technische universität
dortmund

fakultät für informatik
informatik 12

Synthese Eingebetteter Systeme

Sommersemester 2011

15 – Abbildung von Anwendungen: SHAPES/DOL

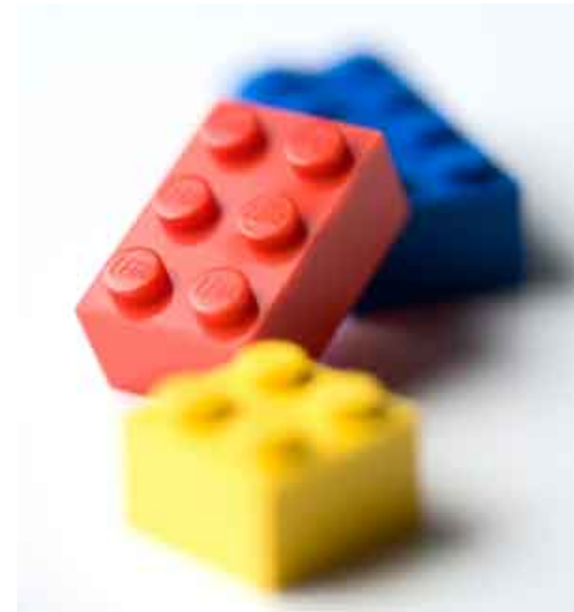
Michael Engel
Informatik 12
TU Dortmund

2011/06/22

– unter Verwendung von Foliensätzen von Dr. Iuliana Bacivarov, ETH Zürich –

Abbildung von Anwendungen auf Multicores

- Problem der Abbildung
- KPNs
- SHAPES/DOL
  - API und Unterschiede zu KPNs
  - Architektur
  - Entwurfsfluss
  - Beispiele



© m. engel, p. marwedel, informatik 12, 2011

Ausgangssituation

- Steigende Komplexität eingebetteter Systeme
  - Anzahl an Logikgattern pro System steigt (Moore)
  - Zukünftige MPSoCs besitzen Milliarden Gatter
- Steigende Entwurfskomplexität
- Problem der Leitungsverzögerung
  - Gatterverzögerungen sinken mit abnehmenden CMOS-Strukturbreiten
  - Verzögerung durch eine Leitung fester Länge steigt aber -> Begrenzung der maximalen Taktfrequenz
- Tiefgreifende Änderungen der Hardware-/Software-Entwurfsmethoden für zukünftige Technologien erforderlich

SHAPES-Architektur

SHAPES (scalable software hardware architecture platform for embedded systems):

- Versuch, diese Probleme mit einer *gekachelten (tiled) Architektur* zu lösen
  - Vordefinierte, miteinander verbundene Prozessorkacheln
  - Heterogene Kacheln: z.B. RISC-CPUs, VLIW-DSPs, Netzwerkprozessoren und Speicher+Peripherie auf Kacheln
- Wichtig: *Skalierbarkeit* – Anwendungen sollen mit geringem Aufwand auf verschiedene SHAPES-Hardwarearchitekturen portierbar sein

SHAPES-Architektur

- SHAPES ermöglicht Abbildung einer Anwendung auf Architekturen mit stark unterschiedlichen Kachelanzahlen
- Skalierbarkeitsziele für SHAPES:

4–8 tiles	low–end single modules for mass market applications
2000 tiles	classic digital signal processing systems (e.g. radar, medical equipment)
32000 tiles	high–end systems requiring massive numerical computation

© m. engel, p. marwedel, informatik 12, 2011

SHAPES-Architektur

- Gekachelte Architektur bietet viele Vorteile
  - Schwierig für Anwendungen, Potential voll auszunutzen
- Möglicherweise große Verzögerungen zwischen weit entfernten Kacheln, überlastete Kommunikations-ressourcen, nicht genügend Parallelität in einer Anwendung sichtbar
  - In diesen Fällen kann nicht die volle Rechenleistung der Architektur genutzt werden
- Systemsoftware muss sicherstellen, dass Anwendungen auf SHAPES-Hardware effizient ausgeführt werden
  - Aufwandsminimierung für den Anwendungsentwickler

SHAPES/DOL

Zwei wesentliche Voraussetzungen

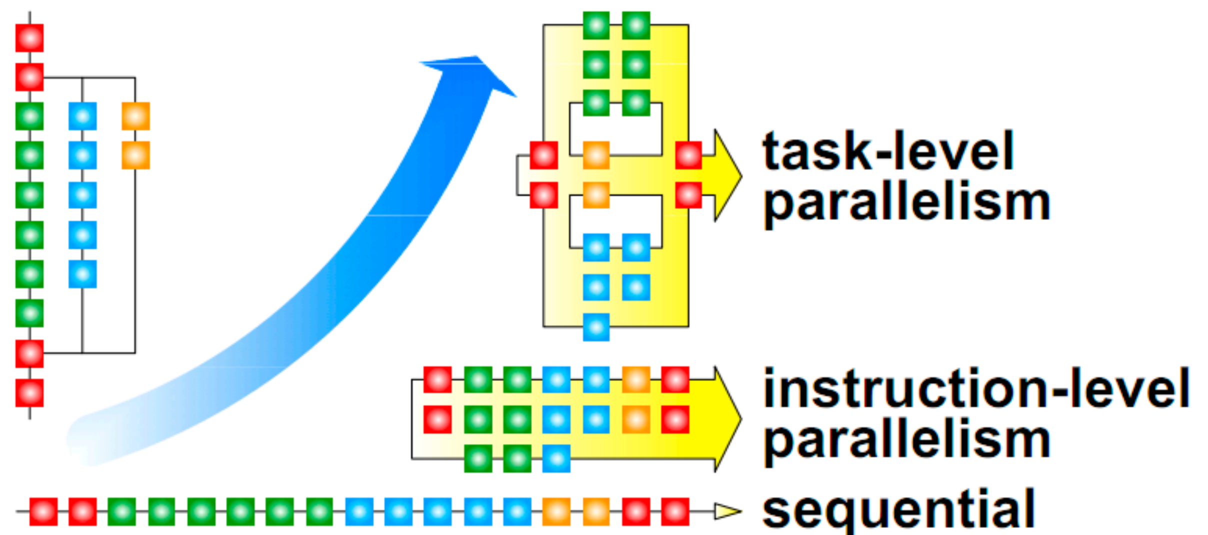
- Da das System hochgradig parallel ist, sollte die Anwendung es auch sein
  - Anwendungsentwickler muss dazu in der Lage sein, die Parallelität des Algorithmus an SHAPES zu vermitteln
  - Dazu müssen konventionelle Arten der Anwendungsentwicklung aufgegeben werden
  - Information über die algor. Struktur muss erhalten bleiben, auch wenn Parallelität erkennbar ist
- Systemsoftware muss wichtige Parameter wie Bandbreite, Rechenleistung und Latenzen kennen

SHAPES/DOL

- DOL (*Distributed Operation Layer*) ist Teil der SHAPES-Systemsoftwareumgebung
- Zweck: Unterstützung des Programmiers einer SHAPES-Plattform, um effiziente Abbildung der Anwendung auf die Hardware zu finden
  - Abbildung von Anwendungs-Tasks auf Rechenressourcen
  - Abbildung von Verbindungen auf Kommunikationsressourcen
- Bestmögliche Ausnutzung der parallelen Hardware
  - Programmierer befolgt Menge von Regeln, verwendet eine Menge von Schnittstellen: *Programmiermodell*

SHAPES/DOL: Ziele

- DOL-Vision: “Write Once, Run Anywhere”
- Einprozessorsysteme
  - Verwenden C/C++ und plattformspezifischen Compiler
- (Heterogene) Multiprozessorsysteme
  - Abbildung: Binding und Scheduling
  - Ausführung: verteilte Berechnung und Kommu-nikation



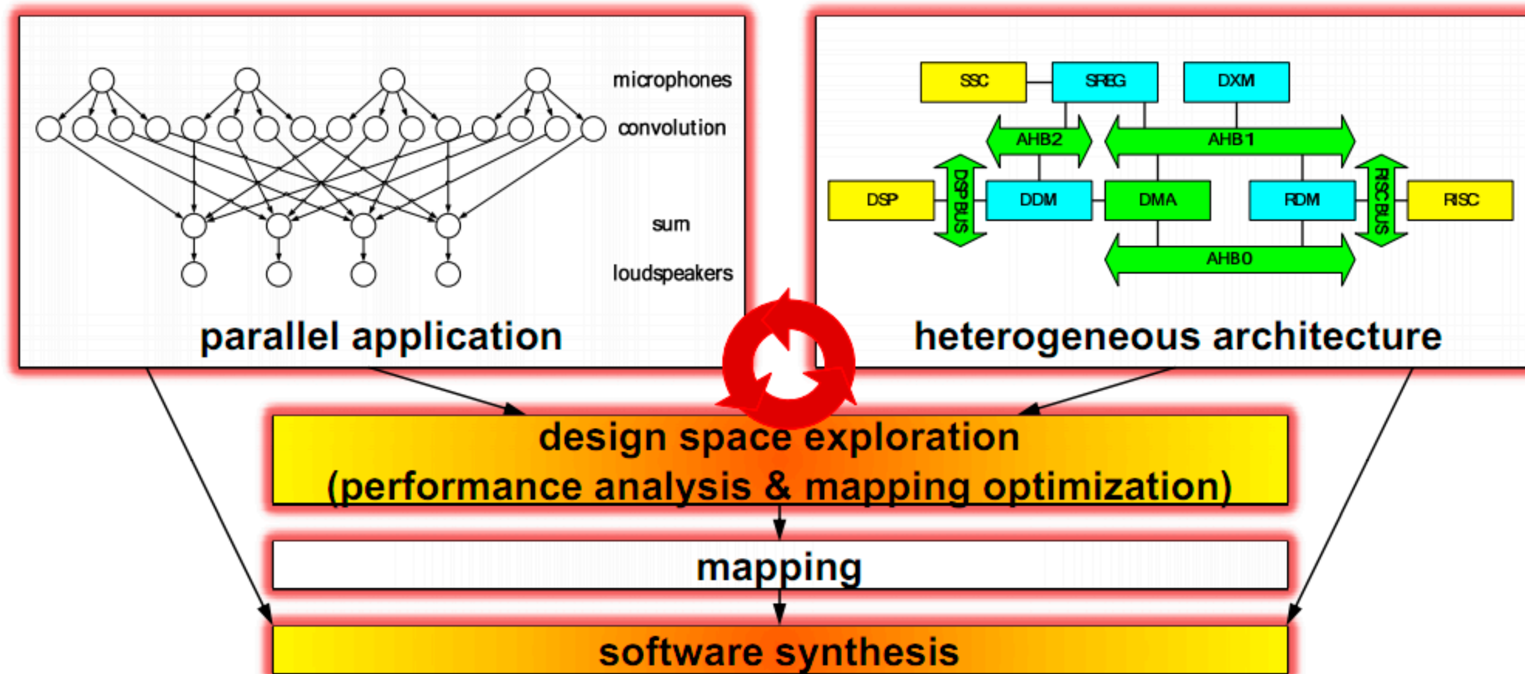
© m. engel, p. marwedel, informatik 12, 2011

SHAPES-Programmiermodell

- Anwendung = verschiedene Prozesse, durch FIFO-Kanäle zu einem Prozessnetzwerk verbunden
  - Kein gemeinsamer Speicher
  - Alle Kommunikation findet über Kanäle statt
- Implementierung eines Algorithmus auf SHAPES:
  - Extraktion der Task-Level-Parallelität des Algorithmus durch den Programmierer
  - Implementierung der zugehörigen Prozesse $P_1 \ldots P_n$
  - …und des Netzwerks N, das die Verbindungen zwischen den n Prozessen beschreibt
- Programmierer beschreibt Parallelität für DOL
- DOL nutzt diese so gut wie möglich bei der Abbildung

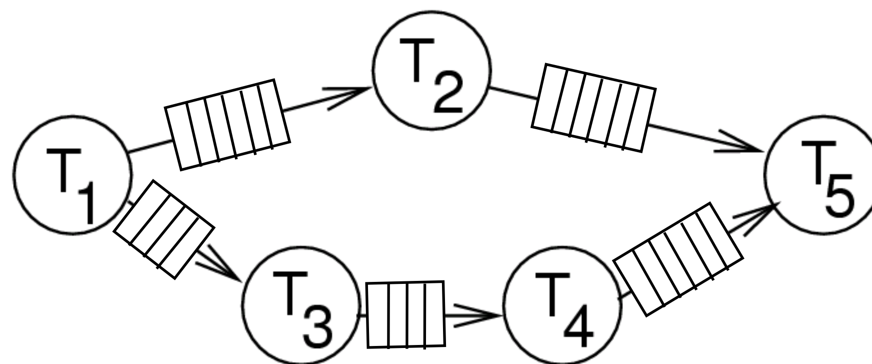
SHAPES-Programmiermodell

- Erweiterung von Kahn-Prozessnetzwerken
- Modell für parallele Berechnungen:
  - Programm besteht aus mehreren Prozessen, die miteinander durch FIFO-Kanäle verbunden sind



Kahn-Prozessnetzwerke

- Jede Komponente ist ein Programm/Task/Prozess (kein endlicher Automat)
- Kommunikation über FIFOs ohne Überlauf
  ☞ Schreibvorgänge müssen nie warten,
  ☞ Lesevorgänge warten, wenn FIFO leer ist
- Nur ein Sender und ein Empfänger je FIFO



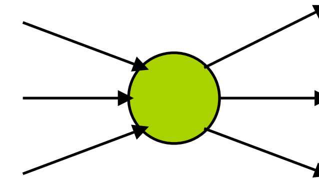
Beispiel

```
Process f(in int u, in int v, out int w){
  int i; bool b = true;
  for (;;) {
   i= b ? wait(u) : wait(v);
                   //wait returns next token in FIFO, waits if empty
   send (i,w);    //writes a token into a FIFO w/o blocking
   b = !b;
   }
```

© R. Gupta (UCSD), W. Wolf (Princeton), 2003

Eigenschaften von KPNs

- Kommunikation ausschließlich über Kanäle
- Abbildung von ≥1 Eingabekanal auf ≥1 Ausgabekanal
- Kanäle übertragen Informationen innterhalb einer nicht vorhersagbaren, aber endlichen Zeitspanne
- Im allg. sind Ausführungszeiten nicht bekannt



© m. engel, p. marwedel, informatik 12, 2011

Eigenschaften von KPNs

- Prozesse eines KPN sind monoton
  - Nur Teilinformation über Gesamteingabe erforderlich, um partielle Ausgabe zu erzeugen
  - Anders gesagt: Zukünftige Eingaben beeinflussen nur zukünftige Ausgaben
- Wichtige Eigenschaft → ermöglicht Parallelität
  - Prozess kann mit Berechnung beginnen, bevor *alle E*ingabewerte angekommen sind

Eleganz von KPN-Modellen

- Ein Prozess kann nicht prüfen, ob Daten verfügbar sind, bevor ein (blockierender) Lesezugriff aufgerufen wird
- Ein Prozess kann nicht auf Daten von mehr als einem Port gleichzeitig warten
- Lesereihenfolge hängt also nur von Daten, nicht von deren Ankunftszeit ab
- Daher sind KPNs **determinate** (!): eine bestimmte Eingabefolge führt immer zur exakt gleichen Ausgabe-folge, unabh. von Scheduling/Geschwindigkeit der Knoten

☞ Wichtig für eingebettete Systeme: Jede Kombination aus langsamer/schneller Simulation und Hardware liefert selbes Ergebnis!

Berechnungsstärke und Analysierbarkeit

- KPNs sind Turing-vollständig
- Herausforderung: Scheduling von KPNs, so dass sich keine Token in den FIFOs ansammeln
- KPNs sind schwierig zu analysieren
  - z.B. Bestimmung der erforderlichen maximalen Puffergröße
- Anzahl der Prozesse ist statisch festgelegt

KPN-API

- Prozess ist ein normales sequentielles Programm
- Erweiterung durch Kommunikationsschnittstelle mit zwei Funktionen:
  - `wait(U)` führt blockierenden Lesezugriff auf Kanal U durch
  - Blockiert den Prozess, bis Daten vorhanden
  - `send I on W` schreibt Variable `I` auf Kanal `W`
  - Prozesse können jederzeit Daten über einen Kanal senden
  - Kommunikation erfolgt also über unendlich große FIFOs

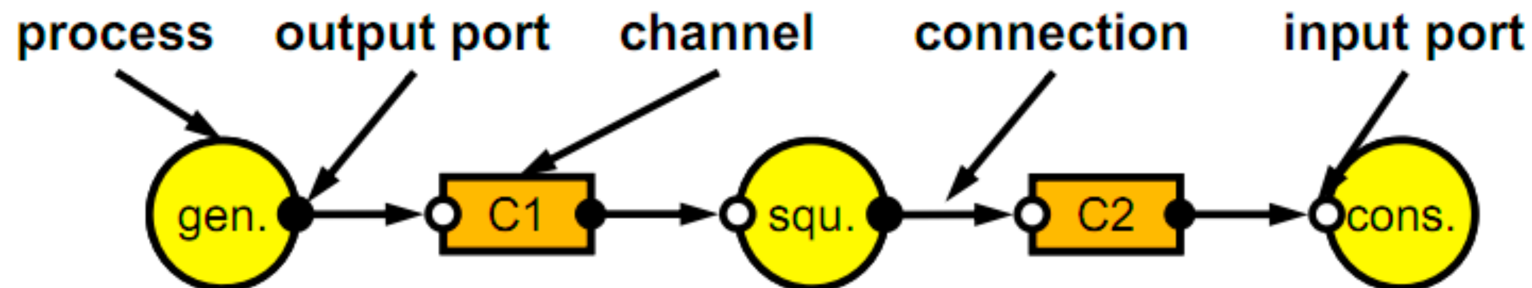
© m. engel, p. marwedel, informatik 12, 2011

SHAPES-API: Unterschiede zu KPNs

- SHAPES DOL API enthält grundlegende Kommunikationsfunktionen `DOL_write()` und `DOL_read()`
- Unterschiede zu KPNs:
  - Unendlich große FIFOs nicht realisierbar, jeder Kabal besitzt maximale Puffergröße
    - DOL_write()-Funktion blockiert den Aufrufer, wenn FIFO voll
  - KPN ermöglichen *nicht* zu testen, ob ein Kanal `U` Daten enthält, bevor `wait(U)` aufgerufen wird
    - Evtl. zu restriktiv und ineffizient
  - Zwei zusätzliche Funktionen erlauben Lese-/ Schreibtest: `DOL_rtest()` und `DOL_wtest()`

SHAPES-API: Unterschiede zu KPNs

- Berechnungsmodell: Kahn-Processnetzwerk
  - Koordination in XML mit Performance-Annotationen
  - Prozessfunktionalität: C/C++ mit spezieller DOL API
- Skalierbarkeit: “iterators” für große Systeme mit vielen Tiles



SHAPES-API: Unterschiede zu KPNs (2)

- Ende der Simulation (oder der Anwendung) durch Aufruf von DOL_detach(): Prozess kann sich selbst beenden
  - Simulation endet, wenn alle Prozesse detached sind
- Vor Aufruf der blockierenden Lesefunktion kann festgestellt werden, ob die FIFO bereits Daten enthält
  - Ermöglicht Laufzeitselektion des Leseports, wenn mehr als ein Eingabeport vorhanden
  - Kann eingesetzt werden, muss aber nicht
  - Zerstört die *determinate*-Eigenschaft!
  - Vorgegebene feste Reihenfolge von Portzugriffen senkt Kommunikationsaufwand und steigert Vorhersagbarkeit

SHAPES-API: Überblick

DOL_read(port, buffer, length, process) Reads *length* bytes from *port* and stores the obtained data in *buffer*. If less than *length* bytes are available, the calling process is blocked.
DOL_write(port, buffer, length, process) Writes *length* bytes from *buffer* to *port*. If the FIFO connected to *port* has less than *length* bytes of free space, the calling process is blocked.
DOL_rtest(port, length, process) Checks whether *length* bytes can be read from *port*.
DOL_wtest(port, length, process) Checks whether *length* bytes can be written to *port*.
DOL_detach(process) Detaches the process and prevents the scheduler from firing the process again.

Vorteile des SHAPES-Ansatzes

- Plattformbasierter Entwurf ermöglicht es, mit der Anwendungsentwicklung zu beginnen, ohne dass die endgültige Hardwarearchitektur fesgelegt ist.
  - DOL ermöglicht funktionale Verifikation einer Anwendung vom Beginn der Entwicklung an
- Anwendungen — als Prozessnetzwerk spezifiziert und unter Verwendung des Programmiermodells und der Prozessdefinitionen — kann zur Simulation automatisch in SystemC-Code transformiert werden
  - Ermöglicht Simulation einer Echtzeitumgebung
- Später: Synthese auf reale Hardware-Plattform

© m. engel, p. marwedel, informatik 12, 2011

Transformation inSystemC

- Synthese
  - DOL-Prozesse und FIFOs: SystemC-Threads und Channels
  - SystemC main: Initialisierung und Schedulin
- Eigenschaften
  - Ausführung: native Code, untimed
  - Debugging: Standardwerkzeuge (gdb)

DOL-Toolflow

Ziele

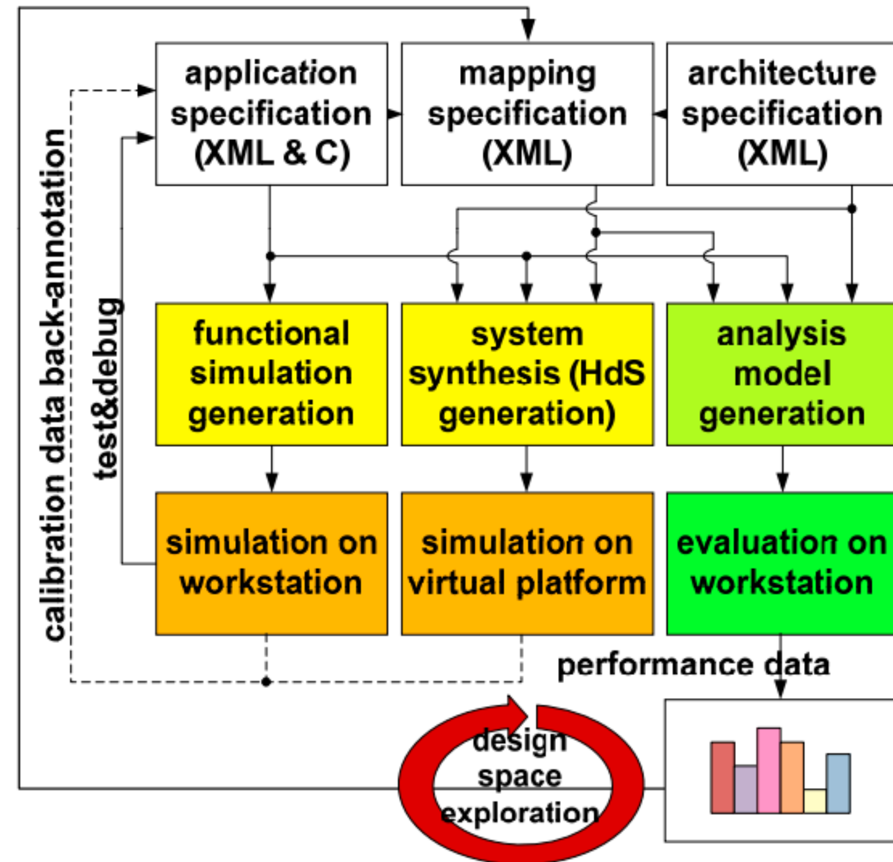
- Effizienz
- Vorhersagbarkeit
- Portabilität

Anforderungen

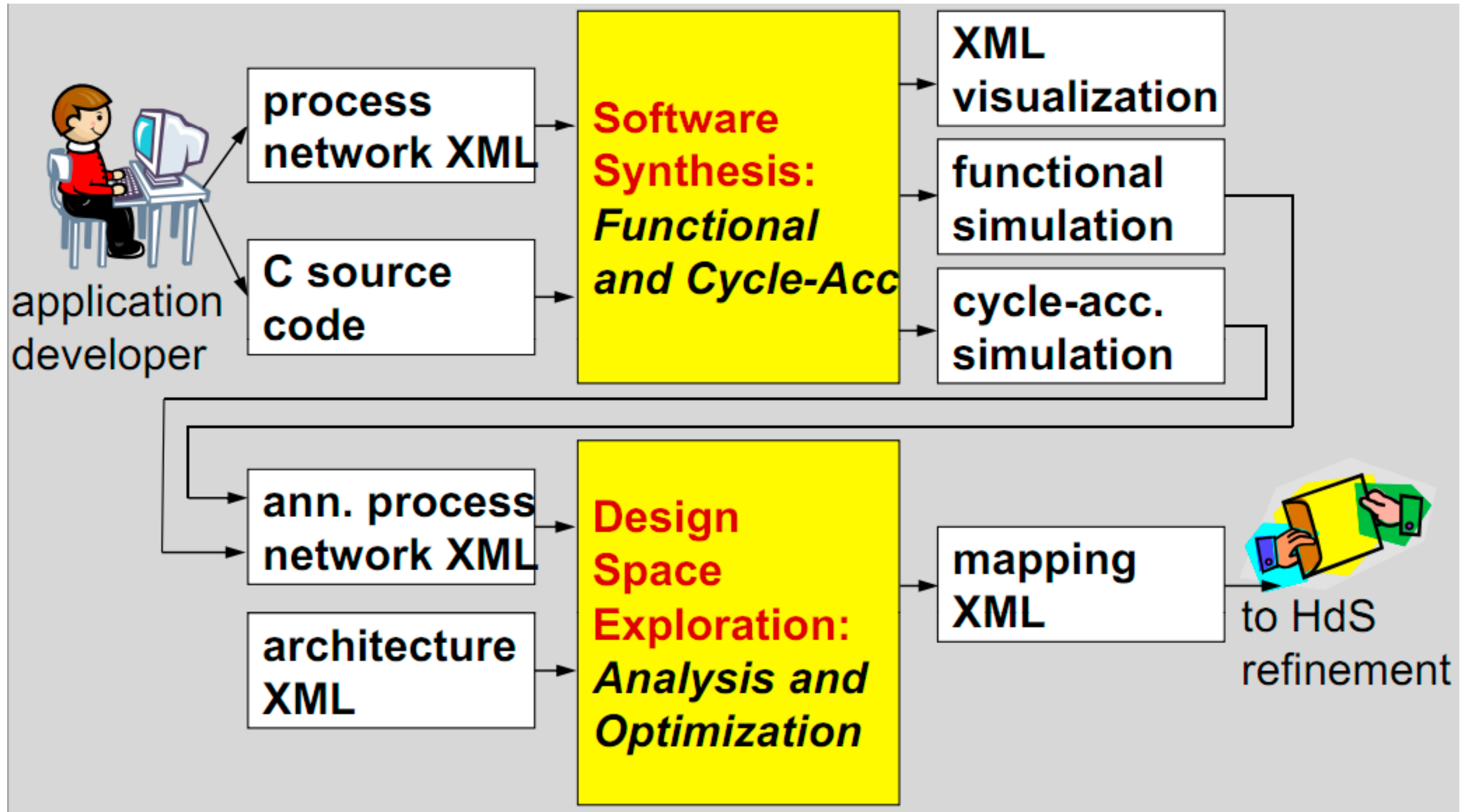
- Skalierbare Spezifikation
- Automatische Synthese
- Entwurfsraumerkundung
- Systemweite Performanceanalyse

Stärken

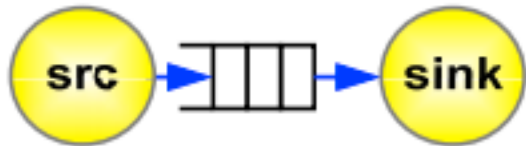
- Abstraktion und Automatisierung



DOL-Entwurfsablauf



DOL-Beispiel

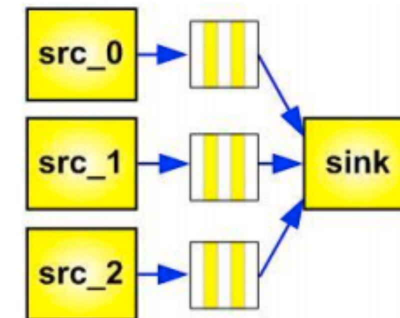



```
01: <process name="src">
02:   <port type="output" name="out"/>
03:   <source type="c" location="src.c"/>
04: </process>
05:
06: <process name="sink">
07:   <port type="input" name="in"/>
08:   <source type="c" location="sink.c"/>
09: </process>
10:
11: <connection name="con_src_fifo">
12:   <origin name="src"><port name="out"/></origin>
13:   <target name="sink"><port name="in"/></target>
14: </connection>
```

```
01: procedure INIT(Process p)
02:   //initialize process state
03: end procedure
04:
05: procedure FIRE(Process p)
06:   DOL_READ(input, size, buf)
07:   //processing
08:   DOL_WRITE(output, size, buf)
09: end procedure
```

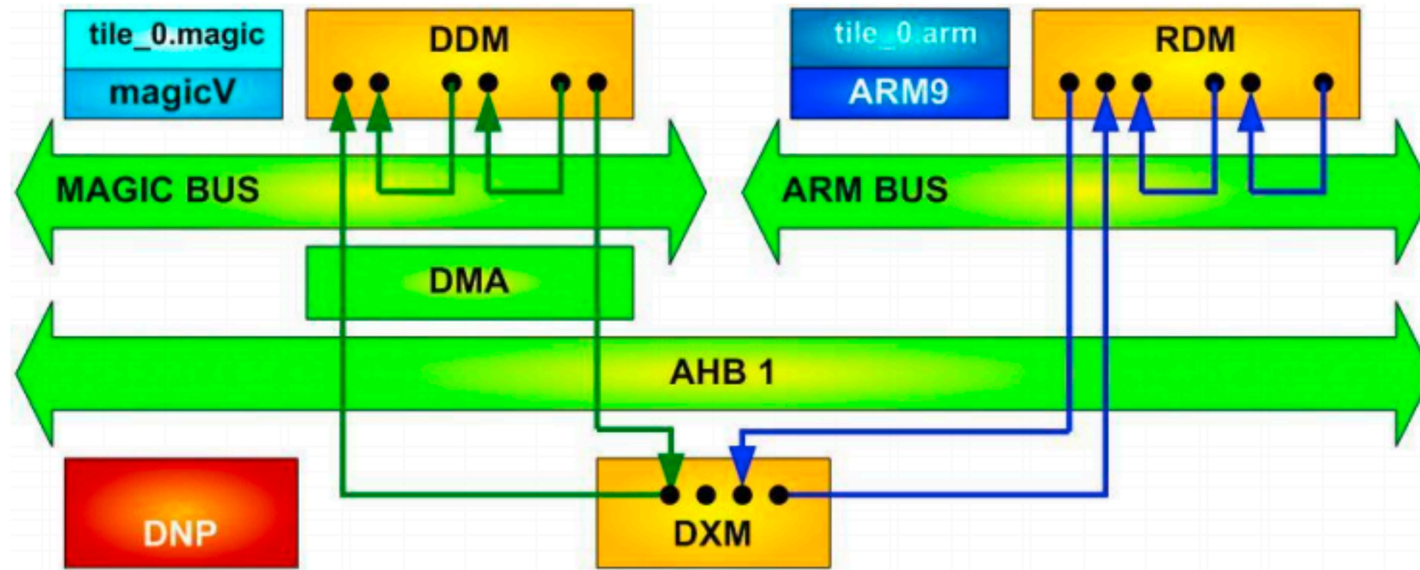
DOL-Beispiel: Skalierbarkeit

```
01: <iterator variable="i" range="N">
02:   <process name="src">
03:   <append function="i"/>
04:      <port type="output" name="out"/>
05:      <source type="c" location="src.c"/>
06:   </process>
07: </iterator>
```



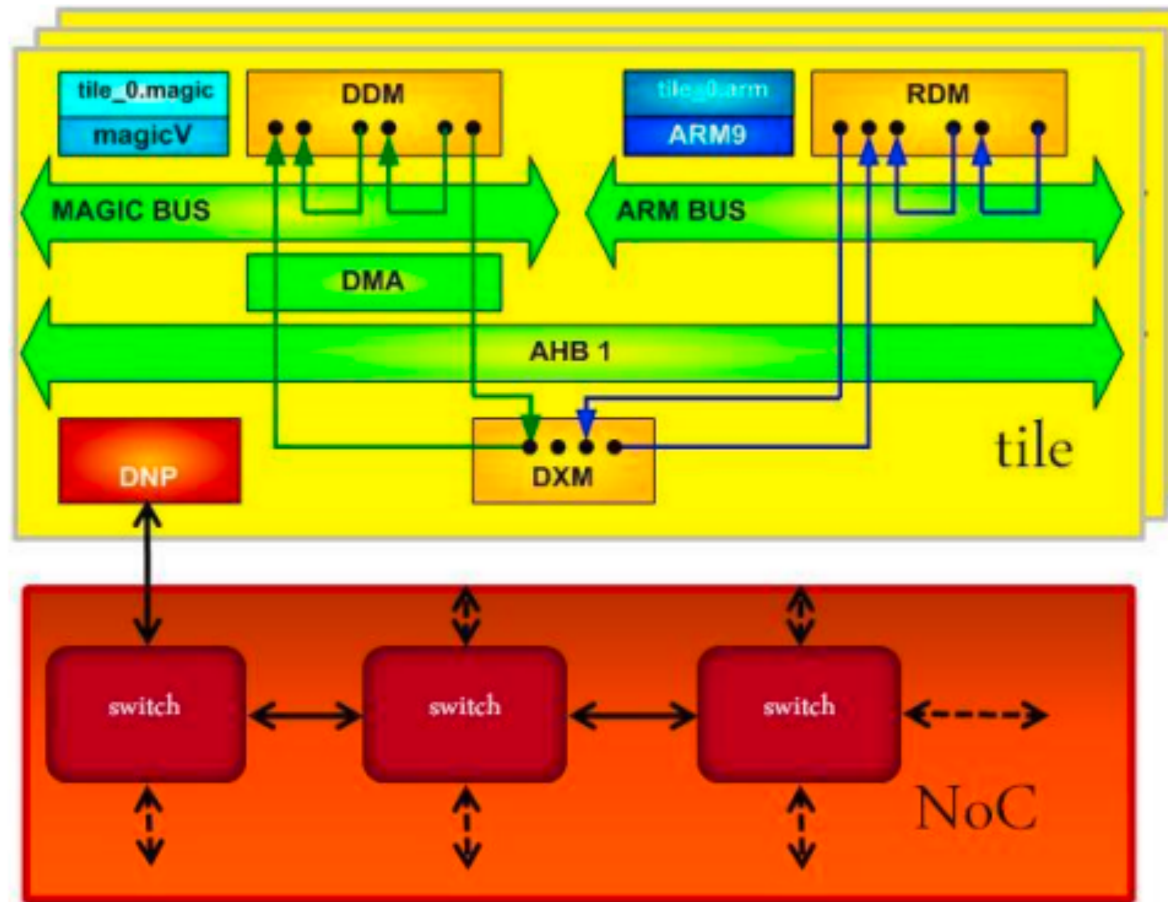
© m. engel, p. marwedel, informatik 12, 2011

Spezifikation der abstrakten Plattform



- Elemente
  - Struktur: Prozessoren, Peripherie, Speicher, Busse
  - Verbindungen: explize Lese-/Schreibpfade
  - Performancedaten (Latenzen, Bandbreite der Kanäle)
- Spezifikation: XML mit “iterators”

Skalierbarkeit der abstrakten Plattform



- Skalierbarkeit der Plattform durch Iteratoren in XML

Spezifikation der Abbildung

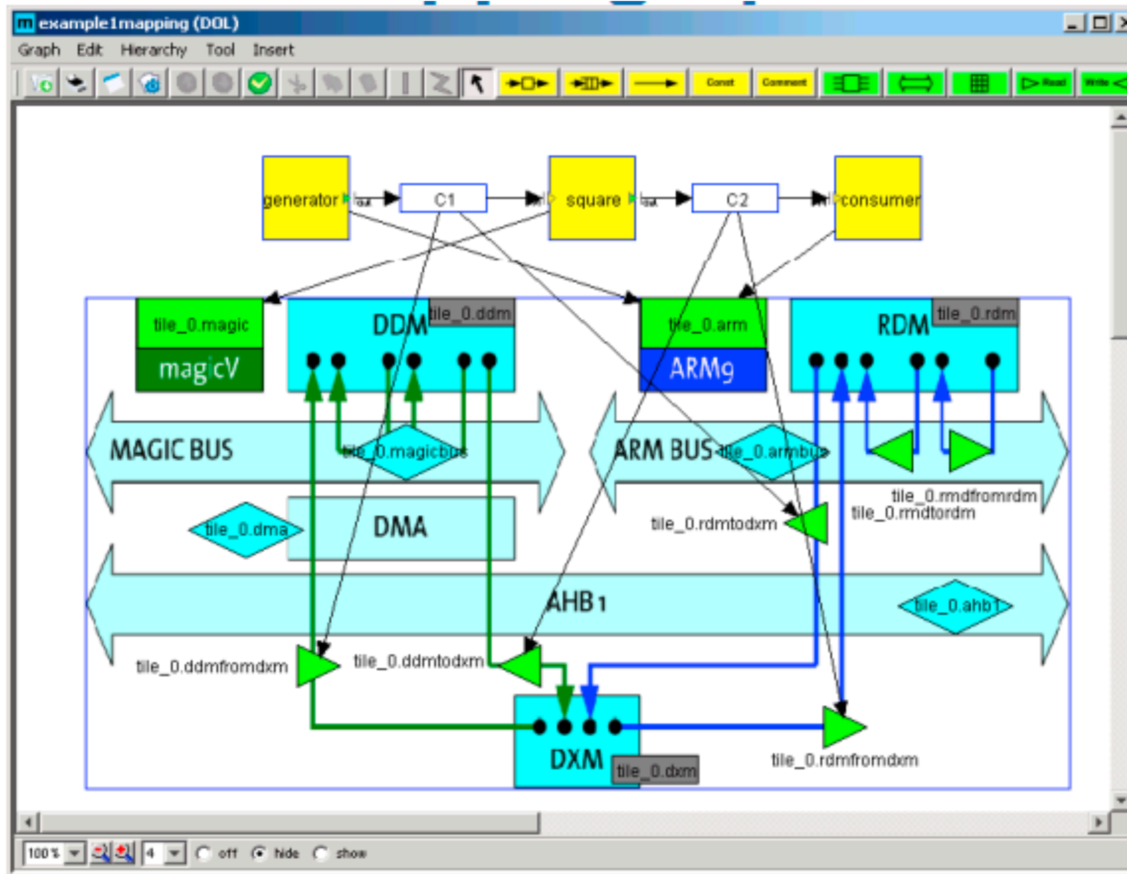


Abbildung:

- Bindung
  - Prozesse an Prozessoren
  - SW-Kanäle an R/W-Pfade
- Scheduling
- Constraints

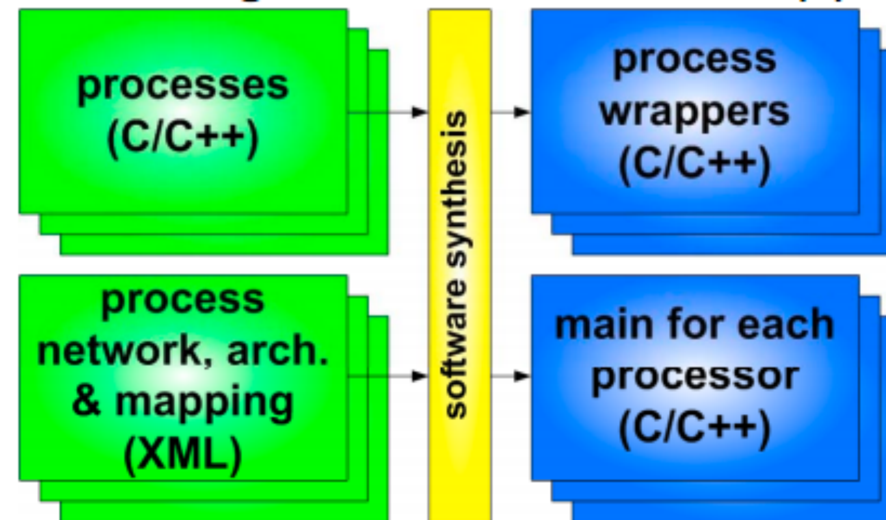
Strategien:

- Manuell (GUI)
- Automatisch

DOL-Softwaresynthese

Ziel

- Plattformunabhängige Spezifikation der Anwendung → Implementatierung entsprechend Architektur und Abbildung



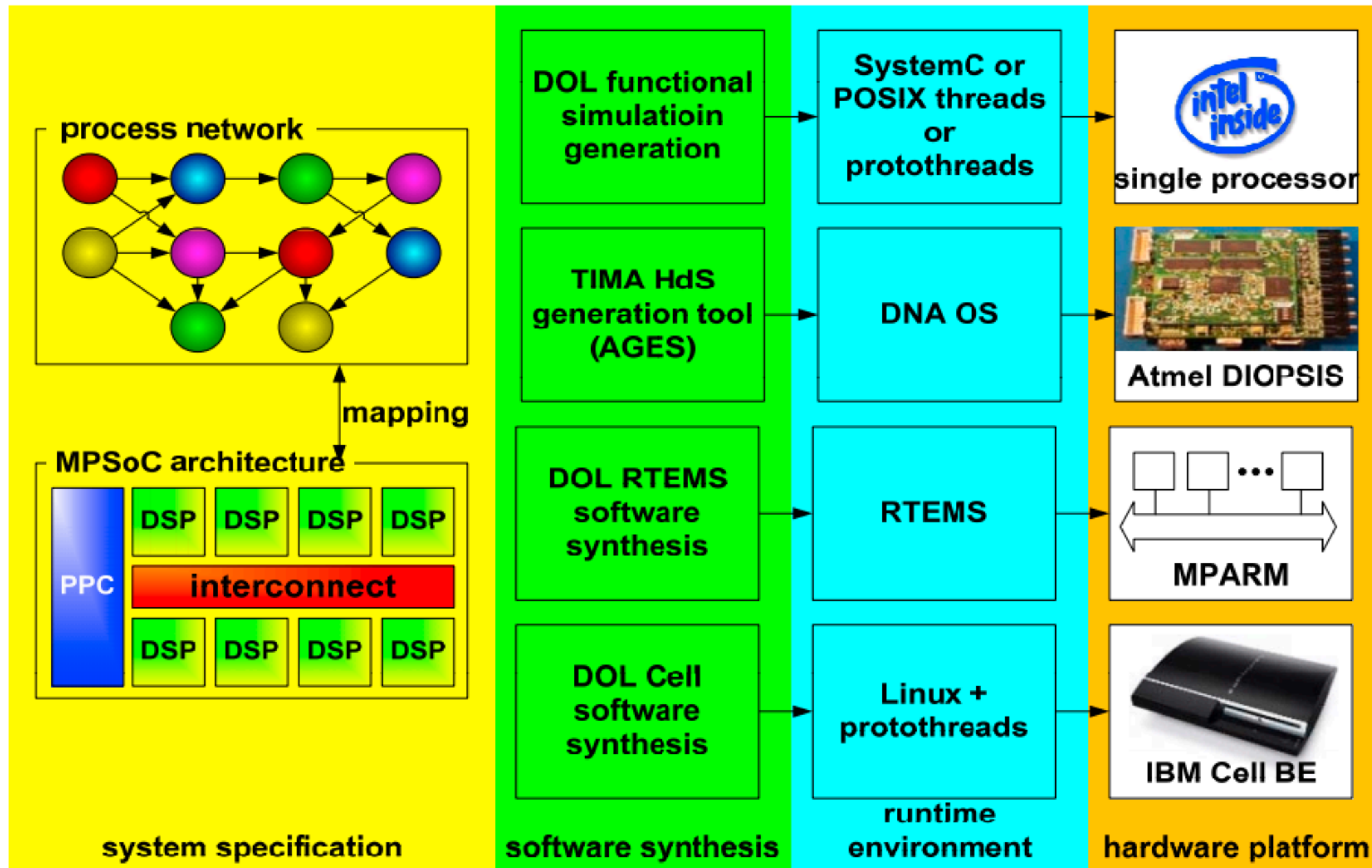
Probleme

- Effizienz
- Vorhersagbarkeit
- Automatisierung

Approach

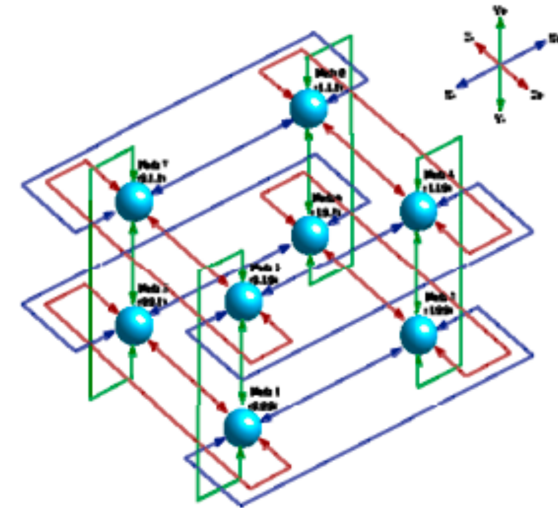
- Codeerzeugung und Source-to-Source-Transformation
- Trennung von Belangen: generische Laufzeitumgebung ↔ anwendungsspezifische Elemente

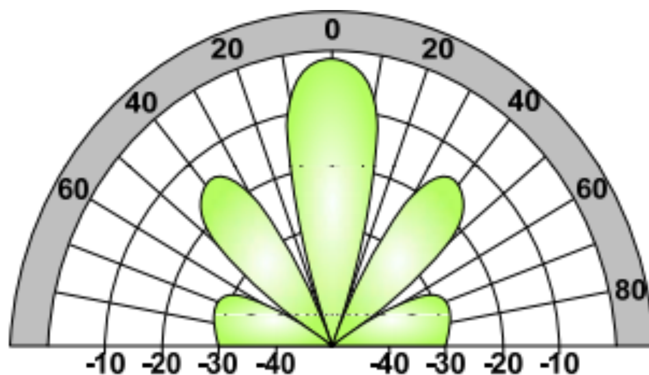
Synthese-Backends:
“Write Once, Run Everywhere”

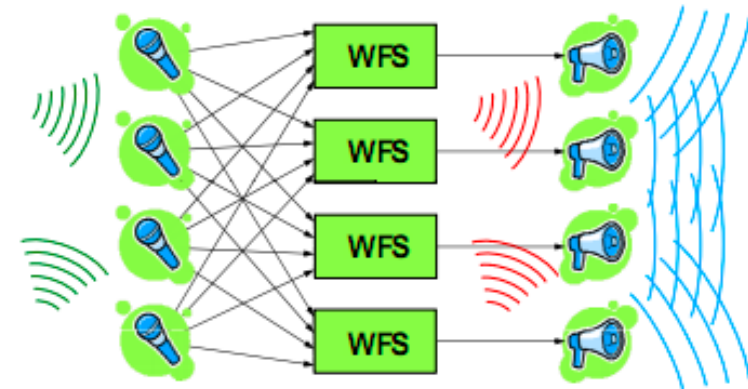


Beispiel: Abbildung auf CELL

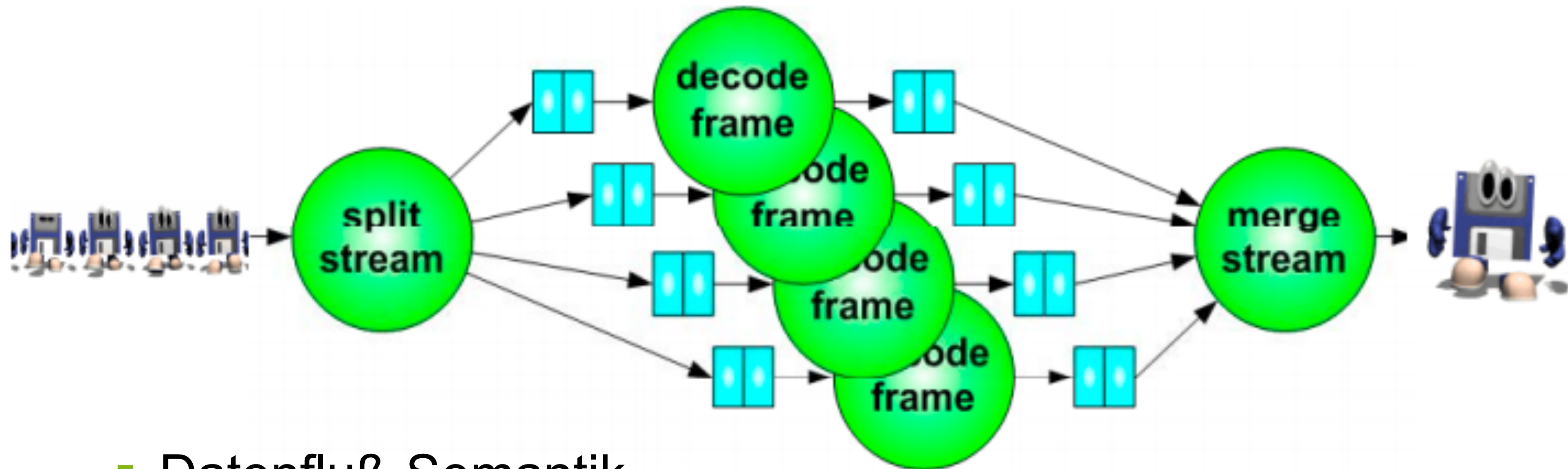
- Datenstrom-Anwendungen
- Anwendungsbereiche
  - Verbraucherelektronik
  - Kommunikationssysteme
  - Medizinische Systeme usw.
- Echtzeitanwendungen
  - (Array) Signalverarbeitung
  - Audio & Video (De-)Codierung
  - Hochleistungsrechnen







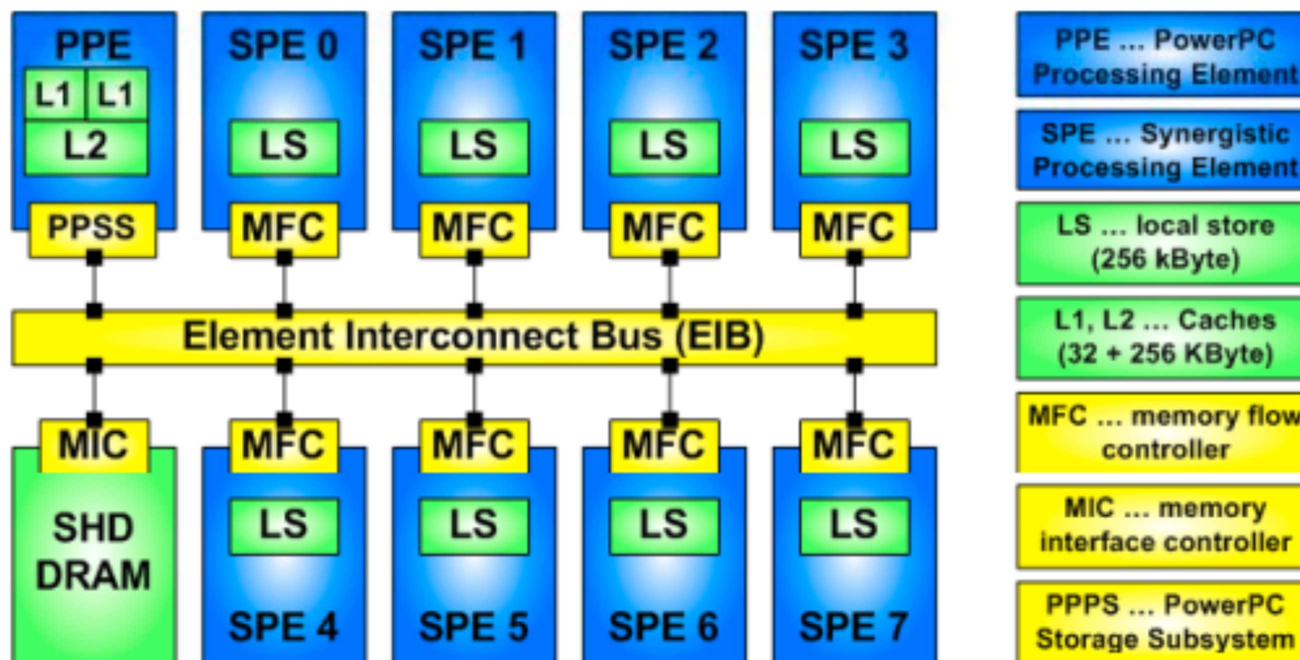
Datenstromanwendungen als KPN



- Datenfluß-Semantik
  - Entspricht Struktur vieler Streaming-Anwendungen
  - Trennt Berechung von Kommunikation
  - Ermöglicht Entwurfsautomatisierung
- Untimed-Berechnungsmodell
  - Ermöglicht Implementierung auf MPSoCs

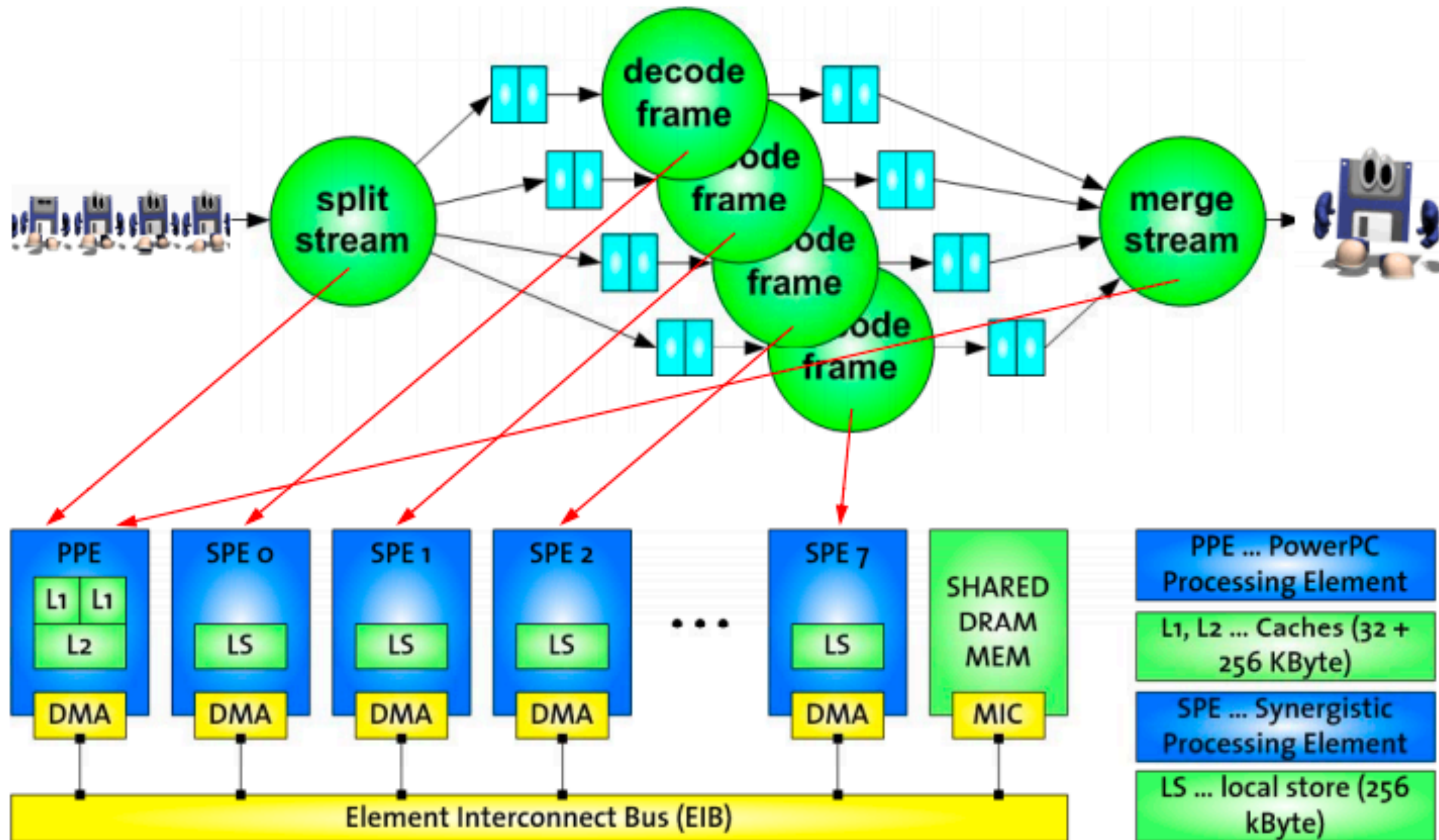
CELL: Verteilter Speicher

- IBM/Sony/Toshiba CELL BE: PowerPC und 8 SPEs, verbunden über Ring
- Problem: effiziente Programmierung, keine Parallelität in SPEs

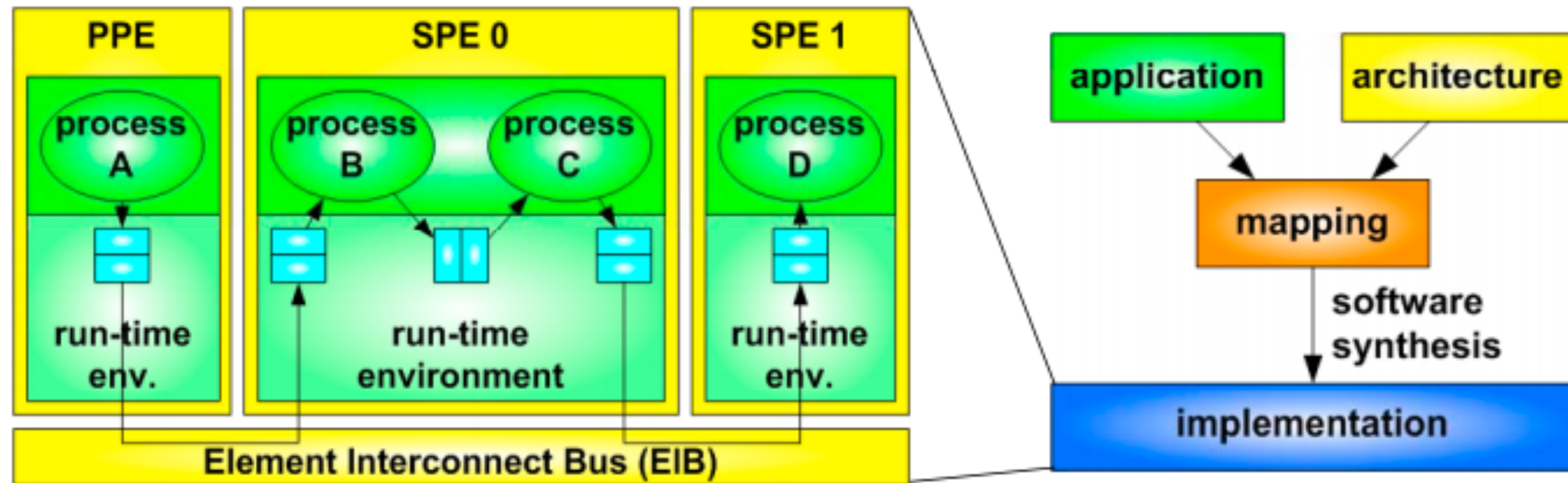


CELL: Abbildung

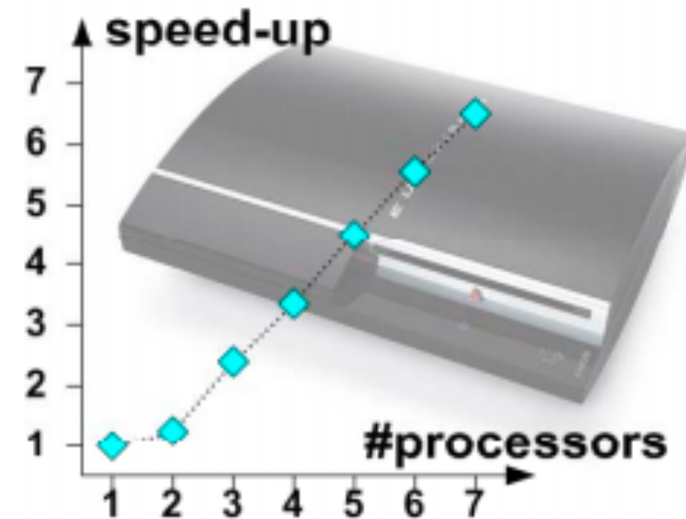
- Ziel: Effiziente Ausführung von auf CELL-MPSoC



CELL: Abbildung



- Ziel
  - Effiziente Ausführung auf CELL
- Hauptprobleme
  - Optimierung d. Abb. → später
  - *Effiziente Laufzeitumgebung*
  - *Automatische Softwaresynthese*

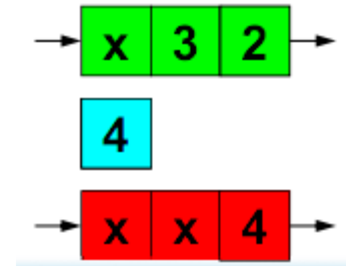


© m. engel, p. marwedel, informatik 12, 2011

Windowed FIFOs

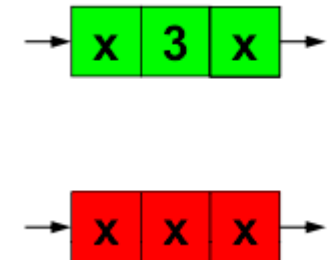
- **Standard FIFO**

```
float i;
read(PORT_IN, &i, sizeof(float));
i = i * i;
write(PORT_OUT, &i, sizeof(float));
```



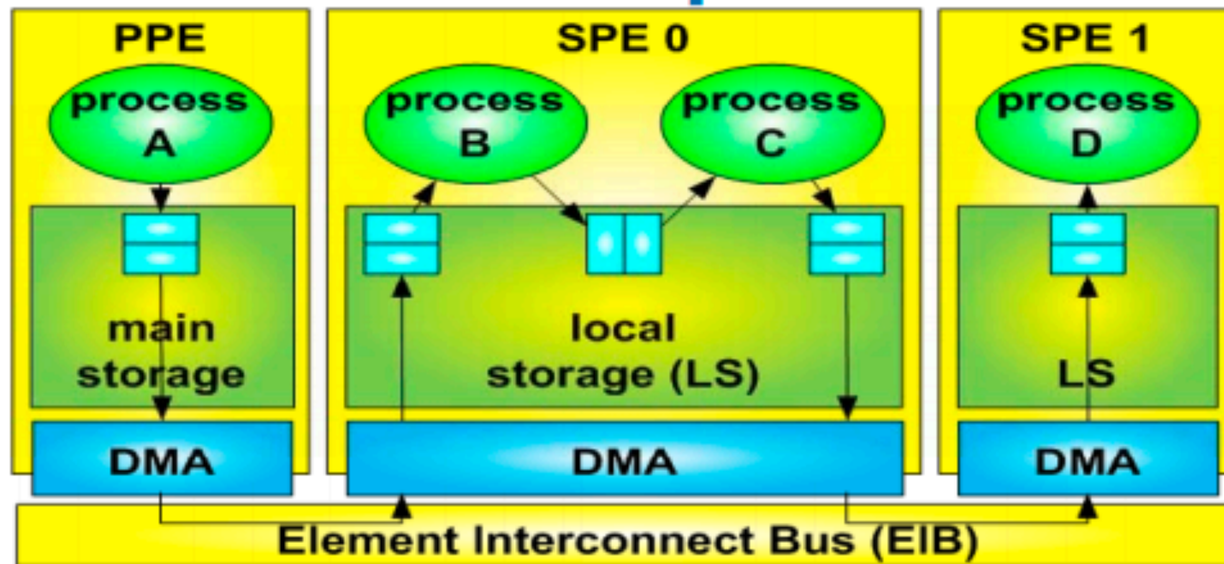
- **Windowed FIFO**

```
float *i, *j;
capture(PORT_IN, &i, sizeof(float));
reserve(PORT_OUT, &j, sizeof(float));
*j = *i * *i;
consume(PORT_IN);
release(PORT_OUT);
```



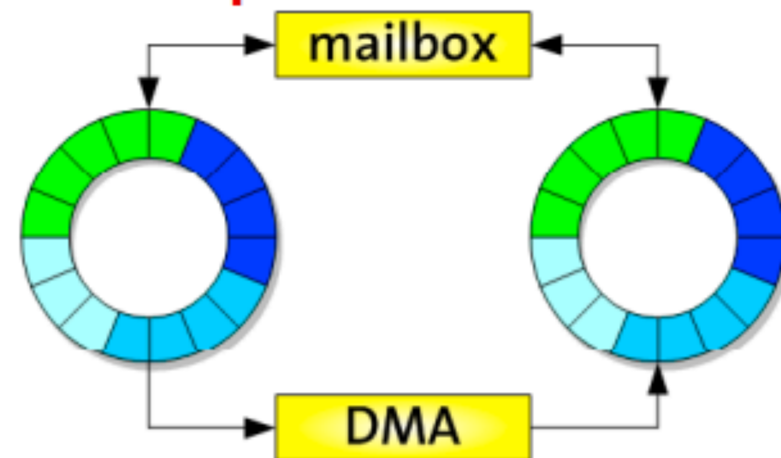
- Anmerkungen
  - Windowed FIFOs erhalten KPN-Semantik [Huang, ASAP07]
  - Interprozessor windowed FIFO (via DMA) is einziger plattformabhängiger Teil der Laufzeitumgebung

Windowed FIFOs: Implementierung



intra-processor WFIFO

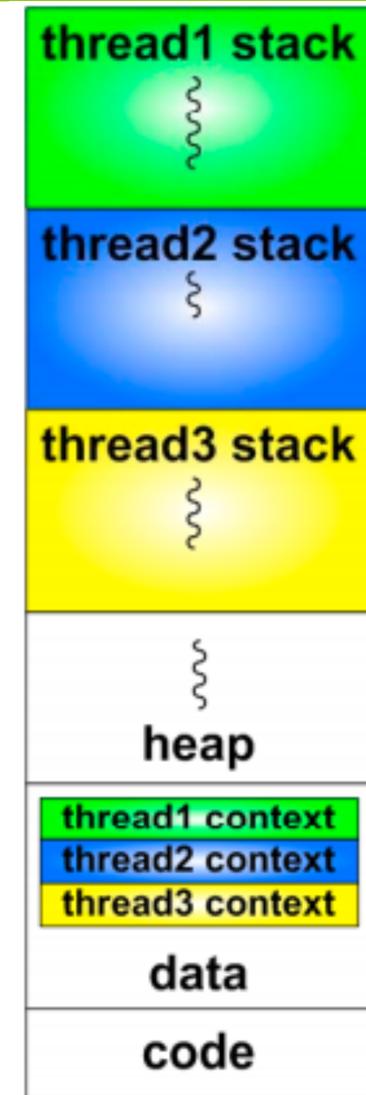
inter-processor WFIFO



© m. engel, p. marwedel, informatik 12, 2011

Quasi-Parallelität mit Threads

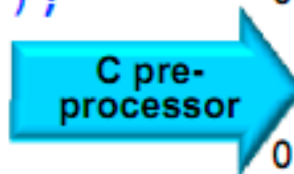
- Komponenten
  - Programmcode und -daten
  - Kontext: Registers, PC, SP usw.
  - Stack
- Einschränkungen
  - Hoher Kontextswitch-Aufwand
  - Bsp.: SPE hat 128 16Byte Register
    → 4kB bei Kontextswitch kopieren
  - Mehrere Stacks verbrauchen Speicher
  - Assembler zur Implementierung notwendig (Register sichern usw.)



Protothreads [Adam Dunkels 2005]

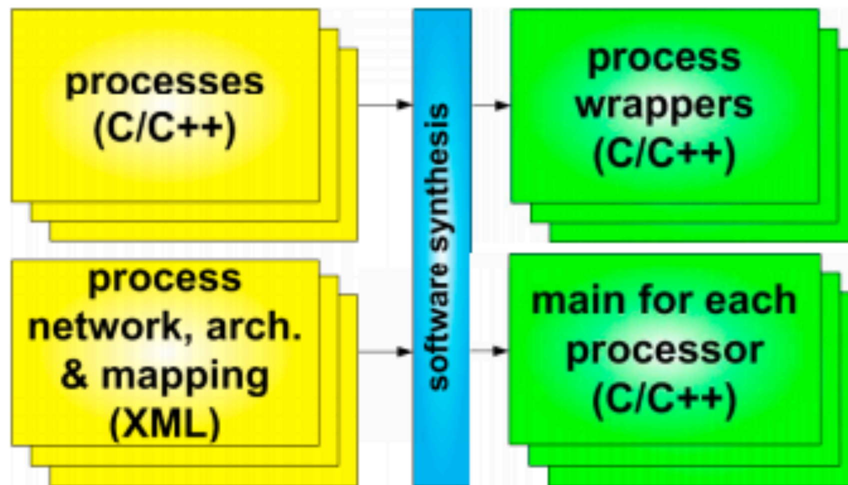
```
struct pt{unsigned short lc;};
#define PT_BEGIN(pt)                      switch(pt->lc){ case 0:
#define PT_WAIT_UNTIL(pt, cond)           pt->lc=__LINE__; case __LINE__:
                                          if(!(cond))
                                              return 0
#define PT_END(pt)                        }
                                          pt->lc=0; return 1
```

```
01 int protothread(struct pt *pt) {
02   PT_BEGIN(pt);
03   ...
04   PT_WAIT_UNTIL(pt,
         wfifo->capture(...));
05   ...
06   PT_END(pt);
07 }
```




```
01 int protothread(struct pt *pt){
02   switch(pt->lc){ case 0:
03     ...
04     pt->lc=4; case 4:
       if(!wfifo->capture(...))
           return 0;
05     ...
06   }
     pt->lc=0; return 1;
07 }
```

Automatische Software-Synthese



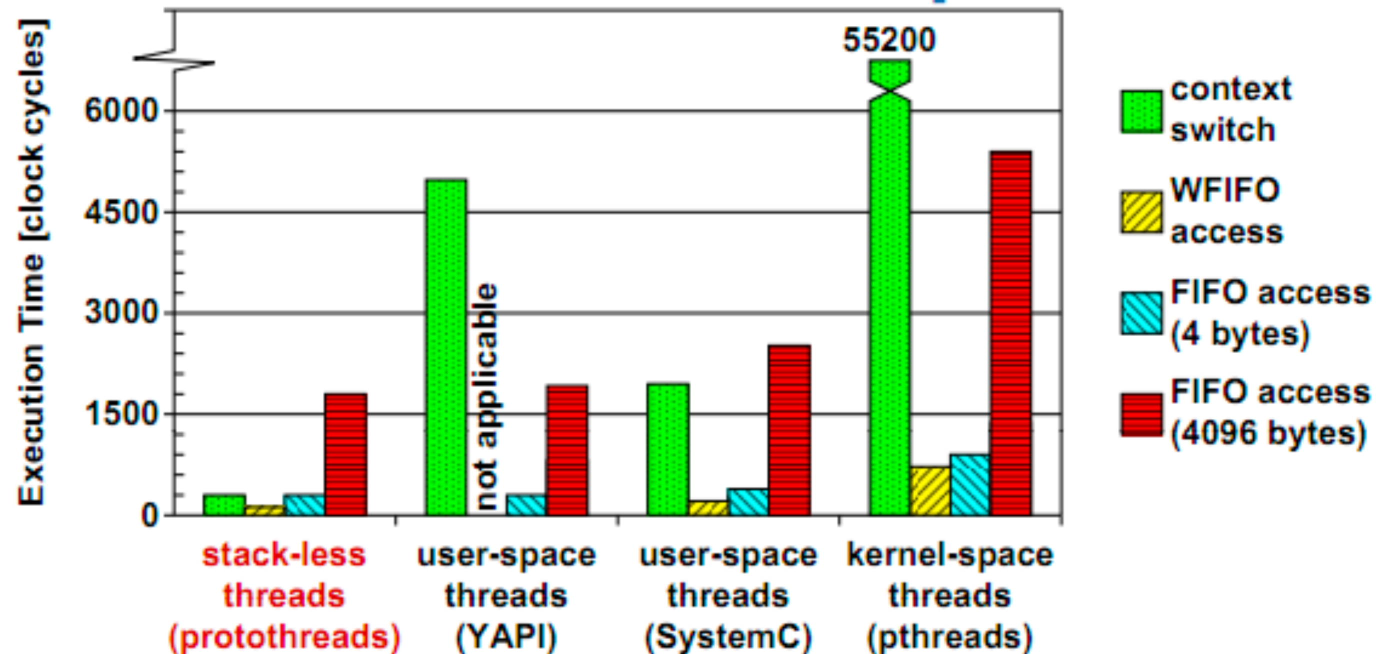

```
int main(){
   //init process network

   while(1){
    producer_fire(p_data);
    square_fire(s_data);
    consumer_fire(c_data);
   }
}
```

```
square_fire(LocalData p){
   READ(PORT_IN,&(p->i),4,p);
   p->i = p->i * p->i;
   WRITE(PORT_OUT,&(p->i),4,p);
}
```

software synthesis

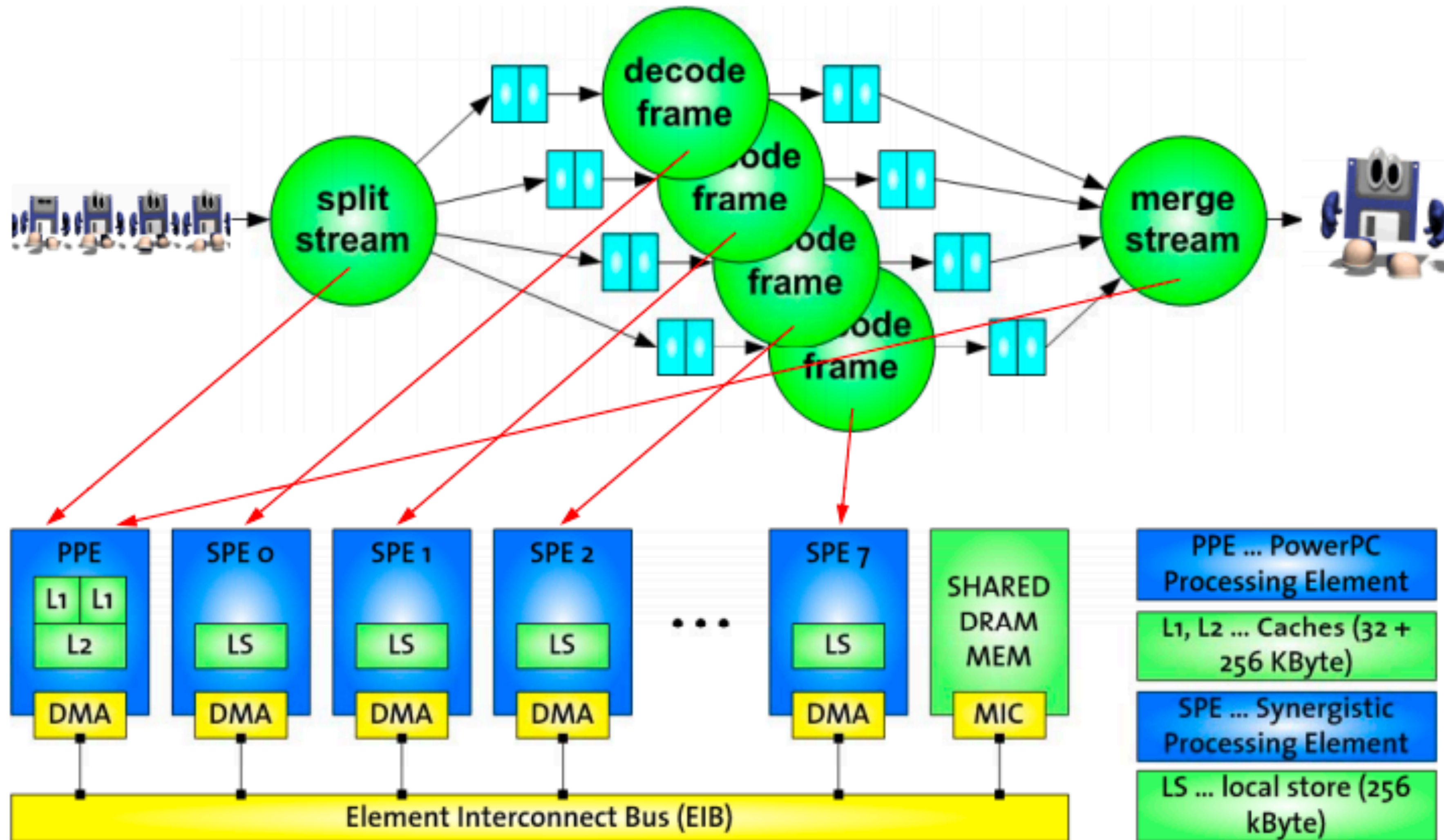
```
square_fire(LocalData p){
   PT_BEGIN(p);
   PT_WAIT_UNTIL(p,
   p->fifo_in->READ(&(p->i),4));
   p->i = p->i * p->i;
   PT_WAIT_UNTIL(p,
   p->fifo_out->WRITE(&(p->i),4));
   PT_END(p);
}
```

Evaluation von Thread-/FIFO-Implementierungen

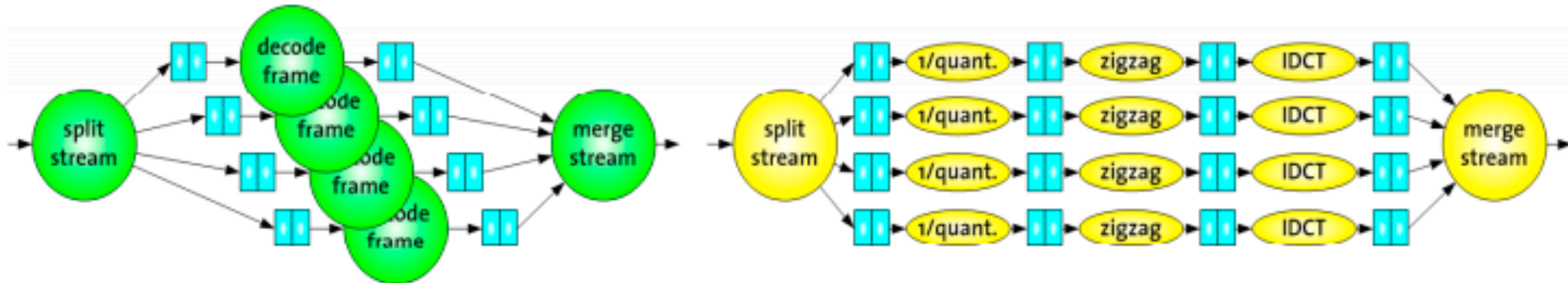


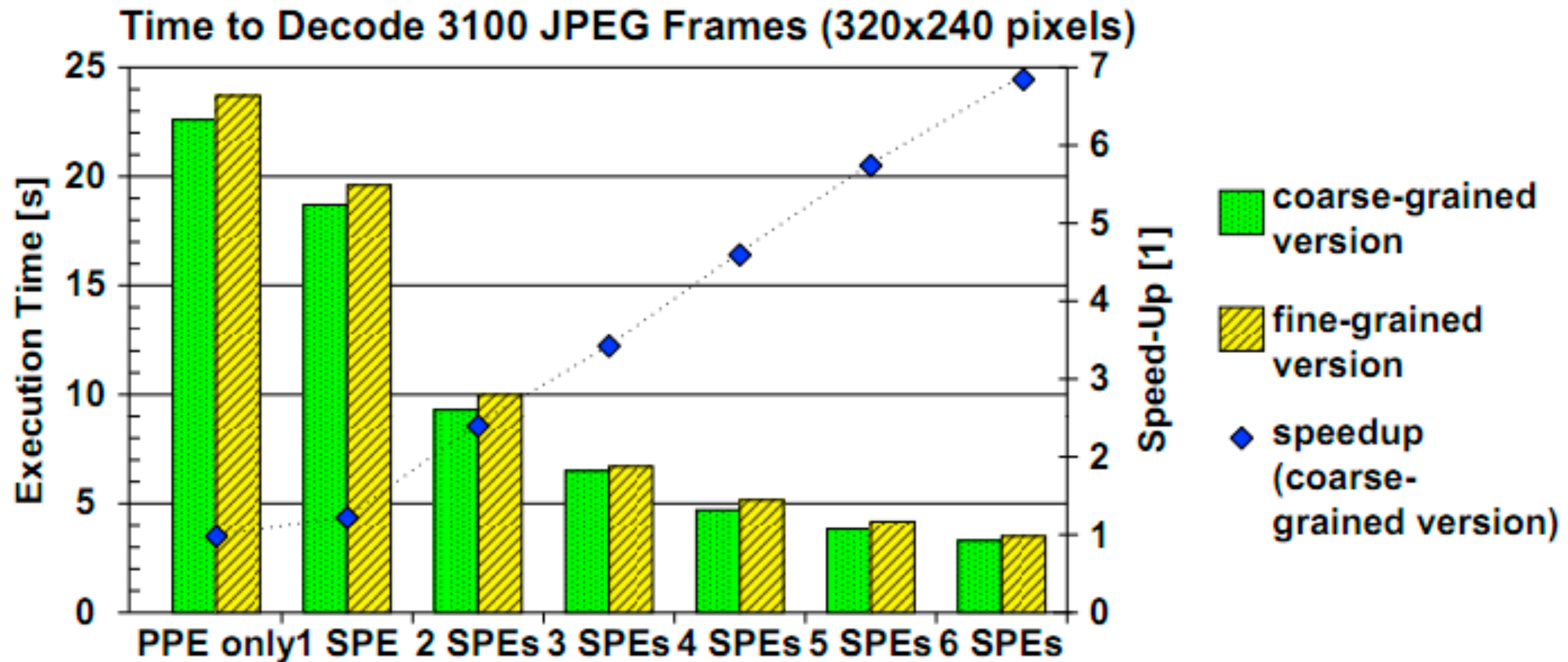
- Protothreads erfordern kleinsten Kontextswitch-Aufwand
  - 8x –18x schneller als Userspace-Threads
  - 200x schneller als Kernelspace-Threads
- Windowed FIFO erheblich effizienter für große Zugriffe
- Protothreads sind effizient
  - Kontextswitch ~300 Zyklen, wFIFO-Zugriff ~150 Zy.

Abbildung: MJPEG-Decoder auf CELL



Unterschiedliche Granularitäten





Zusammenfassung

- Problem der Abbildung
- KPNs
- SHAPES/DOL
  - API und Unterschiede zu KPNs
  - Architektur
  - Entwurfsfluss
- Abbildung auf CELL
  - Laufzeitumgebung
  - Softwaresynthese