm

Die Typen T_{EAG} sind wie folgt festgelegt:

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

Die Funktionssymbole Σ_{EAG}:

Name	**Typ**	**Beschreibung**
Add	$B \times Num \times Num \to Num$	Addition
Alloc	$B \times M \times Num \to M \times P$	Allocate memory
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	$X_0 \times \ldots \times X_n \to B$	Basic block
Call	$B \times Num_0 \times \ldots \times Num_k \to Num_0 \times \ldots \times Num_k \times M$	Function call
Cmp	$B \times Num \times Num \to b_0 \times \ldots \times b_{15}$	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
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	$B \times Num \times Num \to$	Bitwise OR
Phi	$B \times Num_0 \times \ldots \times Num_n \to X$	ϕ-function
Proj	$B \times T \to Num$	Projection node
Return	$B \times M \times Num_0 \times \ldots \times Num_n \to X$	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	$B \times M \times P \times Num \times M$	Write to memory
Sub	$B \times Num \times Num \to Num$	Subtraction
SymConst	$B \times P$	Symbolical constant
Sync	$B \times M_0 \times \ldots \times M_n \to M$	Memory barrier
Unknown	$\to Any$	Undefined value

Tabelle A.2.: FIRM Knoten-Typen T_{EAG} und Signaturen Σ_{EAG} (Entnommen aus [BBZ11])