TECHNISCHE UNIVERSITÄT DORTMUND
FAKULTÄT FÜR INFORMATIK

technische universität dortmund

Arthur Pyka

Multikriterielle Exploration von Compileroptimierungen und Cacheparametern

Diplomarbeit

21. Februar 2011

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl Informatik XII
(Technische Informatik und Eingebettete Systeme)

Gutachter:
Prof. Dr. Peter Marwedel
Dipl-Inform. Sascha Plazar

GERMANY · D-44221 DORTMUND

Ich danke allen voran meinem Betreuer Sascha Plazar, der mit viel Engagement und Geduld die Entstehung dieser Arbeit begleitet hat und mir eine unschätzbare Hilfe war. Danken möchte ich auch meiner Freundin und meinen Freunden, die besonders in der letzten Phase viel Verständnis für meine lange Arbeitszeit gezeigt haben.
Zuletzt gilt ein großer Dank meiner Familie, die mich während meines Studium unterstützt und mir so manche Brücke gebaut hat.

Inhaltsverzeichnis

1 Einleitung

Eingebettete Systeme (ES) durchdringen unser Leben in einer erstaunlichen Tiefe und bleiben dabei oft unbemerkt, in vielen Fällen sogar unsichtbar. Sie sind in größere Produkte integriert und verbergen sich unscheinbar hinter deren Funktionalität. Die Menschen tragen sie in Mobiltelefonen mit sich, fahren sie in Autos durch die Gegend und sind privat und im Beruf vielfach auf sie angewiesen, ohne sich dessen immer bewusst zu sein. Dabei stellt die Verbreitung eingebetteter Systeme die normaler Desktop PCs seit langem in den Schatten. All das macht ES zu einem spannenden und anspruchsvollen Gebiet wissenschaftlicher Forschung, das besondere Herausforderungen mit sich bringt.
Eingebettete Systeme unterscheiden sich vielfach von herkömmlichen Desktop-Systemen. Als Teil eines speziellen Produktes, sollten sie möglichst klein und kostengünstig produziert werden können. Das schlägt sich in einer beschränkten Rechenkapazität und geringem verfügbaren Speicherplatz nieder. Bei mobilen Geräten, die über eine Batterie betrieben werden, muss der begrenzten Energiequelle Rechnung getragen werden. Zusätzlich stellt die Einhaltung von harten und weichen Echtzeitanforderungen eine Herausforderung dar. Diese Besonderheiten müssen bei der Entwicklung von Hardware und Software für eingebettete Systeme berücksichtigt werden.

Software für eingebettete Systeme muss verschiedenen Anforderungen genügen. Neben einem geringen Energieverbrauch und einer geringen Codegröße, spielt die Laufzeit eine entscheidende Rolle. Die durchschnittliche Laufzeit (*Average Case Execution Time - ACET*) muss in einem für die Funktion des Systems angemessenen Rahmen liegen. Die maximale Ausführungszeit (*Worst Case Execution Time - WCET*) darf die (harten) Echtzeitschranken nicht überschreiten.
Bei der Einhaltung dieser speziellen Anforderungen spielen Compiler eine herausragende Rolle. Bei der Übersetzung von Quellcode in den Maschinencode bieten sich zahlreiche Ansätze für Optimierungen, die unterschiedliche Auswirkungen auf den Code haben. Der Wunsch eines Entwicklers mag es sein, nach allen erwähnten Kriterien(Laufzeit, Codegröße, Energie) zu optimieren, um den Anforderungen an das System am besten zu genügen. Jedoch wirken sich die Compileroptimierungen nicht gleichermaßen auf alle Kriterien aus. Eine Verringerung der Laufzeit hat manchmal eine höhere Codegröße zur Folge, während weniger Codegröße nicht unbedingt weniger Energieverbrauch bedeutet. In der Realität muss ein Kompromiss gefunden werden, also eine Auswahl von Optimierungen die den Anforderungen hinreichend genügt. Aber wie sieht diese Auswahl aus? Die Vielzahl von zur Verfügung stehenden Compileroptimierungen macht das Finden der optimalen Auswahl zu einem scheinbar unlösbaren Problem. Zumal die Daten zur ACET, WCET und zum Energieverbrauch der Software nicht direkt zu Verfügung stehen sondern erst nachträglich ermittelt werden können.

In Hardware für eingebettete Systeme wird Cachespeicher eingesetzt um eine Brücke zwischen immer schnelleren Prozessoren und langsamen Hauptspeichern zu schlagen. Bei der Entwicklung eines Systems stehen für den Cache mehrere Parameter (Cachegröße, Cachezeilengröße und Assoziativität) zur Konfiguration zu Verfügung. Diese Parameter stellen eine weitere Möglichkeit dar, Laufzeit und Energieverbrauch der Software zu beeinflussen. Ein Entwickler muss die beste Konfiguration für sein System wählen und auch dabei die möglicherweise konträren Auswirkungen auf Laufzeit und Energieverbrauch berücksichtigen.

Compileroptimierungen und Cacheparameter sind wichtige Varianten bei der Entwicklung eingebetteter Systeme mit schwer vorhersehbaren Auswirkungen auf Laufzeit und Energieverbrauch. Im folgenden werden diese Auswirkungen systematisch untersucht und aus den Resultaten Schlussfolgerungen gezogen, die für die Entwicklung eingebetteter Systeme hilfreich sind.

1.1 Ziele der Diplomarbeit

Diese Diplomarbeit untersucht die Auswirkungen von Compileroptimierungen und Cacheparametern auf die WCET, ACET und den Energieverbrauch von Software für eingebettete Systeme am Beispiel der Zielarchitektur ARM7TDMI. Ziel der Arbeit ist es zum Einen den Zusammenhang zwischen den genannten Kriterien zu ergründen:

- Verändern sich ACET und WCET synchron zueinander?
  Eine Verringerung der ACET muss nicht unbedingt auch die maximale Ausführungszeit reduzieren. Zudem stellt sich die Frage ob sich die WCET genau so stark beeinflussen lässt wie die ACET.
- Bewirkt eine Verringerung der einzelnen Laufzeiten auch einen sinkenden Energieverbrauch?
  Der Energieverbrauch hängt von vielen Faktoren ab. Es soll ergründet werden inwieweit er mit der Ausführungszeit verknüpft ist.

Zum anderen soll die Exploration von Optimierungsequenzen und Cacheparametern Erkenntnisse zu folgenden Fragestellungen liefern.

- Wie sehen die „optimalen Kompromisse“ aus?
  Nach der der Exploration von Compileroptimierungen wird es Sequenzen geben, die „beste Lösungen“ darstellen. Dies kann ein globales Optimum sein, das für alle Kriterien die besten Resultate enthält oder es gibt mehrere Lösungen, die jede für sich genommen nicht besonders gute Ergebnisse erzielen. Untersucht wird auch, ob die Sequenzen bessere Ergebnisse als die bisher verwendeten Optimierungslevel liefern?
- Welche Compileroptimierungen werden dabei verwendet?
  Die besten Lösungen können zeigen, welche Optimierungen entscheidend für gute

Werte sind und welche vielleicht sogar schädlich sind. Zudem könnte die Reihenfolge der Optimierungen von Interesse sein.

- Kann man eine Heuristik für die Wahl der Cacheparameter herleiten?
  Möglicherweise zeigen die Resultate der Exploration von Cacheparametern einen Zusammenhang zwischen den Parameterwerten und den Kriterien, so dass mit einer einfachen Heuristik die bestmögliche Kombination gefunden werden kann.
- Lassen sich Schlüsse für die Entwicklung von eingebetteten Systemen ziehen? Möglicherweise deuten die Resultate auf Zusammenhänge oder Besonderheiten bezügliche Compileroptimierungen und Cacheparameter hin, die für die speziellen Anforderungen von eingebetteten Systemen von Bedeutung sind.

Zuletzt soll untersucht werden in wieweit die Resultate Potential bieten, Compileroptimierungen für einzelne Kriterien zu verbessern.

1.2 Verwandte Arbeiten

Es sind einige Arbeiten zu finden, die sich mit den Themen Exploration vom Compileroptimierungen, Cacheparametern und Energieverbrauch von Instruktionen beschäftigen und für diese Arbeit von Interesse sind.
Zentrales Werkzeug für diese Arbeit ist der WCET-aware C Compiler (WCC) [Lok07]. Er wird hier verwendet um die WCET, ACET und den Energieverbrauch automatisiert zu ermitteln.
Die in dieser Diplomarbeit realisierten Optimierungen bauen auf der Arbeit von Lokuciejewski [LPF+10] auf. Dort wurden bereits zweidimensionale multikriterielle Explorationen von Compileroptimierungen für die Tricore Architektur durchgeführt. Die Kriterien waren ACET, WCET und Codegröße. Im Gegensatz zu [LPF+10] wird in dieser Diplomarbeit erstmalig der Energieverbrauch als zusätzliche Dimension betrachtet.

Der Energieverbrauch bei der Ausführung von Programmcode unter Berücksichtigung der Speicher für den ARM7TDMI wurde in [The00] gemessen. Diese Arbeit greift auf die dort gewonnenen Daten zurück. Das dort entwickelte Energiemodell wurde für die in dieser Arbeit nötige Energiemessung angepasst.
Der Einfluss verschiedener Cacheparameter auf den Energieverbrauch wurde von Zhang [ZVN05] ausführlich untersucht, um eine konfigurierbare Cachearchitektur zu entwickeln. Die Resultate werden als Vergleich zur Exploration in dieser Arbeit herangezogen. Neben der Betrachtung der WCET und ACET ist die Anzahl der untersuchten Konfigurationen und Benchmarks in dieser Arbeit deutlich größer.
Die Messung des Energieverbrauches von unterschiedlichen Cachearchitekturen für den ARM7 war auch in [Lee01] ein Thema. Der Fokus lag dort auf dem Vergleich zwischen Cache- und Scratchpad-Speichern. Es wurde die Cache-Miss Rate und der Energieverbrauch für verschiedene Cachegrößen und Assoziativitäten untersucht.

1.3 Aufbau der Diplomarbeit

Die Diplomarbeit ist in folgende Kapitel gegliedert, die kurz vorgestellt werden:
In Kapitel 2 werden die Grundlagen zu Compileroptimierungen, Cachespeicher, multikriterieller Exploration und genetischen Algorithmen erläutert. Außerdem beschreibt es die verschiedenen Tools, die bei der Exploration zum Einsatz kommen. Es geht detailliert auf den verwendeten Compiler WCC und seine Analyse- und Profilingwerkzeuge ein.

Kapitel 3 beinhaltet die Exploration der Compileroptimierungen und deren Auswertung. Es erläutert die Problembeschreibung, den verwendeten genetischen Algorithmus und beschreibt den Ablauf der Analyse. Die Resultate der Exploration werden in Form von Tabellen und Diagrammen präsentiert und ausführlich besprochen. Im Verlauf der Auswertungen werden interessante Resultate zu einzelnen Compileroptimierungen näher untersucht. In Abschnitt 3.6 werden Optimierungen aus dem dritten Kapitel näher untersucht. Ziel ist es festzustellen, warum es beim Einsatz dieser Optimierungen zu Verschlechterungen in den ermittelten Werten kommt. Es wird untersucht, ob es Möglichkeiten gibt die negativen Auswirkungen zu beseitigen. Am Ende des Kapitels sind die Schlussfolgerungen im Hinblick auf die Fragestellungen der Diplomarbeit formuliert.

Das Kapitel 4 thematisiert die Exploration der Cacheparameter. Der Ablauf der Exploration und die Auswahl der zu untersuchenden Parameterwerte werden besprochen. Die Resultate geben die Auswirkungen der Variation der verschiedenen Cacheparameter wieder und erlauben Rückschlüsse auf die Abhängigkeiten zwischen Parametern, Programmcode und der Systemarchitektur.

Kapitel 5 fasst alle Resultate und und deren Schlussfolgerungen zusammen und zieht ein abschließendes Fazit. Die Arbeit schießt mit einen Ausblick auf zukünftige Arbeiten ab.

2 Grundlagen und technischer Rahmen

Bevor im Hauptteil dieser Arbeit die Explorationen von Compileroptimierungen und Cacheparametern besprochen werden, ist es notwendig einige Grundlagen zu erläutern. Zum einen sind das Details bezüglich des betrachteten Mikroprozessors (Abschnitt 2.1) und des Cachespeichers (Abschnitt 6). Dabei wird insbesondere auf die untersuchtet Cacheparameter eingegangen. Die Explorationen stützen sich auf zahlreiche Werkzeuge deren Funktionsweise ebenfalls kurz erläutert wird. Der Aufbau des WCC-Compilers (Unterabschnitt 2.3.1) und die verschiedenen Stationen der Code-Erzeugung werden detailliert betrachtet. Zum besseren Verständnis sind außerdem theoretische Grundlagen zur multikriteriellen Optimierung (Abschnitt 2.6) und Informationen zur Verwendung des PISA-Frameworks (Abschnitt 2.5) notwendig.

2.1 ARM7TDMI Plattform

Die Zielplattform für die Codeerzeugung und vorgestellten Optimierungen dieser Arbeit ist der ARM7TDMI Mikroprozessor von ARM [Lim04]. Das Kürzel TDMI steht für **T**humb-**D**ebug-**M**ultiplier-**I**CE und beschreibt damit seine besonderen Eigenschaften. Es handelt sich um einen 32-Bit RISC Prozessor mit klassischer Load/Store Architektur. Er verfügt über zwei verschiedene Befehlssätze, das *ARM-Instruction-Set* mit 80 Instruktionen von 32-Bit Wortbreite und das *Thumb-Instruction-Set*, welches auf 36 Befehle beschränkt ist. Thumb-Instruktionen sind nur 16-Bit breit und erlauben weniger Adressierungsarten. Im Vergleich zu den ARM ermöglicht die Verwendung des Thumb-Instruktionssatzes eine höhere Codedichte. Ein Programm kann sowohl ARM- als auch Thumb-Instruktionen enthalten und es ist möglich im laufenden Betrieb mit speziellen Sprung-Instruktionen zwischen ARM- und Thumb-Mode umzuschalten.
Der ARM7TDMI besitzt eine 3-stufige Pipeline und einen gemeinsamen Bus für Daten und Instruktionen. Die geringe Stromaufnahme des ARM7TDMI macht ihn attraktiv für den Einsatz in eingebetteten Systemen. So wird er häufig in tragbaren Kommunikationsgeräten und Spielekonsolen verwendet.

Der LPC2880 Mikrocontrollers von NXP [Sem07] mit ARM7TDMI Kern bietet die Vorlage für das Speicherlayout, welches für das Profiling in dieser Arbeit verwendet wird. Der LPC2880 besitzt einen 60MHz ARM7TDMI Prozessor mit 8kB Cache. Weiter stehen 64kB SRAM, 32kB ROM und 1MB Flashspeicher zu Verfügung.

2.2 Cachespeicher

Seit den 80er Jahren steigt die Taktgeschwindigkeit von Prozessoren deutlich schneller als die der Hauptspeicher [HP07, S.289]. Die Versorgung des Prozessors mit Daten aus dem Hauptspeicher wurde so zum Flaschenhals für die Performanz von Computersystemen. Zwar existieren schnellere Speichertypen, diese sind jedoch zu teuer um den Hauptspeicher in der benötigten Kapazität zu ersetzen. Die Lösung für dieses Problem ist die Kombination unterschiedlicher Speicher und deren intelligente Organisation.
Mit hierarchisch geordneten Speicherebenen machen sich Entwickler die Vorteile verschiedener Speichertypen zu Nutze. Besonders schnelle und energieeffiziente Speicher stehen prozessornah zur Verfügung (z.B. L1-Cache oder Scratch Pad Memorys). Diese sind aber wie bereits erwähnt sehr teuer und haben nur kleine Speicherkapazitäten. Durch diese kleinen und schnellen Speicher ist es Möglich das Prinzip der Lokalität effektiv zu nutzen. Das Prinzip der Lokalität besagt, dass auf Bereiche von Daten, auf die kürzlich zugegriffen wurde, mit hoher Wahrscheinlichkeit in naher Zukunft wieder zugegriffen wird [HP07, S.38]. Es kann also in einem gewissen Rahmen vorausgesagt werden, welche Daten gelesen oder geschrieben werden. Daher ist es sinnvoll kleine und schnelle Speicher zum puffern (z.B.in Cachespeicher) der in Zukunft benötigten Daten zu nutzten. In den darauf folgenden Ebenen, werden die Speicher größer, langsamer und verbrauchen mehr Energie (L2-, L3-Cache). Die letzte Ebene stellt dann der Hauptspeicher dar.

Welche Daten gepuffert werden entscheidet bei Caches nicht der Programmierer, sondern eine eigene Kontrolllogik, die zwischen dem Cache und der CPU liegt. Dabei werden die Adressen des Hauptspeichers auf den Adressraum des Cachespeichers abgebildet. Ein Cachespeicher ist in Blöcke unterteilt, die potentiell mehrere Cachezeilen enthalten. Jede Cachezeile kann wiederum die Daten mehrerer Adressen des Hauptspeichers aufnehmen. Auf die Cachezeilen innerhalb eines Blockes wird mit der selben Block-Adresse zugegriffen. Diese Adresse (Tag) ist ein Teil der gesamten Hauptspeicheradresse. Bei Zugriffen auf den Hauptspeicher wandelt der Cachecontroller die gesuchte Adresse des Hauptspeichers in die Adresse eines Cacheblocks um:

$$\text{Blockadresse} = (\text{ Adresse} - (\frac{\text{Adresse}}{\text{Zeilengröße}})) \; mod \; (\text{ Anzahl der Blöcke} * \text{Zeilengröße})$$

Innerhalb des Blockes wird dann überprüft, ob die gesuchte Adresse vorhanden ist. Befindet sich der Inhalt der Adresse in dieser Cachezeile (sog. Cache-Hit), so greift der Prozessor auf diese Daten zu. Ist der gesuchte Inhalt in der Cachezeile nicht vorhanden (Cache-Miss), so werden die Daten vom Hauptspeicher in den Cache geladen und gleichzeitig dem Prozessor zur Verfügung gestellt. Der nächste Zugriff auf selbe Adresse kann dann wieder über den Cache geschehen. Da der Adressraum des Hauptspeichers deutlich größer ist als der des Caches, sind mehrere Adressen demselben Set zugeordnet. Daher kann es zu Konflikten kommen wenn die Cachezeile, auf die zugegriffen werden soll bereits andere Daten enthält. Dies entspricht einem Cache-Miss und die im Cache gehaltene Daten müssen in den Hauptspeicher zurück geschrieben und mit den aktuell

benötigten ersetzt werden. Die Häufigkeit dieser Konflikte wird verringert wenn ein Set mehrere Cachezeilen aufnehmen kann.

Entscheidend für die Performanz des Systems ist die Effektivität mit der die Daten gecached werden. Befinden sich die Daten, auf die der Prozessor zugreifen will, im Cache, so wirkt sich das ausgesprochen positiv auf die Laufzeit der Software und den Energieverbrauch des Systems aus. Sonst sind zeitintensive Transfers mit dem Hauptspeicher nötig. Dieser Fall lässt sich nie ganz vermeiden, doch ist es oberstes Ziel, bei der Verwendung von Cachespeicher, die Häufigkeit der Cache-Misses (Cache-Miss Rate) so gering wie möglich zu halten. Da ein Zugriff auf den Cache deutlich weniger Energie verbraucht als auf den Hauptspeicher, führen Cache-Hits auch zu sinkendem Energieverbrauch bei der Ausführung eines Programms. Auf der anderen Seite steht aber der potentiell steigende Energieverbrauch des Cachespeichers, wenn beispielsweise die Cachegröße erhöht wird. Diese Auswirkungen müssen gegeneinander abgewogen werden. Mit den im Folgenden beschriebenen Parametern kann auf die Cache-Hit Rate Einfluss genommen werden.

2.2.1 Cachegröße

Die nahe liegende Möglichkeit, die Cache Hit Rate zu erhöhen, ist die Vergrößerung des verfügbaren Cachespeichers. Da mehr Bereiche des Codes im Cache gehalten werden können, treten Cache-Misses seltener auf. Ein größere Cache benötigt aber mehr Energie und Fläche auf dem Chip. Diese negativen Effekte werden von den Einsparungen, die durch reduzierte Zugriffe auf den Hauptspeicher entstehen deutlich übertroffen. Erst wenn eine Erhöhung der Cachegröße keine bessere Cache-Miss Rate mehr bewirkt, werden die Nachteile spürbar. Es ist also sinnvoll zu wissen wie klein der Cachespeichers für das entsprechende System sein darf. Heute sind Caches mit Größen von 2KB - 256KB üblich, in eingebetteten Systemen sind in der Regel nur kleinere L1-Cache zu finden.

2.2.2 Cachezeilengröße

Der Cache ist in einzelne Cachezeilen unterteilt, die üblicherweise 8 Byte - 128 Byte groß sind. Muss im Falle eines Cache-Misses das gesuchte Datum aus dem Hauptspeicher geholt werden, wird in der Regel gleich die ganze Cachezeile befüllt (einige Prozessoren erlauben auch den Zugriff auf einen Teil der Cachezeile, z.B. Tricore). Neben dem geforderten Datum befinden sich nun auch die benachbarten Daten des Hauptspeichers in der Zeile.
Der Vorteil liegt darin, dass Daten auf die mit hoher Wahrscheinlichkeit bald zugegriffen wird, sich dann bereits im Cache befinden; weniger Cache-Misses sind die Folge. Das Befüllen einer ganzen Cachezeile kostet mehr Zeit und Energie, jedoch kann dadurch das Prinzip der Lokalität besser genutzt werden. Dies ist besonders effektiv bei häufigen Zugriffen auf aufeinander folgende Adressen. Ist das nicht der Fall, werden oft überflüssige Daten in den Cache geladen und die Laufzeit sowie der Energieverbrauch unnötig erhöht.

Bei der Wahl der Cachezeilengröße muss also die sinkende Cache-Miss Rate gegen die negativen Auswirkungen abgewogen werden.

2.2.3 Assoziativität

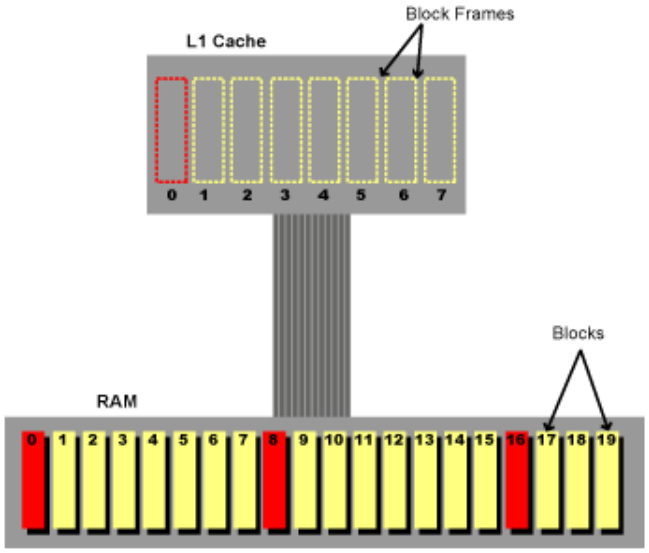


Abbildung 2.1: Illustration eines Direct-Mapped Caches, aus [ars11]

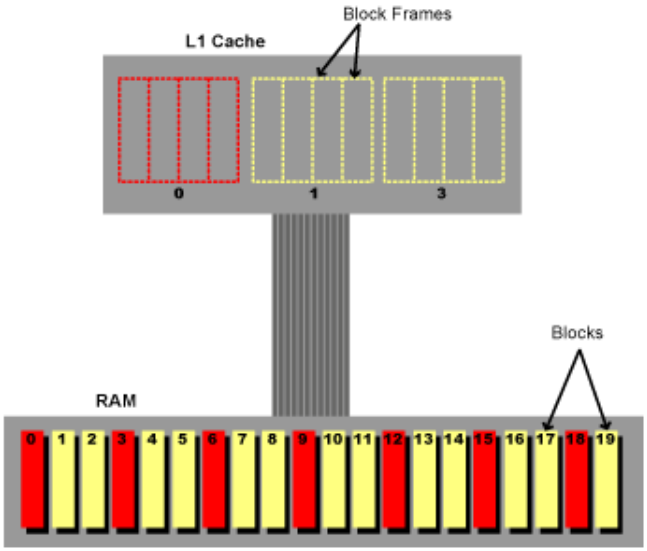


Abbildung 2.2: Illustration eines 4-Way-Associativ Caches, aus [ars11]

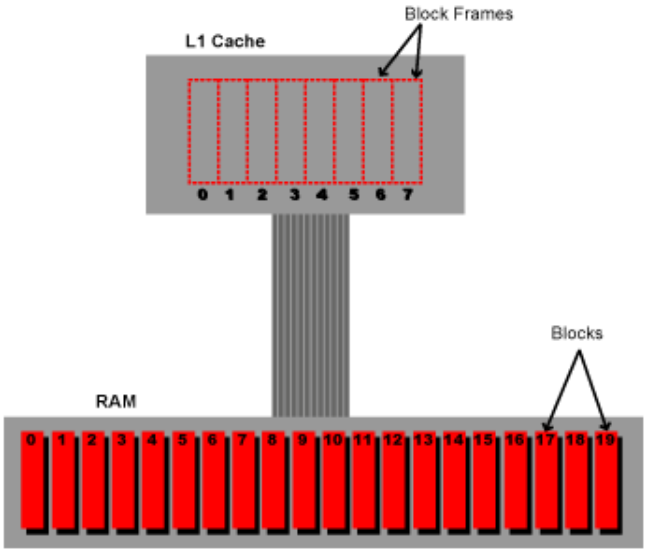


Abbildung 2.3: Illustration eines Fully-Associativ, Caches, aus [ars11]

Die Cache-Assoziativität gibt die Anzahl der in einem Block vorhandenen Cachezeilen an. Diese Zeilen werden parallel über die selbe Adresse angesprochen. Bei Zugriffen auf die gleiche Block-Adresse stehen so mehrere Cachezeilen zur Verfügung, aus denen die nächste freie genutzt wird. Bei Direct Mapped Caches, besteht jeder Block aus einer einzigen Cachezeile (siehe Abb. 2.1). Bei Zugriffen auf den Cacheblock wird immer diese Zeile verwendet. So erhält man die größtmögliche Anzahl an Blöcken, muss aber bei häufigem Zugriff auf den selben Block mit vielen Konflikten rechnen. Das andere Extrem sind Voll-Assoziative Caches, die aus nur einem Block bestehen (siehe Abb. 2.3). Bei jedem Zugriff werden alle Cachezeilen überprüft. Konflikte entstehen hierbei erst wenn alle Cachezeilen gefüllt sind. Dazwischen sind alle Abstufungen denkbar, häufig werden Assoziativitäten von 2 oder 4 (siehe Abb. 2.2) verwendet.
Welche Stufe der Assoziativität die wenigsten Konflikte erzeugt, hängt von der Folge von Zugriffen und damit von der Art des Codes ab, der verwendet wird. Man kann sagen, dass mit höherer Assoziativität die Cache-Miss Rate sinkt, jedoch auch hier auf Kosten der Energie und Komplexität der Cache-Controller-Logik.

2.3 WCC (WCET-aware C Compiler)

Der WCET-aware C Compiler (WCC) wird seit einigen Jahren am Lehrstuhl 12 der TU Dortmund entwickelt und zielt auf die besonderen Anforderungen an Software für eingebettete Systeme ab. Der WCC ist ein ANSI-C C99 [ISO99] Compiler für die Tricore Architekturen TC1796 und TC1797 sowie für den ARM7TDMI. Als einziger Compiler integriert der WCC die Ermittlung der WCET in den Übersetzungsvorgang und er-

möglicht die gezielte Optimierung hinsichtlich der WCET. Mit Hilfe des WCC werden WCET-basierte Optimierungen entwickelt. Für die Tricore Architekturen sind solche Optimierungen bereits implementiert, eine Unterstützung der ARM7TDMI Plattform ist momentan in der Entwicklung.

2.3.1 Aufbau des WCC

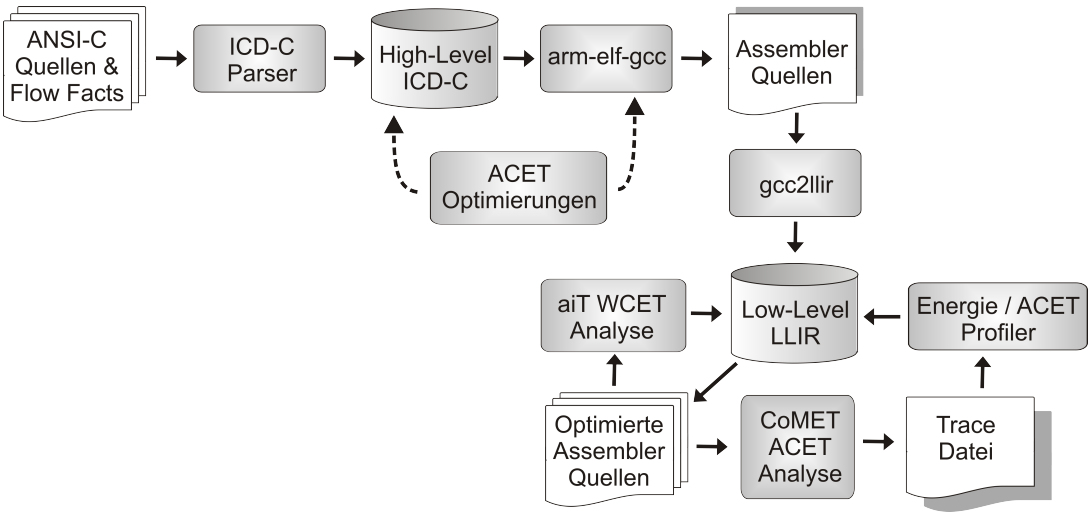


Abbildung 2.4: Aufbau des WCC

Abbildung 2.4 skizziert den Aufbau des WCC für die ARM7TDMI Plattform. Im folgenden sollen kurz die einzelnen Stufen des Übersetzungsvorgangs erläutert werden: Zu Beginn steht der C Quellcode, der in der Regel mit Flowfact-Annotationen versehen ist. Flowfact-Annotationen dienen der WCET-Analyse zur Ermittlung der Anzahl von Schleifeniterationen und Rekursionstiefen. C-Code und Flowfacts werden in die High-Level Repräsentation ICD-C übersetzt.
ICD-C wird von der Abteilung „Eingebettete Systeme“ am Informatik Centrum Dortmund e.V. entwickelt [icd11]. Auf dieser C-nahen Repräsentation können High-Level Optimierungen durchgeführt werden, eine Rücktransformation in ANSI-C ist ebenso möglich. Da der WCC keinen Codeselektor für ARM enthält, wird der optimierte Code an den arm-elf-gcc weitergeleitet. Dort können die im arm-elf-gcc enthaltenen Low-Level Optimierungen angewendet werden. In nächsten Schritt erzeugt der arm-elf-gcc Assemblercode für den arm7tdmi Prozessor.
Der Assemblercode wird im WCC mit Hilfe des gcc2llir-Konverters in die Lowlevel Repräsentation ICD-LLIR transformiert. Auf dieser Repräsentation können wiederum Optimierungen auf Assembler-Ebene durchgeführt werden können. Das Resultat dient nun

als Ausgangsbasis für das Profiling. Aus der ICD-LLIR wird der optimierte Binärcode generiert und zur Ermittlung der ACET und WCET an externe Tools (aiT, CoMET) weitergegeben. Die ermittelten Werte werden dann als Annotationen an die ICD-LLIR zurückgeführt. Die Ermittlung des Energieverbrauches wird mittels einer Trace Datei und einer Energie-Datenbank mit zuvor gemessenen Werten durchgeführt. Im Folgenden wird detaillierter auf die einzelnen Analysen eingegangen.

2.3.2 WCET Analyse

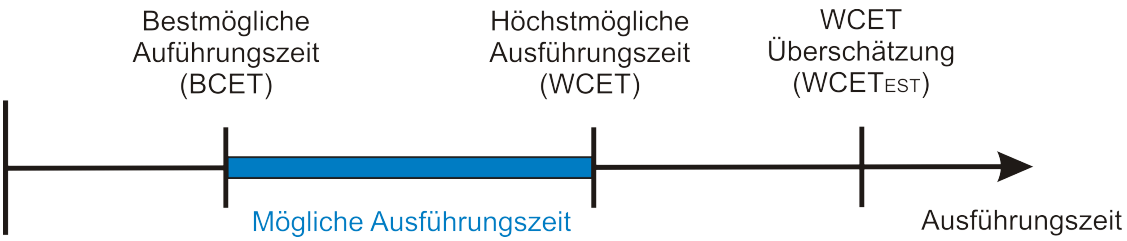


Abbildung 2.5: Obere und untere Schranken für die Ausführungszeit

Die Verteilung der möglichen Ausführungszeiten eines Programms wird von der bestmöglichen (Best-Case Execution Time, BCET) und der höchstmöglichen Laufzeit(WCET) eingeschlossen (siehe Abb. 2.5. Die tatsächliche WCET eines Programms zu berechnen ist in der Regel nicht möglich. Mehrstufige Pipelines und Caches machen die Laufzeitvorhersagen äußerst komplex. Es wird versucht eine obere Grenze für die WCET abzuschätzen ($WCET_{EST}$), die möglichst dich an der tatsächlichen liegt, ohne sie zu unterschreiten. Wenn in dieser Arbeit von einer ermittelten WCET gesprochen wird, dann ist damit immer die ($WCET_{EST}$) Überabschätzung gemeint.
Die WCET des im WCC kompilierten Programmcodes wird mit Hilfe des statischen Analsyetools aiT von AbsInt Angewandte Informatik GmbH [ait11] ermittelt. Die Berechnung umfasst eine Kontrollfluss-, Value-, Cache-, Pipeline- und Pfadanalyse. aiT ermittelt dann eine obere Laufzeitgrenze eines Binärprogramms, die möglichst nah an der tatsächlichen WCET liegt [Tan06].

Ein kritischer Punkt bei der statischen Laufzeitanalyse ist die Ermittlung der Rekursionstiefe von Funktionen und Ausführungshäufigkeiten von Schleifen. Für sehr einfache Fällen erkennt die aiT-eigene Schleifenanalyse die Iterationszahl einer Schleife. Meistens jedoch müssen obere Grenzwerte für die Ausführungshäufigkeit der Schleifen im Code an aiT übergeben werden. Diese Informationen können manuell als Flowfacts-Annotationen in den C-Code geschrieben werden. Mit Hilfe des an Lehrstuhl 12 entwickelten Flowfactmanagers [Sch07] werden diese Annotationen vom Quellcode in die ICD-LLIR transformiert und behalten auch nach Änderungen durch Optimierungen ihre Gültigkeit. Weiterhin

enthält der WCC eine eigene statische Schleifenanalyse [LCFM09], die die ermittelten Schleifengrenzen direkt als Flowfacts in den C-Code einfügt. Im Anschluss an die WCET Analyse von aiT werden die ermittelten WCETs/WCECs jedes Basisblock an die ICD-LLIR übertragen.

2.3.3 ACET Profiling

Zur Bestimmung der ACET greift der WCC auf den CoMET Simulator von Synopsys zurück [com11]. CoMET ist eine Entwicklungs- und Simulationsumgebung, die zahlreiche Plattformen unterstützt, darunter auch ARM7TDMI. In CoMET wird eine virtuelle Plattform erzeugt, die neben dem ARM7TDMI Prozessor auch Speicher- und Bussysteme enthält. Das Speicherlayout wurde entsprechend der Beschreibung des LPC2880 [Sem07] mit SRAM und Flashspeicher konfiguriert.
Nach Eingabe einer Binärdatei erzeugt der CoMET-Simulator ein Tracefile, das die ausgeführten Instruktionen und die benötigten Zyklen sowie Informationen über Cachezugriffe protokolliert. Der im WCC integrierte ACET-Profiler liest diese Datei ein. Zu jedem im Programmcode enthaltenen Basisblock wird die Anzahl der benötigten Zyklen festgehalten. Ebenso werden die verwendeten Pfade zwischen den Basisblöcken protokolliert. So kann die Laufzeit jedes Basisblocks und die des gesamten Programms ermittelt werden.

2.3.4 Energie Profiling

Bisher wurde nur generell vom „Energieverbrauch“ bei der Ausführung von Software gesprochen. Es soll nun erläutert werden, welche Komponenten eines eingebetteten Systemen hier für das Energieprofiling relevant sind. Da es bei den Explorationen zum einen um Compileroptimierungen und zum anderen um Cachearchitekturen geht, werden nur die Komponenten des Systems betrachtet, auf die sich diese Parameter auswirken. In diesem Fall sind das der Prozessor und die verwendeten Speicher. Der hier betrachtete Energieverbrauch entspricht der Summe des Energieverbrauchs,der bei der Ausführung der einzelnen Instruktionen verbraucht wird.

Energiemodell

Ein allgemeines Energiemodell für Prozessor-Instruktionskosten wurde von Tiwari [TMWL96] vorgestellt. Im Rahmen der Diplomarbeit von Theokaridis [The00] wurde dieses Modell um die Auswirkungen auf externen Speicher erweitert und zur Energiemessung für den ARM7TDMI verwendet. Diese Messung dient als Quelle für die Berechnung des Energieverbrauches. Der CoMET-Simulator erzeugt für jeden Benchmark ein TraceFile, das eine Liste der ausgeführten Instruktionen mit Anzahl der benötigten Zyklen enthält. Der Energie-Profiler im WCC ermittelt zu jeder dieser Instruktionen den Energieverbrauch. Mit folgendem vereinfachtem Modell wird der Energieverbrauch einer

Instruktion ermittelt:

$$\begin{aligned} Energy &= Energy\_Inst + Energy\_Mem \\ Energy\_Inst &= Energy\_InstPerCycle * Cycles \\ Energy\_Mem &= Energy\_InstFetch + Energy\_PipelineFill \\ &\quad + Energy\_LoadStore \end{aligned}$$

Der Energieverbrauch einer Instruktion (*Energy*) setzt sich aus dem prozessorinternen Verbrauch (*Energy_ Inst*) und dem Verbrauch beim Zugriff auf externen Speicher und den Cache (*Energy_ Mem*) zusammen. Der prozessorinterne Verbrauch berechnet sich aus den Basiskosten, die von der aufgeführten Instruktion abhängt. Für jede Instruktion ist das die Schaltkreisaktivität in den funktionalen Einheiten pro Zyklus (*Energy_ InstPerCycle*), multipliziert mit der Anzahl der für die Instruktionen benötigten Zyklen. Da die Messungen in [The00] nur für Thumb-Instruktionen durchgeführt wurden, wird für alle 32-Bit ARM-Instruktionen ein durchschnittlicher Wert verwendet.
Zugriffe auf externen Speicher treten beim Fetching einer Instruktion (*Energy_ InstFetch*) und bei Load/Store-Instruktionen (*Energy_ LoadStore*) auf, die Daten zwischen Registern und externen Speicher transferieren. Je nach Speicherart (SRAM oder Flash) wird der entsprechende Energieverbrauch hinzugefügt. Für Sprungbefehle werden zusätzlich die Kosten für das Auffüllen der Pipeline addiert (*Energy_ PipelineFill*).
Bei der Exploration der Cacheparameter muss für die Berechnung der Speicherenergie das Cacheverhalten berücksichtigt werden. Im TraceFile ist für jede ausgeführte Instruktion protokolliert ob ein Cache-Miss aufgetreten ist, im anderen Fall handelt es sich um einen Cache-Hit. Abhängig davon wird der Energieverbrauch berechnet und zu den Kosten zum Fetchen einer zugefügt:

$$\begin{aligned} Energy\_InstCacheFetch &= Energy\_InstFetch \\ &\quad + (Energy\_CacheHit \; bzw. \; Energy\_CacheMiss) \\ Energy\_CacheHit &= Energy\_CacheAccess \\ Energy\_CacheMiss &= Energy\_CacheAccess + Energy\_CacheLineFetch \end{aligned}$$

Im Falle eines erfolgreichen Cachezugriffes wird nur die Energie für den Cachezugriff ermittelt (*Energy_ CacheAccess*). Tritt ein Cache-Miss auf, müssen Daten aus dem Hauptspeicher in die entsprechende Cachezeile geladen werden. Dazu bedarf es je nach Cachezeilengröße mehrerer Leseoperationen, für die jeweils die entsprechenden Energiekosten anfallen (*Energy_ CacheLineFetch*).
Bei der Exploration der Cacheparameter werden verschiedene Organisationen des Cachespeichers bezüglich Größe und Assoziativität untersucht. Dabei muss berücksichtigt werden, dass diese Organisationen einen unterschiedlichen Energiebedarf bei Cachezu-

griffen haben. Daher wurde der Energieverbrauch pro Zugriff für alle untersuchten Cachearchitekturen mit Hilfe des CACTI-Modells vorab ermittelt [cac11]. Einige Werte, besonders die für kleine Cachegrößen unter 4kB, lassen sich mit CACTI nicht berechnen. Bei Annahme einer nahezu linearen Entwicklung des Energieverbrauchs bei sinkenden Cachegrößen, wurden die fehlenden Daten approximiert.
Diese Energiewerte werden in das beschriebene Energiemodell eingesetzt (*Energy_CacheAccess*), mit dem der Energieverbrauch für jede Instruktionen und jeden Basisblock ermittelt wird.

2.3.5 Benchmarks

Die Explorationen in dieser Arbeit benutzen eine Reihe von Benchmarks aus verschiedenen Benchmark-Suiten. Diese Kollektion von Benchmarks ist das Trainingsset, auf dem die Compilersequenzen und Cacheparameter optimiert werden. Ein weiteres Testset mit vom Trainingsset verschiedenen Benchmarks wird dann verwendet um die Qualität der gefundenen Lösungen zu überprüfen.
Diese verwendeten Benchmarks enthalten Code, der typischen Anwendungen aus verschiedenen Bereichen der eingebetteten Systeme entspricht. *DSPstone* [ZMSM94] enthält Standardalgorithmen der digitalen Signalverarbeitung wie *FFT* (Fast-Fourier-Transformation) oder *FIR* (Finite Impulse Response) und typische DSP-Anwendungen (*adpcm-transcode*, *convolution* u.a.). Die Sammlung enthält Code mit Fest- sowie Gleitkommaarithmetik. Unter dem Titel *MRTC* sind eine Vielzahl unterschiedlicher Benchmarks versammelt, zusammengetragen vom Mälardalen Real-Time Research Center [mrt]. Die „University of Toronto Digital Signal Prozessing Benchmark Suite (UTDSP)“ [utd11] enthält weitere Benchmarks mit Code aus der Signalverarbeitung und Implementationen von Modemprotokollen (G.721 etc.).
Diese Auswahl soll eine möglichst umfassende und gute Annäherung an die in eingebetteten Systemen eingesetzte Software darstellen.

2.4 Compileroptimierungen

Die Übersetzung von Quellcode einer Programmiersprache in Maschinencode für eine spezielle Plattform ist ein komplexer Vorgang. Die Codequalität des Binärprogramms entscheidet oft darüber, in wieweit die übersetzte Software effizient eingesetzt werden kann. Compileroptimierungen bieten ausgeklügelte Mechanismen, die die Qualität des Codes auf High-Level (Hochsprache) und auf Low-Level (Assembler) hinsichtlich Laufzeit und anderer Kriterien verbessern. Die meisten Optimierungen zielen auf eine Verbesserung der durchschnittlichen Laufzeit ab. Es gibt Optimierungen die Sprünge im Programmfluss verringern (*Loop Unrolling*), unnötigen Code entfernen (*Redundant Code Detection, Common Subexpression Elimination*) oder spezielle Eigenschaften der Zielarchitektur ausnutzen (*Peephole Optimizations*).

Die Standardoptimierungsstufen eines Compilers geben dem Nutzer die Wahl zwischen mehreren Graden der Optimierung hinsichtlich der ACET. Oft ist auch eine Optimierung hinsichtlich der Codegröße (z.B. beim gcc) möglich. Bei der Entwicklung von Software für eingebettete Systeme spielen aber, wie bereits erwähnt, auch der Energieverbrauch und die WCET eine wichtige Rolle. Jede Optimierung wirkt sich auch auf diese Kriterien aus, was die Auswahl der einzusetzenden Compiler-Optimierungen zu einem wichtigen Faktor bei der Entwicklung von Systemsoftware macht.
Aufgrund der hohen Anzahl verfügbarer Compileroptimierungen ist die Menge der möglichen Kombinationen riesig, zumal auch die Reihenfolge der Anwendung das Ergebnis beeinflusst. Die Standardoptimierungstufen eines Compilers können die Qualität eines Codes nur begrenzt verbessern, da die Auswirkungen der Optimierungen bei jedem Code individuell sind.

2.5 Multikriterielle Optimierung

Die einfachste Form eines Optimierungsproblems ist die eindimensionale Optimierung. Dabei wird für eine Problemstellung, zu der es mehrere mögliche Lösungen gibt, die beste aller gültigen Lösungen hinsichtlich eines Zielkriteriums gesucht. Ziel ist es den Wert des Kriteriums zu minimieren oder zu maximieren. Ein beliebtes Beispiel hierfür ist das Rucksackproblem. Für solche linearen Optimierungsprobleme existieren effiziente Lösungsverfahren, die die optimale Lösung finden.
Anders verhält es sich mit Problemstellungen, die mehr als ein Zielkriterium aufweisen. Für solche Probleme existiert in der Regel keine objektiv „beste“ Lösung, die optimale Werte in allen Kriterien aufweist. Dies ist dann in der Art des Problems begründet: Eine Verbesserung der Werte hinsichtlich eines Kriterium, hat meist eine Verschlechterung in einem anderen Kriterium zur Folge. Ein gutes Beispiel hierfür ist ein Verbrennungsmotor. Dort bewirkt eine Steigerung der Leistung einen höheren Verbrauch von Treibstoff, da dieser ineffizienter verbrannt wird.

Eine Möglichkeit dieses Problem anzugehen, ist es eine Gewichtung der Kriterien vorzunehmen und sie so zu einer einzigen Zielfunktion zusammenzufassen. Bei vielen Anwendungen ist jedoch solch eine Gewichtung nicht möglich oder nicht sinnvoll.
Obwohl für die betrachteten Problemstellungen nicht „die“ optimale Lösung existiert, gibt es Lösungen die eindeutig besser sind als andere. Wenn eine Lösung a in einem Kriterium besser ist als eine Lösung b, und zugleich in allen anderen Kriterien nicht schlechter abschneidet, dann dominiert Lösung a Lösung b. Man spricht von *Pareto-Dominanz* ($\succ$).

$$ParetoDominanz: \qquad a_i \succ b_i \quad := \quad \forall i\, a_i \geq b_i, \quad \exists i\, a_i > b_i$$

Bei einer multikriteriellen Optimierung wird daher eine Menge von Lösungen gesucht, die von keinen anderen Lösungen dominiert wird. Man könnte sie als die „besten Kompromisse“ bezügliche aller Kriterien bezeichnen. Solche Lösungen nennt man *pareto-optimale* Lösungen.

2.5.1 Pareto-optimale Lösungen

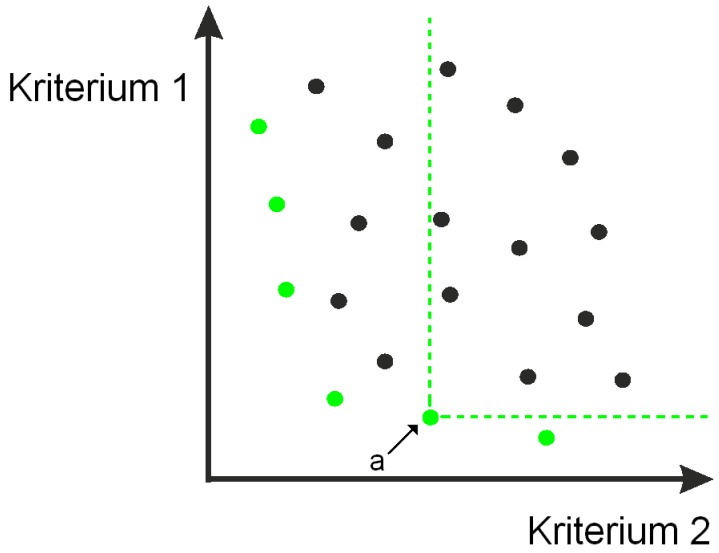


Abbildung 2.6: Darstellung der Pareto-Front (grün) in einer zweidimensionalen Lösungsmenge.

Eine pareto-optimale Lösung zeichnet sich dadurch aus, dass keine andere Lösung existiert, die einen besseren Wert in einem Kriterium besitzt, ohne einen schlechteren Wert in mindestens einem anderen Kriteriums zu haben. Es gibt demnach keine andere Lösung, die diese pareto-optimale Lösung dominiert. Es kann mehrere solcher Lösungen geben. Die Menge der pareto-optimalen Lösungen bezeichnet man als *Pareto-Front*. Die Pareto-Front stellt somit die Menge der „besten Kompromisse“ dar, aus denen der Anwender die für ihn am besten geeignete wählen kann.

Abbildung 2.6 zeigt eine Menge von Lösungen für ein zweidimensionales Optimierungsproblem. In grün ist die Pareto-Front markiert. Am Beispiel von Punkt a ist zu sehen, welche anderen Lösungen von einer Lösung aus der Pareto-Menge dominiert werden.
Um die pareto-optimalen Lösungen zu einem Problem identifizieren, müssten die Lösungswerte aller möglichen Lösungen bekannt sein. Bei vielen Problemstellungen ist die Anzahl aller Lösungen jedoch zu groß um eine vollständige Analyse effizient durchzuführen. Eine gute Approximation der Pareto-Front ist dann das Ziel. Für multikriterielle Optimierungen und die damit verbundene Exploration des Lösungsraumes haben sich genetische Algorithmen als besonderes wirksam erwiesen.

2.5.2 Genetische Algorithmen

Genetische Algorithmen [Hol92] ermöglichen die Lösung komplexer Probleme mit Hilfe von Lösungsstrategien aus der Biologie. Solche Algorithmen arbeiten mit Individuen,

Chromosomen und Genen und simulieren die „natürliche Auslese“ in der Natur. In einem Prozess der künstlichen Evolution werden über mehrerer Generationen Lösungen erzeugt und variiert. Man geht davon aus, dass aus guten Lösungen durch Kombination und Variation der Merkmale bessere Lösungen entstehen. Ziel dieses „Survival of the fittest“ ist es mit jeder neuen Generation den optimalen Lösungen näher zu kommen.
Dabei entspricht ein Individuum einer möglichen Lösung des Problems. Ein solches Individuum besitzt im einfachsten Fall ein Chromosom, dass wiederum aus einer Reihe von Genen besteht. Diese Gene sind nun die einzelnen Elemente einer Lösung, die variiert werden können. Im Falle des Rucksackproblems entspricht jedes Gen einem bestimmten Objekt, dass eingepackt werden soll, und die Menge aller Gene eines Individuums ist die Menge der Objekte, die in den Rucksack sollen. Somit lässt sich jedem Individuum ein entsprechender Lösungsvektor zuordnen. Der Prozess der künstliche Evolution läuft in folgenden Schritten ab:

1. Initialisierung
   Es wird (randomisiert) eine Anfangspopulation generiert.

2. Evaluation
   Zu jedem Individuum wird der Lösungsvektor ermittelt.

3. Selektion
   Aus der aktuellen Population wird nach bestimmten Kriterien eine Teilmenge ausgewählt. Die übrigen Individuen werden verworfen.

4. Variation
   Neue Individuen werden mittels Rekombination der bestehenden Individuen erzeugt (Crossover). Einzelne Gene der Individuen werden verändert (Mutation).

5. Evaluation
   Zu jedem Individuum wird der Lösungsvektor ermittelt.

6. Abbruch oder weiter mit Schritt 3

Jeder Durchlauf in diesem Prozess erzeugt eine neue Generation von Individuen. Ziel ist es immer bessere Individuen zu erzeugen, dabei entscheidet der Algorithmus des Selektors, welche Individuen „besser“ sind. Es wird abgebrochen, wenn eine festgesetzte Anzahl von Generationen durchlaufen ist oder keine besseren Lösungen mehr erzeugt werden.

Die Qualität der gefundenen Lösungen hängt von der Art der Selektion und Variation ab. Besonders für multikriterielle Optimierungen muss bei der Selektion darauf geachtet werden, dass die Population nicht in einem lokalen Optimum versinkt. Für die Variation mittels Crossover und Mutation stehen mehrere Varianten zur Verfügung, die sich durch zahlreiche Parameter, beispielsweise die relative Häufigkeit der variierten Individuen, steuern lassen (vgl. [Obi98]).

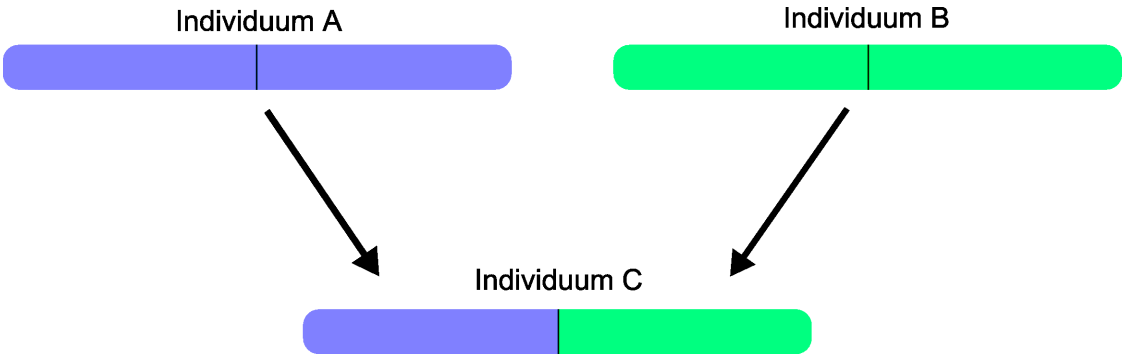


Abbildung 2.7: Crossover zweier Individuen

Crossover

Beim Crossover werden zwei Individuen zu einem neuen kombiniert. Dabei werden die Gensequenzen der Individuen an einer (*One-Point-Crossover*) oder mehreren Stellen (*Multi-Point-Crossover*) getrennt und vermischt wieder zusammen gefügt. Abbildung 2.7 zeigt diesen Mechanismus für den Fall des *one-point-crossovers*. Die Individuen A und B werden an einem Punkt getrennt und Individuum C aus den beiden Enden zusammengefügt.

Mutation

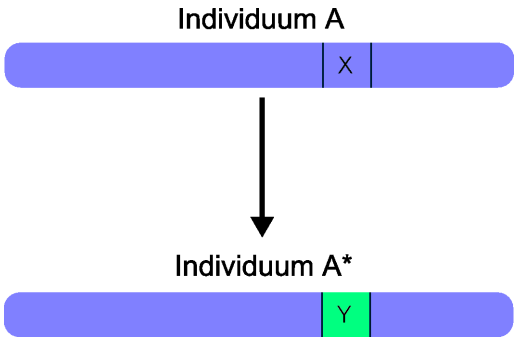


Abbildung 2.8: Mutation eines Individuums

Bei der Mutation werden die Gene eines Individuums variiert: Je nach Art der Mutation werden ein oder mehrere Gene an einer zufälligen Position in der Sequenz durch andere Gene ersetzt. Abbildung 2.8 zeigt die zufällige Variation eines Genes in einem Individuum. Das Gen X wird durch das Gen Y ersetzt. Am Ende dieser künstlichen Evolution steht eine Menge von Lösungen, deren approximierte Pareto-Front möglichst nahe an der richtigen liegen soll.

2.6 PISA Framework

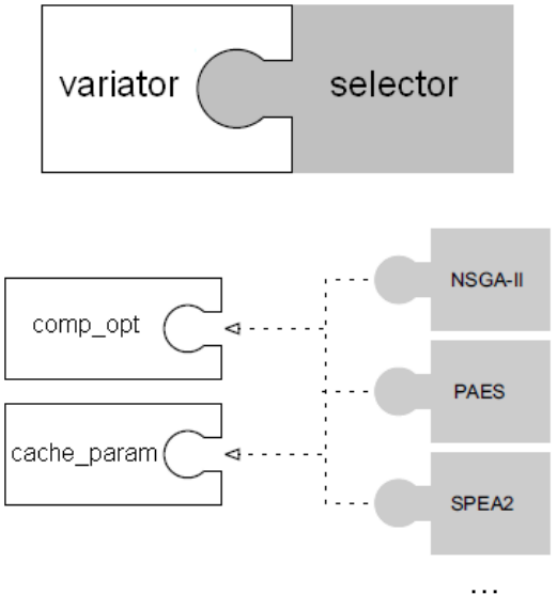


Abbildung 2.9: Zusammenspiel von Variator und Selektor in PISA, nach [BLTZ03]

Für die multikriterielle Exploration steht ein hervorragendes Werkzeug zur Verfügung, das PISA Framwork. *PISA* steht für „**P**latform and Programming Language **I**ndependent Interface for **S**earch **A**lgorithms“ und wurde am Computer Engineering and Networks Laboratory der Eidgenössischen Technischen Hochschule Zürich entwickelt [BLTZ03]. Es bietet ein Interface, welches die multikriterielle Optimierung verschiedener Probleme mit verschiedenen evolutionären Algorithmen vereinfacht.
Der Prozess der Optimierung ist bei PISA in zwei Module aufgeteilt. Der *Variator* umfasst die Problembeschreibung sowie die Operationen zur Variation der Lösung. Der *Selektor* enthält den Optimierungsalgorithmus, der die Eltern-Lösungen für die kommende Generation auswählt.
Wie in Abbildung 2.9 dargestellt, erlaubt es der modulare Aufbau beliebige Problemstellungen mit beliebigen Algorithmen zu kombinieren und so einfach und automatisiert Optimierungen durchzuführen. Auf der PISA-Homepage stehen eine Vielzahl von Variatoren und Selektoren zum Download zur Verfügung. Als Entwickler hat man außerdem die Möglichkeit, eigene Module nach bestimmten Vorgaben zu erstellen. Diese Vorgaben beschreiben die nötigen Schnittstellen, um mit allen anderen Modulen kombiniert werden zu können.

Die Module *comp_opt*(Abschnitt 3.3) und *cache_param*(Abschnitt 4.2) stellen die

Problembeschreibungen für die Optimierungen in dieser Diplomarbeit dar, und wurden entsprechend den Vorgaben für PISA erstellt. Das Monitor-Modul ermöglicht es, die erzeugten Individuen zu protokollieren, und gibt die am Ende der Optimierung entstandene Pareto-Front aus.

3 Multikriterielle Exploration von Compileroptimierungen

Die Vielzahl verfügbarer Compileroptimierungen macht es zu einer schwierigen Aufgaben, die bestmögliche Optimierungssequenz zu finden. Neben der Auswahl der Optimierungen spielt auch die Reihenfolge eine Rolle. Eine unüberschaubar große Menge verschiedener Kombinationsmöglichkeiten, die nicht alle evaluiert werden könne. Genetische Algorithmen sind für diese Art von Problem die beste Methode.
Die erste Exploration untersucht die Auswirkungen verschiedener Sequenzen von Compileroptimierungen auf eine Auswahl von Benchmarks. Es wird untersucht, wie sich die einzelnen Optimierungen auf die untersuchten Kriterien WCET, ACET und Energie auswirken und in welchem Zusammenhang die Kriterien zueinander stehen.
Die multikriterielle Optimierung wird mit Hilfe eines genetischen Algorithmus durchgeführt. Es werden also sukzessiv neue Optimierungssequenzen erzeugt und auf die genannten Kriterien hin untersucht. Mit jeder neuen Generation, die der genetische Algorithmus erzeugt, entstehen erwartungsgemäß bessere Optimierungssequenzen. Besser bedeutet in diesem Kontext, dass die Werte der Kriterien WCET, ACET und Energie verringert werden.
Zum Schluss der Exploration stehen alle erzeugten Individuen mit den entsprechenden Resultaten der Evaluation zur Verfügung. Diese Ergebnisse geben Aufschluss darüber, wie sich die Kriterien im Verlauf der Exploration entwickeln (Unterabschnitt 3.5.1). Anschließend wird untersucht, wie sich einzelnen Optimierungen auf Laufzeit und Energieverbrauch auswirken (Abschnitt 3.2). Die Pareto-Front der Resultate ermöglicht eine Abschätzung des möglichen Optimierungspotentials, verglichen mit den Standardoptimierungsstufen (Unterabschnitt 3.5.3). Zuletzt werden einzelne Benchmarks die zu besonderen Resultate geführt haben genauer betrachtet (Abschnitt 3.6)
Im Folgenden wird näher auf die verwendeten Compileroptimierungen und den PISA-Variator eingegangen, um im Anschluss ein Überblick über den Ablauf der Exploration geben zu können.

3.1 Compileroptimierungen im WCC

Der WCC verfügt über eine Vielzahl verschiedener Compileroptimierungen, von denen einige speziell auf die Optimierung der WCET ausgerichtet sind. Hier werden typische ACET-Optimierungen verwendet, wie sie in Compilern wie dem gcc vorhanden sind. Zum Einsatz kommen 21 Standard High-Level-Optimierungen, die auf der ICD-C IR ausgeführt werden können (siehe Tab. 3.1, linke Spalte). Drei dieser Optimierungen (*Loop-*

High-Level Optimierungen	**Low-Level Optimierungen**
Loop Collapsing	Loop invariant code movement
Create multiple function exit points	Constant Propagation
Dead Code Elimination	Machine specific peephole optimization
Loop deindexing	Local instruction sceduling preRA
Function Inlining	Local instruction sceduling postRA
Local CSE Elimination	
Merge identical Constants	
Optimize Constant Code	
Optimze Loop statements	
Optimze sibling and tail recursive calls	
Value Propagation	
Remove unused Arguments	
Remove unused Returns	
Remove unused Symbols	
Struct scalarization	
Simplify ++/– and redundant casts	
Function specialization	
Split distinct lifetime ranges of loc. variables	
Transform head controlled loops	
Loop unrolling	
Loop unswitching	

Tabelle 3.1: Bei der Exploration verwendete Compileroptimierungen

Unrolling, *Function-Specialization* und *Function-Inlining*) sind parametrisiert und erlauben jeweils 4 Ausprägungen. So ergeben sich 30 verschiedene Optimierungen auf der IR-Ebene.
Weiter sind im WCC einige Low-Level-Optimierungen enthalten. Diese sind jedoch für die TriCore-Architekturen TC1796/TC1797 implementiert. Zum Zeitpunkt dieser Arbeit waren für den ARM7TDMI noch nicht ausreichend viele Optimierungen auf LLIR-Ebene verfügbar. Aus diesem Grund werden 5 der für den TriCore angedachten Optimierungen mit Hilfe des arm-elf-gcc ausgeführt (siehe Tab. 3.1, rechte Spalte).
Durch diesem Umweg entsteht eine Einschränkung bezüglich der Variationsmöglichkeiten. Der arm-elf-gcc bietet keine Möglichkeit die Reihenfolge der verwendeten Optimierungen zu bestimmen. Das hat zur Folge, dass die in den Sequenzen festgelegte Reihenfolge der Low-Level-Optimierungen nicht berücksichtigt wird und keinen Einfluss auf den erzeugten Code hat. Dies reduziert die effektive Anzahl an Kombinationen für die Lowlevel-Optimierungen.

In der Summe werden 35 Compileroptimierungen untersucht. Hinzu kommt die Verwendung eines Dummys, der für keine Optimierung steht. Die Komplexität des Lösungsraumes in weiterhin enorm. Unter Berücksichtigung der oben beschriebenen Kombinations-

möglichkeiten ergibt sich folgende Anzahl:

$$31^{31}\ (High - Level) * 5!\ +\ 1\ (Lowlevel)\ \approx\ 1.707\ *\ 10^{46}$$

3.2 PISA-Selektor nsga2

Auf der Internetpräsenz von PISA [pis11] stehen zahlreiche Selektoren zur Auswahl, die Algorithmen beinhalten mit denen die genetische Analyse durchgeführt werden kann. Für diese Arbeit fiel die Wahl auf NSAG2 (The **N**ondominated **S**orting **G**enetic **A**lgorithm 2 [DAPM94]). NSGA2 gehört aktuell zu den effizientesten Algorithmen für multikriterielle Probleme. Es ist die Fortentwicklung NSGA [DAPM94] und umfasst die Sortierung von Individuen nach den Prinzip der Pareto-Dominanz und der sogenannten *Crowding-Distanz*. Dieser neue Parameter ermöglicht eine besserer Streuung der Individuen.
NSGA2 wurde bereits für die Exploration von Compileroptimierungen verwendet, und hat im Vergleich zu anderen Algorithmen (IBEA [ZK04] und SPEA-2 [ZLT01]) die besten Resultate erzeugt [LPF+10].

3.3 PISA-variator comp_opt

Die Definition der Problemstellung und die Variation und Evaluation der Individuen ist beim PISA-Framework die Aufgabe des Variators (vgl. Abschnitt 2.6). Diese Arbeit greift auf den Variator *comp_opt* zurück, der von Lokuciejewski für die zweidimensionale Exploration von Compileroptimierungen für den Tricore Prozessor entwickelt wurde [LPF+10]. Im Zuge der vorliegenden Arbeit wurde der Variator erweitert, um eine Exploration über drei Dimensionen für den ARM7TDMI durchzuführen. Die wichtigsten Komponenten von *comp_opt* werden im Folgenden beschrieben.

3.3.1 Kodierung der Compileroptimierungen

Bei dem in dieser Arbeit verwendeten genetischen Algorithmus werden die zu untersuchenden Optimierungssequenzen durch Individuen mit Lösungsvektoren in Form von Zeichenketten repräsentiert. Jedes Individuum der genetischen Analyse entspricht einer Sequenz von Compileroptimierungen. Jedes Chromosom (Zeichen) steht für eine Position dieser Sequenz und kodiert die Optimierung, die an dieser Stelle aufgeführt werden soll. Die Möglichkeit an einer Position keine der Optimierungen anzuwenden wird mittels eines Dummys kodiert. In einem Individuum sind somit die verwendeten Compileroptimierungen sowie die Reihenfolge der Ausführung kodiert.

3.3.2 Evaluation der Optimierungssequenzen

Für die Ermittlung der Werte der drei untersuchten Kriterien, wird die Zeichenkette in eine Ausführungsanweisung für den WCC transformiert. Die Zeichenkette ist in separate Teile für High- und Low-Level-Optimierungen getrennt. Aus jedem Zeichen wird ein Kommandozeilenflag generiert, das die entsprechende Optimierung im WCC aktiviert. Der WCC führt die High-Level-Optimierungen dann in der angegebenen Reihenfolge aus. Die Low-Level-Optimierungen werden als Parameter an den arm-elf-gcc weitergegeben. Der detaillierte Ablauf der Evaluation der Kriterien WCET, ACET und Energie wurde bereits in 2.3.1 - 2.3.4 beschrieben. Die Resultate der Evaluation werden mit den Individuen gespeichert.

3.3.3 Variation der Individuen

Die Individuen werden mit den in 2.5.2 und 2.5.2 beschriebenen Operationen variiert. Dabei kann man zwischen verschiedenen Art von Crossover wählen (One-Point- und Multi-Point-Crossover). Die Mutation kann ein oder beliebig viele Gene variieren. Dabei wird die Mutation selbstverständlich separat auf die Sequenzteile der High- und Low-Level-Optimierungen angewendet. Mit zahlreichen weiteren Parametern, lässt sich festlegen mit welcher Wahrscheinlichkeit einzelne Operationen angewendet werden.

3.4 Ablauf der Exploration

Der Ablauf der Exploration entspricht im Kern dem Prozess eines genetischen Algorithmus (siehe 2.5.2). Das PISA-Framework reguliert diesen Prozess und protokolliert die Ergebnisse. Abbildung 3.1 stellt die Abfolge dar.

Ablauf der Exploration:

1. Ermittlung des Referenzwertes
   Zu Beginn werden alle Benchmarks ohne die Verwendung der zu untersuchenden Compileroptimierungen übersetzt (Optimierungsstufe -O0). Die dabei ermittelten WCET-, ACET- und Energiewerte dienen als Referenz für die im weiteren Verlauf evaluierten Werte.

2. Erzeugung der initialen Population
   Die erste Generation von Individuen wird randomisiert erzeugt und stellt die Ausgangsbasis dar. Ein Individuum repräsentiert eine feste Anzahl von Highlevel- und Lowlevel-Optimierungen, die wie in Unterabschnitt 3.3.1 beschrieben als Zeichenkette kodiert sind.

3. Kompilierung und Profiling
   Für jedes Individuum werden die Benchmarks mit dem WCC compiliert. Dabei wird die im Individuum kodierte Sequenz von Compileroptimierungen angewendet.

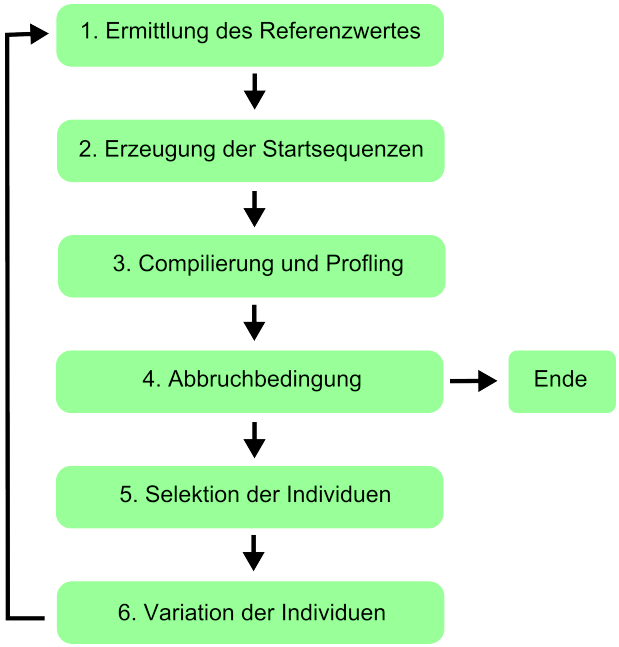


Abbildung 3.1: Ablauf der Exploration der Compileroptimierungen

Die vom WCC ermittelten WCETs, ACETs und Energiewerte zu jedem Benchmark werden aufsummiert und die prozentuale Fitness im Vergleich zum Referenzwert gespeichert.

4. Abbruchbedingung
   Ist die festgelegte Obergrenze an Generationen erreicht, wird die Exploration beendet und die aktuelle Pareto-Front ausgegeben. Diese enthält die pareto-optimalen Individuen der Exploration.

5. Selektion der Individuen
   Der in PISA verwendete Selektor wählt die Individuen aus, die als Eltern-Individuen der nächste Generation dienen. Dies wird anhand der evaluierten Werte für die drei Kriterien entschieden. Dabei wird versucht die besten Individuen (bezüglich der Pareto-Dominanz) zu übernehmen und dabei auch ein breites Spektrum an unterschiedlichen Individuen zu erhalten.

6. Variation der Individuen
   Die Individuen der neuen Generation werden mit Hilfe der in Abschnitt 2.5.2 erläuterten Operationen variiert. Dabei bleibt die Anzahl der kodierten High- und Lowlevel-Optimierungen gleich.

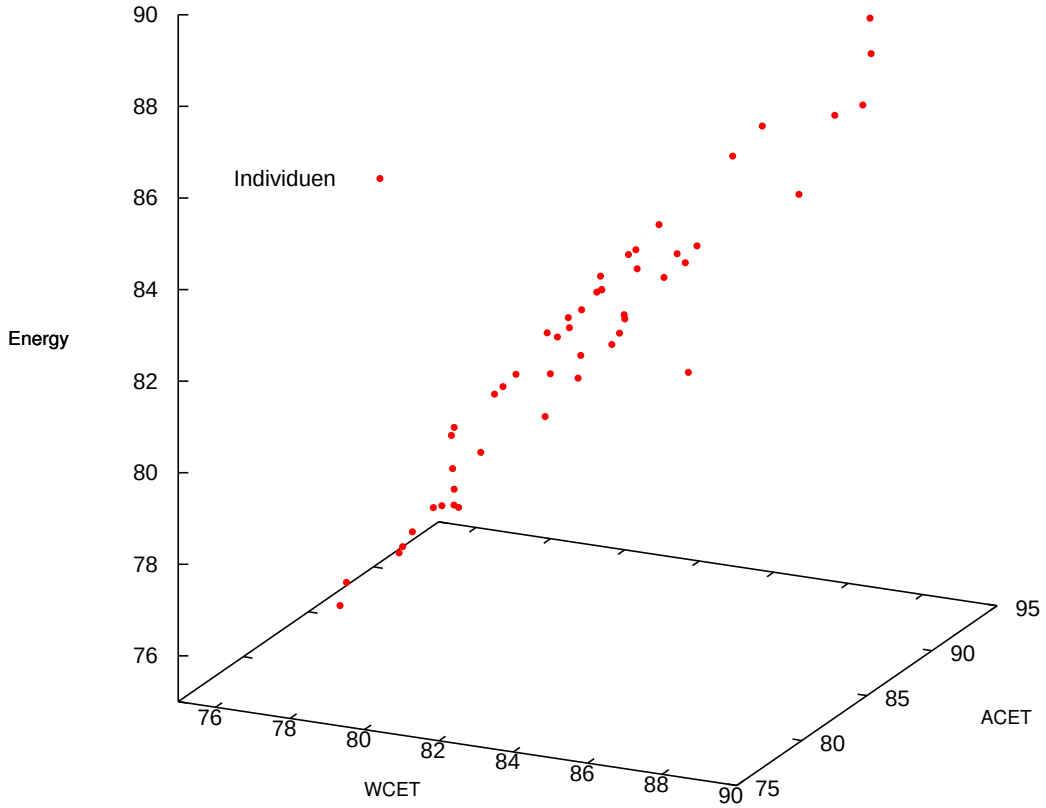


Abbildung 3.2: Beispiel einer dreidimensionalen Darstellungen der Fitness der Individuen. Werte in Prozent, relativ zu einem Referenzwert.

3.5 Resultate

Nach Abschluss einer Exploration mit PISA stehen die in jeder Generation erzeugten Individuen und ihre Werte bezüglich Laufzeit und Energieverbrauch zur Verfügung. Für die Darstellung der Individuen und stehen mehrere Möglichkeiten zur Verfügung. Da ein Individuum Werte zu drei Kriterien umfasst, bietet sich eine Abbildung in einem dreidimensionalen Punktdiagramm an (siehe Abb. 3.2). Diese Art der Darstellung ermöglicht einen schnellen Überblick über die Individuen, macht jedoch das genaue Ablesen der Werte an den Achsen unmöglich. Aus diesem Grund werden die Resultate in dieser Arbeit in zweidimensionalen Diagrammen präsentiert, die jeweils die Werte für zwei Kriterien enthalten (siehe Abb. 3.3).
Um das Verhältnis zwischen allen Kriterien darzustellen sind drei solcher Grafiken nötig. Da diese jedoch viel Platz einnehmen, wird soweit es sinnvoll ist, nur ein Diagramm mit den Werten zu zwei Kriterien gezeigt. Der Leser hat dann die Möglichkeit im Anhang die übrigen Diagramme einzusehen. Der Anhang enthält die vollständigen Ergebnisse zu

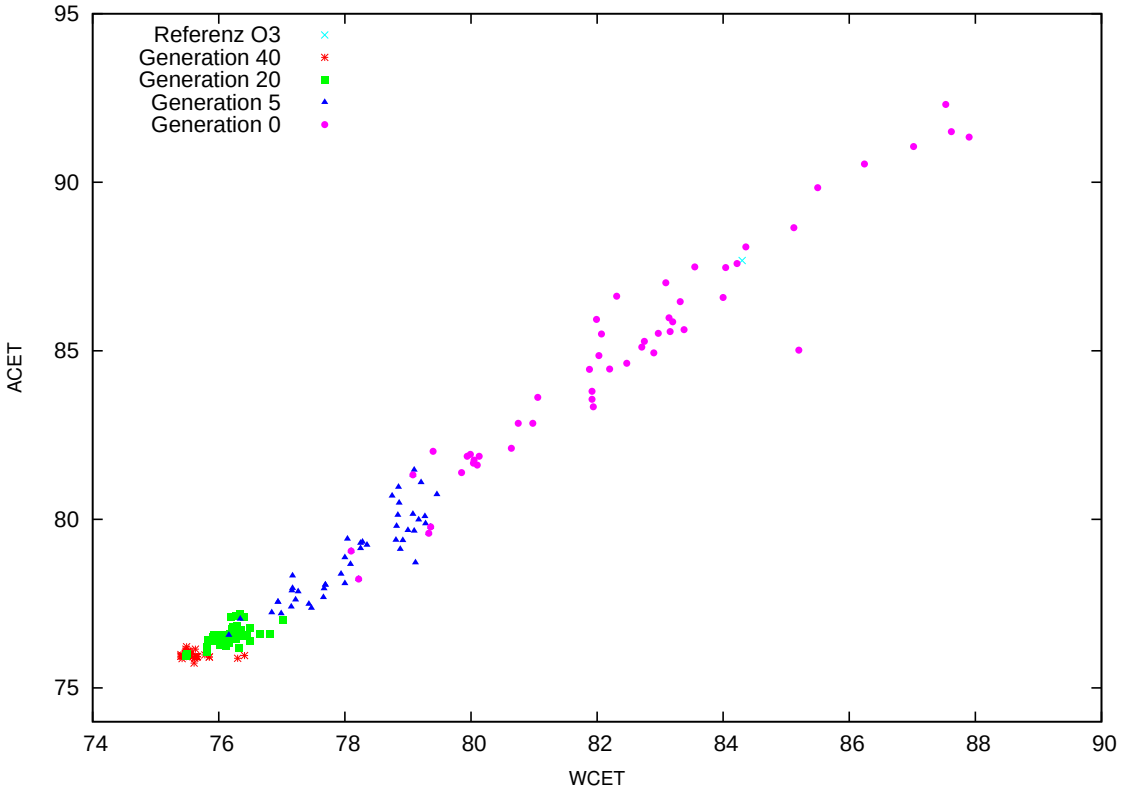


Abbildung 3.3: Exploration aller Optimierungen über alle Benchmarks: WCETs und ACETs in Prozent relativ zum Referenzwert

allen durchgeführten Explorationen.
Es folgen in Unterabschnitt 3.5.1 die Resultate einer ersten vollständigen Exploration über alle Benchmarks. Anschließend werden in Abschnitt 3.2 zwei einzelne Benchmarks untersucht und zuletzt in Unterabschnitt 3.5.3 die pareto-optimalen Lösungen analysiert.

3.5.1 Vollständige Exploration

Zu Beginn wird eine Exploration durchgeführt, die alle Benchmarks und alle Compileroptimierungen umfasst. Es wurden 44 Generationen erzeugt, die jeweils 50 Individuen beinhalten. Abbildung 3.3 (Anhang 3.1) zeigt die Fitness Werte in den Generationen 0, 5, 20 und 40 erzeugten Individuen für die Kriterien WCET und Energie. Wie bereits erwähnt lassen sich an den Achsen die erziehlten Werte der Kriterien relativ zum Referenzwert ohne Optimierungen (Optimierungsstufe O0) ablesen. Als zusätzliche Referenz repräsentiert ein Punkt die Werte, die bei der Übersetzung mit Optimierungsstufe O3

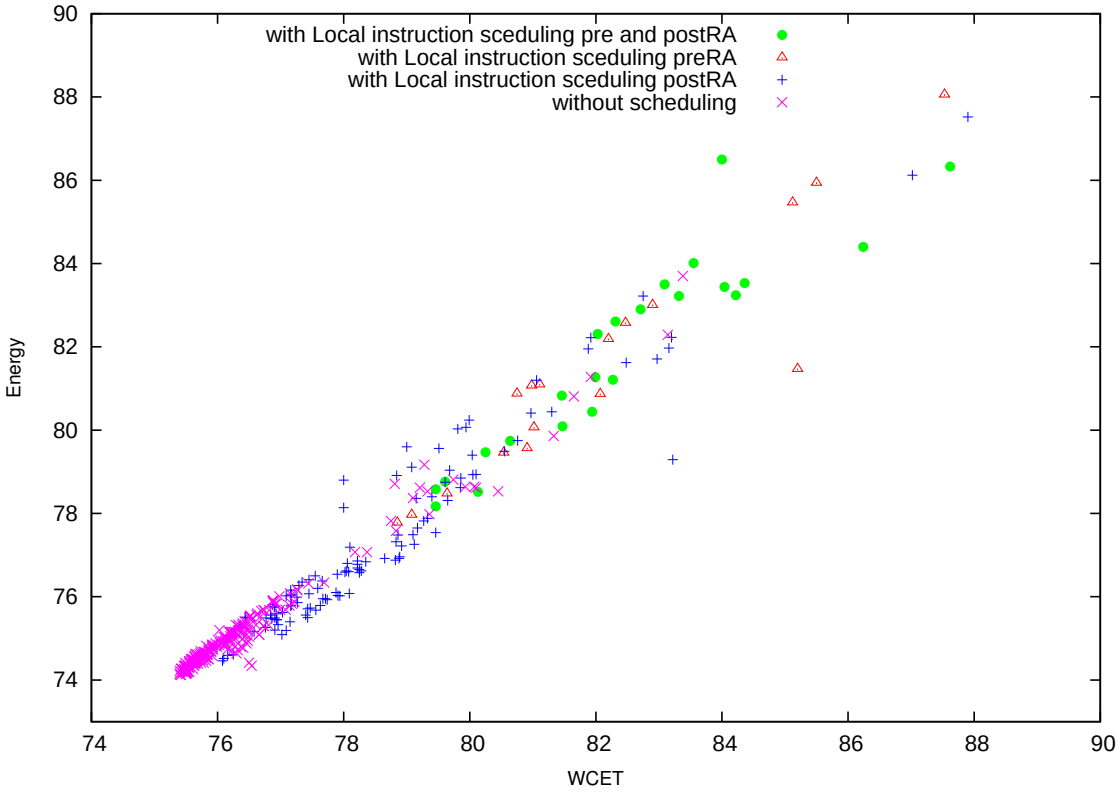


Abbildung 3.4: Exploration aller Optimierungen über alle Benchmarks: WCETs und Energiewerte in Prozent relativ zum Referenzwert

entstehen.
Bereits in der ersten Generation werden Verbesserungen erreicht, mit relativen Werten bis zu 78.22% (WCET), 78.23% (ACET) und 76.86% (Energie). Im Verlauf der Exploration werden immer bessere Individuen erzeugt, deren Werte stetig sinken. Nach der 40. Generation werden keine nennenswerten Verbesserungen mehr erreicht. Im Vergleich zu den Werten die mit der Standard-Optimierungsstufe O3 erreicht werden (WCET: 84.30%, ACET: 87.68%, Energie: 83.13%), sind die bei der Exploration gefundenen Optimierungssequenzen deutlich besser. Generation 40 ereicht Werte um 75.48% (WCET), 75.74% (ACET) und 74.12% (Energie). Verglichen mit O3 sind erzielen die bei der Exploration gefundenen Lösungen also eine weitere Verringerung der WCET um 8,82% (ACET: 11,94%, Energie: 8,93%).

Vergleicht man die Entwicklung der einzelnen Kriterien untereinander dann fällt auf, dass sich WCET, ACET und der Energieverbrauch nahezu synchron zueinander verändern (siehe Abb. 3.3 und 3.4). Die auf eine Minimierung der ACET zielenden Optimie-

rungen wirken sich also genauso positiv auf die WCET und den Energieverbrauch aus. Ein ähnliches Verhalten für ACET und WCET wurde bereits in Lokuciejewski [LPF+10] präsentiert. Die hier gewonnenen Resultate zeigen, dass dies ebenso für die Beziehung zwischen ACET und Energieverbrauch gilt.

Einfluss einzelner Compileroptimierungen

Die bisherigen Resultate zeigen, dass die Auswahl der Optimierungen großen Einfluss auf die Kriterien hat. Nun soll untersucht werden, wie sich einzelne Optimierungen auf die Kriterien auswirken. Dazu werden die in der Exploration erzeugten Individuen darauf untersucht, welche Optimierungen bei ihnen angewendet werden. Im Prozess evolutionärer Optimierung überleben die Individuen mit guten Fitness Werten und werden möglichst weiter verbessert. Mit hoher Wahrscheinlichkeit werden Compileroptimierungen, die zu schlechten Werten führen, im Verlauf der Exploration nur noch selten oder gar nicht mehr in Individuen zu finden sein. Besonders gute Optimierungen hingegen sollten bis zu den letzten Generationen in den Individuen aktiv sein. Auf diese Art kann können einzelne Compileroptimierungen bewertet werden. Anhand der Ergebnisse lassen sich die Optimierungen in folgende Klassen einteilen:

- Zur ersten Klasse gehören Optimierungen, die einen überwiegend positiven Einfluss auf die Kriterien haben. Solche Optimierungen reduzieren die Werte aller drei Kriterien deutlich. Daher lassen sich ab einer bestimmten Generation nur noch Individuen finden, die diese Optimierungen verwenden. Abbildung 3.5 (Anhang 3.2) verdeutlicht dies für die Optimierung *Loop Deindexing*. In dem Diagramm sind alle Individuen enthalten, die im Laufe der Exploration erzeugt in jeder Generation evaluiert. Dabei sind die Individuen gesondert markiert, die *Loop Deindexing* enthalten. Deutlich zu sehen ist, dass die Individuen mit dieser Optimierung die besten Ergebnisse erzielen. Diese Ergebnisse belegen zwar nicht, dass die Optimierungen in allen Benchmarks Verbesserungen erzielen, sie lassen aber den Schluss zu, dass die Optimierungen überwiegend positiven Einfluss haben Daher sollten diese Optimierungen unbedingt Teil einer Optimierungssequenz sein, die auf die Minimierung der drei Kriterien hinzielt. Folgende Optimierungen gehören zu dieser Klasse:
  - *Loop Deindexing*
  - *Local Common Subexpression Elimination*
  - *Unused Argument Removal*
  - *Head Loop Transformation*
  - *Loop-invariant Code Motion*
- Die zweite Klasse bilden Optimierungen, die auf den ersten Blick keinen erkennbar positiven oder negativen Einfluss auf die Kriterien haben. Hierbei lassen die Resultate keine eindeutige Aussage zu (siehe Abb. 3.6, Anhang 3.3). Dafür kann es mehrere Gründe geben:

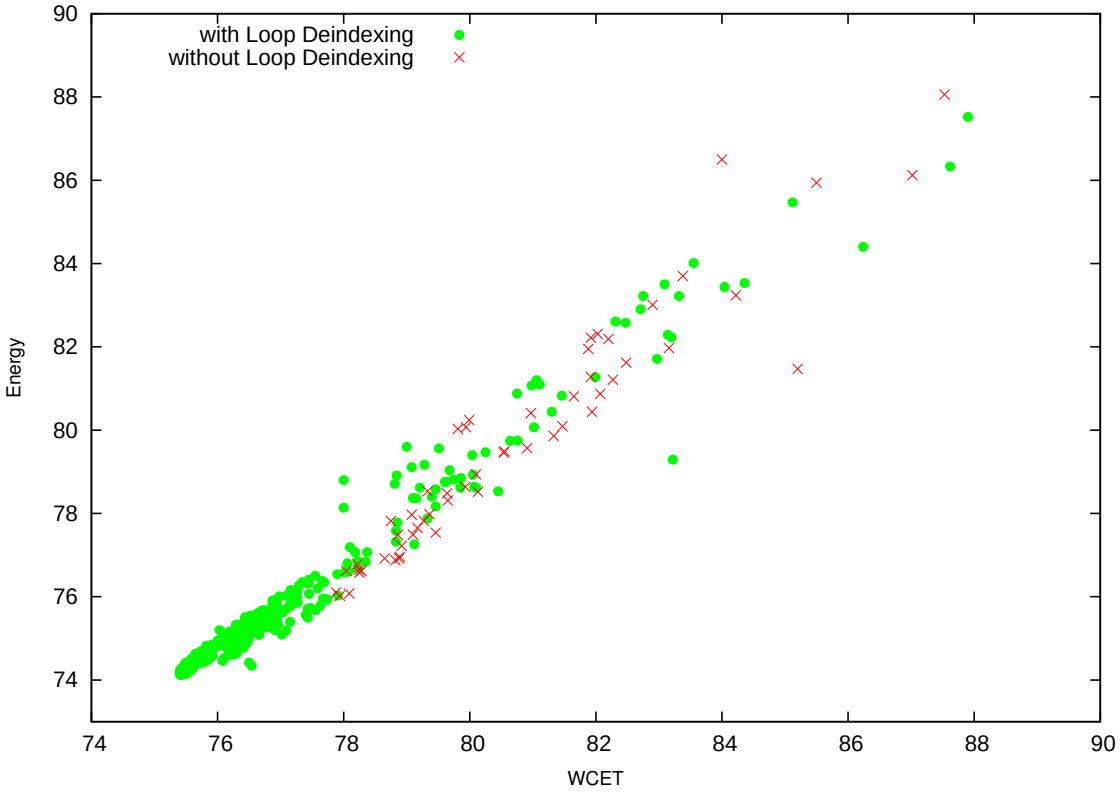


Abbildung 3.5: Einsatz von *Loop Deindexing* in allen erzeugten Individuen: WCETs und Energiewerte in Prozent relativ zum Referenzwert

- Die Optimierungen haben teils positive und teils negative Auswirkungen auf einzelne Benchmarks, die in der Summe nicht mehr auszumachen sind.
- Die Optimierungen können auf die meisten Benchmarks nicht angewendet werden, oder haben nur minimalen Einfluss auf die Kriterien.
- Bei den Optimierungen spielt die Position in der Optimierungssequenz eine Rolle und es entstehen so verschiedene Werte.
- Die Optimierungen sind darauf ausgerichtet die Codegröße zu verringern und haben keinen Einfluss auf die untersuchten Kriterien.

Der Einfluss dieser Optimierungen muss daher auf andere Wege ergründet werden, indem z.B. einzelne Benchmarks untersucht werden. Zu dieser Klasse gehören:

- Loop collapsing

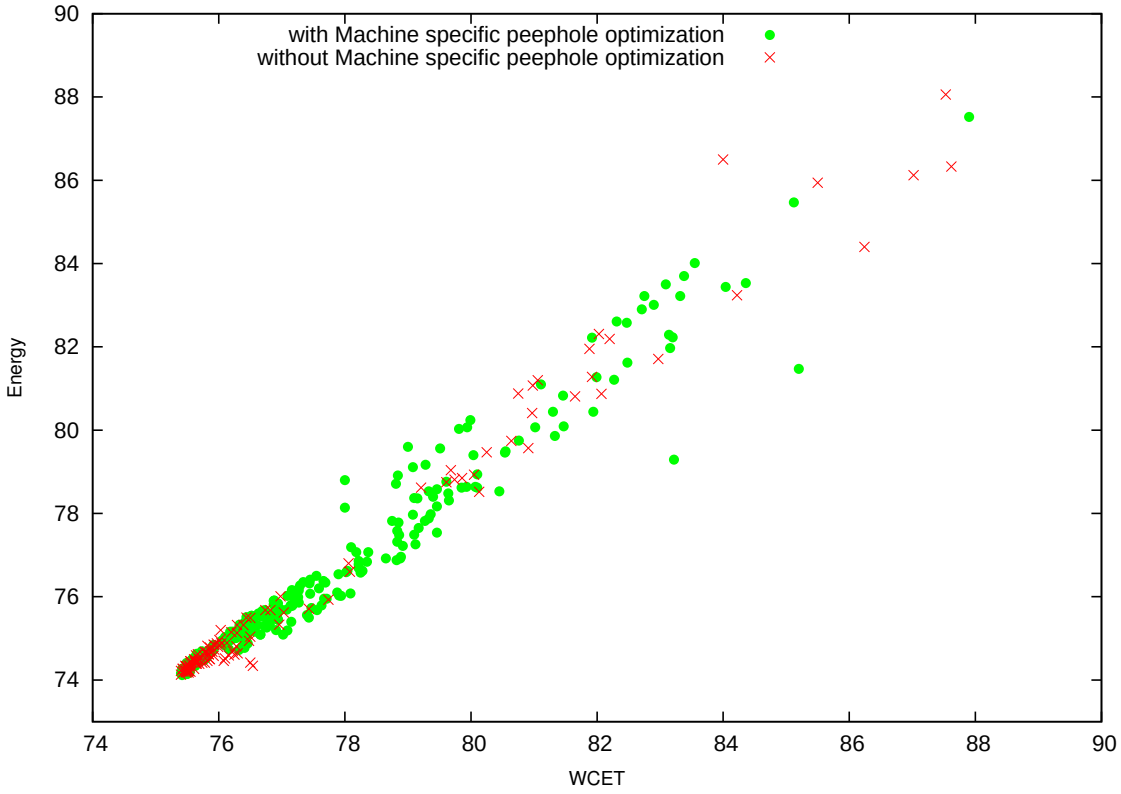


Abbildung 3.6: Einsatz von *peephole optimization* in allen erzeugten Individuen: WCETs und Energiewerte in Prozent relativ zum Referenzwert

- Multiple function exit point creation
- Dead code elimination
- Identical constant merge
- Constant code optimization
- Optimization of loop statements in loop nests
- Optimization of sibling and tail recursive calls
- Unused return removal
- Unused symbol removal
- Struct scalarization
- Simplification of ++/– and redundant casts
- Split distinct lifetime ranges of local variables
- Loop unswitching

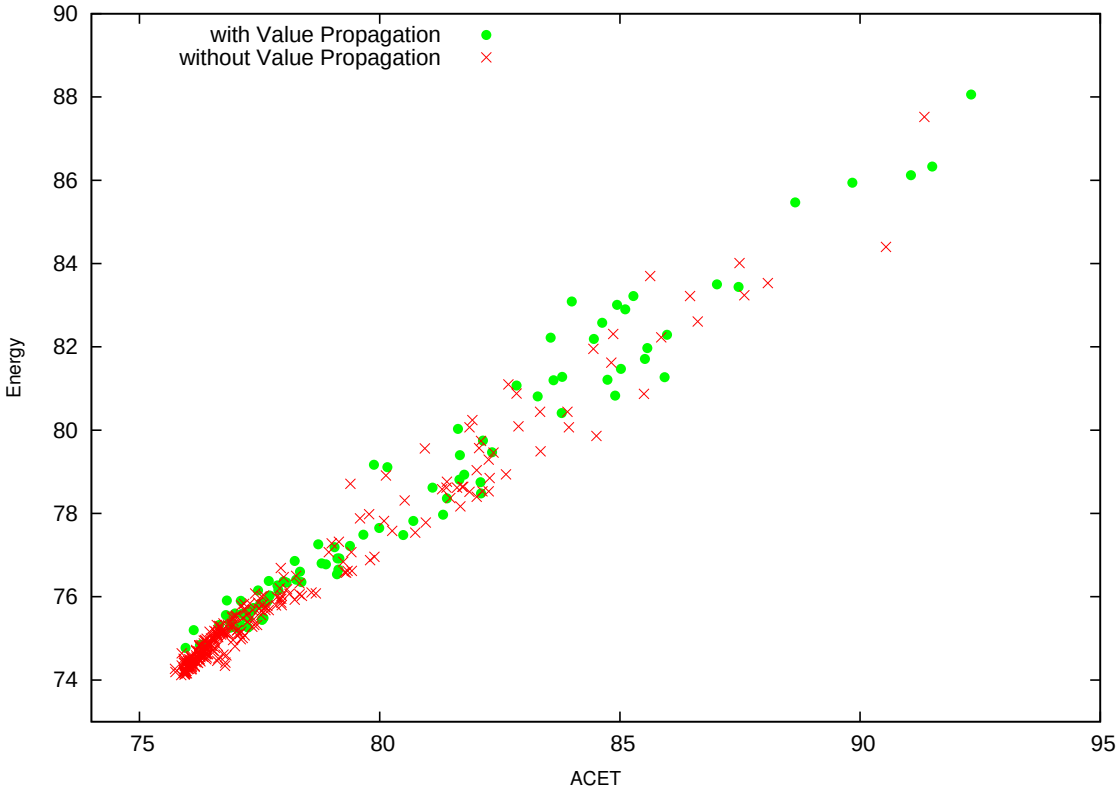


Abbildung 3.7: Einsatz von *Value Propagation* in allen erzeugten Individuen: WCETs und Energiewerte in Prozent relativ zum Referenzwert

  - Constant propagation
  - Peephole optimization

- Die dritte Klasse bilden Optimierungen die den Resultaten nach negativen Einfluss haben. Dies ist in dieser Exploration bei *Value Propagation* der Fall. Wie in Abb. 3.7 (Anhang 3.4) zu sehen, lassen sich in den letzten Generationen überwiegend Individuen ohne diese Optimierung finden, die besten Individuen werden ohne *Value Propagation* erzeugt.
  Interessante Beobachtungen lassen sich auch beim *Instruction Scheduling* machen. Der arm-elf-gcc ermöglicht separates *Instruction Scheduling* vor und nach der Registerallokation (*Pre-RA Scheduling* und *Post-RA Scheduling*). Somit ergeben sich vier Kombinationsmöglichkeiten (inklusive deaktivierten *Instruction Scheduling*). Die Resultate (Abb. 3.8, Anhang 3.5) zeigen, dass *Instruction Scheduling* vor der Registerallokation zu deutlich schlechteren Ergebnissen führt. Gute Individuen verwenden nur *Post-RA Scheduling*, die besten Individuen wurden sogar ohne *Instruc-*

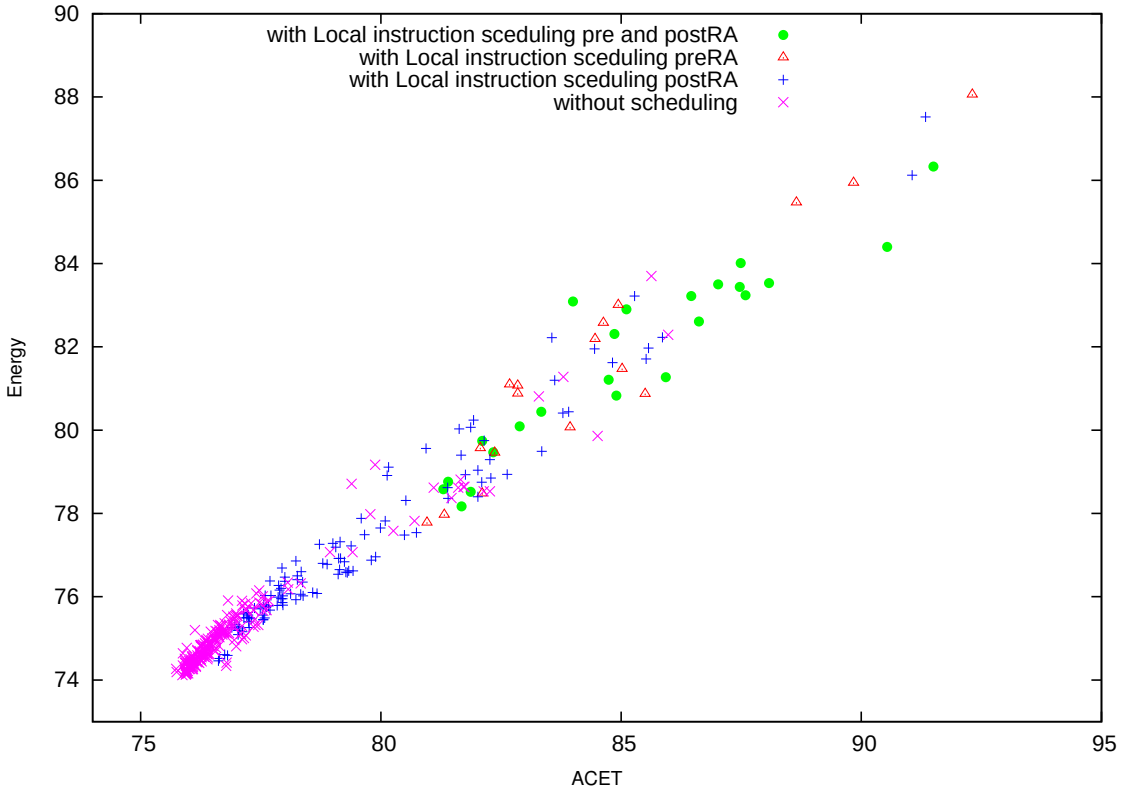


Abbildung 3.8: Einsatz von *Instruction Scheduling* in allen erzeugten Individuen: WCETs und Energiewerte in Prozent relativ zum Referenzwert

tion Scheduling erzeugt.
In Kapitel 3.6 wird genauer untersucht, wie die negativen Auswirkungen für diese beiden Optimierungen zustande kommen

Parameterisierbare Optimierungen

Function Inlining, *Function Specialization* und *Loop Unrolling* besitzen Parameter, mit denen die Optimierung gesteuert werden kann. Hierbei ist es also zusätzlich von Interesse wie sich die einzelnen Parameter auswirken. Daher werden die parameterisierbaren Optimierungen hier gesondert betrachtet.
Function Inlining und *Function Specialisation* sind Highlevel-Optimierungen für die ICD-C und ihnen kann ein Grenzwert für die Anzahl der in einer Funktion enthaltenen C-Expressions angegeben werden. Es werden dann nur diese Funktionen optimiert, die diesen Grenzwert nicht überschreiten.
Für *Loop Unrolling* gibt der Parameter die Anzahl der C-Expressions an, die eine Schlei-

fe enthalten darf, nachdem sie ausgerollt wurde. Schleifen werden also immer nur bis zu dieser Grenze optimiert. Für alle drei Optimierungen werden 4 verschieden Grenzwerte untersucht (20, 50, 100 und 200 C-Expressions).

Bei der Erzeugung und Variation der Individuen kommt es häufig vor, dass eine Optimierung mehrfach und mit verschiedenen Parametern in der Optimierungssequenz auftaucht. Das macht es schwierig, die Auswirkungen der einzelnen Parameter zu unterscheiden. Um dennoch Aussagen treffen zu können, liegt den Auswertungen die Annahme zu Grunde, dass höhere Grenzwerte die niedrigen dominieren. Wenn z.B. *Loop Unrolling* mit einem Grenzwert von 100 Instruktionen in einem Individuum vorkommt, dann überlagert das den Effekt von Aufrufen mit 20 oder 50 als Grenzwert. Für jedes Individuum wird also nur der Aufruf einer Optimierung mit dem höchsten Grenzwert betrachtet. Dieser Grenzwert ist dann ausschlaggebend für die Auswirkungen auf die Kriterien. Dies hat aber auch zur Folge, dass Individuen häufiger höheren Grenzwerten zugeordnet werden als niedrigen, unabhängig davon, wie oft die Grenzwerte tatsächlich vorkommen. Ein Übergewicht in der Anzahl von Individuen mit hohen Grenzwerten spricht daher nicht unbedingt dafür, dass sich diese Grenzwerte im Laufe der Exploration durchgesetzt haben.

Während der Analyse der Auswirkungen der verschiedenen Parameter stellte sich heraus, dass die Behandlung der Parameterwerte bei den drei parametrisierten Optimierungen im WCC fehlerhaft implementiert wurde. Bei mehrfacher Ausführung der Optimierung wurden nicht die gewünschten Werte verwendet, sondern ein Defaultwert. Daher wurde die Implementierung für die folgenden Auswertung und alle folgenden Explorationen korrigiert.

Abbildung 3.9(Anhang 3.6) zeigt den Einfluss der verschiedenen Grenzwerte bei der Optimierung mit *Function Specialization*. Die Resultate zeigen wie erwartet bei den drei Optimierungen eine Mehrheit an Individuen mit hohen Parameterwerten. Neben den Individuen mit einem Grenzwert von 200 sind viele Lösungen mit 100, 50 und vereinzelt auch mit 20 und gänzlich ohne *Function Specialization* zu finden. Diese Verteilung ist wenig aussagekräftig und kann durchaus aus den oben beschriebenen Gründen eine „zufälligen“ Verteilung entsprechen. Dennoch ist die Verwendung von *Function Specialization* zu empfehlen, da keine negativen Auswirkungen festzustellen sind. Spielt die Codegröße keine große Rolle, dann kann sollte der Grenzwert 200 verwendet werden.

Die Resultate für *Function Inlining* und *Loop Unrolling* lzeigen eine deutlichere Tendenz als bei *Function Specialization*. Hier ist die Überzahl an Individuen mit einem Parameterwert von 200 in den besten Lösungsvektoren so hoch, dass eine zufällige Verteilung als Erklärung nicht mehr in Frage kommt (siehe Abb. 3.10, Anhang 3.7). Diese zwei Optimierungen ermöglichen deutliche Verbesserungen in den Kriterien, besonders wenn der höchste Grenzwert verwendet wird. Tatsächlich wird die Menge der untersuchten Sequenzen von den zwei höchsten Grenzwerten dominiert. Demnach sind *Function Inlining* und *Loop Unrolling* eindeutig in die Kategorie 1 einzuordnen. Die beste mit beiden Optimierungen erzeugte Lösung erzielt Werte von 74.1%(WCET), 75.27%(ACET) und 72.77%(Energie). Individuen ohne *Loop Unrolling* kommen in den Generationen gar nicht vor, die einzige ohne *Function Inlining* erzeugte Lösung erreicht nur 91.65%(WCET),

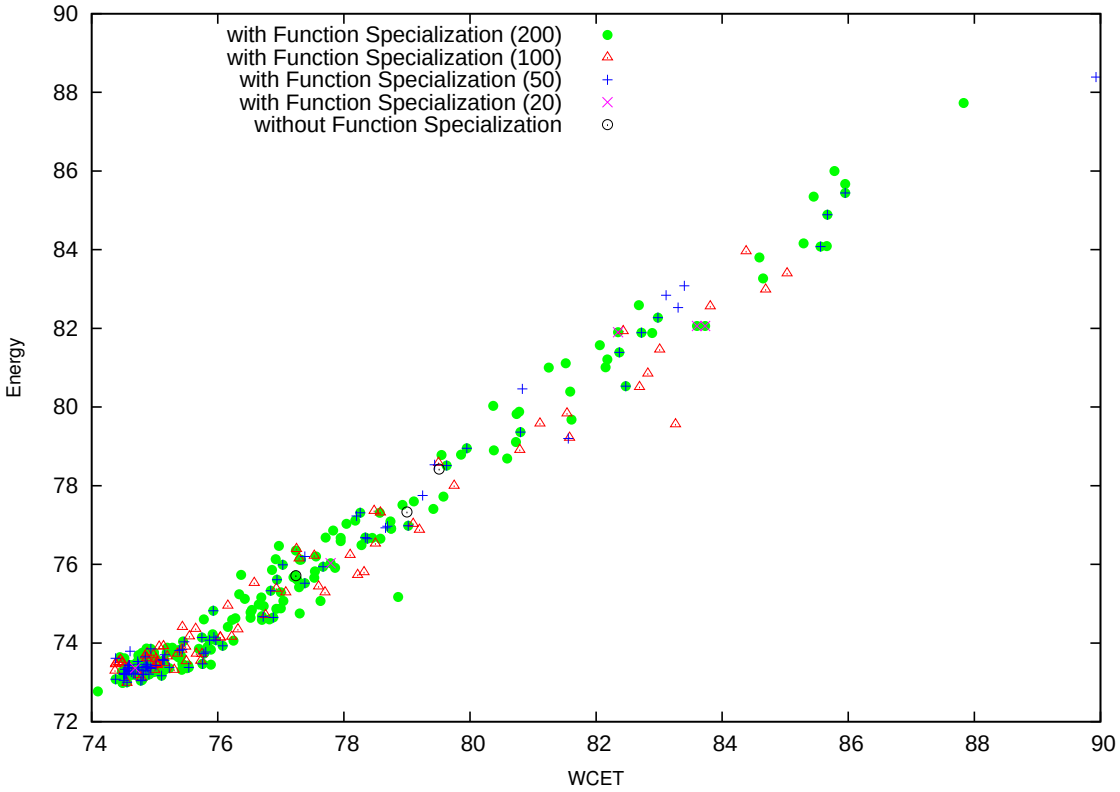


Abbildung 3.9: Einsatz von *Function Specialization* in allen erzeugten Individuen: ACETs und Energiewerte in Prozent relativ zum Referenzwert

93.43%(ACET) 90.58%(Energie).

Die oben dargestellten Erkenntnisse über den Einfluss einzelner Optimierungen auf die drei Kriterien basieren auf der Exploration über alle Benchmarks. Die untersuchten Werte sind die Summen der einzelnen Werte aller Benchmarks. Diese Art der Betrachtung erlaubt generelle Aussagen über die Auswirkungen auf Software für eingebettete Systeme. Jedoch können hierbei die individuellen Auswirkungen auf einzelne Benchmarks verdeckt werden. Im Folgenden werden einzelne Benchmarks einer Exploration unterzogen, um zum einen die Kategorisierung der Optimierungen zu verifizieren. Andererseits ist es so möglich, Effekte zu finden, die in der Summe der gesamten Benchmarks nicht auszumachen sind.

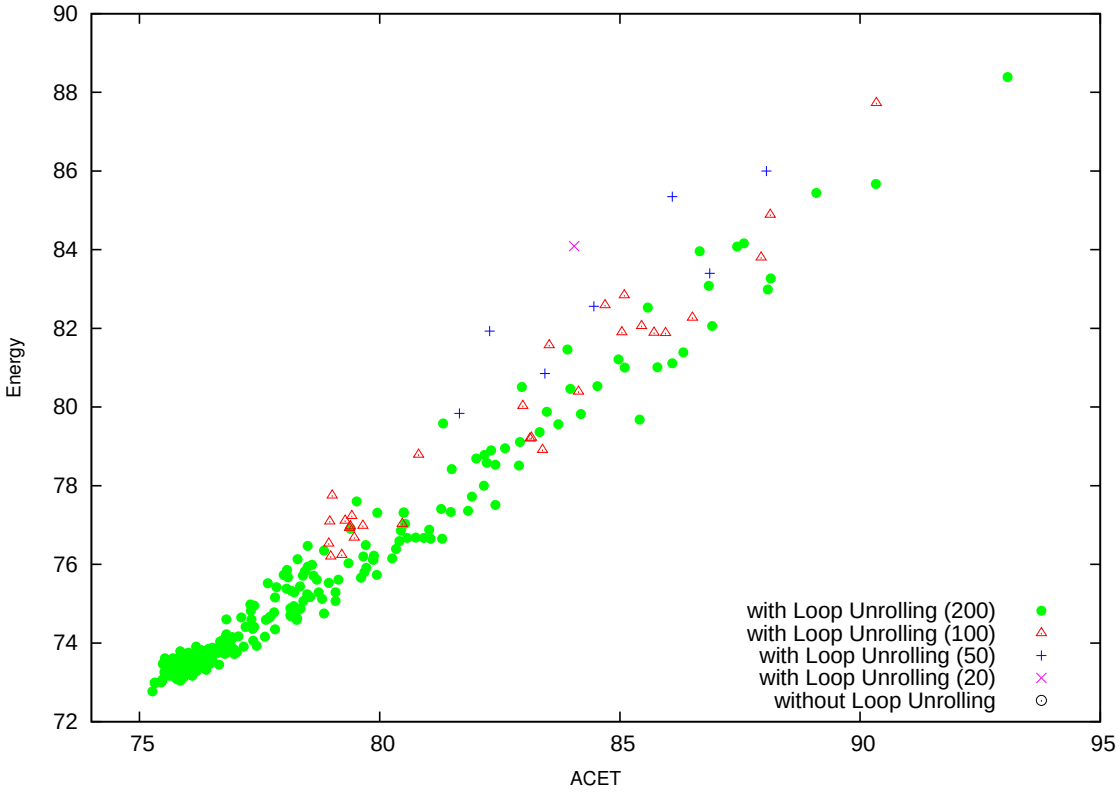


Abbildung 3.10: Einsatz von *Loop Unrolling* in allen erzeugten Individuen: ACETs und Energiewerte in Prozent relativ zum Referenzwert

3.5.2 Exploration einzelner Benchmarks

Für diese Arbeit wurden separate Explorationen mit 6 Benchmarks über 30 Generationen durchgeführt. Es folgt die Resultate zweier Benchmarks, bei denen es zu aussagekräftigen Ergebnissen gekommen ist. Es wird überprüft, ob die Optimierungen der ersten Klasse wieder einen deutlich positiven Einfluss aufweisen. Zum anderen werden die Aussagen zu den Optimierungen der zweiten und dritten Klasse erörtert. Die Darstellungen in diesem Kapitel enthalten alle in den 30 Generationen untersuchten Individuen.

Benchmark fdct

Der Benchmark *fdct (Fast Discrete Cosine Transformation)* enthält Berechnungen aus der Bildverarbeitung, in der die diskrete Kosinustransformation angewendet wird. Innerhalb zweier Schleifen werden zahlreiche mathematische Operationen auf einem Array von Integerwerten durchgeführt. *fdct* ist eine eher simpler Benchmark und bietet nicht viel

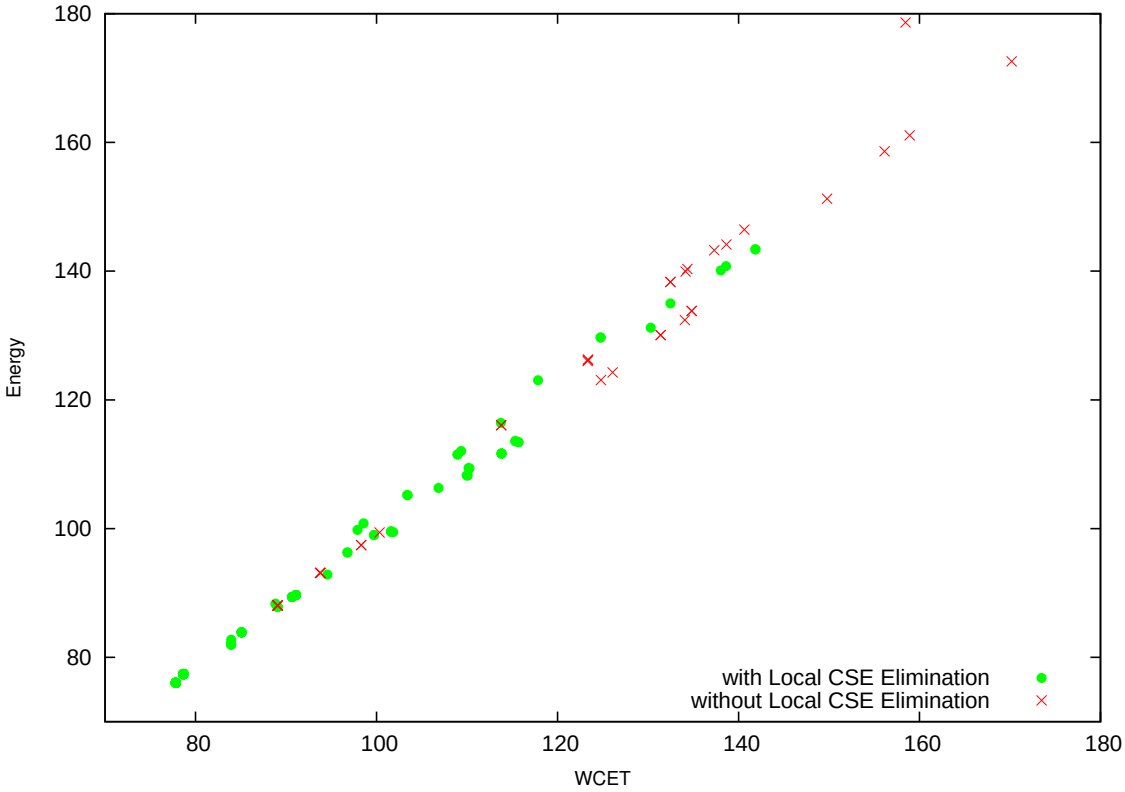


Abbildung 3.11: Benchmark fdct: Verwendung von *Local CSE Elimination* in allen erzeugten Individuen: WCETs und Energiewerte in Prozent relativ zum Referenzwert

Potential für Optimierungen. Er wird hier dennoch untersucht, weil viele Benchmarks in der verwendeten Kollektion diesem Typ entsprechen und er auussagekräftige Resultate zeigt.
Aufgrund des geringen Optimierungspotentials haben nur wenige der Optimierungen aus Klasse 1 einen positiven Einfluss. Verbesserungen lassen sich nur bei *Local CSE Elimination*(siehe Abbildung 3.11, Anhang 3.8) und *Head Loop Transformation* erkennen. Das beste Individuum mit *Local CSE Elimination* erzielt Werte von 77.8%(WCET), 78.88%(ACET) und 76.05%(Energie). Ohne diese Optimierung sind nur 89.07%(WCET), 87.93%(ACET) und 88.06%(Energie) möglich. Die parametrisierten Optimierungen zeigen hier keine Wirkung.

Der Benchmark veranschaulicht jedoch sehr deutlich die negativen Auswirkungen der *Value Propagation* und des *Instruction Schedulings*. Wie in Abbildung 3.12 (Anhang 3.9) zu sehen ist, entstehen große Unterschiede bei der Verwendung der unterschiedlichen

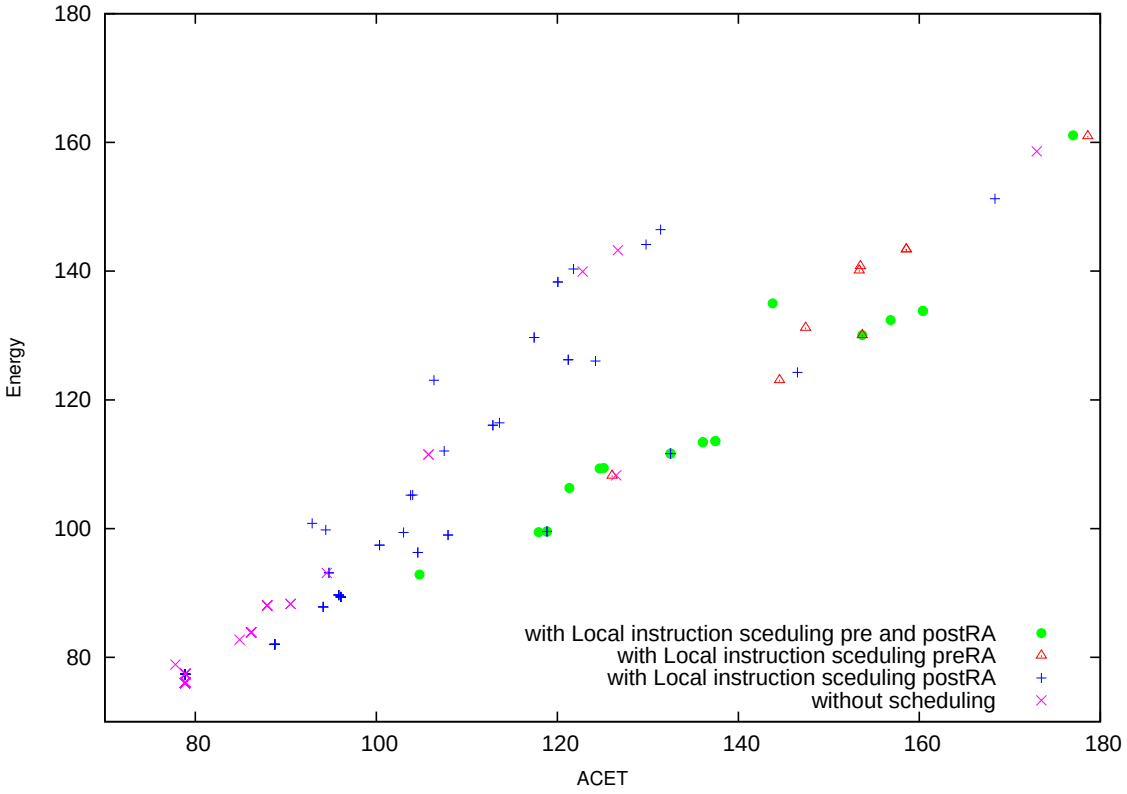


Abbildung 3.12: Benchmark fdct: Verwendung von *Instruction Scheduling* in allen erzeugten Individuen: ACETs und Energiewerte in Prozent relativ zum Referenzwert

Scheduling-Arten. Sobald *Pre-RA Scheduling* angewendet wird verschlechtert sich die Laufzeit und der Energieverbrauch. Verbesserungen sind nur mit *Post-RA Scheduling* möglich, wobei die besten Lösungen ohne jede Art von *Instruction Scheduling* entstehen. Dieser Effekt ist bei zahlreichen Benchmark zu beobachten.
Der Benchmarks *fdct* bestätigt die Annahme, dass *Value Propagation* die Werte der Kriterien verschlechtert kann. Wie in Abbildung 3.13 (Anhang 3.10) sehr gut zu sehen ist, grenzen sich die Individuen mit dieser Optimierung von denen ohne ab. Verbesserungen der Laufzeit und des Energiebedarfes sind nur ohne diese Optimierung möglich.

Benchmark ndes

Als zweites wird ein Benchmark untersucht, der eine höhere Codegröße, mehrere Funktionen und eine Vielzahl von Schleifen aufweist. *ndes* enthält typischen Code für einge-

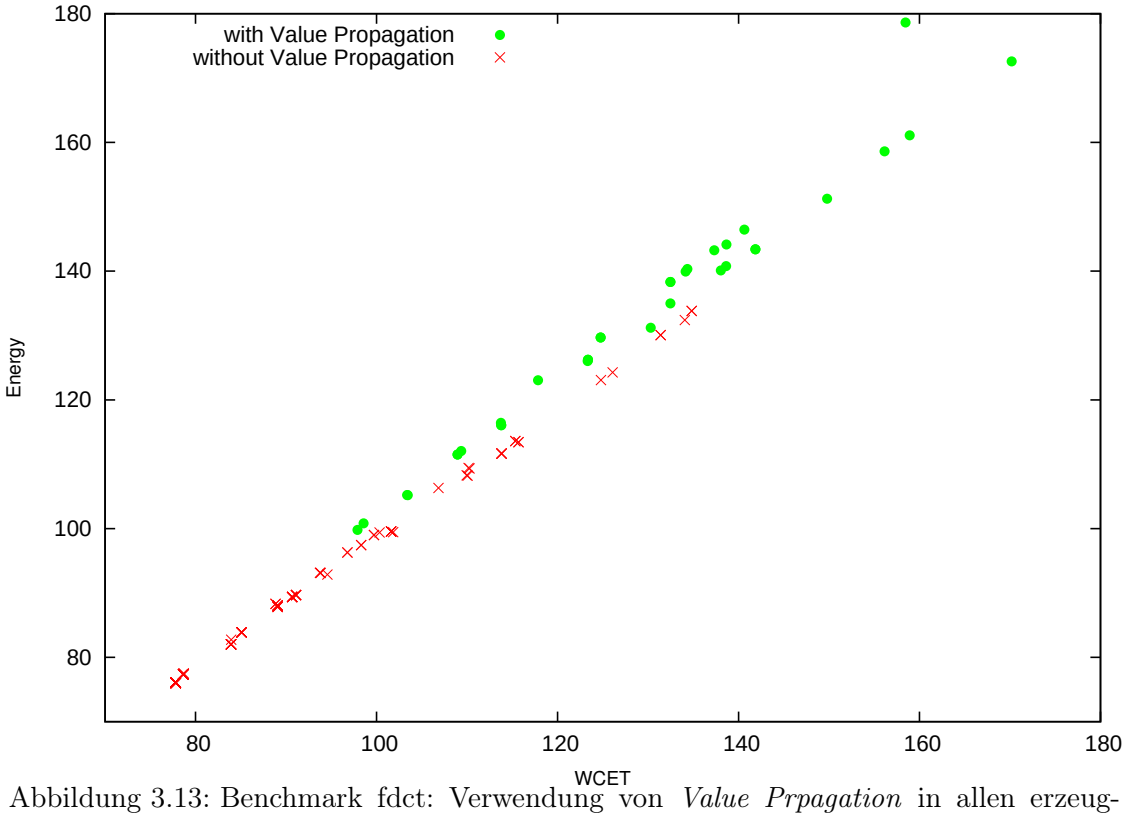


Abbildung 3.13: Benchmark fdct: Verwendung von *Value Prpagation* in allen erzeugten Individuen: WCETs und Energiewerte in Prozent relativ zum Referenzwert

bettete Systeme mit vielen Bit-Manipulationen, Shift und Berechnungen auf Array und Matrizen.
Das *Instruction Scheduling* des arm-elf-gcc wirkt sich hier noch schlechter aus als beim Benchmark zuvor. Es entstehen zwei völlig separate Gruppen von Individuen, je nachdem ob Post-RA Scheduling verwendet wird (siehe Abb. 3.14, Anhang 3.11).
Ansonsten heben sich kaum Optimierungen hervor. Lediglich *Loop Unrolling* und *Struct Scalarization* zeigen eindeutig positive Auswirkungen. Mit *Struct Scalarization* ist tritt hier also eine Optimierung positiv in den Vordergrund, die in Kategorie 2 geführt wird. Ein gutes Beispiel, das dafür spricht die Optimierungen aus Klasse 2 generell zu verwenden.

3.5.3 Bewertung der Pareto-optimalen Lösungen

Das für die Exploration verwendete Set von Benchmarks (siehe 3.2, oberer Teil) stellt ein Trainingsset dar. Um festzustellen ob, die pareto-optimalen Lösungen allgemein gute

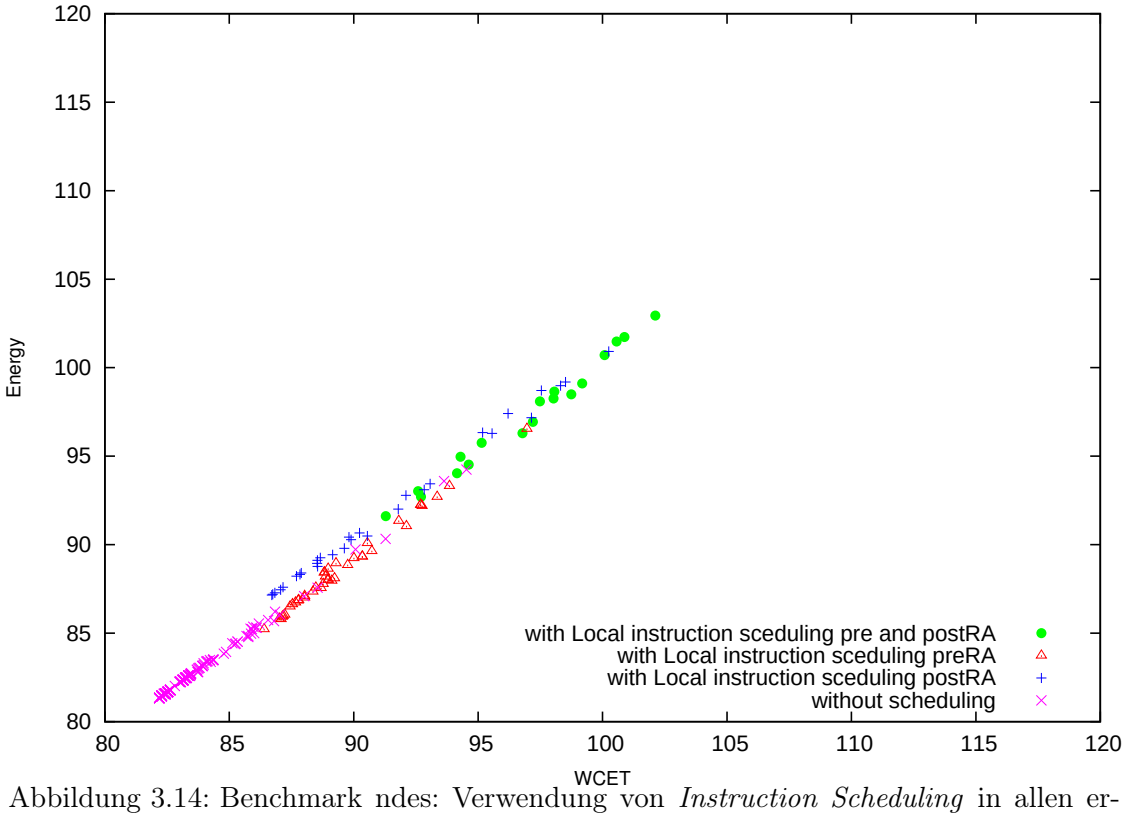


Abbildung 3.14: Benchmark ndes: Verwendung von *Instruction Scheduling* in allen erzeugten Individuen: WCETs und Energiewerte in Prozent relativ zum Referenzwert

Lösungen sind, müssen sie an einer disjunkten Menge von Benchmarks erprobt werden, dem Testset (siehe 3.2, unterer Teil). Erzielen die Optimierungssequenzen auch für dieses zweite Benchmark-Set gute Resultate, so kann man sie als allgemein gute Lösungen betrachten.
Von allen evaluierten Optimierungssequenzen hat sich eine Sequenz ein wenig von den anderen abgehoben. Sie erreichte Werte von 74.1% (WCET), 75.27% (ACET) und 72.77% (Energie). Diese Werte sind ein wenig besser als die vieler anderen Individuen, die in den letzten Generationen ermittelt wurden. Dieser Abstand ist sehr gering, weniger als 0,5% im Verhältnis zum Referenzwert. Deshalb sollte man diese Lösung nicht als die alleinstehend beste betrachten, sondern eher als eine von vielen guten.
Angewendet auf das Testset, erzielt diese Sequenz ebenfalls sehr gute Resultate und erreicht Verbesserungen von 73.16% (WCET), 79.41% ACET)und 67.27% (Energie). Die Standardoptimierungsstufe O3 mit den Werten 81.02% (WCET), 79.02% (ACET), 72.89% (Energie) wird in WCET und Enerige übertroffen, die ACET ist ein wenig höher. Hier stellt sich die Frage in wieweit die Reihenfolge der Compileroptimierungen für die

Trainingsset
complex_multiply_fixed, convolution_fixed, complex_update_fixed, dot_product_fixed, iir_biquad_one_section_fixed, lms_fixed, n_real_updates_fixed, real_update_fixed, startup_fixed, complex_multiply_float, complex_update_float, convolution_float, dot_product_float, lms_float, n_complex_updates_float, iir_biquad_one_section_float, n_real_updates_float, real_update_float, binarysearch, cover, crc, fac, fdct, fft1, insertsort, janne_complex, jfdctint, lcdnum, ludcmp, ndes, petrinet, qurt, sqrt, statemate, adpcm, g721.marcuslee_encoder, g721.marcuslee_decoder, iir_1_1, lmsfir_8_1, mult_4_4, qmf_receive, qmf_transmit, v32.modem_achop, v32.modem_cnoise, v32.modem_eglue
Testset
fft_16_7, fft_16_13, fir_fixed, fir2dim_fixed, matrix3_fixed, fir_float, fir2dim_float, matrix1x3_float, n_real_updates_floats, bsort100, compressdata, countnegative, expint, fibcall, ludcmp, matmult, prime, qsort-exam, select, st, compress, edge_detect, fft_256, fir_32_1, fir_256_64, histogram, iir_4_64, latnrm_8_1, latnrm_32_64, lmsfir_8_1, lmsfir_32_64, mult_10_10, spectral, v32.modem_bencode,

Tabelle 3.2: Trainingsset und Testset der verwendeten Benchmarks

guten Werte verantwortlich ist. Um das zu überprüfen werden die Benchmarks des Testset ein weiteres mal mit der pareto-optimalen Sequenz kompiliert. Diesmal wird die Reihenfolge verwendet, die in WCC festgelegt ist. Es sind also nur die verwendeten Optimierungen variiert. Interessanterweise ergeben sich noch bessere Werte: 71.04% (WCET), 74,94% (ACET) und 65.17% (Energie). Es zeigt sich also, dass die Reihenfolge einen sehr starken Einfluss auf die Kriterien hat und die im WCC verwendete Reihenfolge bessere Ergebnisse erzielt als die bei der Exploration verwendete.

3.6 Analyse einzelner Compileroptimierungen

Bei der Untersuchung der Einflüsse einzelner Compileroptimierungen hat sich herausgestellt, dass die Optimierungen *Value Propagation* und das *Instruction Scheduling* des arm-elf-gcc zu deutlich schlechteren Werten bezüglich WCET, ACET und Energie geführt hat. In diesem Kapitel wird untersucht, wie dieser Effekt zustande kommt und ob es Möglichkeiten gibt, die Implementierung dieser Optimierung anzupassen, so dass die negativen Auswirkungen verhindert werden.

3.6.1 Instruction Scheduling

Von den zweit Arten des Schedulings beim arm-elf-gcc (*Pre-RA Scheduling* und *Post-RA Scheduling*) kommt es besonders beim *Pre-RA Scheduling* zu massiven Verschlechterungen. Aber auch *Post-RA Scheduling* hat sehr oft leicht höhere WCETs, ACETs und Energiewerte zur Folge.

Als erstes soll überprüft werden, ob dieser Effekt bei allen untersuchten Benchmarks auftritt. Von den 45 untersuchten Benchmarks kommt es bei 21 Benchmarks zu Verschlechterungen in mindestens einem Kriterium (siehe Tabelle in Anhang 3.12). In vielen Fällen (z.B. bei lms_float, statemate u.a.) wird dies begleitet durch eine Erhöhung der Codegröße. Nur bei 6 Bechmarks sind positive Auswirkungen auf alle drei Kriterien zu beobachten. *Pre-RA Scheduling* hat besonders auf die ACET und den Energieverbrauch einen negativen Einfluss. Dort sind im Extremfall Verschlechterungen um bis zu 71.80%(ACET) und 38.01%(Energie) zu finden.

Ohne Instruction Scheduling

```
  .loc 1 54 0
  cmp ip, r1
  bne .L6
  .loc 1 58 0
  mov r2, lr
  mov r3, r2, asl \#1
  mov r2, r2, asl \#3
  add r3, r3, r2
  str r3, [r4, r0, asl \#2]

.L6:
```

Mit Pre-RA Scheduling

```
  .loc 1 54 0
  cmp ip, r1
  .loc 1 52 0
  str r3, [lr, r0, asl #2]
  .loc 1 54 0
  .loc 1 58 0
  moveq r2, r3, asl #3
  moveq r3, r3, asl #1
  addeq r3, r3, r2
  streq r3, [lr, r0, asl #2]
.L6:
```

Abbildung 3.15: Assembler-Codestücke des Benchmarks *ludcmp* mit und ohne *Instruction Scheduling*

Genauere Betrachtungen des vom arm-elf-gcc erzeugten Codes haben ergeben, dass durch das *Instruction Scheduling* die Behandlung von bedingten Anweisungen verändert wird. In Abbildung 3.15 ist dies exemplarisch an einem Codestück aus dem Benchmark *ludcmp* zu sehen. Die Abbildung enthält die Implementierung einer if-Anweisung, einmal ohne *Instruction Scheduling* (Abb. 3.15, links) und einmal mit *Pre-RA Scheduling* (Abb. 3.15, rechts) übersetzt wurde. Ansonsten wurden keine weiteren Optimierungen verwendet. Statt eines bedingten Sprungbefehls werden rechts die darauf folgenden Instruktionen mit Ausführungsbedingungen versehen. Das führt dazu, dass diese Instruktionen in jedem Fall durchlaufen werden, und die Ausführungsbedingung immer aufs Neue geprüft wird. Wenn die Bedingung nicht erfüllt ist, werden dauert diese Methode länger als der

Einsatz eines Sprungbefehls.

3.6.2 Value Propagation

Benchmark	WCET	ACET	Energie
crc	99.88	99.89	99.76
fac	100	99.91	99.98
fdct	151.69	139.64	162.35
fft1	97.83	92.85	94.06
janne_complex	99.17	99.29	98.49
ludcmp	97.32	98.64	97.15
qurt	99.92	99.96	99.79
sqrt	99.91	99.98	99.84
statemate	96.89	92.23	94.54
adpcm	99.91	100.14	99.8
qmf_receive	99.32	99.37	99.24
qmf_transmit	99.32	99.59	99.28
v32.modem_cnoise	97.76	97.51	99.02

Tabelle 3.3: Relative Fitness der Kriterien (in Prozent zum Referenzwert) bei der Verwendung von *Value Propagation*

Die Auswertung der Exploration aller Benchmarks hat auch gezeigt, dass die Compileroptimierung *Value Propagation* vorwiegend negative Auswirkungen auf die Kriterien hat. Der Benchmark *fdct* dient an dieser Stelle als Illustrationsbeispiel. Vergleicht man die Referenzwerte der Standardoptimierungsstufe O0 mit denen, die bei der Hinzunahme von *Value Propagation* entstehen, so sind Verschlechterungen zu beobachten: 117.93%(WCET), 112.34%(ACET) und 121.29%(Energie) im Vergleich zum Referenzwert. In Kombination mit anderen Optimierungen ist der Effekt sogar noch größer. Verglichen mit der Standardoptimierungsstufe O3 (inkl. *Value Propagation*) erhält man folgende Werte für die Kriterien wenn die Optimierung deaktiviert wird: 65.92% (WCET), 71.61% ACET) und 61.6% (Energie).
Erstaunlich sind die Resultate für die anderen Benchmarks. Auf eine Vielzahl hat *Value Propagation* gar keinen Einfluss und auf die übrigen einen eher positiven. Tabelle 3.3 zeigt die Werte verglichen mit der Optimierungsstufe O3 (Benchmarks mit unveränderten Werten sind nicht aufgeführt). Ohne die massiven Verschlechterungen beim Benchmark *fdct*, wirkt sich die Optimierung durchaus positiv aus. Eine nähere Betrachtung des Auswirkungen in diesem Benchmark ist also nötig.

Die Highlevel-Optimierung *Value Propagation* erfüllt bei diesem Benchmark seinen Zweck, indem es bekannte Werte von Variablen auf Hochsprachenebene direkt dort einsetzt, wo sie benötigt werden. Abbildung 3.16 zeigt einen für einen Codeteil aus *fdct* wie

Ohne Value Propagation	Mit ValuePropagation
`constant=(-20995);` `z2=(z2 * constant);`	`z2=((-20995) * z2);`

Abbildung 3.16: Vergleich zweier Codestücke aus dem Benchmark *fdct*

das funktioniert: Zu sehen ist eine Multiplikation zweier Variablen, von der *constant* zuvor mit einer Konstanten beschrieben wird (linker Teil). Korrekt ersetzt die Optimierung die Variable durch die entsprechende Konstante (rechter Teil). Erst bei der Betrachtung der in Assemblercode übersetzten Anweisungen wird klar, wie die erhöhten Laufzeiten zustande kommen. In Abbildung 3.17 ist zu sehen, wie sich die erzeugten Assembler-Instruktionen von den Codezeilen aus Abbildung 3.16unterscheiden (Anmerkung: Der unterschiedliche Wert der Konstante in beiden Abbildungen ist kein Tippfehler, sondern wird so vom arm-elf-gcc erzeugt). Der arm-elf-gcc speichert die Zahl nicht mehr als Konstante sondern generiert sie umständlich mit mehreren Shift und Kopier-Instruktionen. Da der Benchmark viele dieser Zuweisungen von Konstanten hat, erhöht sich die Anzahl der aufgeführten Instruktionen und damit die Laufzeit und der Energieverbrauch beträchtlich.

Ohne Value Propagation	Mit ValuePropagation
`mvn r3, #20992`	`mov r3, sl, asl #2`
`sub r3, r3, #2`	`mov r2, sl, asl #5`
`mul sl, r3, sl`	`add r3, r3, r2`
	`mov r2, r3, asl #3`
	`add r3, r3, r2`
	`rsb r3, sl, r3`
	`mov r2, r3, asl #6`
	`add r3, r3, r2`
	`rsb sl, r3, #0`

Abbildung 3.17: Vergleich zweier Übersetzungen eines Codestücks aus dem Benchmark *fdct*

3.6.3 Verbesserungspotential

Instruction Scheduling erzeugt ineffektiven Code bei der Übersetzung mit dem arm-elf-gcc. Die negativen Folgen lassen sich umgehen, indem diese Optimierung nicht angewendet wird. Denkbar wäre, das das Instruction Scheduling im arm-elf-gcc so erweitert wird, dass die betrachteten Codeteile auf eine andere Weise kompiliert werden. Es lohnt

sich, dies in Zukunft zu untersuchen, um die Verwendung des arm-elf-gcc effizienter zu machen.
Trotz der schlechten Resultate für einen Benchmark sollte *Value Propagation* im WCC verwendet werden. Vor allem, da die Probleme nicht in der Highlevel-Optimierung selbst liegen, sondern in dem vom arm-elf-gcc erzeugten Assembler-Code. Jedoch sollte man bei Programmcode, der Multiplikationen von hochwertigen Konstanten enthält, vorsichtig sein. In diesem Fall lohnt es sich die Auswirkungen der Optimierung genau zu prüfen, um große Verschlechterungen in den Kriterien zu vermeiden.

3.7 Schlussfolgerungen

Die Resultate haben gezeigt, dass der Einsatz der Optimierungen einen großen Einfluss auf die untersuchten Kriterien hat. WCET, ACET und der Energieverbrauch verändern sich dabei nahezu synchron zueinander. Die auf eine Verbesserung der ACET hinzielenden Optimierungen verbessern also gleichermaßen die WCET sowie den Energieverbrauch. In Vergleich zur Optimierungstufe O0 wurden Werte von 74.1%(WCET), 75.27%(ACET) und 72.77%(Energie) erzielt.
Bei der Übersetzung wirken sich die einzelnen Optimierungen unterschiedlich aus. Es wurde gezeigt, dass eine bestimmte Menge von Optimierungen die Kriterien so deutlich verbessert, dass sie generell eingesetzt werden soll. Sie haben positiven Einfluss auf die meisten Benchmarks und alle betrachteten Kriterien. Eine zweite Gruppe von Optimierungen zeigt nur bei einzelnen Benchmarks eine Wirkung. Da sie die Werte generell nicht verschlechtert, ist ihre Verwendung trotzdem sinnvoll. Zuletzt wurde eine kleine Menge von Optimierungen bestimmt, die zu negativen Auswirkungen geführt hat und daher nicht eingesetzt werden sollte.
Vom Einsatz des *Instruction Schedulings* vom arm-elf-gcc als Optimierung im WCC muss man nach den vorliegenden Resultaten abgeraten. Für viele Benchmark bringt die Verwendung dieser Optimierung deutlich schlechtere Resultate. Werte von 171.80%(ACET) und 138.01%(Energie) wurden dabei erreicht. Daher sollte diese Optimierung für den ARM im WCC deaktiviert werden. Die Optimierung *Value Proparation* führt bei einem der untersuchtem Benchmarks zu ineffizienterem Code. Dies kann man als Ausreißer betrachten und die Verwendung der Optimierung empfehlen. Die Möglichkeit von Verschlechterungen mit *Value Proparation* sollte aber im Hinterkopf behalten werden.
Man kann also festhalten, dass die Optimierungen im WCC generell sehr positive Auswirkungen auf die Laufzeit und den Energieverbrauch haben. Es ist jedoch nötig, sich näher mit dem arm-elf-gcc als Codeselektor auseinander zusetzten und zu versuchen die negativen Effekte zu unterbinden.
Es wurde eine pareto-optimale Sequenz gefunden, die auch in einer Prüfmenge von Benchmark zu deutlichen Verbesserungen gegenüber der Optimierungstufe O3 geführt hat. Eine zusätzliche Verringerung der Kriterien um 7,86%(WCET), 5,62%(Energie) wurde erreicht. Dies zeigt, dass eine solche Exploration sehr gut dazu geeignet ist, eine verbesserte Optimierungssequenz zu finden. Was die Reihenfolge angeht hat sich gezeigt, dass

der WCC bereits eine sehr gute Folge vorgibt. Zusammen mit den bei der Exploration gefundenen pareto-optimalen Optimierungen, lassen sich die besten Resultate erziehlen. Die durch die Exploration gefundene pareto-optimale Compileroptimierungssequenz gibt einigen Aufschluss darüber, wie eine verbesserte Optimierungssequenz für den WCC aussehen sollte. Sie enthält die drei parameterisierbaren Optimierungen (*Loop Unrolling, Function Specialization und Function Inlining*) mit jeweils dem höchsten Grenzwert von 200. Auch ist jede der Optimierungen aus Klasse 1 enthalten, wohingegen kein *Instruction Scheduling* verwendet wird. *Value Propagation* ist ebenfalls Teil der optimalen Sequenz. Diese Tatsache habt nochmal hervor, dass diese Optimierung trotz möglicher Verschlechterungen eingesetzt werden sollte.

4 Multikriterielle Exploration von Cacheparametern

In diesem Kapitel werden verschiedene Parameter unter die Lupe genommen, die die Größe und Organisation eines Cachespeichers bestimmen. Diese Parameter haben Einfluss auf die Effektivität mit der der Cache genutzt werden kann. Es wird eine Reihe von Kombinationen der Parameterwerte untersucht um festzustellen welche Werte für eingebettete Systemen sinnvoll erscheinen.
Diese Exploration untersucht die Auswirkungen der Cacheparameter auf WCET, ACET und Energieverbrauch. Variiert werden die Cacheparameter Cachegröße, Cachezeilengröße und Assoziativität. Die Analyse beschränkt sich auf die Nutzung eines Instruktions-Caches. D.h. es werden Zugriffe auf den Speicher beim Fetching von Instruktionen gecached, Datenzugriffe auf den Hauptspeicher bleiben ungecached.
Analog zu den Compileroptimierungen sollten ursprünglich auch die Cacheparameter mit Hilfe genetischer Algorithmen über viele Generationen exploriert werden. Im Zuge der Sichtung sinnvoller Werte für die drei Cacheparameter stellte sich heraus, dass die Anzahl der zu untersuchenden Kombinationen geringer ist als angenommen. Die gesamte Lösungsmenge ist klein genug um eine vollständige Exploration durchzuführen. Trotz des beschränkten Lösungsraumes wird dennoch das PISA-Framework für die Exploration herangezogen. Mit Hilfe von PISA kann die Exploration automatisiert durchgeführt und die Ergebnisse effizient gesammelt werden. Der PISA-Selektor ist dabei nicht ausschlaggebend, da er nicht angewendet wird.

4.1 Auswahl der Cacheparameter für ARM

Zu Beginn stand die Frage, welche Werte für die einzelnen Parameter mit in die Exploration einfließen sollen. Verwendet werden gängige Werte, die bei aktuellen Cacheimplementierungen für eingebettete Prozessoren zu finden sind (siehe Tabelle 4.1), sowie einige zusätzliche Grenzwerte.

Assoziativität
Auf dem Markt sind häufig Direct-Mapped Caches zu finden, sowie 2- und 4-fache Assoziativitäten. Seltener auch 8-fach assoziative Caches. Neben diesen wird für die Exploration auch die 16-fache Assoziativität untersucht.

Cachezeilengröße
In den meisten Fällen findet man Caches mit Zeilengrößen von 16 bis 32 Byte, in einigen Fällen auch 64 Byte. Diese drei Werte werden bei der Exploration berücksichtigt.

Cachegröße

Prozessor	Cachegröße	Zeilengröße	Assoziativität
Alchemy AU1000	16 KByte	32 Byte	4-fach
ARM 7	8 KByte	16 Byte	4-fach
ColdFire	0-32 KByte	16 Byte	Direct-Mapped
Hitachi SH7750S (SH4)	8 KByte	32 Byte	Direct-Mapped
Hitachi SH7727	16 KByte	16 Byte	4-fach
IBM750FX	32 KByte	32 Byte	8-fach
IBM403GCX	16 KByte	16 Byte	2-fach
Motorola MPC8240	16 KByte	32 Byte	4-fach
Motorola MPC823E	16 KByte	16 Byte	4-fach
Motorola MPC8540	32 KByte	32/64 Byte	4-fach
Motorola MPC7455	32 KByte	32 Byte	8-fach
NEC VR4181	4 KByte	16 Byte	Direct-Mapped
NEC VR4181A	8 KByte	32 Byte	Direct-Mapped
NEC VR4121	16 KByte	16 Byte	Direct-Mapped
PMC Sierra RM7000A	16 KByte	32 Byte	4-fach
SandCraft sr71000	32 KByte	32 Byte	4-fach
SuperH	32 KByte	32 Byte	4-fach
TriCore TC1796	16 KByte	32 Byte	2-fach
TriMedia TM32A	32 KByte	64 Byte	8-fach
Xilinx Virtex IIPro	16 KByte	32 Byte	2-fach
Triscend A7	8 KByte	16 Byte	4-fach

Tabelle 4.1: Übersicht über in eingebetteten Prozessoren verwendete Parameter für Instruktionscaches, unter anderem aus [ZVN05]

Bei der Suche nach typischen Cachegrößen findet man in der Literatur die verschiedensten Angaben. Mit Werten von 256 Byte bis 64 KB wird hier versucht diese Spannweite möglichst gut abzudecken. Cachespeicher unterhalb von 2 KB sind zwar selten, werden in dieser Arbeit trotzdem untersucht. Die Codegröße von einem Teil der Benchmarks ist sehr gering (unterhalb 1 KB). Der Code dieser Benchmarks findet daher in größeren Cache vollständig Platz und das Profiling zeigt dann sehr geringe Abweichungen bei den Kriterien. Daher werden für die Explorationen bereits Cachegrößen ab 256 Byte verwendet.
Hier mag der Leser anmerken, dass es für so kleinen Programmcode keinen Sinn macht Caches zu verwenden, der Code könnte problemlos in SRAM Platz finden. Es muss dazu nochmal deutlich gemacht werden, welche Benchmarks hier Verwendung finden. Es handelt sich um Codefragmente, die Teil größerer Programme sind und darin entscheidende Aufgaben erfüllen (z.B. Algorithmen zur Signalverarbeitung). Diese Codeteile werden immer wieder ausgeführt und es ist daher sinnvoll das Cacheverhalten an diesen Benchmarks zu untersuchen. Die bei der Exploration verwendeten Analysetools akzeptieren nur 2er-Potenzen als Cachegröße, damit ergeben sich 9 verschieden Byte-Werte (256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536). Insgesamt stehen so 131 mögliche

Kombinationen für die drei Cacheparameter zur Verfügung.

4.2 PISA-Variator cache_param

Der in der ersten Exploration verwendetet Variator „comp_opt“ wurde für die Exploration von Cacheparametern umgebaut. Die Funktionsweise des Variators blieb unverändert, es werden Individuen generiert und evaluiert. Die Individuen einer Generation kodieren die Cacheparameter wiederum als Zeichenketten. Eine Zeichenkette besteht aus 3 Zeichen, in jedem ist der Wert eines der Parameter kodiert. Zu Beginn werden bereits alle zu untersuchenden Individuen erstellt, daher ist eine Variation der Individuen nicht nötig, . Der Variator übergibt die zu untersuchenden Cacheparameter auch hier als Kommandozeilenparameter an den WCC. Dieser reicht die Parameter jeweils an das aiT Analsyetool und den CoMET Simulator weiter und protokolliert die ermittelten Werte.

4.3 Ablauf der Exploration

Die Verwendung des PISA-Frameworks ist wie erwähnt bei dieser Exploration nicht zwingend notwendig, vereinfacht aber die Durchführung. In diesem Ablauf wird nur die Startgeneration erzeugt und evaluiert. Es sind keine Variationen nötig, der Selektor spielt keine Rolle.

Der Ablauf der Exploration:

1. Ermittlung des Referenzwertes
   Die Benchmarks werden mit der Standardoptimierungsstufe -O1 übersetzt, wobei die im Mikrocontroller LPC2880 [Sem07] verwendete Cacheorganisation genutzt wird (Cachegröße 16384 Byte, Cachezeilengröße 16 Byte, Assoziativität 2). Die dabei ermittelten WCET-, ACET- und Energiewerte dienen als Referenz für die im weiteren Verlauf evaluierten Werte.

2. Erzeugung der Individuen
   Es werden alle möglichen Individuen erzeugt, die sich aus der Spanne der Parameter ergeben. In ihnen sind die drei Cacheparameter als Zeichenkette kodiert.

3. Kompilierung und Profiling
   Für jedes Individuum werden die gesammelten Benchmarks mit dem WCC compiliert. Das WCET Analysetool aiT liest die Parameter, die die Cachekonfiguration angeben in Form von ais-Annotationen ein. Für den CoMET-Simulator wird vor jeder Simulation das Speicherlayout der virtuellen Plattform entsprechend der zu untersuchenden Cachekonfiguration angepasst. WCET, ACET und der Energieverbrauch werden so unter Berücksichtigung der Cacheparameter ermittelt. Zuletzt wird die prozentuale Relation zum Referenzwert errechnet und gespeichert.

4. Ende
   Nach der Evaluation der ersten Generation wird die Exploration beendet.

4.4 Resultate

Die 131 verschiedenen Kombinationen von Cacheparametern wurden auf Laufzeit und Energieverbrauch untersucht, die Resultate sollen nun im Hinblick auf den Einfluss der einzelnen Cacheparameter überprüft werden. Die Darstellung der Werte ist die gleiche wie in Kapitel 3. Jeder Punkt entspricht einem Individuum, die prozentuale Relation zum Referenzwert lässt sich an den Achsen ablesen. Im Anhang sind die wiederum übrigen Diagramme oder Tabellen aufgeführt, die aus Gründen der Übersichtlichkeit nicht im Text dargestellt sind.

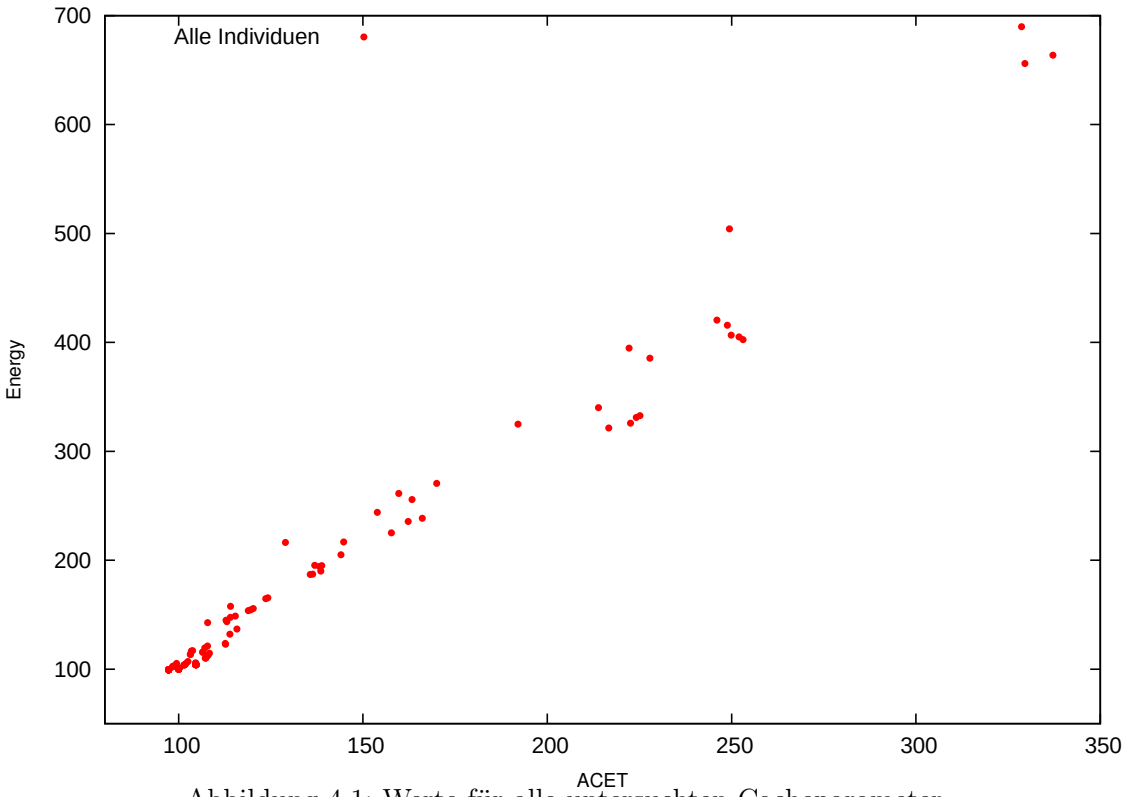


Abbildung 4.1: Werte für alle untersuchten Cacheparameter

Abbildung 4.1 (Anhang 4.1) zeigt die Resultate der Evaluation für ACET und Energie. Auffällig ist die große Streuung der ACET- und Energie-Werte mit Spanweiten zwischen 97 und 700 Prozent. Im Vergleich dazu stellen sich die WCET-Messdaten mit Werten

zwischen 91 und 150 Prozent eher kompakt dar (Anhang 4.1). Hier drängt sich die Frage auf, woher diese Unterschiede in den Auswirkungen auf die Kriterien kommen. Die Antwort ist, wie im Folgenden erläutert wird, in den unterschiedlichen Messverfahren zu suchen.

4.4.1 Cacheparameter und Messverfahren

Der Ermittlung der WCET auf der einen Seite und der ACET und dem Energieverbrauch auf der anderen Seite, liegen zwei grundsätzliche verschiedene Mess- bzw. Ermittlungsverfahren zugrunde. Diese Unterschiede spiegeln sich in den Ergebnissen wieder und es ist daher sinnvoll diesen Aspekt bei der Auswertung zu berücksichtigen.
Die durchschnittliche Laufzeit und der Energieverbrauch eines Benchmarks wird mit Hilfe des vom Analysetool CoMET erzeugten Tracefiles ermittelt. Ein genaues Hardwaremodell für die virtuelle Plattform und die zyklengenaue Simulation ermöglichen dabei exakte Angaben über das Cacheverhalten. Die Variation der Cacheparameter hat daher einen entsprechend großen Einfluss auf die bei der Simulation auftretenden Cache-Hit bzw. Cache-Misses, die Dauer der Schreib- und Lesezugriffe auf den Cache und damit auf die ermittelten Laufzeiten. Dies macht sich in der großen Spannweite der ACET- und Energiewerte deutlich.
Die obere Schranke für die Worst-Case Laufzeit hingegen wird mittels einer statischen Analyse berechnet. Statische Laufzeitanalysen sind besonders bei Berücksichtigung von Caches sehr komplex und die Ermittlung der Anzahl der Cache-Hits bzw. Cache-Misses daher sehr schwierig. Da die Analyse eine sichere obere Laufzeitschranke gewährleisten muss, wird also stets ein pessimistisches Cacheverhalten angenommen. Nur sichere Cache-Hits werden berücksichtigt, was bei Variation der Parameter zu geringeren Veränderungen in den ermittelten Laufzeitgrenzen führt. Daher weichen die WCET-Werte für die Variationen der Parameter nicht so stark voneinander ab.
In den folgenden Abschnitten werden die Auswirkungen der drei Cacheparameter separat betrachtet und interessante Resultate hervorgehoben.

4.4.2 Cachezeilengröße

Als erstes wird untersucht wie sich die Variation der Cachezeilengröße auf die drei Kriterien auswirkt. Die Analyse der verschiedenen Cachezeilengrößen ergab, dass eine Erhöhung der Zeilengröße zu höheren Werten in allen drei Kriterien führt. Der Effekt ist umso größer, je kleiner verwendeten Caches sind. Exemplarisch ist das für die WCETs bei einer Assoziativität von 1 in Tabelle 4.2 dargestellt. ACETs und Enegierwerte sind in Anhang 4.5 zu finden.
Diese Resultate sind bemerkenswert, wenn man davon ausgeht dass mit größeren Cachezeilen das Prinzip der Lokalität besser genutzt werden kann. Auswertungen der Tracefiles haben gezeigt, dass tatsächlich weniger Cache-Misses entstehen wenn die Zeilengröße erhöht wird. Diese Einsparungen werden aber durch die höhere Anzahl von Zyklen zunichte

WCET	Cachezeilengröße		
Cachegröße	16 Byte	32 Byte	64 Byte
256 Byte	104.59	118.58	146.16
512 Byte	99.43	110.61	132.97
1 KByte	94.93	104.7	123.44
2 KByte	93.84	103.05	119.93
4 KByte	92.51	101.42	117.92
8 KByte	91.29	100	116.11
16 KByte	91.29	100	116.11
32 KByte	91.29	100	116.11
64 KByte	91.29	100	116.11

Tabelle 4.2: WCETs (in Prozent relativ zum Referenzwert) für verschiedene Cache- und Cachezeilengrößen bei einer Assoziativität von 1

gemacht, die beim Befüllen einer größeren Cachezeile nötig sind. Es werden zusätzliche Instruktionen gecached, die aber im weiteren Verlauf nicht aufgeführt werden. Aus diesem Grund benötigt die Ausführung des Programmcodes insgesamt mehr Laufzeit. Dieser Effekt wirkt sich auf die ACET und den Energieverbrauch, aber besonders auf die WCET-Werte aus. Abbildung 4.2 (Anhang 4.2) zeigt alle Individuen differenziert nach der Cachezeilengröße. Die WCET-Werte für die verschiedenen Zeilengrößen sind deutlich voneinander abgegrenzt. Die Einsparungen an Cache-Misses wirken sich eher gering auf die oberen Laufzeitgrenzen aus. Die erhöhte Dauer der Zeilenfetches kommt hier besonders zum Tragen.

Auch hier muss erwähnt werden, dass diese Resultate die Auswirkungen über alle Benchmarks darstellen. Bei Stichproben einzelner Benchmarks lassen sich zum Teil abweichende WCETs finden. So zum Beispiel bei Benchmark *petrinet*. Wie in Tabelle 4.3 zu sehen ist steigen die WCET-Werte mit steigender Zeilengröße bei 2 KByte Cachegröße. Bei einer Cachegröße von 4 KByte ist das Gegenteil der Fall. Bemerkenswerterweise trifft dies nicht auf ACETs oder Energiewerte zu (siehe Anhang 4.6).

WCET	Cachezeilengröße		
Cachegröße	16 Byte	32 byte	64 Byte
1 KByte	111.37	113.86	115.85
2 KByte	111.37	113.86	115.85
4 KByte	103.31	102.31	97.76
8 KByte	102.46	100	93.09

Tabelle 4.3: WCETs (in Prozent relativ zum Referenzwert) für verschiedene Cache- und Cachezeilengrößen bei einer Assoziativität von 1

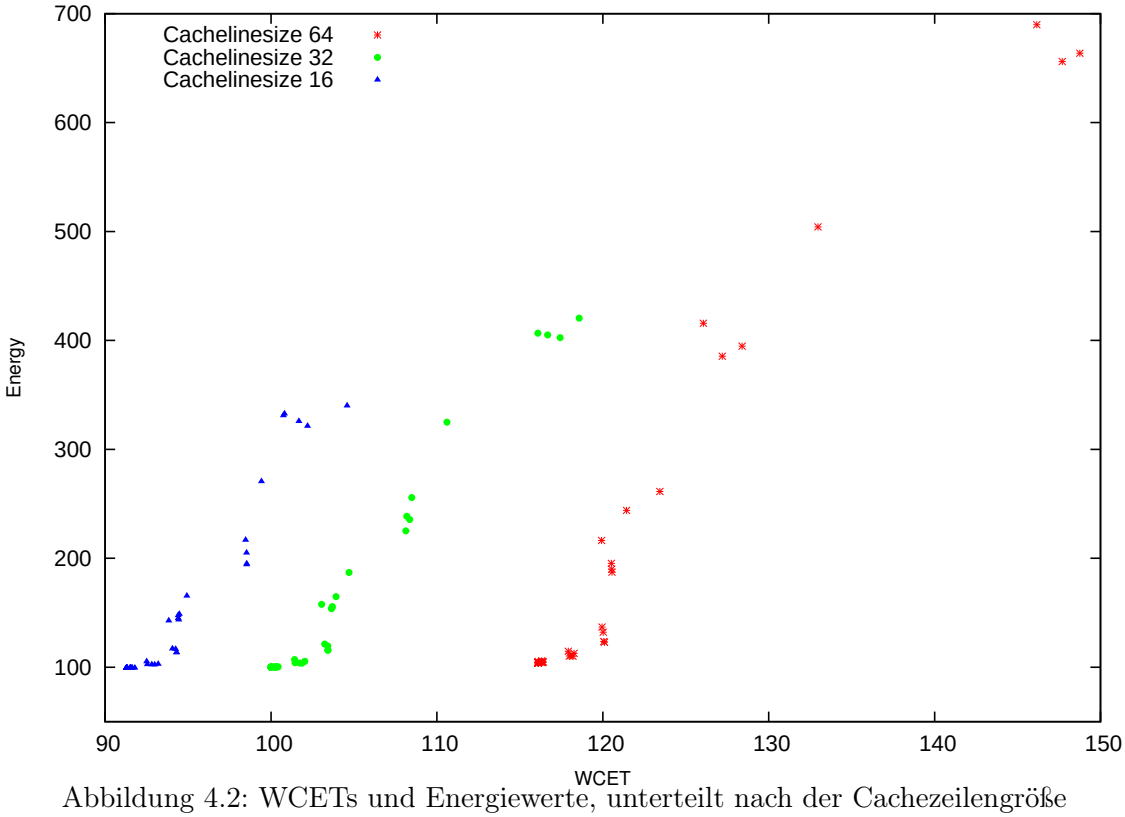


Abbildung 4.2: WCETs und Energiewerte, unterteilt nach der Cachezeilengröße

Wahl der Cachezeilengröße

Die oben getroffenen Aussagen gelten also nicht pauschal für alle Benchmarks. Eine Erhöhung der Cachezeilengröße hat, wie bereits mehrfach erwähnt, abhängig von den anderen Parametern positive oder negative Folgen für die Laufzeit und den Energieverbrauch einen Programms haben. Ob die positiven oder die negativen Folgen überwiegen hängt zum Beispiel von der Art des untersuchten Programmcodes ab. Verschiedene Aspekte wirken sich auf die Anzahl der entstehenden Cache-Hits und -Misses aus. Der Anteil des Codes, der in Schleifen liegt und deren Iterationshäufigkeit zum Beispiel sind wichtige Faktoren. Auf Instruktionen innerhalb von Schleifenkörpern wird besonders häufig zugegriffen. Befinden sich diese dauerhaft im Cache, kann dies ohne Zeitverlust durch externe Speicherzugriffe geschehen.
Ein anderer Aspekt, der in dieser Exploration nicht untersucht wurde, ist ebenso wichtig. Die verwendete Prozessorarchitektur hat großen Einfluss darauf, wie Schreib- und Lesezugriffe auf den Cache und den Hauptspeicher gehandhabt werden. Es geht um Fragen wie:

- Wie ist die Breite des Datenbusses im Vergleich zur Cachezeilengröße?

Je weiter die Busbreite und die Zeilengröße auseinander gehen, desto länger dauert das Fetchen einer Zeile.

- Werden Schreib- oder Lesepuffer verwendet und wie groß sind diese? Solche Puffer können auch als Caches betrachtet werden und die Interaktion mit den Cachespeicher beeinflusst die Laufzeit.

- Gibt es einen Burst-Modus und wie viele Daten werden dabei übertragen? Die Verwendung eines Burst-Modus zum schnelleren Transfer größerer Datenmengen kann das Fetchen einer Cachezeile beschleunigen.

All diese Faktoren können sich auf die Laufzeit und den Energieverbrauch auswirken, die beim Zugriff auf den Cache anfällt. So kann ein Entwickler schon im Voraus feststellen, welche Cachezeilengrößen überhaupt sinnvoll sind. Zuletzt entscheidet die Struktur des Programms und damit die Art wie der Cache genutzt wird, welche Cachezeilengröße die optimale ist. Eine Exploration wie sie in dieser Arbeit gemacht wurde, ist dafür eine gute Möglichkeit. Dabei sollte die geeignete Cachezeilengröße zusammen mit der Assoziativität ermittelt werden, denn beide Parameter sind eng miteinander verknüpft.

4.4.3 Cachegröße

Dass die Cachegröße einen großen Einfluss auf die Laufzeit eines Systems hat ist nahe liegend. Die hier ermittelten Resultate unterstreichen das. Abbildung 4.3 (Anhang 4.3) enthält alle Individuen der Exploration, diesmal differenziert nach den untersuchten Cachegrößen. Wie erwartet führen steigende Cachegrößen zu kleineren Laufzeiten und geringerem Energieverbrauch. Besonders deutlich grenzen sich die Werte für die kleinen Caches unter 4 KB voneinander ab. Die Erklärung dafür liegt in den für die Exploration verwendeten Benchmarks. Der Code der Benchmarks ist überwiegend klein und findet ab einer gewissen Cachegröße vollständig im Cache Platz. Das führt dazu, dass in diesen Fällen Cache-Misses nur beim ersten Befüllen der Cachezeilen stattfinden. Daher lassen sich oberhalb von 4 KB kaum noch Einsparungen in Laufzeit und Energie durch Erhöhung des Cachegröße erreichen. Der Energieverbrauch nimmt dort sogar leicht zu, da mit steigender Speichergröße nimmt der Energiebedarf für einen Cachezugriffe kontinuierlich zu. In Abbildung 4.3 sind die Individuen mit Cachegrößen oberhalb von 8 KByte aber gar nicht mehr auszumachen, da sie von den Punkten den nächst kleineren Cachegröße überdeckt werden.
Für die größeren Benchmarks mit Codegrößen oberhalb von 2KByte ist bei den kleinen Caches (256 und 512 Byte) die Anzahl der verfügbaren Sets so gering, dass Cache-Hits eher selten sind. Dies führt dann zu sehr hohen WCET, ACET und Energiewerten. In diesen Fällen haben die anderen Parameter einen besonders hohen Einfluss.

Wahl der Cachegröße

Das Motto „Mehr ist besser“ gilt eingeschränkt auch für die Wahl der Cachegröße. Mit keinem anderen Parameter lässt sich die Cache-Hit Rate so stark beeinflussen. Dabei ist

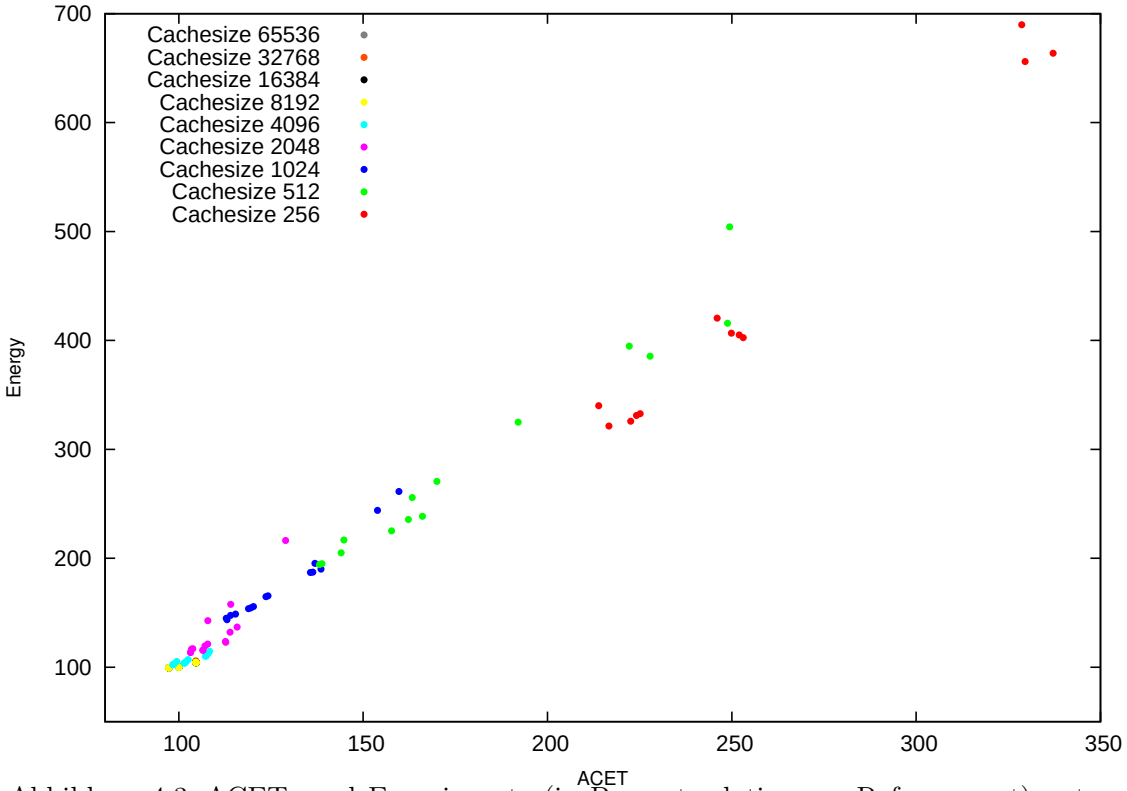


Abbildung 4.3: ACETs und Energiewerte (in Prozent relativ zum Referenzwert) unterteilt nach der Cachegröße

jedoch zu beachten, dass sich die Werte nicht proportional zur Cachegröße verbessern. Abbildung 4.4 zeigt die Entwicklung der Werte für die drei Kriterien bei der Exploration über alle Benchmarks bei steigender Cachegröße (Cachezeilengröße 16 Byte, Assoziativität 2). Es ist gut zu sehen, wie das Verbesserungspotential nimmt mit steigender Cachegröße abnimmt. Wenn die Codegröße wie bei den hier verwendeten Benchmarks nicht sehr groß ist, gibt eine Schwelle ab der sich die Vergrößerung des Caches nicht mehr rentiert.
Auf Prozessoren in eingebetteten Systeme laufen häufig nur ein oder ein paar Programme, so dass die Anforderungen an das System bekannt sind. Da die Wahl der Cachegröße auch immer eine Kostenfrage ist, sollte dem Entwickler der Verlauf, wie er in Abbildung 4.4 dargestellt ist, ungefähr bekannt sein. So kann er entscheiden in wie weit sich eine Erhöhung der Cachegröße lohnt.

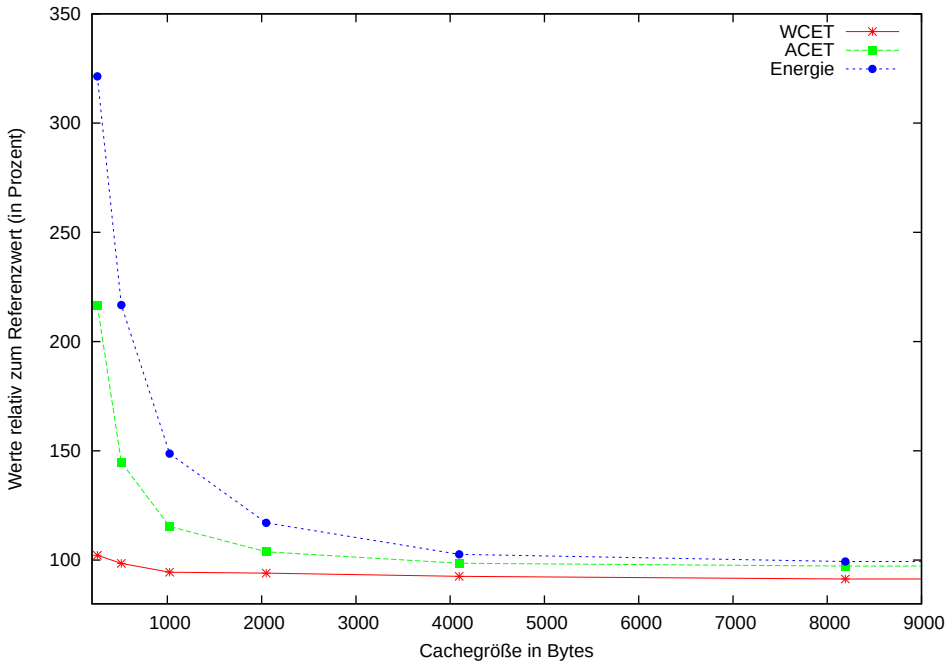


Abbildung 4.4: Entwicklung der WCET-, ACET- und Energie-Werte bei Erhöhung der Cachegröße

4.4.4 Assoziativität

Die Betrachtung der Resultate bezüglich der verschiedenen Assoziativitäten macht aufgrund von Abhängigkeiten zu den anderen Parametern nur in Bezug zu der verwendeten Cache- und Cachezeilengröße Sinn. Die undurchsichtige Darstellung in Abbildung 4.5(Anhang 4.4) macht das deutlich.

Die Auswirkungen der verschiedenen Assoziativitäten sind zum Einen abhängig von der Cachegröße. So kann man zum Beispiel bei steigender Assoziativität wachsende als auch fallende Werte für die drei Kriterien finden (siehe Tabelle 4.4, Anhang 4.7). Es ist keine eindeutige Tendenz zu erkennen. Um aussagekräftige Ergebnisse zu erhalten ist es sinnvoller einzelne Benchmarks zu untersuchen. Im folgenden werden daher Benchmarks mit unterschiedlichen Codegrößen untersucht.

Benchmark *crc*

Viele der verwendeten Benchmarks enthalten einzelne Algorithmen aus Anwendungen für eingebettete Systeme und haben daher sehr geringe Codegrößen. Einer davon ist *crc (cyclic redundancy check)* mit einer Codegröße von 520 Byte. Er führt für die zyklische Redundanzprüfung in mehreren Schleifen einfache Instruktionen aus.

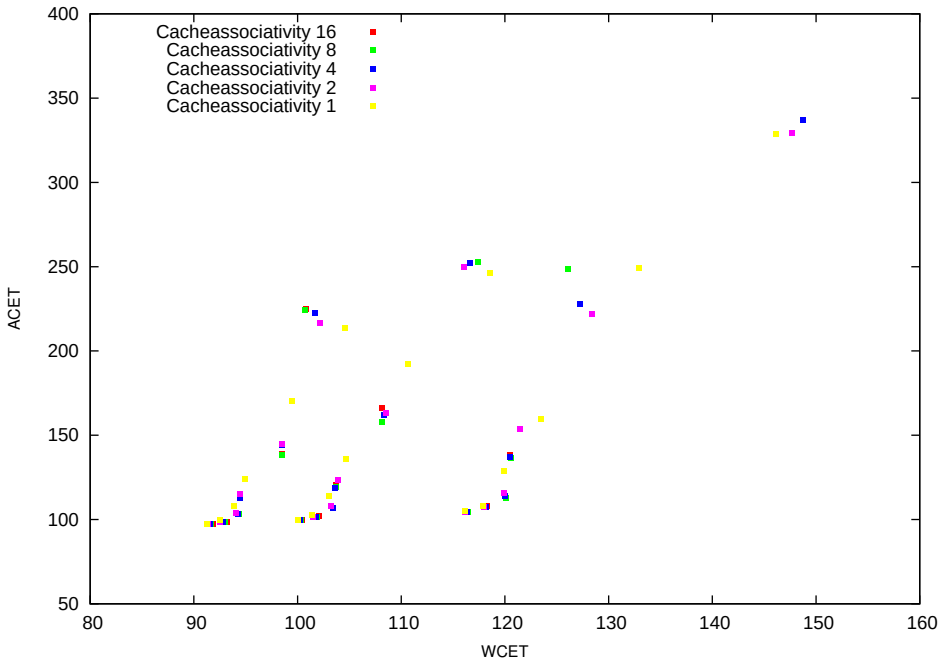


Abbildung 4.5: WCETs und ACETs (in Prozent relativ zum Referenzwert) unterteilt nach der Assoziativität

WCET	Assoziativität				
Cachegröße	1	2	4	8	16
256 Byte	104.59	102.2	101.48	100.75	100.82
512 Byte	99.43	98.47	98.53	98.54	98.54
1 KByte	94.93	94.48	94.41	94.44	94.41
2 KByte	93.84	94.06	94.25	94.3	94.31
4 KByte	92.51	92.55	92.82	92.99	93.2
8 KByte	91.29	91.33	91.54	91.64	91.79
16 KByte	91.29	91.29	91.32	91.52	91.6
32 KByte	91.29	91.29	91.29	91.32	91.5
64 KByte	91.29	91.29	91.29	91.29	91.3

Tabelle 4.4: WCETs (in Prozent relativ zum Referenzwert) für verschiedene Cachegrößen und Assoziativitäten bei einen Cachezeilengröße von 16

Für alle drei Kriterien gilt, dass eine Variation der Cache-Assoziativität zu kaum nennenswerten Veränderungen in den Werten führt. Tabelle 4.5 zeigt die Daten für die Cachegrößen 256, 512 und 1024 Byte bei einer Cachezeilengröße von 16 Byte. Unterhalb

WCET	Assoziativität				
Cachegröße	1	2	4	8	16
256 Byte	100.32	100.35	100.32	100.35	100.35
512 Byte	99.93	99.96	100	99.91	99.91
1 KByte	99.91	99.91	99.91	99.91	99.91
ACET	Assoziativität				
Cachegröße	1	2	4	8	16
256 Byte	100.47	100.52	100.71	100.71	100.71
512 Byte	99.82	99.82	100.1	99.81	99.81
1 KByte	99.82	99.8	99.8	99.8	99.8
Energie	Assoziativität				
Cachegröße	1	2	4	8	16
256 Byte	100.79	100.90	101.31	101.33	101.39
512 Byte	99.49	99.49	99.6	99.51	99.58
1 KByte	99.49	99.49	99.49	99.52	99.58

Tabelle 4.5: Benchmark *crc*: WCETs, ACETs und Energiewerte (in Prozent relativ zum Referenzwert) für verschiedene Cachegrößen und Assoziativitäten bei einen Cachezeilengröße von 16 Byte

von 1 KByte nehmen die Werte mit steigender Assoziativität leicht zu. Die Daten zu Codegrößen ab 1 KByte sind nicht mehr in der Tabelle aufgeführt, da nur noch minimale Steigerungen bei den Energiewerten zu erkennen sind, WCETs und ACETs bleiben unverändert.
Ausschlaggebend für diese Werte ist die sehr geringe Codegröße dieses Benchmarks. Bereits im zweit-kleinsten Cache (512 Byte) finden die Instruktionen des Benchmarks nahezu vollständig Platz. Auch beim 256 Byte großen Cache passt dass der laufzeitrelevante Schleifencode in den Cache. In diesen Fällen spielt die Assoziativität keine große Rolle mehr und kann die Anzahl der Cache-Misses nicht spürbar reduzieren. Beim Energiebedarf ist sogar eine leichte Steigerung zu erkennen, wenn die Assoziativität erhöht wird.

Benchmark *adpcm_g721_verify*

Adpcm_g721_verify enthält zahlreiche Funktionen, Schleifen und Berechnungen auf Arrays. Mit 5332 Byte ist er deutlich größer als *crc* und bei der Ausführung ist mit deutlich mehr Cache-Misses zu rechnen.

Bei diesem Benchmark bewirkt die Variation der Assoziativitäten deutlich größere Schwankungen in den gemessenen Werten. Tabelle 4.6 zeigt die WCETs bei einer Cachezeilengröße von 16 Byte. Je nach Cachegröße sind jeweils niedrige oder höhere Assoziativitäten besser. Während bei einen 1 KByte großen Cache die 16-fache Assoziativität zu empfehlen ist, schneidet bei einem Cache von 2 KByte die Assoziativität 1 am besten

WCET	Assoziativität				
Cachegröße	1	2	4	8	16
256 Byte	240.19	248.1	263.57	263.56	264.3
512 Byte	206.78	205.1	204.86	204.86	204.86
1 KByte	205.11	204.39	203.47	203.93	202.32
2 KByte	186.64	193.76	199.74	200.91	200.68
4 KByte	147.72	149.03	157.45	162.76	167.07
8 KByte	108.53	109.9	116.33	118.96	122.35
16 KByte	108.53	108.53	109.44	115.18	116.9
32 KByte	108.53	108.53	108.53	109.44	114.26
64 KByte	108.53	108.53	108.53	108.53	108.53

Tabelle 4.6: Benchmark *adpcm_g721_verify*: WCETs (in Prozent relativ zum Referenzwert) für verschiedene Cachegrößen und Assoziativitäten bei einen Cachezeilengröße von 16 Byte

Energie	Assoziativität				
Cachezeilengröße	1	2	4	8	16
16 Byte	231.25	188.99	190.72	188.24	207.79
32 Byte	248.13	209.11	210.65	215.92	261.67
64 Byte	279.05	263.31	263.05	281.64	324.07

Tabelle 4.7: Energie-Werte für verschiedene Cachezeilengrößen und Assoziativitäten bei einen Cachegröße von 4096 Byte

ab. Das gleiche gilt für die anderen Kriterien (siehe Anhang 4.8). Neben der Cachegröße ist die Assoziativität auch von der Cachezeilengröße abhängig. Tabelle 4.7 (Anhang 4.9) enthält die ermittelten WCETs für einen 4 KByte großen Cache. Die Zahlen machen deutlich, dass die optimale Assoziativität je nach Cachezeilengröße variiert und bei 2, 4 oder 8 liegen kann. Die ermittelten ACETs führen zu den gleichen Schlussfolgerungen. Lediglich bei den WCETs sind für diese Cachegröße niedrige Assoziationen vorteilhaft.
Adpcm_g721_verify ist einer von mehreren untersuchten Benchmarks, bei denen die Resultate unterschiedlich stark von der Größe des Caches und der Cachezeilen abhängt. Bei keinem war eine pauschale Aussage möglich, ob eher eine hohe oder niedrige Assoziativität von Vorteil ist.

4.4.5 Wahl der Assoziativität

Über die optimale Assoziativität des verwendeten Caches sollte man sich schon Gedanken machen, bevor die Cachezeilengröße festgelegt ist. Beide Parameter organisieren den verfügbaren Cache und sollten auch gemeinsam betrachtet werden. Ist die Codegröße so gering, dass der Hauptteil des Programmes in den Cache passt, spielt die Assoziativität keine Rolle mehr. Aber in diesem Fall kann der Cache auch durch einen SRAM ersetzt

werden.
Eine generelle Aussage ob eher Direct-Mapped Caches oder hohe Assoziativitäten bessere Ergebnisse erzielen ist nicht möglich. Zu sehr hängen die Auswirkungen vom System und dem verwendeten Code ab. Ist einem Entwickler beides genau bekannt, so kann er die optimale Konfiguration durch Simulation und Analyse ermitteln.

4.5 Schlussfolgerungen

Die Resultate der Exploration haben gezeigt, dass für eingebettete Systeme mit ihren speziellen Anforderungen eine differenzierte Auswahl der Cacheparameter notwendig ist. In der Literatur findet man häufig eine Heuristik um die beste Cachekonfiguration zu erreichen (z.B. [ZV03]). Demnach wählt man zuerst eine typische Konfiguration und optimiert nacheinander die Cachegröße, Cachezeilengröße und die Assoziativität. Die vorliegenden Resultate bestätigen zumindest, dass die Cachegröße der hervorstechende Parameter ist, die daher bei der Konfiguration zuerst festgelegt werden sollte. Die Auswertung Resultate lassen aber erkennen, dass für die anderen beiden Parameter keine feste Reihenfolge bzw. Heuristik zu empfehlen ist. Man kann zwar sagen, dass für eingebettete Echtzeitsysteme, bei denen oberer Laufzeitgrenzen entscheidend sind, die Cachezeilengröße von großer Bedeutung ist. So lassen sich, selbst bei optimaler Wahl der Cachegröße, große Differenzen in den WCET-Werten feststellen, wenn die Cachezeilengröße variiert wird. Allerdings sind diese Werte auch immer von der Assoziativität abhängig.
Nichts desto trotz ist haben sich bei der Exploration über alle Benchmarks einige Parameterkombinationen als „pareto-optimale“ erwiesen. Nicht unbedingt für jeden einzelnen Benchmarks, aber dennoch für die Verwendung der gesamten Benchmark-Kollektion.
Pareto-optimale Konfigurationen:

1. Cachegröße 8 KByte, Zeilengröße 16 Byte, Assoziativität 2
   WCET: 91.33% ACET: 97.26% Energie: 99.37%

2. Cachegröße 16 KByte, Zeilengröße 16 Byte, Assoziativität 2
   WCET: 91.29% ACET: 97.26% Energie: 99.39%

3. Cachegröße 16 KByte, Zeilengröße 16 Byte, Assoziativität 4
   WCET: 91.32% ACET: 97.26% Energie: 99.38%

Diese drei Konfiguration führen zu fast identischen Werten mit nur leichten Abweichungen in WCET und Energie. Hervorzuheben ist aber die Konfiguration 1 mit 8 KByte, da sie im Vergleich zu den anderen nur die halbe Cachegröße benötigt.
Zuletzt sollten diese Werte noch mit der Referenz verglichen werden. Das in dieser Arbeit untersuchte System, ein LPC2880 Mikrocontroller mit ARM7TDMI-Prozessor, besitzt in der Original-Konfiguration einen 16 KByte großen Cache mit einer Zeilengröße von 32 Byte und einer Assoziativität von 2. Unsere optimale Konfiguration besitzt also einen kleineren Cache und die Zeilengröße ist ebenfalls kleiner. In diesem Fall ist die Reduzierung der Cachezeilengröße für die besseren Werte verantwortlich. Für die Art von

Software, die die verwendeten Benchmark darstellen, war der Cache des Referenzsystems also unnötig groß und durch eine Reduzierung der Cachezeilengröße konnte eine Verbesserung in den Kriterien erreicht werden.

5 Zusammenfassung und Ausblick

Mit Hilfe multikriterieller Explorationen wurden in dieser Arbeit die Auswirkungen von Compileroptimierungen und Cacheparametern auf drei Kriterien untersucht: Neben der höchstmöglichen und der durchschnittlichen Laufzeit war der Energieverbrauch von ausführbarem Code von Interesse. Um die nötigen Daten zu gewinnen, wurden die Explorationen auf zahlreiche Benchmarks für die ARM7TDMI Plattform angewendet. Nach Auswertung der Resultate können die zu Beginn der Arbeit gestellten Fragen beantwortet werden:

- Verändern sich ACET und WCET synchron zueinander?
  Wie schon für den Tricore gezeigt wurde [LPF+10], verhalten sich ACET und WCET auch für den ARM7TDMI bei Berücksichtigung aller Benchmarks gleichermaßen, wobei erhebliche Verbesserungen erzielt wurden. Bei Betrachtung einzelner Benchmarks kann, je nachdem welche Optimierungen besonders wirksam sind, eines der Kriterien stärker beeinflusst werden. In der Summe verändern sich ACET und WCET jedoch nahezu synchron.

- Bewirkt eine Verringerung der einzelnen Laufzeiten auch einen sinkenden Energieverbrauch?
  Der Energieverbrauch verhält sich bei der Variation der Compileroptimierungen im Durchschnitt über alle Benchmarks ebenfalls synchron zur Laufzeit. Mit sinkender Laufzeit nimmt auch der Energieverbrauch ab. Es konnte eine Verbesserung um 27,23% erreicht werden.

Diese Resultate zeigen, dass der Einsatz von typischen ACET orientierten Compileroptimierungen auch für eingebettete Systeme sinnvoll ist, bei denen die WCET und der Energieverbrauch von Bedeutung sind. Solche Optimierungen wirken sich auch auf die anderen Kriterien positiv aus.
Die Bestimmung der für ein System am besten geeigneten Compileroptimierungssequenz, wird erst durch den Einsatz von genetischen Algorithmen aufgrund des komplexen Suchraums möglich. Die hier gefundene pareto-optimale Lösung ist für das Prüfset ist um 7,86%(WCET) und 5,62%(Energie) besser als die Optimierungsstufe O3 des WCC. Die ACETs sind gering höher (0,39%).
Bei der Analyse einzelner Optimierungen haben sich drei Mengen von Optimierungen ergeben, die sich in ihren Auswirkungen auf die betrachteten Kriterien unterscheiden. Die erste Menge beinhaltet Optimierungen, die einen durchweg positiven Einfluss auf alle Kriterien haben und daher unbedingt Teil der Optimierungsseqeuenz sein sollten. In der zweiten Menge sind Optimierungen versammelt, die keinen eindeutig positiven oder negativen Einfluss haben. Optimierungen, die einen deutlich negativen Einfluss auf die

drei Kriterien ACET, WCET und Energie haben, wurden in einer dritten Menge zusammengefasst. Nach einer Analyse kann vom Einsatz eines Teils dieser Optimierungen abgeraten werden.
Gesondert betrachtet wurden die Compileroptimierungen *Instruction Scheduling* und *Value Propagation*. Das *Instruction Scheduling* des arm-elf-gcc produziert für einen Großteil der Benchmarks ineffektiven Code und sollte daher nicht für die Optimierung verwendet werden. Der Einsatz von *Value Propagation* ist trotz der sehr schlechten Resultate in einem Benchmark zu empfehlen. Jedoch sollte bei Programmcode, der viele hochwertige Konstanten enthält, besonders auf die Auswirkungen von *Value Propagation* geachtet werden.

Im zweiten Teil der Arbeit wurden Cacheparameter exploriert. Ausgehend vom Referenzsystem (LPC 2880 Mikrokontroller) wurden zahlreiche Cachegrößen, Cachezeilengrößen und Assoziativitäten evaluiert und auf die Auswirkungen auf WCET, ACET und den Energieverbrauch untersucht. Die Resultate wurden im Hinblick auf die verbreitete Heuristik ausgewertet, nach der die Parameter in der Reihenfolge: 1.Cachegröße 2. Cachezeilengröße und 3. Assoziativität gewählt werden können. Die Ergebnisse zeigen, dass für eingebettete Systeme die Cachezeilengröße und die Assoziativität nicht getrennt voneinander festgelegt werden sollten. Der Einfluss auf des einen Parameters ist immer stark vom anderen abhängig.
Die bei der Exploration gefundenen pareto-optimalen Parameterkombinationen erzielten leichte Verbesserungen in den drei Kriterien im Vergleich zum Referenzsystem (WCET: 91.33% ACET: 97.26% Energie: 99.37%). Eine Konfiguration erreichte diese Verbesserung sogar mit der halben Cachegröße.

Ausblick
Ausgehend von den Ergebnissen dieser Arbeit, sind viele interessante Fragestellungen für zukünftige Arbeiten zu erkennen:

- Weitere Explorationen für Compileroptimierungen
  Die hier durchgeführten Explorationen haben bereits sehr gute Lösungen zustande gebracht. Es ist aber denkbar, mit aufwändigeren Explorationen zu noch besseren Ergebnissen zu kommen. So kann die Anzahl der Generationen und die in einer Generation untersuchten Individuen weiter erhöht werden. Zudem können andere Selektoren und unterschiedliche Parameter für die Variation erprobt werden.

- Reihenfolge der Compileroptimierungen
  Es hat sich gezeigt, dass die Reihenfolge der angewendeten Optimierungen großen Einfluss auf die Kriterien hat. Eine aufwändige aber dennoch interessante Aufgabe ist es, die besonders guten Lösungen auf deren Optimierungsreihenfolge zu untersuchen und mit der im WCC verwendeten zu vergleichen.

- Codegröße
  Die drei untersuchten Kriterien verhalten sich sehr ähnlich. Es bietet sich an, eine

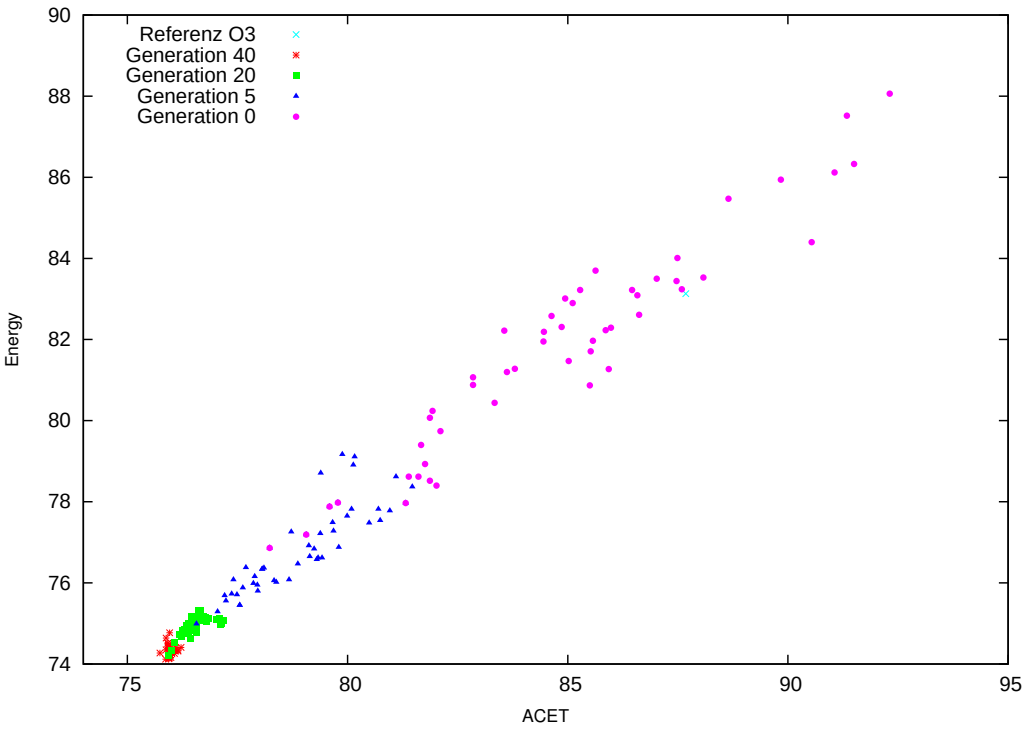
neue Exploration durchzuführen, die die Codegröße als vierte Dimension hinzunimmt.

- Cacheparameter und Prozessorarchitektur
  Es ist vorstellbar, bei der Suche nach den optimalen Cacheparametern weitere Faktoren mit einzubeziehen. Da wären Ersetztungsstrategien für die Cachezeilen sowie Parameter für Schreib- und Lesepuffer und den Burst-Modus beim Zugriff auf den Hauptspeicher. Die Variation dieser Parameter könnte zu weiteren Verbesserungen führen.

6 Anhang

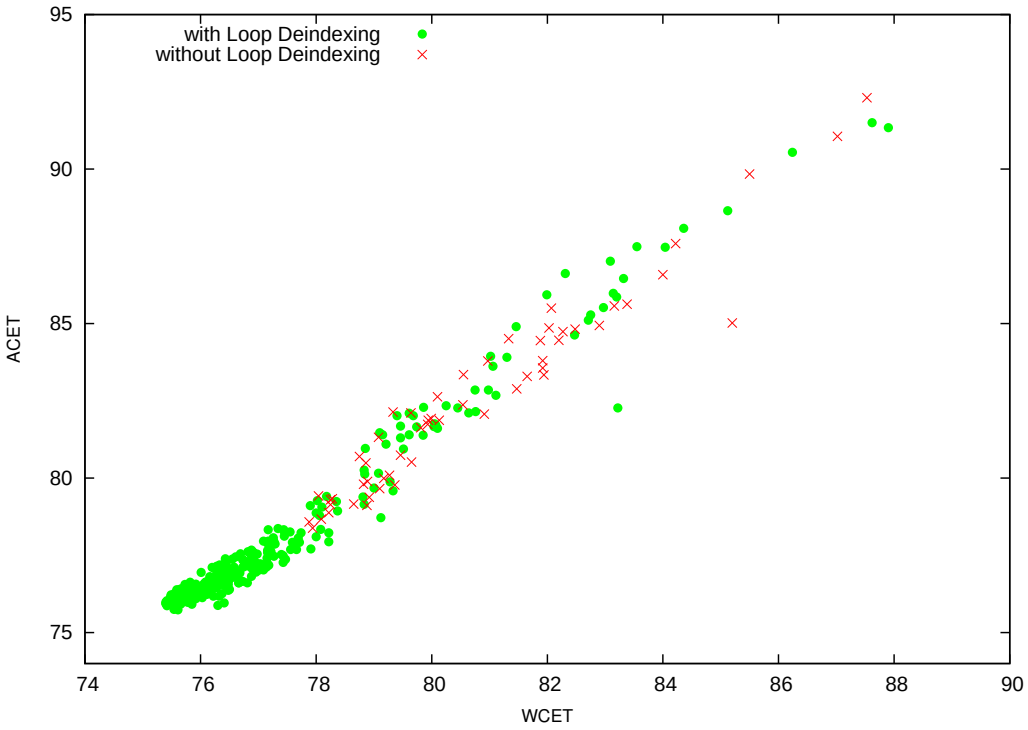
Anhang 3.1

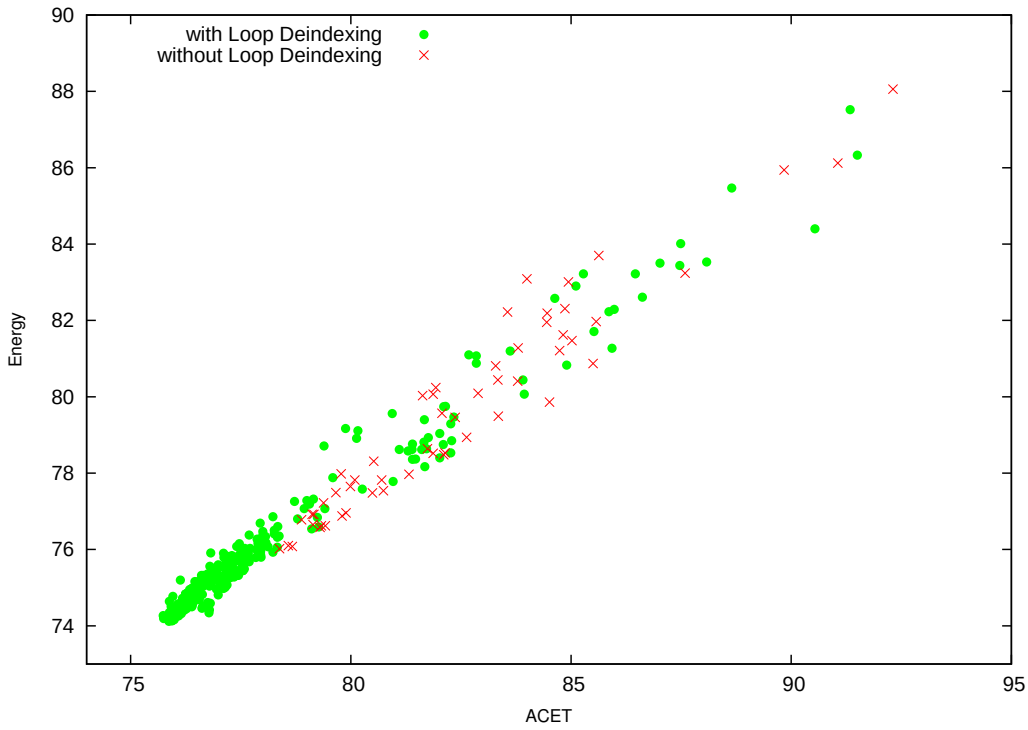
Alle Benchmarks, 50 Generationen, 50 Individuen pro Generation, Abbildung der Generationen 0, 5, 10 und 40



Anhang 3.2

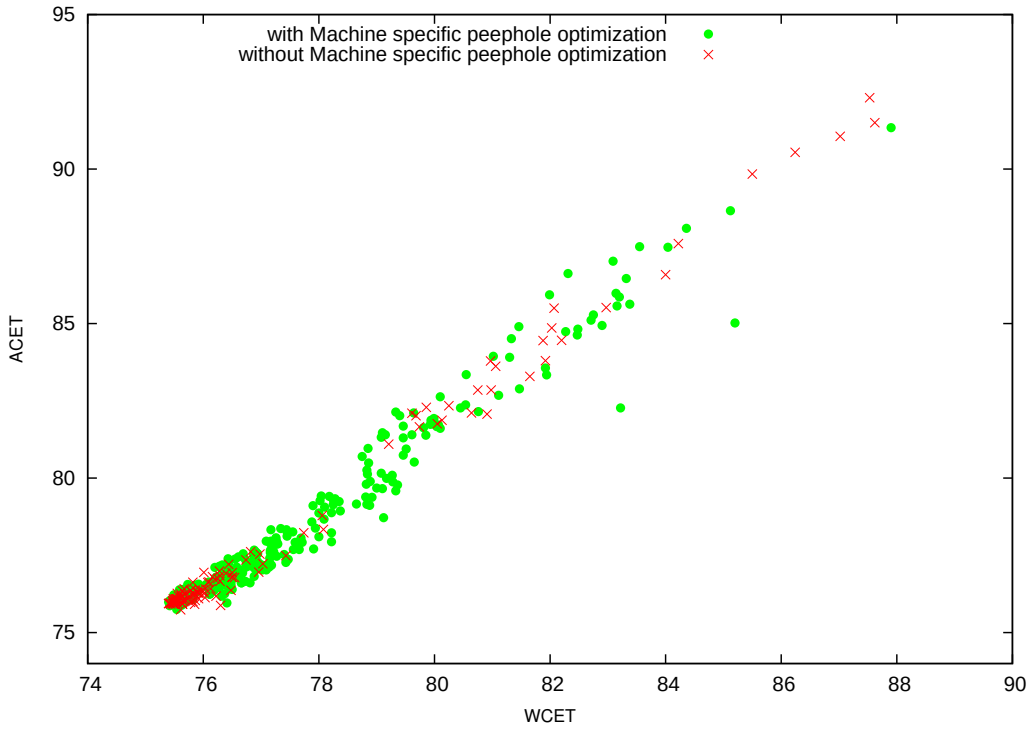
Alle Benchmarks, 50 Generationen, 50 Individuen pro Generation, Überblick über die Verwendung von *Loop Deindexing*

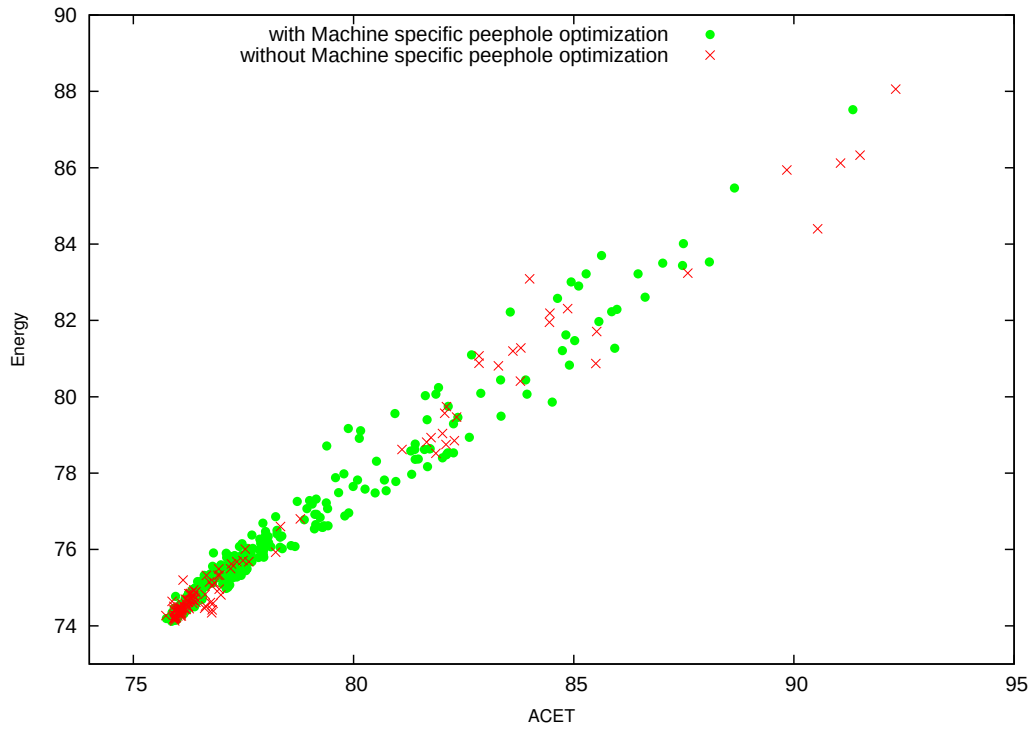




Anhang 3.3

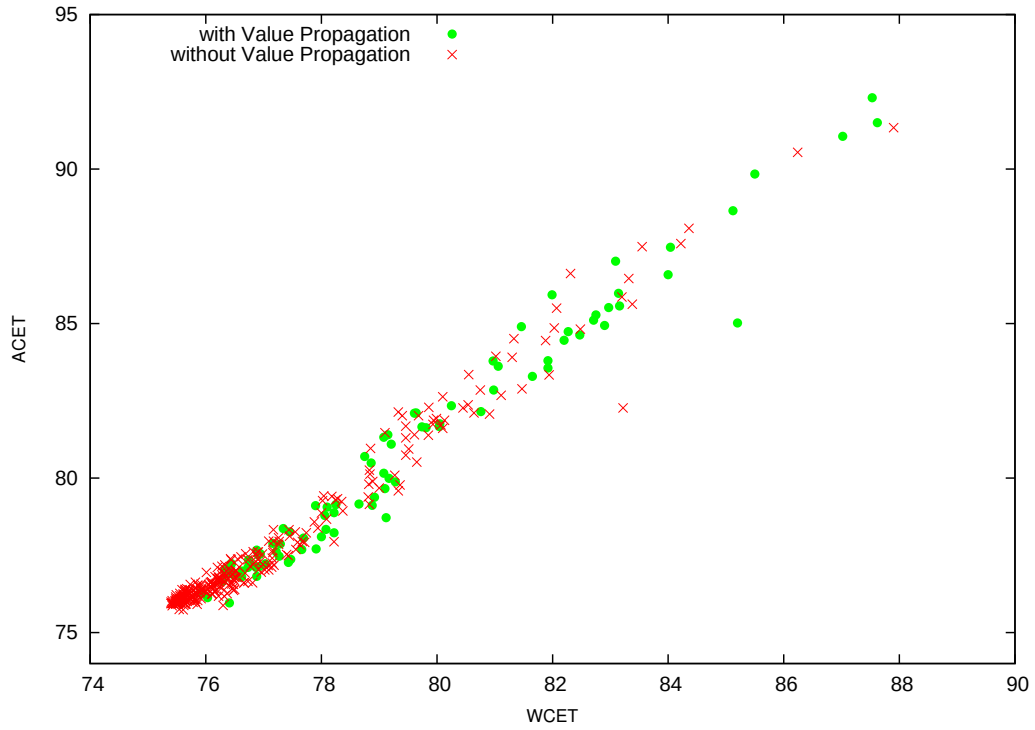
Alle Benchmarks, 50 Generationen, 50 Individuen pro Generation, Überblick über die Verwendung von *Machine specific peephole optimization*

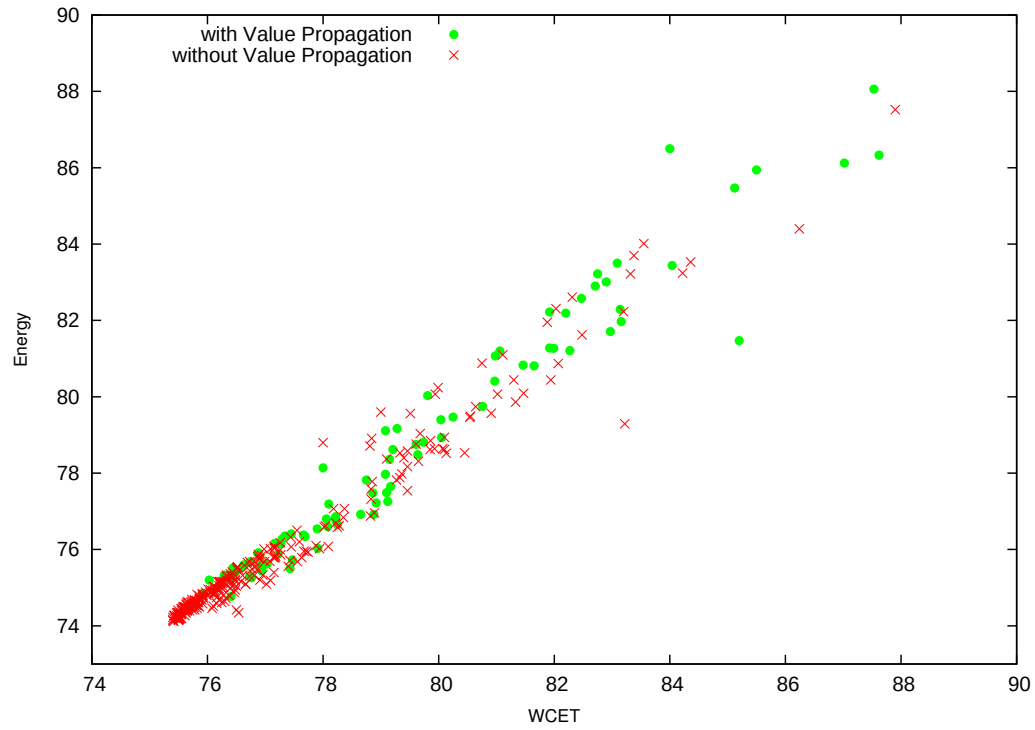




Anhang 3.4

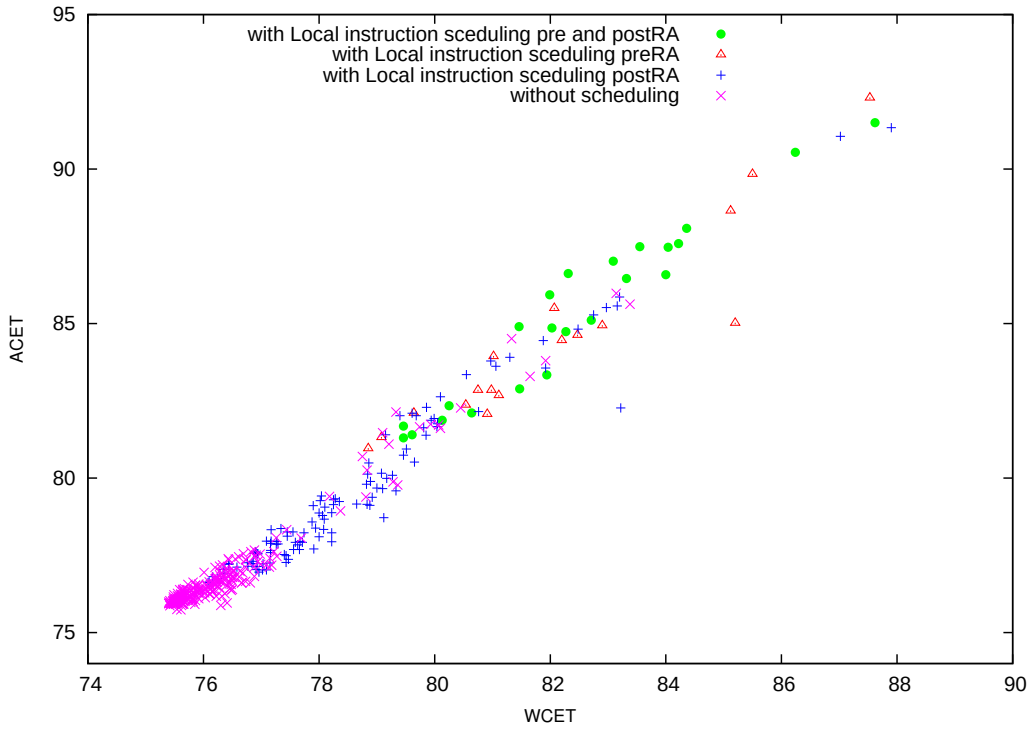
Alle Benchmarks, 50 Generationen, 50 Individuen pro Generation, Überblick über die Verwendung von *Value Propagation*



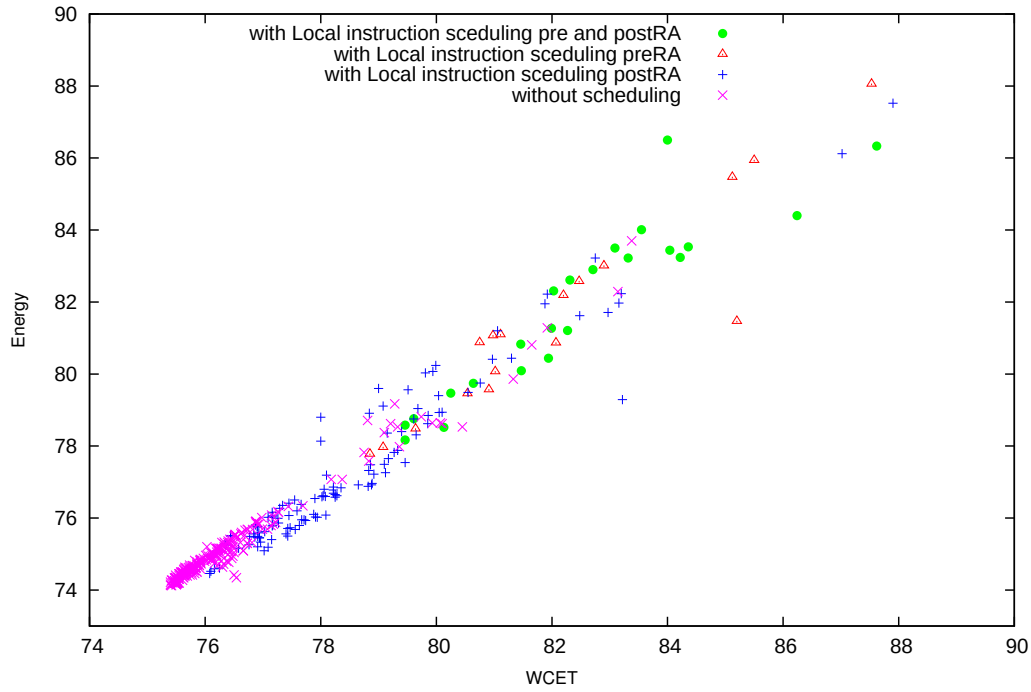


Anhang 3.5

Alle Benchmarks, 50 Generationen, 50 Individuen pro Generation, Überblick über die Verwendung von *Instruction Scheduling*

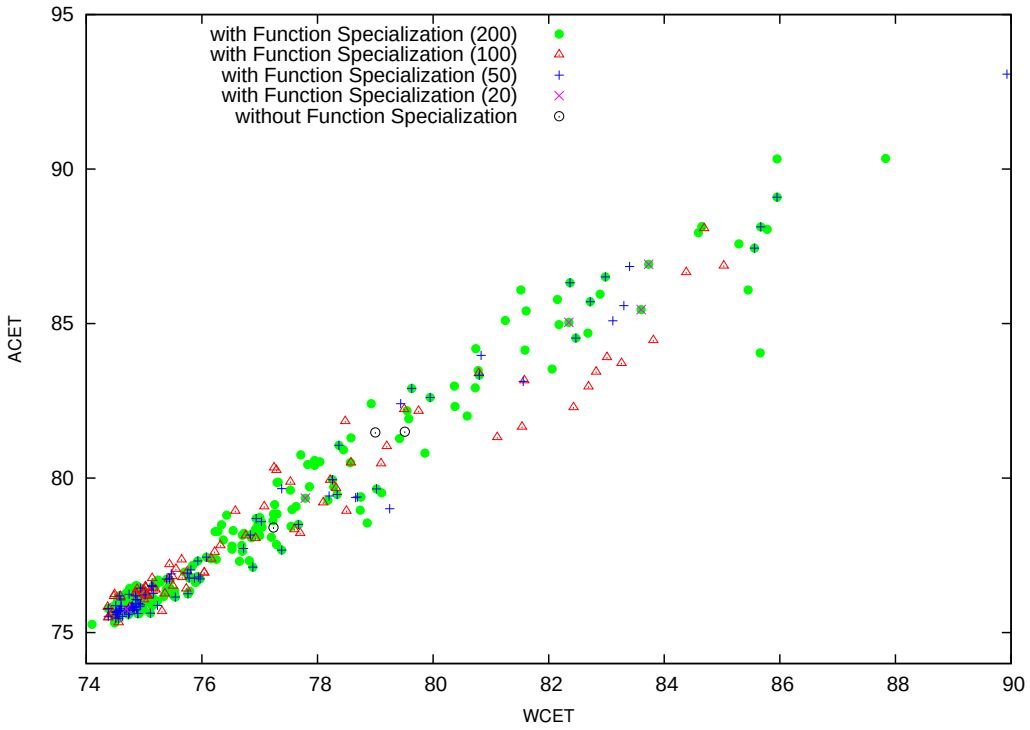


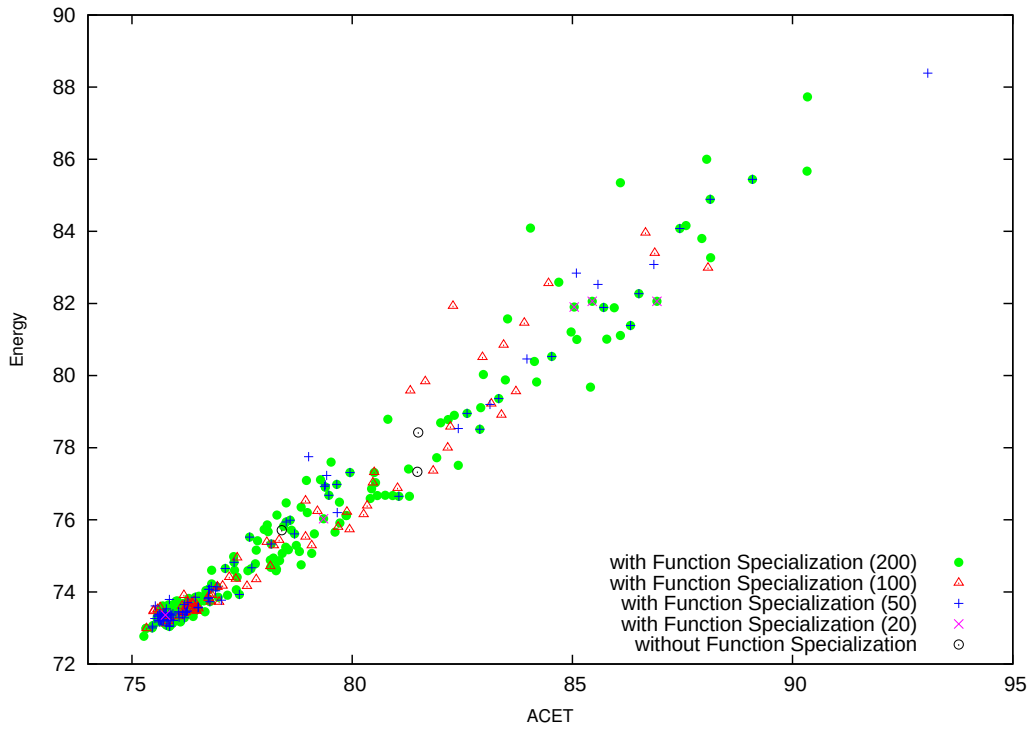
Probe Benchmark



Anhang 3.6

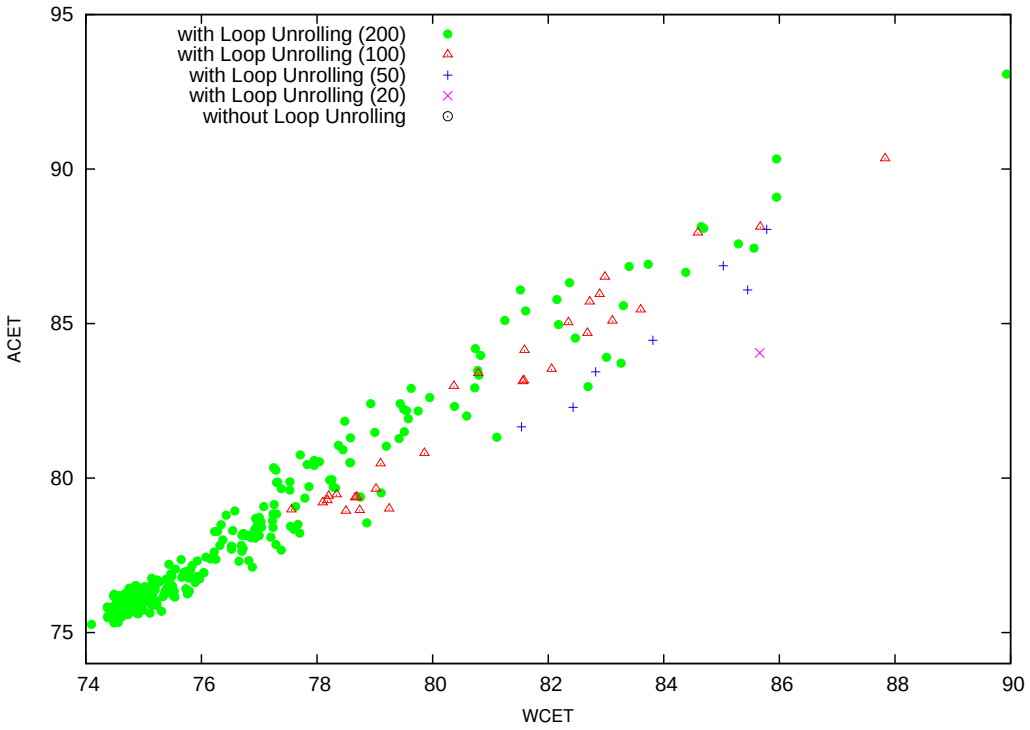
Alle Benchmarks, 50 Generationen, 50 Individuen pro Generation, Überblick über die Verwendung von *Function Specialization*

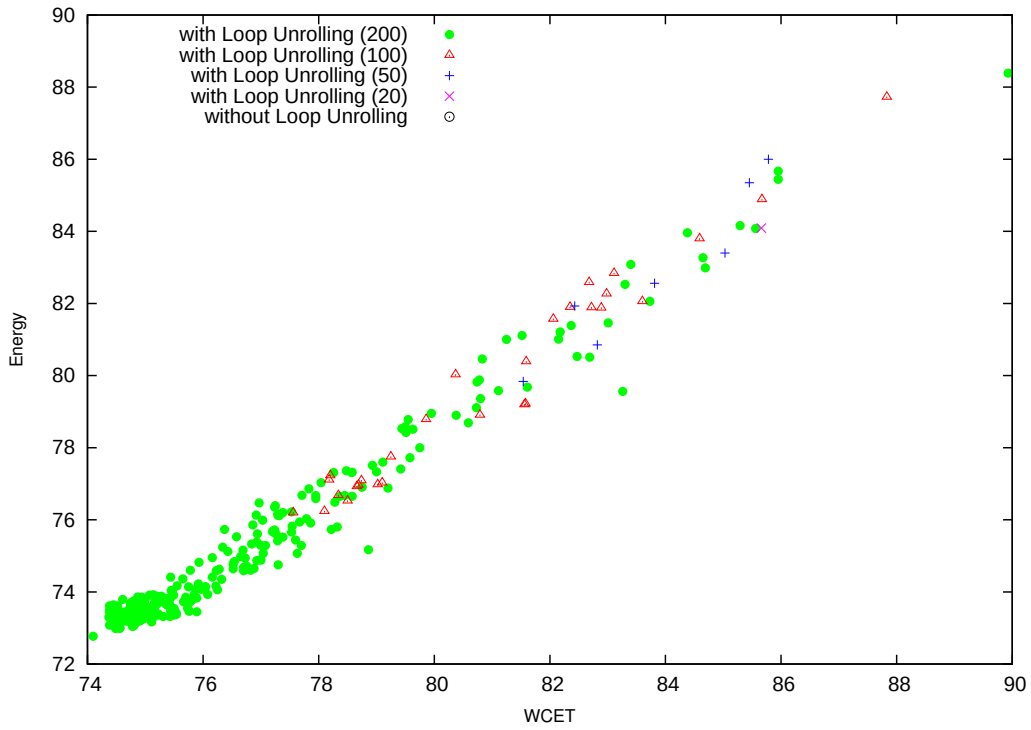




Anhang 3.7

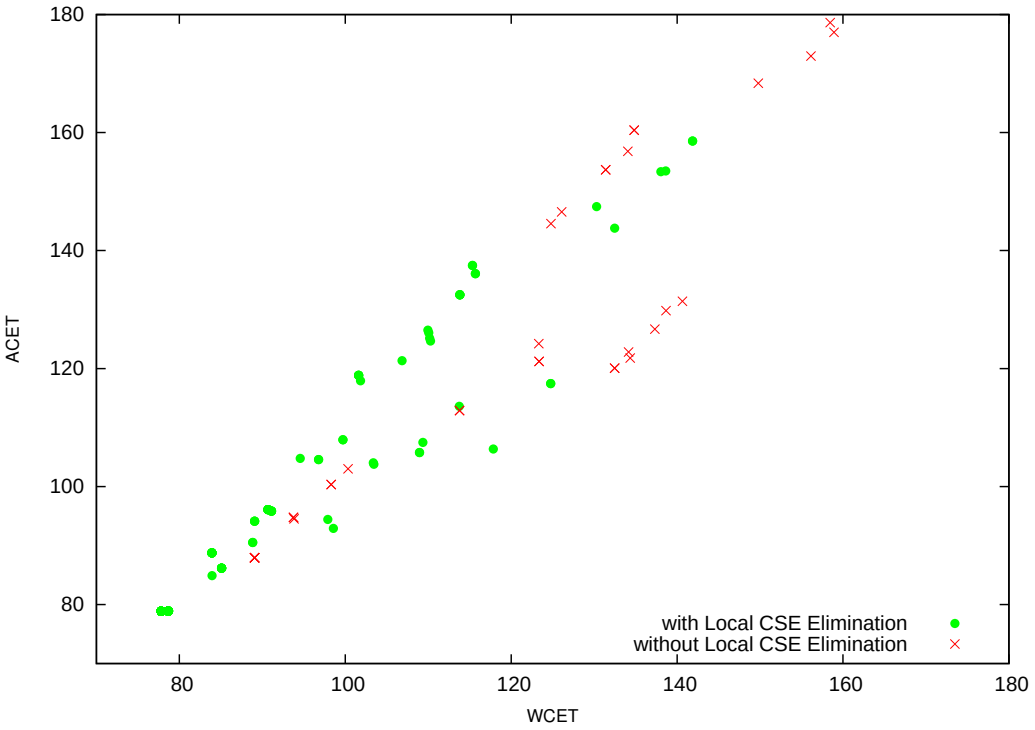
Alle Benchmarks, 50 Generationen, 50 Individuen pro Generation, Überblick über die Verwendung von *Loop Unrolling*

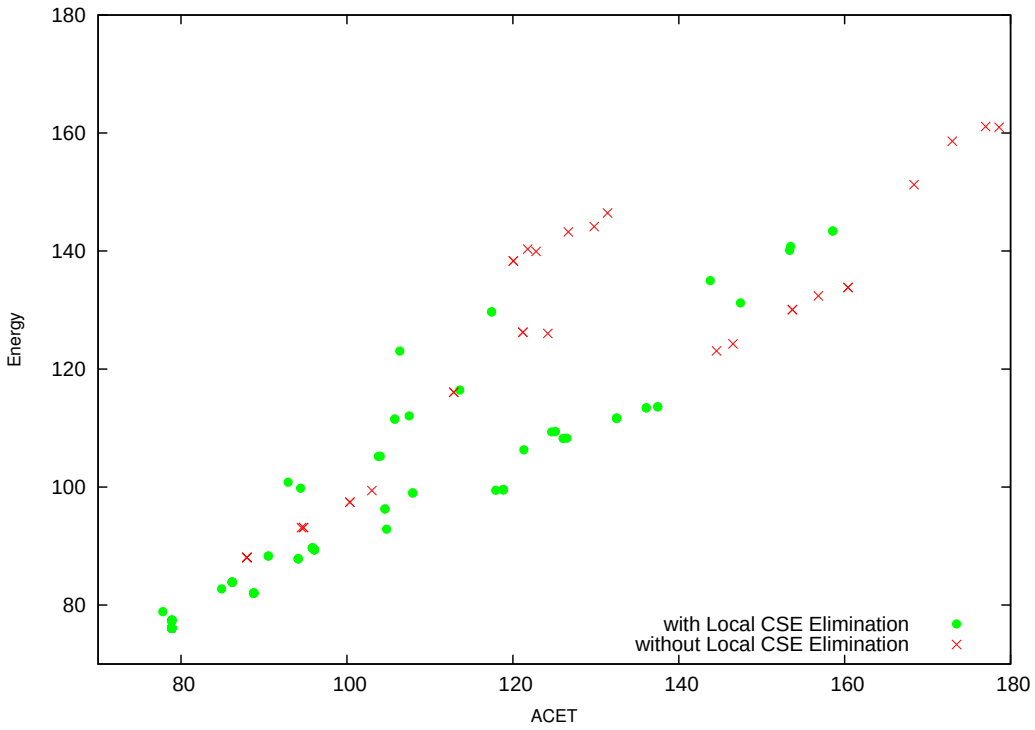




Anhang 3.8

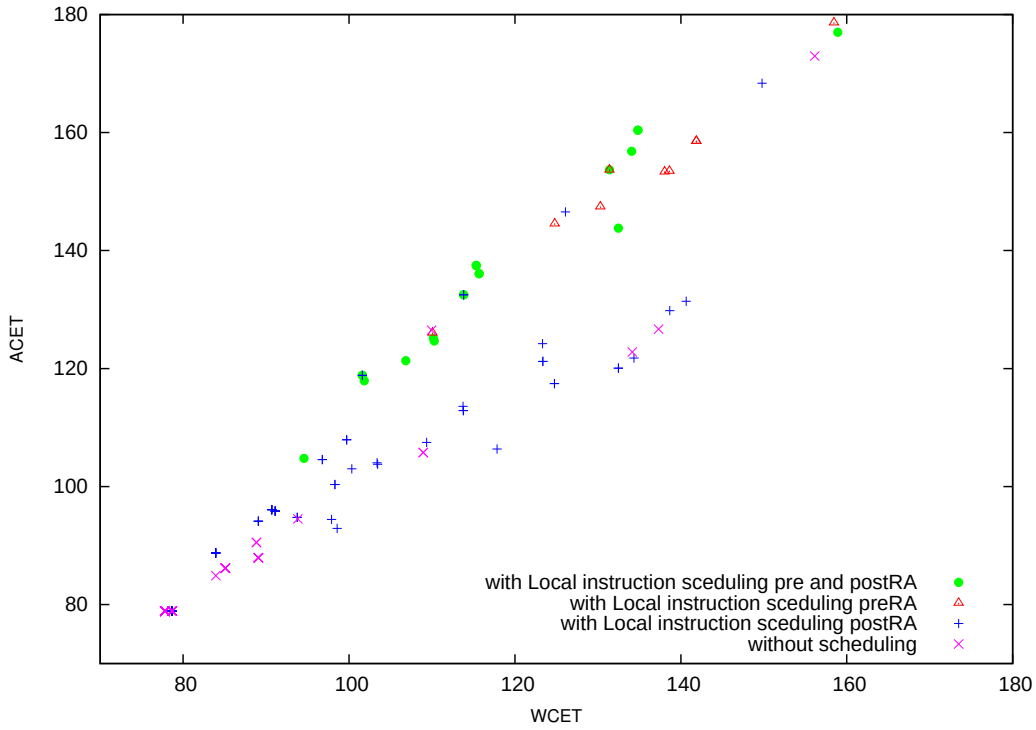
Benchmark *fdct*, 50 Generationen, 50 Individuen pro Generation, Überblick über die Verwendung von *Local CSE Elimination*

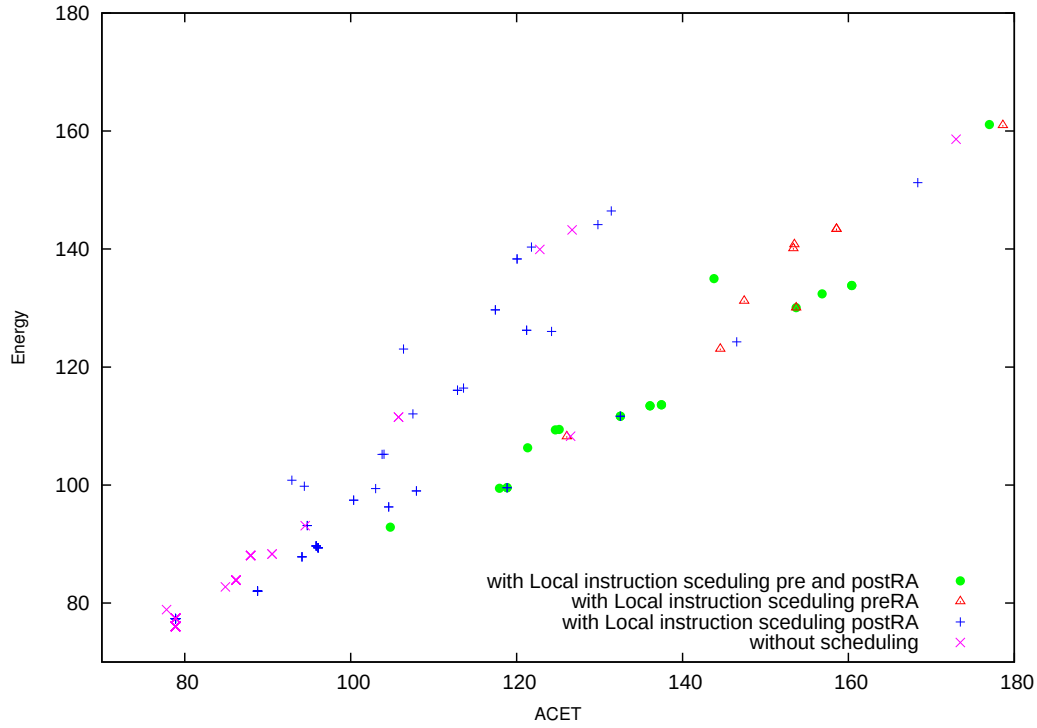




Anhang 3.9

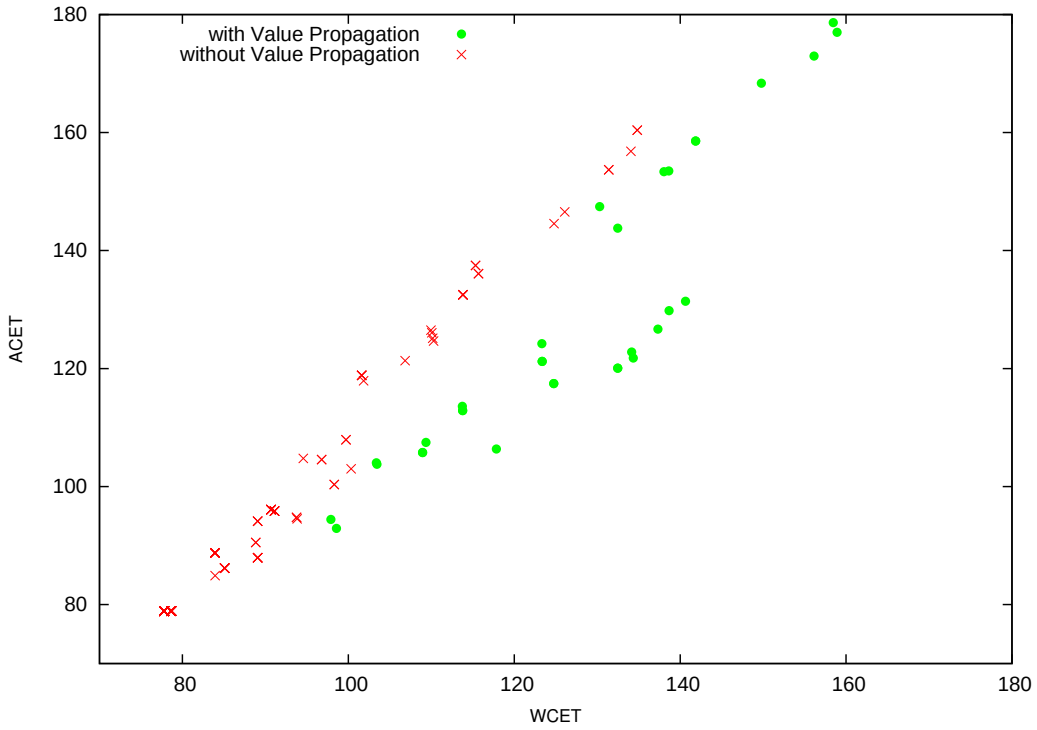
Benchmark *fdct*, 50 Generationen, 50 Individuen pro Generation, Überblick über die Verwendung von *Instruvtion Scheduling*

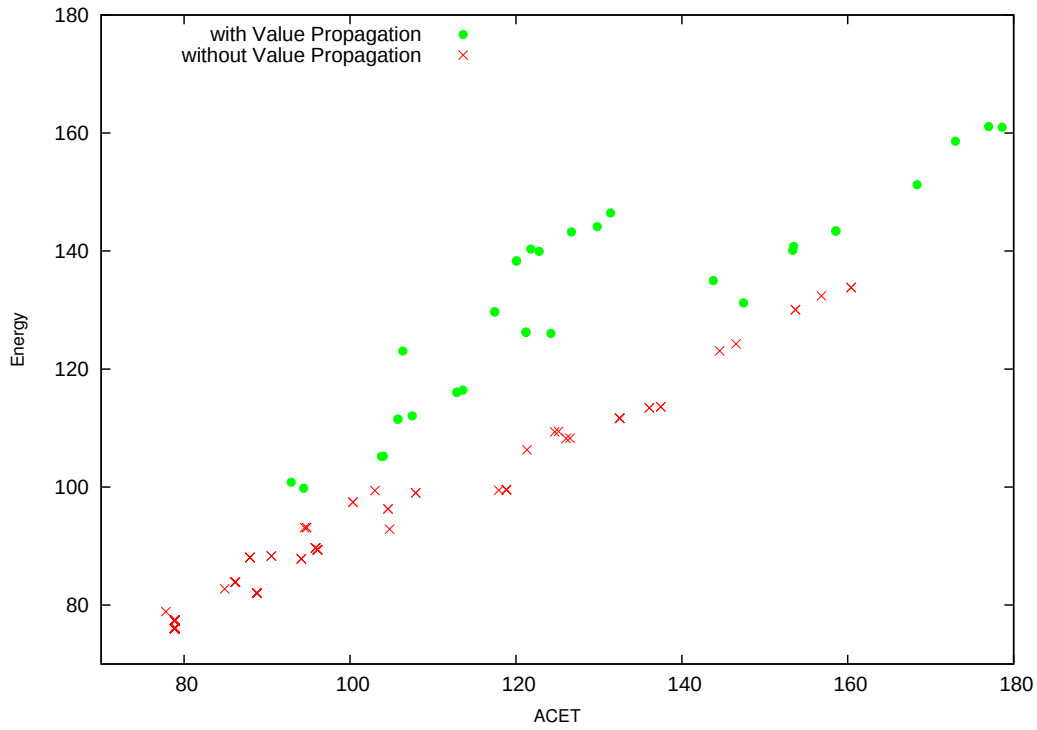




Anhang 3.10

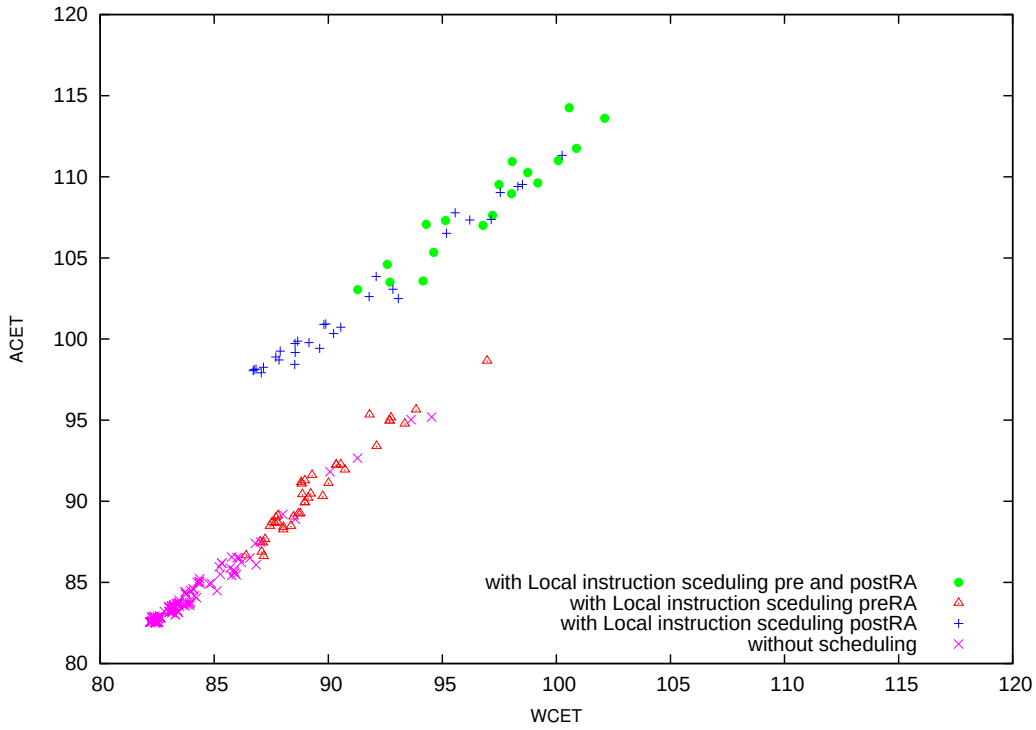
Benchmark *fdct*, 50 Generationen, 50 Individuen pro Generation, Überblick über die Verwendung von *Value Propagation*

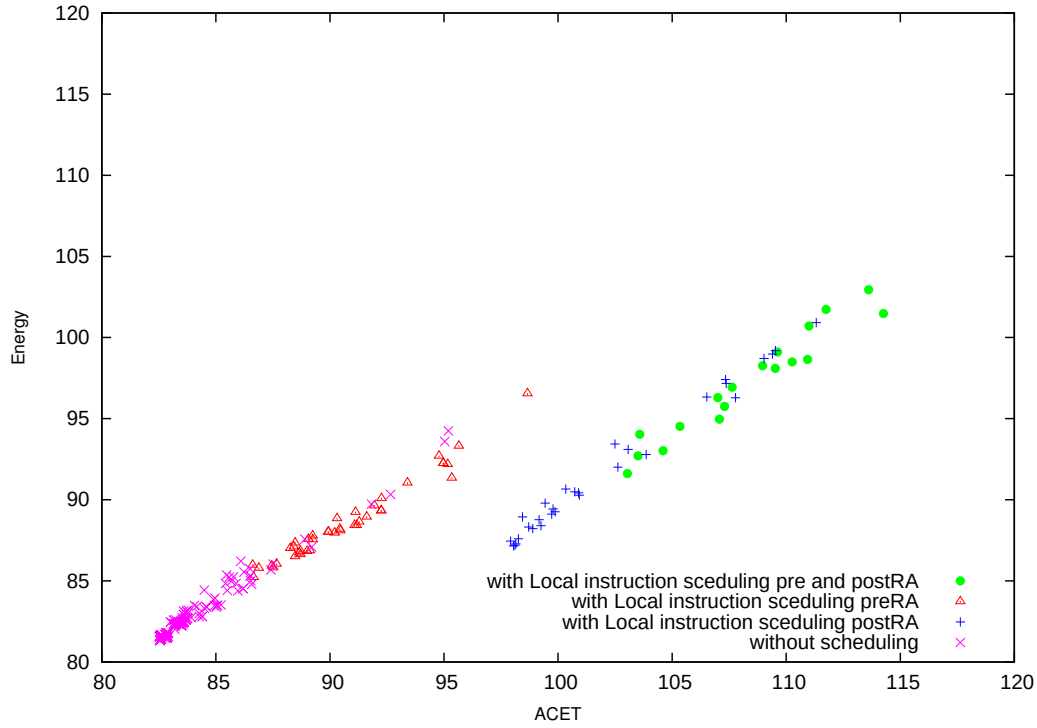




Anhang 3.11

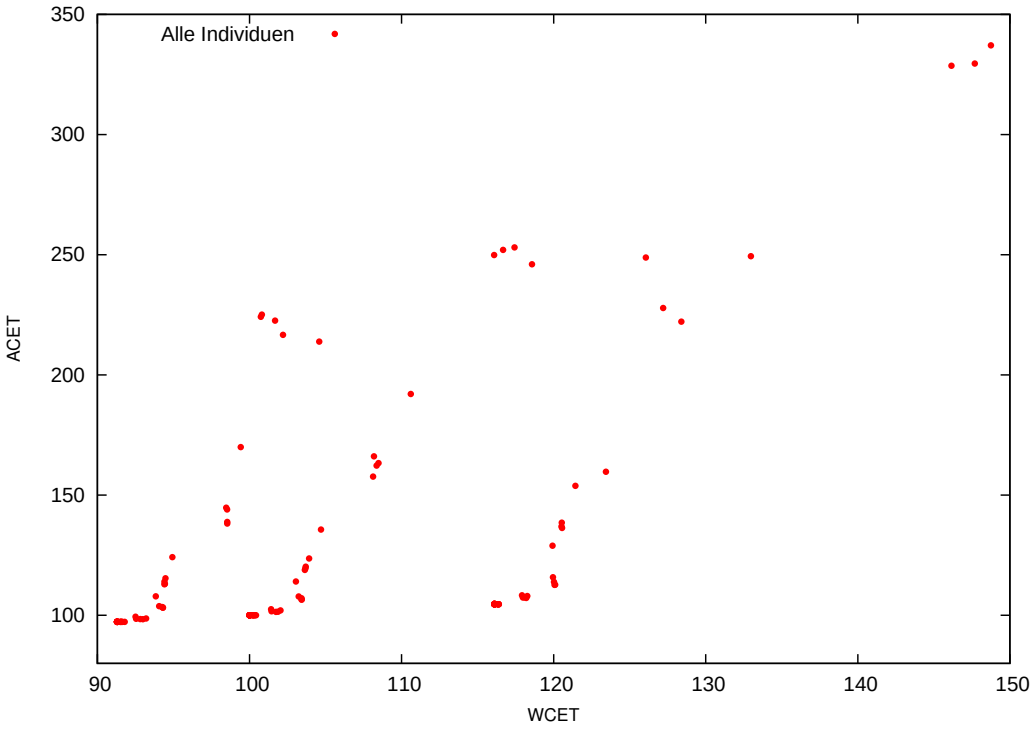
Benchmark *ndes*, 50 Generationen, 50 Individuen pro Generation, Überblick über die Verwendung von *Instruction Scheduling*

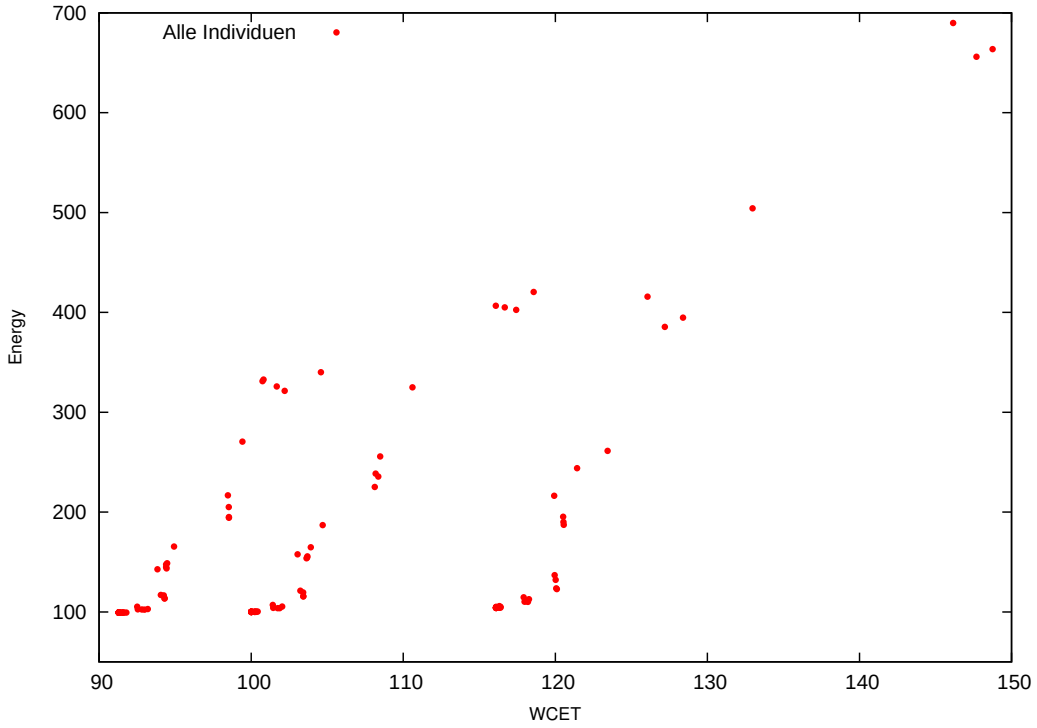




Anhang 4.1

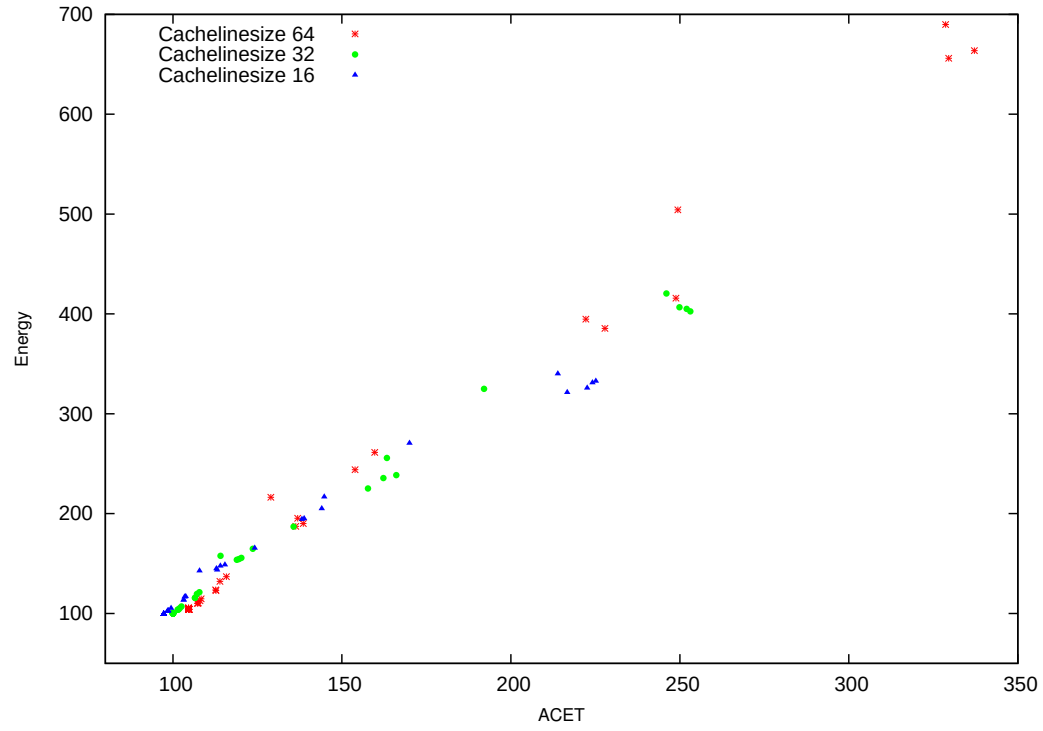
Alle Benchmarks, Überblick über die Cacheparameter

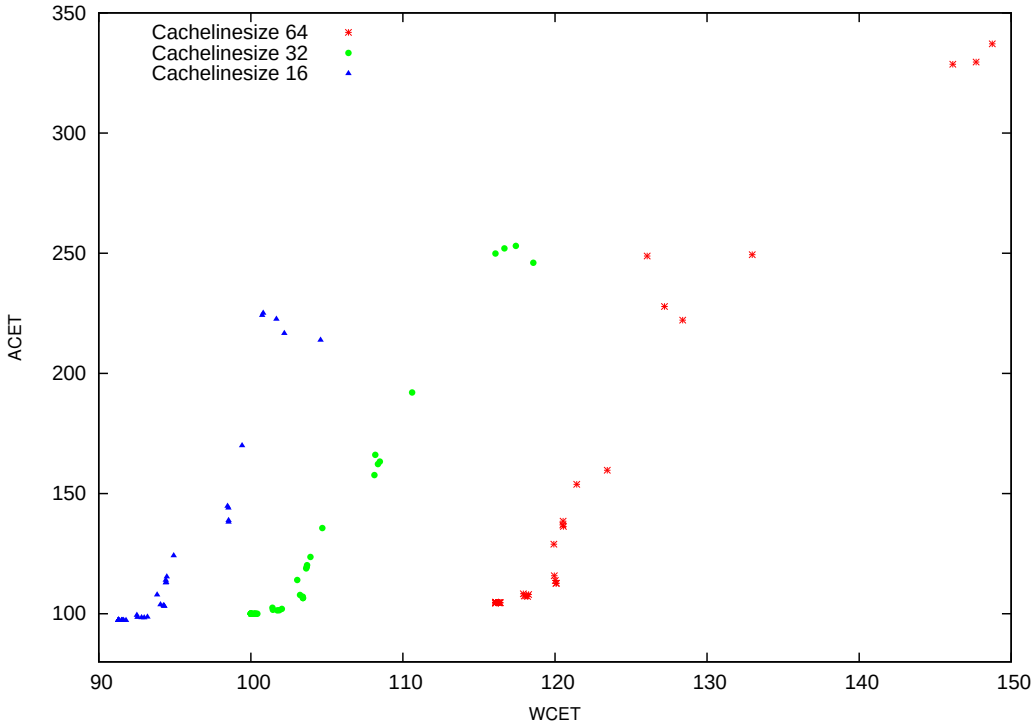




Anhang 4.2

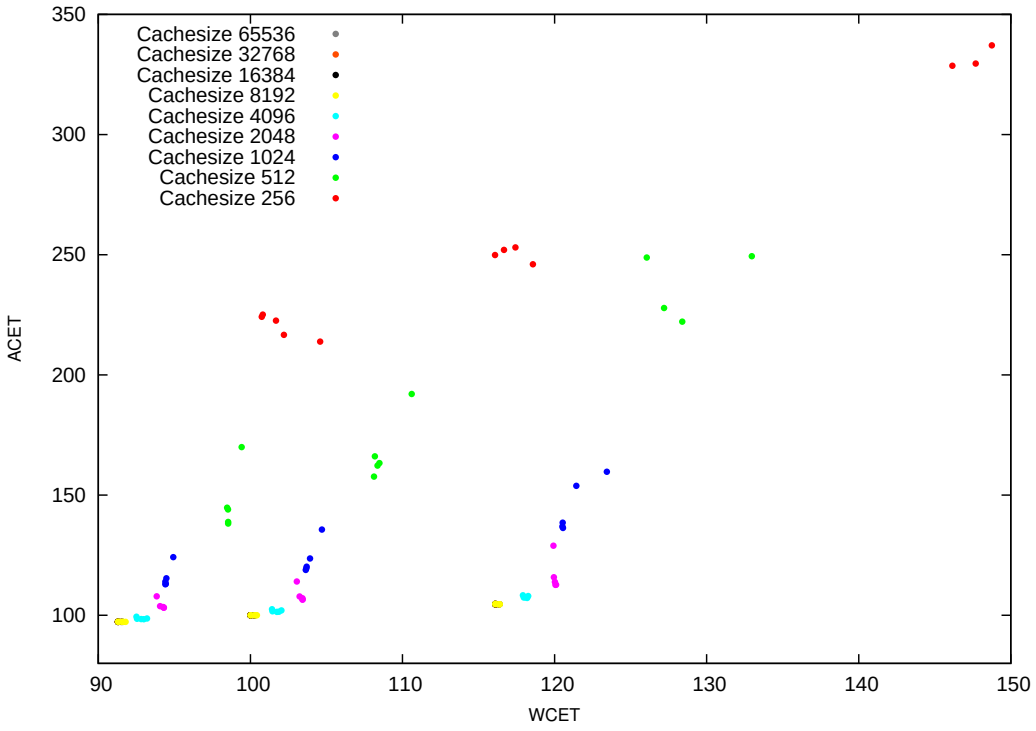
Alle Benchmarks, Überblick über die Cachezeilengröße

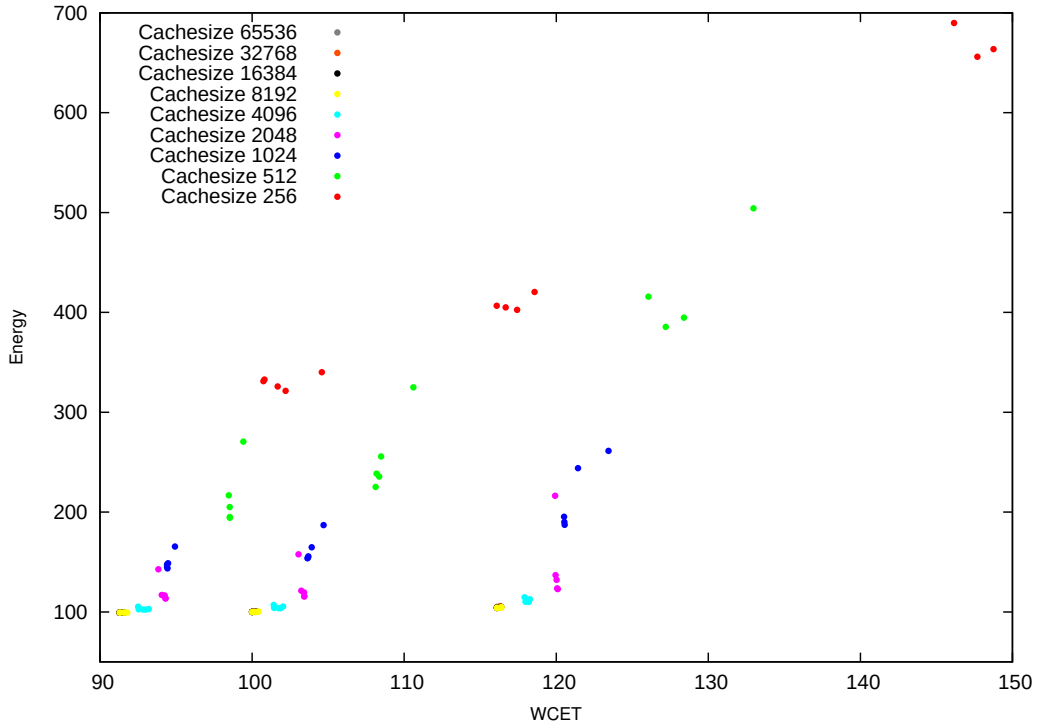




Anhang 4.3

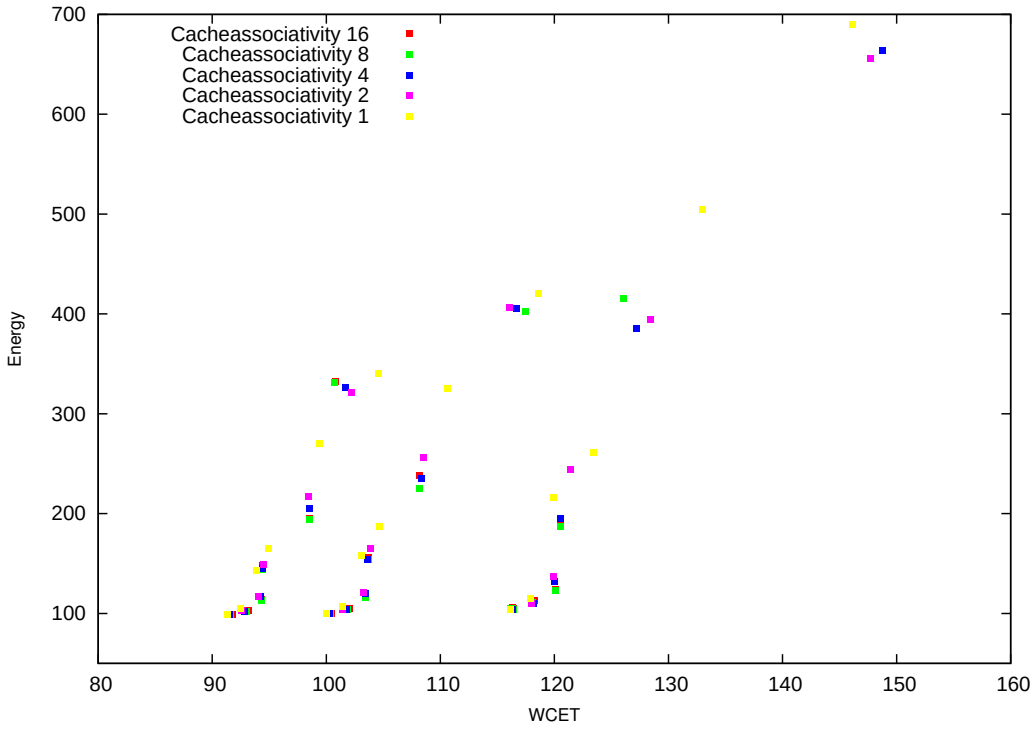
Alle Benchmarks, Überblick über die Cachegröße

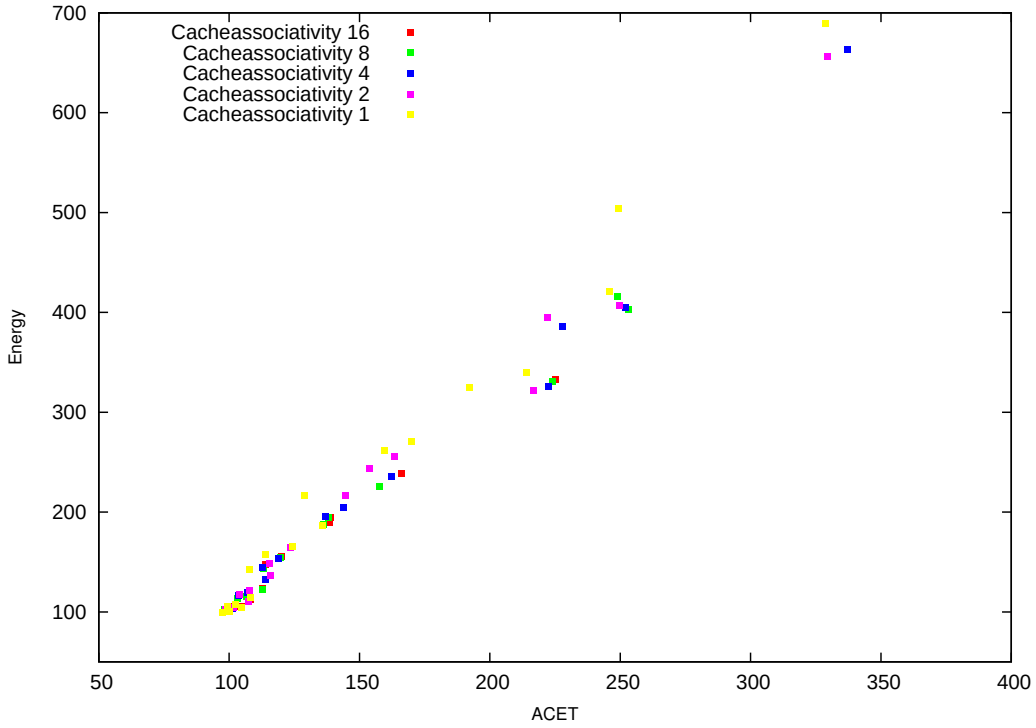




Anhang 4.4

Alle Benchmarks, Überblick über die Cacheassoziativität





Anhang 4.5

ACET	Cachezeilengröße		
Cachegröße	16 Byte	32 Byte	64 Byte
256 Byte	213.88	246.02	328.64
512 Byte	170.01	192.07	249.42
1 KByte	124.21	135.69	159.7
2 KByte	107.89	114.08	128.97
4 KByte	99.44	102.52	108.33
8 KByte	97.32	100.03	104.8
16 KByte	97.32	100.03	104.8
32 KByte	97.32	100.03	104.8
64 KByte	97.32	100.03	104.8

Tabelle 6.1: ACETs (in Prozent relativ zum Referenzwert) für verschiedene Cache- und Cachezeilengrößen bei einer Assoziativität von 1

Energie	Cachezeilengröße		
Cachegröße	16 Byte	32 Byte	64 Byte
256 Byte	340.14	420.54	689.92
512 Byte	270.6	324.89	504.31
1 KByte	165.45	186.91	261.3
2 KByte	142.66	157.71	216.35
4 KByte	105.29	107	114.7
8 KByte	99.4	100.01	104.05
16 KByte	99.4	100.02	104.06
32 KByte	99.41	100.05	104.09
64 KByte	99.44	100.07	104.13

Tabelle 6.2: Energie-Werte (in Prozent relativ zum Referenzwert) für verschiedene Cache- und Cachezeilengrößen bei einer Assoziativität von 1

Anhang 4.6

ACET	Cachezeilengröße		
Cachegröße	16 Byte	32 byte	64 Byte
1 KByte	140.26	159.22	195.44
2 KByte	120.82	143.04	191.92
4 KByte	91.81	102.81	127.35
8 KByte	91.27	100.05	117.97

Tabelle 6.3: ACETs (in Prozent relativ zum Referenzwert) für verschiedene Cache- und Cachezeilengrößen bei einer Assoziativität von 1

Energie	Cachezeilengröße		
Cachegröße	16 Byte	32 byte	64 Byte
1 KByte	157.81	173.45	212.5
2 KByte	131.36	153.6	209.94
4 KByte	93.63	103.75	129.08
8 KByte	92.89	100.01	118.78

Tabelle 6.4: Energiewerte (in Prozent relativ zum Referenzwert) für verschiedene Cache- und Cachezeilengrößen bei einer Assoziativität von 1

Anhang 4.7

ACET	Assoziativität				
Cachegröße	1	2	4	8	16
256 Byte	213.88	216.69	222.57	224.16	225.13
512 Byte	170.01	144.76	144.03	138.17	138.84
1 KByte	124.21	115.38	112.88	113.09	114.02
2 KByte	107.89	103.78	103.46	103.19	103.2
4 KByte	99.44	98.55	98.43	98.37	98.66
8 KByte	97.32	97.26	97.26	97.26	97.26
16 KByte	97.32	97.26	97.26	97.26	97.26
32 KByte	97.32	97.26	97.26	97.26	97.26
64 KByte	97.32	97.26	97.26	97.26	97.26

Tabelle 6.5: ACETs (in Prozent relativ zum Referenzwert) für verschiedene Cachegrößen und Assoziativitäten bei einen Cachezeilengröße von 16

Energie	Assoziativität				
Cachegröße	1	2	4	8	16
256 Byte	340.14	321.36	325.89	331.07	332.68
512 Byte	270.6	216.77	205.08	194.31	195.11
1 KByte	165.45	148.74	144.87	143.58	147.55
2 KByte	142.66	117.03	116.63	113.55	113.61
4 KByte	105.29	102.62	102.42	102.28	102.94
8 KByte	99.4	99.37	99.38	99.39	99.43
16 KByte	99.4	99.39	99.38	99.41	99.43
32 KByte	99.41	99.39	99.4	99.41	99.44
64 KByte	99.44	99.42	99.42	99.42	99.45

Tabelle 6.6: Energiwerte (in Prozent relativ zum Referenzwert) für verschiedene Cachegrößen und Assoziativitäten bei einen Cachezeilengröße von 16

Anhang 4.8

ACET	Assoziativität				
Cachegröße	1	2	4	8	16
256 Byte	299.81	320.86	341.08	341.88	345.74
512 Byte	251.29	249.98	249.54	249.74	249.74
1 KByte	247.74	246.68	246.43	244.54	243.66
2 KByte	206.97	225.28	236	240.68	240.66
4 KByte	152.46	134.57	134.72	133.66	142.84
8 KByte	99.53	99.53	99.53	99.53	99.53
16 KByte	99.53	99.53	99.53	99.53	99.53
32 KByte	99.53	99.53	99.53	99.53	99.53
64 KByte	99.53	99.53	99.53	99.53	99.53

Tabelle 6.7: Benchmark *adpcm_g721_verify*: ACETs (in Prozent relativ zum Referenzwert) für verschiedene Cachegrößen und Assoziativitäten bei einen Cachezeilengröße von 16 Byte

Energie	Assoziativität				
Cachegröße	1	2	4	8	16
256 Byte	577.91	619.97	675.48	676.58	683.24
512 Byte	465.74	463.18	461.2	461.94	461.98
1 KByte	458.19	455.46	455.16	452.13	450.72
2 KByte	360.22	405.36	432.12	445.11	445.35
4 KByte	231.25	188.99	190.72	188.24	207.79
8 KByte	99.56	99.56	99.58	99.59	99.65
16 KByte	99.57	99.59	99.57	99.62	99.66
32 KByte	99.59	99.59	99.6	99.62	99.67
64 KByte	99.63	99.64	99.64	99.64	99.7

Tabelle 6.8: Benchmark *adpcm_g721_verify*: Energiewerte (in Prozent relativ zum Referenzwert) für verschiedene Cachegrößen und Assoziativitäten bei einen Cachezeilengröße von 16 Byte

Anhang 4.9

WCET	Assoziativität				
Cachezeilengröße	1	2	4	8	16
16 Byte	147.72	149.03	157.45	162.76	167.07
32 Byte	145.57	145.93	155.01	158.7	161.07
64 Byte	147.53	148.8	155.12	149.21	151.14

Tabelle 6.9: WCETs für verschiedene Cachezeilengrößen und Assoziativitäten bei einen Cachegröße von 4096 Byte

ACET	Assoziativität				
Cachezeilengröße	1	2	4	8	16
16 Byte	152.46	134.57	134.72	133.66	142.84
32 Byte	160.68	144.29	142.94	145.1	164.69
64 Byte	176.81	169.82	167.45	176.16	193.72

Tabelle 6.10: ACETs für verschiedene Cachezeilengrößen und Assoziativitäten bei einen Cachegröße von 4096 Byte

Anhang 3.12

Benchmark	WCET	ACET	Energie	Codegröße
complex_multiply_fixed	100	98.06	99.53	100
complex_update_fixed	101.98	100.92	101.60	102.33
convolution_fixed	100	100	100	100
dot_product_fixed	99.58	92.27	98.35	100
iir_biquad_one_section_fixed	99.58	97.80	99.42	100
lms_fixed	100.09	106.40	100.76	100
n_real_updates_fixed	100	100	100	100
real_update_fixed	100	100	100	100
startup_fixed	99.97	101.17	100.12	100
complex_multiply_float	98.82	95.86	96.34	100
complex_update_float	99.22	100.17	99.39	98.73
convolution_float	102.81	100	102.43	102.17
dot_product_float	100	100	100	100
iir_biquad_one_section_float	100	100	100	100
lms_float	101.89	100.08	101.75	103.41
n_complex_updates_float	100	100.46	100.15	100
n_real_updates_float	100	100	100	100
real_update_float	100	100.70	100.21	100
binarysearch	100	98.48	99.76	100
cover	100	100	100	100
crc	100	98.33	99.83	100
fac	100	100	100	100
fdct	137.86	159.63	137.04	132.04
fft1	100.13	100.29	100.66	103.81
insertsort	100	99.95	99.99	100
janne_complex	100	100	100	100
jfdctint	141.82	171.80	138.01	148.22
lcdnum	100	100.98	100.12	100
ludcmp	100.37	107.02	100.91	99.27
ndes	102.37	102.80	102.24	104.44
petrinet	99.99	99.75	99.97	100
qurt	100.13	100.05	101.05	101.46
sqrt	100	100	100	100
statemate	108.73	111.75	113.19	104.20
adpcm	100.44	99.84	99.86	100.05
g721.marcuslee_encoder	100	100.98	100.18	100
g721.marcuslee_decoder	100	101.92	100.22	100
iir_1_1	100	99.58	99.81	100
lmsfir_8_1	100.32	100.44	100.19	100
mult_4_4	100	100	100	100
qmf_receive	99.79	96.09	99.46	100
qmf_transmit	99.79	96.53	99.52	100
v32.modem_achop	103.18	101.37	103.34	103.45
v32.modem_cnoise	100.89	100.75	101.08	100.84
v32.modem_eglue	102.49	101.22	102.51	102.78

Tabelle 6.11: Relative Fitness der Kriterien (in Prozent zum Referenzwert) bei der Verwendung von *Pre-RA Scheduling*

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ait11] *aiT Worst-Case Execution Time Analyzers.* `http://www.absint.com/ait/index.html`. Version: 2011

[ars11] *Understanding CPU caching and performance.* `http://www.tik.ethz.ch/~sop/pisa/?page=selvar.php`. Version: 2011

[BLTZ03] Bleuler, Stefan ; Laumanns, Marco ; Thiele, Lothar ; Zitzler, Eckart: PISA — A Platform and Programming Language Independent Interface for Search Algorithms. In: *Evolutionary Multi-Criterion Optimization (EMO 2003).* Berlin : Springer, 2003 (Lecture notes in Computer Science), S. 494 – 508

[cac11] *CACTI - An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model.* `http://quid.hpl.hp.com:9081/cacti/`. Version: 2011

[com11] *Tools to Build, Distribute and Use Virtual Prototypes.* `http://www.synopsys.com/Systems/VirtualPrototyping/Pages/VP-Tools.aspx`. Version: 2011

[DAPM94] Deb, Kalyanmoy ; Agrawal, Samir ; Pratap, Amrit ; Meyarivan, T.: A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In: *Proceedings of The International Conference on Signal Processing Applications & Technology (ICSPAT)* (1994)

[Hol92] Holland, John: *Adaptation in Natural and Artificial Systems.* MIT Press, 1992

[HP07] Hennessy, John L. ; Patterson, David A.: *Computer Architecture - A Quantitative Approach.* Fourth. Morgan Kaufmann Publishers, 2007

[icd11] *ICD-C Compiler framework.* `http://www.icd.de/es/index.html`. Version: 2011

[ISO99] ISO/IEC: International Standard 9899:1999 Programming languages C / ISO/IEC. 1999. – Forschungsbericht

[LCFM09] Lokuciejewski, Paul ; Cordes, Daniel ; Falk, Heiko ; Marwedel, Peter: A Fast and Precise Static Loop Analysis based on Abstract Interpretation, Program Slicing and Polytope Models. In: *Proceedings of The International Symposium on Code Generation and Optimization (CGO)* (2009)

[Lee01] Lee, Bo-Sik: *Vergleich des Energieverbrauchs von Cache- und Scratch-Pad-Speichern für den ARM7-Prozessor*, Technische Universität Dortmund, Lehrstuhl Informatik XII, Diplomarbeit, 2001

[Lim04] Limited, ARM: *ARM7TDMI Technical Reference Manual*. Revision: r4p1, 2004

[Lok07] Lokuciejewski, Paul: *A WCET-Aware Compiler. Design, Concepts and Realization*. VDM Verlag Dr. Müller, 2007

[LPF+10] Lokuciejewski, Paul ; Plazar, Sascha ; Falk, Heiko ; Marwedel, Peter ; Thiele, Lothar: Multi-Objective Exploration of Compiler Optimizations for Real-Time Systems. In: *Proceedings of The 13th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)* (2010)

[mrt]

[Obi98] Obitko, Marek: *Introduction to Genetic Algorithms*. `http://www.obitko.com/tutorials/genetic-algorithms/index.php`. Version: 1998

[pis11] *Download of Selectors, Variators and Performance Assessment*. `http://arstechnica.com/old/content/2002/07/caching.ars`. Version: 2011

[Sch07] Schulte, Daniel: *Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler*, Technische Universität Dortmund, Lehrstuhl Informatik XII, Diplomarbeit, 2007

[Sem07] Semiconductors, NXP: *UM10208 - LPC2880/LPC2888 User manual*, 2007

[Tan06] Tan, Lili: The Worst Case Execution Time Tool Challenge 2006: Technical Report for the External Test / Mälardalen-Universität Schweden. 2006. – Forschungsbericht

[The00] Theokharidis, Michael: *Energiemessung von ARM7TDMI Prozessor-Instruktionen*, Technische Universität Dortmund, Lehrstuhl Informatik XII, Diplomarbeit, 2000

[TMWL96] Tiwari, Vivek ; Malik, Sharad ; Wolfe, Andrew ; Lee, Mike Tien-Chien: Instruction Level Power Analysis and Optimization of Software. In: *VLSI Design, International Conference on* (1996), S. 326

[utd11] *UTDSP Benchmark Suite*. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`. Version: 2011

[ZK04] Zitzler, Eckart ; Künzli, Simon: Indicator-Based Selection in Multiobjective Search. In: Yao, Xin (Hrsg.) u. a.: *Parallel Problem Solving from Nature (PPSN VIII)*. Berlin, Germany : Springer-Verlag, 2004, S. 832–842

[ZLT01] Zitzler, Eckart ; Laumanns, Marco ; Thiele, Lothar: SPEA2: Improving the Strength Pareto Evolutionary Algorithm / Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich. 2001. – Forschungsbericht

[ZMSM94] Zivojnovic, Vojin ; Martinez, Juan ; Schläger, Christian ; Meyr, Heinrich: DSPstone: A DSP-Oriented Benchmarking Methodology. In: *Proceedings of The International Conference on Signal Processing Applications & Technology (ICSPAT)* (1994)

[ZV03] Zhang, Chuanjun ; Vahid, Frank: Cache Configuration Exploration on Prototyping Platforms. In: *14th IEEE International Workshop on rapid system prototyping*, IEEE Computer Society, 2003, S. 164

[ZVN05] Zhang, Chuanjun ; Vahid, Frank ; Najjar, Walid: A Highly Configurable Cache for Low Energy Embedded Systems. In: *ACM Transactions on Embedded Computing Systems (TECS)* Bd. Vol. 4 No. 2, 2005, S. 366