Automatic parallelization of nested loop programs with data dependent behavior

Tjerk Bijlsma
Stefan Geuns
Joost Hausmans
Marco Bekooij

NXP
Enschede - The Netherlands

University of Twente
Outline

• Application domain
• Case-study radio application
• State-of-the-art
• Parallelization approach
• Buffers with overlapping windows
• Access pattern types
• Multiprocessor compiler
• Conclusions
Application domain

Real-time stream processing car-infotainment systems

Advanced radios contain multiple processors
Case-study radio application

```cpp
mode=0;
while(1){
    in=input();
    switch(mode){
        case 0: {
            detect(in, out mode@);}
        case 1: {
            decode(in, out mode@, out o1);
            process1(o1, out o2);
            process2(o2);}
    }
}
```
Case-study radio application

```
mode=0;
while(1){
in=input();
switch(mode){
    case 0: {
        detect(in, out mode@);}
    case 1: {
        decode(in, out mode@, out o1);
        process1(o1, out o2);
        process2(o2);}
}
```
State-of-the-art

- **Parallelization approaches**
  - Decoupled SoftWare Pipeling (Princeton)
    - Derives parallelism based on a control dataflow graph
  - Compaan (Leiden)
    - Derives maximum parallelism based upon exact data dependence analysis
  - Daniel Cordes (Dortmund)
    - Derives parallelism based upon data dependence analysis and control flow
    - *No support for data dependent behavior* (while-loops, if-statements)

- **Temporal analysis models**
  - Most end-2-end throughput analysis techniques have difficulties with input data dependent behavior
Parallelization approach

- Every function becomes a task
- A shared variable is replaced by a buffer with overlapping windows
  - Buffers support multiple readers
  - Buffers can have multiple mutual exclusive writers
    - That writes are mutual exclusive is explicit in the NLP but not in the task-graph
- A corresponding CSDF model can be derived
  - Guarantee throughput for real-time constraint
- Less restrictive form of single assignment required

```c
mode=0;
while(1){
in=input();
switch(mode){
  case 0: {
    detect(in, out mode);
  }
  case 1: {
    decode(in, out mode, out o1);
    process1(o1, out o2);
    process2(o2);
  }
}
```
Buffers with overlapping windows

- Read and write windows may overlap
  - Each written value can be read immediately, instead of at the moment that it falls outside the write window
  - Prevents deadlock in case of cyclic dependencies
- Multiple reading and writing tasks
- Array size as buffer capacity, sufficient for deadlock freedom

```
read window       write window
|   |   |   |   | x |   |   |
|   |   |   |   |   | x |  x |
|   |   |   |   |   |   |   |
|   |   |   |   |  x |   |   |
|   |   |   |   |   |   |  x |
|   |   |   |   |   |   |   |
|----------------|
| Full-bits      |
```
Buffers with overlapping windows

- Read and write windows may overlap
  - Each written value can be read immediately, instead of at the moment that it falls outside the write window
  - Prevents deadlock in case of cyclic dependencies

- Multiple reading and writing tasks

- Array size as buffer capacity, sufficient for deadlock freedom

```
read window       write window
\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c} \hline
| & | & | & | & | & | & | & | & \\
\hline
| O | O | O | O | O | O | O | O | O | O | O |
\hline
| X | X | X | X | X | X | X | X | X | X | X |
\hline
\end{array} \]
```
Access pattern types

- Access type indicates that there is FIFO access for variable, such that FIFO buffer can be used
  - Results in smaller buffers
  - Allows acquires and releases inside the switch statements
Access pattern types

- Access type indicates that there is FIFO access for variable, such that FIFO buffer can be used
  - Results in smaller buffers
  - Allows acquires and releases inside the switch statements
Multiprocessor compiler

NLP

C-functions

temporal constraints
architecture

use-cases
streams

real-time multiprocessor compiler

Platform specific

Hapi

sequential-C

Debug

ublaze
Arm
Pthread
Multiprocessor compiler flow

- C-functions
- use-cases
- architecture
- temporal constraints
- streams
- NLP

CSDF

Omphale parallelization

task graph

UET analysis

task graph + UET

Hebe dataflow analysis

task graph + UET + Cap + Settings

Helios resource allocation

task graph + table

(g Helix) distributed real-time OS

com-lib

gcc

ublaze

Arm

Pthread

Hapi

sequential-C
Conclusion

- Automatic parallelization
  - We can automatically extract a task graph from a data dependent NLP
    - Is (should be 😊 ) correct by construction
    - May contain an infinite loop for endless stream processing
    - May contain if-statements
  - Buffers with overlapping windows can be used for multiple reading and writing tasks
  - The access pattern type has been introduced

- The automatic parallelization approach is implemented in a multiprocessor compiler
Questions