1. Einführung

Mit der steigenden Technisierung unseres Lebens steigt die Nachfrage nach eingebetteten Systemen stetig an. Ein Leben ohne Mobiltelefone, Navigationssysteme und andere tragbare computergestützte Systeme ist für viele heute nicht mehr vorstellbar.

Eine der Hauptforderung der Kunden an diese eingebetteten Systeme ist, dass sie nicht nur immer leistungsfähiger werden, sondern dabei auch immer kleiner werden und längere Laufzeiten besitzen. Längere Laufzeiten durch größere Batterien und Akkumulatoren stehen allerdings im Widerspruch zur Bestrebung, die Geräte möglichst kein zu bauen. Da größere Geräte von den Benutzern nicht akzeptiert werden, bleibt den Entwicklern nur die Möglichkeit, den Stromverbrauch der Geräte so weit wie möglich zu reduzieren.

Neben der Antenne und dem Prozessor ist der Hauptspeicher dabei das Bauteil, das den größten Energiebedarf besitzt. In einem handelsüblichen Mobiltelefon entfällt ca. 20% des gesamten Energiebedarfs auf den Hauptspeicher (siehe Abbildung 1.1). Je weniger Speicher die Programme auf einem eingebetteten System benötigen, desto weniger Speicher muss verbaut werden. Damit lässt sich Energie sparen und die Produktionskosten werden reduziert.

Da die Benutzer aber außerdem immer umfangreichere und leistungsfähigere Programme für die Systeme wünschen, stehen wir vor einem Problem. Leistungsfähige und damit komplizierte Programme lassen sich nur in Hochsprachen mit vertretbaren Kosten entwickeln. Diese Hochsprachen verfügen aber über mehrere Abstraktionsschichten. Diese machen zwar die Entwicklung einfacher, schneller und weniger fehleranfällig, haben aber das Problem, dass sie mehr Speicher benötigen. Um den Spagat zwischen gesteigerter Produktivität und Wartbarkeit auf der einen Seite und minimalen Code auf der anderen Seite zu schaffen, wird in der Industrie eine manuelle Nachoptimierung des Assembler-Codes durchgeführt. Das Problem

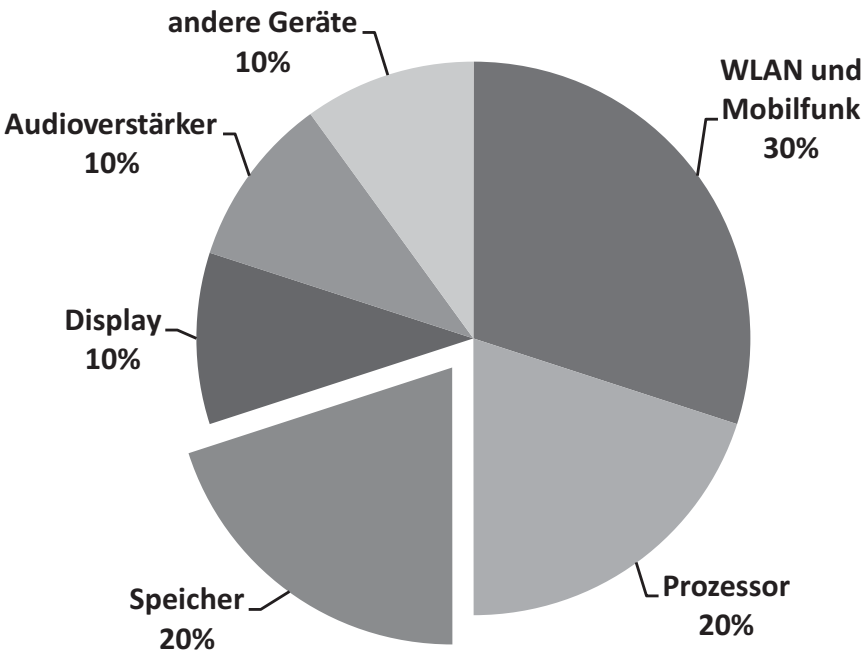

Abbildung 1.1.: Typischer Energieverbrauch eines Mobiltelefons [93]

bei diesem Ansatz ist, dass diese manuelle Optimierung sehr kompliziert und damit fehleranfällig ist. Außerdem kann sie nicht in den Entwicklungsprozess integriert werden, sondern muss nachgeschaltet werden.

Unser Ziel muss somit sein, ein automatisches Verfahren zu entwickeln, das es möglich macht, in beliebigen Hochsprachen zu entwickeln und dennoch Programme mit einem minimalen Speicherverbrauch zu erzeugen. Dieses Verfahren muss dabei schnell genug sein, damit es direkt in die Entwicklung integriert werden kann, um gegebenenfalls auftretende Probleme so früh wie möglich erkennen zu können.

1.1. Prozedurale Abstraktion

In Rahmen dieser Arbeit haben wir ein System zur graphbasierten prozeduralen Abstraktion geschaffen, das es ermöglicht, unabhängig von der verwendeten Hochsprache die Code-Größe auf ein Minimum zu reduzieren. Dabei haben wir mehrere Probleme bestehender Ansätze gelöst. Abbildung 1.2 gibt

einen schematischen Überblick über die verschiedenen Phasen, die während der Programmkompaktifizierung durchlaufen werden müssen.

Bisherige Ansätze nehmen das Programm zusammen mit Meta-Informationen und bereiten die Daten soweit auf, dass das Programm optimiert werden kann, ohne seine Semantik zu verändern. Auf dem so aufbereiteten Programm wird anschließend nach häufigen mehrfach vorkommenden Programm-Code gesucht. Diese doppelten Programmteile, oder auch Code-Fragmente genannt, werden gesammelt und anschließend gesammelt der Duplikatauswahl übergeben. Diese wählt ein Fragment aus, das in einem nächsten Schritt ausgelagert werden soll. Bei der Auslagerung werden alle Vorkommen des mehrfach vorkommenden Programm-Codes durch Sprünge oder Funktionsaufrufe an eine einzige Stelle im Programm ersetzt. Damit wird die, im Programm vorhandene, Redundanz in jedem Auslagerungsschritt verringert. Nachdem das Fragment ausgelagert wurde, beginnt die Suche nach häufigem Code erneut von vorne. Das Ergebnis wird also iterativ immer weiter verbessert, bis keine weiteren Redundanzen mehr im Code gefunden werden können. Damit ist die prozedurale Abstraktion abgeschlossen und das Programm kann in einer Nachverarbeitung in kompaktifizierter Form zurückgeliefert werden.

Bei den in Abbildung 1.2 grau hinterlegten Bereichen der prozeduralen Abstraktion wird bei den derzeit verwendeten Ansätzen viel Potential verschenkt. Die Notwendigkeit von Meta-Informationen für die Vorverarbeitung schränkt das Einsatzgebiet der prozeduralen Abstraktion stark ein. Diese Informationen müssen von speziell modifizierten Übersetzern und Bindern zur Verfügung gestellt werden. Eine Anpassung der Werkzeugkette ist aber nur möglich, wenn der Quelltext der Programme frei verfügbar ist. Dies ist nur in seltenen Fällen im Bereich der eingebetteten Systeme der Fall. Unser Ansatz verzichtet auf diese Meta-Informationen und versucht alle nötigen Informationen für die Optimierung des Programms durch verschiedene Analysen selbst zu rekonstruieren. Damit ist unser Ansatz nicht an eine bestimmte Werkzeugkette gebunden und kann unabhängig von Übersetzern und Bindern eingesetzt werden.

Im Gegensatz zur Suffixbaum-Suche nach häufigen Code-Fragmenten ist unser graphbasierter Ansatz fähig, unabhängig von der konkreten Reihenfolge, in der Instruktionen im Programm vorkommen, semantisch äquivalente Ausdrücke zu identifizieren. Dies ermöglicht, wesentlich größere Einsparungen zu erzielen. Moderne Übersetzer ändern die Reihenfolge von Instruk-

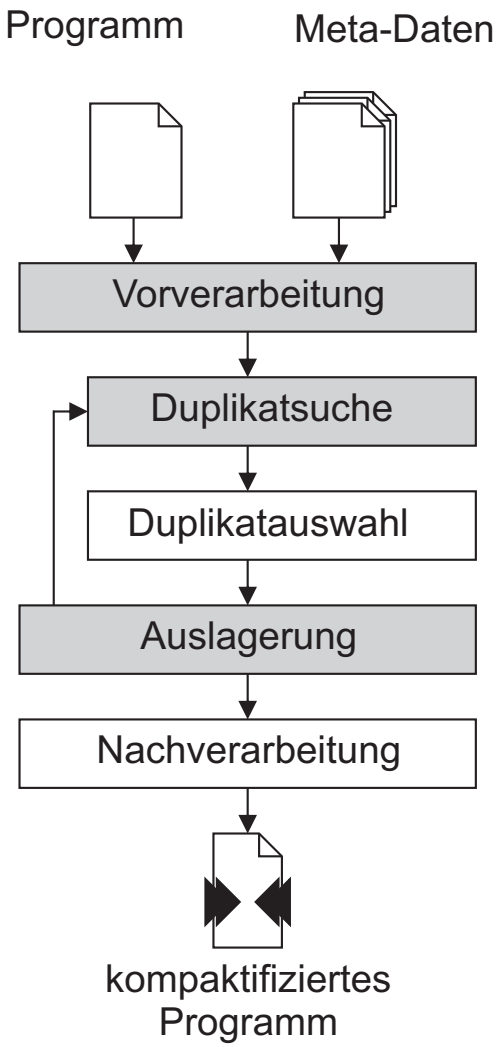

Abbildung 1.2.: Schematische Übersicht über die verschiedenen Phasen der Programmkompaktifizierung

tionen, um so Recheneinheiten, Fließbänder und Zwischenspeicher optimal auszulasten.

Zusätzlich kann ein Übersetzer unterschiedliche Register in äquivalenten Fragmenten nutzen, um damit die Speicherung von Registerwerten in den Hauptspeicher zu vermeiden. Aufgrund der wesentlich größeren Latenz des Hauptspeichers, im Vergleich zum Hauptprozessor, können damit beachtliche Geschwindigkeitssteigerungen erzielt werden. Da bekannte Ansätze meist nur Instruktionen als gleich erkennen, wenn diese dieselben Register verwenden, verschenken sie großes Optimierungspotential, da sie nicht mit unterschiedlichen Registern zurechtkommen.

Wir lösen dieses Problem, indem wir vor der Suche von den konkreten Registern abstrahieren und eine kanonische Darstellung für die Register wählen. Damit sind wir in der Lage, das Optimum aus der prozeduralen Abstraktion herauszuholen. Um Code-Fragmente mit nicht einheitlicher Registerverwendung gemeinsam auslagern zu können, müssen wir bei der Auslagerung eine optimale (minimale) Abbildung unterschiedlicher Register auf eine einheitliche Form finden. Zu diesem Zweck haben wir ein mathematisches Verfahren (die sogenannte *Ungarische-Methode*), das ursprünglich zur Zuteilung von Versorgungsgütern in der Armee entwickelt wurde, soweit angepasst, dass wir damit eine optimale Registerabbildung berechnen können.

Um den vollständigen Suchraum bearbeiten zu können, mussten wir einen vollständig neuen Graphmining-Algorithmus entwickeln. Dieser Algorithmus ermöglicht es gerichtete, azyklische Graphen effizient zu durchsuchen ohne dabei Arbeit mehrfach zu verrichten.

Die meisten bisherigen Ansätze begnügen sich damit, auf Grundblockebene nach häufigen Code-Fragmenten zu suchen. Da Grundblöcke im allgemeinen nur aus wenigen Instruktionen bestehen, vereinfacht dies das Problem. Um aber den größtmöglichen Effekt erzielen zu können, dürfen wir die einzelnen Grundblöcke nicht getrennt voneinander betrachten. Wir müssen alle Grundblöcke, einer Funktion, zusammen betrachten und auch über die Grenzen der Blöcke hinweg häufigen Code auslagern. Um die Suche über alle Grundblöcke hinweg dabei möglichst schnell und effizient zu gestalten, haben wir Verfahren entwickelt, die überflüssige Kanten auf den Graphen entfernen und die Graphen automatisch zerteilen, ohne dabei die Optimierung negativ zu beeinflussen. Dadurch sind wir in der Lage, alle Grenzen bisheriger Ansätze hinter uns zu lassen und die größtmögliche Einsparung zu erzielen. Gleichzeitig können wir allerdings die Systemanforderungen für unser

Verfahren so niedrig halten, dass jeder handelsübliche PC in der Lage ist, graphbasierte prozedurale Abstraktion durchzuführen.

1.2. Kapitelübersicht

In Kapitel 2 behandeln wir zunächst verwandte Themen zur Reduzierung der Größe von Programm-Code, insbesondere Ansätze zur Programmkomprimierung und -kompaktifizierung. Desweiteren geben wir einen Einblick in bisherige Ansätze zur prozeduralen Abstraktion und grenzen unseren graphbasierten Ansatz von den bisherigen Arbeiten ab.

Aufbauend auf der allgemeinen Einführung in den Themenkomplex der Programmkompaktifizierung erklären wir in Kapitel 3, wie Programme zerlegt werden können, ohne dabei auf spezielle Zusatzinformation angewiesen zu sein. Zusätzlich zur Dekomposition von Programmen geht dieses Kapitel auch darauf ein, wie die für prozedurale Abstraktion nötigen Informationen, wie Sprungziele oder Funktionsgrenzen, rekonstruiert werden können.

Anschließend geben wir in Kapitel 4 einen allgemeinen Überblick über die Terminologie und den grundsätzlichen Aufbau von Graphminern. Wir werden die zurzeit besten Ansätze für die Suche nach häufigen Fragmenten kurz erläutern und miteinander vergleichen. Zusätzlich werden wir unsere Erweiterungen an diesen bestehenden Algorithmen erläutern, um diese für die prozedurale Abstraktion nutzbar zu machen.

Ausgehend von den allgemeinen Graphminern in Kapitel 4 wird in Kapitel 5 ein neuer von uns entwickelter Algorithmus zur Suche in gerichteten unzusammenhängenden Graphen vorgestellt. Dieser neue Algorithmus wurde von uns speziell für die besonderen Anforderungen der prozeduralen Abstraktion entwickelt.

Um das volle Potential unseres Ansatzes nutzen zu können, müssen wir sowohl den Speicherbedarf als auch die Laufzeit unseres Optimierers reduzieren. Kapitel 7 stellt dazu verschiedene Optimierungen vor. Anschließend stellen wir die verwendeten Benchmark-Programme vor, mit deren Hilfe wir die verschiedenen Phasen der Entwicklung vermessen. Mit diesen Programmen können wir die Überlegenheit unseres neuen Ansatzes nachweisen, indem wir zeigen, wie unser Verfahren vorhandene Code-Redundanzen ver-

ringert und sich dabei, durch verschiedene Optimierungen, die Laufzeit und der Speicherbedarf unseres Ansatzes reduziert.

Schließlich fassen wir im letzten Kapitel die Arbeit zusammen und geben einen Ausblick auf mögliche zukünftige Forschungsthemen.