technische universität
dortmund

Masterarbeit

Schedulingverfahren zur WCET-Reduktion in eingebetteten Multicore-Systemen

Hendrik Borghorst
14. November 2013

Gutachter:
Prof. Dr. Peter Marwedel
Dipl.-Inform. Timon Kelter

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12 (Eingebettete Systeme)
http://ls12-www.cs.tu-dortmund.de

Inhaltsverzeichnis

1 Einleitung

Durch die aktuell stetig steigende Verbreitung von eingebetteten Systemen, ist das Themengebiet der Optimierungen dieser Systeme für die Forschung sehr relevant. Ein spezielles Ziel der Forschung ist es, die Systeme in Bezug auf Entwurfs- und Fertigungskosten zu optimieren. Eine Möglichkeit für Einsparungen bei den Entwicklungskosten, den Herstellungskosten und dem Energieverbrauch ist der Einsatz von Multiprozessorsystemen, welche es ermöglichen mehrere Systemaufgaben parallel auszuführen. Jedoch ist es erforderlich, bestehende Optimierungsstrategien auf die neuen Systeme anzupassen, damit deren volles Optimierungspotenzial ausgenutzt werden kann. Dazu ist es oft erforderlich neue Techniken zur Optimierung zu entwickeln, da sich die Systeme im Aufbau teilweise unterscheiden, wodurch neue Eigenschaften bei der Optimierung beachtet und ausgenutzt werden müssen. An eingebettete Systeme wird in der Regel die Anforderung gestellt, harte Zeitschranken einzuhalten, weshalb es wichtig ist, eine höchstmögliche Laufzeit für ein Programm zu bestimmen, welche *Worst-Case-Execution-Time* (WCET) genannt wird. Für sicherheitskritische Systeme wird die Anforderung gestellt, dass eine bestimmte Aufgabe stets innerhalb dieser Zeitschranken ausgeführt wird, damit ein Unfall mit eventuell katastrophalen Folgen nicht eintritt. Zur Reduktion der Gesamtkosten eines Systems ist es wünschenswert diese *WCET* zu reduzieren, was durch verschiedene Optimierungen möglich ist.

Es existiert eine Vielzahl von Prozessoren, die speziell für die Verwendung in eingebetteten Systemen ausgelegt sind. Aktuell ist die Verbreitung der *ARM-Prozessoren* hoch, weshalb sich diese Prozessoren gut für die Basis dieser Arbeit eignen [ARM05].

Ein wichtiger Unterschied zwischen Multiprozessorsystemen und Einzelprozessorsystemen ist, dass sich Multiprozessorsystemen häufig einen gemeinsamen Speicher zur Kommunikation untereinander teilen. Der Zugriff auf diesen gemeinsamen Speicher erfolgt über einen geteilten Bus, welcher alle Prozessorkerne miteinander verbindet. Da es sich um einen geteilten Bus handelt, ist es erforderlich, dass die Kommunikation, welche über diesen Bus verläuft, geregelt wird, wozu verschiedene Verfahren existieren. Diese Verfahren sind in der Regel parametrisiert, was bedeutet, dass eine gute Einstellung gewählt werden muss, damit ein System mit maximaler Effizienz arbeitet.

Die Güte der Parameter ist dabei abhängig vom jeweiligen Anwendungsfall, da beispielsweise ein System, welches häufig Kommunikation zwischen den Prozessorkernen durchführt, andere Parameter benötigt, als ein System bei dem alle Prozessorkerne unabhängig

voneinander rechnen können. Dieses Verhalten wird durch die Abbildung 1.1 dargestellt. Dort ist eine Beispielsystem modelliert, welches zwei unterschiedlich aufwändige Anwendungen ausführt. Es wird eine zeitliche Einteilung der Buszugriffe durchgeführt, was bedeutet, dass jeder Prozessorkern einen festen Zeitraum innerhalb einer Periode auf den Bus zugreifen darf. Die Anwendung des Prozessorkerns 1 (*Core 1*) benötigt pro Datenzugriff weniger Zeit als die Anwendung des Prozessorkerns 2 (*Core 2*). Daher ist eine gleichmäßige Einteilung der Buszugriffe für diesen Anwendungsfall ungeeignet und es kann durch eine flexiblere Aufteilung eine bessere Laufzeit erreicht werden. Diese Aufteilung ist im unteren Teil der Abbildung 1.1 skizziert.

Aus diesem Grund ist eine automatische Bestimmung möglichst guter Parameter wünschenswert. Eine Möglichkeit zur Bestimmung dieser Parameter besteht in der Verwendung von *evolutionären Algorithmen* [Wei07], welche eine multikriterielle Optimierung einer Parametermenge ermöglichen. Dieses Verfahren wird im Verlauf dieser Arbeit vorgestellt und eine Auswertung über das Optimierungspotenzial gegeben.

Eine weitere Möglichkeit zur Optimierung der Laufzeit besteht durch das Anordnen der Maschineninstruktionen, sodass Zugriffe auf den Bus in Bündel gruppiert werden, was zu einer höheren Busauslastung führen kann. Ebenfalls ist es möglich im Falle eine Blockade durch einen anderen Prozessorkern, Instruktionen, welche keinen Zugriff auf den Bus benötigen, zu bevorzugen, damit der Prozessorkern nicht durch unnötiges Warten still steht. Das Anordnen von Instruktionen wird *Instruction-Scheduling* genannt. In der Abbildung 1.2 wird ein Beispiel gezeigt, welches das Potenzial des Instruction-Schedulings demonstrieren soll. So lässt sich erkennen, dass der ursprüngliche Programmauszug, skizziert in der Abbildung 1.2(a), eine Laufzeit von 12 Taktzyklen benötigt. Die zweite Operation (*LDR R3,[R12] - 0x4*) benötigt drei Taktzyklen Wartezeit, in denen der Prozessorkern nicht arbeitet. Gibt es Instruktionen, welche keinen Buszugriff benötigen und keine Abhängigkeiten zu anderen Instruktionen aufweisen, können diese früher ausgeführt werden, so-

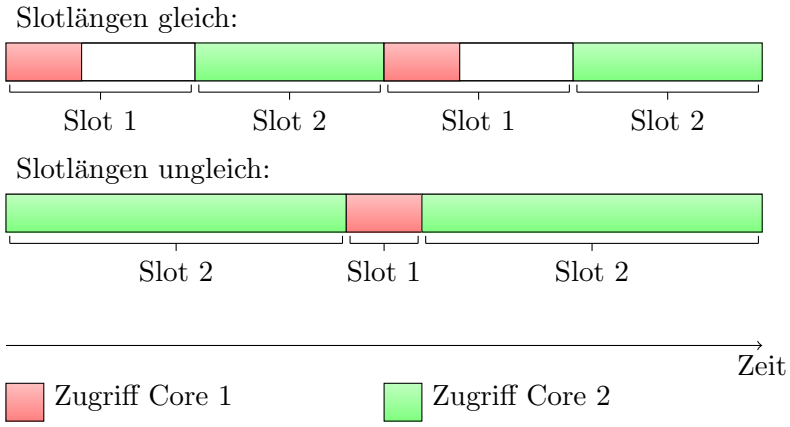


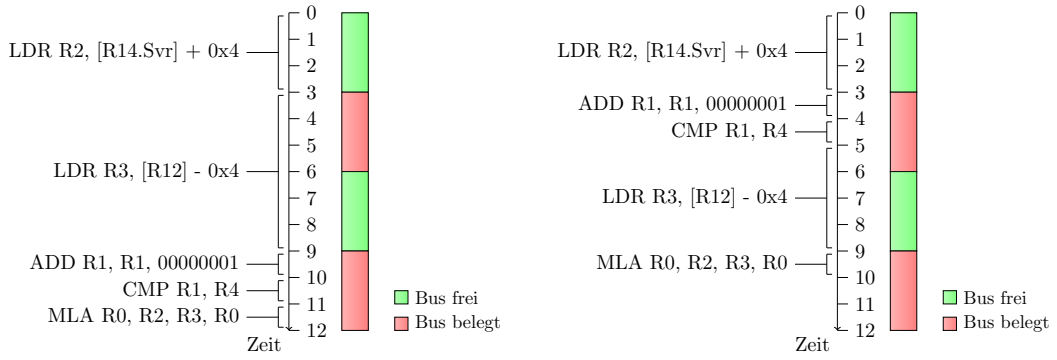
Abbildung 1.1: Optimierungspotenzial bei TDMA-Zugriff

dass der Prozessorkern durchgehend beschäftigt ist. Dies wird durch die Abbildung 1.2(b) dargestellt. In diesem Fall benötigt der Programmauszug nur noch 10 Taktzyklen, da der Prozessorkern durch die Optimierung nur noch einen Wartezyklus benötigt.

1.1 Ziele der Arbeit

Das Hauptziel dieser Arbeit ist es, das Optimierungspotenzial der verschiedenen Buskonfigurationen und deren Parameter zu evaluieren. Das Verfahren soll mithilfe des *WCC*-Compilers (WCET-aware C Compiler) des Lehrstuhls 12 durchgeführt werden [FL10]. Ebenfalls soll das Bündeln von Instruktionen zur Reduktion der *WCET* untersucht werden und eine Kombination der beiden vorgestellten Optimierungen geprüft werden. Die Hauptziele dieser Arbeit lassen sich wie folgt definieren:

- *Optimierung der Busarbitrierungsparameter durch einen evolutionären Algorithmus:* Mithilfe eines evolutionären Algorithmus soll eine Menge guter Parameter bestimmt werden, welche primär die *WCET* reduzieren soll. Zusätzlich sollen für die Optimierung die bestmögliche Laufzeit (englisch: best-case-execution-time, kurz: BCET), die durchschnittliche Laufzeit (englisch: average-case-execution-time, kurz: ACET) sowie die Busauslastung in die Optimierung einbezogen werden. Das heißt, dass eine multikriterielle Optimierung mit den genannten Kriterien durchgeführt werden soll. Die Optimierung bezieht sich jeweils auf eine Menge von Tasks und soll keine allgemeingültige gute Konfiguration der Busarbitrierung bestimmen.
- *Optimierung der WCET durch Instruction-Scheduling:* Durch die Verwendung eines *Instruction-Scheduler* soll die *WCET* reduziert werden. Der verwendete Scheduler soll eine Heuristik zur Auswahl von Instruktionen verwenden, welche speziell das Verhalten des gemeinsamen Busses des *ARM*-Prozessors ausnutzt. Zu diesem Zweck sollen verschiedene Heuristiken erstellt und evaluiert werden.



(a) Zugriff vor Scheduling

(b) Zugriff nach Scheduling

Abbildung 1.2: Optimierung mittels Instruction-Schedulings

Abschließend soll eine Kombination der beiden Verfahren erstellt und überprüft werden, ob die Kombination der Verfahren möglicherweise zu einer weiteren Reduktion der *WCET* führt.

1.2 Verwandte Arbeiten

Eine Evaluation von verschiedenen Busarbitrierungsverfahren wird in der Arbeit von T. Harde [Har13] durchgeführt. Für die Evaluation wurden jedoch keine Variation der Parameter der Busarbitrierungsverfahren durchgeführt, sodass ein eventuell vorhandenes Optimierungspotenzial ungenutzt bleibt.
Die Arbeit von A. Smolarczyk [Smo10] beschreibt, wie sich die WCET durch *Instruction-Scheduling* reduzieren lässt. Diese Arbeit betrachtet speziell einen *Infineon TriCore* Prozessor und nutzt Eigenschaften, wie die Existenz mehrerer Instruktionspipelines, dieser Hardwareplattform speziell aus. Dabei handelt es sich um einen Einzelkernprozessor, sodass das Instruction-Scheduling nicht direkt auf Mehrkernsysteme übertragbar ist.
Die Arbeiten von T. Kelter et al. [KFM+11] [KHMF13] beschreiben, wie eine Analyse der *WCET* für Multiprozessorsysteme mit gemeinsamen Bus durchzuführen ist. Es wird ein Modell der Mikroarchitekturanalyse präsentiert, welches es erlaubt eine Analyse über das Zeitverhalten für verschiedene Buszugriffsverfahren durchzuführen. Die Analyse wird in Abschnitt 2.4 im Detail vorgestellt.
Einen anderen Ansatz zur Optimierung und Analyse von echtzeitkritischen Systemen, die einen gemeinsamen Bus mit TDMA-Arbitrierung verwenden, wird von E. Wandeler et al. [WT06] beschrieben. Die Arbeit stellt eine *WCET-Analyse* auf Basis von Service- und Arrivalcurves vor, durch die die benötigte Bandbreite für den gemeinsamen Bus modelliert wird. Für die Analyse ist es erforderlich, dass für die zu analysierenden Programme der Kommunikationsbedarf auf einen gleichbleibenden Datenfluss reduziert wird, was gut für Programme mit homogenen Datenfluss funktioniert. Für Programme mit stark variierenden Datenfluss ist diese Abstraktion jedoch sehr grobgranular, wodurch es zu einer starken Überabschätzung der notwendigen Kommunikation, für Phasen in denen keine Kommunikation stattfindet, kommt. Durch diese Einschränkungen wird das vorgestellte Verfahren unflexibel und ungenau.
Einen ähnlicher Ansatz zur Optimierung mittels eines *evolutionären Algorithmus* wird von J. Rosen et al. [RNE+11] vorgestellt. Der in dieser Arbeit vorgestellte Ansatz beschäftigt sich ebenfalls mit der Optimierung der Parameter einer Busarbitrierung. Als Kriterien für die Optimierung wird nicht die WCET direkt verwendet, sondern die maximal mögliche Verzögerung von Buszugriffen (englisch: worst-case global delay kurz WCGD) und die durchschnittliche Verzögerung von Buszugriffen (englisch: average-case global delay kurz. ACGD). Durch diesen Ansatz soll gewährleistet werden, dass nicht nur der seltene Fall der maximal möglichen Laufzeit optimiert wird. Als Basis dient ebenfalls das TDMA-

Verfahren zur Busarbitrierung, welches den gemeinsamen Bus in Zeitslots für die Busteilnehmer einteilt [RNE+11]. Im Gegensatz zur in dieser Arbeit vorgestellten Optimierung mittels eines evolutionären Algorithmus wird keine evolutionäre Optimierung durchgeführt [RNE+11]. Die Parameter werden, durch das vorgestellte Verfahren, nur ausgehend von einer Startbasis, progressiv verbessert. Es findet keine *Mutation* und *Rekombination* statt, wie es durch die evolutionäre Algorithmen geschieht (siehe Kapitel 3).
Das Problem von konkurrierenden Zugriffe auf einen geteilten Speicher wird ebenfalls durch S. Bak et al. behandelt [BYPC12]. In dieser Arbeit werden verschiedene Möglichkeiten zum Scheduling der Zugriffe auf eine geteilte Ressource durch eine Simulation analysiert. Dabei werden spezielle Benchmarks verwendet, welche sich in zwei Phasen einteilen lassen. Eine Phase ist eine Zugriffsphase, während der die Zugriffe auf den gemeinsamen Speicher stattfinden. Die andere Phase ist die Rechenphase, in der die eigentlichen Berechnungen der Benchmarks stattfinden. Diese Einteilung wird *PREM* (englisch: PRedictable Execution Model) genannt [BYPC12]. Mit dem verwendeten Modell wurde, im Gegensatz zu dieser Arbeit, ein Online-Scheduling durchgeführt, welches die Taskmenge auf die einzelnen Prozessoren verteilt. Ein Online-Scheduling arbeitet während der Ausführung eines Systems. Das durch diese Arbeit präsentierte Instruction-Scheduling hingegen ist ein Offline-Scheduling, was bedeutet, dass die Einteilung vor der Ausführung eines Systems bestimmt wird. Ein Nachteil des PREM-Modells ist, dass es nicht möglich ist, unangepasste Programme zu verwenden. Die verwendeten Benchmarks müssen speziell an dieses Modell angepasst werden, damit die Einteilung der beiden Phasen ersichtlich ist.

1.3 Aufbau der Arbeit

Zu Beginn der Arbeit werden in Kapitel 2 die für diese Arbeit notwendigen Grundlagen vorgestellt. Dafür wird die notwendige Hardwareplattform, sowie die durch die Hardware verwendeten Busarbitrierungsverfahren, beschrieben. Im weiteren Verlauf wird das WCC-Framework vorgestellt und beschrieben wie die Informationen über die Laufzeit eines Systems bestimmt werden können.
Des Weiteren wird die Optimierung der Busparameter mithilfe eines evolutionären Algorithmus in Kapitel 3 beschrieben. Hierfür werden im Allgemeinen die evolutionären Algorithmen vorgestellt und dargestellt, wie sich das spezielle Problem der Verbesserung der Busparameter mithilfe dieser Algorithmen lösen lässt.
Anschließend wird durch Kapitel 4 die Optimierung auf Basis des *Instruction-Schedulers* vorgestellt, wozu die Arbeitsweise eines Instruction-Schedulers erläutert wird. Des Weiteren wird dargelegt, wie speziell die WCET unter Anbetracht des gemeinsamen Busses optimiert wird.
In Kapitel 5 wird ein Fazit gezogen und ein Ausblick auf weitere Optimierungen in diesem Themengebiet gegeben.

2 Grundlagen

Dieses Kapitel beschreibt die zum Verständnis dieser Arbeit notwendigen Grundlagen. Der Abschnitt 2.1 stellt die Umgebung vor, für welche die Optimierungen ausgelegt sind. Im weiteren Verlauf wird in Abschnitt 2.3 das *WCC-Framework* und in Abschnitt 2.4 die Analyse zur Bestimmung der *WCET* und *BCET* beschrieben.

2.1 Hardwareumgebung

Für diese Arbeit wurde als Hardwarebasis die ARM7TDMI-Kerne gewählt [ARM05], was sich durch die stetig wachsende Verbreitung von ARM-Prozessoren begründen lässt. Es wird keine aktuellere ARM-Architektur, wie zum Beispiel ARMv7, gewählt, da diese eine wesentlich kompliziertere Mikroarchitektur besitzt, was die Analyse weitaus komplexer macht. Dies würde für eine ungenaue Bestimmung der *WCET* sorgen, was in Abschnitt 2.4 erläutert wird. Die hier verwendete Hardwarearchitektur kann entweder einen, zwei, vier oder acht Prozessorkerne besitzen. Die Abbildung 2.1 zeigt schematisch, wie dieser Prozessor aufgebaut ist. Die grünen Blöcke im oberen Bereich der Abbildung bilden jeweils einen Prozessorkern ab. Innerhalb eines Kernes liegt die eigentliche Recheneinheit, gekennzeichnet mit *ARM7TDMI*, welche über einen internen Bus mit mehreren Speichereinheiten verbunden ist. Unter diesen Speichereinheiten ist ein *BootROM* vorhanden, in welchen das für den Prozessorkern bestimmte Programm geladen wird, welches nach dem Laden eines generischen Programms, auf den jeweiligen Prozessorkernen ausgeführt wird. Der Bus, für den die Zugriffe durch diese Arbeit optimiert werden, ist durch *Arbitration Bridge* gekennzeichnet. An diesem Bus sind die Speichereinheiten angebunden, welche innerhalb des orangefarbenen Blockes angegliedert sind. Innerhalb dieses Blockes liegen, jeweils getrennt für Programmdaten und Programminstruktionen, ein RAM mit Cache und ein RAM ohne Cache.

2.2 Busarbitrierungsverfahren

Das Busarbitrierungsverfahren bestimmt, welcher Prozessorkern zu welcher Zeit auf den gemeinsamen Bus zugreifen darf. Zur Bestimmung dessen gibt es mehrere Möglichkeiten, wobei die hier vorgestellten Verfahren alle nicht preemptiv arbeiten, was bedeutet, dass ein begonnener Buszugriff garantiert zu Ende ausgeführt wird.

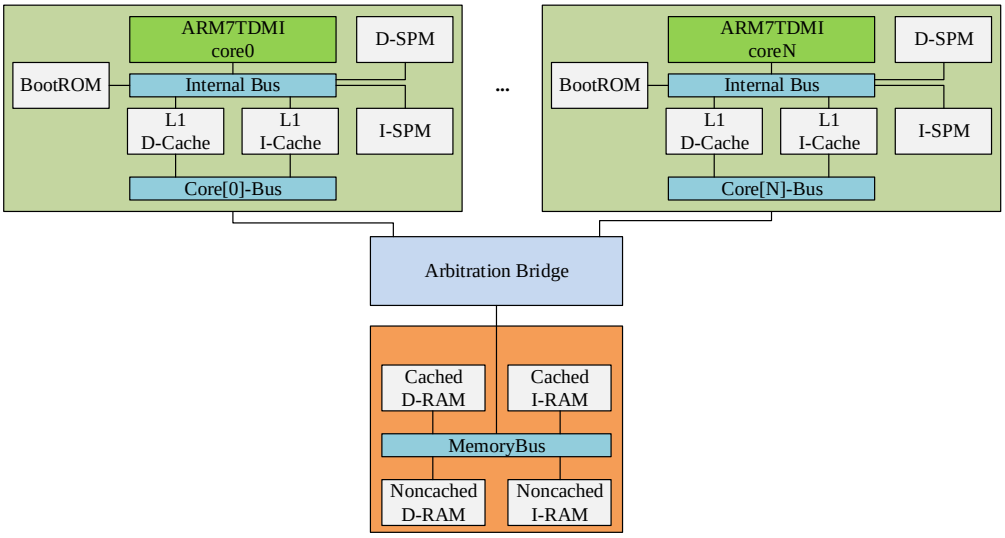


Abbildung 2.1: Hardwaremodell des ARM7-Prozessors [Har13]

2.2.1 Faire Arbitrierung

Eine simple Methodik zur Busarbitrierung ist die sogenannte *faire Arbitrierung*. Dieses Busarbitrierungsverfahren besitzt keinerlei Parameter. Es wird eine Reihenfolge festgelegt, beispielsweise identisch zu der Prozessorkernnummer, die angibt, in welcher Reihenfolge die Prozessorkerne auf den Bus zugreifen dürfen. Jeder Prozessorkern der auf den Bus zugreift, kann seine gesamte Anforderung jedes mal komplett durchführen. Nachdem ein Prozessorkern seine Anfrage beendet hat, wird der Zugriff auf den Bus an den nächsten Prozessorkern weitergegeben und dieser kann ebenfalls seine Anforderung beenden. Nachdem alle Prozessorkerne ihren Zugriff durchgeführt haben, wiederholt sich die Runde. Aus diesem Grund wird dieses Verfahren auch *Round-Robin* genannt. Eine schematische Durchführung dieses Verfahren veranschaulicht die Abbildung 2.2 für vier Prozessorkerne, welche durch unterschiedliche Farben gekennzeichnet sind. Die Abbildung zeigt ein Beispiel, bei dem die Zugriffszeiten unterschiedlich lang sind. Ebenfalls ist verdeutlicht, dass ein Prozessorkern auch auf seinen Zugriff verzichten kann, wie es für den Prozessorkern 4 (*Core* C_4) der Fall ist. Die Pfeile mit der Bezeichnung der Prozessorkerne geben an, wann eine Anfrage für den Bus eintrifft.

Für Anwendungen, die sehr intensive Buszugriffe haben, führt dieses Verfahren zu einer gleichmäßigen Auslastung des Busses. Ebenfalls ist durch dieses Verfahren gewährleistet, dass es zu keinem Verhungern (englisch: *starvation*) kommen kann, sodass jeder Prozessorkern bedient wird. Jedoch ist dieses Busarbitrierungsverfahren nicht ohne Schwächen. So kann es vorkommen, dass es zu einer Ungerechtigkeit kommt, wenn ein Prozessorkern immer sehr lange Anfragen über den Bus durchführt. In diesem Fall würden die anderen Busteilnehmer insgesamt weniger Zugriffszeit auf den Bus bekommen. In der Abbildung 2.2 könnte dies zum Beispiel der Prozessorkern 1 (*Core* C_1) sein, welcher längere Anforderun-

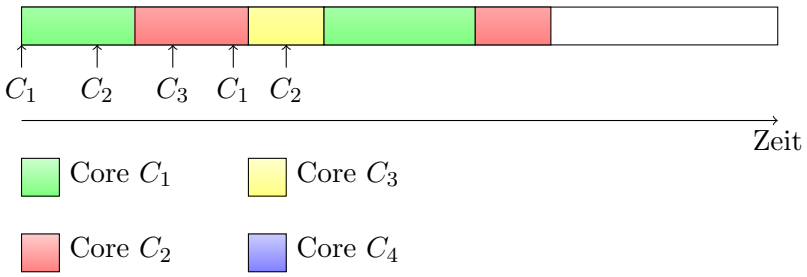


Abbildung 2.2: Buszugriff mit Fair-Arbitrierung

gen als die restlichen Prozessorkerne durchführt. Ein Nachteil dieser Busarbitrierung ist, dass eine präzise Analyse für die WCET komplex durchzuführen ist, da eine genaue Bestimmung der WCET verlangt, dass das Verhalten aller Prozessorkerne betrachtet wird. Dies wird in Abschnitt 2.4 genauer beschrieben.

2.2.2 Prioritätsbasierte Arbitrierung

Eine Alternative zur *fairen Arbitrierung* stellt die *prioritätsbasierte Arbitrierung* dar. Für dieses Busarbitrierungsverfahren wird jedem Teilnehmer eine feste Priorität zugeordnet. Diese kann beispielsweise der Prozessorkernnummer entsprechen. Gibt es mehrere konkurrierende Anfragen für den gemeinsamen Bus, so entscheidet die Priorität über die Zuteilung. Der Prozessorkern mit der höchsten Priorität bekommt den Bus zugeteilt. Eine solche Zuweisung ist in der Abbildung 2.3 dargestellt, wobei eine niedrige Prozessorkernnummer einer hohen Priorität entspricht, sodass der Zugriff mit der geringsten Prozessorkernnummer ausgewählt wird.
Auch hier geben die Pfeile an, wann eine Anforderung für den Bus eintrifft. Die ersten drei Anfragen erscheinen von C_1, C_2 und C_3. Da jedoch vor Abschluss der Anforderung von C_2 eine neue Anfrage durch C_1 eintrifft und dieser die höhere Priorität besitzt, wird dessen Anfrage bevorzugt und erst C_1 bedient, sodass C_3 warten muss. Dies zeigt einen Mangel dieses Busarbitrierungsverfahren. Es ist möglich, dass es zum Verhungern einzelner Prozessorkerne kommt, wenn höher priorisierte Prozessorkerne den Bus permanent

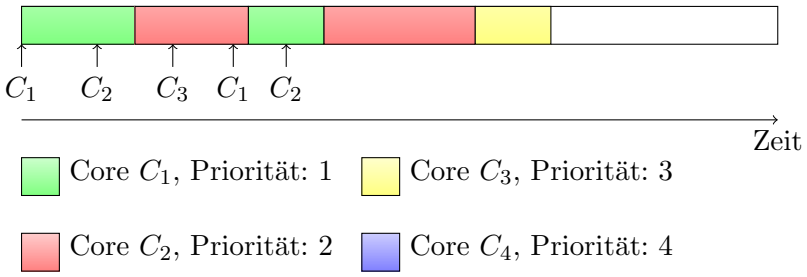


Abbildung 2.3: Buszugriff mit prioritätsbasierter Arbitrierung

belegen. Das durch die Abbildung 2.3 dargestellte Verfahren arbeitet nicht preemptiv. Es existieren jedoch auch preemptive Varianten von diesem Busarbitrierungsverfahren. Der Unterschied ist, dass bei einer preemptiven Variante Zugriffe abgebrochen werden, wenn ein höher priorisierter Zugriff angefordert wird. Ein weiterer Nachteil dieses Verfahrens ist, dass es nicht möglich ist eine gute Analyse der Laufzeit für die Prozessorkerne, welche nicht mit höchster Priorität arbeiten, durchzuführen, da diese direkt von dem Busverhalten der höher priorisierten Prozessorkerne abhängen. Dieses Verhalten wird in Abschnitt 2.4 im Detail erläutert.

2.2.3 TDMA-Arbitrierung

Ein weiteres Verfahren zur Busarbitrierung ist die *Time-Division-Multiple-Access* (kurz: TDMA) Variante. Dabei handelt es sich um ein zyklisches Verfahren mit einer Periode der Länge L. Diese Periode wird eingeteilt in n sogenannte *Slots* der Länge l_n. Diese Slots haben einen Besitzer, welcher exklusiven Zugriff während der Slotperiode hat. Damit jeder Prozessorkern bedient wird, ist es erforderlich, dass jedem Prozessorkern mindestens ein *TDMA-Slot* zugeteilt wird. Es kann nicht zum Verhungern einzelner Prozessorkerne kommen, da jeder Prozessorkern Besitzer von mindestens einem Slot ist und die Zugriffszeit durch die Länge der Slots begrenzt ist. Jedem Slot kann eine eigene Länge zugeteilt werden, wodurch auch flexible Konfigurationen möglich sind. Eine schematische Durchführung ist durch die Abbildung 2.4 gegeben. Es ist ein System mit vier Prozessorkernen und unterschiedlich langen Slotlängen modelliert. Es ist zu erkennen, dass jeder Prozessorkern innerhalb seines Slots exklusiven Zugriff auf den Bus erhält.

Ein Vorteil dieses Verfahrens ist, dass jedem Prozessorkern eine feste Zugriffszeit auf den Bus pro Periode zugesichert wird, wodurch es sich gut für harte Zeitschranken eignet. Ebenfalls sind flexible Buskonfigurationen möglich. Ein Nachteil ist jedoch, dass es vorkommen kann, dass der Bus nicht optimal ausgelastet wird, wenn zu hohe Leerlaufzeiten innerhalb eines Slots vorhanden sind. Dieses Problem soll durch die Realisierung eines evolutionären Algorithmus gemindert werden und wird in Kapitel 3 beschrieben.

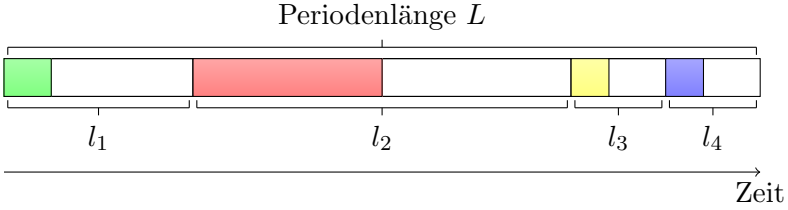


Abbildung 2.4: Buszugriff mit TDMA-Arbitrierung

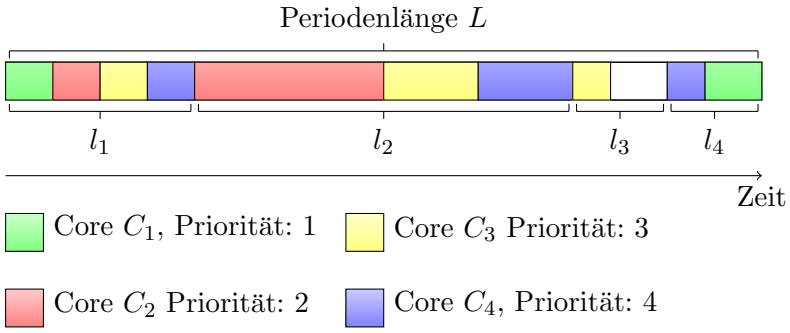


Abbildung 2.5: Buszugriff mit Priority Division-Arbitrierung

2.2.4 Priority-Division-Arbitrierung

Eine auf dem *TDMA-Verfahren* basierende Variante ist die *Priority-Division-Arbitrierung* (kurz: PD). Genau wie beim TDMA-Verfahren arbeitet das PD-Verfahren ebenfalls zyklisch in Perioden der Länge L, welche in n Slots eingeteilt sind, die eine Länge l_n als Parameter besitzen. Im Gegensatz zum TDMA-Verfahren sind die Zeitslots des PD-Verfahrens nicht exklusiv zugeteilt. Während ein Slot beim TDMA-Verfahren komplett einem Prozessorkern zugeordnet ist, ist es beim PD-Verfahren möglich, dass sich mehrere Prozessorkerne einen Slot teilen. Dazu erhält jeder Prozessorkern eine Priorität. Jeder Slot kann einen Besitzer haben, sodass dieser vor den anderen Prozessorkernen bevorzugt wird. Außerdem besteht die Möglichkeit einen Slot in einen TDMA-Slot zu konfigurieren, was bedeutet, dass dieser, genau wie beim TDMA-Verfahren, exklusiv für einen Prozessorkern zur Verfügung steht. Eine beispielhafte Durchführung einer PD-Periode ist in der Abbildung 2.5 dargestellt. Ähnlich wie beim TDMA-Verfahren (siehe Abbildung 2.4) sind auch bei diesem Beispiel Slots mit unterschiedlicher Länge vorhanden. Jedoch sind die Slots hier besser ausgenutzt und es gibt weniger Leerlaufzeiten innerhalb der Periode. Jeder Prozessorkern besitzt in diesem Beispiel einen Slot, in dem dieser bevorzugt behandelt wird.

2.3 WCC-Umgebung

Als Grundlage für die Optimierung dient der Übersetzer *WCC* (WCET-aware C Compiler) [FL10], der ein modular aufgebauter Übersetzer ist, wodurch er sich gut als Basis für die Optimierungen eignet. Beim WCC handelt es sich um einen Übersetzer, welcher die Hochsprache *C* in Maschinencode übersetzt. Im Gegensatz zur gängigen Praxis bei Übersetzern, welche primär die durchschnittliche Laufzeit (englisch: average-case-execution-time, kurz: ACET) optimieren, betrachtet der WCC speziell die höchstmögliche Laufzeit (englisch: worst-case-execution-time, kurz: WCET). Eine Übersicht über den Aufbau und Ablauf des WCC-Frameworks liefert die Abbildung 2.6. Zu Beginn des Übersetzungsvorgangs, wird ein C-Programm, welches zusätzliche Informationen enthält, durch einen Parser in eine

High-Level-Intermediate-Representation (kurz: IR) übersetzt. Dieser Schritt wird durch den *ICD-C Parser* durchgeführt, welcher schon einige Optimierungen auf dieser Ebene vornimmt [ICD13]. Bei der *IR* handelt es sich um eine Zwischenrepräsentation des Programmcodes. Dabei handelt es sich um eine Form, welche durch den Übersetzer besser weiter verarbeitet werden kann. Die zusätzlichen Informationen zu dem C-Programm müssen angeben, wie häufig beispielsweise eine Schleife maximal durchläuft. Diese Informationen werden *Flow-Facts* genannt [KKP+11]. Die Informationen sind unabdingbar für eine spätere Analyse der *WCET* und *BCET* und werden innerhalb des C-Programms durch das Sprachkonstrukt *Pragma* angegeben. Der nächste Schritt des Übersetzungsvorgangs bringt die Zwischenrepräsentation in eine maschinennähere Sprache, welche jedoch auch eine abstrakte Zwischenrepräsentation ist. Diese Zwischenrepräsentation wird aufgrund ihrer Hardwarenähe *Low-Level-Intermediate-Representation* (kurz: LLIR) genannt. Der Übersetzungsschritt von der IR zur LLIR wird durch den *Codeselector* durchgeführt. Mithilfe dieser LLIR ist es gut möglich, architekturspezifische Optimierungen durchzuführen. Daher werden die für diese Arbeit relevanten Optimierungen auf dieser Ebene realisiert. Die für diese Arbeit relevanten Bereiche sind in der Abbildung 2.6 grün hervorgehoben. Für die Optimierungen dieser Arbeit müssen die Informationen der WCET, BCET und der ACET bestimmt werden. Während die WCET und die BCET durch eine Analyse bestimmt werden, wird die ACET durch eine Simulation bestimmt. Die Informationen über die Laufzeiten sind für die weiteren Optimierungen des Übersetzers erforderlich. Die *HW-Beschreibung* enthält unter anderem die Informationen über die Busarbitrierung. Daher ist diese für die Optimierung auf Basis eines evolutionären Algorithmus von Relevanz, da diese Optimierung speziell diese Parameter variiert. Das *Instruction-Scheduling* findet auf Basis der LLIR statt. Sind alle angeforderten Optimierungen durchgeführt, wird mit der LLIR eine finale Übersetzung vorgenommen, welche das fertige Maschinenprogramm erzeugt. Dies geschieht durch den *Code-Generator*. Das fertige Maschinenprogramm ist eine Binärdatei, die Programmcode zum Start der Prozessorkerne und den eigentlichen Programmcode der jeweiligen Prozessorkerne enthält.
Für Multitaskanwendungen erhält der Übersetzer als Eingabe zusätzlich zu den *C-Programmdateien* eine Beschreibungsdatei, welche jedem Prozessorkern einen Task zuordnet. Außerdem ist es möglich für jeden Task noch verschiedene Übersetzeroptionen zu setzen, die beispielsweise spezielle Optimierungen aktivieren oder deaktivieren.

2.4 Bestimmung der Laufzeiten

Wie zuvor in Abschnitt 2.3 erwähnt, sind für die Optimierungen des WCCs verschiedene Informationen über die Laufzeit eines Systems notwendig. Diese sind die *WCET*, *BCET* und *ACET*. Dabei gibt die *WCET* an, welche maximale Laufzeit ein Programm hat. Es ist nicht möglich die genaue WCET zu bestimmen. Der Zusammenhang der Laufzeiten ist in

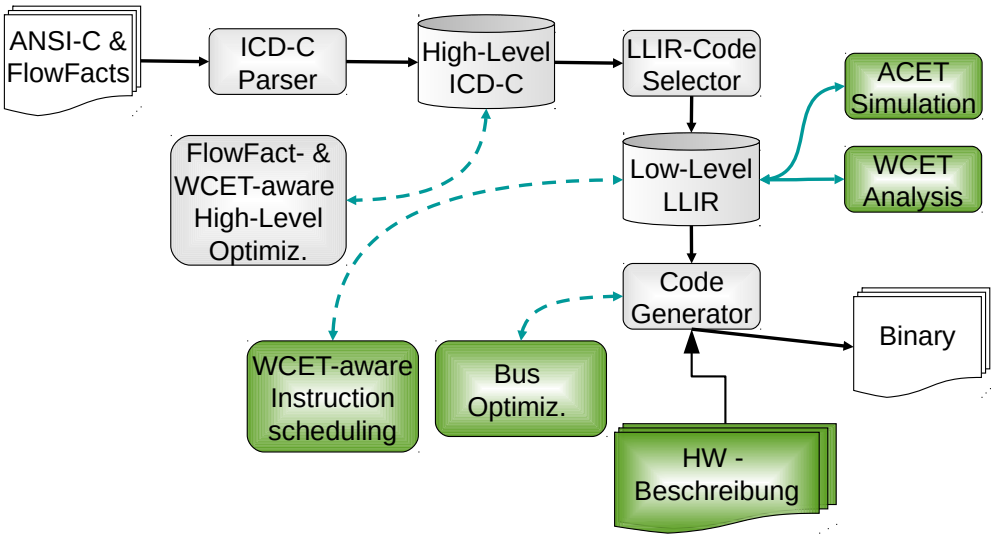


Abbildung 2.6: Aufbau des WCC-Frameworks (nach [FL10])

Abbildung 2.7 dargestellt. Die $WCET_{bestimmt}$ ist eine Überabschätzung der $WCET_{real}$, die die reale WCET des Systems darstellt. Dadurch ist sichergestellt, dass Systeme mit harter Echtzeitbedingung diese garantiert einhalten. Ähnlich ist die Bestimmung der BCET, welche die geringstmögliche Laufzeit eines Systems angibt. Auch hier findet eine Abschätzung der realen BCET ($BCET_{real}$) statt, jedoch wird die $BCET_{bestimmt}$ nach unten abgeschätzt. Zwischen der $BCET_{real}$ und der $WCET_{real}$ liegen die möglichen Werte der ACET, welche eine durchschnittliche Laufzeit angibt. Die ACET ist nicht genau zu bestimmen, da es für ein Programm nicht nur eine ACET gibt. Vielmehr gibt es einen Bereich, in dem die ACET liegen kann, da diese direkt von den Eingaben eines Programms abhängt.

Zur Bestimmung dieser Informationen gibt es mehrere Möglichkeiten. Zum einen gibt es die *dynamische Analyse*, bei der ein Programm mit verschiedenen Eingabemöglichkeiten simuliert wird. Mit diesen Eingaben wird versucht, die maximale Laufzeit zu bestimmen, was nur mit erheblichen Aufwand möglich ist, da jede mögliche Eingabe getestet werden muss, was einen massiven Zeitaufwand bedeutet, weshalb diese Form der Analyse nicht gut für Systeme, die eine harte Zeitschranke einhalten müssen, geeignet ist.

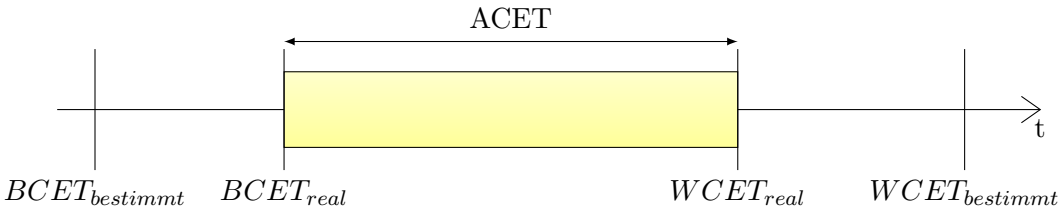
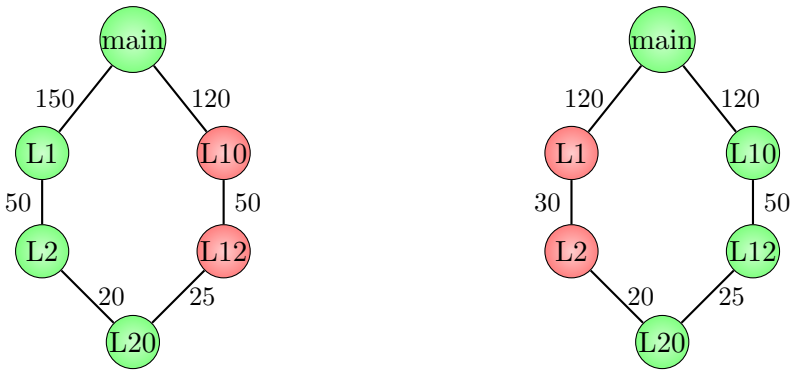


Abbildung 2.7: Verhältnis von BCET, ACET und WCET

Zum anderen gibt es die *statische Analyse* des Programmcodes. Hierbei wird das Programm nicht simuliert. Stattdessen wird der Programmcode mithilfe einer Analyse untersucht und so die Laufzeit bestimmt. Damit solche Analysen genau arbeiten, muss die spezielle Hardwarearchitektur betrachtet werden. Die statische Analyse benötigt für eine Bestimmung der BCET und WCET als zusätzliche Informationen die *Flow-Facts* [KKP+11]. Diese Flow-Facts geben beispielsweise an, wie häufig eine Schleife im bestmöglichen und im schlechtestmöglichen Fall durchlaufen wird oder wie häufig ein rekursiver Aufruf geschieht.

Für die Analyse wird ein *Kontrollflussgraph* (englisch: control flow graph, kurz: CFG) erstellt, welcher die verschiedenen Kontrollflussmöglichkeiten eines Programms modelliert. Dieser Kontrollflussgraph ist eine Abstraktion des Kontrollflusses eines Programms. Dazu wird ein Programm in *Basisblöcke* gegliedert, welche die kleinstmögliche Struktur im Kontrollflussgraph sind. Ein Basisblock fasst mehrere Instruktionen zusammen, welche keine weiteren Verzweigungen enthalten. Verzweigungen und Zusammenführungen sind bei Basisblöcken nur am Ende beziehungsweise am Anfang erlaubt. Die Basisblöcke sind im Kontrollflussgraphen durch die Knoten repräsentiert. Die Knoten werden durch gerichtete Kanten verbunden, welche angeben, wie der Kontrollfluss zwischen den Basisblöcken verläuft. Eine solche Kante, kann die BCET und WCET für den Ausführungspfad als Attribut enthalten. Verzweigungen im Programm werden durch mehrere ausgehende Kanten an einem Knoten dargestellt. Zur Bestimmung der WCET und BCET wird der längste Pfad (englisch: worst-case-execution-path, kurz: WCEP) ermittelt und die Summe der einzelnen *WCET-Attribute* berechnet.

Durch die Abbildung 2.8 wird ein schematischer Ausschnitt eines Kontrollflussgraphen dargestellt. Der *WCEP* ist durch grüne Knotenpunkte dargestellt. Ein Problem bei Optimierungen der *WCET* ist, dass der *WCEP* nicht konstant ist. Das heißt, dass sich der *WCEP* durch eine Optimierung ändern kann. Dieses Verhalten wird durch die Abbildung 2.8 gezeigt. Vor der Optimierung ist der *WCEP* über die Basisblöcke $main \rightarrow L1 \rightarrow$



vor WCET Optimierung nach WCET Optimierung

Abbildung 2.8: Änderung des WCEPs nach der Optimierung

$L2 \rightarrow L20$ mit einer *WCET* von 220 Zyklen. Nach der Optimierung ist die *WCET* dieses Pfades jedoch nur noch 150 Zyklen und der neue *WCEP* geht über die Basisblöcke $main \rightarrow L10 \rightarrow L12 \rightarrow L20$ mit einer *WCET* von 195 Zyklen. Damit eine Optimierung der WCET möglichst akkurat durchgeführt wird, muss die WCET-Analyse wiederholt werden und der *WCEP* erneut bestimmt werden.

2.4.1 Mikroarchitekturanalyse

Zur Analyse der WCET wird das Verhalten der Hardwareplattform einbezogen [KHMF13]. Dazu wird in die Analyse ein weiterer Schritt hinzugefügt, welcher die *Mikroarchitektur* der Plattform betrachtet. Ziel dieser Analyse ist es, die *WCET* und die *BCET* möglichst akkurat für Mehrkernprozessoren mit gemeinsamen Bus zu bestimmen. Diese Analyse findet an jedem Knoten des Kontrollflussgraphen statt und analysiert das Verhalten der Pipeline, des privaten Caches, des gemeinsamen Busses und des geteilten Caches. Der Aufbau dieser Analyse ist hierarchisch, sodass die Speicherhierarchie gut abgebildet wird. Für die Bestimmung der *WCET* (*BCET*) findet eine Simulation der Mikroarchitektur statt, die über die Hierarchiestufen des Speichers eine genaue Analyse des Zeitverhalten durchführt. Eine Anfrage für den gemeinsamen Bus wird daher von allen relevanten Zwischenschichten der Architektur betrachtet und analysiert. Nach dieser Analyse werden die bestimmten Informationen in der Hierarchie nach oben zurückgegeben, wo sie jeweils von der obenliegenden Hierarchieebene weiter verarbeitet werden können.

Für die Analyse der Buszugriffe wird ein abstrakter Buszustand (englisch: abstract bus state) eingeführt [KFM+11]. Dieser Buszustand ist eine Menge von *TDMA-Offsets*, die beschreiben in welchem Bereich die Busarbitrierung innerhalb einer *TDMA-Periode* liegen kann. Ein beispielhafter TDMA-Offset ist in der Abbildung 2.9 durch den gelben Bereich abgebildet. Für die Analyse wird die Annahme gemacht, dass eine maximale Zeit für jeden Zugriff auf den gemeinsamen Bus bekannt ist, welche als m_{max} bezeichnet wird. Wird ein TDMA-Verfahren mit einer Slotanzahl n und einer homogenen Slotlänge von l betrachtet [KHMF13], so ist der TDMA-Offset zu Beginn des Basisblockes b die Teilmenge

$$O_b^{in} \subseteq \{0, \ldots, nl - 1\}. \tag{2.1}$$

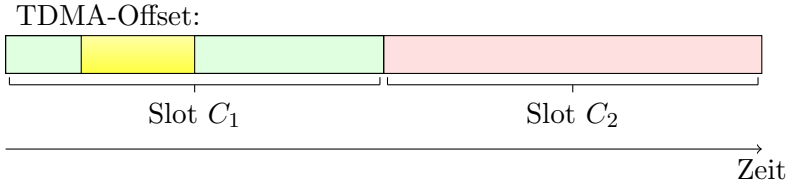


Abbildung 2.9: TDMA-Offset Werte (gelb dargestellt) innerhalb einer TDMA-Periode

Die Analyse des Busses berechnet, ausgehend von einem Start-Offset O_b^{in}, die Offsets innerhalb des Basisblockes. Dazu erhält die Busanalyse für jeden Buszugriff a_i einer Instruktion i von der Pipeline-Analyse Laufzeitinformationen T_{a_i}, die beschreiben, wie viel Zeit seit dem letzten Zugriff vergangen ist. Dazu muss die Busanalyse bestimmen, wie lange der Speicherzugriff dauert, was als D_{a_i} gekennzeichnet wird. Mit diesen Informationen kann die Busanalyse ein Offset O_i für eine Instruktion $i \in b$ bestimmen. Der Offset

$$O_i = \mu_c(O_b^{i-1}, T_{a_i}, D_{a_i}) \tag{2.2}$$

beschreibt für eine Instruktion i die Zeitwerte, welche nach der Ausführung der Instruktion i gelten können [KHMF13].

Allgemein lässt sich die Funktion μ_c definieren als

$$\mu(O, T, D) = \bigcup_{o \in O, t \in T} \{\Phi_c(o + t \bmod nl, D)\}, \tag{2.3}$$

welche das Offset bestimmt, das nach einem Zugriff auf den Speicher gilt. Die Funktion $\Phi(o, D)$ beschreibt das Verhalten des jeweiligen Busarbitrierungsverfahrens. Das heißt, es muss eine Φ-Funktion für jedes verfügbare Busarbitrierungsverfahren definiert werden. Für das TDMA-Verfahren ist die Φ-Funktion definiert als

$$\Phi_c^{TDMA}(o, D) = \begin{cases} \{o\} \oplus D & \text{wenn } o \in \omega(s_c), \\ \{s_c l\} \oplus D & \text{ansonsten,} \end{cases} \tag{2.4}$$

welche, in Abhängigkeit des aktuellen Offset-Wertes o, die Zeit angibt, die für den Zugriff notwendig ist [KHMF13]. Die Menge $\omega(s_c)$ beschreibt eine Hilfsmenge, welche die Offset-Werte des *TDMA-Fensters* s_c für den Prozessorkern c enthält. Der Slot s_c ist der erste Slot, der dem Prozessorkern c zugeordnet ist [KHMF13]. Der erste Fall der Gleichung 2.4 beschreibt den Fall, dass der aktuelle Zugriff innerhalb des TDMA-Fensters des Prozessorkerns c liegt. In dem Fall wird auf die Menge der Offset-Werte der Wert der Zugriffszeit D addiert. Der Operator $\oplus$ beschreibt die Funktion

$$\oplus(X, Y) = \{x + y | x \in X, y \in Y\}, \tag{2.5}$$

die eine Menge beschreibt, welche die Elemente einer Addition von zwei Mengen enthält.

Der zweite Fall der Gleichung 2.4 beschreibt den Fall, dass sich der aktuelle Zugriff nicht innerhalb eines Slots von Prozessorkern c befindet. In diesem Fall kann der Zugriff erst in der nächsten TDMA-Periode zu Beginn des Slots s_c durchgeführt werden, weshalb der Wert $s_c l$ auf das Offset addiert wird, was der Wartezeit entspricht, die notwendig ist bis der Prozessorkern c erneut auf den Bus zugreifen darf.

Für die PD-Busarbitrierung muss zusätzlich betrachtet werden, dass die Slots nicht exklusiv zugeteilt sind. Die abgeänderte Φ-Funktion

$$\Phi_c^{PD}(o,D) = \begin{cases} \{o\} \oplus \{0,\ldots,m_{max}\} \oplus D & \text{wenn } o \in \omega(s_c), \\ \left(\bigcup_{s_i \in s_0,\ldots,s_c-1} \phi_c(s_i,D) \right) \cup \Phi_c^{TDMA}(o,D) & \text{wenn } o \notin \omega(s_c), \\ \emptyset & \text{wenn } \nexists s_c, \end{cases} \tag{2.6}$$

beschreibt das Zeitverhalten für die PD-Busarbitrierung. Zur Beschreibung des Zugriffsverhalten außerhalb der Slots des Prozessorkerns c wird die Menge

$$\phi_c(s,D) = \begin{cases} \{sl,\ldots,(s+1)*l-m_{max} & \text{wenn } p_{s_c} > 0, \\ \emptyset & \text{ansonsten} \end{cases} \tag{2.7}$$

definiert.

Der erste Fall aus der Gleichung 2.6 beschreibt den Fall, dass der Zugriff innerhalb eines Slots erfolgt, für den der Prozessorkern c die höchste Priorität besitzt. In diesem Fall muss der Prozessor maximal auf einen aktuell bestehenden Zugriff eines anderen Prozessorkerns warten, der eine maximale Dauer von m_{max} Taktzyklen benötigt. Zusätzlich verstreicht noch die Laufzeit des Zugriffs selbst, welche mit D benannt ist. Der zweite Fall der Gleichung 2.6 beschreibt den Fall, dass ein Zugriff angefordert wird und aktuell ein Slot aktiv ist, für den der Prozessorkern c nicht die höchste Priorität besitzt. Die Funktion $\phi_c(s,D)$ beschreibt den Fall, dass der Prozessorkern c innerhalb des Slots s einen Zugriff erhält. Dies ist möglich wenn der Prozessorkern für den Slot s eine Priorität $p_{s_c} > 0$ besitzt. Der dritte Fall der Gleichung 2.6 beschreibt, dass der Prozessorkern c in keinem Slot die höchste Priorität besitzt, was bedeutet, dass keine Aussage über die maximale Laufzeit getroffen werden kann.

Für die *Faire Arbitrierung* gilt die Φ-Funktion

$$\Phi_c^{FAIR}(o,D) = \{o\} \oplus \{0,\ldots,nm_{max}\} \oplus D, \tag{2.8}$$

die zur Bestimmung der benötigten Zeit ausschließlich die Anzahl der Prozessorkerne eines Systems benötigt [KHMF13]. Die benötigte Zugriffszeit kann 0 Taktzyklen benötigen, falls der Prozessorkern c nach dem *Round-Robin*-Verfahren aktuell auf den Bus zugreifen darf. Hat der Prozessorkern c einen Zugriff gerade durchgeführt, so muss dieser alle n Zugriffe der anderen Prozessorkerne abwarten, was einer Wartezeit von nm_{max} entspricht, da ein Zugriff eine maximale Dauer von m_{max} benötigt.

Das Zugriffsverhalten für die *Prioritätsbasierte Arbitrierung* lässt sich durch

$$\Phi_c^{PRIO}(o, D) = \begin{cases} \{o\} \oplus \{0, \ldots, m_{max}\} \oplus D & \text{wenn } \forall i \in \{1, \ldots, n\} : p_i \leq p_c, \\ \emptyset & \text{ansonsten} \end{cases} \tag{2.9}$$

beschreiben, welche Ähnlichkeiten zur Beschreibung des PD-Verfahrens aufweist.
Es ist zu erkennen, dass eine Angabe über die Offset-Werte nur für den Prozessorkern mit der höchsten Priorität möglich ist. Für diesen Prozessorkern sind Wartezeiten von 0 bis zur maximalen Dauer eines Zugriffs m_{max} möglich, da dieser Prozessorkern seine Anforderungen immer erfüllt bekommt.
Ebenfalls zeigt die Gleichung 2.8, dass für die WCET bei der *fairen Arbitrierung* nur eine Überabschätzung der maximalen *Round-Robin* Zykluszeit möglich ist, da nicht bekannt ist, welche Zugriffe die anderen Prozessorkerne durchführen. Zur Lösung dieses Problems müsste das Verhalten aller beteiligten Prozessoren analysiert werden und jede mögliche Verschränkung der Buszugriffe betrachtet werden, wodurch die *faire Arbitrierung* im Vergleich zum *TDMA-Verfahren* und *PD-Verfahren* schlechter zur Bestimmung der *WCET* geeignet ist. Die Offset-Werte, die während der Analyse bestimmt werden, stellen für den *Instruction-Scheduler* in Kapitel 4 die Datenbasis für eine Heuristik zur Auswahl der Instruktionen dar, welche in Unterabschnitt 4.3.2 erläutert wird.
Des Weiteren wird für die Analyse der WCET eine *Offset-Relokation* [KFM+13] durchgeführt, durch die TDMA-Offsets von aufeinanderfolgenden Zugriffen zusätzlich präzisiert werden können. Dieses Verfahren wird an allen Schleifenköpfen angewendet.
Die *ACET* und die *Busauslastung* wird durch eine Simulation des Programmcodes ermittelt. Dafür wird der *CoMET*-Simulator der Firma Synopsys [CoM13] verwendet, wozu durch den *WCC-Übersetzer* ein minimales Startabbild erstellt wird, was die einzelnen Programme auf den einzelnen Prozessorkernen startet und ausführt. Dieser Simulator erzeugt einen sogenannten *Trace*, welcher Informationen über die ausgeführten Maschineninstruktionen enthält, die zur Bestimmung des Zeitverhalten verwendet werden können. Diese Informationen werden durch den Übersetzer eingelesen und die *ACET* für die gesamten Programme berechnet.

3 Optimierung durch einen evolutionären Algorithmus

Dieses Kapitel beschäftigt sich mit der Optimierung der Parameter der Busarbitrierungsverfahren, die in Abschnitt 2.2 vorgestellt wurden, auf Basis eines *evolutionären Algorithmus*. Dazu wird in Abschnitt 3.1 zunächst vorgestellt, wie evolutionäre Algorithmen prinzipiell funktionieren und daraufhin in Abschnitt 3.2 eine problemspezifische Beschreibung gegeben. Eine Evaluation der Optimierung wird in Abschnitt 3.4 gegeben.

3.1 Einführung in evolutionäre Algorithmen

Die evolutionären Algorithmen bieten eine Möglichkeit zur multikriteriellen Optimierung [Wei07]. Dazu greifen diese auf das Verständnis der *Populationsgenetik* zurück. Dazu wird eine Abstraktion der natürlichen Evolution von Individuen und einer Population erstellt. Es wird angenommen, dass ein Individuum ein *Genom* besitzt, welches aus einzelnen *Genen* besteht. Damit eine Evolution stattfinden kann, ist eine Menge von Individuen notwendig, welche als *Population* bezeichnet wird. Das Genom wird einer Mutation unterzogen, wenn ein neues Genom durch Reproduktion entsteht. Diese Mutation manipuliert die Gene auf eine zufällige Art und Weise. Bei den Genen des Menschen ist die Auswirkung der Mutation im Vergleich zur Vielzahl der Gene sehr gering [Wei07]. Findet diese Mutation jedoch über viele Generationen statt, so kann dies zu einer größeren Änderung des Genoms führen.

Kommt es zur Reproduktion zwischen zwei Individuen, findet eine Rekombination der Genome statt, wobei jedoch keine Evolution stattfindet, da keine neuen Gene eingeführt werden, sondern lediglich eine Kombination aus bereits vorhandenen Genen erzeugt wird.

Für die Rekombination ist die *Selektion* erforderlich, welche aus der Population zwei Individuen zur Reproduktion auswählt. Diese Selektion ist ein entscheidender Faktor für die Evolution, da diese bestimmt, welche Genome sich durchsetzen. Es sollten möglichst Genome mit hoher Qualität ausgewählt werden, damit die Population insgesamt eine hohe Qualität aufweist. Eine Population sollte jedoch nicht ausschließlich gute Genome enthalten, sondern auch eine hohe Diversität besitzen, da dies eine bessere Anpassungsfähigkeit der Population ermöglicht [Wei07].

Die evolutionären Algorithmen nutzen diese Kenntnisse zur Verbesserung von Problemstellungen aus. Dazu wird ausgehend von einer initialen Konfiguration versucht, ein Optimierungsproblem durch Rekombination und Mutation an das Optimum anzunähern. Dafür ist es notwendig, dass für jeden evolutionären Algorithmus das zu optimierende Problem formal definiert werden muss.

3.1.1 Definition (Optimierungsproblem). Ein Optimierungsproblem $(\Omega, f, \succ)$ besteht aus dem Suchraum Ω, der Bewertungsfunktion $f : \Omega \Rightarrow \mathbb{R}$ und dem Vergleichsoperator $\succ \in \{<, >\}$. Dadurch lässt sich die Menge der globalen Optima $X \subseteq \Omega$ angeben als:

$$X = \{x \in \Omega \mid \forall x' \in \Omega : f(x) \succeq f(x')\}. \tag{3.1}$$

Damit die Operationen der Evolution durchgeführt werden können, muss der Suchraum so definiert werden, dass durch zufälliges Mutieren und Rekombinieren ein gültiges Element aus dem Suchraum Ω erreicht wird. Viele Probleme lassen sich direkt als Bitstring kodieren, sodass eine zufällige Mutation durch Invertieren der Bits realisierbar ist. Die Bewertungsfunktion muss ebenfalls speziell für das zu optimierende Problem ausgelegt sein, welche mehrere Kriterien in Betracht ziehen kann. Mit der formalen Definition des Optimierungsproblem ist es möglich einen evolutionären Algorithmus problemspezifisch zu realisieren.
Durch die Abbildung 3.1 ist ein allgemeiner Ablauf von evolutionären Algorithmen skizziert. Zu Beginn eines evolutionären Algorithmus wird zunächst eine Population erstellt, die rein zufällig sein kann oder zur Reduktion der Laufzeit des Algorithmus schon Vorwissen enthalten kann. Ist diese zufällig, so muss jedes Individuum, im Falle einer Bit-

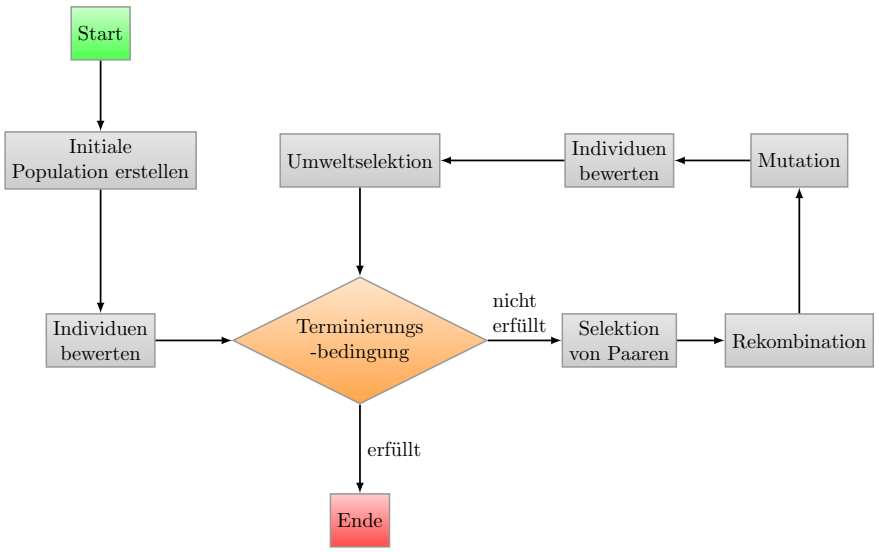


Abbildung 3.1: generischer Ablauf eines evolutionären Algorithmus (nach [Wei07])

stringkodierung der Genome, nur einen zufälligen Bitstring zugeordnet bekommen. Wurde die initiale Population erstellt, so muss jedes Individuum bewertet werden. Dazu wird die Bewertungsfunktion f mit den Parametern eines jeden Individuums aufgelöst. In einem weiterem Schritt wird mithilfe der Population die Terminierungsbedingung überprüft. Als Terminierungsbedingung kann beispielsweise die Anzahl der Generationen oder ein ϵ-Kriterium verwendet werden. Ist die Terminierungsbedingung nicht erfüllt, so werden Paare selektiert, aus denen neue Individuen erzeugt werden sollen. Mit diesen Paaren wird eine *Rekombination* durchgeführt, das heißt es werden die Genome gekreuzt. Im Falle einer Bitstringkodierung ist es beispielsweise möglich den Bitstring an einer zufälligen Position zu teilen und alle Bits vor der Teilung von einem Individuum zu übernehmen und den Rest des Bitstrings mit den Bits eines anderen Individuums aufzufüllen. Die erzeugten Individuen werden im nächsten Schritt der *Mutation* unterzogen, das heißt es werden zufällige Veränderungen an den Genomen durchgeführt. Die neu generierten Individuen müssen ebenfalls bewertet werden, dazu wird ebenfalls die Bewertungsfunktion f verwendet. Als nächster Schritt findet noch eine *Umweltselektion* statt, welche Individuen bestimmt, die für eine weitere Paarung in der Population erhalten bleiben. Nach Auswertung der Terminierungsbedingung wiederholt sich der Kreislauf des evolutionären Algorithmus bis die Terminierungsbedingung erfüllt ist.
Im folgenden Abschnitt wird beschrieben, wie es möglich ist, durch das Variieren der Parameter der verschiedenen Busarbitrierungsverfahren, eine Reduktion der WCET, BCET und ACET zu erreichen.

3.2 Repräsentation der Parameter als Genom

Wie in Abschnitt 2.2 beschrieben, gibt es zur Arbitrierung des gemeinsamen Busses mehrere Varianten. Allerdings besitzen nicht alle Verfahren Parameter, die variiert werden können. Die *faire Arbitrierung* beispielsweise besitzt keine Parameter und ist daher für die Optimierung auf Basis des *evolutionären Algorithmus* irrelevant und wird lediglich als Vergleichswert für die spätere Evaluation verwendet. Ähnliches gilt für die *prioritätsbasierte Arbitrierung*, bei der es zwar die Prioritäten als Parameter gibt, jedoch ist dieses Verfahren nicht für eine Analyse der *WCET* geeignet (siehe Abschnitt 2.4), wodurch dieses Verfahren ebenfalls für diese Optimierung ausscheidet. Es könnte lediglich bestimmt werden, wie groß der Einfluss der Optimierung auf die ACET ist, welche sich auch für die prioritätsbasierte Arbitrierung bestimmen lässt.
Das *TDMA-Verfahren* bietet verschiedene Parameter, die variiert werden können. So gibt es, wie durch die Abbildung 2.4 gezeigt, die Periodenlänge L, die Anzahl der Slots N und die Slot-Längen l_n, sowie die Slot-Besitzer O_n. Das *Priority-Division-Verfahren* besitzt zusätzlich zu den Parameter des *TDMA-Verfahrens*, noch Parameter die den Modus eines einzelnen Slots bestimmen (rein TDMA oder PD), welche im folgenden die Bezeichnung

M_n tragen. Des Weiteren besitzt jeder Prozessorkern für das *PD-Verfahren* noch einen Parameter, der die Priorität des Prozessorkerns angibt und im Folgenden mit P_c bezeichnet wird. Diese Parameter gelten im evolutionären Algorithmus als *Gene* und bilden als Menge das *Genom* eines Individuums.

3.2.1 Kodierung der Parameter

Wie im vorherigen Abschnitt bereits erwähnt, gibt es mehrere Möglichkeiten ein Genom zu repräsentieren. Eine Möglichkeit zur Kodierung ist die Konkatenation der einzelnen Gene im Bitstringformat. Ein solcher Bitstring hätte die folgende Form für das *TDMA-Verfahren*:

- $B_{Modus} = b_m$ - Busarbitrierungsverfahren
- $B_{Slotanzahl} = bn_0 \cdots bn_{nl}$ - Anzahl der Slots (Bitstring-Länge nl)
- $B_{Slotlänge,n} = bl_{n,0} \cdots bl_{n,ll}$ - Länge des Slots n (Bitstring-Länge ll)
- $B_{Slotbesitzer,n} = bo_{n,0} \cdots bo_{n,ol}$ - Besitzer des Slots n (Bitstring-Länge ol)

Daraus resultiert der Bitstring

$$\begin{aligned} B_{TDMA} = & B_{Modus} | B_{Slotanzahl} | B_{Slotlänge,0} \cdots B_{Slotlänge,N} | \\ & B_{Slotbesitzer,0} \cdots B_{Slotbesitzer,N}. \end{aligned} \tag{3.2}$$

Das erste Bit B_{Modus} gibt an, welches Busarbitrierungsverfahren für dieses Individuum ausgewählt ist. Beispielsweise wäre $b_m = 0$ das TDMA-Verfahren und $b_m = 1$ das PD-Verfahren. Der Wertebereich für die Anzahl der Slots ist in dieser Arbeit auf $[P, 64]$ beschränkt, wobei P für die Anzahl der Prozessorkerne steht. Dadurch ist sichergestellt, dass jeder Prozessorkern mindestens einen Slot als Besitzer erhalten kann. Für eine Binärkodierung der Slotanzahl sind daher sechs Bits erforderlich. Die Slotlängen werden in Taktzyklen angegeben, für die ein Wertebereich von $[3, 256]$ erlaubt ist, weshalb acht Bits pro Slot zur Kodierung notwendig sind. Diese Werte wurden gewählt, da die maximale Zugriffszeit auf den Bus pro Zugriff 3 Taktzyklen beträgt, wodurch eine Slotlänge von weniger als 3 Taktzyklen die erfolgreiche Ausführung der generierten Programme unmöglich machen würde. Für das Feld $B_{Slotbesitzer,n}$ sind die Werte $[1, P]$ erlaubt, sodass jeder Prozessorkern Besitzer werden kann. Für eine Binärkodierung sind daher $\log_2(P)$ Bits erforderlich.
Für die *PD-Arbitrierung* sind zusätzlich die Bitstrings

- $B_{Slotmodus,n} = bm_n$ und
- $B_{Kernpriorität,p} = bp_{p,0} \cdots bp_{p,pl}$

notwendig, welche den Modus des PD-Slots n und die Priorität des Prozessorkerns p definieren. Die Möglichkeiten des Bits für den Slotmodus sind $bm_n = 0$ für den TDMA-Modus oder $bm_n = 1$ für den PD-Modus. Dieses Bit ist für jeden Slot vorhanden. Die möglichen Werte für die Prozessorkern-Prioritäten sind $[1, P]$. Insgesamt resultiert daraus der Bitstring

$$\begin{aligned} B_{PD} = & B_{Modus} | B_{Slotanzahl} | B_{Slotlänge,0} \cdots B_{Slotlänge,N} | \\ & B_{Slotbesitzer,0} \cdots B_{Slotbesitzer,N} | B_{Slotmodus,0} \cdots B_{Slotmodus,N} | \\ & B_{Kernpriorität,0} \cdots B_{Kernpriorität,P}. \end{aligned} \tag{3.3}$$

Die Operationen *Rekombination* und *Mutation* lassen sich mit dieser Kodierung leicht realisieren. Zur Rekombination von zwei Elternindividuen werden jeweils ein Teilstring extrahiert und diese wieder zu einem neuen Bitstring gleicher Länge zusammengefügt. Die Mutation lässt sich so durch ein Invertieren von Bits implementieren. Dazu kann ein Bit an einer zufälligen Position des Bitstrings invertiert werden, was *Single-Bit-Mutation* genannt wird. Als Alternative können mehrere Bits, ebenfalls an einer zufällig bestimmten Positionen, invertiert werden, was als *Multi-Bit-Mutation* bezeichnet wird. Die Mutationsoperation arbeitet auf Basis einer gegebenen Wahrscheinlichkeit. Für die Single-Bit-Mutation heißt das, dass mit der gegebenen Wahrscheinlichkeit ein Bit invertiert wird. Für die Multi-Bit-Mutation wird für jedes Bit mit der gegebenen Wahrscheinlichkeit bestimmt, ob eine Invertierung stattfinden soll.

Ein Problem der Bitstringkodierung ist, dass es durch zufälliges Wählen der Bits durch die Mutation zu großen Sprüngen der Parameter kommen kann. Beispielsweise hätte eine Slotanzahl von zwei Slots die Binärkodierung $(000010)_2$. Wählt die Mutationsoperation das höchstwertige Bit aus, so entsteht der Bitstring $(100010)_2$, welcher einer 34 entspricht. Dadurch sind die Auswirkung der Mutation schwer zu kontrollieren, sodass jegliche Mutation große Auswirkung auf das Genom haben kann. Es kann zur Folge haben, dass Individuen hoher Qualität durch eine geringe Mutation zu Individuen schlechter Qualität werden. Ebenfalls ist die Kodierung mithilfe eines Bitstrings unflexibel, da die maximal gültigen Werte durch die Länge des Bitstrings begrenzt werden.

Eine Alternative zur Kodierung als Bitstring ist die Kodierung durch Dezimalzahlen. Bei einer solchen Kodierung wird das Genom als Tupel repräsentiert. Die Gene sind die einzelnen Elemente des Tupels. Für das PD-Verfahren ist das Genom G durch

$$G_{PD} = (M, N, Sl_0, \ldots, Sl_N, So_0, \ldots, So_N, Sm_0, \ldots, Sm_N, Cp_0, \ldots Cp_P) \tag{3.4}$$

zu kodieren. Die Elemente des Tupels entsprechen den direkten Werten der Buskonfiguration. Das heißt M ist der Wert des Busmodus, N entspricht der Anzahl der Slots pro Periode, Sl_n entspricht der Länge des Slots n, So_n dem Besitzer des Slots n, Sm_n dem

Modus (TDMA oder PD) des Slots n und Cp_p gibt die Priorität des Prozessorkerns p an. Das Genom für eine TDMA-Konfiguration ist ähnlich dem des PD-Verfahrens, jedoch fehlen die Slotmodi und die Prozessorkernprioritäten. Das Genom für das TDMA-Verfahren wird durch das Tupel

$$G_{TDMA} = (M, N, Sl_0 \ldots, Sl_N, So_0 \ldots, So_N) \tag{3.5}$$

definiert.
Der Vorteil dieser Kodierung ist, dass sich die Operationen eines evolutionären Algorithmus besser kontrollieren lassen. Die Rekombination ist so speziell für den Kontext des Problems der verschiedenen Busparameter möglich. Das heißt, dass nun die speziellen Felder miteinander gekreuzt werden können. Beispielsweise werden nur die Slotlängen eines Genom mit Slotlängen des anderen Genoms gekreuzt. Bei der Bitstringkodierung wurde eine Rekombination an einer zufälligen Position durchgeführt, sodass durch eine Rekombination Genome entstehen können, die große Unterschiede zu den Elterngenomen aufweisen.
Die Auswirkungen der Mutation lassen sich durch diese Kodierung ebenfalls besser kontrollieren. Statt eine Manipulation auf einzelnen Bits durchzuführen werden bei dieser Kodierung die Werte direkt durch einen Zufallsgenerator mutiert. Dadurch existiert die Möglichkeit, die oben genannten Sprünge der Parameter zu verhindern, indem der Bereich der Zufallszahlen mit einem δ-Wert begrenzt wird.

3.2.1 Definition (δ-Zufallsgenerator). Der δ-Zufallsgenerator $y = rand(x)$ erzeugt eine Zufallszahl y für die gilt: $(x - \delta) \leq y \leq (x + \delta)$.

Solche Zufallszahlen lassen sich mit dem in Definition 3.2.1 definierten Generator erzeugen. Würde beispielsweise die Slotanzahl von $N = 8$ mit einem $\delta = 4$ mutiert werden, so wären nur die Werte $[4, 12]$ als Mutationsergebnis erlaubt. Dadurch sind die Sprünge der Individuen eingeschränkt und die Mutation hat einen angemessenen Einfluss auf die Qualität der Individuen.
Ähnlich wie für die Bitstringkodierung erlaubt diese Kodierung verschiedene Formen der Mutation. Es können entweder einzelne Gene mutiert werden, ähnlich der *Single-Bit-Mutation* oder mehrere Gene mutiert werden, was der *Multi-Bit-Mutation* entspricht. Für die Mutation einzelner Gene wird durch einen Zufallsgenerator aus der Anzahl aller Gene ein Gen bestimmt, in dem Fall der Tupelkodierung ein Element des Tupels, welches anschließend mithilfe des δ-Zufallsgenerator mutiert wird. Für die Mutation mehrerer Gene wird für jedes Element des Tupels mit einer gegebenen Wahrscheinlichkeit bestimmt, ob dieses Gen mutiert werden soll. Falls dies der Fall ist, so wird auf das Gen ebenfalls der δ-Zufallsgenerator angewendet. Als Parameter δ wird ein Wert von $\delta = 5$ für die Anzahl der zu verwendeten Slots gewählt und $\delta = 30$ für die Längen der Slots, da dies einem Vielfachen der maximalen Zugriffszeit auf den Bus entspricht.

3.2.2 Nebenbedingungen

Optimierungsprobleme besitzen häufig Nebenbedingungen, welche durch den Optimierer eingehalten werden müssen. Dies ist auch bei der Optimierung der Busparameter notwendig. So muss gelten, dass die Anzahl der Slots mindestens der Anzahl der Prozessorkerne entspricht. Das heißt, dass die Bedingung

$$N \geq P \tag{3.6}$$

stets erfüllt sein muss.
Die Bedingung

$$\forall p \in \{0, \ldots, P\}\ \exists So_n\ :\ So_n = p,\ n \in \{0, \ldots, N\} \tag{3.7}$$

beschreibt, dass jedem Prozessorkern mindestens ein Slot zugeordnet wird, da ansonsten die Ausführbarkeit der generierten Programme nicht gewährleistet ist.
Durch die Bedingung

$$\forall So_n :\ So_n \leq P,\ n \in \{0, \ldots N\} \tag{3.8}$$

wird beschrieben, dass die Gene, welche die Besitzer der Slots repräsentieren, keine höheren Werte als die Anzahl der Prozessorkerne annehmen darf. Eine Verletzung dieser Bedingung würde dazu führen, dass ein nicht existenter Prozessorkern Besitzer eines Slots sein würde, was zu einem Fehler bei der Ausführung führen würde.
Damit sichergestellt ist, dass jede vergebene Priorität der Prozessorkerne eindeutig ist, muss die Bedingung

$$\forall Cp_i, Cp_j : Cp_i \neq Cp_j,\ i, j \in \{0, \ldots, P\}\ \text{, wenn } i \neq j \tag{3.9}$$

gelten.
Die Natur der zufälligen Mutation und Rekombination der evolutionären Algorithmen erfordert, dass diese Bedingungen entweder zum Zeitpunkt der Mutation beziehungsweise Rekombination eingehalten werden oder im Nachhinein kontrolliert werden. Findet eine Kontrolle nach den Operationen statt, so muss im Falle einer Verletzung der Nebenbedingungen eine Reparatur stattfinden. Für die Bedingungen 3.6 und 3.8 ist es möglich, diese Nebenbedingungen direkt während der Mutation einzuhalten, indem der Bereich der Zufallszahlen eingeschränkt wird. Diese Beschränkung wird nach der Bestimmung einer Zufallszahl überprüft, im Falle einer Verletzung der Bedingung wird der Wert auf den höchst- beziehungsweise geringst-möglichen Wert zurückgesetzt, was dem Verhalten einer Sättigungsarithmetik entspricht.
Die Nebenbedingungen der Gleichung 3.7 und Gleichung 3.9 können nicht während der Mutation einzelner Gene eingehalten werden, da zum Zeitpunkt der Mutation eines So_n

oder Cp_p Gens nicht die Ergebnisse der anderen So_n und Cp_p Gene vorliegen. Um die Nebenbedingungen zu garantieren, ist eine *Reparatur* des Genoms notwendig. Dazu wird nach Abschluss der Mutation überprüft, ob jeder Prozessorkern Besitzer eines Slots ist und keine Priorität mehrfach vergeben wurde. Ist dies nicht der Fall, so wird den Prozessorkernen, die keinen Slot besitzen, ein zufälliger Slot und jeder mehrfach vorhandenen Priorität ein neuer Zufallswert zugeteilt. Da dies ebenfalls eine zufällige Aktion ist, muss dieser Schritt wiederholt werden, bis jeder Prozessorkern einen Slot besitzt und keine Priorität mehrfach vorhanden ist. Die Reparatur wird auf zufälliger Basis durchgeführt, damit die insgesamt zufällige Natur des evolutionären Algorithmus nicht verletzt wird.

3.2.3 Bewertung des Genoms

Nachdem dargestellt wurde, wie sich die Problemstellung der Parameter der Buskonfigurationen für die Optimierung durch einen evolutionären Algorithmus repräsentieren lässt, ist es erforderlich die Bewertung eines Individuums zu erläutern. Die Bewertungsfunktion $f(x)$, welche eine Güte der Individuen bestimmt, ist im Falle dieser Optimierung eine Funktion, welche die *WCET*, *BCET*, *ACET* und die Auslastung des Busses (englisch: utilization) der einzelnen Individuen vergleicht. Dazu wird die Summe der einzelnen Laufzeiten der Prozessorkerne gebildet, wie die folgenden Gleichungen zeigen. Dabei ist P die Anzahl der Prozessorkerne und $WCET_p$ die WCET des Prozessorkerns p.

$$WCETSum = \sum_{p=0}^{P} WCET_p \tag{3.10}$$

$$BCETSum = \sum_{p=0}^{P} BCET_p \tag{3.11}$$

$$ACETSum = \sum_{p=0}^{P} ACET_p \tag{3.12}$$

$$UtilizationSum = \sum_{p=0}^{P} Utilization_p \tag{3.13}$$

3.2.2 Definition (Bewertungsfunktion $f(x)$). Eine Bewertungsfunktion für die multikriterielle Optimierung lässt sich durch die Summe der einzelnen Kriterien festlegen [Wei07]:

$$f(x) = WCETSum + BCETSum + ACETSum + (1 - UtilizationSum). \tag{3.14}$$

Insgesamt soll der realisierte evolutionäre Algorithmus die Zielfunktion minimieren. Das heißt es werden bei der Selektion Individuen mit einem niedrigen Wert von $f(x)$ bevorzugt ausgewählt. Die Busauslastung wird invertiert, da es ein Ziel ist diese zu maximieren. Durch die Gleichung 3.14 lässt sich ein erzeugtes Individuum x bewerten und mit anderen Individuen vergleichen. In Abschnitt 3.3 wird eine Alternative zu der genannten Bewertungsfunktion genannt, welche auch für die Implementierung verwendet wurde.
Die initiale Population des evolutionären Algorithmus bilden verschiedene Individuen. Dazu zählen zwei Individuen, die der Standardkonfiguration des WCC-Übersetzers für die TDMA- und PD-Arbitrierung entsprechen. Dazu werden, abhängig von einem Parameter. z zufällige Individuen mit dem TDMA-Verfahren und z zufällige Individuen mit dem PD-Verfahren in die initiale Population hinzugefügt. Die zufälligen Individuen werden dazu komplett zufällig erstellt, das heißt es wird nicht der δ-Zufallsgenerator angewendet. Dadurch ist sichergestellt, dass alle möglichen Individuen eine Chancengleichheit erfahren und alle Möglichkeiten durch den evolutionären Algorithmus ausgenutzt werden können. Erst nach der Bewertung der initialen Population wird für weitere Mutationen der δ-Zufallsgenerator verwendet.
Als Terminierungsbedingung wird eine Kombination aus Mindestanzahl und Maximalanzahl der Generationen und des ϵ-Kriteriums verwendet. Es wird überprüft, ob eine maximale Anzahl der Generationen erreicht wurde, ist dies der Fall, kann die Optimierung beendet werden. Ist dies nicht der Fall, so wird die Optimierung wiederholt, bis die Mindestanzahl der Generationen und keine Verbesserung von ϵ erreicht wurde. Der Wert von ϵ ist eine prozentuale Verbesserung gegenüber der vorherigen Generation. Da die Analyse zur Bestimmung der *WCET* und *BCET* zeitaufwändig ist, muss abgewogen werden zwischen Laufzeit der Optimierung und Qualität der Optimierung. Weitere Angaben dazu folgen in der Evaluation in Abschnitt 3.4.

3.3 Realisierung

Zur Realisierung innerhalb des WCC-Frameworks wurde das *PISA*-Framework (englisch: A Platform and Programming Language Independent Interface for Search Algorithms) verwendet [BLTZ03]. Dieses Framework bietet eine Schnittstelle für das Optimieren mithilfe von evolutionären Algorithmen. Darin enthalten ist die Umweltselektion und die Paarungsselektion von Individuen. Das Framework implementiert dazu den *SPEA2* evolutionären Algorithmus, was eine Erweiterung des *SPEA*-Algorithmus (englisch: Strength Pareto Evolutionary Algorithm) darstellt [ZLT01]. Der *SPEA2*-Algorithmus weißt jedem Individuum einen *Stärke-Wert* zu, welcher zum Vergleich verschiedener Individuen genutzt wird. Dieser Stärke-Wert wird bestimmt durch

$$S(i) = \frac{j}{p+1}. \tag{3.15}$$

Der Stärke-Wert $S(i)$ für das Individuum i wird aus der Anzahl der durch i dominierten Individuen und der Populationsgröße p bestimmt. Der Stärke-Wert kann Werte innerhalb des Intervalls $[0, 1)$ annehmen. Ein Individuum dominiert ein Anderes, falls ein Individuum in einem der Kriterien besser als das Andere ist. Die Kriterien für die vorgestellte Optimierung sind, wie zuvor genannt, die *WCET*, *BCET*, *ACET* und die *Busauslastung*. Dieses Dominanzverhalten wird als *paretodominiert* bezeichnet.

Den Stärke-Wert benötigt der *SPEA2*-Algorithmus, um den sogenannten *Fitness-Wert* [ZLT01] für ein Individuum zu bestimmen.

Der Fitness-Wert wird durch

$$F(i) = 1 + \sum_{d \in D} S(d) \tag{3.16}$$

beschrieben.

Die Menge D enthält alle Individuen, welche das Individuum i dominieren oder gleich stark wie i sind. Der SPEA2-Algorithmus wird für den im WCC-Framework implementierten evolutionären Algorithmus ausschließlich zur Selektion von Individuen verwendet. Die Bewertung der Individuen, sowie die Operationen Mutation und Rekombination werden durch interne Funktionen des WCCs realisiert. Ebenfalls wird die Terminierungsbedingung des evolutionären Algorithmus durch eine interne Bibliothek des WCCs überprüft. Für die Selektion wählt der *SPEA2*-Algorithmus alle *nicht-dominierten* Individuen aus und füllt die Population, falls notwendig, bis zur einer bestimmten Größe mit *dominierten* Individuen auf.

Das *PISA*-Framework ist mithilfe einer Bibliothek in das *WCC-Framework* integriert, sodass eine direkte Verwendung für die Optimierung möglich ist. Für die Durchführung muss ein Individuum implementiert werden, welches die in Unterabschnitt 3.2.1 beschriebene Kodierung der Parameter und die Operationen *Mutation* und *Rekombination* des Individuums umsetzt. Zusätzlich muss das Individuum eine Funktion zur Qualitätsbewertung für das *PISA*-Framework bereitstellen. Diese Bewertungsfunktion initiiert, für die Parameter des Individuums, die Analyse zur Bestimmung der WCET und BCET sowie die Simulation zur Bestimmung einer ACET und der Busauslastung. Mit diesen vier Parametern wird durch das PISA-Framework die Selektion der Individuen, welche für eine Paarung bestimmt werden, durchgeführt und anschließend dem WCC-Übersetzer mitgeteilt. In einem weiteren Schritt führt der WCC-Übersetzer die Rekombination und die Mutation dieser Paare durch und bewertet die neu entstandenen Individuen. Dieser Vorgang wird wiederholt bis die Terminierungsbedingung, welche in Unterabschnitt 3.2.3 beschrieben wird, erfüllt ist.

Um eine Reduktion der Laufzeit des evolutionären Algorithmus zu ermöglichen, wurde zusätzlich eine Datenbank auf Basis von *SQLite* [SQL13] in den WCC-Übersetzer integriert, in die, nach jedem Durchlauf des Algorithmus, das beste Genom eingetragen wird. Für die initiale Population werden eine bestimmte Anzahl an passenden Individuen, von

denen angenommen werden kann, dass diese nah an einer guten Konfiguration liegen, aus dieser Datenbank in die Population aufgenommen. Dafür wird speziell die Anzahl der verwendeten Prozessorkerne überprüft und aus der Datenbank Individuen mit der höchsten Verbesserung der WCET ausgewählt. Die Größe der Datenbank steigt durch die Anzahl der erfolgreichen Durchläufe der Optimierung stetig an, was zu einem Lerneffekt des Algorithmus führt.

3.4 Evaluation

Zur Evaluation des Optimierungspotenzial des realisierten evolutionären Algorithmus wurden verschiedene Hardwarekonfigurationen untersucht. Es wurden Tests für eine Hardwareplattform mit zwei, vier und acht Prozessorkernen des Typs *ARM7TDMI* durchgeführt. Zur Auswahl der Taskmengen wurde eine automatische Gruppierung von verschiedenen Benchmarks gewählt. Für die Erstellung der Gruppierung wurden eine Menge diverser C-Programmdateien nach ihrer durchschnittlichen Laufzeit sortiert und ein Fenster mit jeweils zwei, vier oder acht Tasks über diese Liste geschoben und zu Taskmengen gruppiert, welche im Folgenden mit *„package-NR“* bezeichnet werden. Diese Gruppierung ist gut geeignet für die Evaluation des evolutionären Algorithmus, da dieser die Summe aus den Laufzeiten der einzelnen Prozessorkernen bildet und daher gewährleistet ist, dass der Einfluss auf die Gesamtlaufzeiten durch jeden Task ungefähr gleich groß ist. Die einzelnen Benchmarks enthalten Programmbeispiele, die verschiedenen Benchmark-Sammlungen entnommen wurden. Dazu zählen Programme aus der *Mediabench* [LPMS97], welche diverse Programme aus dem Multimediabereich enthält, aus dem *MRTC*-Benchmark [MRT13], eine Sammlung von Tests speziell zur Überprüfung von WCET-basierten Optimierungen und der *UTDSP-Benchmark Suite* [UTD13], die verschiedene Tests aus dem Bereich der DSP-Anwendungen enthält, verwendet.
Für die Evaluation wurde eine maximale Generationszahl von 50 Generationen und eine minimale Generationszahl von 11 Generationen gewählt. Als ϵ-Wert wurde 1 % verwendet, das heißt, dass die Optimierung so lange fortgesetzt wird, bis keine weitere Laufzeitreduktion von 1 % eintritt oder die maximale Generationszahl erreicht wurde. Als initiale Population wurden die zuvor erwähnten Standardindividuen, jeweils 10 zufällige TDMA-Individuen und 10 zufällige PD-Individuen verwendet. Zusätzlich wurden für jeden Durchlauf noch 5 Individuen auf Basis der SQLite-Datenbank hinzugefügt. Im Folgenden werden die besten Individuen eines Benchmarks mit den angegebenen Vergleichsindividuen verglichen. Die besten Individuen können TDMA- und PD-Konfigurationen sein.
Die folgenden Abbildungen zeigen, welche Konfiguration (TDMA-Arbitrierung: rot und PD-Arbitrierung: blau) die beste Konfiguration eines Benchmarkes erzielt hat. Auf der Abbildung 3.2 ist die Reduktion der Worst-Case Laufzeit durch den evolutionären Algorithmus abgebildet, welcher auf einer Hardwarekonfiguration mit zwei Prozessorkernen

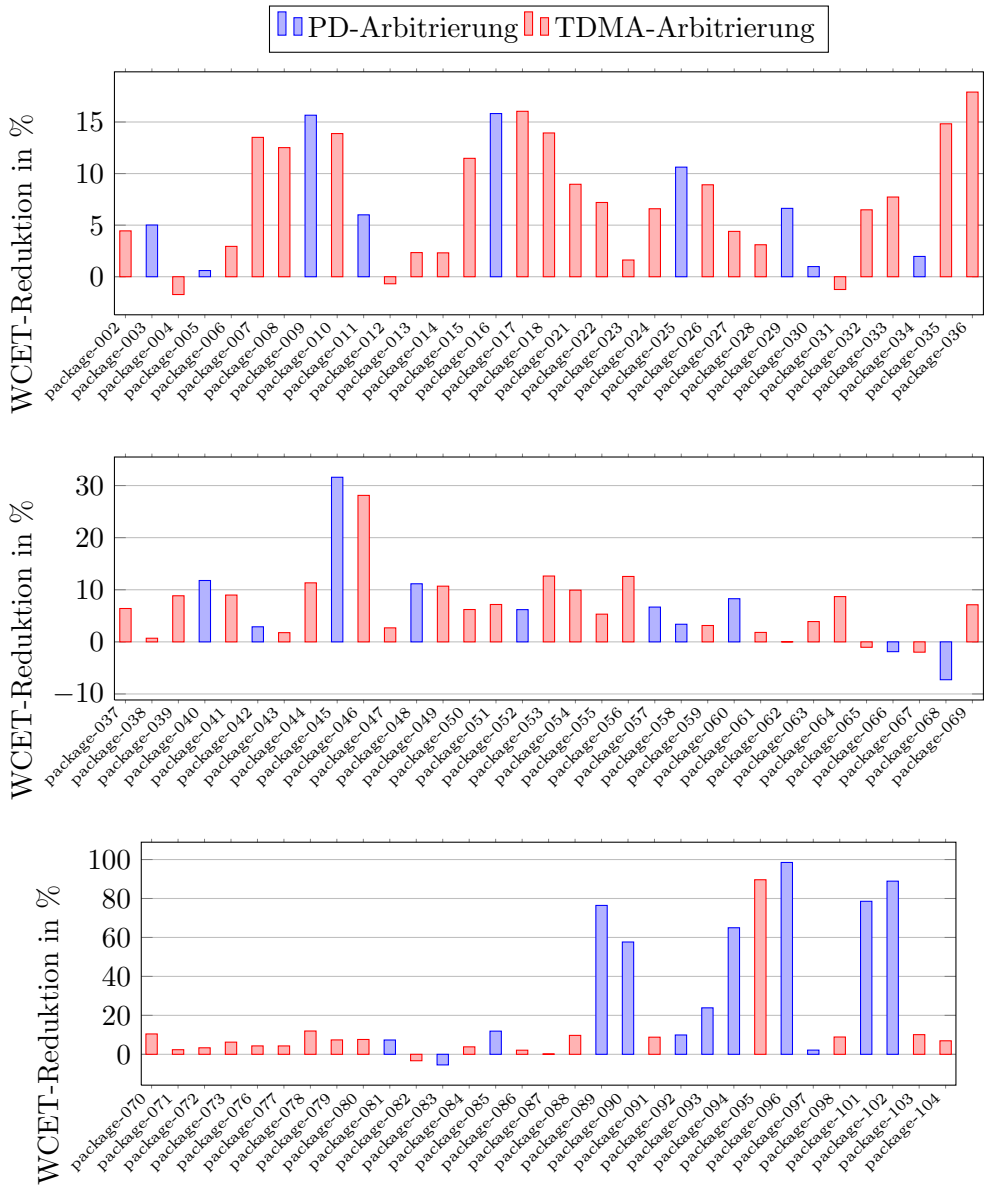


Abbildung 3.2: maximale WCET-Reduktion unter Verwendung des evolutionären Algorithmus (rot: durch TDMA-Konfiguration, blau: durch PD-Konfiguration) gegenüber fairer Arbitrierung (ARM7-x2)

gearbeitet hat. Die Reduktion der WCET ist durch einen Vergleich zur WCET mit *fairer Arbitrierung* und ohne weiteren Optimierungen ermittelt worden. Es ist zu erkennen, dass der evolutionäre Algorithmus in den meisten Fällen eine Verbesserung der Worst-Case Laufzeit erzielen kann. Auffällig sind die Werte für die Pakete, die eine hohe ACET besitzen. So wurde für das Paket *package-096* eine WCET-Verbesserung von fast 99 % erreicht. Ein Programm dieser Taskmenge ist der Task *fft_1024*, welcher eine Fourier-Transformation auf einem Array von 1024 Elementen durchführt. Der andere Task ist *lms*, welcher eine DSP-Anwendung zur Verbesserung von Signalen implementiert. Dieser Task besitzt nur zwei globale Variablen, sodass die benötigten Zugriffe auf den Bus gering sind, was durch die Abbildung 4.25 bestätigt wird. Die maximale WCET-Reduktion wurde für eine Konfiguration erreicht, bei dem der Task *fft_1024* einen Slot von 128 Taktzyklen und der Task *lms* nur 3 Taktzyklen zugewiesen bekommt. Dabei wurde die WCET des Tasks *fft_1024* von 724 083 000 Taktzyklen auf 4 549 000 Taktzyklen reduziert. Die WCET des Tasks *lms* ist allerdings von 1 171 820 Taktzyklen auf 6 348 650 Taktzyklen gestiegen, was voraussichtlich durch die gestiegenen Wartezeit für einen Buszugriff zu erklären ist. Da der evolutionäre Algorithmus die Summe der WCETs betrachtet und die Summe stark reduziert wurde, wurde diese Konfiguration als Beste ausgewählt. Ein ähnliches Verhalten ist für die anderen Taskmengen, für welche ebenfalls eine sehr hohe Verbesserung der WCET erreicht wurde, zu beobachten. In wenigen Fällen liefert die *faire Arbitrierung* bessere Ergebnisse, als die durch den evolutionären Algorithmus ermittelten Konfigurationen für das *TDMA-* und *PD-Verfahren*. Im schlechtesten Fall der Ergebnisse der Abbildung 3.2 ist zu erkennen, dass die *faire Arbitrierung* eine um 7,3 % bessere WCET erreicht.

Die WCET-Reduktion gegenüber der Standardkonfiguration des WCC-Übersetzer für das PD-Verfahren ist in der Abbildung 3.3 aufgetragen, in der erkennbar ist, dass die durch den evolutionären Algorithmus erzeugten Konfigurationen in allen Fällen besser sind als die Standardkonfiguration für das PD-Verfahren. Erneut ist das hohe Potenzial einzelner Taskmengen deutlich zu erkennen, falls die Optimierung über die Summe der WCETs durchgeführt wird. Unabhängig von den Ausreißern wurde eine maximale Verbesserung der WCET von 64 % für die Taskmenge *package-090* erreicht. Die geringste Verbesserung wurde für die Taskmenge *package-083* mit 0,67 % erzielt. Diese Taskmenge enthält die Tasks *adpcm_decoder* und *adpcm_encoder*, bei denen es sich um zwei ähnliche Tasks handelt, sodass eine Optimierung eventuell dazu beiträgt, dass der eine Tasks eine Reduktion der Laufzeit erfährt, der andere jedoch dadurch eine Steigerung, sodass sich die WCET in der Summe ausgleicht. Die WCET für das beste Individuum des Tasks *adpcm_decoder* beträgt 471 660 Taktzyklen und die des Tasks *adpcm_encoder* 472 695 Taktzyklen, wodurch die Ähnlichkeit der Tasks ersichtlich ist.

Ein Vergleich gegenüber der Standardkonfiguration der *TDMA-Arbitrierung* ist durch die Abbildung 3.4 dargelegt. Es ist ersichtlich, dass für das TDMA-Verfahren nicht in allen Fällen eine Verbesserung erzielt werden kann. Auch im Vergleich zur Standardkonfigura-

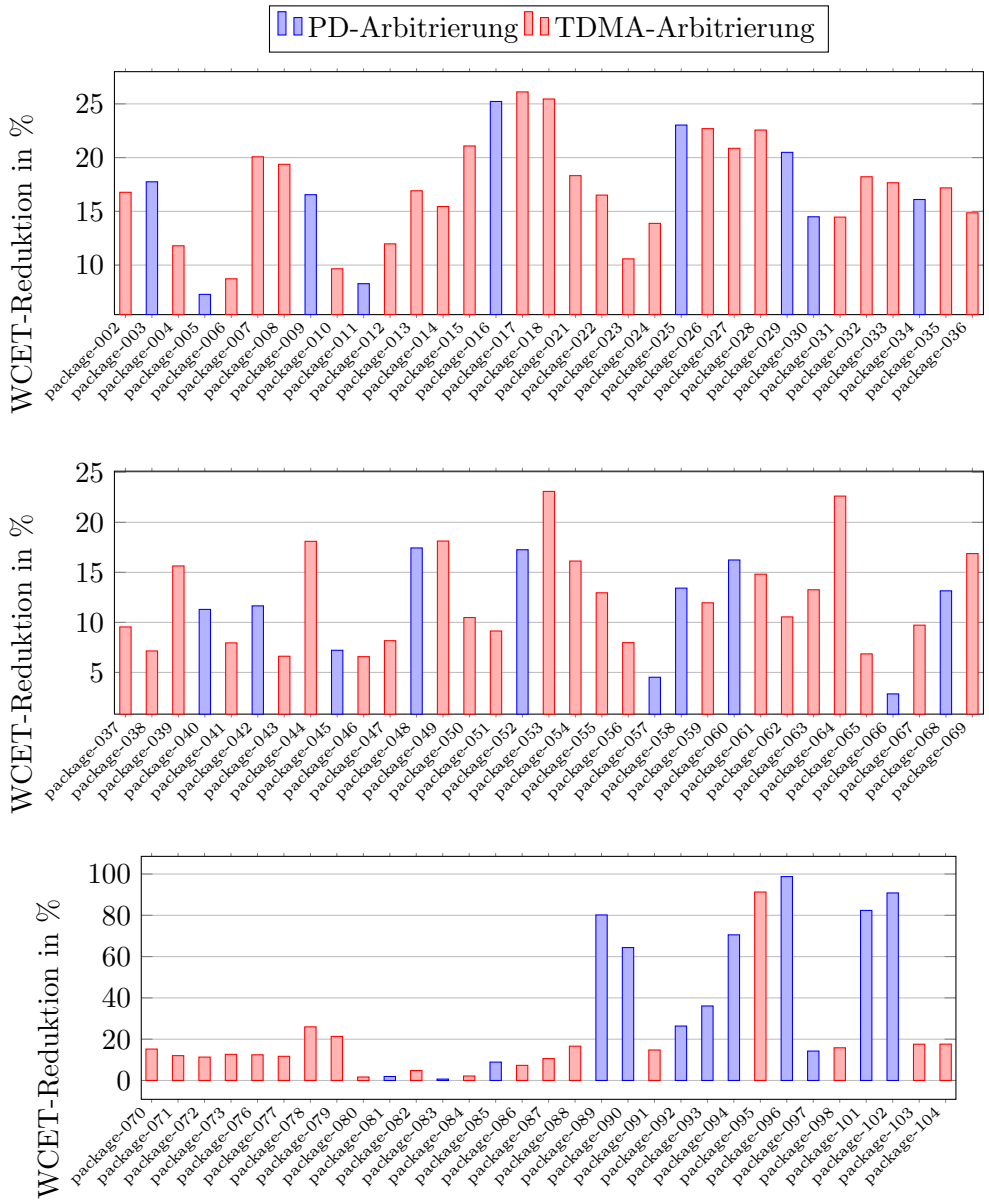


Abbildung 3.3: maximale WCET-Reduktion unter Verwendung des evolutionären Algorithmus (rot: durch TDMA-Konfiguration, blau: durch PD-Konfiguration) gegenüber Standard-PD-Konfiguration (ARM7-x2)

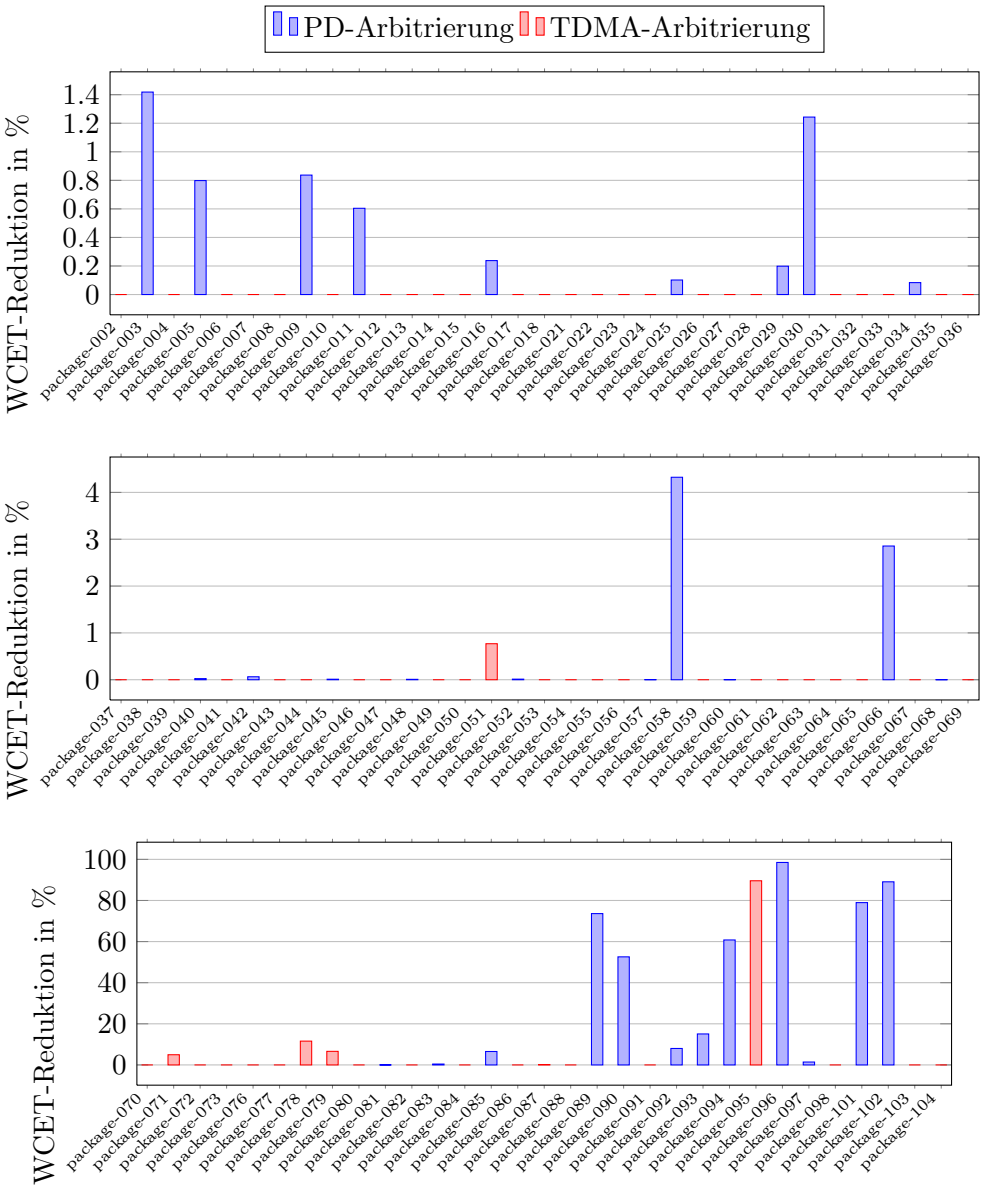


Abbildung 3.4: maximale WCET-Reduktion unter Verwendung des evolutionären Algorithmus (rot: durch TDMA-Konfiguration, blau: durch PD-Konfiguration) gegenüber Standard-TDMA-Konfiguration (ARM7-x2)

tion des TDMA-Verfahrens wurde für die komplexen Taskmengen eine hohe Reduktion erzielt. Dies zeigt jedoch, dass der evolutionäre Algorithmus gerade für Taskmengen, die unterschiedliche Tasks, bezüglich des Zugriffsverhaltens auf den gemeinsamen Bus, beinhalten, eine Verbesserung der Laufzeit erzielen kann. Zusätzlich ist es durch die Anwendung der Optimierung nicht möglich, dass sich die WCET im Vergleich zur TDMA- und PD-Standardkonfiguration erhöht, da die Individuen der Standardkonfigurationen ebenfalls ein Teil der Population sind.

Im Folgenden werden die Evaluationsergebnisse für eine Hardwareplattform mit vier Prozessorkernen vorgestellt. Durch die Abbildung 3.5 sind die Evaluationsergebnisse im Vergleich zur fairen Arbitrierung für die Hardwareplattform mit vier Prozessorkernen dargestellt. Es ist ähnlich, wie in der Abbildung 3.2 zu erkennen, dass die Ergebnisse der TDMA- und PD-Konfigurationen des evolutionären Algorithmus bessere Ergebnisse als die faire Arbitrierung liefern. Die maximale Reduktion der WCET beträgt 80,2 % für die Taskmenge *package-092* und die minimale Reduktion 5,3 % für die Taskmenge *package-056*.

Außerdem zeigt die Abbildung 3.6, dass auch für die Hardwarekonfiguration mit vier Prozessorkernen eine Reduktion der WCET, gegenüber der Standardkonfiguration für das PD-Verfahren, durch den evolutionären Algorithmus bewirkt wurde. Der evolutionäre Algorithmus konnte die WCET für die Taskmenge *package-092* um 81,8 % reduzieren und für die Taskmenge *package-043* konnte die WCET noch um 7,88 % gesenkt werden.

Für die TDMA-Arbitrierung konnte ebenfalls eine Reduktion der WCET, im Vergleich zur Ausgangskonfiguration des TDMA-Verfahrens, erreicht werden, sodass eine maximale WCET-Reduktion von 76,6 % für die Taskmenge *package-092* erreicht wurde. Für viele Taskmengen konnte gegenüber der Standardkonfiguration des TDMA-Verfahrens nur eine geringfügige Verbesserung erreicht werden, wodurch die geringe durchschnittliche Verbesserung in der Abbildung 3.14 zu erklären ist.

Die Abbildungen für die Evaluation der Hardwarekonfiguration, welche acht Prozessorkerne beinhaltet, zeigen, dass das Optimierungspotenzial für diese Anzahl Prozessorkerne geringer ist im Gegensatz zu Hardwarekonfigurationen mit zwei oder vier Prozessorkernen. Allerdings wurde für zwei Taskmengen eine äußerst hohe Verbesserung der WCET erreicht. Die WCET der Taskmenge *package-083* konnte im Vergleich zur fairen Arbitrierung um 83,35 % reduziert werden. In einigen Fällen konnte, durch die Verwendung der durch den evolutionären Algorithmus erzeugten Scheduling-Konfigurationen, keine Verbesserung gegenüber der fairen Arbitrierung gewonnen werden. In einem Fall erzielte die faire Arbitrierung eine um 7,82 % bessere WCET.

Den Vergleich zur Standardkonfiguration für das PD-Verfahren zeigt die Abbildung 3.9, auf der zu erkennen ist, dass im Vergleich zu der Standardkonfiguration für das PD-Verfahren noch Optimierungspotenzial vorhanden ist. Es wurde für alle Taskmengen eine Verbesse-

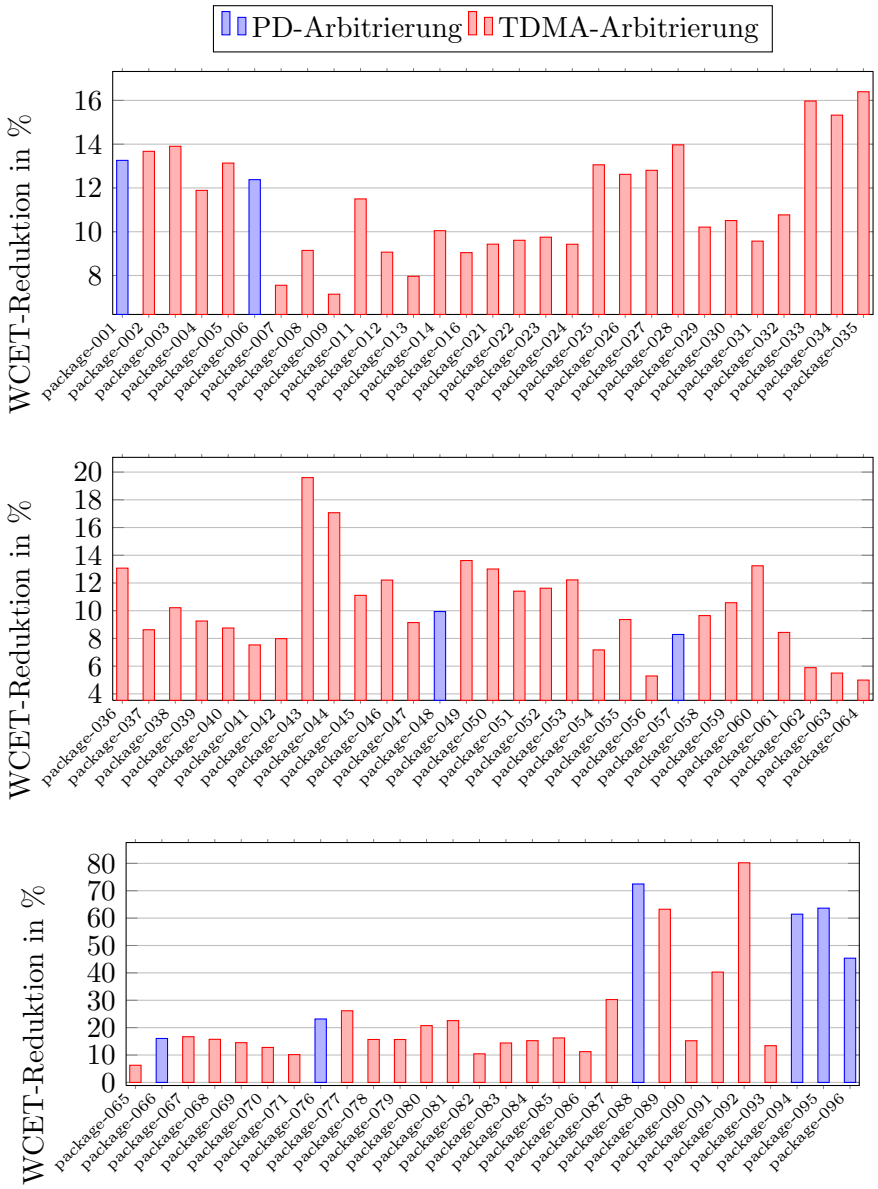


Abbildung 3.5: maximale WCET-Reduktion unter Verwendung des evolutionären Algorithmus (rot: durch TDMA-Konfiguration, blau: durch PD-Konfiguration) gegenüber fairer Arbitrierung (ARM7-x4)

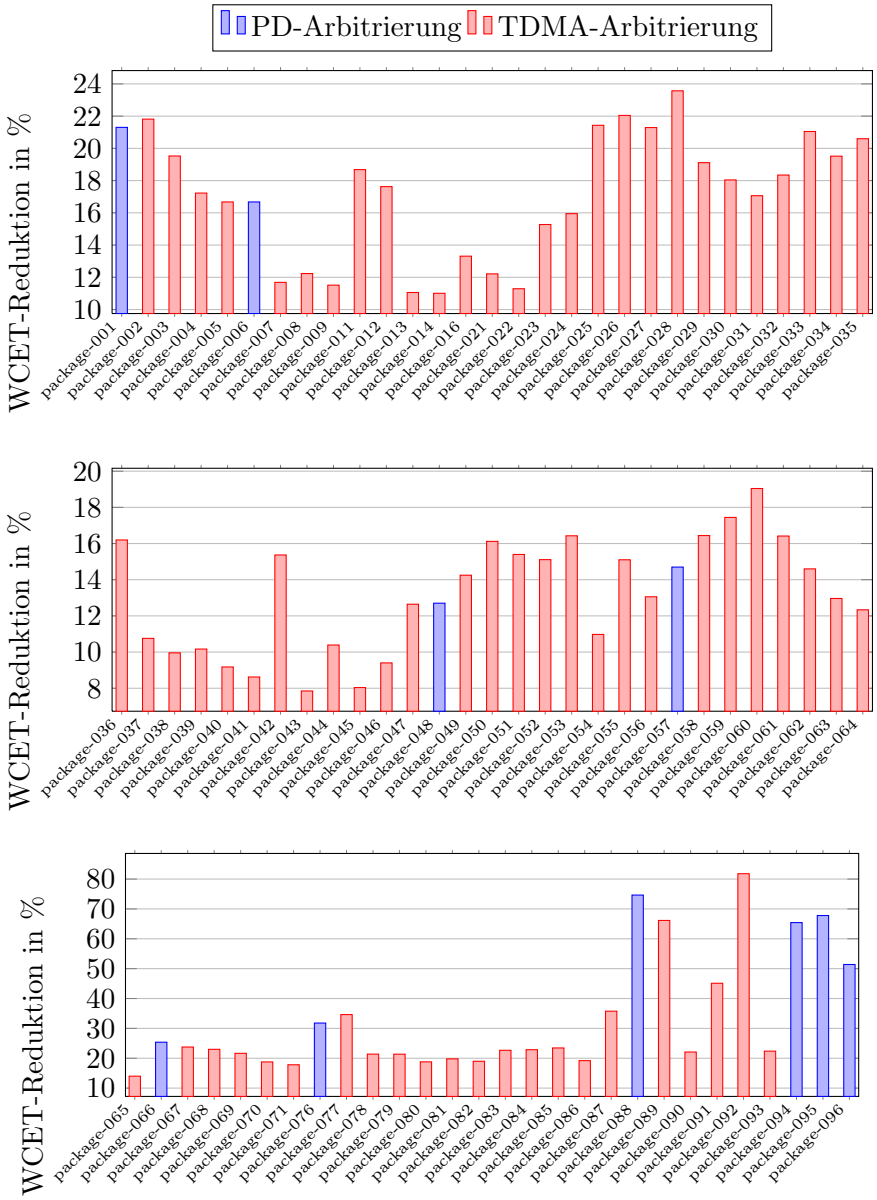


Abbildung 3.6: maximale WCET-Reduktion unter Verwendung des evolutionären Algorithmus (rot: durch TDMA-Konfiguration, blau: durch PD-Konfiguration) gegenüber Standard-PD-Konfiguration (ARM7-x4)

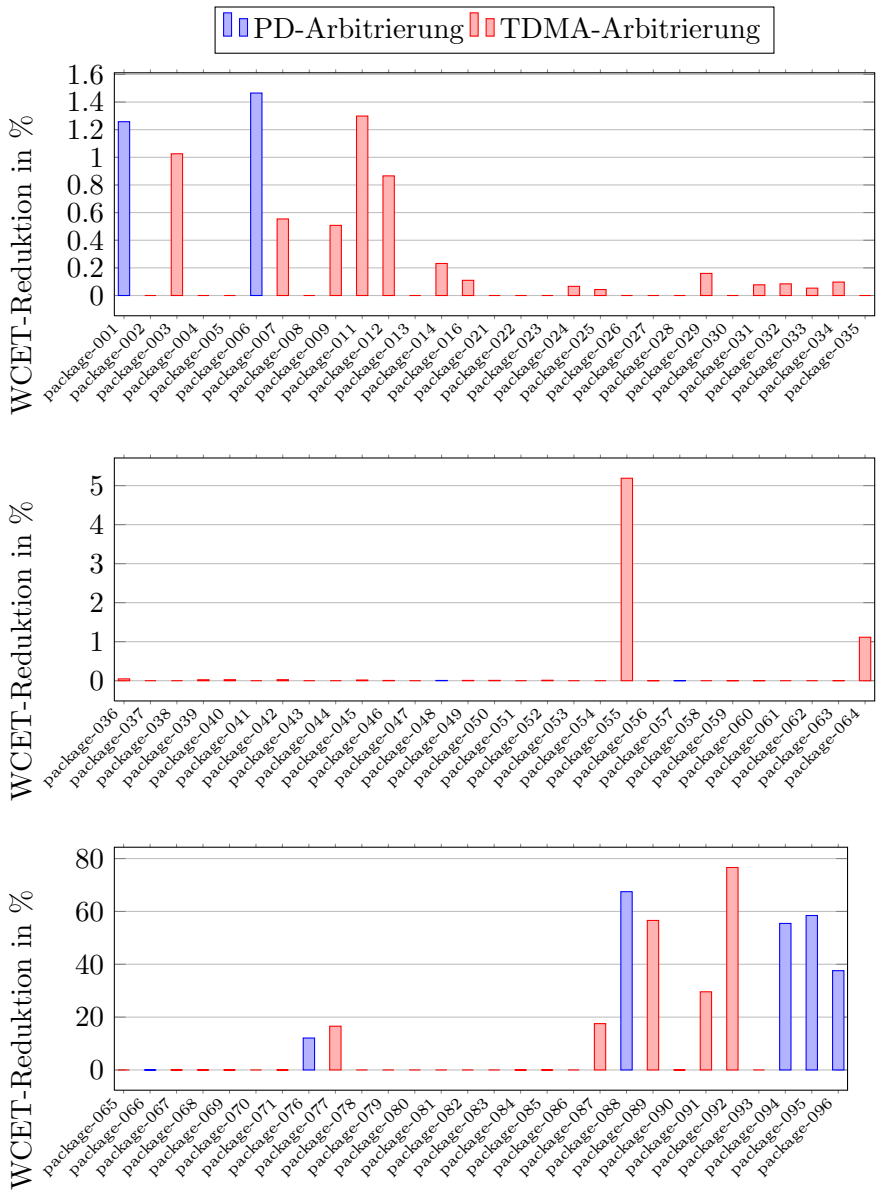


Abbildung 3.7: maximale WCET-Reduktion unter Verwendung des evolutionären Algorithmus (rot: durch TDMA-Konfiguration, blau: durch PD-Konfiguration) gegenüber Standard-TDMA-Konfiguration (ARM7-x4)

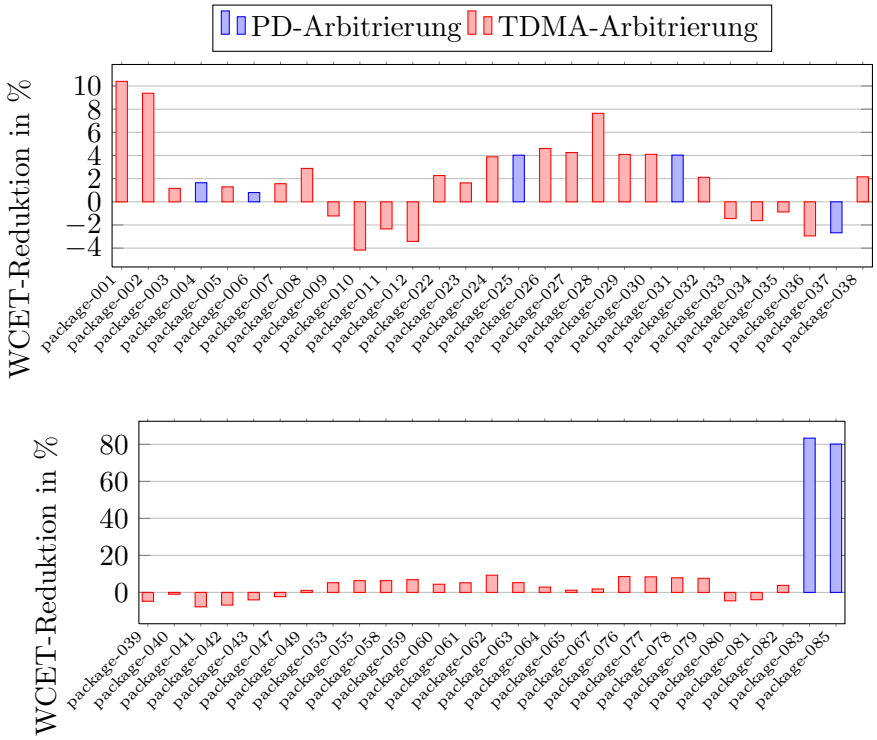


Abbildung 3.8: maximale WCET-Reduktion unter Verwendung des evolutionären Algorithmus (rot: durch TDMA-Konfiguration, blau: durch PD-Konfiguration) gegenüber fairer Arbitrierung (ARM7-x8)

rung gegenüber der Standartkonfiguration erreicht und für zwei Taskmengen, ähnlich zu der Abbildung 3.8, erneut eine hohe Verbesserung erzielt.

Ebenfalls ist durch die Abbildung 3.10 erkennbar, dass für die Standardkonfiguration der TDMA-Arbitrierung nur wenig Optimierungspotenzial vorhanden ist. So konnte für die Taskmenge *package-001* eine maximale Reduktion der WCET von 1,16 % erreicht werden. Für den Großteil der anderen Taskmengen konnte nur eine sehr geringe und in einigen Fällen keine Verbesserung erzielt werden. Jedoch ist auch in der Abbildung 3.10 zu erkennen, dass für die gleichen zwei Taskmengen, wie auf den Abbildungen 3.8 und 3.9 eine äußerst hohe WCET-Reduktion erreicht wurde.

Auf Tabelle 3.1 ist die Taskmenge *package-083* mit der WCET für die faire Arbitrierung und für das beste durch den evolutionären Algorithmus erstellten Individuums beschrieben. Durch die Tabelle ist eine Problematik des evolutionären Algorithmus zu erkennen. Da nur die Summe der WCET, BCET, ACET und Auslastung aller Tasks betrachtet wird, kann es für Taskmengen, mit inhomogener Verteilung der WCET vorkommen, dass zwar die Summe der Taskmenge reduziert wird aber die WCET der einzelnen Tasks steigt. So hat der Task *fft_1024_7* für die faire Arbitrierung die höchste WCET mit 900 951 000 Zyklen und der Task *qmf_receive* die zweit höchste WCET mit 8 366 180 Zyklen, was weniger als ein Zehntel der höchsten WCET ist. Dies zeigt, dass es nicht für alle Taskmengen

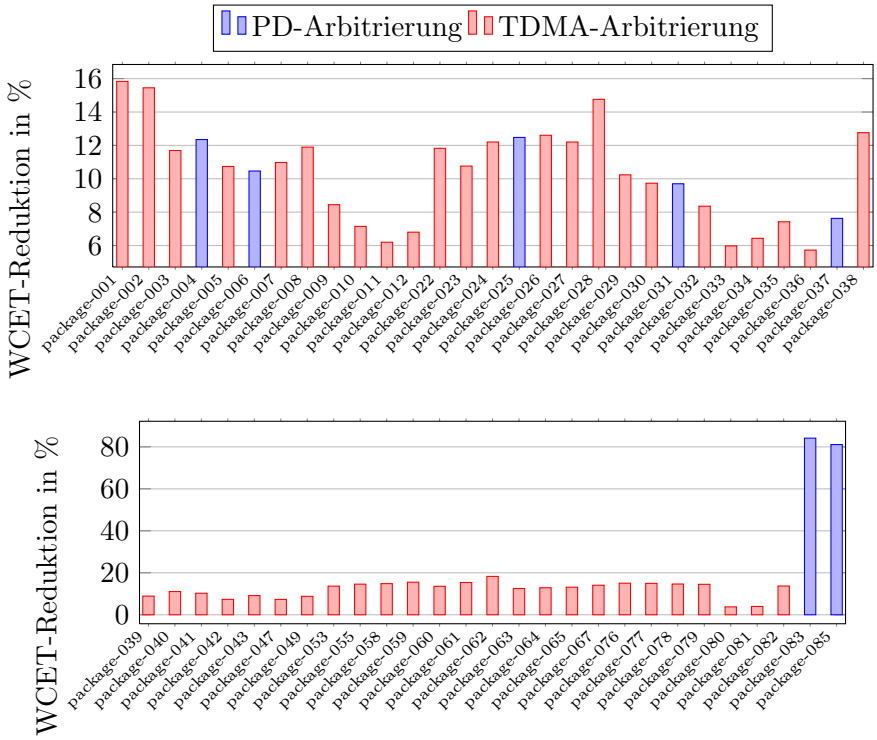


Abbildung 3.9: maximale WCET-Reduktion unter Verwendung des evolutionären Algorithmus (rot: durch TDMA-Konfiguration, blau: durch PD-Konfiguration) gegenüber Standard-PD-Konfiguration (ARM7-x8)

günstig ist, die Summe der Laufzeiten zu betrachten. Es könnte anstelle der Summe auch das jeweilige Maximum von WCET, BCET und ACET gewählt werden, sodass durch den evolutionären Algorithmus jeweils der schlechteste Task weiter optimiert wird, was in diesem Fall jedoch weiterhin der Task *fft_1024_7* wäre. Die große Abweichung der Tasks bezüglich der WCET kann auch durch eine unterschiedlich große Abweichung der WCET zur ACET liegen, da die Taskgruppierungen über die ACET erstellt wurde.

Die beste Konfiguration, welche durch den evolutionären Algorithmus für *package-083* erzeugt wurde, ist in der Abbildung 3.11 abgebildet. Es wurde eine Konfiguration des PD-Verfahrens mit 13 Slots unterschiedlicher Länge erstellt, welche unterschiedliche Besitzer haben. Die Slots 6, 7, 9, 10 und 12 sind als reine TDMA-Slots konfiguriert, was bedeutet, dass diese dem jeweiligem Besitzer exklusiv zur Verfügung stehen. Der Task *fft_1024_7*, welcher als einziger Task eine Reduktion der WCET erfahren hat, wird auf Prozessorkern C_8 ausgeführt, welcher Besitzer der Slots 2 und 3 ist, die insgesamt einen großen Teil der gesamten PD-Periode ausmachen.

Durch die Abbildung 3.12 ist die Busauslastung der Taskmengen, für die Hardwarekonfiguration mit acht Prozessorkernen, dargestellt, auf der zu erkennen ist, dass die Taskmenge *package-083* für die Standardkonfiguration des TDMA-Verfahren eine sehr geringe Auslastung von nur 26,11 % besitzt, wodurch das Optimierungspotenzial zu erklären ist. Eben-

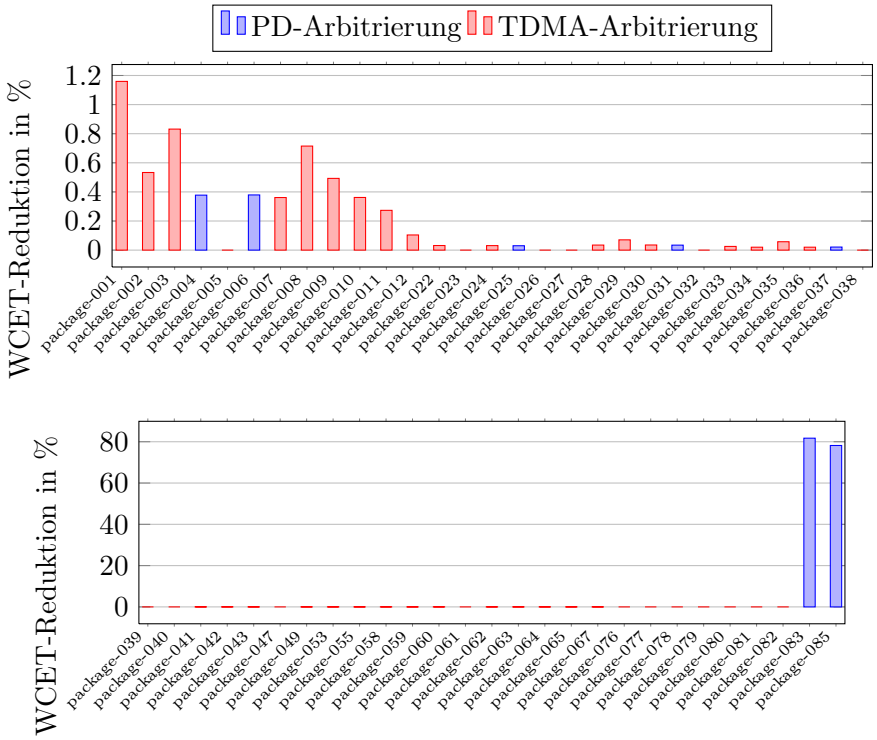


Abbildung 3.10: maximale WCET-Reduktion unter Verwendung des evolutionären Algorithmus (rot: durch TDMA-Konfiguration, blau: durch PD-Konfiguration) gegenüber Standard-TDMA-Konfiguration (ARM7-x8)

Kern	Taskname	Fair-WCET	WCET-optimiert	Verbesserung
C_1	adpcm_decoder	459138	1802320	-292%
C_2	adpcm_encoder	461196	1814720	-293%
C_3	g721_encode	3861600	19083700	-394%
C_4	g723_encode	3861600	19333800	-400%
C_5	v32.modem.bencode	6685050	26176700	-291%
C_6	st	971442	3956080	-300%
C_7	qmf_receive	8366180	38798100	-363%
C_8	fft_1024_7	900951000	43183900	95%
$\sum$		928601844	154149320	83,4%

Tabelle 3.1: WCET-Vergleich unoptimiert und optimiert für Taskmenge *package-083*

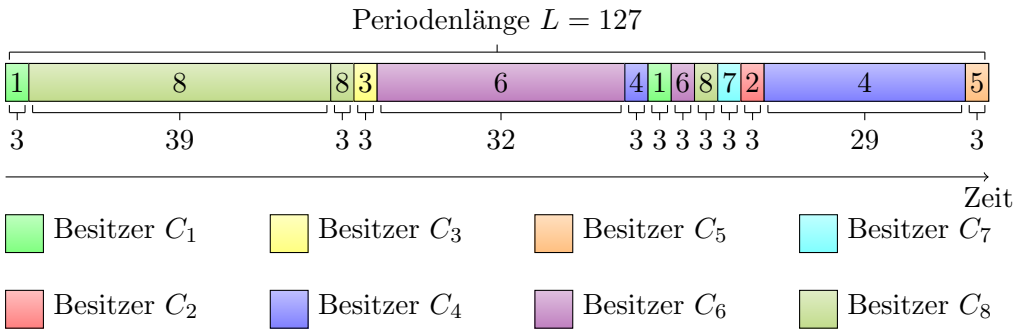


Abbildung 3.11: Konfiguration (PD-Arbitrierung) des besten Individuums von *package-083*, Slotlängen unterhalb der Slots, Slotbesitzer innerhalb des Slots

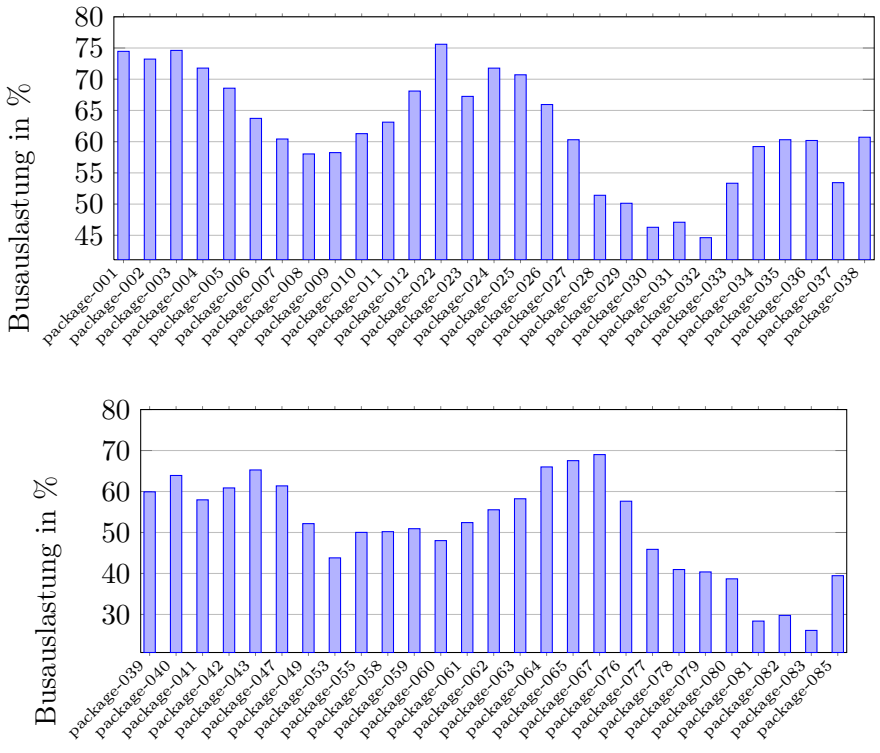


Abbildung 3.12: Busauslastung für Standardkonfiguration der TDMA-Arbitrierung (x8)

falls ist erkennbar, dass viele Taskmengen bereits für die Standardkonfiguration eine relativ hohe Busauslastung besitzen, weshalb die Optimierung mithilfe des evolutionären Algorithmus eventuell keine Verbesserung erzielt.

Es ist auf den Ergebnissen der Benchmarks zu erkennen, dass das Optimierungspotenzial abhängig von den Taskmengen ist. Dies ist durch die unterschiedlichen Zugriffsmuster der Anwendungen zu erklären. Die maximale Reduktionen der WCET für die jeweils verwendeten Programme sind durch die Abbildung 3.13 aufgezeigt. Dort ist zu erkennen, dass das Optimierungspotenzial stark variiert. So weisen die Tasks, die eine *Fourier-Transformation* bestimmen (siehe Tasks *fft_1024*, *fft_1024_7*, usw), alle ein sehr hohes Optimierungspotenzial auf. Dazu sind viele Zugriffe auf das globale Array notwendig, was das Optimierungspotenzial erklärt. Als Gegenbeispiel sei der Task *sqrt* genannt, welcher die Wurzel aus einer Gleitkommazahl mithilfe von *Taylor-Reihen* bestimmt. Für diese Berechnung werden nur lokale Variablen genutzt, sodass keine Zugriffe auf den gemeinsamen Bus notwendig sind (siehe Abbildung 4.25).

Zusammenfassend zeigt die Abbildung 3.14, die Verbesserung der WCET und BCET im Vergleich zur fairen Arbitrierung und zur Standardkonfiguration für das TDMA- und PD-Verfahren. Für die WCET ist zu erkennen, dass diese im Vergleich zur Standardkonfiguration des TDMA-Verfahren nur geringfügig reduziert werden konnte. Allerdings wurde die WCET, im Vergleich zur Standardkonfiguration des PD-Verfahrens und der fairen Arbitrierung, reduziert. Für die BCET ist zu erkennen, dass diese im Vergleich zur Standardkonfiguration des TDMA- und PD-Verfahren verbessert wurde. Die größte Verbesserung wurde dabei für die Hardwarekonfiguration mit acht Prozessorkernen erzielt. Die Ausführungszeiten der Optimierung betrugen, je nach benötigter Zeit für die Analyse der Taskmengen, zwischen 49 Minuten und 33 Stunden. Dabei ist jedoch zu beachten, dass für die hier verwendete Konfiguration von 27 Individuen und mindestens 11 Generationen eine Anzahl von ungefähr 500 Individuen pro Taskmenge evaluiert werden muss. Abhängig von der Hardwarekonfiguration sind dies zwischen 1 000 und 8 000 Analysedurchläufe für die WCET und BCET.

Die durch den evolutionären Algorithmus evaluierten Individuen, für die Taskmenge *package-080* und der Hardwarekonfiguration mit vier Prozessorkernen, sind durch die Abbildung 3.15 und die Abbildung 3.16 visualisiert. Auf der Abbildung 3.15 sind die Optimierungskriterien WCET, ACET und die Busauslastung gegeneinander aufgetragen. Zur verbesserten Visualisierung sind die Achsen logarithmisch skaliert und die Achse der Busauslastung invertiert. Die Farbe der Punkte visualisiert zusätzlich die Höhe (ACET). Blau steht für eine niedrige ACET und Rot für eine hohe ACET. Die Paretofront wird durch die roten und grünen Quadrate dargestellt. Es ist ersichtlich, dass die ACET und die Busauslastung ein korrelierendes Verhalten aufweisen. Mit sinkender Busauslastung steigt die ACET. Außerdem ist zu erkennen, dass die Paretofront sich ausschließlich auf der Kante der höchstmöglichen Busauslastung befindet. Dies ist ein verständliches Verhalten, da eine

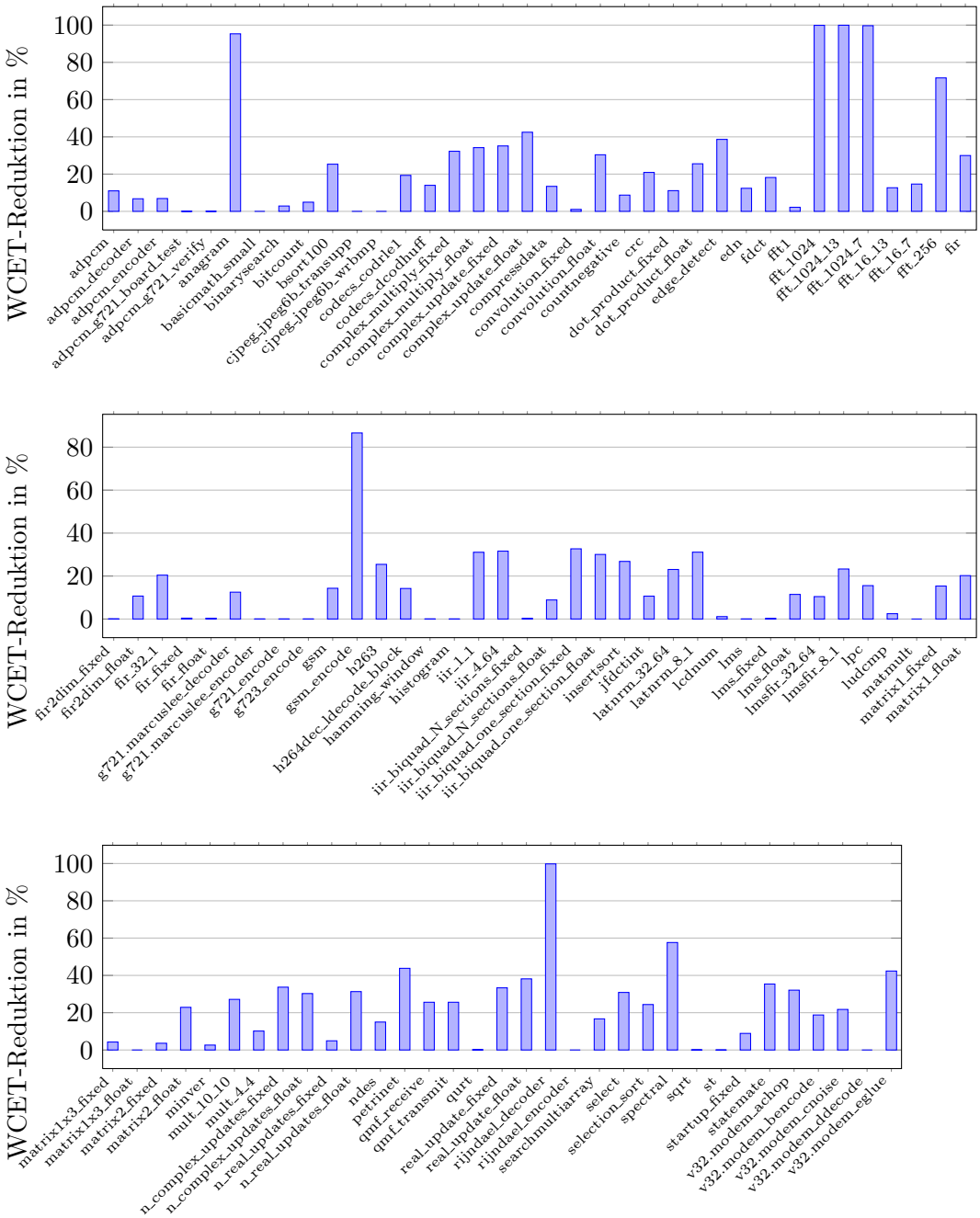
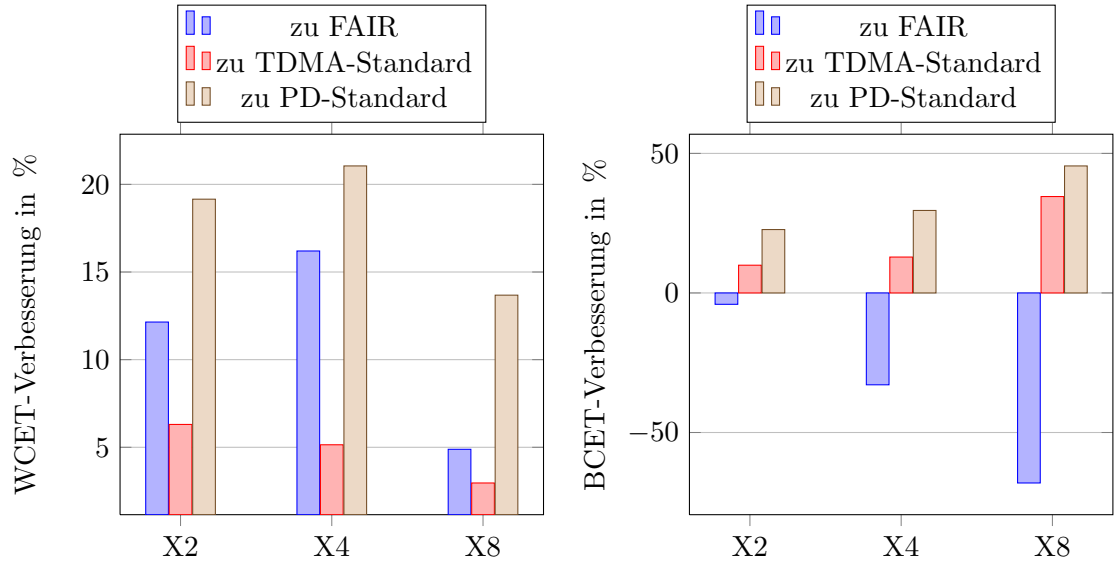


Abbildung 3.13: maximal erreichte WCET-Reduktion für einzelne Tasks im Vergleich über alle Taskmengen zu TDMA-Standardkonfiguration (ARM7-x4)



(a) WCET-Verbesserung (b) BCET-Verbesserung

Abbildung 3.14: durchschnittliche Verbesserung der WCET/BCET durch den evolutionären Algorithmus im Vergleich zur *Fairen Arbitrierung* und der Standardkonfigurationen des *TDMA-Verfahrens* und *PD-Verfahrens*

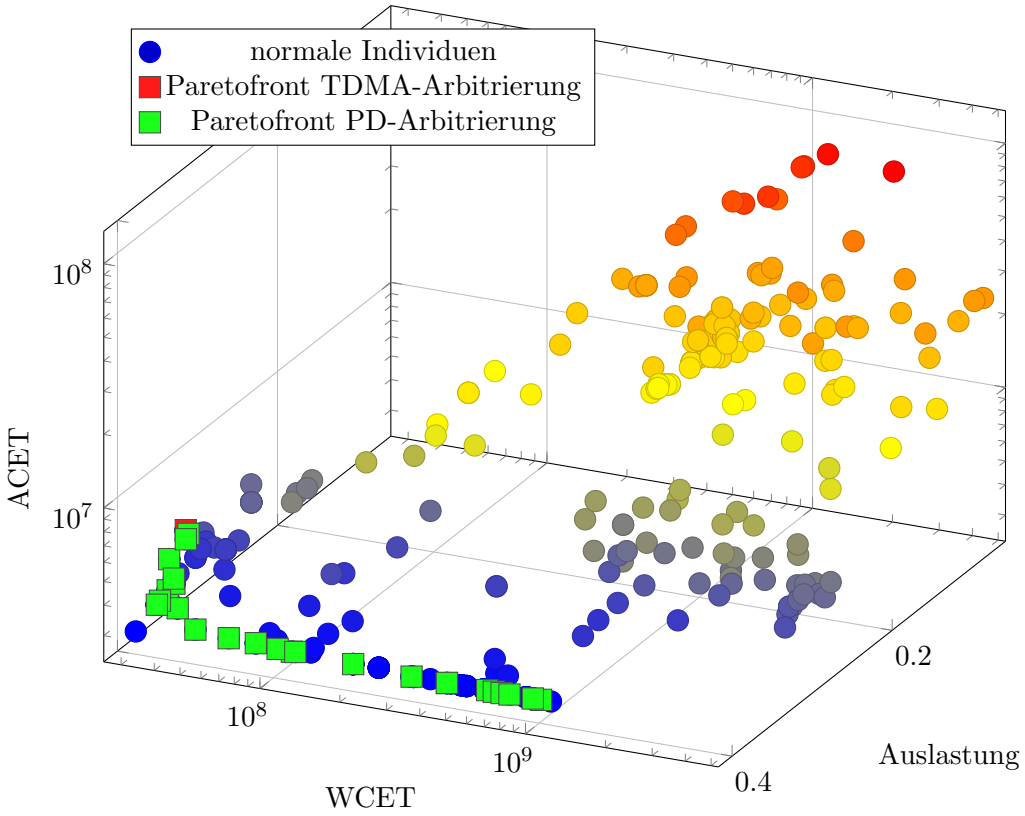


Abbildung 3.15: Individuen von package-080 (x4) mit Auslastung

geringe Busauslastung nicht zu geringen Laufzeiten führen kann, da die Prozessorkerne in diesem Fall Taktzyklen mit Wartezeit verschwenden würden.

Die Abbildung 3.16 zeigt die Optimierungskriterien WCET, BCET und ACET gegeneinander aufgetragen. Ebenfalls wie für die Abbildung 3.15 hervorgehoben ist die Paretofront. Zu erkennen ist in der Abbildung, dass sowohl WCET, BCET als auch ACET eine schwache Korrelation miteinander aufweisen. In den meisten Fällen bedeutet eine geringere WCET auch eine geringere BCET und ACET. Allerdings gibt es einzelne Fälle, bei denen dies nicht der Fall ist. Dies erklärt die Paretodominanz der Individuen, welche in der Mitte der Abbildung liegen. Diese dominieren die Individuen, welche links in der Abbildung liegen, durch eine geringere ACET. Das heißt, dass mit steigender WCET sowohl die BCET als auch die ACET steigt. Die Paretofront verläuft an dieser steigenden Front entlang. Erkennbar ist, dass entlang der Paretofront auf der Abbildung 3.15 und der Abbildung 3.16 viele Individuen evaluiert wurden, was sich durch den SPEA2-Selektor begründen lässt (siehe Abschnitt 3.3), welcher ebenfalls auf Basis der Paretodominanz agiert.

Die Evaluation des in diesem Kapitel vorgestellten evolutionären Algorithmus hat gezeigt, dass dieser in der Regel immer zu einer Reduktion der Worst-Case Laufzeit führt. So wurde durch die Abbildungen 3.2, 3.5 und 3.8 gezeigt, dass die *faire Arbitrierung* für den gemeinsamen Bus fast ausschließlich schlechtere Ergebnisse als die TDMA- und PD-Arbitrierung

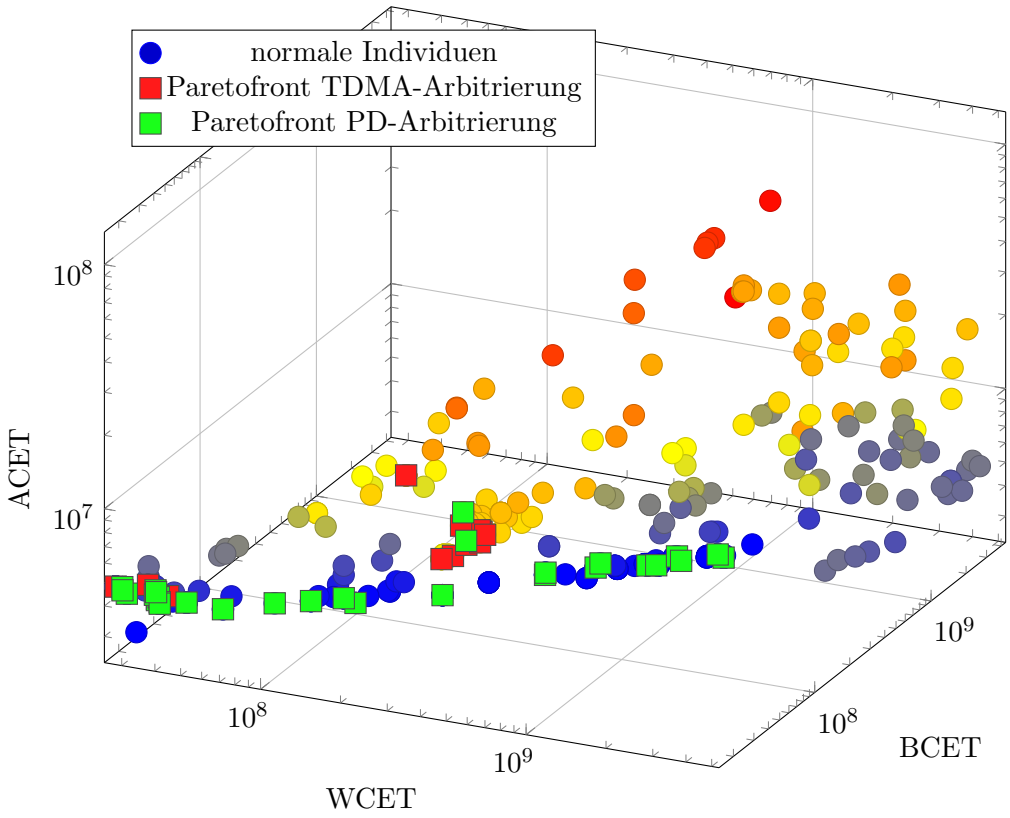


Abbildung 3.16: Individuen von package-080 (x4) mit BCET

liefert. Dieses Verhalten kann zusätzlich an der Analyse der WCET und BCET liegen, welche, wie in Abschnitt 2.4 vorgestellt, generell für das TDMA- und PD-Verfahren genauere Ergebnisse liefert, da für die faire Arbitrierung eine starke Überabschätzung durchgeführt wird.
Außerdem konnte gezeigt werden, dass die Standardkonfiguration des WCC-Übersetzers für das TDMA- und PD-Verfahren, welche jedem Prozessorkern einen gleich verteilten Anteil Zugriffszeit für den gemeinsamen Bus zuteilen, noch Potenzial für Verbesserungen bieten. Dieses Potenzial ist jedoch anwendungsspezifisch und kann daher nicht allgemein formuliert werden, weshalb jede Taskmenge speziell durch die vorgestellte Optimierung behandelt werden muss.

4 Optimierung durch Instruction-Scheduling

Durch dieses Kapitel wird der zweite Teil der Arbeit vorgestellt, bei dem es sich um die Optimierung mithilfe eines Instruction-Schedulers handelt. Dieser Instruction-Scheduler soll durch eine Steigerung der Busauslastung mittels der Neuanordnung von Maschineninstruktionen eine Reduktion der WCET erreichen. Dazu werden zunächst in Abschnitt 4.1 die Grundlagen und in Abschnitt 4.2 die Arbeitsweise eines *Instruction-Schedulers* vorgestellt. Im weiteren Verlauf wird erläutert, welche verschiedenen Möglichkeiten zur Neuanordnung existieren. Dafür wird in Unterabschnitt 4.4.1 das *Basisblock-Scheduling*, in Unterabschnitt 4.4.2 das *Trace-Scheduling* und in Unterabschnitt 4.4.3 ein *Superblock-Scheduling* vorgestellt werden. Anschließend werden in Abschnitt 4.5 Details zur Realisierung innerhalb des WCC-Übersetzers geliefert und in Abschnitt 4.6 eine Evaluation der Optimierung präsentiert.

4.1 Datenabhängigkeiten

Für das Neuanordnen von Maschineninstruktionen gelten Begrenzungen, da zwischen den einzelnen Maschineninstruktionen Abhängigkeiten bestehen können, welche nicht verletzt werden dürfen, da dies eine Veränderung der Programmsemantik zur Folge hätte. Solche Abhängigkeiten entstehen durch den Zugriff auf dieselben Maschinenregister oder Speicherbereiche von verschiedenen Instruktionen. Diese Form der Abhängigkeiten wird als Datenabhängigkeit benannt [ALSU06].
Es gibt verschiedene Formen der Datenabhängigkeiten, welche im Folgenden vorgestellt werden. Die erste Datenabhängigkeit wird *echte Abhängigkeit* genannt. Diese tritt auf, wenn ein Wert geschrieben wird und dieser Wert anschließend gelesen wird (englisch: read-after-write). Das folgende Beispiel zeigt eine solche Abhängigkeit.

```
1 ADD R1, R2, R3
2 ADD R4, R1, R2
```

Die Instruktion in Zeile 1 berechnet die Addition aus den Registern `R2` und `R3` und speichert das Ergebnis in das Register `R1`. Die nächste Instruktion in Zeile 2 berechnet die Addition mit den Werten aus den Registern `R1` und `R2`, was einer Datenabhängigkeit entspricht, da vor der Ausführung der zweiten Instruktion, das Ergebnis der ersten Instruktion in das Register `R1` geschrieben werden muss.

Eine weitere Form der Datenabhängigkeit ist die *Antidatenabhängigkeit* (englisch: antidependence). Diese tritt auf, wenn ein Wert geschrieben wird, der in einer vorherigen Instruktion gelesen wird (englisch: write-after-read). Eine solche Abhängigkeit wird durch das nachfolgende Codebeispiel skizziert.

```
1 ADD R1, R2, R3
2 ADD R2, R3, R4
```

Die zweite Instruktion beschreibt das Register `R2`, welches von der ersten Instruktion für die Addition gelesen wird, weshalb eine Antidatenabhängigkeit besteht, da die zweite Instruktion in der Form nicht vor der ersten Instruktion ausgeführt werden darf. Diese Form der Abhängigkeit lässt sich in vielen Fällen vermeiden, indem für das Speichern von Ergebnissen andere Register beziehungsweise Speicherbereiche verwendet werden. So ist es möglich das Ergebnis der zweiten Instruktion im obigen Beispiel in das Register `R5` zu speichern, was eine Auflösung der Antidatenabhängigkeit zur Folge hätte.
Zusätzlich gibt es noch die *Ausgabeabhängigkeit*, welche beschreibt, dass ein Wert mehrfach hintereinander beschrieben wird (englisch: write-after-write), welche durch das nachfolgende Beispiel beschreiben wird.

```
1 ADD R1, R2, R3
2 ADD R5, R1, R4
3 LDR R1, A
```

Zwischen der Instruktion aus Zeile 1 und der aus Zeile 3 besteht eine Antidatenabhängigkeit, da beide Instruktionen das Register `R1` beschreiben. Genau wie für die Antidatenabhängigkeit lässt sich diese Abhängigkeit durch das verwenden anderer Register oder Variablen vermeiden. Aus diesem Grund wird nur die *read-after-write* Abhängigkeit als *echte Abhängigkeit* bezeichnet.
Eine andere Form der Abhängigkeit von Instruktionen ist die *Kontrollabhängigkeit*. Diese Abhängigkeit beschreibt, dass die Reihenfolge der Instruktionen den Kontrollfluss des Programms beeinflussen. Im Folgenden wird ein Beispiel aufgezeigt, welches die Kontrollabhängigkeit beschreibt.

```
1 start:    CMP R1, R2
2           BEQ addition
3           SUB R1, R1, R2
4           B ende
5 addition: ADD R1, R1, R2
6           B ende
7 ende:
```

Dieses Beispiel beschreibt eine einfache Verzweigung. Sind die Werte der Register `R1` und `R2` identisch, so bedeutet das für die Instruktion `BEQ addition`, dass ein Sprung an die angegebene Marke durchgeführt wird. Ist dies nicht der Fall, so wird kein Sprung ausgeführt und die Subtraktion durchgeführt. Die Instruktionen `ADD R1, R1, R2` und `SUB R1, R1, R2` sind kontrollabhängig gegenüber der Instruktion `BEQ addition`, da diese bestimmt,

welcher Kontrollfluss gewählt wird. Zusätzlich zu dieser Abhängigkeit ist die Instruktion `BEQ addition` abhängig von der Instruktion `CMP R1, R2`, da der Vergleich bestimmt, ob ein Sprung durchgeführt wird [ARM05].
Die ARM-Architektur arbeitet mit *bedingter Ausführung*, das heißt, dass ein Großteil der Maschineninstruktionen eine Bedingung enthalten können, welche angibt, ob die Instruktionen ausgeführt werden sollen. Ist die Bedingung erfüllt, so wird der Befehl normal ausgeführt. Ist die Bedingung nicht erfüllt, so wird anstelle des Befehls ein `NOP` (kurz für: No-Operation) ausgeführt. Beispielsweise könnte eine Befehlsfolge wie folgt aussehen.

```
1 CMP   R1, R2
2 CMPNE R2, R3
3 ADDGE R1, R1, R2
4 SUBLE R1, R2, R3
5 STR   R1, [R9, #4]
```

Soll eine Instruktion abhängig von einer Bedingung ausgeführt werden, wird dies mit einem Suffix gekennzeichnet. Die Suffixe `NE`, `GE` und `LE` beschreiben die Bedingungen *ungleich*, *größer gleich* und *kleiner gleich*. Dies sind Beispiele der Bedingungen, die für den ARM-Prozessor erlaubt sind [ARM05]. Das Fehlen eines Suffixes bedeutet, dass die Instruktion unabhängig einer Bedingung ausgeführt wird. Abhängig von den Daten der Register sind im obigen Programmbeispiel verschiedene Kontrollflussmöglichkeiten vorhanden, da die Instruktionen nur für die jeweiligen Bedingungen ausgeführt werden. Die für die Bedingungen relevanten Felder im Prozessor werden durch sogenannte *Daten-Instruktionen* (englisch: data-processing instructions) modifiziert, weshalb diese speziell behandelt werden müssen. Da es für den Instruction-Scheduler zum Zeitpunkt des Schedulings nicht ersichtlich ist, wie die Ergebnisse der Instruktionen sind, wird zwischen Daten-Instruktionen und Instruktionen, welche eine Bedingung zur Ausführung besitzen, immer von einer Kontrollabhängigkeit ausgegangen, sodass die Semantik der optimierten Programme keinesfalls verändert wird.

4.1.1 Abhängigkeitsgraph

Zur Darstellung von Abhängigkeiten zwischen Instruktionen wird ein sogenannter *Abhängigkeitsgraph* erstellt [ALSU06].

4.1.1 Definition (Abhängigkeitsgraph). Der Abhängigkeitsgraph $G = (N, E)$ ist ein gerichteter, azyklischer Graph, dessen Knoten $n \in N$ die Instruktionen einer Instruktionsfolge und dessen Kanten $e \in E$ die Abhängigkeiten zwischen den Instruktionen beschreiben.

Im weiteren Verlauf wird beschrieben, wie der Abhängigkeitsgraph aufgebaut wird, dazu wird auf die Beschreibung aus der Arbeit von A. Smolarczyk [Smo10] zurückgegriffen. Existiert eine Instruktionsfolge $I := (i_1, \ldots, i_n)$, so wird über jede Instruktion iteriert und

dabei ein Knoten für die Instruktion i_j in den Graphen eingefügt. Im nächsten Schritt wird für die Instruktion i_j überprüft, welche Abhängigkeiten bestehen, wozu über die Instruktionen $i_i \in (i_1, \ldots, i_{j-1})$ iteriert wird und für jedes Instruktionspaar (i_j, i_i) überprüft, ob eine Abhängigkeit besteht. Zur Überprüfung auf Abhängigkeiten werden die Datenfelder der Instruktionen verglichen, sodass Abhängigkeiten beispielsweise bei Registerzugriffen erkannt werden können. Für Datenfelder, welche eine indirekte Adressierung nutzen ist es nicht in allen Fällen möglich, sicher zu bestimmen, ob eine Datenabhängigkeit besteht, weshalb eine Abhängigkeit in den Graph eingefügt wird, damit die Programmsemantik nicht gefährdet wird.

4.2 Listen-Scheduler

Mithilfe des Abhängigkeitsgraphen ist es möglich das *Instruction-Scheduling* durchzuführen. Dazu wird eine abgeänderte Variante des *List Scheduling*-Verfahren, wie es in der Arbeit von A. Smolarczyk [Smo10] vorgestellt wird, verwendet. Die Realisierung ist in Pseudocode durch Algorithmus 4.1 angegeben.

Eingabe: Abhängigkeitsgraph DG
Ausgabe: Instruktionsfolge schedule

```
 1: zyklus = 1;
 2: ReadyList = Berechne ReadyList (DG);
 3: schedule = {};
 4: bestInstruction = 0;
 5: while ReadyList ≠ ∅ do
 6:     maxPrio = 0;
 7:     for all i ∈ ReadyList do
 8:         prio = Bestimme Priorität (i);
 9:         if maxPrio < prio then
10:             bestInstruction = i;
11:             maxPrio = prio;
12:         end if
13:     end for
14:     schedule = schedule + bestInstruction;
15:     zyklus = zyklus + 1;
16:     ReadyList = Berechne ReadyList (DG);
17: end while
18: return schedule;
```

Algorithmus 4.1: List Scheduler der Optimierung

Als Eingabe erhält der Algorithmus den Abhängigkeitsgraphen. Zu Beginn des Algorithmus wird die Liste der Instruktionen bestimmt, welche bereit für das *Scheduling* sind. Die Liste beschreibt alle Instruktionen, die im aktuellen Zyklus durch den Scheduler ausgeführt werden dürfen. Für jede Instruktion aus dieser Liste wird eine Priorität bestimmt (Zeile 8).

Diese Priorität wird mithilfe einer Heuristik bestimmt, welche in Abschnitt 4.3 vorgestellt wird. Dafür werden drei verschiedene Heuristiken verwendet, welche auf einer unterschiedlichen Datenbasis ihre Entscheidung treffen. Zwei Heuristiken verwenden als Datenbasis die Parameter, welche den gemeinsamen Bus konfigurieren (siehe Unterabschnitt 4.3.1) und eine Heuristik verwendet die *TDMA-Offsets* (siehe Unterabschnitt 4.3.2), wie sie in Abschnitt 2.4 vorgestellt wurden. Nachdem für jede Instruktion die Priorität bestimmt wurde, wird anschließend die Instruktion mit der höchsten Priorität ausgewählt und durch den Scheduler zur Ausführung geplant (Zeile 14). Nach dem Planen einer Instruktion wird die Liste der bereiten Instruktionen neu bestimmt. Ist jede Instruktion aus der gegebenen Instruktionsfolge durch den Scheduler verplant worden, so terminiert der Algorithmus und gibt die neue Instruktionsfolge *schedule* zurück, was der geordnete Liste der Instruktionen entspricht, wie sie durch den Scheduler bestimmt wurde.

4.3 Instruktions-Selektions Heuristiken

Ein Ziel der Arbeit ist es, die WCET durch Neuanordnen der Maschineninstruktionen zu reduzieren (siehe Abschnitt 1.1). Dazu soll speziell die Buskonfiguration einbezogen werden. Für dieses Ziel wurden verschiedene Heuristiken zur Auswahl von Instruktionen entworfen und in Abschnitt 4.6 evaluiert. Dazu nutzen die Heuristiken die durch die *WCET-Analyse* (siehe Abschnitt 2.4) bestimmten Daten über die Laufzeit der verschiedenen Instruktionen.

4.3.1 Slotlängen-basierte Heuristik

Eine Möglichkeit zur Bestimmung der Prioritäten von Instruktionen ist die Busparameter der Busarbitrierung auszunutzen. Im Falle des *TDMA-* und *PD-Verfahren* (siehe Unterabschnitt 2.2.3 und Unterabschnitt 2.2.4) kann die maximale Slotlänge eines Prozessorkerns als Maßstab für die Größe möglicher Instruktionsbündel verwendet werden.
Die Heuristik versucht, Instruktionen zu bündeln, bis die Slotlänge des aktuellen Prozessorkerns mit Instruktionen aufgefüllt ist. Dazu wird jeweils die Instruktion benötigt, welche vor der aktuellen Instruktion vom Scheduler gewählt wurde, damit erkennbar ist, ob aktuell eine Folge von Recheninstruktionen oder von Instruktionen mit Buszugriff geplant wird. War diese Instruktion eine Instruktion, welche einen Zugriff auf den gemeinsamen Bus benötigt hat, so wird versucht wieder eine Instruktion mit Buszugriff zu wählen bis die maximale Slotlänge des Prozessorkerns aufgefüllt ist. Ist eine solche Instruktion bereit zur Ausführung, so erhält diese eine höhere Priorität als die Instruktionen ohne Buszugriff. Dies bewirkt, dass der *List-Scheduler* diese Instruktion bevorzugt. Wurden mehrere Instruktionen, die einen Buszugriff benötigen, in Folge durch den Scheduler ausgewählt, so wird überprüft, ob die Slotlänge des Prozessorkerns überschritten wurde. Falls dies der Fall

ist, wird versucht eine Instruktion ohne Buszugriff zu bevorzugen, damit der Prozessorkern nicht unnötig auf einen Zugriff auf den gemeinsamen Bus warten muss.

Es wurde während der Implementierung die Beobachtung gemacht, dass es einen Unterschied macht, ob vor dem Wählen der ersten Instruktion, die einen Buszugriff voraussetzt, bereits überprüft wird, ob diese in den Slot des Prozessorkerns passt. Dies liegt daran, dass die Ausführungsdauer einer Instruktion, welche durch die WCET-Analyse geliefert wird, bereits die Wartezeit für den Buszugriff beinhaltet. Das heißt, dass eine Instruktion, die ihren Slot genau verpasst hat, bei einer TDMA-Konfiguration mit 4 Slots mit je 3 Taktzyklen Länge eine Wartezeit von 9 Taktzyklen auf addiert bekommt. Daher wird die Heuristik in zwei Varianten bei der Evaluation erforscht. Die eine Variante bevorzugt jede erste Instruktion mit Buszugriff für eine Folge, unabhängig von der Slotlänge und Ausführungsdauer der Instruktion. Diese Heuristik wird im weiteren Verlauf als *Always-First Heuristik* bezeichnet. Die andere Variante überprüft für jede Instruktion, ob diese in den Slot passt, indem die Slotlänge des Prozessorkerns mit der Ausführungsdauer der Instruktion verglichen wird. Diese Heuristik wird fortan als *Only-Fitting Heuristik* bezeichnet. Die unterschiedlichen Ergebnisse der Heuristik werden in Abschnitt 4.6 vorgestellt.

Eine Problematik dieser Heuristiken ist, dass die Annahme gemacht wird, dass zum Zeitpunkt der ersten gewählten Instruktion, welche einen Buszugriff benötigt, der gemeinsame Bus für den Prozessorkern zur Verfügung steht. Dies kann nicht garantiert werden, sodass die Heuristik nicht in allen Fällen gute Ergebnisse liefern kann. Die Abbildung 4.1 skizziert die Problematik für ein System mit zwei Prozessorkernen und zwei TDMA-Slots. Das Optimum wird im obersten Teil der Grafik dargestellt. Eine Folge von Instruktionen, die einen

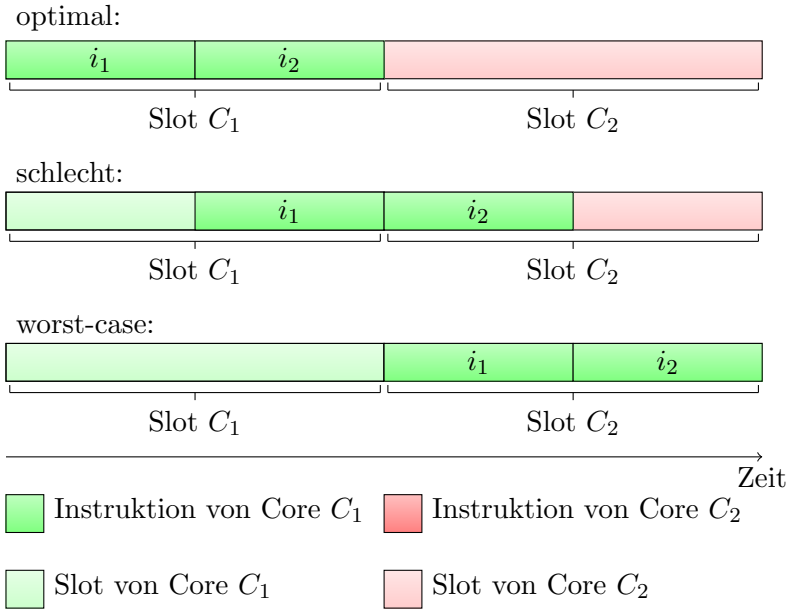


Abbildung 4.1: Problematik der Busparameter-basierten Heuristik

Buszugriff benötigen, wird passend zum aktuellen TDMA-Schedule gewählt, was zu einer optimalen Ausnutzung des TDMA-Slots führt. Ein schlechterer Fall wird in der Mitte der Abbildung 4.1 dargestellt, für den die Hälfte der Zugriffszeit des Slots des Prozessorkerns C_1 schon abgelaufen ist, sodass die Instruktion i_2 in den Slot des Prozessorkerns C_2 geplant werden würde. Dies würde zu einer Wartezeit von einer Slotlänge für den Prozessorkern C_1 führen. Der schlechteste Fall wird im unteren Teil der Abbildung skizziert. Der Slot des Prozessorkerns ist in diesem Fall zum Zeitpunkt der geplanten Instruktion abgelaufen, sodass die Instruktionen i_1 und i_2 in den Slot des Prozessorkerns C_2 geplant werden, was zur Folge hätte, dass beide Instruktionen frühstens im nächsten TDMA-Zyklus ausgeführt werden. Dieses Problem versucht die *TDMA-Offset-basierte Heuristik* zu beheben, welche im folgenden Abschnitt vorgestellt wird.

4.3.2 TDMA-Offset-basierte Heuristik

Die TDMA-Offset-basierte Heuristik versucht, zu vermeiden, dass Instruktionen verplant werden, welche auf den Bus zugreifen, wenn der Prozessorkern aktuell keinen Zugriff auf den Bus zugeteilt bekommt. Dafür wird auf die in Abschnitt 2.4 vorgestellten *TDMA-Offsets* zurückgegriffen, die angeben, welcher Zeitpunkt innerhalb der *TDMA-Periode* gerade aktiv ist. Diese Werte sind keine absolute Zeitangabe, sondern beschreiben einen Bereich in dem der aktuelle Zeitpunkt liegen kann. Die Heuristik betrachtet den TDMA-Offset, der zu Beginn eines Basisblockes gilt und betrachtet die Ausführungsdauer von den Instruktionen, die durch die Heuristik ausgewählt werden, damit dies bei der Auswertung der TDMA-Offsets berücksichtigt werden kann. Wird die Periodenlänge durch die Ausführungsdauer der bereits geplanten Instruktionen überschritten, so wird durch eine Modulo-Rechnung der Wert wieder auf 0 zurückgesetzt, was die periodische Arbeitsweise des TDMA- und PD-Verfahrens widerspiegelt.

Zur Bestimmung der Priorität einer Instruktion wird auf die Mikroarchitekturanalyse zurückgegriffen (siehe Abschnitt 2.4). Es wird für jeden Basisblock der Zustand der Mikroarchitektur, der zu Beginn des Basisblockes aktuell ist, abgespeichert und für die weitere Zuteilung der Priorität verwendet. Dieser Zustand beinhaltet den Wert des TDMA-Offsets, welcher angibt, welche Position des Busschedules gerade aktiv ist. Dies ist ein Zeitbereich und kein absoluter Zeitwert. Liegt dieser TDMA-Offset innerhalb eines Slots, der dem aktuellen Prozessorkern zugeordnet ist, so wird eine Instruktion mit Buszugriff bevorzugt. Liegt der Offset innerhalb eines Slots, der einem anderem Prozessorkern zugeordnet ist, so wird eine Instruktion bevorzugt, welche keinen Zugriff auf den Bus benötigt. Für jede Instruktion die ausgewählt wird, wird eine Mikroarchitekturanalyse durchgeführt, wodurch der Zustand der Mikroarchitektur für den Basisblock, in dem die Instruktion liegt, verändert wird. Die Zustandsänderung beinhaltet auch die Verschiebung des TDMA-Offsets,

sodass dieser Wert stets aktuell bleibt und die bereits verplanten Instruktionen berücksichtigt. Durch

$$Priorität(O_i, B_i) = \begin{cases} 3 & \text{wenn } O_i \subseteq \omega(s_c) \wedge B_i = 1, \\ 2 & \text{wenn } B_i = 0, \\ 1 & \text{ansonsten} \end{cases} \tag{4.1}$$

wird die Bestimmung der Priorität formalisiert.
Die Priorität einer Instruktion kann drei mögliche Werte annehmen. Die Priorität einer Instruktion i, wird mithilfe des TDMA-Offsets O_i bestimmt, welcher die Position innerhalb des Busschedules angibt, der zum Zeitpunkt vor der Instruktion i gilt. Die Variable B_i gibt an, ob durch die Instruktion ein Buszugriff durchgeführt wird. Die Menge $\omega(s_c)$ enthält die Zeitwerte, zu denen es für den Prozessorkern c erlaubt ist, eine Instruktion mit Buszugriff auszuführen. Bei diesen Werten ist berücksichtigt, dass Buszugriffe nicht unterbrochen werden dürfen und eine maximale Laufzeit von m_{max} benötigen.
Benötigt eine Instruktion den gemeinsamen Bus und der Prozessorkern c kann auf den Bus zugreifen, so erhält die Instruktion die höchstmögliche Priorität. Benötigt eine Instruktion keinen Zugriff auf den Bus, so wird dieser Instruktion eine Priorität von 2 zugewiesen. Alle anderen Instruktionen bekommen die minimale Priorität von 1 zugewiesen. Instruktionen die den Bus benötigen, aber aktuell nicht zugreifen dürfen, zählen auch zu der niedrigsten Priorität, wodurch sichergestellt ist, dass in diesem Fall Instruktionen bevorzugt werden, die keinen Buszugriff benötigen, was für eine bessere Prozessorauslastung sorgt.
Durch die Abbildung 4.2 ist der Fall, dass ein TDMA-Offset innerhalb eines Slots des Prozessorkerns liegt, skizziert. Es ist erneut ein Modell mit zwei Prozessorkernen und einer TDMA-Arbitrierung modelliert. Der gelbe Bereich symbolisiert den Bereich des TDMA-Offsets, welcher innerhalb eines Slots liegt, der dem aktuellem Prozessorkern (C_1) gehört,

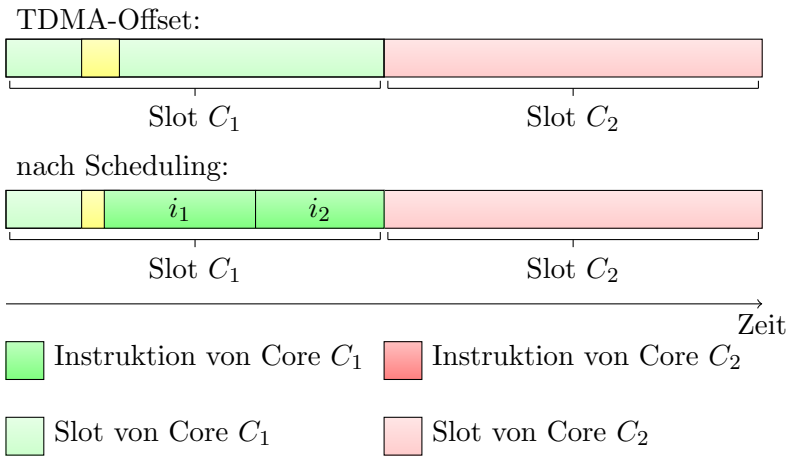


Abbildung 4.2: TDMA-Offset Werte, Scheduling von Bus-Instruktionen

weshalb ein Scheduling von Instruktionen mit Buszugriff möglich ist. Die Instruktionen i_1 und i_2 sind in diesem Fall Instruktionen, welche einen Zugriff auf den gemeinsamen Bus voraussetzen und werden daher geplant.
Liegt der aktuelle TDMA-Offset Wert nicht in einem Slot des Prozessorkerns, so werden, nach Gleichung 4.1, Instruktionen bevorzugt, welche keinen Zugriff auf den gemeinsamen Bus benötigen. Dieser Fall wird durch die Abbildung 4.3 skizziert. Der aktuelle TDMA-Offset, dargestellt als gelbe Fläche, liegt in dem Bereich von Prozessorkern C_2, was bedeutet, dass ein Buszugriff für den Prozessorkern C_1 aktuell nicht möglich ist. Die Heuristik priorisiert in diesem Fall die Instruktionen $i_1^{'}$ und $i_2^{'}$ hoch, welche Instruktionen ohne Buszugriff sind, sodass der Prozessorkern stets gut ausgelastet wird.

4.4 Scheduling-Verfahren

Es gibt verschiedene Möglichkeiten, wie das Instruction-Scheduling durchgeführt werden kann. So gibt es das *lokale Scheduling*, bei dem der List-Scheduler nur die Instruktionen eines Basisblockes betrachtet. Des Weiteren gibt es die *globalen Scheduling-Verfahren*, wie das *Trace-Scheduling* und das *Superblock-Scheduling*, bei denen der Instruction-Scheduler Instruktionen über die Grenzen von Basisblöcken verschieben kann.

4.4.1 Lokales Scheduling

Das lokale Scheduling ist die einfachste Methodik ein Instruction-Scheduling durchzuführen. Es werden jeweils nur Basisblöcke durch den Instruction-Scheduler betrachtet. Für das lokale Scheduling wird der Abhängigkeitsgraph, wie in Unterabschnitt 4.1.1, beschrieben erstellt. Ein Basisblock ist eine Sequenz von Maschineninstruktionen, welche, außer zu Beginn und am Ende, keine Verzweigungen enthalten darf. Der Instruction-Scheduler wird

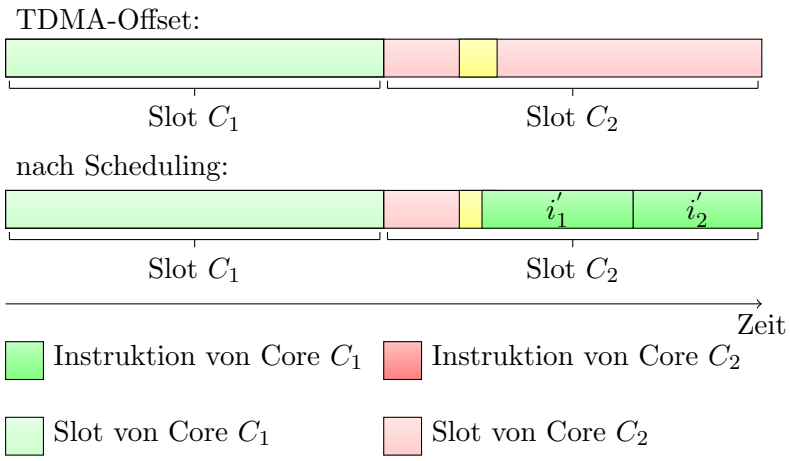


Abbildung 4.3: TDMA-Offset Werte, Scheduling von Recheninstruktionen

in einem nächsten Schritt für alle Basisblöcke aufgerufen, sodass jeder Basisblock einmal durch den *List-Scheduler* optimiert wird. Für das lokale Scheduling ist es nicht erforderlich Kontrollabhängigkeiten zu beachten, sofern gewährleistet wird, dass eine Sprunginstruktion, welche nur am Ende eines Basisblockes vorkommen kann, nicht verschoben wird. Zusätzlich muss für die ARM-Architektur der Spezialfall von bedingten Instruktionen, welche in Abschnitt 4.1 vorgestellt wurden, betrachtet werden.

Es kann für ARM-Assemblerprogramme vorkommen, dass ein Basisblock keine Recheninstruktionen enthält, sondern lediglich einen Datenbereich enthält, der beispielsweise auf globale Datenbereiche zeigt. Ein solcher Basisblock darf nicht modifiziert werden, da dies die Datenstrukturen eines Programms verändern würde, was die Semantik einer Anwendung verändern würde.

Das lokale Scheduling-Verfahren ist nicht sehr effektiv, da die Basisblöcke im Durchschnitt nur fünf Instruktionen enthalten, was dem List-Scheduler nur wenige Möglichkeiten zur Anordnung der Instruktionen bietet [ALSU06]. Zusätzlich betrachtet das lokale Scheduling keinerlei Informationen über den WCEP, sodass auch Basisblöcke optimiert werden, welche keinen direkten Einfluss auf die WCET haben, wodurch das Verfahren weiter an Effizienz verliert.

Die globalen Schedulingverfahren versuchen durch verbesserte Datenstrukturen ein Scheduling über die Grenzen eines Basisblockes zu ermöglichen. Dafür gibt es verschiedene Möglichkeiten, welche im Folgenden vorgestellt werden.

4.4.2 Trace-basiertes Scheduling

Alternativ zum Scheduling von Basisblöcken ist es möglich, ein Scheduling durchzuführen, welche Instruktionen über Basisblockgrenzen betrachtet und verschiebt. Eine Möglichkeit dazu ist durch die Verwendung von *Traces* (Programmpfaden) gegeben, welche einen möglichen Ablauf eines Programms im Kontrollflussgraphen angeben [Fis81]. Die Idee ist, dass der List-Scheduler wie für das lokale Scheduling, für eine Instruktionsfolge, welche nicht auf die Instruktionen eines Basisblockes beschränkt ist, verwendet werden kann. Dazu müssen jedoch einige Besonderheiten beachtet werden, welche in diesem Abschnitt erläutert werden. Für die Beschreibung des Trace-basierten Schedulings wird auf die Beschreibung aus der Arbeit von A. Smolarczyk [Smo10] zurückgegriffen, welche als Basis für das in dieser Arbeit vorgestellte Instruction-Scheduling diente.

Zur formalen Definition eines Traces wird zunächst eine Hilfsfunktion definiert, welche für die weitere Definition notwendig ist.

4.4.1 Definition (Followers). Die Funktion $Followers(b_i)$ gibt für einen Kontrollflussgraphen $G = (N, E)$ die Menge der Basisblöcke b_j an, für die gilt, dass eine Kante $(b_i, b_j) \in E$ zwischen $b_i \in N$ und $b_j \in N$ existiert.

Mithilfe dieser Funktion kann angegeben werden, zu welchen Basisblöcken der Kontrollfluss ausgehend von b_i wechseln kann. Der Trace lässt sich mit der Funktion wie folgt definieren [Smo10].

4.4.2 Definition (Trace). Ein **Trace T** ist eine Folge von Basisblöcken $T_i = (bb_1, \ldots, bb_t)$, sodass $\forall j : 1 \leq j \leq t-1$ die Bedingungen

- $bb_{j+1} \in Followers(bb_j)$,
- $\nexists bb_k \in T_i : bb_j = bb_k$ und
- $bb_j \in T_i \Rightarrow bb_j \notin T_l, i \neq l$

gelten.

Mit dieser Definition ist sichergestellt, dass ein Basisblock immer ein Element der Menge der Funktion *Followers* seines Vorgängers ist, dass kein Basisblock mehrfach innerhalb eines Traces liegt und dass jeder Basisblock nur Teil eines Traces sein kann. Ein beispielhafter Trace ist durch die Abbildung 4.4 abgebildet. Die Basisblöcke sind auf dieser Abbildung durch die roten Bereiche eingegrenzt und der Trace ist der grüne Bereich, welcher die Basisblöcke umschließt. Für die Skizze gilt, dass $Followers(A) = \{B, E\}$, da für den Kontrollfluss aus dem Basisblock A ein Wechsel zu B und E möglich ist. Der abgebildete Trace hat einen *Seitenausgang* für den Basisblock A und einen *Seiteneingang* auf den Basisblock C.

Trace-Selektion

Für das Trace-Scheduling ist es erforderlich, dass ein Trace für ein Programm ausgewählt wird, welcher durch den Instruction-Scheduler optimiert werden soll, wofür die *Trace-Selektion* verantwortlich ist. Da der Instruction-Scheduler, im Fall dieser Arbeit speziell

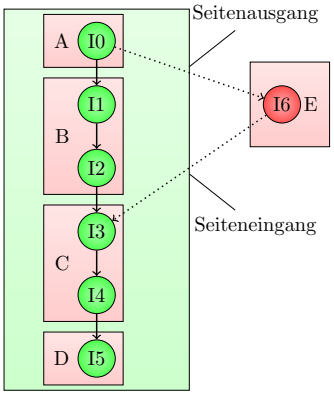
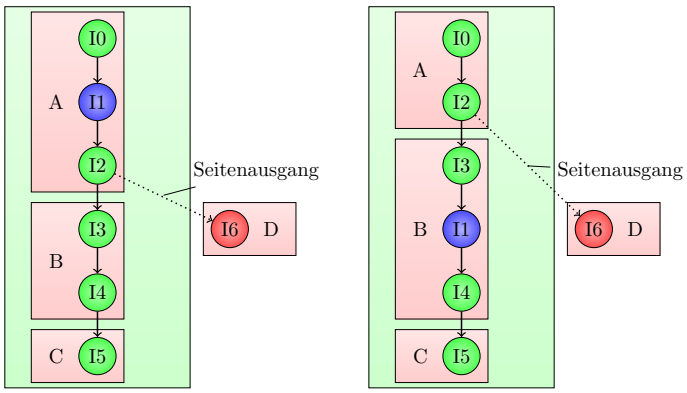


Abbildung 4.4: schematische Darstellung eines Traces

die WCET reduzieren soll, sollte der ausgewählte Pfad für die WCET relevant sein. Daher wird auf die Trace-Selektion aus der Arbeit von A. Smolarczyk [Smo10] zurückgegriffen, welche ebenfalls für eine WCET-spezifische Optimierung entwickelt wurde. Zur Auswahl eines Traces wird die *Longest-Path-Selektion* vorgestellt, welche versucht, einen für die WCET relevanten möglichst langen Pfad im Kontrollfluss zu ermitteln, damit dieser gezielt optimiert werden kann [Smo10]. Zur Ermittlung eines solchen Pfades bestimmt die Selektion zu Beginn die Knoten b_{seed} und b_{source} und ermittelt den längsten Pfad zwischen diesen Knoten. Der Knoten b_{source} ist entweder der Kopf einer Schleife, falls b_{seed} teil einer Schleife ist oder der erste Knoten im Kontrollflussgraphen. Zusätzlich zu den beiden Knoten wird noch ein Knoten als *Supersenke* definiert, welcher als Zusammenführung aller möglichen Kontrollflussmöglichkeiten agiert. Zwischen der Supersenke und dem Knoten b_{seed} wird ebenfalls ein längster Pfad ermittelt. Die Verbindung der Pfade $b_{source} \rightarrow b_{seed}$ und $b_{seed} \rightarrow b_{sink}$ bildet den Trace, welcher im weiteren Verlauf durch den List-Scheduler optimiert werden soll. Dieser Trace kann durch den List-Scheduler wie ein Basisblock behandelt werden, was jedoch eine Kompensation nach dem Scheduling benötigt, da es zu einer Änderung in der Programmsemantik durch das Verschieben von Instruktionen, über die Basisblockgrenzen hinweg, kommen kann.

Kompensation

Findet eine Verzweigung im Programmcode statt, beispielsweise durch einen bedingten Sprung, so wird dies als *Split* bezeichnet und erzeugt einen *Seitenausgang* aus einem Trace heraus, wodurch beim Scheduling von Instruktionen der Bedarf für eine Kompensation entstehen kann.



(a) Vor List-Scheduling (b) Nach List-Scheduling

Abbildung 4.5: Kompensationsbedarf eines Splits

Durch die Abbildung 4.5(a) ist ein Trace mit einem Seitenausgang visualisiert, welcher durch einen normalen List-Scheduler optimiert wird. Der List-Scheduler wählt die Instruk-

tion I1 aus und verschiebt diese weiter nach unten im Kontrollfluss. Durch die geänderte Reihenfolge der Maschineninstruktionen hat sich die Semantik des Programms verändert, da die Maschineninstruktion I1 nun erst nach der Instruktion I3 ausgeführt wird, was bedeutet, dass ein Kontrollfluss, welcher durch $I0 \Rightarrow I1 \Rightarrow I2 \Rightarrow I6$ verläuft, nun nur noch durch $I0 \Rightarrow I2 \Rightarrow I6$ verläuft und so die Instruktion I1 für diesen Kontrollfluss nicht mehr ausgeführt wird, was einer fehlerhaften Semantik entspricht. Die Kompensation muss diesen Fehler im Kontrollfluss reparieren, wozu eine Kopie der Instruktion I1 in den Basisblock D eingefügt werden muss und der Sprung aus dem Basisblock A muss auf die neue Instruktion I1' verlegt werden, wie es in der Abbildung 4.6 dargestellt ist.
Im Folgendem wird die Kompensation formal definiert, wozu die folgende Hilfsfunktion definiert wird [Smo10].

4.4.3 Definition (ContainingBB). Die Funktion $ContainingBB(ins_i)$ ordnet der Maschineninstruktion ins_i, den Basisblock bb_j zu, welcher Teil des *Trace* $T = (bb_1, \ldots, bb_t)$ ist und die Maschineninstruktion ins_i enthält.

Mithilfe dieser Funktion, welche jeder Instruktion den eindeutigen Basisblock eines Traces zuweist und der Funktion $Followers$ welche zu einem Basisblock die Menge der Nachfolger angibt, ist es möglich, die Instruktionen zu definieren, welche eine Split-Kompensation benötigen [Smo10].

4.4.4 Definition (Split-Instruktion). Gilt für eine Instruktion ins_i $|Followers(ContainingBB(ins_i))| \geq 2$, so wird diese Instruktion als Split-Instruktion bezeichnet.

Für das Beispiel aus der Abbildung 4.5(a) ist die Instruktion $I2$ eine Split-Instruktion, da der Basisblock A, welche die Instruktion enthält, die Nachfolger B und D besitzt.
Es gibt zwei Fälle, für die eine Kompensation nach dem Scheduling erforderlich ist.

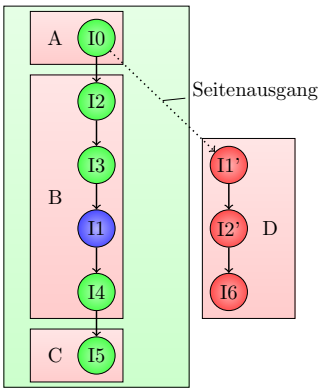
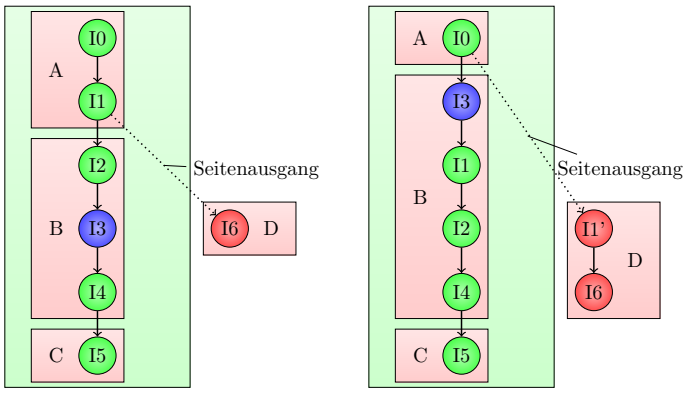


Abbildung 4.6: Split-Kompensation



(a) vor List-Scheduling (b) nach Split-Kompensation

Abbildung 4.7: Kompensation nach Verschiebung aufsteigend im Kontrollfluss

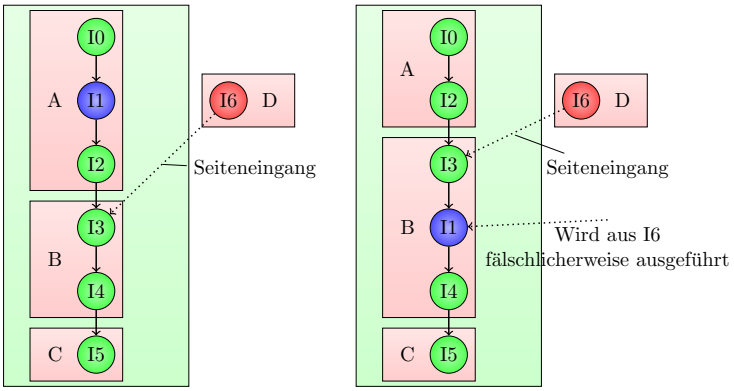
- Wird eine Maschineninstruktion von oberhalb einer Split-Instruktion nach unten im Kontrollfluss verschoben (siehe Abbildung 4.5(b)), muss die Maschineninstruktion, welche nach unten im Kontrollfluss verschoben wurde als Kopie in den Kontrollfluss, welcher dem Seitenausgang folgt, eingefügt werden. Die Kopie der Instruktion muss direkt nach dem Seitenausgang eingefügt werden, sodass eine Korrektur der Verzweigung und des folgenden Kontrollflusses erforderlich ist.
- Wird eine Maschineninstruktion von unterhalb einer Split-Instruktion nach oben im Kontrollfluss verschoben (siehe Abbildung 4.7), so müssen alle Maschineninstruktionen zwischen der Split-Instruktion und der neuen Position der verschobenen Instruktion in den Kontrollfluss des Seitenausganges kopiert werden. Die Kopien der Instruktionen müssen in identischer Reihenfolge zu Beginn des Kontrollflusses des Seitenausganges eingefügt werden und der Kontrollfluss entsprechend angepasst werden.

Zusätzlich zu den Split-Instruktionen gibt es die Join-Instruktionen, welche im folgendem Abschnitt formal definiert werden, wozu vorerst eine Hilfsfunktion definiert wird.

4.4.5 Definition (Predecessors). Die Funktion $Predecessors(b_i)$ gibt für einen Kontrollflussgraphen $G = (N, E)$ die Menge der Basisblöcke b_j an, für die gilt, dass eine Kante $(b_j, b_i) \in E$ zwischen $b_j \in N$ und $b_i \in N$ existiert.

Diese Funktion beschreibt die Vorgänger eines Basisblockes im Kontrollflussgraphen, womit es möglich ist, die Join-Instruktion zu definieren. Für die Instruktion I3 aus der Abbildung 4.8 liefert die Funktion $Predecessors(ContainingBB(I3))$ die Menge $\{A, D\}$ zurück.

4.4.6 Definition (Join-Instruktion). Eine Instruktion ins_i für die gilt, dass $|Predecessors(ContainingBB(ins_i))| \geq 2$ wird als Join-Instruktion bezeichnet.



(a) vor List-Scheduling (b) nach List-Scheduling

Abbildung 4.8: Bedarf für Join-Kompensation

Ähnlich dem Fall für eine Split-Instruktion, ist es für die Join-Instruktion ebenfalls notwendig, dass eine Kompensation durchgeführt wird, falls der List-Scheduler Instruktionen über eine Join-Instruktion verschiebt. Es gibt ebenfalls zwei Fälle, bei denen eine Kompensation erforderlich ist.

- Wird eine Instruktion über eine Join-Instruktion im Kontrollfluss nach unten verschoben, so führt dies dazu, dass der Kontrollfluss, welcher nicht auf dem Trace liegt, diese Instruktion fälschlicherweise auch ausführt. Dies wird durch Kopieren aller Instruktionen zwischen der Join-Instruktion und der Instruktion, zu der die neue Instruktion verschoben werden soll, erreicht. Diese Kompensation ist durch die Abbildung 4.9 dargestellt. Die Instruktion I3 wird kopiert und in den abgelegenen Kontrollfluss des Basisblockes D eingefügt. Anschließend wird der Sprung aus Basisblock D zu Basisblock C korrigiert.

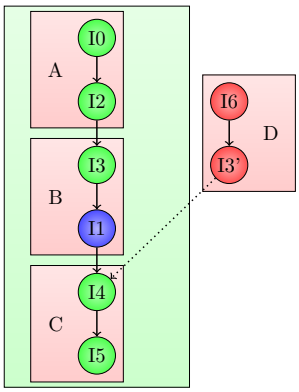
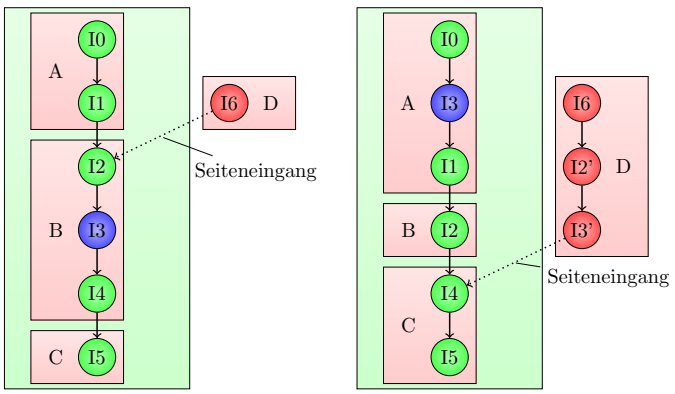


Abbildung 4.9: Join-Kompensation nach Abwärtsverschiebung

- Für den Fall, dass eine Instruktion im Kontrollfluss aufwärts über eine Join-Instruktion verschoben wird, ist ebenfalls eine Join-Kompensation notwendig, da die Instruktion, welche verschoben wird nicht mehr durch den fremden Kontrollfluss ausgeführt wird. Daher müssen alle Instruktionen, welche zwischen der Join-Instruktion und der ursprünglichen Position der verschobenen Instruktion liegen kopiert werden und in den abgelegenen Kontrollfluss kopiert werden. Abschließend ist eine Korrektur der Sprünge erforderlich. Dieser Fall ist durch die Abbildung 4.10 skizziert, welche zeigt, dass die Instruktionen I2 und I3 kopiert wurden, damit der Kontrollfluss des Basisblockes D semantisch korrekt bleibt.

Ist es nach dem List-Scheduling erforderlich, dass Kompensationscode erzeugt wird, so wächst die Größe des Programmcodes an, was dazu führen kann, dass die WCET steigt. Wird durch den List-Scheduler beim Verschieben einer Instruktion weder eine Split-Instruktion, noch eine Join-Instruktion überschritten, so ist keine Kompensation notwendig und die Verschiebung kann als abgeschlossen betrachtet werden, wie es beim lokalen Scheduling der Fall ist (siehe Unterabschnitt 4.4.1).
Zusätzlich zur Kompensation von Split- und Join-Instruktion müssen für das Trace-Scheduling die bedingten Ausführungen, welche in Abschnitt 4.1 vorgestellt wurden, besonders betrachtet werden. Dazu gehören auch Instruktionen, welche einen Sprung durchführen. Der Kompensationscode, der für das Verschieben von Sprüngen innerhalb eines Traces notwendig ist, ist sehr komplex, weshalb auf die Neuanordnung von Sprüngen verzichtet wird [Smo10]. Daher werden im Kontrollflussgraphen zusätzliche Kanten erstellt, welche die Reihenfolge der Sprünge repräsentieren, sodass ihre Reihenfolge in dem Kontrollflussgraphen beibehalten werden kann. Für bedingte Maschineninstruktion wird davon ausgegangen, dass die Instruktionen nicht umsortiert werden dürfen, da nicht gewährleistet werden kann, dass die Semantik dadurch nicht verändert wird.



(a) Vor List-Scheduling (b) Nach Join-Kompensation

Abbildung 4.10: Join-Kompensation nach Aufwärtsverschiebung

4.4.3 Superblock Scheduling

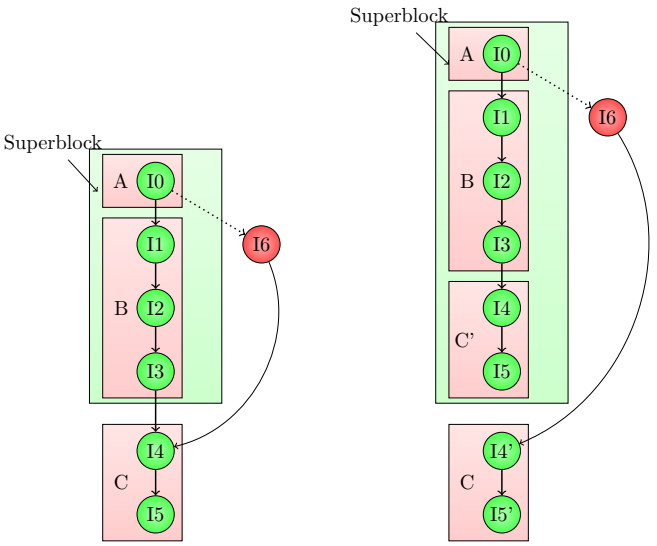
Eine Variante des Trace-Schedulings stellt das *Superblock Scheduling* dar, was auf einer Spezialisierung der Traces arbeitet. Da die Join-Kompensation verhältnismäßig aufwendig ist [HMC+93], besitzen Superblöcke keine Seiteneingänge, sodass keine Join-Instruktionen vorhanden sind und keine komplexen Kontrollstrukturen zur Kompensation notwendig sind.

4.4.7 Definition (Superblock). Ein Superblock S ist ein Trace $T = (b_0, \ldots, b_n)$, für den $\forall b_i \in \{b_1, \ldots, b_n\} : |Predecessors(b_i)| = 1$ gilt.

Da es sich bei einem Superblock um eine Spezialisierung eines Traces handelt, kann das Trace-Scheduling ohne weitere Anpassungen verwendet werden. Allerdings sind die natürlich auftretenden Superblöcke einer Anwendung in der Regel sehr klein, sodass nur ein geringer Gewinn gegenüber dem Basisblock-Scheduling vorhanden ist [Smo10].

Tail-Duplication

Mit der *Tail-Duplication* wird ein Verfahren vorgestellt, welches das Problem der zu kleinen Superblöcke löst. Dazu wird ein normaler Trace ausgewählt, welcher auch Seiteneingänge besitzen darf. Auf diesem Trace wird anschließend die Tail-Duplication durchgeführt, wozu eine Kopie des Basisblockes, welcher auf den Seiteneingang folgt, angelegt wird.
Eine solche Tail-Duplication wird in der Abbildung 4.11 schematisch für einen Superblock durchgeführt. Das Beispiel zeigt den Superblock, bestehend aus den Basisblöcken



(a) Vor Tail-Duplication (b) Nach Tail-Duplication

Abbildung 4.11: Vergrößerung von Superblöcken durch Tail-Duplication

A und B mit insgesamt vier Instruktionen. Wird eine Tail-Duplication des Basisblockes C durchgeführt, besitzt der Superblock die Basisblöcke A,B und C' mit insgesamt sechs Instruktionen, welche eine größere Freiheit für den Instruction-Scheduler bieten. Der Basisblock C' stellt eine Kopie des ursprünglichen Basisblockes C dar. Der Kontrollfluss muss ebenfalls angepasst werden, damit nach dem Superblock kein Sprung zu dem ursprünglichen Basisblock C durchgeführt wird. Zu beachten ist jedoch, dass es durch die Tail-Duplication immer zur Vergrößerung des Programmcodes kommt. Ebenfalls ist es durch die Tail-Duplication möglich, dass der WCEP (siehe Abschnitt 2.4) verändert wird, sodass eine erneute Analyse der WCET erforderlich wird. Durch die Steigerung der Codegröße und der Komplexität des Kontrollflussgraphen, welche durch das Kopieren von Basisblöcken entsteht, ist es möglich, dass sich die WCET durch das Superblock-Scheduling nicht reduziert.

4.5 Realisierung

Wie zuvor erwähnt, wurde als Basis für die Realisierung des Instruction-Schedulers der WCC-Übersetzer verwendet. Dafür wurde auf die bestehenden Arbeiten von A. Smolarczyk [Smo10] zurückgegriffen, welche einen Instruction-Scheduler für den *Infineon TriCore* implementiert. Allerdings ist dieser Instruction-Scheduler nicht ohne Anpassung für die ARM-Plattform zu übernehmen, da sich die TriCore- und die ARM-Architektur teilweise stark unterscheiden. Der TriCore-Prozessor ist ein Prozessor, welcher stark im Automobilbereich eingesetzt wird und daher spezifische Eigenschaften für die Anforderungen dieses Bereiches mit sich bringt. So bietet der TriCore-Prozessor einen großen Befehlssatz, der Befehle enthält, die denen eines DSP-Prozessors entsprechen. Ebenfalls besitzt der TriCore-Prozessor zwei Pipelines, welche unterschiedliche Befehle bearbeiten. Die sogenannte *Integer-Pipeline* führt arithmetische und logische Befehle wie beispielsweise die Addition oder eine UND-Verknüpfung aus, während die *Load-Store-Pipeline* Befehle ausführt, welche für den Zugriff auf den Hauptspeicher zuständig sind [Inf03]. Das Ziel der Arbeit von A. Smolarczyk [Smo10] war es, einen Instruction-Scheduler zu schreiben, welcher die verschiedenen Pipelines des Prozessors gut auslastet, während das Ziel dieser Arbeit ist, den Bus zwischen den Prozessorkernen gut auszulasten. Der Scheduler aus der Arbeit von A. Smolarczyk [Smo10] versucht durch Instruktionsbündel die verschiedenen Pipelines aufzufüllen. Dieses Modell mit Instruktionsbündel ist für das Scheduling der Instruktionen eines ARM-Prozessors ungeeignet, da nur einzelne Instruktionen verplant werden müssen.
Zusätzlich müssen andere architekturspezifischen Eigenschaften des TriCore Prozessors durch die des ARM-Prozessors ersetzt werden. Diese beinhalten die Kompensation, welche für Split- und Join-Instruktionen notwendig ist, sowie die Tail-Duplication, welche ebenfalls architekturspezifischen Programmcode enthält, welcher den Kontrollfluss der Anwendung

regelt. Des Weiteren muss für die Erstellung des Kontrollflussgraphen das Vorhandensein von bedingten Maschineninstruktionen beachtet werden (siehe Abschnitt 4.1), welche als kontrollabhängig behandelt werden.

Durch die Abbildung 4.12 ist skizziert, wie die Optimierung mithilfe des Instruction-Schedulers im WCC-Übersetzer abläuft. Zu Beginn wird eine Sicherung der *LLIR* angelegt, damit für die wiederholten Aufrufe des Instruction-Schedulers die originale LLIR wiederhergestellt werden kann, sodass sichergestellt ist, dass der Instruction-Scheduler für jeden Aufruf auf der selben Basis arbeitet. Im nächsten Schritt wird der Instruction-Scheduler mit der TDMA-Offset-basierten Heuristik aufgerufen. Anschließend wird die LLIR wiederhergestellt und der Instruction-Scheduler für die Always-First und anschließend, nach erneuter Wiederherstellung der LLIR, für die Only-Fitting Heuristik aufgerufen. Anschließend wird die beste LLIR in Bezug auf WCET ausgewählt. Diese Schritte werden für alle Tasks aus der zu optimierenden Taskmenge wiederholt.

Die Implementierung des List-Schedulers arbeitet auf Basis einer Abstraktion einer Region, wodurch es möglich ist, die verschiedenen Scheduling-Verfahren (lokales, Trace-basiertes und Superblock-basiertes Scheduling) mit dem selben Scheduler durchzuführen. Lediglich die Kompensation der Join- und Split-Instruktionen muss für die jeweiligen Verfahren nach dem Scheduling durchgeführt werden. Ebenfalls ist es erforderlich für das Superblock-Scheduling die Bildung der Superblöcke vor dem Scheduling durchzuführen, damit der Scheduler auf der fertigen Datenstruktur arbeiten kann.

4.6 Evaluation

Zur Evaluation des Instruction-Schedulers wurden verschiedene Tests zur Bestimmung der Leistungsfähigkeit und der Korrektheit durchgeführt. So wurde für die Optimierung eine Sammlung von verschiedenen Benchmarks überprüft, sodass für die Optimierung bestimmt werden kann, dass die Programmsemantik nicht verfälscht wurde. Die verwendeten Benchmarks setzen sich erneut aus verschiedenen Benchmarksammlungen zusammen, welche bereits in Abschnitt 3.4 vorgestellt wurden.

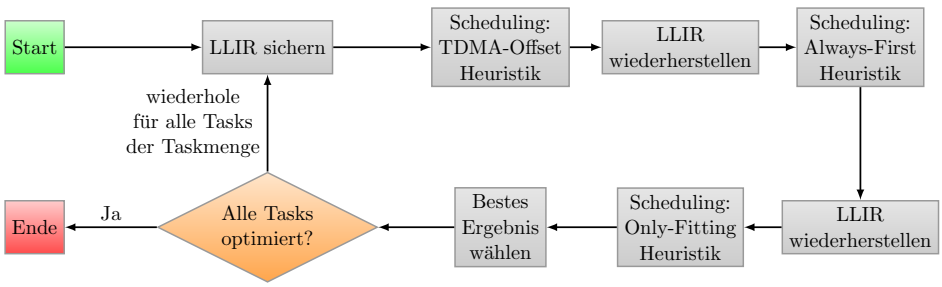


Abbildung 4.12: Ablauf der Optimierung durch den Instruction-Scheduler

Diese Tests enthalten verschiedene Programme, welche zunächst mit dem *GCC* kompiliert und anschließend simuliert werden, was anschließend mit dem *WCC* wiederholt wird. Die resultierenden Ergebnisse werden verglichen und Abweichungen werden vermerkt. Die Tests auf Korrektheit haben gezeigt, dass die Optimierung die Semantik der Anwendungen nicht verändert, sodass die Ergebnisse mit denen des GCC-Übersetzers übereinstimmen.
Die Evaluation wurde auf einem Computer mit zwei *Intel Xeon X5650* Prozessoren und 20 GB Arbeitsspeicher durchgeführt. Als Testmenge wurden erneut die Bündel aus Abschnitt 3.4 gewählt. Der Instruction-Scheduler wurde mit verschiedenen Heuristiken ausgeführt und anschließend ein Vergleich durchgeführt, damit die Güte der einzelnen Instruktionsauswahlheuristiken bestimmt werden kann. Außerdem wurde die Evaluation für das lokale Scheduling, das Trace-basierte und das Superblock-basierte Scheduling wiederholt, sodass die Optimierungsgüte der einzelnen Verfahren bestimmt wird. Alle Tests wurden für das TDMA- und PD-Verfahren und jeweils mit einer Slotlänge pro Prozessorkern von 3, 6 und 12 Taktzyklen durchgeführt, da dies Vielfache der maximalen Zugriffszeit für eine Instruktion mit Buszugriff, welche 3 Taktzyklen ist, sind.
Eine erste Evaluation des Instruction-Schedulers hat ergeben, dass der Einfluss der Optimierung nur äußerst gering war, da nur wenige Instruktionen einen Zugriff auf den Bus benötigt haben. Daher wurde für die Evaluation des Instruction-Schedulers zusätzlich, zum Datenbereich der Anwendungen, der Stack hinter den gemeinsamen Bus gelegt, was zu einer Steigerung der Busauslastung führt, wodurch sich ebenfalls der Einfluss des Instruction-Schedulers erhöht.
In der Abbildung 4.13 ist die Verbesserung der WCET für verschiedene Programme aufgetragen, welche durch den Instruction-Scheduler optimiert wurden. Dafür wurde eine Konfiguration mit zwei Prozessorkernen und das lokale Instruction-Scheduling verwendet. Die Abbildung zeigt die relative maximale Verbesserung, welche für die verschiedenen Slotlängen (3,6 und 12 Taktzyklen) für die TDMA- und PD-Konfigurationen erreicht wurde. Für jeden Task ist die relative Verbesserung separat für die verschiedenen Heuristiken, welche zur Auswahl der Instruktionen durch den Scheduler verwendet werden, angegeben. Die Angabe *TDMA-Offsets* bezieht sich auf die Heuristik, welche sich die TDMA-Offsets aus Abschnitt 2.4 zu Nutze macht. Die Angabe *Always-First* gibt die Verbesserung für die Heuristik an, welche nur die Slotlänge der jeweiligen Prozessorkerne nutzt und jeder ersten Instruktion einer Folge, die einen Buszugriff durchführt, eine hohe Priorität zuweist. Durch *Only-Fitting* ist die Verbesserung durch die Variante der *Always-First* Heuristik angegeben. Diese Heuristik prüft auch für die erste Instruktion mit Buszugriff, ob diese in einen Slot passt.
Es ist zu erkennen, dass für viele Tasks nur eine geringe Verbesserung durch das Instruction-Scheduling erzielt werden konnte. Allerdings wurde für einzelne Tasks eine Verbesserung von ungefähr 20-30 % erreicht. Außerdem ist erkennbar, dass keine Heuristik die anderen dominiert. Für jede Heuristik gibt es Fälle, in denen sie besser ist als andere. Teilweise

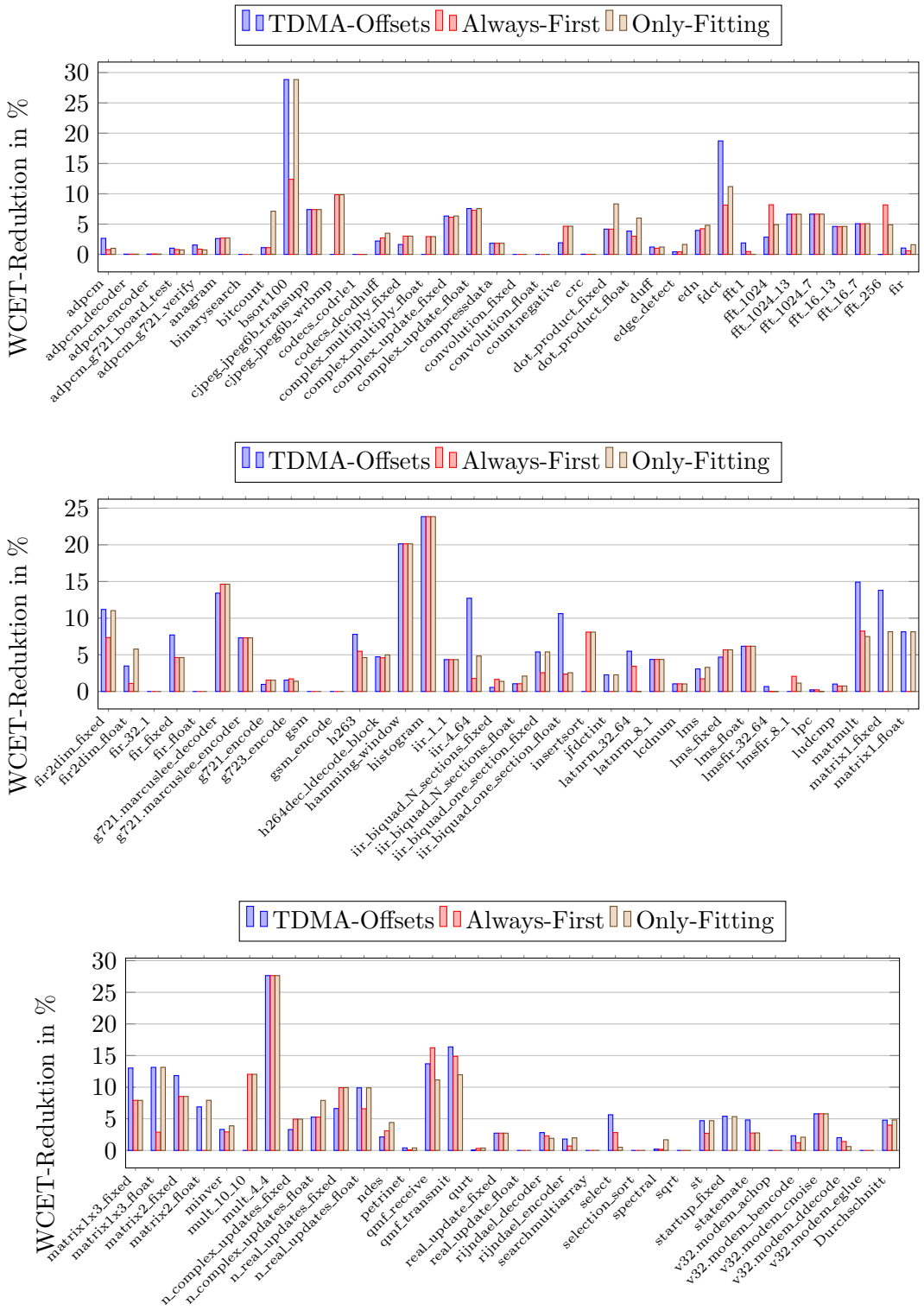


Abbildung 4.13: Vergleich der Selektionsheuristiken, maximale WCET-Verbesserung über die Slotlängen 3,6 und 12 Taktzyklen, lokales Scheduling (ARM7-x2)

erzielen alle drei verwendeten Heuristiken gleich gute Ergebnisse. In wenigen Fällen ist zu beobachten, dass nur eine der drei Heuristiken zu einer Verbesserung der WCET führt.
Die Ergebnisse der Optimierung durch das lokale Scheduling für eine Hardwarekonfiguration mit vier Prozessorkernen sind in der Abbildung 4.14 abgebildet. Es konnte eine maximale Verbesserung von 31,8 % für den Task *histogram* erreicht werden. Die Verbesserung wurde durch alle drei Heuristiken erreicht. Für den Task *v32.modem_ddecode* konnte für eine Heuristik nur eine Verschlechterung der WCET erreicht werden, was aber durch eingebaute Mechanismen zur Wiederherstellung der ursprünglichen Anordnung der Instruktionen verhindert werden kann. Auch für diese Konfiguration konnte für einzelne Tasks durch eine einzige Heuristik bessere Ergebnisse als durch die anderen Heuristiken erreicht werden. So konnte beispielsweise für den Task *fdct* nur die TDMA-Offset-basierte Heuristik eine hohe Verbesserung erreichen.
Durch die Abbildung 4.15 sind die Ergebnisse der Evaluation des lokalen Instruction-Schedulers für eine Hardwarekonfiguration mit acht Prozessorkernen aufgetragen. Es ist zu erkennen, dass nur noch für eine geringe Anzahl von Tasks eine Verbesserung der WCET erreicht werden konnte, was eventuell durch eine Lange Periode des TDMA- und PD-Verfahren zu erklären ist, da es schwer ist, eine Wartezeit von 21, 42 und 84 Taktzyklen mit Instruktionen, welche keinen Zugriff auf den Bus benötigen, zu überbrücken. Auffällig ist die WCET-Verbesserung des Tasks *fft_1024*, welche mit 81 % sehr hoch ausfällt. Es handelt sich dabei erneut um eine *Fourier-Transformation*, welche in Abschnitt 3.4 durch eine ebenfalls hohe Reduktion der WCET auffiel. Die Fourier-Transformation verwendet vier globale Arrays mit jeweils 1024 Gleitkommazahlen. Es wurden 144 Instruktionen durch den Scheduler behandelt, von denen 49 Instruktionen sind, welche einen Zugriff auf den Bus benötigen. Das heißt, dass 34 % der Instruktionen einen Zugriff auf den Bus benötigen. Für den Task *g721.marcuslee_decoder* beispielsweise, wurden 49 Instruktionen behandelt, von denen nur 3 Instruktionen einen Zugriff auf den Bus benötigen, was 6,66 % entspricht. Durch die hohe Abhängigkeit vom gemeinsamen Bus, kommt eventuell die große Varianz bei der Bestimmung der WCET für den Task *fft_1024* zustande.
Wird der Task *fft_1024* als Ausreißer betrachtet, so wurde für den Task *g721.marcuslee_decoder* die größte Verbesserung der WCET mit 25 % erreicht. Zusätzlich ist für die Hardwarekonfiguration mit acht Prozessorkernen feststellbar, dass sich die Heuristiken häufiger in ihrer Güte unterscheiden. Das heißt, dass die TDMA-Offset-basierte Heuristik sich häufiger unterschiedlich zu der Always-First und Only-Fitting Heuristik verhält.
Durch die Abbildung 4.16 wird die durchschnittliche Verbesserung der WCET für alle drei Heuristiken und Hardwarekonfiguration dargestellt. In der Abbildung 4.16 ist zu erkennen, dass die TDMA-Offset Heuristik für die Hardwarekonfigurationen mit vier und acht Prozessorkernen die höchste Reduktion der WCET erreicht. Für die Hardwarekonfiguration mit zwei Prozessorkernen liefert die Only-Fitting Heuristik geringfügig bessere Ergebnisse im Durchschnitt. Auffällig ist die hohe Verbesserung von 4,8 % für vier Prozessorkerne, die

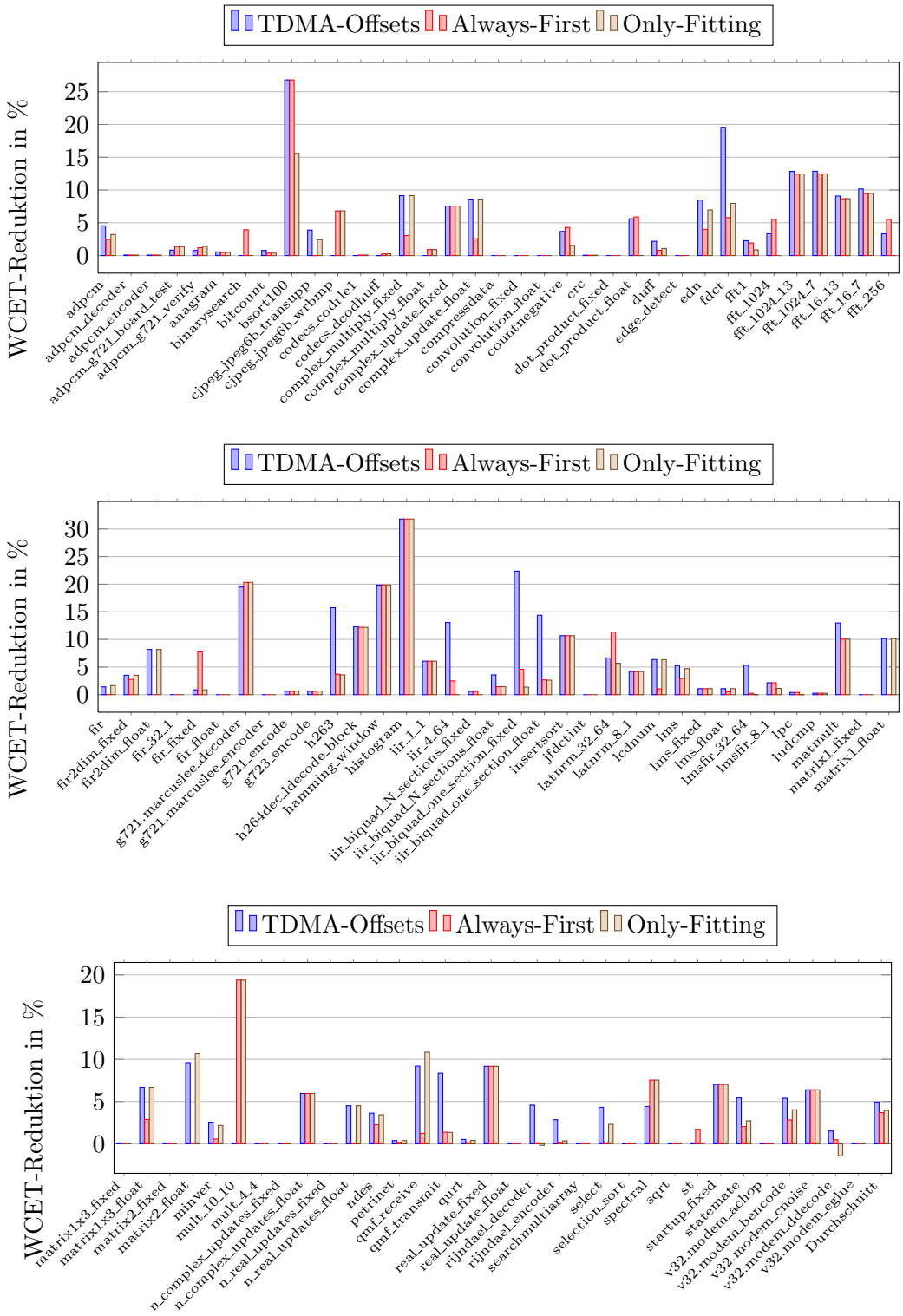


Abbildung 4.14: Vergleich der Selektionsheuristiken, maximale WCET-Verbesserung über die Slotlängen 3,6 und 12 Taktzyklen, lokales Scheduling (ARM7-x4)

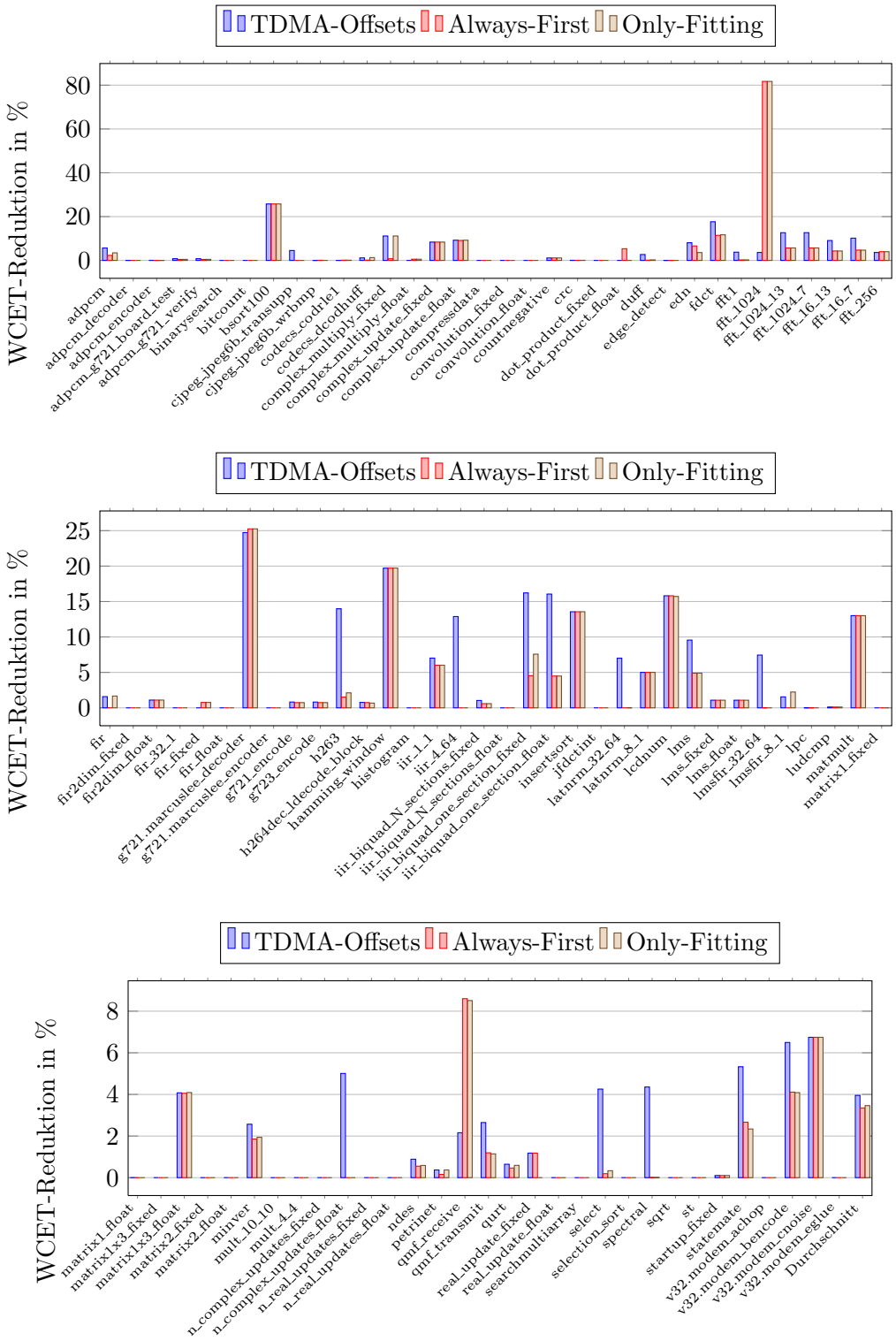


Abbildung 4.15: Vergleich der Selektionsheuristiken, maximale WCET-Verbesserung über die Slotlängen 3,6 und 12 Taktzyklen, lokales Scheduling (ARM7-x8)

durch die TDMA-Offset Heuristik erreicht wurde. Für diesen Fall liegt die durchschnittliche Verbesserung zirka 1 % höher als die der Only-Fitting Heuristik. Für das lokale Scheduling liefert erwartungsgemäß die Always-First Heuristik die geringste Verbesserung.

Evaluationsergebnisse für das Trace-basierte Instruction-Scheduling sind in der Abbildung 4.17 für die Hardwarekonfiguration mit zwei Prozessorkernen aufgetragen. Die maximale Verbesserung konnte für den Task *mult_4_4* mit ungefähr 27 % erreicht werden. In vielen Fällen konnte die Always-First Heuristik keine oder nur eine geringere Verbesserung als die TDMA-Offset-basierte und Only-Fitting Heuristik erzielen.

Durch die Abbildung 4.18 ist die Verbesserung der WCET für vier Prozessorkerne dargestellt. Die maximale Verbesserung der WCET konnte für den Task *g721.marcuslee_decoder* durch die Only-Fitting Heuristik mit ungefähr 20 % erreicht werden. Für die TDMA-Offset-basierte Heuristik ist zu erkennen, dass diese für einige Tasks als einzige Heuristik eine Verbesserung erzielen kann. Die Always-First Heuristik erzielt häufiger keine Verbesserung als die TDMA-Offset-basierte und Only-Fitting Heuristik.

Für die Hardwarekonfiguration mit acht Prozessorkernen sind die Ergebnisse in der Abbildung 4.19 für das Trace-basierte Scheduling aufgetragen. Ähnlich wie durch die Abbildung 4.18 gezeigt, konnte für den Task *g721.marcuslee_decoder* erneut die maximale WCET-Verbesserung mit zirka 25 % durch die Only-Fitting und TDMA-Offset Heuristiken erreicht werden. Für viele Tasks konnte jedoch oft keine Verbesserung erzielt werden. Die Laufzeiten für das Kompilieren inklusive des Instruction-Schedulings lagen für das Trace-basierte Scheduling zwischen 17 Sekunden und 85 Minuten je nach optimierter Taskmenge.

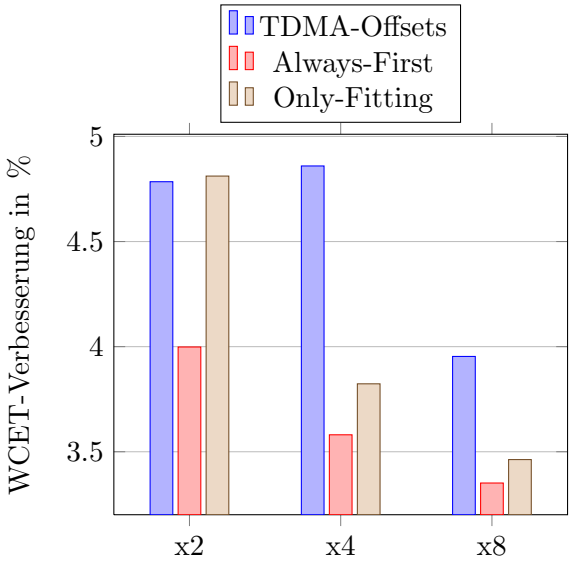


Abbildung 4.16: durchschnittliche Verbesserung der WCET für lokales Scheduling und Vergleich der Heuristiken

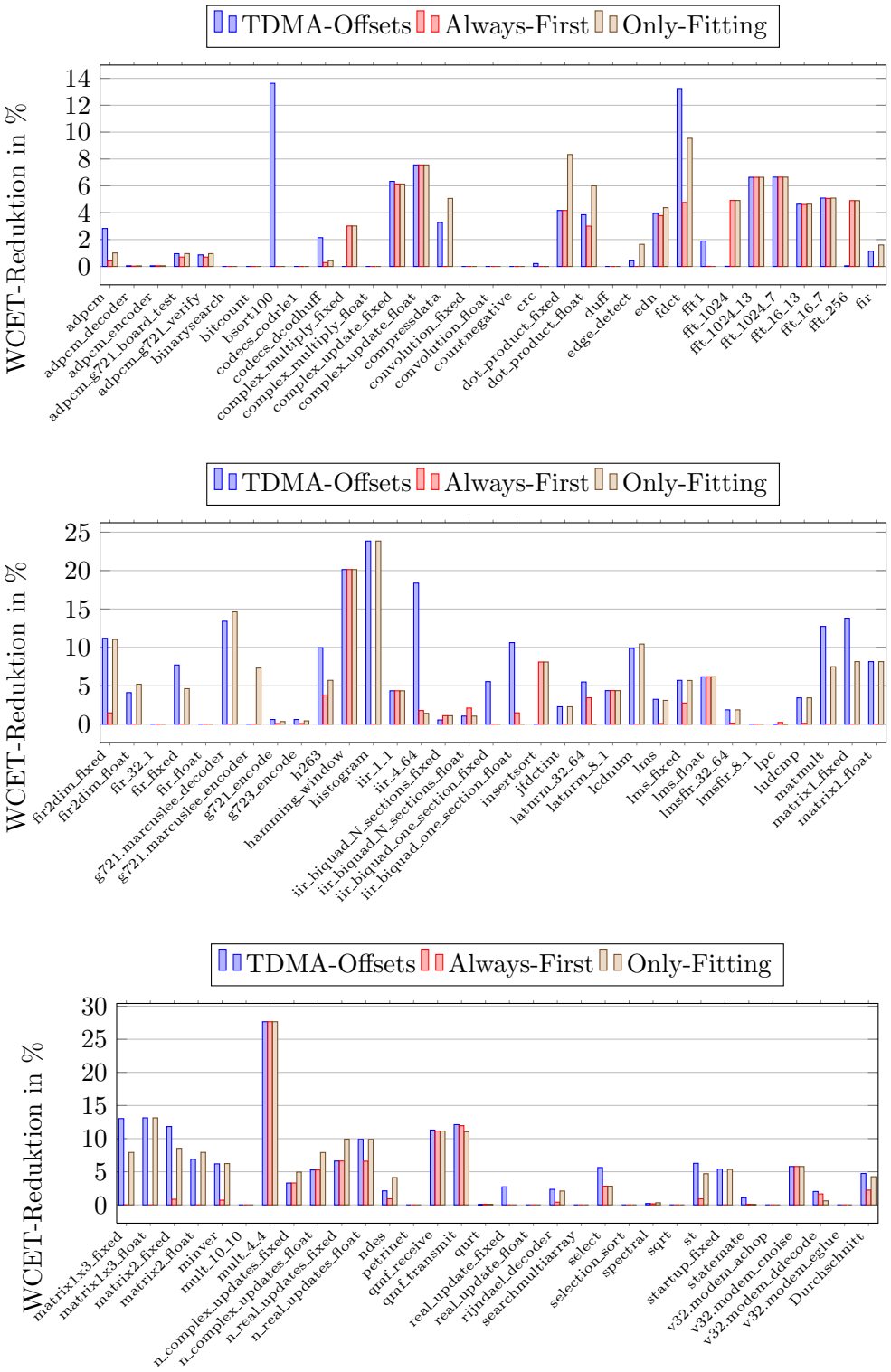


Abbildung 4.17: Vergleich der Selektionsheuristiken, maximale WCET-Verbesserung über die Slotlängen 3,6 und 12 Taktzyklen, Trace-Scheduling (ARM7-x2)

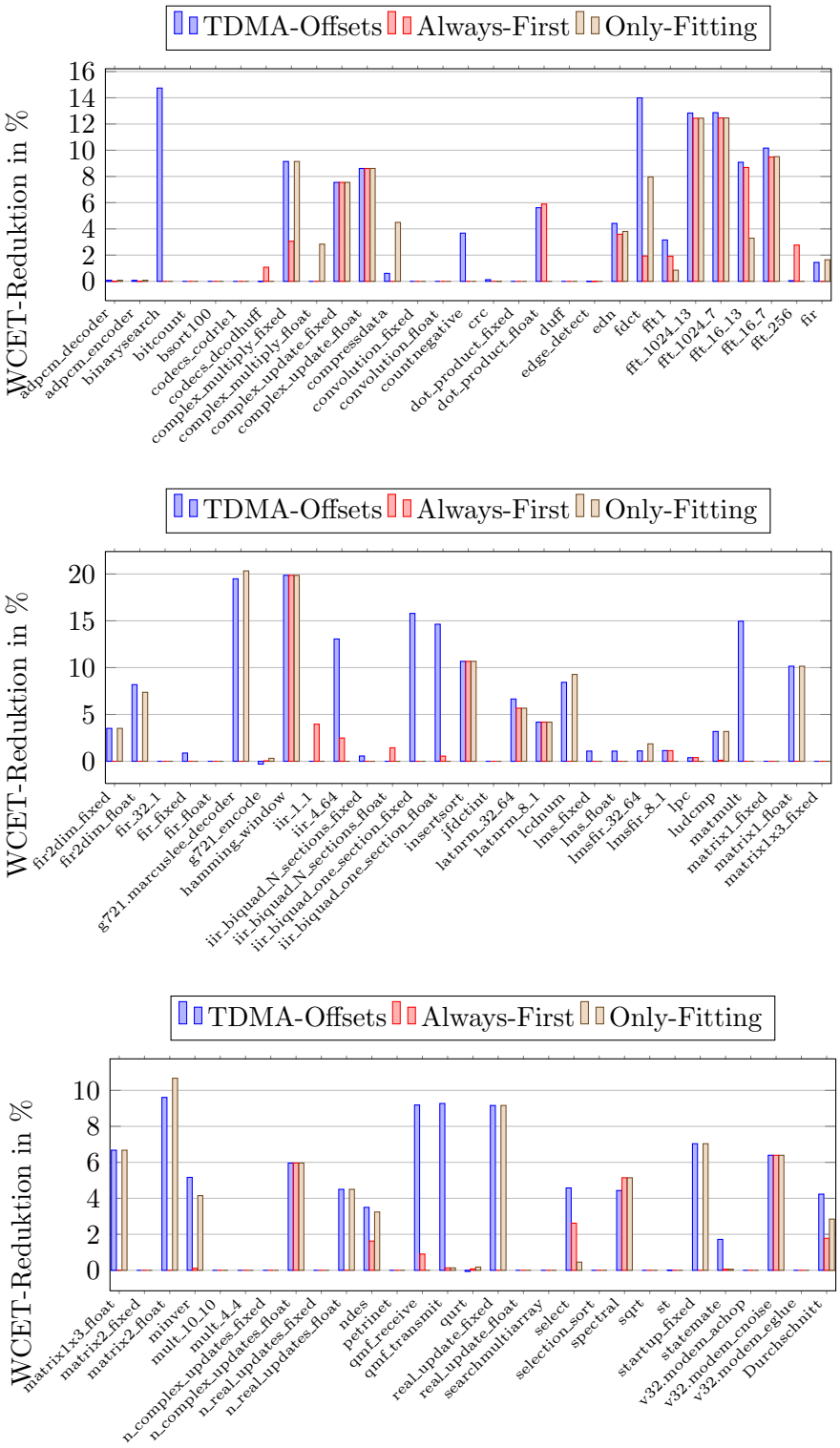


Abbildung 4.18: Vergleich der Selektionsheuristiken, maximale WCET-Verbesserung über die Slotlängen 3,6 und 12 Taktzyklen, Trace-Scheduling (ARM7-x4)

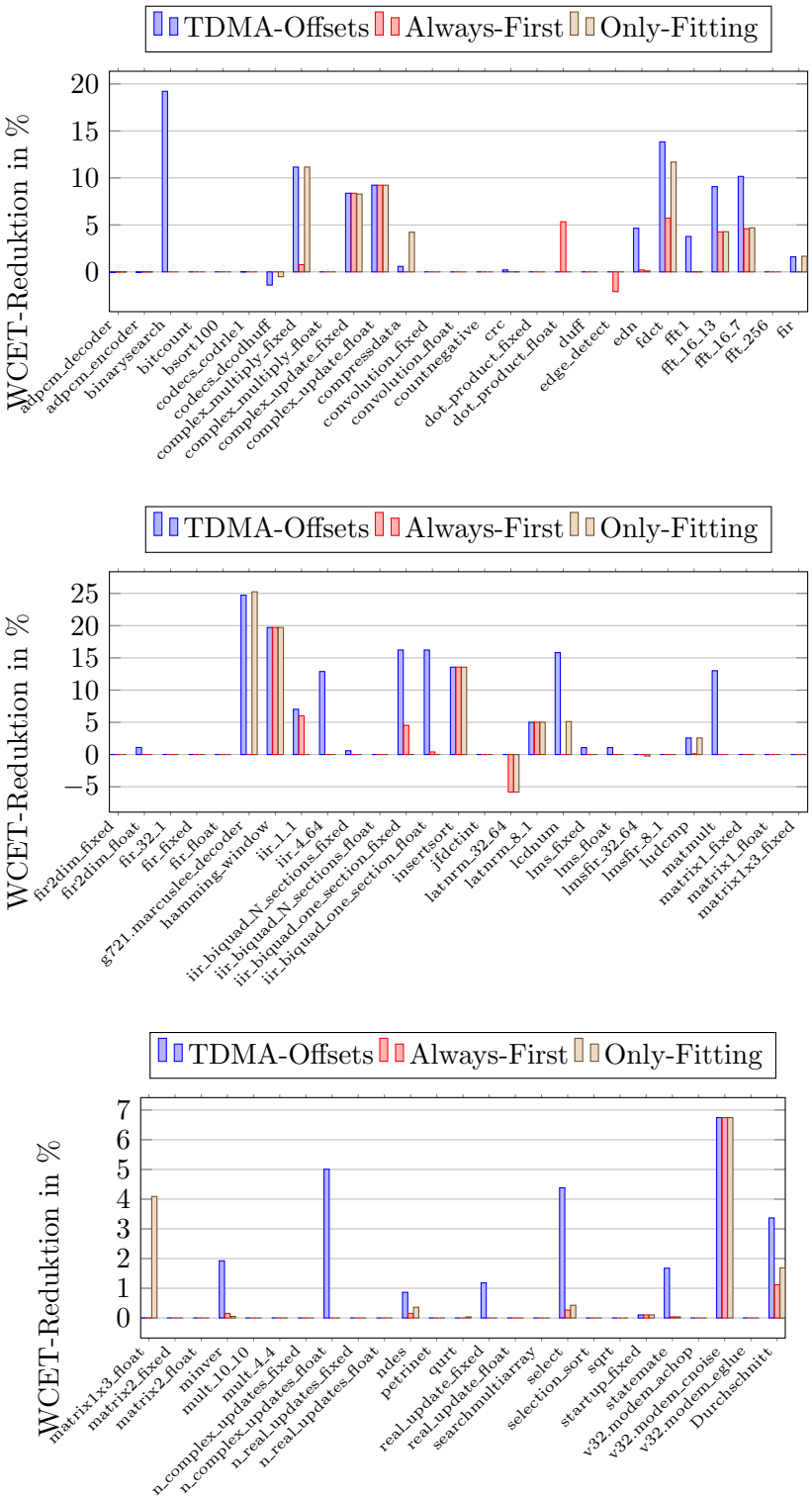


Abbildung 4.19: Vergleich der Selektionsheuristiken, maximale WCET-Verbesserung über die Slotlängen 3,6 und 12 Taktzyklen, Trace-Scheduling (ARM7-x8)

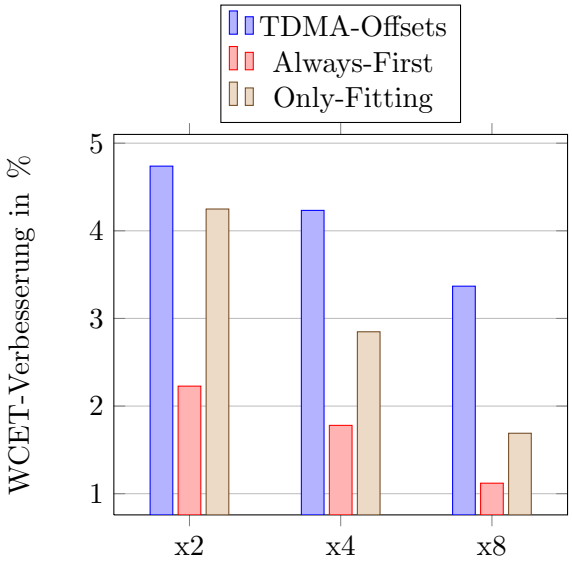


Abbildung 4.20: durchschnittliche Verbesserung der WCET für Trace-Scheduling und Vergleich der Heuristiken

In der Abbildung 4.20 ist die durchschnittliche Verbesserung der WCET für das Trace-Scheduling dargestellt. Es ist erkennbar, dass die TDMA-Offset Heuristik für die Hardwarekonfigurationen mit zwei, vier und acht Prozessorkernen die höchste Verbesserung erreicht. Für die Hardwarekonfiguration mit zwei Prozessorkernen liegt diese bei ungefähr 4,85 %. Erneut erzielt die Always-First Heuristik die geringste Durchschnittsverbesserung der WCET mit nur ungefähr 2 %. Für die Hardwarekonfiguration mit zwei Prozessorkernen liefert die Only-Fitting Heuristik ähnlich gute Ergebnisse wie die TDMA-Offset Heuristik. Für eine Hardwarekonfiguration mit vier oder acht Prozessorkernen ist die Differenz zwischen Only-Fitting und TDMA-Offset Heuristik gestiegen.

Insgesamt fällt die Reduktion der WCET für das Trace-basierte Instruction-Scheduling geringer aus, als für das lokale Instruction-Scheduling. Da für das Trace-basierte Scheduling zusätzlicher Programmcode erzeugt werden muss, kann es vorkommen, dass die WCET steigt, falls mehr Instruktionen ausgeführt werden müssen. Ist der Gewinn durch das Neuanordnen geringer als die Steigerung durch den Kompensationscode, so kann die WCET in diesem Fall nicht weiter reduziert werden.

Die Abbildung 4.21 stellt Ergebnisse für das Superblock-basierte Instruction-Scheduling auf Basis der Hardwarekonfiguration mit zwei Prozessorkernen vor. Die Ergebnisse weisen eine starke Ähnlichkeit zu den Ergebnissen des Trace-basierten Scheduling (siehe Abbildung 4.17) auf, was aufgrund der Ähnlichkeit der beiden Verfahren zu erklären war. Es wurde ebenfalls für den Task *mult_4_4* die maximale Verbesserung mit ungefähr 27 % erzielt. Im Vergleich zum Trace-basierten Scheduling wurde durch das Superblock-basierte

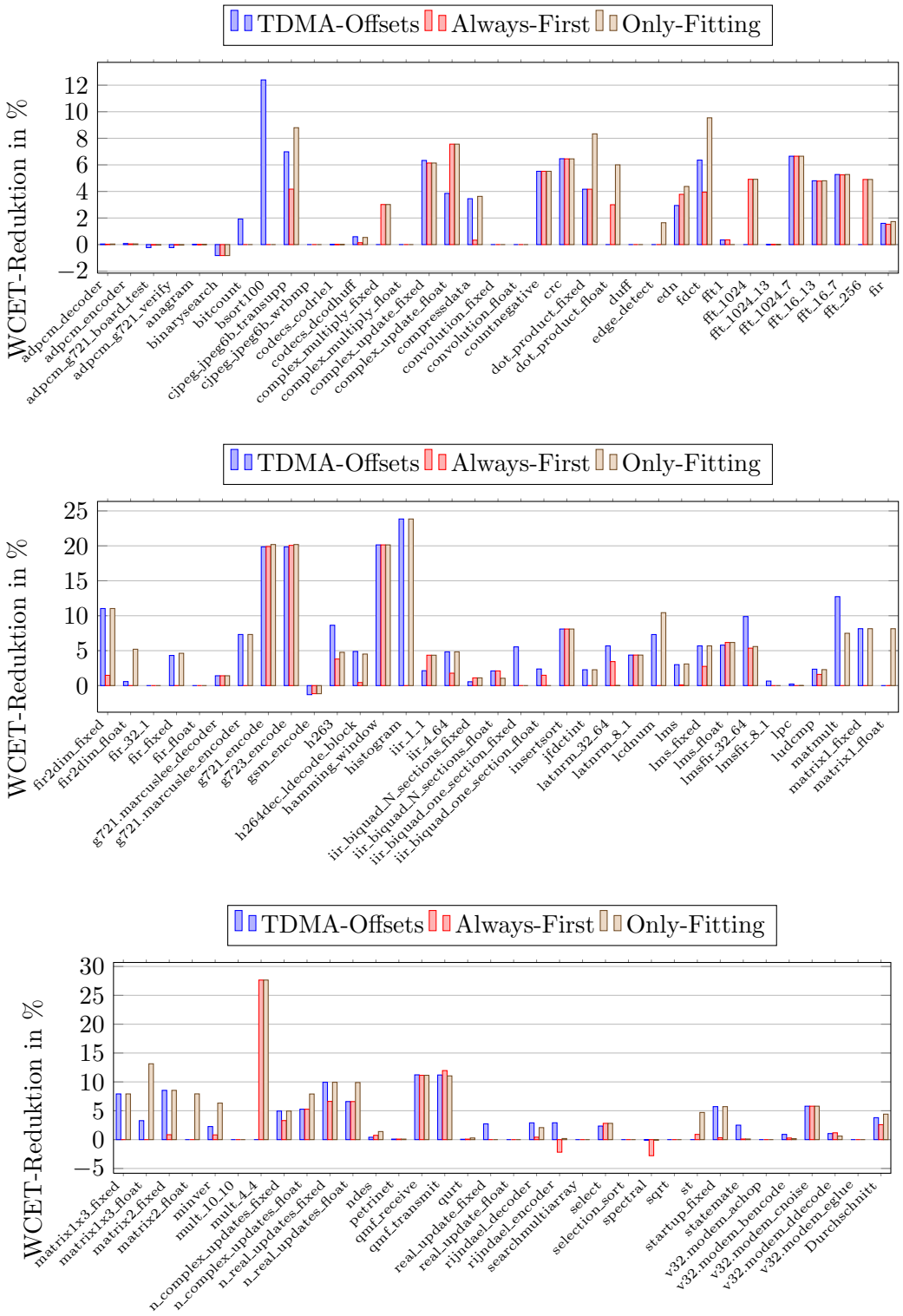


Abbildung 4.21: Vergleich der Selektionsheuristiken, maximale WCET-Verbesserung über die Slotlängen 3,6 und 12 Taktzyklen, Superblock-Scheduling (ARM7-x2)

Scheduling häufiger eine Verschlechterung der WCET durch den Instruction-Scheduler bewirkt.

Für die Hardwarekonfiguration mit vier Prozessorkernen sind die Ergebnisse ähnlich, wie in der Abbildung 4.22 erkennbar ist. Die maximale Reduktion von ungefähr 27 % konnte für die Tasks *g721_encode* und *g723_encode* erzielt werden.

Für die Hardwarekonfiguration mit acht Prozessorkernen ist zu erkennen (siehe Abbildung 4.23), dass die Reduktion der WCET durch den Instruction-Scheduler insgesamt geringer ausfällt, als für die beiden anderen Hardwarekonfigurationen. Es konnte für den Task *hamming_window* eine maximale Verbesserung von 19,7 % erzielt werden.

Für das Superblock-basierte Instruction-Scheduling ist die durchschnittliche Verbesserung der WCET für die Hardwarekonfiguration mit zwei Prozessorkernen und der Only-Fitting Heuristik 4,7 %, wie die Abbildung 4.24 zeigt. Die TDMA-Offset-basierte Heuristik konnte für die gleiche Hardwarekonfiguration eine Reduktion von 4 % und die Always-First Heuristik von 2,8 % erreichen. Für die Hardwarekonfigurationen mit vier und acht Prozessorkernen liefert die TDMA-Offset-basierte Heuristik die besten Ergebnisse mit 3,6 % beziehungsweise 2,4 %.

Der Vergleich des Superblock-Schedulings zu dem Trace-basierten Scheduling und dem lokalen Scheduling zeigt, dass das Superblock-Scheduling eine geringere Verbesserung für alle drei Hardwarekonfigurationen erzielt hat. Da das Superblock-Scheduling immer zusätzlichen Programmcode erzeugt, ist es möglich, dass die WCET dadurch nicht zu reduzieren ist.

Ebenfalls ist es das Ziel des Instruction-Schedulers die TDMA- und PD-Periode möglichst gut auszunutzen. Da die Tests mit 3, 6 und 12 Taktzyklen als Periodenlänge durchgeführt wurden, ist es möglich, dass die gesteigerte Freiheit durch das Trace-basierte und Superblock-basierte Instruction-Scheduling nicht erforderlich ist, da eine Instruktion mit Buszugriff im Falle eines erlaubten Zugriffes ebenfalls 3 Taktzyklen benötigt. Im schlechtesten Fall benötigt eine solche Instruktion für eine Hardwarekonfiguration mit vier Prozessorkernen 12 Taktzyklen, falls diese gerade ihren TDMA-Slot verpasst hat. Da jedoch verhältnismäßig kurze Periodenlängen verwendet werden, ist es nur erforderlich ein, zwei oder vier Instruktionen mit Zugriff auf den Bus auszuführen, damit der TDMA-Slot ausgenutzt wird. Diese Anzahl an Instruktionen ist durch das Basisblock-basierte Instruction-Scheduling schon gegeben, da die Auswertung, für die für diese Evaluation verwendeten Programme, ergeben hat, dass die durchschnittliche Länge von Basisblöcken 5,25 Instruktionen beträgt. Die durchschnittliche Anzahl an Businstruktionen pro Basisblock liegt bei nur 1,01 Instruktionen.

In der Abbildung 4.25 ist das Verhältnis von Instruktionen, die einen Buszugriff benötigen, und Instruktionen, die keinen Buszugriff benötigen, angegeben. Es ist zu erkennen, dass der Anteil der Instruktionen mit Buszugriff verhältnismäßig gering ist, wodurch das

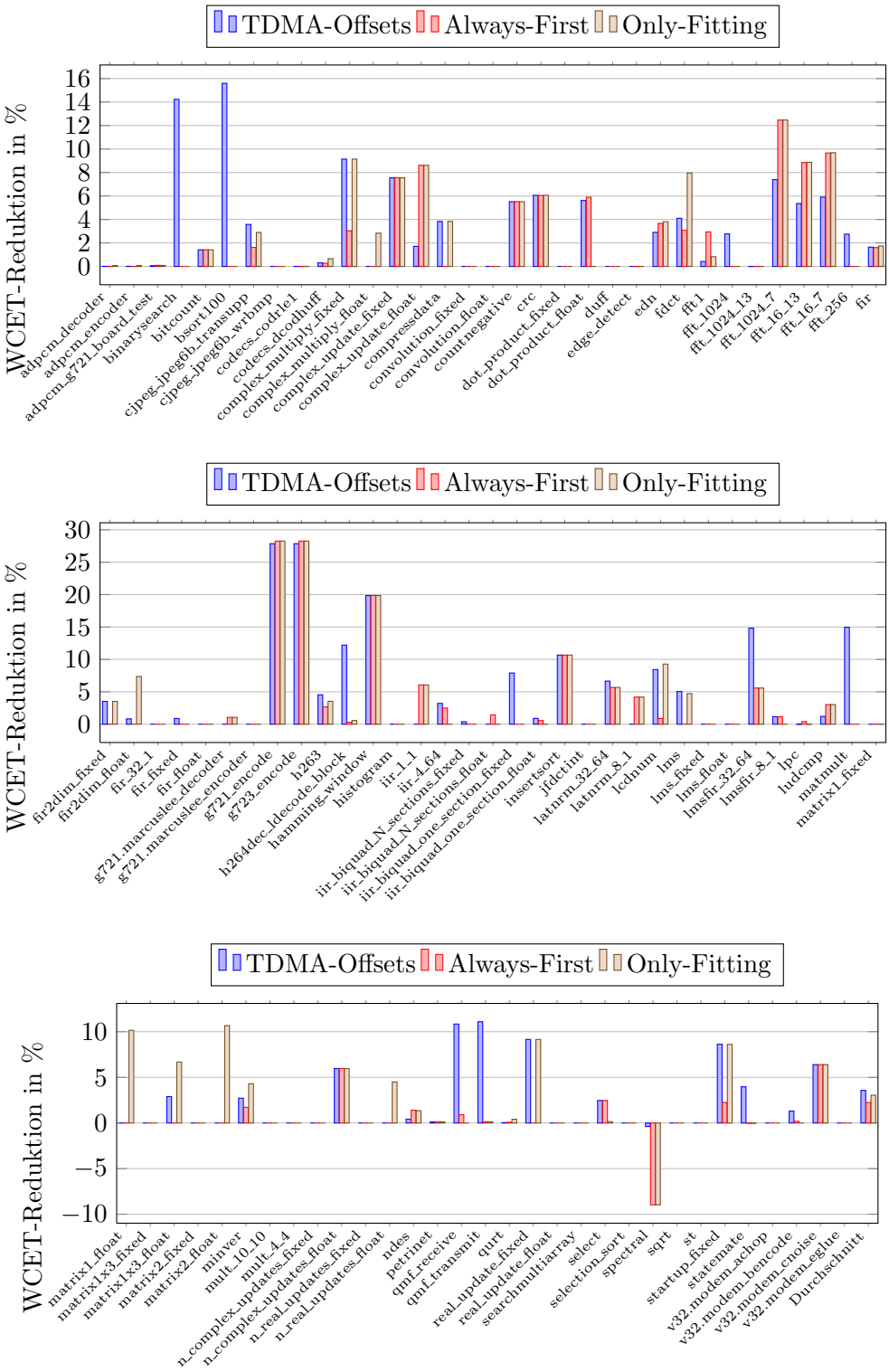


Abbildung 4.22: Vergleich der Selektionsheuristiken, maximale WCET-Verbesserung über die Slotlängen 3,6 und 12 Taktzyklen, Superblock-Scheduling (ARM7-x4)

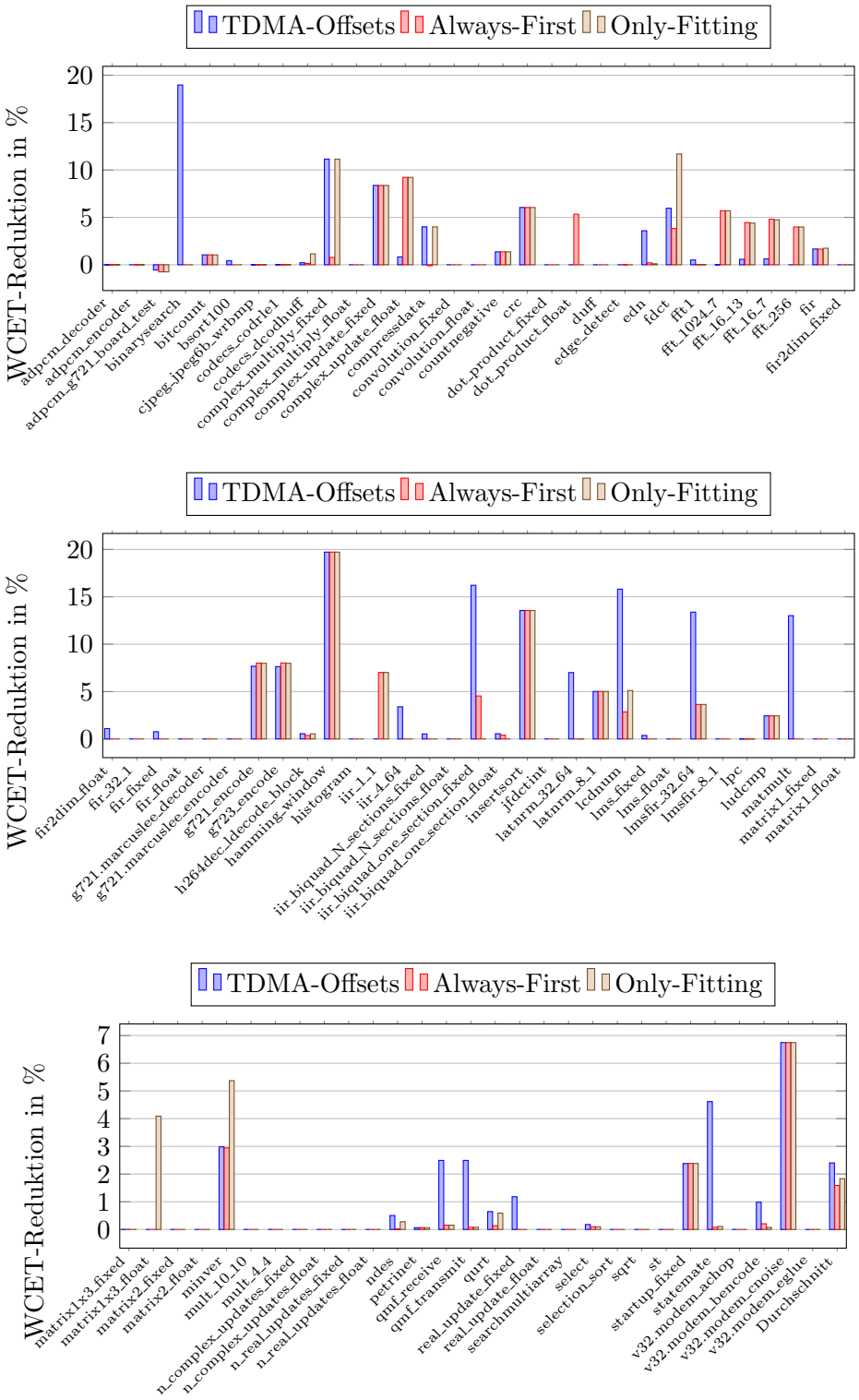


Abbildung 4.23: Vergleich der Selektionsheuristiken, maximale WCET-Verbesserung über die Slotlängen 3,6 und 12 Taktzyklen, Superblock-Scheduling (ARM7-x8)

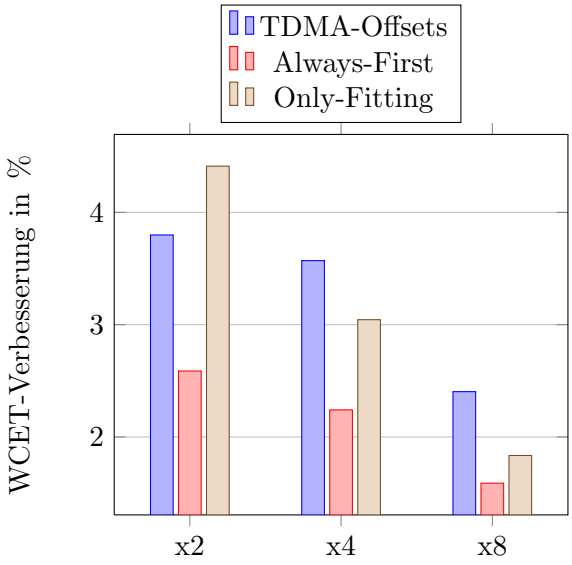


Abbildung 4.24: durchschnittliche Verbesserung der WCET für Superblock-Scheduling und Vergleich der Heuristiken

teilweise sehr geringe Optimierungspotenzial zu erklären ist. Der durchschnittliche Anteil an Buszugriffen liegt bei 19,44 %.

Die Laufzeiten für das Instruction-Scheduling auf Basis von Superblöcken lag zwischen 100 Sekunden und 117 Minuten. Ein großer Teil der Laufzeit wird jedoch für die Analyse der Laufzeiten benötigt und nicht für das eigentliche Scheduling der Instruktionen. Für das Superblock-basierte und Trace-basierte Scheduling müssen die Analysen der WCET und BCET nach dem Scheduling eines jeden Superblock oder Traces wiederholt werden, was bei dem lokalen Scheduling nicht erforderlich ist, da das lokale Scheduling keine gezielte Optimierung des WCEPs durchführt und daher eine Änderung des WCEPs nicht relevant ist.

Zusätzlich zur Evaluation des evolutionären Algorithmus aus Kapitel 3 und des Instruction-Schedulers wurde eine Evaluation aus der Kombination dieser beiden Optimierungen durchgeführt. Der Ablauf der Kombination ähnelt dem Ablauf des evolutionären Algorithmus stark. Es wird wie in Abschnitt 3.3 beschrieben der gleiche evolutionäre Algorithmus verwendet. Zusätzlich zur normalen Bewertung der Individuen wird jedoch noch auf der Konfiguration eines jeden Individuums der Instruction-Scheduler ausgeführt. Da der Rechenaufwand des evolutionären Algorithmus und des Instruction-Schedulers hoch ist, wurde die Evaluation auf zwei Taskmengen beschränkt. Als Verfahren für den Instruction-Scheduler wurde das lokale Scheduling gewählt, da die Evaluation des Instruction-Schedulers gezeigt hat, dass durch die Verwendung von globalen Schedulingverfahren in der Regel kein weiterer Gewinn erzielt werden konnte. Als Taskmengen für die Optimierungen wurde das Taskbündel *package-006* und *package-076* gewählt, da für diese Taskmengen

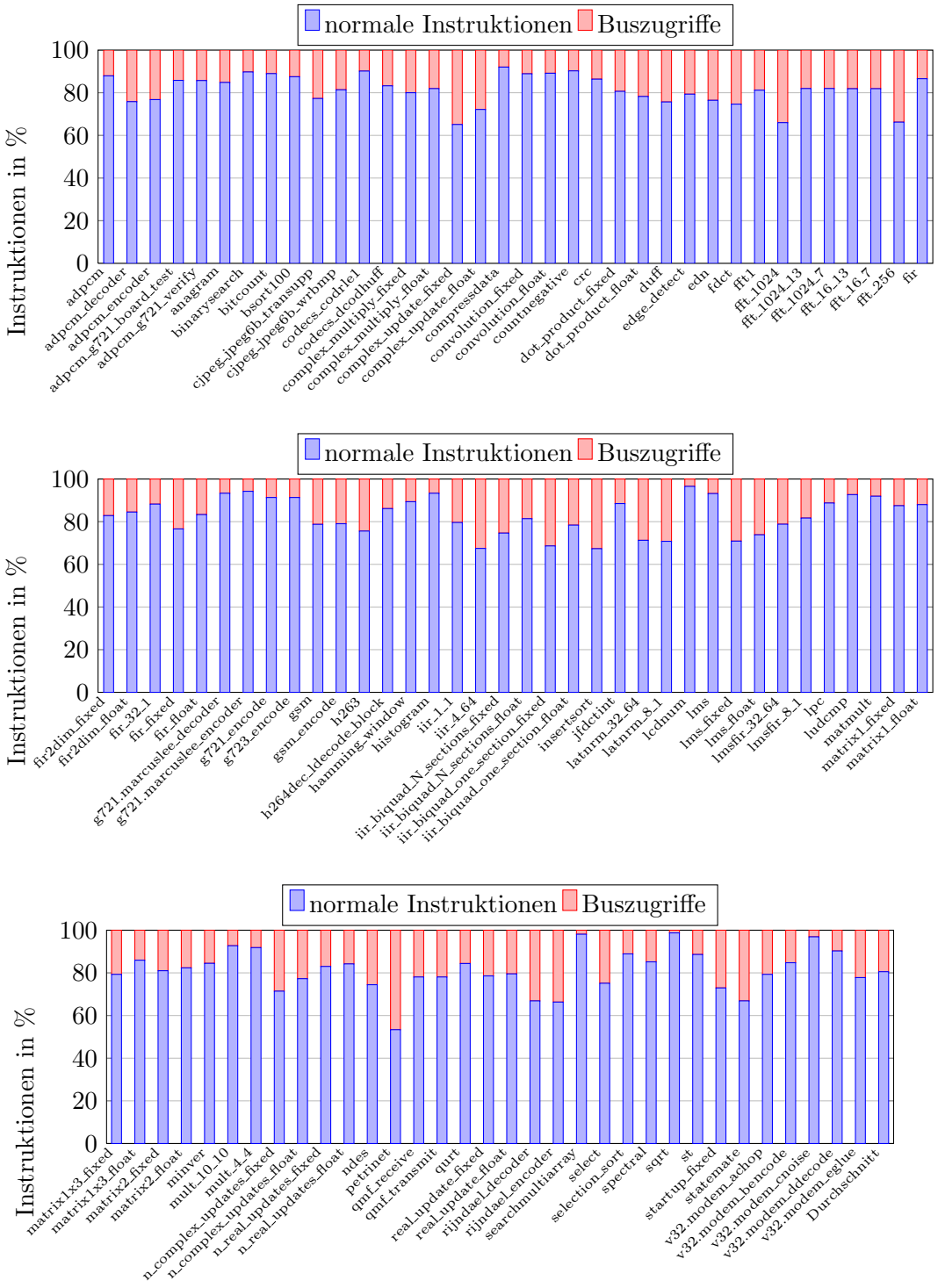


Abbildung 4.25: Verhältnis von Instruktionen mit und ohne Zugriff auf gemeinsamen Bus

durch den evolutionären Algorithmus im Vergleich zur Standardkonfiguration des TDMA-Verfahren eine Reduktion der WCET erreicht werden konnte. Die Tests wurden für eine Hardwarekonfiguration mit vier Prozessorkernen ausgeführt.

Durch Tabelle 4.1 sind die Ergebnisse der Kombination der Optimierungen für die Taskmenge *package-006* aufgezeigt. Es ist zu erkennen, dass der evolutionäre Algorithmus (gekennzeichnet mit: EA) eine geringe Reduktion der WCET gegenüber der TDMA-Standardkonfiguration erzielen konnte. Bei der optimierten Konfiguration handelt es sich um eine TDMA-Konfiguration mit vier Slots, die jeweils eine Slotlänge von 3 Taktzyklen besitzen. Dies entspricht der Standardkonfiguration, jedoch wurde durch evolutionären Algorithmus eine andere Ordnung der Zuweisung von Prozessorkern auf Slotnummer bestimmt. Anstelle der Standardzuweisung von Prozessorkernnummer auf Slotnummer wurde für die WCET-Reduktion die Zuweisung $S_0 = C_0$, $S_1 = C_3$, $S_2 = C_2$ und $S_3 = C_1$ gewählt. Durch das Instruction-Scheduling wurde keine weitere Verbesserung der WCET erzielt. Der Instruction-Scheduler hat eventuell die bereits gute Verteilung auf die Slots, durch eine Neuanordnung der Instruktionen, wieder verschlechtert, sodass die WCET nicht reduziert wird.

Die Ergebnisse der Kombination der Optimierungen für die Taskmenge *package-076* sind durch Tabelle 4.2 angegeben. Es ist zu erkennen, dass der evolutionäre Algorithmus die Gesamtkosten der Taskmenge reduzieren konnte im Vergleich zur fairen Arbitrierung und zur Standardkonfiguration des TDMA-Verfahrens, jedoch durch Steigerung der WCET für drei der vier Tasks. Dieses Verhalten ist ähnlich zur in Abschnitt 3.4 vorgestellten Konfiguration für die Taskmenge *package-083*. Für diese Konfiguration war es für den Instruction-Scheduler nicht möglich, eine weitere Reduktion der WCET zu erreichen. Für zwei der vier Tasks hat sich die WCET durch das Instruction-Scheduling verschlechtert, was jedoch durch die Wiederherstellung der LLIR verhindert wird. Auch in diesem Fall ist davon auszugehen, dass das Instruction-Scheduling gegen die Konfigurationen des evolutionären Algorithmus arbeitet. Das Ziel des evolutionären Algorithmus ist es, anwendungsspezifisch gute Konfigurationen für den gemeinsamen Bus zu finden. Das Instruction-Scheduling hingegen, soll die vorgegebene Konfiguration möglichst gut ausnutzen. Die Ergebnisse aus Tabelle 4.1 und Tabelle 4.2 zeigen, dass diese Ziele nicht gut harmonieren. Allerdings wurde

Kern	Taskname	Fair	TDMA-Standard	EA	EA+IS
C_1	binarysearch	394	339	339	339
C_2	real_update_float	328	318	312	312
C_3	matrix1x3_fixed	841	729	729	729
C_4	dot_product_float	458	432	426	426
$\sum$		2021	1818	1806	1806

Tabelle 4.1: WCET-Vergleich der Optimierungen für Taskmenge *package-006*

Kern	Taskname	Fair	TDMA-Standard	EA	EA+IS
C_1	matmult	739973	582615	704184	704184
C_2	codecs_dcodhuff	1847870	1615750	3841110	3841110
C_3	lmsfir_32_64	370827	343497	381714	381714
C_4	fft_256	55915700	48944200	55282000	55282000
$\sum$		58874370	51474592	47883918	47883918

Tabelle 4.2: WCET-Vergleich der Optimierungen für Taskmenge *package-076*

für die kombinierte Evaluation der beiden Optimierungsstrategien auf die Mechanismen, welche die Verschlechterung der WCET verhindern, verzichtet, sodass eine Auswertung des vollen Potenzials möglich ist.

5 Fazit

Die Ziele dieser Arbeit waren, das Optimierungspotenzial verschiedener Optimierungen bezüglich konkurrierender Zugriffe auf einen gemeinsamen Bus zu untersuchen. Dazu wurde in Abschnitt 2.1 beschrieben, welche Hardwareplattform für die Optimierungen als Basis dienen soll.

Das erste Ziel der Arbeit war, zu erforschen, ob durch die Suche von möglichst guten Konfigurationen der verschiedenen Busarbitrierungsverfahren, eine Reduktion der WCET erreicht werden kann. Zur Bestimmung dieser Konfigurationen wurde in Kapitel 3 eine Möglichkeit dargestellt. Es wurde mithilfe eines evolutionären Algorithmus eine Suche nach guten Konfigurationen durchgeführt. Dieses Verfahren wurde in Abschnitt 3.4 einer Evaluation unterzogen, welche gezeigt hat, dass die Konfigurationen der Busarbitrierungsverfahren Potenzial für weitere Optimierungen bieten. Es wurden Tests für verschiedene Hardwareplattformen und Taskmengen durchgeführt, sodass gezeigt werden konnte, dass eine Reduktion der WCET je nach Vergleichswert und Hardwarekonfiguration von bis zu zirka 80 % möglich ist. Die durchschnittliche Verbesserung zur Standardkonfiguration für das TDMA-Verfahren liegt bei zirka 3,5 % über alle getesteten Hardwarekonfigurationen. Es konnte auch gezeigt werden, dass nicht jede Konfiguration und Taskmenge das gleiche Optimierungspotenzial besitzt, sodass davon auszugehen ist, dass keine optimale Konfiguration der Busarbitrierungsverfahren für alle Taskmengen existiert und jede Taskmenge eine individuelle Konfiguration benötigt. Die Evaluation des verwendeten evolutionären Algorithmus hat gezeigt, dass der praktische Einsatz einer solchen Optimierung jedoch fraglich ist, da die Optimierung einen sehr hohen Rechenaufwand aufweist, was zur Folge hat, dass die Optimierung für manche Taskmengen eine Laufzeit von bis zu 33 Stunden benötigt. Dadurch eignet sich das Verfahren beispielsweise für Anwendungen, bei denen die Zeit der Übersetzung nicht relevant ist, jedoch eine geringe WCET wichtig ist. Dieser hohe Rechenaufwand ist durch die Analyse zur Bestimmung der WCET und BCET geschuldet, welche für jede Konfiguration, die durch den evolutionären Algorithmus bestimmt wird, wiederholt werden muss.

Des Weiteren war ein Ziel dieser Arbeit zu ermitteln, ob sich die WCET durch den Einsatz eines Instruction-Schedulers, welcher die Gegebenheiten eines gemeinsamen Busses ausnutzt, weiter reduzieren lässt. Dazu wurde in Kapitel 4 ein Verfahren zur Neuanordnung der Maschineninstruktionen vorgestellt. Dieses Verfahren bestimmt für jede Maschineninstruktionen eine Priorität, welche beschreibt, wann eine Instruktion am besten ausgeführt

wird. Dazu ist beachten, dass eine Instruktion häufig abhängig von anderen Instruktionen ist, was in Abschnitt 4.1 für die verwendete Hardwareplattform beschrieben wurde. Zur Bestimmung der Prioritäten der Instruktionen wurden in Abschnitt 4.3 verschiedene Heuristiken vorgestellt, welche unterschiedliche Informationen über den gemeinsamen Bus verwenden. So wurden in Unterabschnitt 4.3.1 zwei Heuristiken präsentiert, welche nur die Parameter des Busarbitrierungsverfahrens nutzen und keine weiteren Informationen über das Zeitverhalten der Busarbitrierung. Mit der TDMA-Offset-basierten Heuristik wurde in Unterabschnitt 4.3.2 eine Heuristik vorgestellt, welche explizit die Zeitinformationen der WCET-Analyse verwendet und versucht, die aktuelle Position innerhalb des TDMA- und PD-Schedules zu verwenden. Zur eigentlichen Neuanordnung wurden verschiedene Techniken präsentiert, welche unterschiedliche Freiheitsgerade bei der Neuanordnung bieten. Die Evaluation des Instruction-Schedulings hat gezeigt, dass sich die verschiedenen Tasks unterschiedlich gut optimieren lassen. Manche Tasks konnten durch eine Neuanordnung der Instruktionen nicht verbessert werden. Andere konnten je nach verwendeter Busarbitrierung und verwendetem Scheduling-Verfahren um bis zu 80 % verbessert werden. Es konnten im Durchschnitt, je nach getesteter Hardwarekonfiguration, eine Verbesserung der WCET von ungefähr 4 % erreicht werden. Die Laufzeit der Optimierung lag mit einer maximalen Dauer von 110 Minuten im Rahmen einer praktischen Nutzung.

5.1 Ausblick

Die Evaluation des evolutionären Algorithmus hat gezeigt, dass die Optimierung sehr zeitaufwändig ist und viel Rechenleistung für die Analyse der WCET und BCET benötigt. Dabei werden häufig Konfigurationen überprüft, welche eine sehr hohe WCET und BCET aufweisen und daher nicht zur Verbesserung der WCET und BCET beitragen. Bei den Konfigurationen handelt es sich beispielsweise um eine Konfiguration mit deutlich mehr Slots als Prozessorkernen, was für die WCET-Analyse zu einer sehr hohen Überabschätzung führt. Für weitere Arbeiten in der Zukunft ist denkbar, dass versucht wird, diese Konfigurationen, die zu keiner Reduktion der WCET führen zu vermeiden. Dazu ist es denkbar, dass die zu optimierende Taskmenge auf mögliche Buszugriffe analysiert wird und so versucht wird eine Struktur in den Buszugriffen zu erkennen, mit der es möglich ist, eine Richtung für den evolutionären Algorithmus vorzugeben, sodass der Suchraum eingeschränkt werden kann, was eine Reduktion der Laufzeit zur Folge haben könnte und eine weitere Steigerung der Güte der Optimierung bringen könnte. Des Weiteren könnte zur Realisierung des evolutionären Algorithmus eventuell ein anderer Selektor gewählt werden, welcher nicht exklusiv auf Basis der Paretodominanz agiert. Beispielsweise existieren Verfahren, welche durch das Erstellen eines Modells versuchen, die Ergebnisse einer Konfiguration vorherzusagen, was eine Einsparung bei den WCET- und BCET-Analysen mit sich bringen könnte.

Weitere Arbeiten sind auch im Bereich des Instruction-Schedulers denkbar. Aktuell wird durch die Heuristiken exklusiv das Verhalten des gemeinsamen Bus betrachtet. Allerdings ist dies nicht der einzige Faktor für die WCET, sodass vorstellbar ist, dass andere Heuristiken zur Auswahl möglicherweise ebenfalls eine Reduktion der WCET erzielen können. Zusätzlich gibt es weitere Scheduling-Verfahren für Instruktionen. So gibt es ein Verfahren, welches Basisblöcke in einer Baumstruktur anordnet und so dem List-Scheduler weitere Freiheiten bietet [RC06].

Abbildungsverzeichnis

Literaturverzeichnis

[ALSU06] Aho, Alfred V., Monica S. Lam, Ravi Sethi und Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006, ISBN 0321486811.

[ARM05] ARM Holdings: *ARM Architecture Reference Manual*. 2005.

[BLTZ03] Bleuler, Stefan, Marco Laumanns, Lothar Thiele und Eckart Zitzler: *Pisa - a platform and programming language independent interface for search algorithms*. 2632:494–508, 2003.

[BYPC12] Bak, Stanley, Gang Yao, Rodolfo Pellizzoni und Marco Caccamo: *Memory-aware scheduling of multicore task sets for real-time systems*. In: *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, Seiten 300–309, 2012.

[CoM13] *Synopsys CoMET*, Oktober 2013. `http://www.synopsys.com/Systems/VirtualPrototyping/Pages/CoMET-METeor.aspx`.

[Fis81] Fisher, Joseph A.: *Trace scheduling: A technique for global microcode compaction*. Computers, IEEE Transactions on, C-30(7):478–490, 1981, ISSN 0018-9340.

[FL10] Falk, Heiko und Paul Lokuciejewski: *A compiler framework for the reduction of worst-case execution times*. Journal on Real-Time Systems, 46(2):251–300, Oktober 2010. DOI 10.1007/s11241-010-9101-x.

[Har13] Harde, Tim: *Vergleichende Studie von Arbitrierungsverfahren für Kommunikationsstrukturen in eingebetteten Multicoresystemen*. Bachelorarbeit, Technische Universität Dortmund, 2013.

[HMC+93] Hwu, Wen-mei W., Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm und Daniel M. Lavery: *The superblock: An effective technique for vliw and superscalar compilation*. The Journal of Supercomputing, 7:229–248, 1993.

[ICD13] *ICD-C Compiler framework*, Oktober 2013. `http://www.icd.de/index.php/de/eingebettete-systeme/icd-c-compiler/icd-c`.

[Inf03] Infineon Technologies: *TriCore Compiler Writer's Guide*, 2003.

[KFM+11] Kelter, Timon, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay und Abhik Roychoudhury: *Bus-aware multicore wcet analysis through tdma offset bounds.* In: *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, Seiten 3–12, Porto / Portugal, Juli 2011.

[KFM+13] Kelter, Timon, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay und Abhik Roychoudhury: *Static analysis of multi-core tdma resource arbitration delays.* Real-Time Systems, 2013.

[KHMF13] Kelter, Timon, Tim Harde, Peter Marwedel und Heiko Falk: *Evaluation of resource arbitration methods for multi-core real-time systems.* In: Maiza, Claire (Herausgeber): *Proceedings of the 13th International Workshop on Worst-Case Execution Time Analysis (WCET)*, Paris, France, Juli 2013.

[KKP+11] Kirner, Raimund, Jens Knoop, Adrian Prantl, Markus Schordan und Albrecht Kadlec: *Beyond loop bounds: comparing annotation languages for worst-case execution time analysis.* Software & Systems Modeling, 10(3):411–437, 2011, ISSN 1619-1366.

[LPMS97] Lee, Chunho, Miodrag Potkonjak und William H. Mangione-Smith: *Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems.* In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, Seiten 330–335, Washington, DC, USA, 1997. IEEE Computer Society, ISBN 0-8186-7977-8.

[MRT13] *Mälardalen Real-Time Research Centre Benchmarks*, Oktober 2013. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[RC06] Rosier, Michael C. und Thomas M. Conte: *Treegion instruction scheduling in gcc.* In: *Proceedings of the 2006 GCC Developers Summit*, April 2006.

[RNE+11] Rosen, Jakob, Carl-Frederik Neikter, Petru Eles, Zebo Peng, Paolo Burgio und Luca Benini: *Bus access design for combined worst and average case execution time optimization of predictable real-time applications on multiprocessor systems-on-chip.* In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, Seiten 291–301, 2011.

[Smo10] Smolarczyk, Andre: *Instruction Scheduling-Verfahren zur Minimierung der WCET.* Diplomarbeit, Technische Universtität Dortmund, Mai 2010.

[SQL13] *About SQLite*, November 2013. `http://www.sqlite.org/about.html`.

[UTD13] *UTDSP Benchmark Suite* , Oktober 2013. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`.

[Wei07] Weicker, Karsten: *Evolutionäre Algorithmen (Leitfäden der Informatik) (German Edition)*. Vieweg+Teubner Verlag, 2007, ISBN 3835102192.

[WT06] Wandeler, Ernesto und Lothar Thiele: *Optimal tdma time slot and cycle length allocation for hard real-time systems.* In: *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, Seiten 479–484, Piscataway, NJ, USA, 2006. IEEE Press, ISBN 0-7803-9451-8.

[ZLT01] Zitzler, Eckart, Marco Laumanns und Lothar Thiele: *Spea2: Improving the strength pareto evolutionary algorithm.* Technischer Bericht, 2001.