Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor

P. Aubry\textsuperscript{1}, P.-E. Beaucamps\textsuperscript{2}, F. Blanc\textsuperscript{2}, B. Bodin\textsuperscript{2}, S. Carpov\textsuperscript{1}, L. Cudennec\textsuperscript{1}, V. David\textsuperscript{1}, P. Dore\textsuperscript{1}, P. Dubrulle\textsuperscript{1}, B. Dupont de Dinechin\textsuperscript{2}, F. Galea\textsuperscript{1}, T. Goubier\textsuperscript{1}, M. Harrand\textsuperscript{2}, S. Jones\textsuperscript{2}, J.-D. Lesage\textsuperscript{1}, S. Louise\textsuperscript{1}, N. Morey-Chaisemartin\textsuperscript{2}, T.-H. Nguyen\textsuperscript{1}, X. Raynaud\textsuperscript{2} and R. Sirdey\textsuperscript{1}

\textsuperscript{1} CEA LIST
\textsuperscript{2} KALRAY
Agenda

- History of the project.
- Overview of the MPPA architecture.
- Overview of the $\Sigma C$ programming model & language.
- Overview of the compilation process.
- Experimental results on a video encoding application.
- Conclusion
History of the project

- **2007-2008: CEA LIST internal R&D:**
  - First sketches of the ΣC dataflow programming technology in an attempt to answer the question « how to efficiently program a clusterized 256 cores chip ? ».

- **2008: creation of KALRAY.**

- **2009: CEA – KALRAY joint lab creation.**
  - 2010 : first end-to-end compiler and micro-kernel demonstrators in mono-cluster simulation environment.
  - 2012 : industrial transfer and ΣC technology integration in Kalray’s Software Development Kit (SDK).
  - 2013 : MPPA-256 and ΣC-powered AccessCore SDK commercial availability.
Main characteristics:

- 256 PE organized in 16 computing clusters with
  - 16+1 PE per cluster.
  - Some shared memory.
  - A NoC interface.

- Each PE is a custom energy efficient VLIW processor.

- Each cluster includes an additional PE for resource management.
  - Scheduling, DMA interrupt handling, ...

- Four specific 4 PE clusters are also available for handling the chip IOs.

- No L1 cache coherence within a cluster.
Overview of the ΣC programming model

- A dataflow programming model...
  - Network of processes communicating via and only via FIFO channels.
  - Data-driven synchronization.

- ... Leading to The ΣC language.
  - Suitable for a wide spectrum of embedded applications.
  - Hierarchical component-based approach for structuring large-scale applications.
  - Explicit parallelism.
  - Close-to-machine programming, if necessary.
  - C language extension.
Agents & subgraphs.
- Interface.
  - In, out & inout ports.
  - Cyclostatic specs.
- Map sections.
- Exchange functions.
System agents.
- Split & join.
- Dup.
- Select & merge.
IO agents.
- E.g. for DMA engines.
Shared constants.
Preload & delays.
Strict process semantics.
Strict C pointer equivalence.

```
agent LineFilter(int width) {
    interface {
        in<int> line;
        out<int> output1, output2;
        spec {{line[width]; output1[width]; output2[width]});
    }
    shared const int g1[11] = {-1, -6, -17, -17, 18, 46, 18, -17, -17, -6, -1},
        g2[11] = {0, 1, 5, 17, 36, 46, 36, 17, 5, 1, 0};
    void start() exchange (line a[width], output1 b[width], output2 c[width]) {
        int i,j;
        for(i=0;i<width;i++) {
            b[i] = 0; c[i] = 0;
            if(i<width-11)
            for(j=0;j<11;j++) {
                b[i] += g1[j] * a[i+j];
                c[i] += g2[j] * a[i+j];
            }
        }
    }
}
```

```
subgraph LinePass(int width, int height) {
    interface {
        in<int> imag;
        out<int> output1,output2;
        spec {{imag[width*height];
            output1[width*height];
            output2[width*height]});
    }
    map {
        int i;
        agent srr=new split<int>(height,width);
        agent jrr1=new join<int>(height,width);
        agent jrr2=new join<int>(height,width);
        for(i=0;i<height;i++) {
            agent filter=new LineFilter(width);
            connect(srr.output[i],filter.line);
            connect(filter.output1,jrr1.input[i]);
            connect(filter.output2,jrr2.input[i]);
        }
        connect(imag,srr.input);
        connect(jrr1.output,output1);
        connect(jrr2.output,output2);
    }
}
```
Compilation process overview

- A fully automated compilation chain...
  - The « usual » lexing, parsing, etc.
  - C code generation.
  - Parallelism instantiation & reduction.
  - System agents inlining.
  - Deadlock-free buffer sizing.
  - Task mapping & routing.
  - μ-kernel parameterization & loadbuild.
  - Feedback-directed iterative compilation.

- ... Paired with a low overhead execution model (μ-kernel).
  - Logical vector time- or dependency counter-based.
  - Correction by construction.
  - Functional determinism.
Pattern-based system agents « inlining »:

- Split(s) with pointer equivalence => 1 multi-reader buffer.
- Join(s) with pointer equivalence => 1 multi-writer/mono-reader buffer.
  - With orthogonal writers, hence no need for synchronization.
- Dup => 1 multi-reader buffer.
- Join(s) (with pointer equivalence) + dup => 1 multi-writer/multi-reader buffer.
  - Also with orthogonal writers, hence no need for inter-writer synchronization.

Buffer sizing (aka. getting rid of the FIFO channels):

- Step 1: solve the balance equations from cyclostatic dataflow theory.
- Step 2: perform a symbolic execution of the network with a task selection heuristic leading to small buffer occupancy.
  - This leads to small safe buffer sizes which can be expanded to meet the application throughput constraints thanks to the monotony properties inherited from the underlying KPN model.

These optimizations are critical.

- In order to transform the intrinsically inefficient (but formally convenient) FIFO channels-based programming model into an efficient and low footprint implementation on the embedded target.
Focus: tasks mapping & comm. routing

- **Step 1**: process network partitioning.
  - Node capacited graph partitioning under multidimensional knapsack constraints (NP-hard, relatively large size instances).

- **Step 2**: partitions-to-cluster mapping.
  - Quadratic assignment problem (NP-hard although amenable to exact resolution for an MPPA-256).

- **Step 3**: inter-cluster communication routing.
  - Mono-routed multi-flow (NP-hard although still amenable to exact resolution for an MPPA-256 using off-the-shelf ILP solvers).

- **Solution strategies:**
  - Beginning of the dev. cycle: problems solved in sequence, fast heuristic for step 1.
  - End of the dev. cycle: parallel global resolution.
Some experimental results

- The overall approach has now been stress tested.
  - From a software engineering viewpoint in terms of expressing several real-world applications (video coding, channel coding, sonar, etc.).
  - From an embedded performance viewpoint on the real MPPA chip.

- For example, a $\Sigma C$ implementation of the x264 on the MPPA processor lead to the following results (720p):
  - Better overall quality (SSIM and PSNR criteria) due the additional e.g., motion vectors and intra predictors tested (in parallel) without throughput impact.
  - Intra I-frame: 110 fps.
  - Inter P-frame: 40 fps.
  - Inter B-frame: 55 fps.

Parallel arch. of the x264 port.
Conclusion and perspectives

A 5 year tightly-coupled R&D project between a fabless semiconductor startup and a research institute.

Leading to a complete end-to-end solution for dataflow programming one of the world’s first 256 core embedded architecture.

- From the programming language itself down to a dedicated distributed μ-kernel.
- Well-suited for signal or image processing intensive applications.

Still, many R&D perspectives are lying ahead:

- More advanced robust resources allocation (notably taking into account execution time uncertainty) under multiple criteria.
  - As well as « true » iterative compilation.
- Dataflow process network mapping & routing on multi-chip systems.
- Compile-time and online support for fault tolerance.
- Support for mixed hard real-time/computation intensive soft real-time apps.
Related papers

Merci !