Toward Extreme-Scale High-Performance Computing Using a Fine-Grain Dataflow-Inspired Execution Model

Stéphane Zuckerman

Computer Architecture & Parallel Systems Laboratory
Electrical & Computer Engineering Dept.
University of Delaware
140 Evans Hall Newark, DE 19716, United States

September 10, 2014

A Short Introduction to Execution Models
- The Von Neumann Model
- The Dataflow Model

The Codelet Model: Harnessing Parallelism in Shared-Memory Multi/Many Core Systems

DARTS: An Implementation of the Codelet Model
- DARTS: Implementation of the Codelet Machine Model
- DARTS: Experimental Results
  - Running DGEMM in DARTS
  - Running Graph500 in DARTS

Running DARTS on a Dataflow-Enabled Multi-Core Architecture
- The TERAFLUX Project
- Porting DARTS to COTSOn
- DARTS/COTSOn: Experimental Results

The Future of Codelets
2. A Short Introduction to Execution Models
   - The Von Neumann Model
   - The Dataflow Model
3. The Codelet Model: Harnessing Parallelism in Shared-Memory Multi/Many Core Systems
4. DARTS: An Implementation of the Codelet Model
   - DARTS: Implementation of the Codelet Machine Model
   - DARTS: Experimental Results
     - Running DGEMM in DARTS
     - Running Graph500 in DARTS
5. Running DARTS on a Dataflow-Enabled Multi-Core Architecture
   - The TERAFLUX Project
   - Porting DARTS to COTSon
   - DARTS/COTSon: Experimental Results
6. The Future of Codelets
2004–2005: Apparition of Multi-Core Systems

- The power wall leads to the first multi-core processors
- Memory wall: a major performance issue (See Wulf and McKee 1995)
- GPUs become more programmable (but still through dirty hacks)

2006–2007: Real Multi-Core Processors Appear

- Intel proposes “real” multi-core processors (but still use a front-side bus)
- AMD provides an efficient interconnect for NUMA architectures
- IBM unveils the POWER6, Cell B.E. and Cyclops-64
- Nvidia uncovers CUDA (No need to resort to dirty hacks anymore)

2008–2010: Toward “Many-Core” Compute Nodes

- Compute nodes start to propose a large number of cores
  - e.g., 8-Core Intel Nehalem EX: 4 × 16 threads per node, with a NUMA Interconnect
- Nvidia commercializes boards dedicated to supercomputing
2004–2005: Apparition of Multi-Core Systems

- The power wall leads to the first multi-core processors
- Memory wall: a major performance issue (See Wulf and McKee 1995)
- GPUs become more programmable (but still through dirty hacks)

2006–2007: Real Multi-Core Processors Appear

- Intel proposes “real” multi-core processors (but still use a front-side bus)
- AMD provides an efficient interconnect for NUMA architectures
- IBM unveils the POWER6, Cell B.E. and Cyclops-64
- Nvidia uncovers CUDA (No need to resort to dirty hacks anymore)
2004–2005: Apparition of Multi-Core Systems

- The power wall leads to the first multi-core processors
- Memory wall: a major performance issue (See Wulf and McKee 1995)
- GPUs become more programmable (but still through dirty hacks)

2006–2007: Real Multi-Core Processors Appear

- Intel proposes “real” multi-core processors (but still use a front-side bus)
- AMD provides an efficient interconnect for NUMA architectures
- IBM unveils the POWER6, Cell B.E. and Cyclops-64
- Nvidia uncovers CUDA (No need to resort to dirty hacks anymore)

2008–2010: Toward “Many-Core” Compute Nodes

- Compute nodes start to propose a large number of cores
  - e.g., 8-Core Intel Nehalem EX: $4 \times 16$ threads per node, with a NUMA Interconnect
- Nvidia commercializes boards dedicated to supercomputing
Parallel Programming in 2005–2010

Meanwhile, in Versailles...

- 2006: Compiler transformation – Deep Jam
- 2007–2008: Methodology to fine-tune kernels on multicore systems
- 2009–2010: A balanced approach to application performance tuning
- 2010: Tackling cache line stealing in multicore systems

## Parallel Programming in 2005–2010

### Meanwhile, in Versailles...

- **2006:** Compiler transformation – Deep Jam
- **2007–2008:** Methodology to fine-tune kernels on multicore systems
- **2009–2010:** A balanced approach to application performance tuning
- **2010:** Tackling cache line stealing in multicore systems


### Main Parallel Programming Models

- **MPI**
- **OpenMP**
- **CUDA**

...for adventurers only

### What to Expect for the Next Generation HPC Systems?

- Core/thread count per processor is rising
- Amount of cache per core/thread is decreasing
- Memory is becoming a *severe* bottleneck
  - Many people think coherence will have to go
Parallel Programming in 2005–2010

Meanwhile, in Versailles...

- 2006: Compiler transformation – Deep Jam
- 2007–2008: Methodology to fine-tune kernels on multicore systems
- 2009–2010: A balanced approach to application performance tuning
- 2010: Tackling cache line stealing in multicore systems


Main Parallel Programming Models

- MPI
- OpenMP
- CUDA
  ...for adventurers only

What to Expect for the Next Generation HPC Systems?

- Core/thread count per processor is rising
- Amount of cache per core/thread is decreasing
- Memory is becoming a severe bottleneck
  - Many people think coherence will have to go

How will we program the next parallel processors?
Outline

2. A Short Introduction to Execution Models
   - The Von Neumann Model
   - The Dataflow Model
3. The Codelet Model: Harnessing Parallelism in Shared-Memory Multi/Many Core Systems
4. DARTS: An Implementation of the Codelet Model
   - DARTS: Implementation of the Codelet Machine Model
   - DARTS: Experimental Results
     - Running DGEMM in DARTS
     - Running Graph500 in DARTS
5. Running DARTS on a Dataflow-Enabled Multi-Core Architecture
   - The TERAFLUX Project
   - Porting DARTS to COTSon
   - DARTS/COTSon: Experimental Results
6. The Future of Codelets
A Short Introduction to Execution Models
The von Neumann Model – a High-Level View

I/Os
- Inputs (kbd, mouse, HDD, ...)
- Outputs (displays, LED, HDD, ...)

Memory
- MAR
- MDR

CPU
- Central Processing Unit
  - Control Unit
  - Processing Unit

Control Bus
Address Bus
Data Bus

S.Zuckerman
Driving HPC Computing with Codelets
A Short Introduction to Execution Models
The von Neumann Model – Advantages and Limits

Advantages of the von Neumann Model

▶ Simple
▶ Can almost be implemented “directly”
  ▶ However nobody would design a processor this way nowadays

Limitations of the von Neumann Model

▶ Relies on a sequence of instructions
▶ Time is thus an integral part of the model
▶ Makes use of an accumulator: side-effects are inherent to the model
  ▶ Reduces the potential for parallelism

Working Around Those Limitations

▶ Duplicate several “von Neumann machines,” each with their own PC
▶ Add buses to share both memory and I/Os between processors
▶ ... Is it still a von Neumann machine then?
A Short Introduction to Execution Models
The (Static) Dataflow Model

Static Dataflow Actors

Components of a regular actor:
▶ Input arcs which may contain at most 1 token each
▶ Output arcs which may contain at most 1 token each
▶ The operation provided by the actor
▶ Tokens

Firing Rule: Static Dataflow
An actor may fire when:
▶ All of its input arcs contain a token, and
▶ Its output arcs are empty.
A Short Introduction to Execution Models
The (Static) Dataflow Model

Static Dataflow Actors
Components of a regular actor:
- Input arcs which may contain at most 1 token each
- Output arcs which may contain at most 1 token each
- The operation provided by the actor
- Tokens

See Dennis, Fosseen, and Linderman 1972; Dennis 1974; Dennis and Misunas 1974
**A Short Introduction to Execution Models**

**The (Static) Dataflow Model**

### Static Dataflow Actors

Components of a regular actor:

- Input arcs which may contain at most 1 token each
- Output arcs which may contain at most 1 token each
- The operation provided by the actor
- Tokens

See Dennis, Fosseen, and Linderman 1972; Dennis 1974; Dennis and Misunas 1974

### Firing Rule: Static Dataflow

An actor may *fire* when:

- All of its input arcs contain a token, and
- Its output arcs are empty.
A Short Introduction to Execution Models
The (Static) Dataflow Model

Static Dataflow Actors
Components of a regular actor:

- Input arcs which may contain at most 1 token each
- Output arcs which may contain at most 1 token each
- The operation provided by the actor
- Tokens

See Dennis, Fosseen, and Linderman 1972; Dennis 1974; Dennis and Misunas 1974

Firing Rule: Static Dataflow
An actor may *fire* when:

- All of its input arcs contain a token, and
- Its output arcs are empty.
A Short Introduction to Execution Models

An Example of Dataflow Program

3 \rightarrow \times \rightarrow 10 \rightarrow +
A Short Introduction to Execution Models

An Example of Dataflow Program
A Short Introduction to Execution Models
An Example of Dataflow Program
A Short Introduction to Execution Models
The Architecture Model of Static Dataflow

Figure: Inspired by J. Dennis’ article (Encyclopedia of Parallel Computing)
A Short Introduction to Execution Models
Applying Our Example on the Static-DF Arch.
Outline

2. A Short Introduction to Execution Models
   - The Von Neumann Model
   - The Dataflow Model
3. The Codelet Model: Harnessing Parallelism in Shared-Memory Multi/Many Core Systems
4. DARTS: An Implementation of the Codelet Model
   - DARTS: Implementation of the Codelet Machine Model
   - DARTS: Experimental Results
     - Running DGEMM in DARTS
     - Running Graph500 in DARTS
5. Running DARTS on a Dataflow-Enabled Multi-Core Architecture
   - The TERAFLUX Project
   - Porting DARTS to COTSon
   - DARTS/COTSon: Experimental Results
6. The Future of Codelets
The Codelet Model: Harnessing Parallelism in Shared-Memory Multi/Many Core Systems

Objectives

▶ Fine-grain parallelism
▶ Scalable
▶ Expose maximal parallelism
▶ Limits non-determinism (determinate-by-default)
▶ Handles dynamic events (power, resiliency, resource constraints in general)

Definition

A codelet is a sequence of machine instructions which act as an atomically-scheduled unit of computation.

The Codelet Model: Harnessing Parallelism in Shared-Memory Multi/Many Core Systems

Properties

- Event-driven (availability of data and resources)
- Communicates only through its inputs and outputs
- Non-preemptive (with very specific exceptions)
- Requires all data and code to be “local”

See Zuckerman, Suetterlein, et al. 2011
The Codelet Abstract Machine

See Zuckerman, Suetterlein, et al. 2011
Codelet Firing Rule

- Codelet actors are *enabled* once tokens are on each input arc.
- Codelet actors fire by
  - consuming tokens
  - performing the operations within the codelet
  - producing a token on each of its output arcs

States of a Codelet

- Dormant: Not all tokens are available
- Enabled: All *data* tokens are available
- Ready: All tokens are available
- Active: The codelet is executing internal operations

See Zuckerman, Suetterlein, et al. 2011
Threaded Procedures (TPs)

TPs are containers for codelet graphs, with additional meta-data.

**Description**

- Invoked in a control-flow manner
- Called by a codelet from another CDG
- Feature a frame which contains the context of the CDG

See Zuckerman, Sueterlein, et al. 2011
An Example of Computation Using Threaded Procedures

See Zuckerman, Suettlerlein, et al. 2011
2. A Short Introduction to Execution Models
   - The Von Neumann Model
   - The Dataflow Model
3. The Codelet Model: Harnessing Parallelism in Shared-Memory Multi/Many Core Systems
4. DARTS: An Implementation of the Codelet Model
   - DARTS: Implementation of the Codelet Machine Model
   - DARTS: Experimental Results
     - Running DGEMM in DARTS
     - Running Graph500 in DARTS
5. Running DARTS on a Dataflow-Enabled Multi-Core Architecture
   - The TERAFLUX Project
   - Porting DARTS to COTSOn
   - DARTS/COTSOn: Experimental Results
6. The Future of Codelets
Objectives

- Faithfulness to the codelet execution model
- Modularity
  - So that portions of the runtime can be added or changed easily
  - For example: we have several codelet schedulers from which to choose
- Portability: Object-oriented, written in C++98, and makes use of open-source libraries:
  - `hwloc`: to determine the topology of the underlying system (HW threads/cores, caches, etc.)
  - If present on the system, it uses Intel TBB’s lock-free queues

See Suetterlein, Zuckerman, and Gao 2013
DARTS: Implementation of the Codelet Machine Model

- Computation Units (CUs) embed a single producer/consumer ring buffer to store ready codelets.
- Synchronization Units (SUs) embed two pools: Threaded Procedures and ready codelets.
- Heavy reliance on lock-free data structures.
- SUs can temporarily assume the role of CUs if all other CUs are busy and there are ready codelets left to execute.

See Suetterlein, Zuckerman, and Gao 2013
### Experimental Setup

#### AMD Opteron 6234 (Bulldozer) – Mills – 128 GiB DDR DRAM

<table>
<thead>
<tr>
<th>Cache Level</th>
<th>Shared By</th>
<th>Size (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock (GHz)</td>
<td>2.4</td>
<td>1 core</td>
</tr>
<tr>
<td>Threads / core</td>
<td>1</td>
<td>1 core</td>
</tr>
<tr>
<td>Cores / socket</td>
<td>12</td>
<td>2 cores</td>
</tr>
<tr>
<td>Sockets / node</td>
<td>4</td>
<td>6 cores</td>
</tr>
<tr>
<td>L1 Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Unified</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Unified</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Compiler**
- gcc v4.6

**Math Library**
- AMD Core Math Library (ACML) v5.3

Note: FPUs are shared between 2 cores.

#### Intel Xeon E5-2670 (Sandy Bridge) – FatNode – 64 GiB DDR3 DRAM

<table>
<thead>
<tr>
<th>Cache Level</th>
<th>Shared By</th>
<th>Size (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock (GHz)</td>
<td>2.6</td>
<td>2 threads</td>
</tr>
<tr>
<td>Threads / core</td>
<td>2</td>
<td>2 threads</td>
</tr>
<tr>
<td>Cores / socket</td>
<td>8</td>
<td>2 threads</td>
</tr>
<tr>
<td>Sockets / node</td>
<td>2</td>
<td>8 threads</td>
</tr>
<tr>
<td>L1 Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Unified</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Unified</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Compiler**
- gcc v4.7

**Math Library**
- Intel Math Kernel Library (MKL) v11.1

Note: Functional units are shared between 2 threads.
Running DGEMM in DARTS – Codelet Graph

Description of DGEMM

- **Double precision GEneral Matrix Multiplication**
- Used ACML or MKL as sequential building blocks (no tiling/blocking, etc., needed)
- We compared several codelet scheduling policies within a cluster of cores

Figure: Our Codelet Graph decomposition for a parallel DGEMM

See Sutterlein, Zuckerman, and Gao 2013
Figure: 10000 × 10000 Square DGEMM – Strong Scaling.

See Suetterlein, Zuckerman, and Gao 2013
Running DGEMM in DARTS – Mills – Weak Scaling

Figure: 48 cores – Square DGEMM – Weak Scaling.

See Suetterlein, Zuckerman, and Gao 2013
Running DGEMM in DARTS – FatNode – Strong Scaling

Figure: 3072 × 3072 Square DGEMM – Strong Scaling.
Figure: 32 threads – Square DGEMM – Weak Scaling.
Description of Graph500

- Reused reference code (http://graph500.org)
- Only modified the breadth-first search phase (BFS)
- Compared with reference OpenMP parallelization
- Unit: Traversed Edges Per Second (TEPS)

See Suetterlein, Zuckerman, and Gao 2013
Running Graph500 in DARTS – Mills – Strong Scaling

Figure: \( \text{Scale} = 2^{18} \) – Graph500 – Strong Scaling

See Suetterlein, Zuckerman, and Gao 2013
Running Graph500 in DARTS – Mills – Weak Scaling

Figure: 48 cores – Graph500 – Weak Scaling

See Souterlein, Zuckerman, and Gao 2013
Running Graph500 in DARTS – FatNode – Strong Scaling

Figure: Scale = $2^{18}$ – Graph500 – Strong Scaling

See Suetterlein, Zuckerman, and Gao 2013
Running Graph500 in DARTS – FatNode – Weak Scaling

Figure: 32 cores – Graph500 – Weak Scaling

See Suetterlein, Zuckerman, and Gao 2013
Project Objectives

*Future Teradevice systems will expose a large amount of parallelism (1000+ cores) that cannot be exploited efficiently by current applications and programming models. The aim of this project is to propose a complete solution that is able to harness the large-scale parallelism in an efficient way. The main objectives of the project are the programming model, compiler analysis, and a scalable, reliable, architecture based mostly on commodity components. Data-flow principles are exploited at all levels as to overcome the current limitations.*

For more details, see [http://teraflux.eu](http://teraflux.eu)

A DataFlow Thread (DF-Thread) is a non-preemptive piece of code which is ready to be fired when all its data dependencies are met.

A DF-Frame contains all the data required by the DF-Thread to run.
- While there are dependencies left, a DF-Frame is write-only
- Once all dependencies are met, the frame becomes read-only

The TERAFLUX abstract machine model features:
- A Thread Scheduling Unit (equivalent of the Codelet Model’s SU)
- A Fault-Detection Unit (to handle fault-tolerance)

The DF-Thread Model

- A *DataFlow Thread* (DF-Thread) is a non-preemptive piece of code which is ready to be *fired* when all its data dependencies are met.
  - I’ve heard that line somewhere...

- A DF-Frame contains all the data required by the DF-Thread to run.
  - While there are dependencies left, a DF-Frame is write-only
  - Once all dependencies are met, the frame becomes read-only

- The TERAFLUX abstract machine model features:
  - A Thread Scheduling Unit (equivalent of the Codelet Model’s SU)
  - A Fault-Detection Unit (to handle fault-tolerance)

Porting DARTS to COTSOn
Mapping Codelets to DF-Threads

Two key differences:

- 1 level of parallelism (DF-Threads) vs. 2 (Codelets + TPs)
- Each DF-Thread has its own private frame
- All codelets belonging to a TP share the same TP frame (and data)

DARTS maps each codelet to a DF-Thread, with a minimal DF-frame

The TP frame shared by codelets siblings is allocated on the heap

All codelets belonging to a TP are constrained to the same node

Porting DARTS to COTSon
Implementation Details

- Adding a codelet to the graph during execution triggers the call to
  `df_tschedule(&Fire, nb_deps, sizeof(Codelet*))`

- The `Fire` function is tasked to call the `Codelet::fire()` function and then clean up after using `df_destroy()`

- Threaded procedures are called using the `invoke<ThdProc>(parameters)` function:
  - Parameters are marshalled along with the TP type, and bundled within a DF-Thread
  - When firing, the DF-Thread allocates the TP on the heap, along with all of its codelets

DARTS/COTSOn: Experimental Results

Experimental Setup

All latencies were obtained using CACTI. We used COTSOn’s dynamic samplers to measure time (sample = 5M instructions).

<table>
<thead>
<tr>
<th></th>
<th>Private / Shared</th>
<th>Size</th>
<th>Number of Sets</th>
<th>Cache Line Size (Bytes)</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1D cache</td>
<td>private</td>
<td>16 KiB</td>
<td>4</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>L1I cache</td>
<td>private</td>
<td>32 KiB</td>
<td>4</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>L2U cache</td>
<td>private</td>
<td>64 KiB</td>
<td>4</td>
<td>64</td>
<td>5</td>
</tr>
<tr>
<td>L3U cache</td>
<td>shared</td>
<td>4 MiB</td>
<td>8</td>
<td>128</td>
<td>10</td>
</tr>
</tbody>
</table>

Running Fibonacci in DARTS – TERAFLUX

Strong Scaling

Figure: $\text{Cutoff} = 18 - \text{Fibonacci} - n = 36 - \text{Strong Scaling}$
Running Fibonacci in DARTS – TERAFLUX

Weak Scaling

Figure: Cutoff = 18 – Fibonacci(n) – Weak Scaling
Running Merge Sort in DARTS – TERAFLUX

Strong Scaling

Figure: Cutoff = 18 – Merge Sort – n = 5M elements – Strong Scaling
Running Merge Sort in DARTS – TERAFLUX

Weak Scaling

Figure: Cutoff = 10000 – Merge Sort(n) – Weak Scaling

2. A Short Introduction to Execution Models
   - The Von Neumann Model
   - The Dataflow Model
3. The Codelet Model: Harnessing Parallelism in Shared-Memory Multi/Many Core Systems
4. DARTS: An Implementation of the Codelet Model
   - DARTS: Implementation of the Codelet Machine Model
   - DARTS: Experimental Results
     - Running DGEMM in DARTS
     - Running Graph500 in DARTS
5. Running DARTS on a Dataflow-Enabled Multi-Core Architecture
   - The TERAFLUX Project
   - Porting DARTS to COTSOn
   - DARTS/COTSOn: Experimental Results
6. The Future of Codelets
The Future of Codelets
The Story So Far

- We proposed the codelet execution model to answer the need for scalability, performance, energy efficiency, fault-tolerance, and programmability.
- Experimental results show that the Codelet Model can be competitive with current multicore environments.
- With hardware support, the Codelet Model displays very high potential to scale to large numbers of cores.
The Future of Codelets
The Story So Far

- We proposed the codelet execution model to answer the need for scalability, performance, energy efficiency, fault-tolerance, and programmability.
- Experimental results show that the Codelet Model can be competitive with current multicore environments.
- With hardware support, the Codelet Model displays very high potential to scale to large numbers of cores.
The Future of Codelets
The Story So Far

- We proposed the codelet execution model to answer the need for scalability, performance, energy efficiency, fault-tolerance, and programmability
- Experimental results show that the Codelet Model can be competitive with current multicore environments
- With hardware support, the Codelet Model displays very high potential to scale to large numbers of cores

<table>
<thead>
<tr>
<th>Fine-Grain Multithreading</th>
<th>Dataflow-Inspired Codelet Execution Model</th>
<th>EXADAPT ’11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TERAFLUX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EuroMicro/DSD ’13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Micpro ’14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROME ’13</td>
<td>DFM ’14a</td>
</tr>
<tr>
<td></td>
<td>MTAAP ’13</td>
<td>Euro-Par ’13</td>
</tr>
<tr>
<td></td>
<td>Runtime Systems, Scheduling, Resource Management</td>
<td></td>
</tr>
</tbody>
</table>
The Future of Codelets
The Story So Far

- We proposed the codelet execution model to answer the need for scalability, performance, energy efficiency, fault-tolerance, and programmability.
- Experimental results show that the Codelet Model can be competitive with current multicore environments.
- With hardware support, the Codelet Model displays very high potential to scale to large numbers of cores.

---

<table>
<thead>
<tr>
<th>Fine-Grain Multithreading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataflow-Inspired Codelet Execution Model</td>
</tr>
</tbody>
</table>

**TERAFLUX**

<table>
<thead>
<tr>
<th>TERAFLUX</th>
<th>Streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>EuroMicro/DSD '13</td>
<td>DFM '14b</td>
</tr>
<tr>
<td>Micpro '14</td>
<td>ICCS '14</td>
</tr>
<tr>
<td>ROME '13</td>
<td>DFM '14a</td>
</tr>
<tr>
<td>MTAAP '13</td>
<td>Euro-Par '13</td>
</tr>
</tbody>
</table>

**Runtime Systems, Scheduling, Resource Management**
The Future of Codelets
The Story So Far

- We proposed the codelet execution model to answer the need for scalability, performance, energy efficiency, fault-tolerance, and programmability.
- Experimental results show that the Codelet Model can be competitive with current multicore environments.
- With hardware support, the Codelet Model displays very high potential to scale to large numbers of cores.

<table>
<thead>
<tr>
<th>Fine-Grain Multithreading</th>
<th>MTAAP ’14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataflow-Inspired Codelet Execution Model</td>
<td>EXADAPT ’11</td>
</tr>
<tr>
<td>EuroMicro/DSD ’13</td>
<td>DFM ’14b</td>
</tr>
<tr>
<td>Micpro ’14</td>
<td>ICCS ’14</td>
</tr>
<tr>
<td>ROME ’13</td>
<td>DFM ’14a</td>
</tr>
<tr>
<td>MTAAP ’13</td>
<td>Euro-Par ’13</td>
</tr>
</tbody>
</table>

Streaming

Runtime Systems, Scheduling, Resource Management
Extending the Codelet Model
Extending Codelets to Streams

Streaming Codelet Program + Hints

Compiler
- General Optimization
- Generate Optimization Hints for Runtime

Intra-tile Runtime System
- Fine-grained Parallelism
- Buffer Allocation
- Bandwidth Allocation
- Energy Efficiency

Heterogeneous Tile-Based Architecture
- Each tile contains various compute capabilities, a local NoC and local memory.

Extending the Codelet Model
Extending Codelets to Streams

Acknowledgements

Past and present CAPSL members,

<table>
<thead>
<tr>
<th>Brian</th>
<th>Lucas</th>
<th>Jaime</th>
<th>Arteaga</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joseph</td>
<td>Manzano</td>
<td>Chen</td>
<td>Chen</td>
</tr>
<tr>
<td>Daniel</td>
<td>Orozco</td>
<td>Elkin</td>
<td>Garcia</td>
</tr>
<tr>
<td>Robert</td>
<td>Pavel</td>
<td>Souad</td>
<td>Koliaï</td>
</tr>
<tr>
<td>Sergio</td>
<td>Pino</td>
<td>Aaron</td>
<td>Landwehr</td>
</tr>
<tr>
<td>Jürgen</td>
<td>Ributzka</td>
<td>Josh</td>
<td>Landwehr</td>
</tr>
<tr>
<td>Sunil</td>
<td>Shrestha</td>
<td>Kelly</td>
<td>Livingston</td>
</tr>
<tr>
<td>Pouya</td>
<td>Fotouhi</td>
<td>Joshua</td>
<td>Suetterlein</td>
</tr>
<tr>
<td>José</td>
<td>Monsalve</td>
<td>Haitao</td>
<td>Wei</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yao</td>
<td>Wu</td>
</tr>
</tbody>
</table>

...And of course, Professor Gao!
A Short Introduction to Execution Models

The Synchronous Dataflow Model (Lee and Messerschmitt 1987)

Components of a regular actor:
▶ Input arcs; each input arc $i_l$ can contain a certain number $k_l$ of tokens
▶ Output arcs; each output arc $o'_l$ can contain a certain number $k'_l$ of tokens
▶ The operation provided by the actor
▶ Tokens

Firing Rule: Synchronous Dataflow (SDF)

An actor may fire when:
▶ Each of its input arcs in the tuple $<i_0, i_1, \cdots, i_{k-1}>$ contains at least $<n_0, n_1, \cdots, n_{k-1}>$ tokens, and
▶ The number of slots available on each of the output arcs $<o_1, o_2, \cdots, o_{k'-1}>$ is sufficient to receive an additional count of $n'_l$ tokens.
A Short Introduction to Execution Models

The Synchronous Dataflow Model (Lee and Messerschmitt 1987)

Synchronous Dataflow Actors

Components of a regular actor:

- Input arcs; each input arc $i_i$ can contain a certain number $k_i$ of tokens
- Output arcs; each output arc $o'_i$ can contain a certain number $k'_i$ of tokens
- The operation provided by the actor
- Tokens
A Short Introduction to Execution Models

The Synchronous Dataflow Model (Lee and Messerschmitt 1987)

Synchronous Dataflow Actors

Components of a regular actor:

- Input arcs; each input arc \( i_l \) can contain a certain number \( k_i \) of tokens
- Output arcs; each output arc \( o'_l \) can contain a certain number \( k'_l \) of tokens
- The operation provided by the actor
- Tokens

Firing Rule: Synchronous Dataflow (SDF)

An actor may \textit{fire} when:

- Each of its input arcs in the tuple \(< i_0, i_1, \cdots, i_{k-1} >\) contains at least \(< n_0, n_1, \cdots, n_{k-1} >\) tokens, and
- The number of slots available on each of the output arcs \(< o_1, o_2, \cdots, o_{k'-1} >\) is sufficient to receive an additional count of \( n' \) tokens.
A Short Introduction to Execution Models

The Synchronous Dataflow Model (Lee and Messerschmitt 1987)

Synchronous Dataflow Actors

Components of a regular actor:

- Input arcs; each input arc $i_i$ can contain a certain number $k_i$ of tokens
- Output arcs; each output arc $o'_i$ can contain a certain number $k'_i$ of tokens
- The operation provided by the actor
- Tokens

Firing Rule: Synchronous Dataflow (SDF)

An actor may fire when:

- Each of its input arcs in the tuple $< i_0, i_1, \cdots, i_{k-1} >$ contains at least $< n_0, n_1, \cdots, n_{k-1} >$ tokens, and

- The number of slots available on each of the output arcs $< o_1, o_2, \cdots, o_{k'-1} >$ is sufficient to receive an additional count of $n'$ tokens.
### Other Dataflow Models

#### Dynamic Dataflow
- Allows for arbitrary recursions
- Relies on “color-matching:”
  - Each iteration is assigned a “color,”
  - An actor only fires if all tokens from the same color are present on its input arcs.
- Proved to provide maximum parallelism
- However: Color matching is slow (use of hash tables, ...)

#### Macro-Dataflow
- Idea: Instead of relying on fine-grain, one-operation-at-a-time actors, let's use a bunch of instructions/operations in sequence within the actor
- Still relies on inputs and outputs, but now the buffers may become much bigger, due to the amount of work and data required
- Offers a compromise to reduce the signaling overhead of fine-grain dataflow, token matching, etc.

See Watson and Gurd 1982; Arvind and Culler 1986; Papadopoulos and Culler 1990; Arvind and Gostelow 1982
Future extreme-scale systems will most likely feature thousands of cores on a chip, and deep memory hierarchies. The Codelet Model was created with this in mind. However, several problems still need to be tackled:

- Memory movements are expected to cost much more than computations in terms of energy consumption.
- There will be a need for fine-grain resource management to target goals such as:
  - Maximum or average power envelope during computation required by the user.
  - Degree of parallelism in the application declared by the user.
  - Maximum acceptable temperature levels.
  - ...
- We want to augment codelets and threaded procedures with meta-data which describe their resource usage.
- A low-level runtime will then be able to make smart decisions based on static meta-data as well as updated data collected during the codelets executions.
- The runtime will then be able to decide when to turn on/off parts of the manycore ships, when to rely on DVFS techniques, etc.

Extending the Codelet Model
Extending Codelets to Streams

Dataflow naturally maps to streams. However, the nature of future extreme-scale manycore processors will most likely be widely different:

- Heterogeneity is already becoming a reality at the chip level
  - AMD released its first Fusion processor (CPU+GPU on the same chip) in 2013
  - Next-generation Intel “co-processors” will be on package with traditional multicore chips
  - Nvidia just recently produced an accelerator board which embed arm processors to deal with more control-heavy workloads
- We predict heterogeneity will be ingrained at a much deeper level in processors
- This is a great opportunity to do research in that direction — and in particular by targeting streams

The NSF just accepted to provide funding to explore this venue. A high-level view of our objectives was recently published. See Zuckerman, Wei, et al. 2014 for more details.
References 1
Compiler Optimization, Performance Analysis, and Code Transformations in Multicore Systems


References II

Compiler Optimization, Performance Analysis, and Code Transformations in Multicore Systems


References – Codelet Model I
Specification and Implementation


References – Codelet Model II
Specification and Implementation


References – TERAFLUX
Making Codelets converge with DF-Threads


Other Dataflow and Data-Driven Related Work

- Jaime Arteaga et al. (May 2014). “Position Paper: Locality-Driven Scheduling of Tasks for Data-Dependent Multithreading”. In: Workshop on Multi-Threaded Architectures and Applications (MTAAP 2014). Phoenix, USA

Other References I


Other References II


- Nicholas P Carter et al. (2013). “Runnemede: An Architecture for Ubiquitous High-Performance Computing”. In: HPCA. Shenzhen, China