Multithreaded Code from Synchronous Programs: Generating Software Pipelines for OpenMP

Daniel Baudisch, Jens Brandt, Klaus Schneider

Embedded Systems Group
Department of Computer Science
University of Kaiserslautern, Germany
Motivation

1. Introduction

2. Creating Multi-Threaded Code

3. Results
1 Introduction

2 Creating Multi-Threaded Code

3 Results
Motivation

- synchronous languages, e.g. Esterel, Lustre, Quartz can be used in Embedded Systems
- generation of single threaded code so far
- multicore processors more frequently used
  - adapt SW synthesis $\implies$ generation of multithreaded code
● synchronous languages ⇒ see talk of Mike Gemünde
● here: from synchronous guarded actions to multi-threaded code using OpenMP
Guarded Actions

System (Example)

<table>
<thead>
<tr>
<th>Interface:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
<td>$i, c$</td>
</tr>
<tr>
<td>Output:</td>
<td>$o$</td>
</tr>
<tr>
<td>Locals:</td>
<td>$x, y, z$</td>
</tr>
</tbody>
</table>

Guarded Actions:
- $c \Rightarrow o = x + y$
- $true \Rightarrow x = i \cdot i$
- $true \Rightarrow z = 2 \cdot i$
- $true \Rightarrow next(y) = z + 1$
Guarded Actions

System (Example)

Interface:
   Inputs: $i, c$
   Output: $o$
   Locals: $x, y, z$

Guarded Actions:
   $c \Rightarrow o = x + y$
   true $\Rightarrow x = i \cdot i$
   true $\Rightarrow z = 2 \cdot i$
   true $\Rightarrow \text{next}(y) = z + 1$

Dependency Graph

Multithreaded Code from Synchronous Programs (Daniel Baudisch, Jens Brandt, Klaus Schneider)
Guarded Actions

System (Example)

Interface:
  - Inputs: \( i, c \)
  - Output: \( o \)
  - Locals: \( x, y, z \)

Guarded Actions:
  - \( c \Rightarrow o = x + y \)
  - true \( \Rightarrow x = i \cdot i \)
  - true \( \Rightarrow z = 2 \cdot i \)
  - true \( \Rightarrow \text{next}(y) = z + 1 \)

Dependency Graph
Guarded Actions

System (Example)

Interface:
  Inputs: \( i, c \)
  Output: \( o \)
  Locals: \( x, y, z \)

Guarded Actions:
  \( c \Rightarrow o = x + y \)
  \( \text{true} \Rightarrow x = i \cdot i \)
  \( \text{true} \Rightarrow z = 2 \cdot i \)
  \( \text{true} \Rightarrow \text{next}(y) = z + 1 \)

Dependency Graph

- \( i \) to \( x = i \cdot i \)
- \( c \) to \( o = x + y \)
- \( x \) to \( z = 2 \cdot i \)
- \( 	ext{next}(y) = z + 1 \) to \( o \)
Guarded Actions

System (Example)

Interface:
- Inputs: \( i, c \)
- Output: \( o \)
- Locals: \( x, y, z \)

Guarded Actions:
- \( c \Rightarrow o = x + y \)
- \( \text{true} \Rightarrow x = i \cdot i \)
- \( \text{true} \Rightarrow z = 2 \cdot i \)
- \( \text{true} \Rightarrow \text{next}(y) = z + 1 \)

Dependency Graph:
- \( i \) leads to \( x = i \cdot i \)
- \( c \) leads to \( o = x + y \)
- \( \text{true} \) leads to \( z = 2 \cdot i \)
- \( \text{true} \) leads to \( \text{next}(y) = z + 1 \)
Outline

1 Introduction

2 Creating Multi-Threaded Code

3 Results
Extracting Independent Threads

First Approach

intuitive:
- group dependent actions: create "vertical slices"
- execute groups in parallel

Dependency Graph

\[
\begin{align*}
  i & \\
  c & \\
  x = i \cdot i & \\
  y & \\
  z = 2 \cdot i & \\
  c \Rightarrow o = x + y & \\
  \text{next}(y) = z + 1 & \\
\end{align*}
\]
Extracting Independent Threads

First Approach

intuitive:

- group dependent actions: create "vertical slices"
- execute groups in parallel

Dependency Graph Multithreaded

\[
M_1 \
\begin{align*}
  x &= i \cdot i \\
  \Rightarrow o &= x + y
\end{align*}
\]

\[
M_2 \
\begin{align*}
  z &= 2 \cdot i \\
  \text{next}(y) &= z + 1
\end{align*}
\]
Extracting Independent Threads

First Approach

intuitive:

- group dependent actions: create "vertical slices"
- execute groups in parallel
- due to sync. overhead: only applicable for large groups
- problem: what if creation of large groups fails?

Dependency Graph Multithreaded

Multithreaded Code from Synchronous Programs (Daniel Baudisch, Jens Brandt, Klaus Schneider)
Second Approach

- group dependent actions: create "horizontal slices"
- basic idea: execute groups in parallel like a pipeline

Example - Pipelining

```plaintext
π_1

\[ x = i \cdot i \]

π_2

\[ c \Rightarrow o = x + y \]

\[ z = 2 \cdot i \]

\[ \text{next}(y) = z + 1 \]

\[ o \]
```
Pipelining

Problems:

- How to partition dependency graph?
  ⇒ use a legal partitioning
  ⇒ optimal partition depends on target architecture (not goal of this approach)

- What about values that are read in several stages?
  ⇒ insert intermediate variables

- How to store intermediate values?
  ⇒ use queues

- Where to write values at?

- Do we require something like stalling?
  ⇒ implicit by queues
A partitioning is legal iff

- control/data flow goes to one direction
- NOTE: a delayed write access may go to a previous stage
Insertion of Intermediate Variables (IV)

- copies of variables (comparable to pipeline register)
- implemented by queues
- whenever a variable is read by a stage
Pipelining - Write Access

- forward write accesses to first intermediate variable (in spatial dimension)
  - ≈ forwarding in hardware
  - ⇒ order values using merge-element
Pipelining - Merge Element

**Dependency Graph**

- **$\pi_1$**
- **$\pi_2$**
- **$\pi_3$**

**Merger for $x$**

Immediate actions in $\pi_1$:
- $x(I_9)$
- $x(I_6)$

Delayed actions in $\pi_1$:
- $x(I_8)$
- $x(I_7)$

Immediate actions in $\pi_3$:
- $x(I_4)$
- $x(I_3)$
- $x(I_2)$
- $x(I_1)$

Merger for variable $x$
Translation to C

- using OpenMP: API for programming MT
- create thread for each stage
- each stage is executed in an own loop ⇒ allows stages to run desynchronized
Outline

1. Introduction
2. Creating Multi-Threaded Code
3. Results
Benchmark

**MergeSort - 2x Xeon Quad Core**

![Speedup for Dual Xeon Quad Core](image)

Speedup for Dual Xeon Quad Core

Speedup 1Thread vs. Pipelined-MThread

`Multithreaded Code from Synchronous Programs (Daniel Baudisch, Jens Brandt, Klaus Schneider)`
Thank you for your attention!

Questions? Suggestions? Ideas?