FORMLESS: Scalable Utilization of Embedded Manycores in Streaming Applications

Matin Hashemi¹, Mohammad H. Foroozannejad², Christoph Etzel³, Soheil Ghiasi²

Sharif University of Technology
University of California, Davis
University of Augsburg
Streaming Applications

- Widespread
  - Cell phones, mp3 players, video conference, real-time encryption, graphics, HDTV editing, hyperspectral imaging, cellular base stations

- Definition
  - Infinite sequence of data items
  - At any given time, operates on a small window of this sequence
  - Moves forward in data space

```cpp
//53° around the z axis
const R[3][3] = {
  {0.6, -0.8, 0.0},
  {0.8, 0.6, 0.0},
  {0.0, 0.0, 1.0}
};

Rotation3D {
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      B[i] += R[i][j] * A[j]
}
```
Application Model:
Dataflow Task Graph

- Vertices or actors
  - functions, computations

- Edges
  - data dependency, communication between actors

- Execution Model
  - any actor can perform its computation whenever all necessary input data are available on incoming edges.

- SDF is one special case
  - statically schedulable [Lee ‘87]
Example

Vocoder Task Graph
http://www.cag.csail.mit.edu/streamit
Manycore Model

- Distributed memory
- Interconnect network for sending/receiving messages
- Examples: Tilera TILE64, UC Davis ASAP
Software Synthesis

- Compile the high-level dataflow specification into the parallel software modules.
- Task Assignment
  - Assign tasks to processors
- Task Scheduling
  - Schedule the tasks assigned to the same processor for periodic sequential execution on that processor.
Baseline Software Synthesis

```c
#include msort.h; // msort.h
void sort(int* x, int n){...}
void merge(int* x, int* y, int* z, int n){...}

#include msort.h; // core1.c
int x[100];
while() for i=1:100 x[i]=read(in);
for i=1:25 x1[i]=x[i];
for i=1:25 write(x1[i+25],2);
for i=1:25 write(x1[i+50],3);
for i=1:25 write(x1[i+75],3);
sort(x1,25);
for i=1:25 write(x1[i],2);

#include msort.h; // core2.c
int x1[25],x2[25];
int y1[50];
while() for i=1:25 x2[i]=read(1);
sort(x2,25);
for i=1:25 x1[i]=read(1);
merge(x1,x2,y1,25);
for i=1:50 write(y1[i],4);

#include msort.h; // core3.c
int x3[25],x4[25];
while() for i=1:25 x3[i]=read(1);
for i=1:25 x4[i]=read(1);
sort(x3,25);
for i=1:25 write(x3[i],4);
for i=1:25 write(x4[i],4);

#include msort.h; // core4.c
int x3[25],x4[25],y1[50];
int y2[50],y3[100];
while() for i=1:25 x3[i]=read(3);
for i=1:25 x4[i]=read(3);
merge(x3,x4,y2,25);
for i=1:50 write(y2[i],2);
merge(y1,y2,y3,50);
for i=1:100 write(y3,out);
```
Observation

- In principle “behavior” and “implementation” are separated
- Nevertheless, “some” inflexible structure is dictated by the designer
  - Implementations on a few vs. many processors
Motivating Example

- Automatic task assignment to 7 processors:

- Execution period = \(10^7\) (x1000 clk @ 100MHz)

- Experiment platform is FPGA prototyped multiprocessor
  - NiosII/f @ 200MHz
  - 32KB data cache
  - 8KB inst. cache
  - Inter-processor connections are FIFO buffers of depth 1024
  - Offchip DDR2-800 main memory
Motivating Example (Cont’d)

- Programmer **manually** constructs the task graph for 7 processors.

- Automatic task assignment to 7 processors:

- Execution period = **110**
Motivating Example (Cont’d)

- Automatically generated task graph for 7 processors.
- Automatic task assignment to 7 processors
- Execution period = 94
Observation

- In principle “behavior” and “implementation” are separated.
- Nevertheless, “some” inflexible structure is dictated by the designer.
  - Implementations on a few vs. many processor.
- Solution: Raise the abstraction level in specification.
  - Functionally consistent.
  - Admitting transformations, i.e., structurally malleable.
Higher-Level Specification

- Functionally-consistent Structurally-Malleable Streaming Specification (FORMLESS)
- Tasks functionality, ports, rates and their composition are governed by forming parameters.
FORMLESS Task Graph
Case Study 1: Merge Sort

- Parameters:
  - $\phi_1$: number of parallel sort tasks
  - $\phi_2$: fan-in degree of merge and fan-out degree of scatter tasks.

scatter2 ($r[]$, $x[]$, $y[]$, $n$)
scatter3 ($r[]$, $x[]$, $y[]$, $z[]$, $n$)
sort ($x[]$, $s[]$, $n$)
merge2 ($x[]$, $y[]$, $r[]$, $n$)
merge3 ($x[]$, $y[]$, $z[]$, $r[]$, $n$)

$\Phi=(3,3)$
$\Phi=(8,2)$
$\Phi=(1,1)$
Case Study 2: Matrix Multiply

Matrix Multiply:
$A_{mxn} \times B_{nxp} = C_{mp}$

Parallelized Matrix Multiply:
e.g., $A_2 \times B_1 = C_{21}$

Parameters:
- $\phi_1$ and $\phi_2$: number of divisions in rows and columns of $A$ and $B$. 

$\Phi = (3, 2)$
Case Study 3: Fast Fourier Transform

- Parameters:
  - $\varphi_1$: radix of the butterfly tasks.
  - $\varphi_2$: number of butterfly tasks grouped together.
Case Study 4: Advanced Encryption Standard (AES)

- Basic operations:
  - substitute byte (sub): data parallel across elements
  - shift row (shf): data parallel across rows
  - add round key (ark): data parallel across elements
  - mix column (mix): data parallel across columns
- $\phi_1 \ldots \phi_4$ represent the number of parallel sub, shf, ark and mix tasks.
Case Study 5:
Low Density Parity Check (LDPC)

H Matrix:

\[
\begin{array}{cccccccccccc}
V_1 & V_2 & V_3 & V_4 & V_5 & V_6 & V_7 & V_8 & V_9 & V_{10} & V_{11} & V_{12} \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

Tanner graph is constructed by adding an edge for every cell of value 1 in matrix.

Task graph is constructed by condensing all C nodes into one and all V nodes into one, and then, unroll the graph to remove the bi-directional edges.

Row-split LDPC with split factor of \( \varphi=2 \) based on work by Mohsenin et al.

Task graph is constructed from the Tanner graph as before:
### Benchmark Summary

| Alphabet | Vector $\Phi$ | $\Delta = \text{Domain of } \Phi$ | $|\Delta| |$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>$(\phi_1, \phi_2, \phi_3, \phi_4)$</td>
<td>$\delta_1 = \delta_2 = \delta_3 = \delta_4 = {1, 2, 4}$</td>
<td>81</td>
</tr>
<tr>
<td>FFT</td>
<td>$(\phi_1, \phi_2)$</td>
<td>$\delta_1 = {2, 4, 8, 16}, \delta_2 = {1, 2, 4, \ldots, 128}$</td>
<td>32</td>
</tr>
<tr>
<td>SORT</td>
<td>$(\phi_1, \phi_2)$</td>
<td>$\delta_1 = {1, \ldots, 100}, \delta_2 = {1, \ldots, 10}, \phi_1 = \phi_2^n$</td>
<td>26</td>
</tr>
<tr>
<td>MMUL</td>
<td>$(\phi_1, \phi_2)$</td>
<td>$\delta_1 = \delta_2 = {1, \ldots, 16}$</td>
<td>256</td>
</tr>
<tr>
<td>LDPC</td>
<td>$(\phi_1)$</td>
<td>$\delta_1 = {1, 2, 4, 8, 16}$</td>
<td>5</td>
</tr>
</tbody>
</table>
FORMLESS Design Flow

- Programmer specifies the application in the proposed malleable format

- Tool explores the design space to hammer out a specific task graph from the malleable specification based on the target architecture.
Experiment Setup

- Emulated multicore systems with Nios soft processors on FPGA.
  - Cores: NiosII/f 200MHz, Floating Point Unit,
  - Memory: 8KB instruction and 32KB data cache, DDR2 main memory.
  - Network: Adjacent cores are connected with point-to-point FIFO, depth of each FIFO is 1024.

- Area constraint
  - Only 8 cores fit on the FPGA. For more cores we use Sequential Execution Abstraction Model (SEAM)
  - Simulate the entire system with Modelsim
  - Simplify the sequential sections (no inter-core communications) as wait functions in order to speed up the simulation.
  - Our previous experiments show the error is small [Huang ‘08].
Sequential Execution Abstraction Model (SEAM)

```
while(1) {
    for i=1..n
        S[i] = readf()
    for i=1..n
        X[i] = S[i] + S[n-i]
    for i=1..n
        Y[i] = readf()
    for i=1..n
        Z[i] = X[i] * Y[n-i]
    for i=1..n
        writef(Z[i])
}
```

code3.c

```
while(1) begin
    read ( N1 )
    #W1
    read ( N2 )
    #W2
    write( N3 )
end
```

code3.v

system-level behavior extraction
Experiment Results

- Application throughput on manycore platforms normalized with respect to single-core throughput.
- DSE instantiated task graphs have higher throughput than fixed task graphs.
Coverage and throughput degradation vs the number of forming vectors considered by the DSE.

On average, 15% of forming vectors are enough to form the task graphs with at most 10% throughput degradation.