

Projektarbeit:

**Simulation der Einbettung von
Anwendungen mit Bindungs- und
Routinganforderungen in
Mehrprozessorsystemen**

von

Daniel Jäger

Matrikel-Nr.: 21448104

Betreuung:

Prof. Dr.-Ing. Jürgen Teich

Dr.-Ing. Stefan Wildermann

31. März 2015

Inhaltsverzeichnis

1	Motivation	1
2	Problemspezifikation	3
2.1	Selbsteinbettung in Mehrprozessorsysteme	3
2.2	Constraint–Satisfaction–Problem (CSP)	6
2.3	Min-Conflicts-Embedder	6
3	Model zur Lösung des CSP-Problems	9
3.1	Implementierung des Min-Conflicts-Embedder	9
3.2	Klassendiagramm zum Lösen der Bindungs- und Routinganforderungen .	10
3.2.1	Bindungsanforderungen	10
3.2.2	Routinganforderungen	13
3.2.3	Hinzufügen von Anforderungen	14
3.2.4	Überprüfen von Anforderungen	15
3.2.5	Zuordnen von Anforderungen	15
4	Evaluierung	17
4.1	Implementierung und Testumgebungen	17
4.1.1	Testumgebung 1	17
4.1.2	Testumgebung 2	18
4.2	Vergleich zwischen Backtracking mit und ohne Forward-Checking	18
4.3	Vergleich der Variablenordnungsverfahren	19
4.4	Vergleich zwischen Min-Conflicts und Backtracking	23
5	Zusammenfassung	27
	Literatur	29

Aufgabenstellung zur Projektarbeit

Ziel dieser Arbeit ist es, einen Laufzeitmechanismus für hybride Einbettungsverfahren in heterogenen, NoC-basierten Mehrprozessorsystemen umzusetzen. Dazu müssen einerseits die grundlegenden Datenstrukturen und Algorithmen bereitgestellt werden, siehe [Jä14]. Diese sollen dann in der PGAS-Programmiersprache X10 [X1014] umgesetzt werden. Letztendlich soll das umgesetzte Verfahren durch Einsatz des InvadeSIM – Simulators [RHT12] simuliert werden, und ausgewertet werden, wie es sich in einem realen Vielkernsystem verhalten würden. Hierbei soll vor allem der mit dem Constraint-Solving verbundene Rechenaufwand quantifiziert werden.

Im Rahmen der Arbeit sind folgende Arbeitsschritte zu tätigen:

- Einarbeitung in die Thematik und die grundlegenden Literatur.
- Festlegung der Datenstrukturen und Algorithmen. Für die Arbeit ausreichend ist die Umsetzung der Min-Conflict-Heuristik (siehe [Jä14]).
- Deren Umsetzung in der Programmiersprache X10.
- Simulation und experimentelle Auswertung des Laufzeitverhaltens mit Hilfe von InvadeSIM [RHT12].
- Erstellung einer schriftlichen Projektarbeit und der Dokumentation aller Programme. Archivierung der relevanten Daten auf einer CD/DVD.

1 Motivation

Zukünftige Mehrprozessorsysteme bestehen aus einer Vielzahl heterogener Ressourcen [1], in denen die Kommunikation zwischen Ressourcen durch Network-on-Chips (NoCs) ermöglicht wird (siehe z. B. [2]). Diese Technologie unterstützt Anwendungsszenarien, in denen eine sehr große Zahl an Programmen dynamisch starten und terminieren. Allerdings führt der Einsatz von immer mehr gemeinsam genutzten, heterogenen Hardwareressourcen dazu, dass die Vorhersagbarkeit nichtfunktionaler Eigenschaften der Ausführung eines Programms erschwert wird. Dies betrifft z. B. den erwarteten Durchsatz, die Echtzeit Fähigkeit, Zuverlässigkeits- oder Sicherheitseigenschaften.

Neue Ansätze wie [3, 4] schlagen daher hybride Verfahren zur Einbettung von Anwendungen (engl. hybrid application mapping) vor. Im Fokus stehen Programme zur Bild- und Signalverarbeitung, die nach dem Starten Daten periodisch verarbeiten. Zur Entwurfszeit wird im Rahmen einer Entwurfsraumexploration analysiert, welchen Einfluss verschiedene Allokationen heterogener Ressourcen für die Programmausführung auf deren nichtfunktionale Eigenschaften haben. Die Exploration evaluiert und optimiert dabei eine Vielzahl solcher Implementierungsalternativen, wobei nur Alternativen beibehalten werden, die bezüglich ihrer Zielgrößen nicht durch andere dominiert werden (sog. Betrieb- spunkte [5]). Die Grundidee dieses Vorgehens ist, dass die ermittelten nichtfunktionalen Eigenschaften einer Programmausführung garantiert werden können, wenn die geforderten Ressourcen zur Laufzeit bereitgestellt werden. Hierbei müssen nach [4] bestimmte Nebenbedingungen (Constraints) eingehalten werden: Einerseits sind dies Bindungsanforderungen bezüglich der Ressourcentypen, auf die die Anwendungstasks gebunden werden können. Andererseits bestehen Routinganforderungen für die Realisierung der Datenabhängigkeiten zwischen Anwendungstasks, z. B. bezüglich der maximalen Anzahl an Sprüngen (engl. hops) durch das NoC oder der benötigten Bandbreite.

2 Problemspezifikation

In diesem Kapitel wird die Selbsteinbettung von Taskgraphen in Mehrprozessorsystemen vorgestellt. Am Beispiel von einem Network-on-Chip (NoC) werden dessen Architektur und ihre Besonderheiten näher betrachtet. Danach wird das Constraint-Satisfaction-Problem (CSP) definiert und die Bedingungen (engl. *Constraints*) für die Architektur festgelegt.

2.1 Selbsteinbettung in Mehrprozessorsysteme

In einem Mehrprozessorsystem [HN11] arbeiten mehrere Zentralprozessoren zusammen. Mehrprozessorsysteme erlauben die Verteilung der Aufträge auf mehrere physische Zentralprozessoren und helfen somit den Durchsatz zu erhöhen. Das Betriebssystem verteilt die Anwendungen (engl. *Application*) und deren Teilaufgaben (engl. *Task*) an die Zentralprozessoren.

Bei der betrachteten Architektur (Abbildung 2.1) handelt es sich um ein heterogenes Mehrprozessorsystem. Kennzeichnend dafür ist die Verwendung verschiedener Typen von Zentralprozessoren. Diese werden im Weiteren Kacheln genannt. Die einzelnen Kacheln sind in einem Netzwerk – einem sogenannten Network-on-Chip (NoC) [WWT11] [Fra11] – miteinander verbunden. Das Netzwerk ist zu einem regelmäßigen rechteckigen Gitter von Kacheln geordnet, welches N hoch und M breit ist. Die direktbenachbarten Kacheln sind mit gerichteten Links $l_i \in L$ (Abbildung 2.2) in beide Richtungen verbunden. Die Kacheln $k \in K$ sind durch die Koordinaten (x,y) definiert:

Eine Kachel kann einen Task ausführen, wenn alle Constraints (Abschnitt ??) erfüllt sind. Einer dieser Randbedingungen ist der Max-Hop-Constraint. Diese besagt, dass zwei miteinander agierenden Task eine Manhattan-Distanz d_m (Formel 2.1) nicht überschreiten dürfen.

$$d_m(k_1, k_2) = |k_1.x - k_2.x| + |k_1.y - k_2.y| \quad (2.1)$$

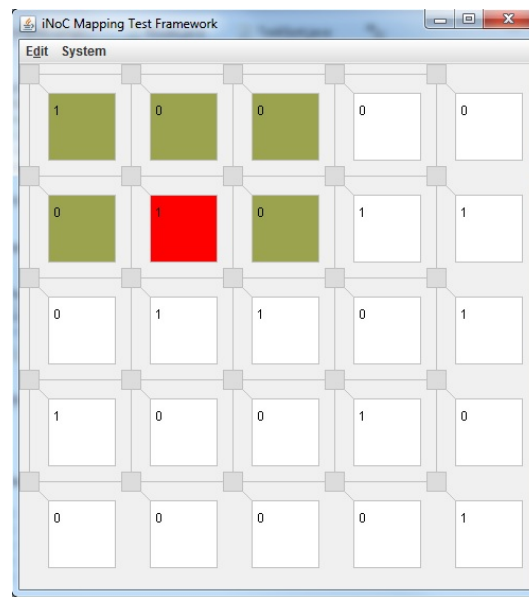


Abbildung 2.1: Simulation eines heterogenes Network-On-Chip: Die belegten Kacheln sind mit grün und die defekten Kacheln mit rot markiert. In jeder Kachel steht der Ressourcentyp (0 oder 1).

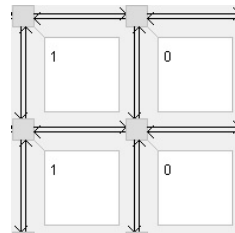


Abbildung 2.2: Die benachbarten Kacheln sind mit gerichteten Links in beide Richtungen verbunden.

Jede Anwendung, die aus mehreren Teilaufgaben, den sogenannten Tasks, besteht, besitzt einen Taskgraphen (Abbildung 2.3). Dieser wird zur Entwurfszeit des Programms erstellt und stellt die Anforderungen, die für eine erfolgreiche Einbettung notwendig sind, graphisch dar.

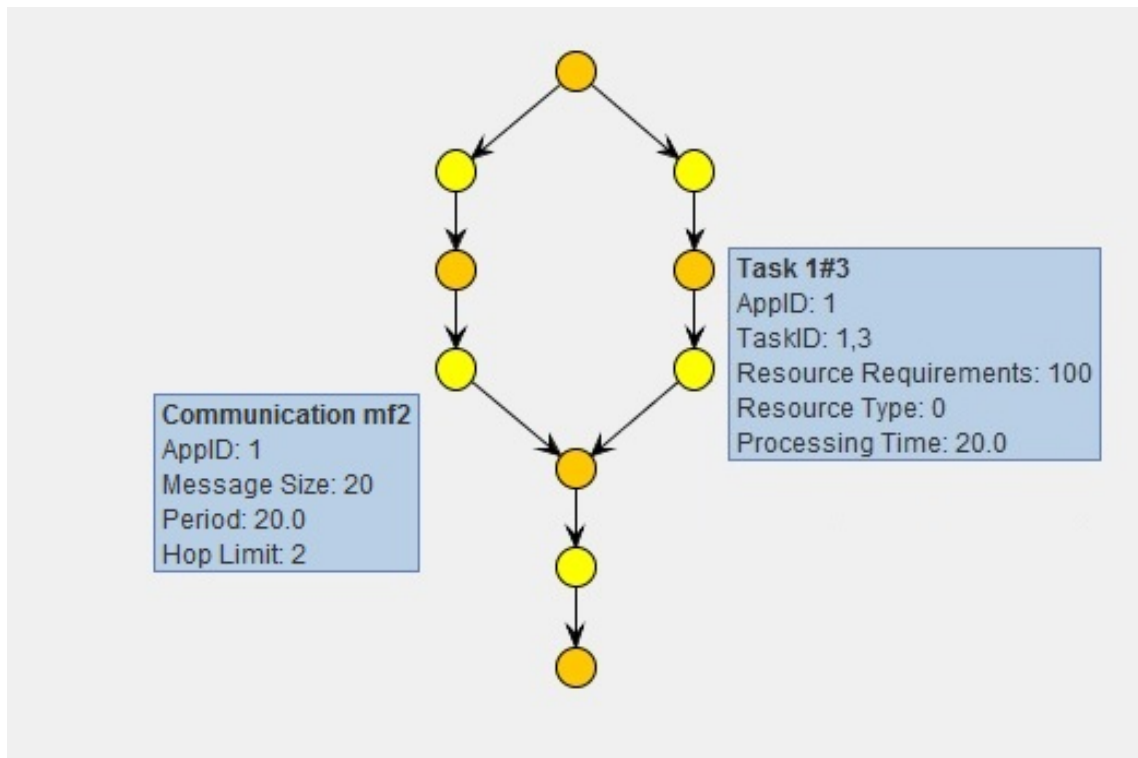


Abbildung 2.3: Der Taskgraph besteht aus orangefarbenen Task- und gelbgefärbten Kommunikationsknoten

Eine Applikation besteht aus miteinander verbundenen Taskknoten $t \in T$ und Kommunikationsknoten $c \in C$. Es sind nur Kanten zwischen diesen beiden Knotentypen erlaubt. Außerdem besitzt jeder Kommunikationsknoten **genau** eine Eingangskante und eine Ausgangskante. Dem Task- und dem Kommunikationsknoten werden verschiedenartige Bedingungen übergeben, die zu erfüllen sind. Die verschiedenen Typen von Constraints sind je nach Anwendungsgebiet erweiterbar.

- Taskknoten $t_i \in T$
 - *TypeConstraint*:
Der Task benötigt einen bestimmten Kacheltyp
 - *TaskWorkloadConstraint*:
Der Task benötigt die Kachel zu einem gewissen Prozentsatz
 - ...
- Kommunikationsknoten $c_i \in C$:
 - *BandwidthConstraint*:
Jeder Link auf der Route zwischen den eingebetteten Tasks muss eine bestimmte Bandbreite für die Kommunikation freihaben.

- *MaxHopConstraint*:
Maximale Manhattan-Distanz zwischen zwei Tasks, die mit einer Kommunikation in Verbindung stehen.
- ...

Das Ziel des Algorithmus ist es, für jeden Task eine Kachel zu finden, so dass alle Anforderungen des Taskgraphen erfüllt sind. Ist dies nicht möglich, soll der Algorithmus sich beenden und einen Fehlschlag der Einbettung melden. Im Abschnitt wird mit dem Min-Conflicts-Embedder ein stochastischer Ansatz vorgestellt. Es gibt auch die Möglichkeit, die Problemstellung mit dem systematischen Ansatz zu lösen. Dies wird in der folgenden Quelle näher erläutert.

abschnitt
hin-
zufügen

2.2 Constraint–Satisfaction–Problem (CSP)

Das Problem der Taskeinbettung lässt sich als Constraint-Satisfaction-Problem (CSP) formulieren. Ein CSP wird mit dem Tripel (V, D, B) (Definition ??) beschrieben. Ziel des Constraint-Satisfaction-Problems ist es, eine global konsistente Wertebelegung für alle Constraint-Variablen zu finden. Das bedeutet, es müssen alle Bedingungen erfüllt sein. Eine Bedingung ist genau dann erfüllt (Definition ??), wenn die Wertezuweisungen den Anforderungen der Bedingung entspricht [GRJ00]. Die Bedingungen lassen sich wie in [Jä14] umformulieren.

quelle

2.3 Min-Conflicts-Embedder

Die Min–Conflicts–Heuristik ist in der Praxis eine häufig eingesetzte Methode zum Lösen von CSPs. Es handelt sich nicht um ein systematisches Verfahren wie beim Backtracking, sondern um ein Stochastisches. Als Startpunkt wird eine zufällige Belegung der Variablen erzeugt. Falls dies keine Lösung des CSPs ist, wovon auszugehen ist, wird eine konfliktzeugende Variable zufällig ausgewählt. Dieser wird ein Wert zugewiesen, der weniger Randbedingungen verletzt als der vorherige.

Wie bei anderen stochastischen Methoden besteht auch hier die Gefahr, dass das System in einem lokalen Minimum terminiert. Falls das eintritt, muss eine erneute zufällige Belegung der Variablen erzeugt werden und eine neue Runde beginnt. Eine Tabu-Liste kann dabei helfen, Belegungen, die zu einem lokalen Minimum geführt haben, nicht mehr zu wählen. Nach einer vorher festgelegten Anzahl von Runden beendet sich die Heuristik und das Scheitern der Einplanung wird ausgegeben. So kann es vorkommen, dass die Min–Conflicts-Heuristik keine konsistente Lösung findet, obwohl es eine im Constraint–System gibt.

Oftmals wird die Heuristik verwendet, wenn sich das Constraint–System geringfügig

verändert hat und für das vorherige System schon eine Lösung vorhanden ist. Aus diesem Grund wird der Algorithmus auch als "heuristisches Reparieren" [MJPL90] bezeichnet.

3 Model zur Lösung des CSP-Problems

In diesem Kapitel wird zuerst die Implementierung des Min-Conflicts-Embedder (Abschnitt 3.1) mit seinen einzelnen Methoden vorgestellt. Anschließend werden anhand von Klassendiagrammen die Bindungs- (Abschnitt 3.2.1) und Routinganforderungen (Abschnitt 3.2.2) näher betrachtet. Dabei wird die innere Struktur betrachtet, die diese Anforderungen hinzufügen, testen und lösen.

3.1 Implementierung des Min-Conflicts-Embedder

Die Implementierung des Min-Conflicts-Embedder ist anhand des Aktivitätsdiagramm (Abbildung 3.1) graphisch dargestellt. Zuerst wird die Schleifenvariable der äußeren Schleife mit Null initialisiert und die Methode **mapTaskRandomly** aufgerufen. In **mapTaskRandomly** werden alle Tasks einer zufällig gewählten Kachel, die im Programmcode Unit genannt wird, zugewiesen. Es werden dabei Verletzungen der Constraints ignoriert. Anschließend wird der inneren Schleifenvariable k dem Wert Null zugewiesen und die Funktion **findRandomConflictingTask** aufgerufen. Diese Funktion wählt einen Task aus der Menge C der Tasks aus, die mindestens einen Constraint verletzen. Falls es keinen Task gibt (**conflictingTask==null**) und somit die Menge C leer ist, wurde eine Lösung gefunden (**success**). Ansonsten wird überprüft, ob die maximale Anzahl an Schleifendurchgängen (**kMax**) erreicht wurde. Wenn dies nicht der Fall ist, ruft das Programm die Funktion **minConflicts** auf. **minConflicts** bekommt als Parameter den **conflictingTask** übergeben und versucht für ihn eine geeignete Kachel zu finden, die zu keiner Verletzung eine Bindungs- bzw Routingbedingung führt. Falls es eine derartige Kachel nicht gibt, wird schrittweise der **MaxHopConstraint** gelockert. Das heißt, dass die Manhattendistanz schrittweise um eins erhöht wird, bis der Algorithmus eine geeignete Kachel gefunden hat. Ist dies der Fall, wird k inkrementiert und die Funktion **findRandomConflictingTask** wiederum aufgerufen.

Hat k den Wert **kMax** erreicht, so ist der Durchgang beendet und alle Tasks werden ihrer Kachel entzogen (**unmapTasks**). Die Funktion **mapTaskRandomly** wird aufgerufen und ein neuer Durchgang beginnt. Falls nach **jMax** Durchgängen noch keine Lösung gefunden wurde, beendet sich das Programm (**fail**).

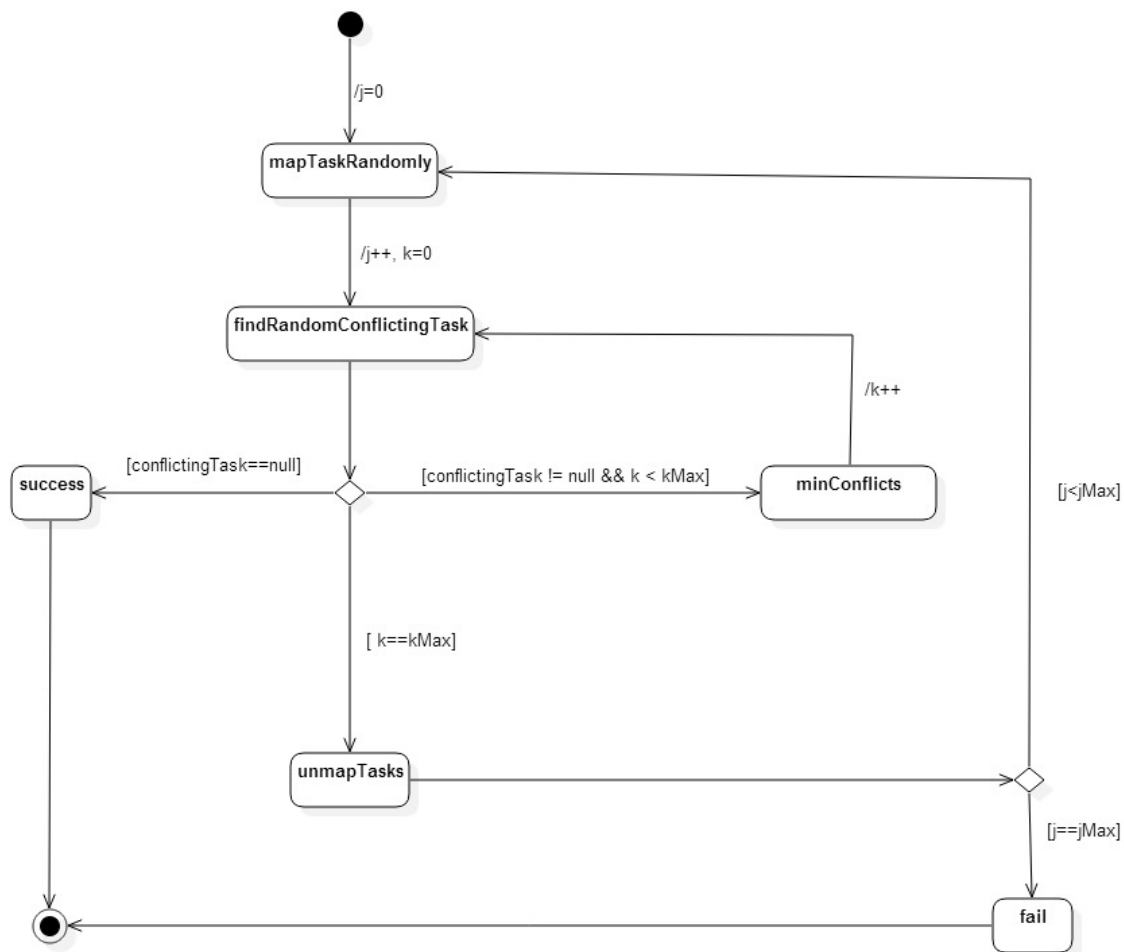


Abbildung 3.1: Das Aktivitätsdiagramm zur Min-Conflicts-Implementierung

3.2 Klassendiagramm zum Lösen der Bindungs- und Routinganforderungen


3.2.1 Bindungsanforderungen

Die Abbildung 3.2 zeigt das Klassendiagramm zum Lösen von Bindungsanforderungen. Die Klasse `Task` speichert in einer Liste alle Bindungsanforderungen (engl. `TaskConstraint`) ab und stellt folgende Funktionen zur Verfügung:

- *addTaskConstraint*
fügt ein Objekt einer abgeleiteten Klasse von `TaskConstraints` der Liste hinzu.
- *removeTaskConstraint*
löscht ein Objekt einer abgeleiteten Klasse von `TaskConstraint` aus der Liste

- *taskConstraintsAreSatisfied*
ruft für alle Objekte in der Liste die Funktion *isSatisfied* aus der Klasse *TaskConstraint* auf. *isSatisfied* überprüft, ob die jeweilige Bindungsanforderung erfüllt ist. Die Funktion gibt *True* zurück, wenn alle Bindungsanforderungen erfüllt sind.
- *numberOfFailingConstraint*
gibt die Anzahl der fehlgeschlagenen Bindungsanforderungen zurück, wenn man dem Task eine Unit *u* zuweist.
- *mapConstraints*
weist dem Task eine Unit zu. Alle Objekte, die sich in der Liste befinden, rufen die Methode *map* auf.
- *unmapConstraints*
entzieht dem Task die Unit. Alle Objekte, die sich in der Liste befinden, rufen die Methode *unmap* auf.

Um von der abstrakten Klasse *TaskConstraints* zu erben, müssen diese abstrakten Methoden implementiert werden:

- *isFeasible*
überprüft, ob bei Einbetten des Tasks zur Unit *u* die durch das Objekt repräsentierte Anforderungen erfüllt werden würde. Hierzu wird die Methode *check* mit der Objektvariable der zugehörigen Instanz der Klasse *UnitAttribute* aufgerufen. 
- *sSatisfied*
überprüft, ob der eingebettete Task, die durch das Objekt repräsentierte Anforderung, erfüllt. Dazu wird die Methode *check* der zugehörigen *UnitAttribute*-Klasse mit dem Übergabewert *null* aufgerufen.
- *map*
ruft von der zugehörigen *UnitAttribute* Unterklasse die Methode *update* auf. Der Parameterwert ist die eigene Objektvariable
- *unmap*
ruft von der zugehörigen *UnitAttribute* Unterklasse die Methode *update* auf. Der Parameterwert ist der negierte Wert der eigenen Objektvariable

Die Klasse *Unit* repräsentiert eine Kachel dem Network-on-Chip. Jede Instanz ist durch die Positionsangabe (x,y) eindeutig identifizierbar. Eine Instanz erhält Objekte der abgeleiteten Unterklassen von der abstrakten Klasse *UnitAttributes*. Die Klasse *Unit* befinden sich folgende Methoden:

- *addUnitAttribute*
fügt der Liste eine Objekt einer Unterklasse von *UnitAttribute* hinzu

- *removeUnitAttribute*
löscht ein Objekt einer Unterklasse von UnitAttribute aus der Liste

Die Kindklassen von UnitAttribute müssen die zwei abstrakten Methoden *check* und *update* implementieren. Jede Kindklasse von TaskConstraint wird dabei von genau einer Kindklasse von UnitAttribute verwendet und überprüft (*check*) oder aktualisiert (*update*) diese.

- *check*
überprüft, ob es möglich ist, dass Attribut zu aktualisieren.
- *update*
aktualisiert die Objektvariable des Attributs.

TypeAttribute ist ein Spezialfall. Hier kann die Objektvariable nicht, wie z. B. bei UnitWorkloadAttribute, aktualisiert werden, da sich der Ressourcentyp nicht während der Laufzeit verändert. Deshalb ruft die Funktion *update* die Funktion *check* auf und überprüft, ob der gewünschte Ressourcentyp vorliegt.

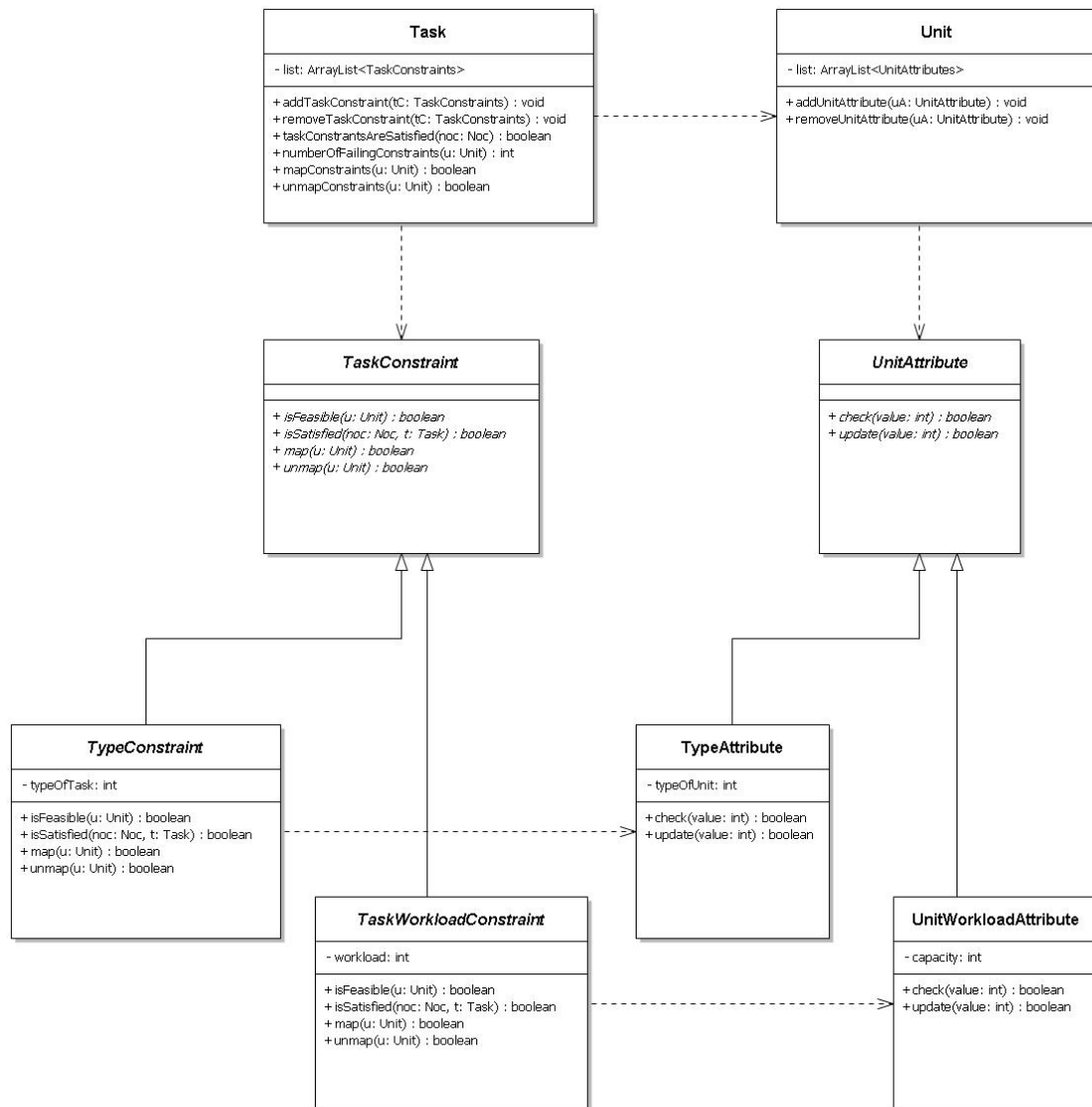


Abbildung 3.2: Klassendiagramm zum Lösen von Bindungsanforderungen

3.2.2 Routinganforderungen

Das Klassendiagramm zum Lösen von Routinganforderungen (siehe Abbildung 3.3) ist dem von Bindungsanforderungen sehr ähnlich. So sind die Methoden und deren Funktionalitäten gleich. Der Unterschied zu den Bindungsanforderungen besteht darin, dass nicht wie bei den Bindungsanforderungen die Attribute von nur einer Kachel (Unit) überprüft bzw. aktualisiert werden. Bei den Routinganforderungen werden die Attribute von mehreren Links, die zuvor mithilfe eines Routingalgorithmuses ermittelt wurden, nachgeprüft oder erneuert.

Die Routinganforderung MaxHopConstraint ist ein Spezialfall. Diese Anforderung benötigt

kein LinkAttribut wie z. B. BandwidthConstraint, da sie nur überprüft, ob die Anzahl der Links in der zuvor berechneten Route eine maximal Zahl (maxHops) nicht überschritten wird.

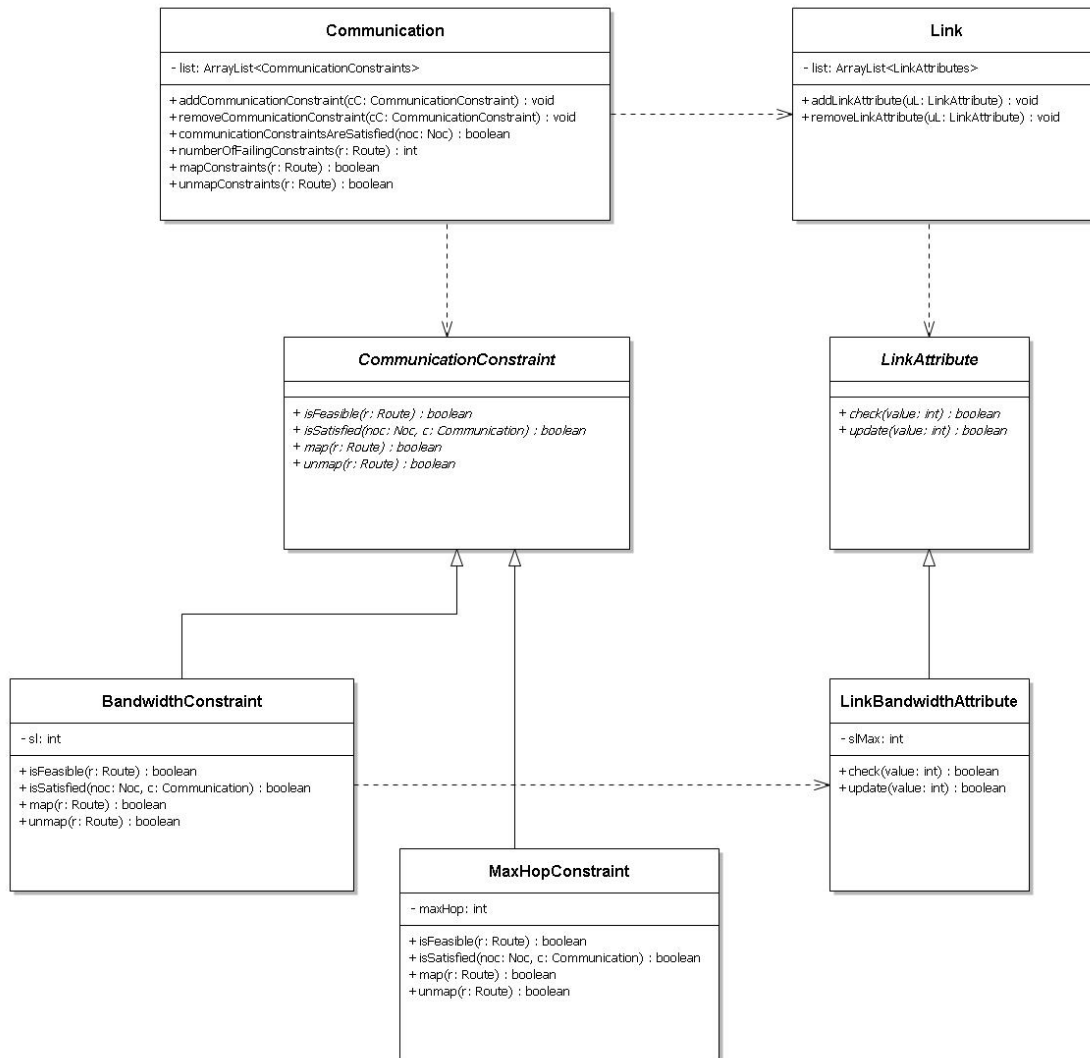


Abbildung 3.3: Klassendiagramm zum Lösen von Routinganforderungen

3.2.3 Hinzufügen von Anforderungen

Um Anforderungen hinzuzufügen, wird in der Klasse **Task** bzw. **Communication** die Funktion **addTaskConstraint** bzw. **addCommunicationConstraint** aufgerufen. Die Anforderung wird in der **ArrayList** **list** gespeichert. Falls eine Anforderung vom gleichen Typ schon vorhanden ist, wird diese durch die neue Anforderung ersetzt. Jede Anforderung (außer der **MaxHopsConstraint**) benötigt ein **Unit**- oder **LinkAttribute**, die beim Konfigurieren des **NoCs** schon mit **addUnitAttribute** bzw. **addLinkAttribute** hinzugefügt werden müssen.

3.2.4 Überprüfen von Anforderungen

Um zu überprüfen, ob alle Bindungsanforderungen erfüllt sind, wird in der Klasse Task die Methode `taskConstraintsAreSatisfied` aufgerufen. Diese Funktion ruft bei jedem Listeneintrag die Funktion `isSatisfied`. `isSatisfied` überprüft, ob der Task schon auf eine Kachel des Network-on-Chips eingebettet wurde und ob das dazugehörige `UnitAttribute` einen Fehler meldet.

Mit der Funktion `numberOfFailingConstraints` in der Klasse Task kann man herausfinden, wie viele Bindungsanforderungen missachtet werden, wenn der Task zu einer Kachel zugeordnet wird. Hierzu wird für jeden Listeneintrag die `isFeasible`-Methode aufgerufen.

Die Überprüfung der Routinganforderungen funktionieren nach dem gleichen Schema.

3.2.5 Zuordnen von Anforderungen

Wenn man die Funktion `mapConstraints` in der Klasse Task aufruft, werden alle für alle Listenelemente die Funktion `map` aufgerufen. Diese rufen, die Funktion `update` mit dem Übergabewert der Objektvariable des Listenelements vom dazugehörigen Attribut auf. Dadurch wird die Objektvariable des Attributs aktualisiert.

Wenn man wieder die Anforderungen entziehen möchte, ruft man `unmapConstraints` auf. In dieser Funktion rufen alle Listenelemente `unmap` auf. `Unmap` ruft die Funktion `update` des dazugehörigen Attributes auf. Diesmal ist der Übergabewert die negierte Objektvariable des Listenelements.

4 Evaluierung

4.1 Implementierung und Testumgebungen

Die Implementierung der CSP-Solver (Backtracking-Algorithmus [??] und Min-Conflicts-Algorithmus [??]) und der Variablenheuristiken (MWO [??], MBO [??], MaxHops [??]) erfolgte in der Programmiersprache Java. Für die Simulation eines Network-On-Chips (NoC) wurde das vom Lehrstuhl “Hardware-Software-Co-Design“ (Universität Erlangen-Nürnberg) entwickelte CoNoC-Framework verwendet. Die Taskgraphen wurden mit Hilfe des Programms *Task Graphs for Free* [?] erstellt.

4.1.1 Testumgebung 1

Die Testumgebung besteht aus 50 verschiedenen NoCs, die jeweils eine Breite und eine Länge von 10 Kacheln haben. Eine Kachel ist entweder vom Ressourcentyp 0 oder 1. Der Ressourcentyp der Kacheln wurde zufällig und gleichverteilt bestimmt. Auf den NoCs laufen zu Beginn der Einplanung keine Anwendungen.

Für die Randbedingungen wurden folgende Vereinfachungen getroffen:

- $b_{resource}$: $\forall t_i \in T : t_i.ressourceType \in \{0,1\}$
50 % der Tasks benötigen Ressourcentyp 0 und 50 % der Tasks Ressourcentyp 1.
- b_{isOK} : $\forall k_i \in K : k_i.fehlerfrei = True$
Alle Kacheln sind funktionsfähig.
- $b_{capacity}$: $\forall t_i \in T : t_i.ressourceRequirements = 100$
Jeder Task benötigt die Kachel zu 100 %. Das heißt, das auf einer Kachel nur ein Task laufen kann.
- $b_{distance}$: $\forall t_i \in T : t_i.HopLimit \in \{1,2,3\}$
Das HopLimit ist gleichverteilt zwischen 1 und 3.
- $b_{routing}$: $\forall c_i \in C : c_i.bandwidth \cdot 2 = \forall l_j \in L : l_j.capacity$.
Die Bandbreite jedes Kommunikationsknoten ist die Hälfte der Kapazität eines jeden Links.

Tasks können mit bis zu vier eingehenden und bis zu fünf ausgehenden Kommunikationsknoten verbunden sein.

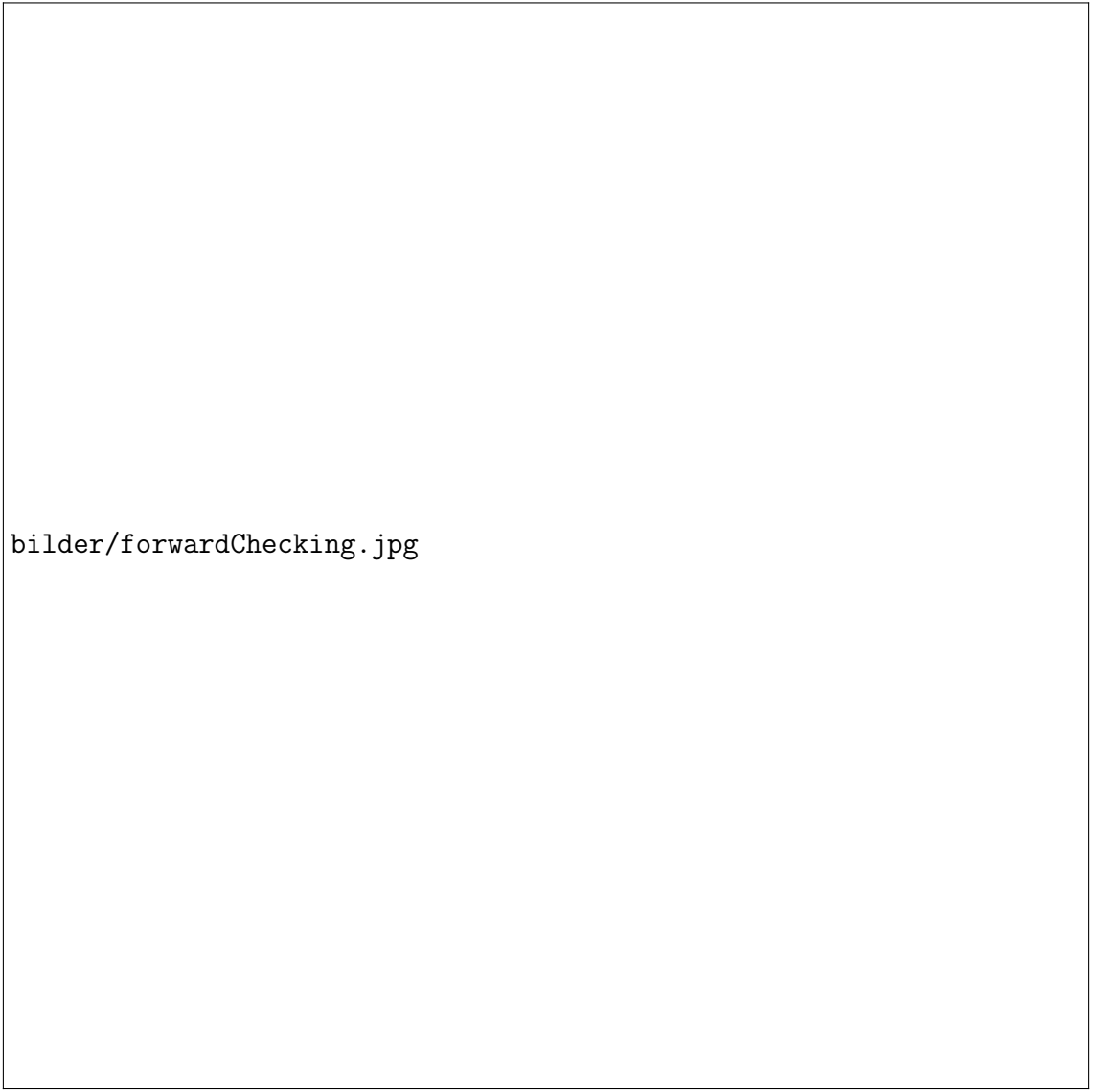
4.1.2 Testumgebung 2

Die Testumgebung 2 unterscheidet sich von Testumgebung 1 einzig bei der Rahmenbedingung $b_{routing}$.

- $b_{routing}$: $\forall c_i \in C: c_i.bandwidth \cdot 10 = \forall l_j \in L: l_j.capacity$.
Die Bandbreite jedes Kommunikationsknoten ist ein Zehntel der Kapazität eines jeden Links.

4.2 Vergleich zwischen Backtracking mit und ohne Forward-Checking

Wie man in Abbildung 4.1 sehen kann, benötigt man sehr viel weniger Einplanungsversuche, falls Forward-Checking (FC) zur Anwendung kommt. Wenn Forward-Checking benutzt wird, wird nur auf Kacheln einplant, die keine Randbedingung $b_i \in B$ verletzen. Vor allem die Randbedingung $b_{distance}$ verringert hierbei den Suchraum enorm. Ohne FC ist der Suchraum so groß wie das komplette NoC. Die Maximalzahl an Einplanungen (bei 15 Tasks) betrug ohne FC 1.183.082, mit FC hingegen nur 56.385. So gab es beim Verfahren ohne FC bis zu 20 mal mehr Einplanungsversuche als mit der Verwendung von FC. Dies spiegelt in etwa das Verhältnis der Größe der Domänen in beiden Varianten wieder.



bilder/forwardChecking.jpg

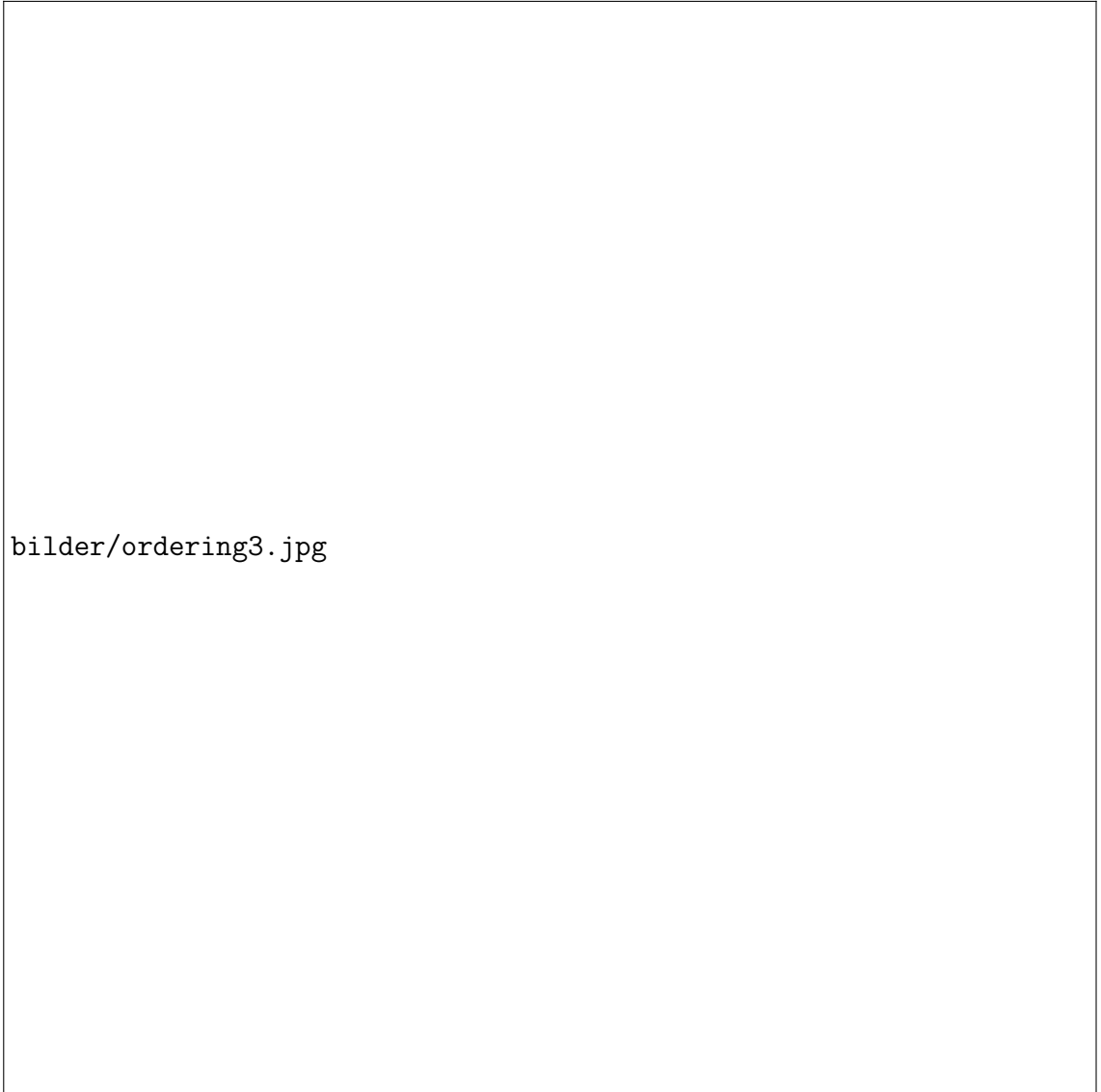
Abbildung 4.1: Vergleich zwischen Backtracking mit Forward-Checking und ohne Forward-Checking

4.3 Vergleich der Variablenordnungsverfahren

In diesem Abschnitt werden die Variablenordnungsverfahren aus Kapitel ?? verglichen. Hierbei wurden die drei vorgestellten Variablenordnungsverfahren (Minimal-Bandwidth-Ordering, Minimal-Width-Ordering, MaxHops) auf die Testfälle aus Abschnitt 4.1 angewandt und ausgewertet.

Der Testfall 1 (Abschnitt 4.1.1) wurde in Abbildung 4.2 und der Testfall 2 (Abschnitt 4.1.2) in Abbildung 4.3 ausgewertet. Wie zu erwarten war, steigt der Lösungsaufwand mit der Anzahl an Tasks. Zum einen zeigt es sich, dass die schwer zu erfüllende $b_{routing}$ -

Bedingung aus Testfall 1 einen maßgeblichen Anteil an der Anzahl der Einplanungen auf die Kacheln besitzt. Es werden bis zu 67 mal mehr Einplanungen beim Testfall 1 als beim Testfall 2 getätigt. Zum anderen erkennt man, dass die verschiedenen Testumgebungen keinen Einfluss auf die Rangfolge der Ordnungsverfahren hat. In beiden Testumgebungen benötigt Minimal-Bandwidth-Ordering (MBO) die wenigsten Einbettungen. Mit einem eher kleinen Abstand folgt Minimal-Width-Ordering (MWO) auf Rang zwei. Sehr viel mehr Einbettungen benötigt MaxHops im Vergleich zu MBO und MWO. Deshalb ist MaxHops besonders für große Taskgraphen nicht empfehlenswert.



bilder/ordering3.jpg

Abbildung 4.2: Vergleich zwischen den drei vorgestellten Variablenordnungsverfahren. Testumgebung 1 stammt aus Abschnitt 4.1.1.

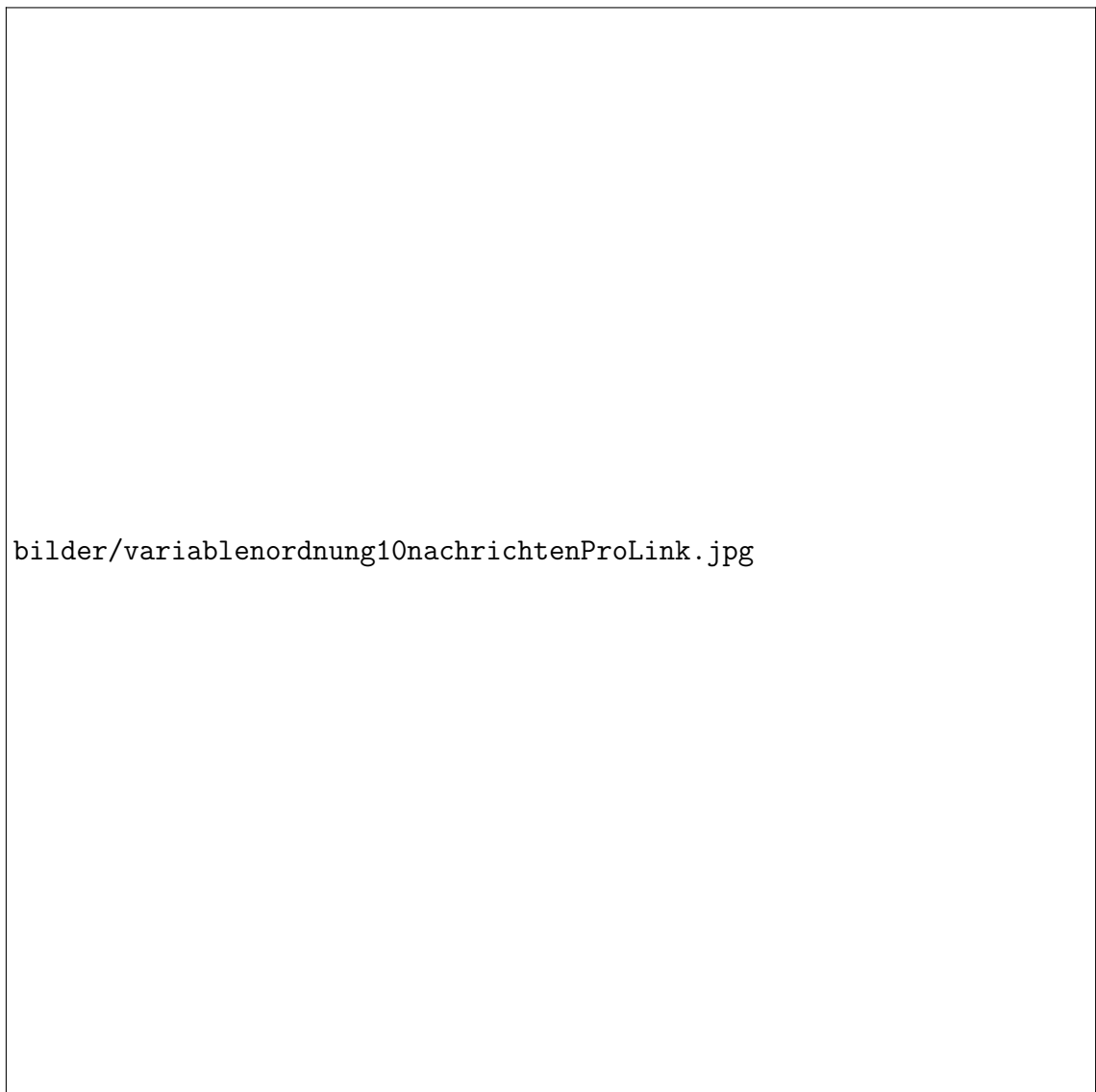
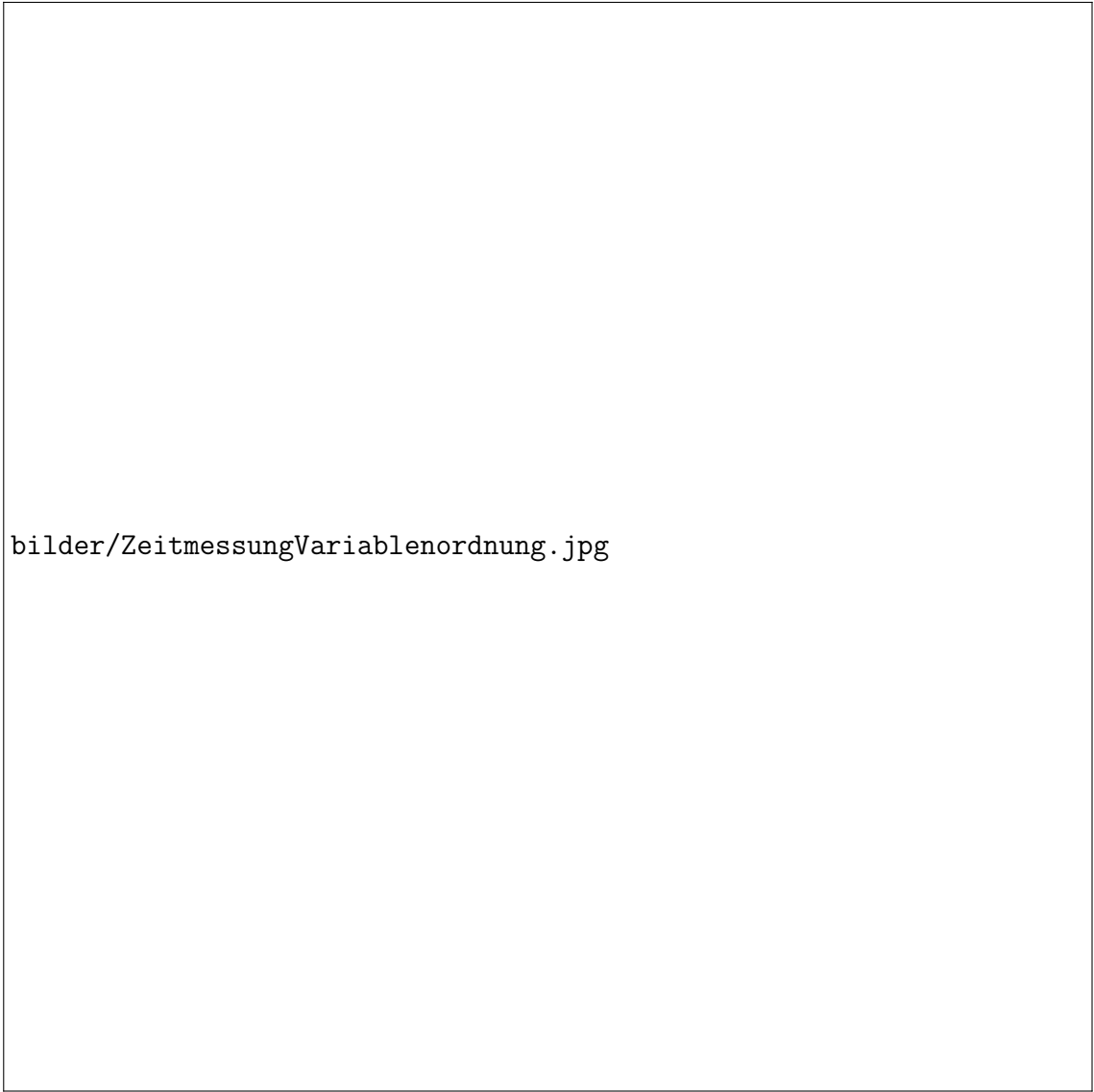


Abbildung 4.3: Vergleich zwischen den drei vorgestellten Variablenordnungsverfahren. Testumgebung 2 stammt aus Abschnitt 4.1.2.

Obwohl MBO die wenigsten Einbettungen benötigt, ist er nicht empfehlenswert. MBO benötigt für die Berechnung der Taskreihenfolge eine Zeitkomplexität von $\mathcal{O}(n^k)$ (n Anzahl der Tasks und k minimale Bandbreite). MWO und MaxHops haben lediglich eine Zeitkomplexität von $\mathcal{O}(n^2)$ (n Anzahl der Tasks). Tabelle 4.1 und die dazugehörige Abbildung 4.4 zeigen die Berechnungsdauer für die Ordnungsverfahren. Es zeigt sich, dass die Berechnungszeit für MBO zu lange dauert, um es in der Praxis anzuwenden. Deshalb ist MWO als guter Kompromiss zwischen wenigen Einplanungen und kurze Berechnungsdauer empfehlenswert.

	MWO	MBO	MaxHops
5 Tasks	6 ms	6 ms	6 ms
7 Tasks	6 ms	12 ms	6 ms
10 Tasks	7 ms	18 ms	7 ms
12 Tasks	8 ms	199 ms	7 ms
15 Tasks	8 ms	13.250 ms	9 ms

Tabelle 4.1: Zeitdauer (in Millisekunden) für die Berechnung der Variablenreihenfolge.



bilder/ZeitmessungVariablenordnung.jpg

Abbildung 4.4: Zeitdauer für die Variablenordnungsverfahren

4.4 Vergleich zwischen Min-Conflicts und Backtracking

In diesem Abschnitt wird Min-Conflicts (Abschnitt ??) mit dem Backtracking-Verfahren (Abschnitt ??) verglichen. Backtracking verwendet hierbei Forward-Checking (Abschnitt ??) und das Variablenordnungsverfahren Minimal-Width-Ordering (MWO, Abschnitt ??). Die Ergebnisse aus Testumgebung 1 (Abschnitt 4.1.1) werden in Abbildung 4.5 und die Ergebnisse aus Testumgebung 2 (Abschnitt 4.1.2) werden in Abbildung 4.6 dargestellt.

Der Min-Conflicts-Algorithmus darf pro Testfall 100 Durchgänge durchlaufen. Das heißt, es können alle Tasks 100 mal neu und zufällig gesetzt werden. Pro Durchgang ist es 500 mal möglich, einen konfliktzeugenden Task auszuwählen und die Konflikte zu lösen bzw. zu minimieren. Falls es dennoch zu keiner gültigen Belegung der Tasks kommt, wird der Testfall als *nicht gelöst* bewertet.

Die gestrichelte Linie in beiden Abbildungen zeigt auf, mit welcher Wahrscheinlichkeit der Min-Conflicts-Embedder eine Lösung für das gegebene Problem findet. Die Wahrscheinlichkeit sinkt mit der Anzahl an Tasks. Besonders in Abbildung 4.6 sieht man deutlich, wie die Wahrscheinlichkeit, dass der Min-Conflicts-Embedder eine Lösung findet, schwindet. Die Erfolgswahrscheinlichkeit beim Backtracking-Verfahren ist jedoch immer bei 100 %.

Die durchgezogenen Linien stellen den Mittelwert der Einbettungen auf den Kacheln graphisch dar. Es werden hierbei nur erfolgreiche Einbettungsversuche gewertet.

In Abbildung 4.5 erkennt man, dass die Erfolgswahrscheinlichkeit für das Min-Conflicts-Verfahren deutlich sinkt. So findet der Min-Conflicts-Embedder bei der Anzahl von 15 Tasks nur in 18 % der Fälle eine Lösung. Der Grund liegt in den schwer erfüllbaren Bedingungen der Testumgebung 1, insbesondere Bedingung $b_{routing}$.

Der Verlauf der Einbettungsfunktion zeigt Folgendes: Bei einer geringen Anzahl an Tasks ist das Backtracking- und das Min-Conflicts-Verfahren annähernd gleich. Steigt die Anzahl der Tasks¹, so benötigt der Min-Conflicts-Embedder erheblich mehr Einbettungen. Der Knick an der Stelle zwischen zehn und zwölf Tasks lässt sich damit erklären, dass die Erfolgsquote von Min-Conflicts in dieser Auswertungsspanne erheblich gesunken ist.

¹das zu lösende Problem wird schwieriger

bilder/MinConflicts2Links.jpg

Abbildung 4.5: Vergleich zwischen Backtracking mit Minimal-Width-Ordering (MWO) und Min-Conflicts. Testumgebung 1 stammt aus Abschnitt 4.1.1. Die durchgezogenen Linien geben die Einbettungen auf den Kacheln an. Die gestrichelte Linie gibt die Erfolgswahrscheinlichkeit der Einbettung an.

Abbildung 4.6 stellt die Auswertung der Testumgebung 2 (Abschnitt 4.1.2) dar. Die Erfolgsquote ist hierbei höher als bei Abbildung 4.6. Das liegt daran, dass es mehr Lösungen auf den Network-on-Chip (NoC) gibt. Der Min-Conflicts-Embedder schneidet aber wiederum viel schlechter als der Backtracking-Embedder ab.

bilder/MinConflicts10Links.jpg

Abbildung 4.6: Vergleich zwischen Backtracking mit Minimal-Width-Ordering (MWO) und Min-Conflicts. Testumgebung 2 stammt aus Abschnitt 4.1.2. Die durchgezogenen Linien geben die Einbettungen auf den Kacheln an. Die gestrichelte Linie gibt die Erfolgswahrscheinlichkeit der Einbettung an.

Die Evaluierung hat ergeben, dass das Min-Conflicts-Verfahren insbesondere für schwer zu lösende Probleme ungeeignet ist. Die Verwendung des Min-Conflicts-Verfahren für die Einbettungen einer Anwendung in einem Network-on-Chip (NoC) ist nicht zu empfehlen, falls es sich um ein schwer zu erfüllendes Problem handelt. Min-Conflicts eignet sich jedoch als Reparaturverfahren [MJPL90], wenn bereits eine Lösung gefunden wurde, aber eine von einem Task benutzte Kachel defekt wird. In diesem Fall kann Min-Conflicts eine Lösung mit wenigen Einbettungsversuchen finden.

5 Zusammenfassung

In dieser Arbeit wurde das Problem der Selbsteinbettung von Taskgraphen bei Manycore-Architekturen beschrieben und in ein Constraint-Satisfaction-Problem umgewandelt.

Weiterhin wurden das Backtracking-Verfahren (systematisches Verfahren) sowie das Min-Conflicts-Verfahren (stochastisches Verfahren) als zwei mögliche Varianten von CSP-Solvern näher erläutert und auf die Problemspezifikation angewandt.

Als Mechanismus zum Löschen inkonsistenter Werte wurde Forward-Checking im Min-Conflicts- und Backtracking-Algorithmus verwendet.

Variablenordnungen tragen in systematischen Verfahren dazu bei, die Anzahl der Einbettungen zu verringern. Es wurden daher drei verschiedene Arten von Variablenordnungen näher betrachtet.

Abschließend wurden verschiedene Varianten anhand der Anzahl der Einbettungen miteinander verglichen.

Im Rahmen dieser Arbeit konnte gezeigt werden, dass Forward-Checking maßgeblich dazu beiträgt, die Anzahl der Einplanungen zu verringern. Es konnte festgestellt werden, dass eine Verwendung von Backtracking ohne Forward-Checking nicht sinnvoll ist.

Bei den Variablenordnungen hat Minimal-Bandwidth-Ordering die geringste Zahl an Einplanungen benötigt. Die hohe Berechnungsdauer von $\mathcal{O}(n^k)$ macht diese Variablenordnung für die praktische Anwendung unbrauchbar, wohingegen sich Minimal-Width-Ordering als guter Kompromiss zwischen Berechnungsaufwand und Anzahl an Einplanungen gezeigt hat. Die MaxHops-Variablenordnung benötigte eine deutlich höhere Anzahl an Einplanungen.

In dieser Arbeit wurden die drei genannten Variablenordnungen jeweils statisch vor der Einplanungsphase berechnet. Eine andere, in dieser Arbeit nicht untersuchte Möglichkeit ist es, die nächste Variable erst in der Einplanungsphase dynamisch zu bestimmen. Minimum-Remaining-Values (MRV) [?], auch unter dem Namen Fail-First-Principle (FFP) [?] bekannt, kann dazu verwendet werden. In auf dieser Arbeit aufbauenden Untersuchungen ist zu prüfen, ob der erhöhte Rechenaufwand in der dynamischen Einplanungsphase durch eine Verringerung der Einplanung gerechtfertigt ist.

Das Min-Conflicts Verfahren hat sich aufgrund abnehmender Erfolgswahrscheinlichkeiten als unbrauchbar für die praktische Anwendung erwiesen. Das Backtracking-Verfahren, welches in diesen Untersuchungen deutlich besser abgeschnitten hat, sollte weiterentwickelt werden. So könnte der Suchraum durch das Ausnutzen von Symmetrien [?] weiter eingeschränkt werden.

Literatur

- [Fra11] Thomas Frank. Network on chip. In *Architekturen, Herausforderungen, Lösungen*. Dresden, 2011.
- [GRJ00] G. Görz, C.-R. Rollinger, and J. Schneeberger. *Handbuch der Künstlichen Intelligenz*, chapter Constraints, pages 267–285. Oldenbourg Verlag, München, 3. edition, 2000.
- [HN11] Hans Robert Hansen and Gustav Neumann. Wirtschaftsinformatik 1. In *Grundlagen und Anwendungen*. UTB GmbH, 2011.
- [Jä14] Daniel Jäger. *Selbsteinbettung von Anwendungen mit Bindungs- und Routin-ganforderungen in Mehrprozessorsystemen*. Bachelorarbeit, Department of Computer Science 12, University of Erlangen-Nuremberg, February 2014.
- [MJPL90] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the eighth National conference on Artificial intelligence*, pages 17–24. AAAI Press, University of Leeds, 1990.
- [RHT12] S. Roloff, F. Hannig, and J. Teich. Approximate time functional simulation of resource-aware programming concepts for heterogeneous mpsoes. In *Design Automation Conference (ASP-DAC)*, pages 187 – 192, New York, NY, USA, January 2012. 17th Asia and South Pacific.
- [WWT11] Andreas Weichslgartner, Stefan Wildermann, and Jürgen Teich. Dynamic decentralized mapping of tree-structured applications on noc architectures. In *In Proceedings of NOCS*, pages 201–208. ACM, New York, NY, USA, 2011.
- [X1014] X10 Programmiersprache. <http://x10-lang.org/>, 2014.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 31. März 2015

Daniel Jäger