Algorithmen im Chip-Entwurf 8

Reale FPGA-Router: PathFinder/VPR

Andreas Koch
FG Eingebettete Systeme
und ihre Anwendungen
TU Darmstadt

Übersicht

- Problem
- Ideen
- Modellierung
- Algorithmus
- Details
- Verbesserungsmöglichkeiten

Problem

- Verdrahtung auf FPGAs
- Begrenzte Anzahl von Ressourcen
  - Verbindungssegmente
- Feste Kanalbreite
  - Unterschied zu vielen ASICs
- Verdrahtbarkeit ausschlaggebend
  - Geschwindigkeit zweitrangig

Idee

- Berücksichtige *Verdrahtbarkeit*
  - Bei Lösung des gesamten Verdrahtungsproblems
- Bestimme
  - Nachfrage nach Ressourcen
    - Metallsegmente, Pins, etc.
- Nachfrage bestimmt Preise
  - Verschiedene „Verbraucher“ akzeptieren unterschiedliche Preise
    - „Verbraucher“ = Netze
  - „Billige“ Lösungen haben Nachteile
    - Sind z.B. langsamer
- Versuche Gesamtbedarf zu decken

Vorgehen

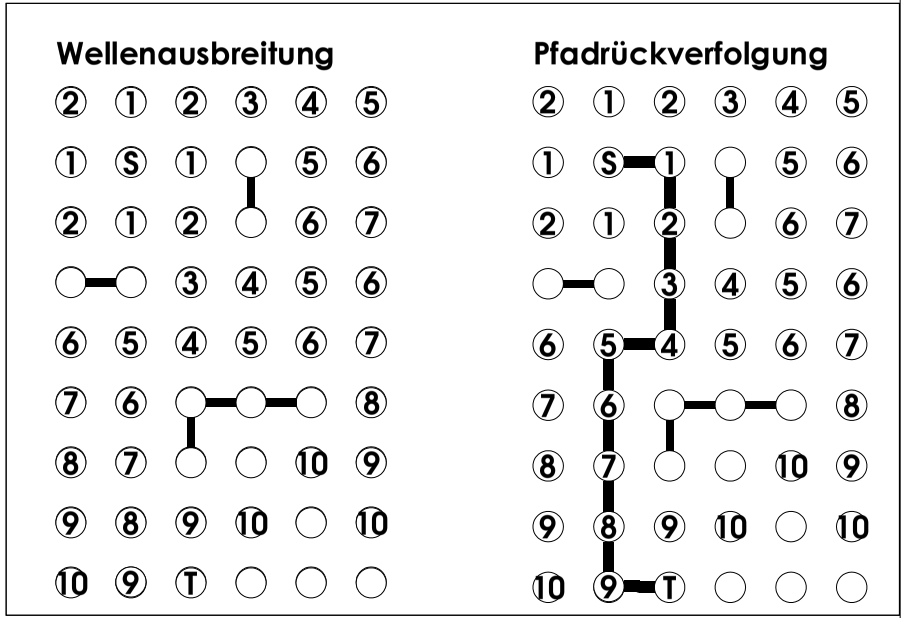
- Verdrahte jedes Netz für sich alleine
  - Mit den aktuellen Resourcenkosten
  - Jeweils optimal
    - ... für gegebenen Algorithmus
  - Ignoriere Ressourcenbegrenzungen
- Zähle Mehrfachbelegungen
- Grundlage für Nachfrageberechnung
- Solange Mehrfachbelegungen
  - Erhöhe Kosten für stark nachgefragte Rsrc.
  - Verwerfe gesamte Verdrahtung
  - Verdrahte nochmal mit den neuen Kosten
- Sollte nach 30-45 Iterationen konvergieren

Zwei Stufen

- Signal Router
  - Verdrahtet einzelne Netze
  - Maze Router (Lee)
    - Aber Verbesserungen möglich
- Global Router
  - Verdrahtet gesamte Schaltung

```
globalrouter() {
        count = 0;
        while (sharedresources() && count < limit) {
                foreach (n in Nets)
                        signalrouter(n);
                count++;
        }
        if (count == limit)
                return „unroutable“
}
```


Maze Router



Vorgehen und Kosten

- Beim Maze-Router
  - Breitensuche
    - Wellenfront
  - Kosten: Manhattan-Distanz
    - D = |x1-x2| + |y1-y2|
  - Kosten nur bei Rückverfolgung berücksichtigt
    - Nicht bei Wellenausbreitung
      - In alle Richtungen
- Variation für Signal Router
  - Hohe Nachfrage verursacht hohe Kosten
  - Bevorzugt in billige Richtungen ausbreiten
    - Später Verfeinerung
      - Zeitkritische Netze dürfen höhere Kosten verursachen

Signal Router 1

```
Tree<RtgRsrc>
signalrouter(Net n) {
        Tree<RtgRsrc> RT;
        RtgRsrc i, j, v = nil, w;
        PriorityQueue<int,RtgRsrc> PQ;
        HashMap<RtgRsrc,int> PathCost;

        i = n.source();
        RT.add(i, ());          // Quelle ist Bestandteil der Verdrahtung
        PathCost[*] = +Inf;  // Zunächst alles unerreichbar
        PathCost[i] = 0;       // Kosten von Quelle zu Quelle sind 0

        foreach (SinkTerminal j in n.sinks())  {
                /* route Verbindung zur Senke  j */
        } ;

        return (RT);
}
```


Signal Router 2

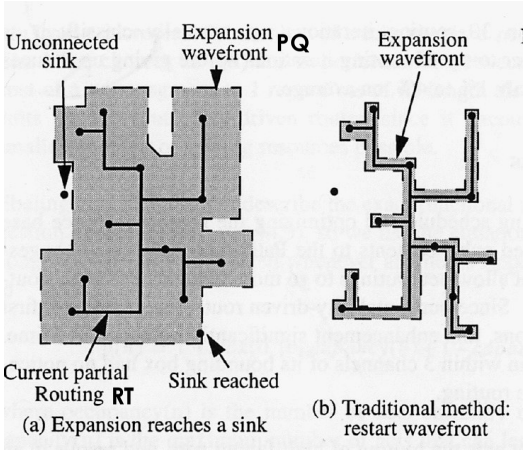
```
foreach (SinkTerminal j in n.sinks) {
        PQ.clear();                          Ganze bisherige Route ist Ausgangs„punkt“
        foreach (v in RT.nodes())
                PQ.add(0, v)
        do  {                                                Kostenbasierte Wellenausbreitung
                v = PQ.removeLowestCostNode();
                if (v != j)
                        foreach (w in v.neighbors()) {          /* Kosten ≠ Distanz ! */
                                if (PathCost[w] > PathCost[v] + w.cost()) {
                                        PathCost[w] = PathCost[v] + w.cost();
                                        PQ.add(PathCost[w], w);
                                }
                        }
        } while (v != j)

        while (! (v in RT.nodes())) {                                  Pfadrückverfolgung
                w = v.findCheapestNeighbor(PathCost);
                RT.add(v,(w,v));
                v.updateCost();      /  * Rsrc jetzt benutzt, für Nachfolger teurer */
                v = w;
        }
}
```


Signal Router Details 1

- Verdrahtungsressourcen sind persistent
  - Z.B. Globale Variablen
- v.cost() über alle Netze berechnet
  - Mehrere Aufrufe von Signal Router
  - Auch mehrere Iteration vom Global Router (später)
- v.updateCost() aktualisiert die Daten
- v.neighbors() definiert Verdrahtungsarchitektur
  - Erklärung später (Routing Resource Graph)
  - Idee: Breitensuche
  - Sinnvolle Begrenzung:
    - Nicht mehr als 3 Kanäle ausserhalb des umschliessenden Rechtecks suchen
    - Verkleinert Suchraum
      - Minimale Qualitätsminderung

Signal Router Details 2

- Pfadrückverfolgung und Anschluss



Verdrahtbarkeit

- Fliesst via Kostenfunktion v.cost() ein

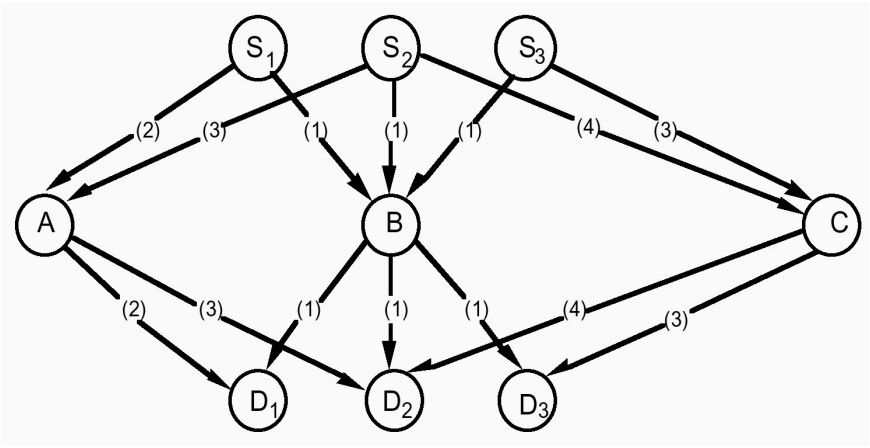
$$c_v = b_v \cdot p_v$$

- Idee
  - Basiskosten b_v eines Knotens *v*
    - Zunächst annehmen b_v=1 (wird später verfeinert)
  - Verteuerungsfaktor p_v bei Mehrfachbelegung
    - Erfasst hohe Nachfrage
    - Beginnt klein, wächst im Laufe der Zeit an

$$p(v) = 1 + \max(0, [\text{occupancy}(v) + 1 - \text{capacity}(v)] \cdot p_{fac})$$

    - Occupancy(v): Aktuelle Belegungsanzahl der Ressource v
    - Capacity(v): Belegungskapazität der Ressource v
    - P_{fac0}=0.5, p'_{fac} = 1.5 p_{fac} nach Iteration vom Global Router
    - Bei jeder Netzänderung occupancy(v) aktualisieren
      - Passiert in v.updateCost

Beispiel: Entwicklung von p_v



Einfacher Maze Router: 1,2,3 = 14 2,1,3=12 3,2,1=Fehlschlag

1. Iteration Global Router: B dreifach belegt, Kosten v.cost() merken
M. Iteration Global Router: 1 nun über A billiger, Kosten merken
N. Iteration Global Router: 3 nun über C billiger, keine Überbelegung mehr

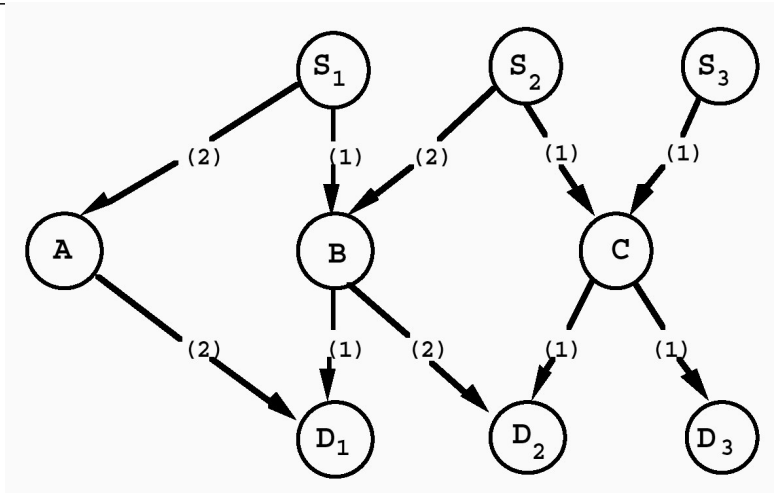
Algorithmus: 1. Versuch

```
globalrouter(Set<Nets> N) {
        HashMap<Net,Tree<RtgRsrc>> NRT;
        count = 0;
        while (sharedresources() && count < limit) {
                foreach (n in N) {
                        NRT[n].unroute();    // muss p_v aktualisieren!
                        NRT[n] = signalrouter(n);
                }
                count++;
        }
        if (count == limit)
                return „unroutable“
}
```


Beispiel

- An Tafel

Weitergehendes Beispiel



Mit einfachem Maze Router in Reihenfolge 1,2,3: C doppelt belegt

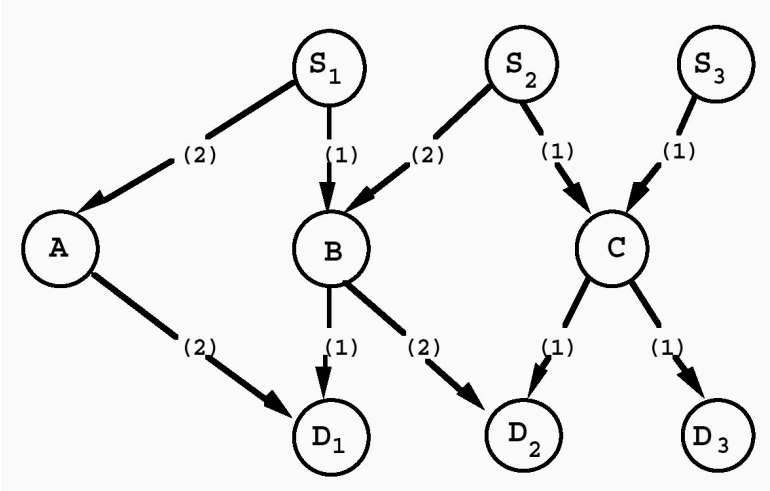
Lösung: 1 aus dem Weg schaffen, 2 neu verdrahten
Aber: 1 ist gar nicht behindert, geht also nicht freiwillig

Lösung

- p_v reicht alleine nicht aus
- Besseres „Gedächtnis“ einführen
  - Historische Überbelegungen erhöhen akt. Preis
  - h_v akkumuliert alle Mehrfachbelegungen
    - p_v sieht nur aktuelle Belegung
  - Kostenfunktion erweitern: $c_v = b_v \cdot p_v \cdot h_v$
- Aktualisiere einmal pro Global Router Iteration *i*

$$h(v)^i = \begin{cases} 1, i = 1 \\ h(v)^{i-1} + \max(0, \text{occupancy}(v) - \text{capacity}(v)), i > 1 \end{cases}$$

Wirkung von h_v



1,2,3: C doppelt belegt
Weitere Iterationen: C wird immer teurer durch Akkumulieren der h_C
2 weicht dann auf B aus, Doppelbelegung via p_B,h_B, 1 weicht auf A aus

Basiskosten b_v

- Erster Ansatz: Verzögerung
  - Bei uns nur T_{switch}
- Besser: Feste Kosten
  - 10% weniger Tracks benötigt
  - Bevorzuge Input Pins
    - Niedrigere Kosten
    - “Lockt” Maze Router via PriorityQueue PQ schneller zu Sinks
      - Werden eher abgearbeitet
- Vorschlag
  - Input Pins b_v=0.95
  - Alle anderen Elemente b_v=1

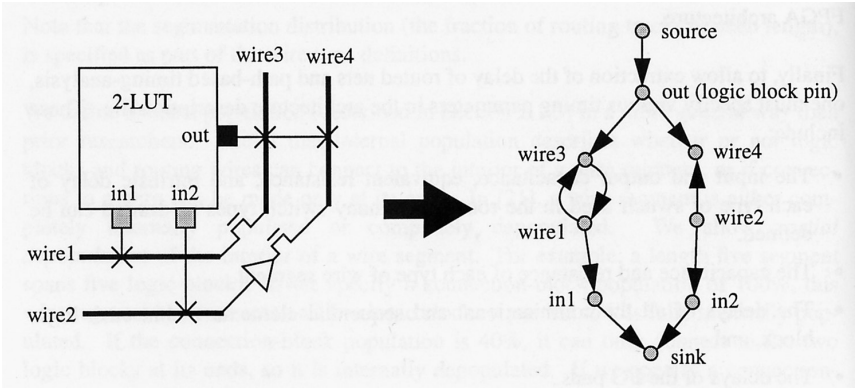
Vervollständige globalrouter()

```
Graph<RtgRsrc>     Interconnect;          // Kanten (RtgRsrc,RtgRsrc)

globalrouter(Set<Nets> N) {
        HashMap<Net,Tree<RtgRsrc>> NRT;
        count = 0;
        while (sharedresources() && count < limit) {
                foreach (n in N) {
                        NRT[n].unroute();     // muss p_v aktualisieren!
                        NRT[n] = signalrouter(n);
                }
                count++;
                foreach (r in Interconnect.nodes())
                        r.updateHistory();    // h_v aktualisieren
        }
        if (count == limit)
                return „unroutable“
}
```

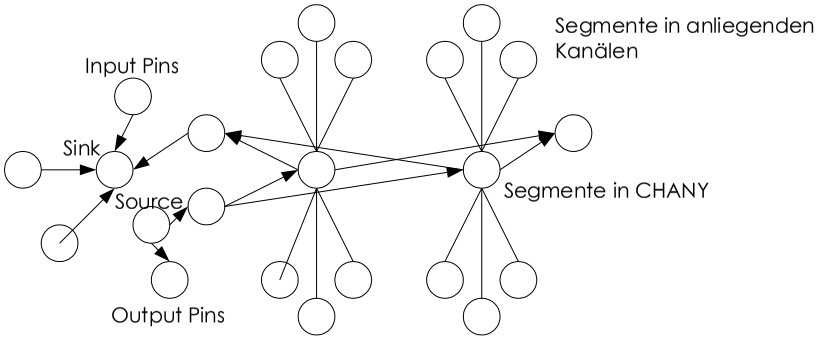

Routing Resource Graph

- Fundamentale Datenstruktur
- Modelliert Verbindungsnetzwerk
- Knoten
  - Leitungen
  - Pins
- Kanten
  - Schalter (Pass-Transistoren, bidirektional)
  - Buffer (unidirektional)
- Äquivalente Pins
  - Outputs: Source-Knoten
  - Inputs: Sink-Knoten
- Fassungsvermögen (capacity)
  - Bei Source und Sink-Knoten: Anzahl der Pins

Beispiel 1



Beispiel 2



- Verzögerung $d_{u,v}$
  - T_{switch} zwischen Metallsegment-Knoten *u,v*

Ausbau auf Verzögerung

- **Optimiere auch noch Verzögerung**
- **Erweiterung der Kostenfunktion v.cost(u)**

$$C_{u,v} = \mathrm{Crit}(i,j) \cdot d_{u,v} + [1 - \mathrm{Crit}(i,j)] \cdot b_v \cdot h_v \cdot p_v$$

- **$d_{u,v}$: Verzögerung von *u* nach *v***
- **Crit(i,j): Abart der Criticality(i,j)**

$$\mathrm{Crit}(i,j) = \max(0.99 - \frac{\mathrm{slack}(i,j)}{D_{max}}, 0)$$

  - **Idee: Auch kritische Netze achten *etwas* auf Verdrahtbarkeit**

Änderung signalrouter()

```
foreach (SinkTerminal j in n.sinks ordered decreasing Crit(i,j)) {
        PQ.clear();
        foreach (v in RT.nodes())
                PQ.add(0, v)
        do {
                v = PQ.removeLowestCostNode();
                if (v != j)
                        foreach (w in v.neighbors()) {
                                if (PathCost[w] > PathCost[v] + w.cost(v)) {
                                        PathCost[w] = PathCost[v] + w.cost(v);
                                        PQ. add(PathCost[w], w);
                                }
                        }
        } while (v != j)

        while (! (v in RT.nodes())) {
                w = v.findCheapestNeighbor(PathCost);
                RT.add(v,(w,v));
                v.updateCost();
                v = w;
        }
}
```


Änderung globalrouter()

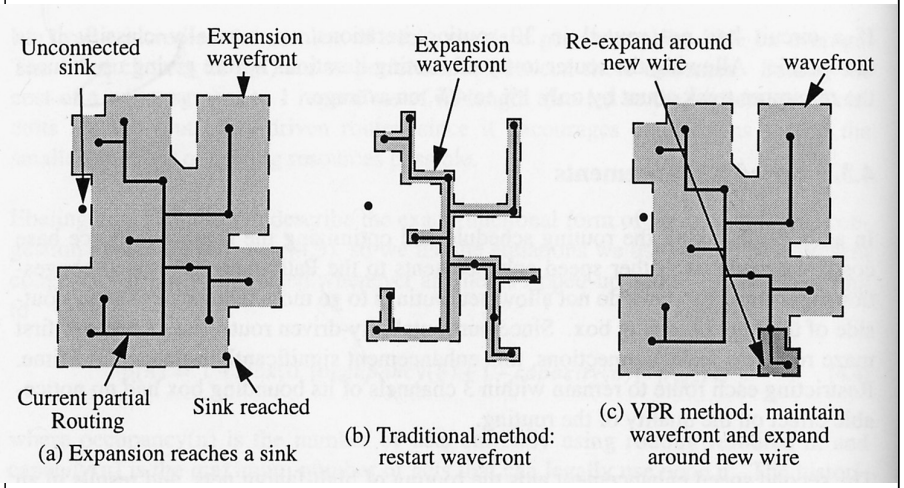
```
Graph<RtgRsrc>    Interconnect;

globalrouter(Set<Nets> N) {
        HashMap<Net,Tree<RtgRsrc>> NRT;
        count = 0;
        foreach (n in N)
                foreach (j in n.sinks())
                        Crit[n.source(), j] = 1
        while (sharedresources() && count < limit) {
                foreach (n in N) {
                        NRT[n].unroute();    // muss p_v aktualisieren!
                        NRT[n] = signalrouter(n);
                }
                count++;
                foreach (r in Interconnect)
                        r.updateHistory();   // h_v aktualisieren
                N.timingAnalysis();          // Crit(i,j) des Netzes aktualisieren
        }
        if (count == limit)
                return „unroutable“
}
```

Im ersten Durchgang Nachfrage bei minimaler Verzögerung bestimmen

Vergleich

- **PathFinder [McMurchie&Ebeling 1995]**
  - **Zunächst nur verdrahtungsorientiert**
  - **Keine vorgegebene Sink-Reihenfolge**
  - **Wellenausbreitung**
    - **Bis alle Sinks erreicht**
  - **Verbesserbar**
    - **Alte Wellenfront in PQ nicht verwerfen (in VPR)**
      - **Einfach neue Sink an RT anschliessen**
      - **Neue Segmente in PQ übernehmen**
  - **Bei Verzögerungsorientierung**
    - **Jetzt steht Sink-Reihenfolge fest**
      - **Absteigende A_{ij} (vergleichbar Criticality)**

Wellenausbreitung



Verbesserungen

- Swartz, Betz, Rose 1998
- Optimierung auf Geschwindigkeit
  - Qualitätsverlust?
- Zwei Kernideen
  - Gezielte Ausbreitung statt breiter Wellenfront
  - Beschränkung auf sinnvolle Startpunkte
- Diverse Detailverbesserungen

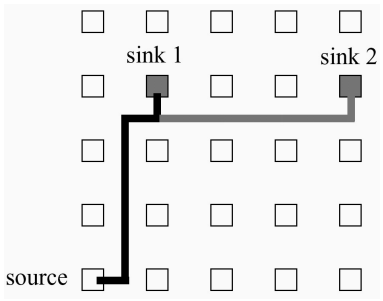
Ausbreitung 1

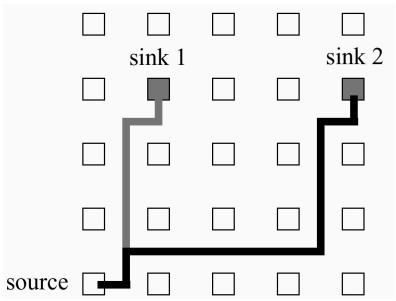
- Gerichtete Tiefensuche DDFS statt BFS
  - Suche bevorzugt in Richtung auf Ziel *j* zu

$$\mathrm{Cost}(i,v) = \mathrm{PathCost}(i,u) + C_0 + \alpha \cdot \Delta D$$

  - PathCost(*i*,*u*): Kosten bis zum Vorgänger *u*
    - C_0: Verdrahtungsabhängige Basiskosten von *v*
    - Vergleichbar c_n, wächst aber viel stärker
    - Weniger Iterationen
  - ΔD: Manhattan-Distanz von *v* zum Ziel *j*
    - <0: *v* liegt näher an *j* als *u* (= billiger)
    - >0: *v* liegt weiter von *j* als *u* (= teurer)
  - α: Richtungsfaktor
    - =0: BFS, keine richtungsabhängigen Komponenten
    - >>0: Nicht mehr verdrahtungsorientiert, Greedy
    - =1.5: Empfohlen, hohe Beschleunigung, gute Qualität

Ausbreitung 2

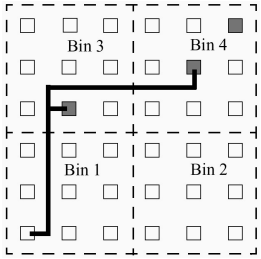




- Reihenfolge der Sinks
  - Nächstgelegene zuerst
    - Besser Anschliessbarkeit der Folgenden
- Reihenfolge der Netze
  - Viele Terminals zuerst
    - Vermeidung von Blockaden

Sinnvolle Startpunkte

- PathFinder/VPR
  - Ausbreitung von gesamten RT aus
    - Übernahme in PQ mit Kosten 0
  - Ineffizient, gerade bei vielen Terminals
- Idee
  - Nur Segmente aus RT „nahe“ beim Ziel in PQ
  - Aufteilen der gesamten Fläche in Bins
    - Hier:
      - Nur Segmente in Bin 4 expandieren
- Lohnend bei
  - Netzen mit vielen Terminals



Binning Details

- Bin-Größe
  - Sollte passen
  - Berechnung pro Netz
    - Durchschnittliche Fläche pro Sink A_s = netBB/#sinks
    - Bewährt: Bin-Größe 4x A_s
  - Expandiere
    - Nur Segmente im gleichen Bin wie nächstes Ziel
      - Einfache Entfernungsberechnung, kein Bin-Raster
- Leere Bins
  - Bin mit Ziel enthält noch keine Segmente
  - Erweitere Suchradius auf 8 Nachbar-Bins
  - Falls immer noch leer
    - Suche von ganzem RT aus

Auswirkungen

- Low-Stress Routing
  - >10% mehr Tracks als minimal erforderlich
- 15 Beispielschaltungen
- Durchschnittliche Rechenzeit
  - BFS in VPR: 731s
  - DDFS: 14s
  - DDFS+Bins: 7s
- Durchschnittlicher Qualitätsverlust
  - BFS in VPR: 15.5 Tracks
  - DDFS: 15.5 Tracks
  - DDFS+Bins: 15.8 Tracks

Praktische Arbeiten

- Nicht genau nachprogrammieren
  - Viele Details nicht gezeigt
- Konzepte verstehen
- Inspiration für eigene Ideen
- Sinnvoll
  - Routing Graph
  - Darin nach Verdrahtungen suchen
- Papers auf Web-Site
  - PathFinder, McMurchie & Ebeling 1995
  - Verbesserungen von Swartz et al., 1998
  - Auszüge aus VPR Beschreibung, 1999 [19MB!]

Zusammenfassung

- Verdrahtungsproblem auf FPGAs
- Verdrahtbarkeitsorientierte Verdrahtung
- PathFinder-Algorithmus
  - Gewichteter Maze-Router
  - p_v, h_v
- Erweiterung auf Verzögerung
  - Durch Criticality
- Verbesserungen
  - Bessere Suchalgorithmen
- Dienstag 13.12.: Abgabe Phase 2
  - Verdrahtung 2