Analysis and software synthesis of KPN applications

Jeronimo Castrillon
Chair for Compiler Construction
TU Dresden
jeronimo.castrillon@tu-dresden.de

DREAMS Seminar
UC Berkeley, CA. October 22nd 2015
Acknowledgements

- Institute for Communication Technologies and Embedded Systems (ICE), RWTH Aachen
- Silexica Software Solutions GmbH
- German Cluster of Excellence: Center for Advancing Electronics Dresden (www.cfaed.tu-dresden.de)
Context: Cfaed

- Programming flows for future heterogeneous systems

**SW Layers and programming models**

- **Hardware abstractions**

... [Chemical processing [Voigt14]]
[Reconfigurable HW with Si-Nanowires [Trommer15]]
[Plasmonic wave-guides]

Courtesy: Thorsten Lars-Schmidt
Outline

- Motivation
- Input specs
- Analysis and synthesis
- Code generation and evaluation
- Summary
Outline

- **Motivation**
- Input specs
- Analysis and synthesis
- Code generation and evaluation
- Summary
Multi-processor on systems and applications

- **HW complexity**
  - Increasing number of cores
  - Increasing heterogeneity

- **Multi-cores everywhere**
  - Ex.: Smartphones, tablets and e-readers

- **SW “complexity”**
  - Not anymore a simple control loop
  - Need expressive models

---

![Graph showing PE Count in SoCs](image)

- **OMAP Family**
  - OMAP1
  - OMAP2
  - OMAP3530
  - OMAP3640
  - OMAP4430
  - OMAP4470
  - OMAP5430

- **Snapdragon Family**
  - S1 QSD8650
  - S2 MSM8250
  - S3 MSM8960
  - S4 APQ8064
  - S5 MSM8940

---

![Graph showing System Shipments](image)

- **Single CPU**
- **Multicore**

---

© J. Castrillon. DREAM Seminar, Oct. 2015
SW productivity gap

- SW-productivity gap: complex SW for ever-increasing complex HW
  - Cannot keep pace with requirements
  - Cannot leverage available parallelism
- Difficult to reason about time constraints
  - Even more difficult about energy consumption

- Need domain-specific programming tools and methodologies!
  - Parallelizing sequential codes
  - Parallel abstractions and mapping methods

In this talk: One such a flow for KPN applications (multimedia & signal processing domains)
Programming flow: Overview

KPN Application

Architecture model

Analysis

Synthesis

Code generation

Property models (timing, energy, error, ...)

Non-functional specification

© J. Castrillon. DREAM Seminar, Oct. 2015
Outline

- Motivation
- **Input specs**
- Analysis and synthesis
- Code generation and evaluation
- Summary
Kahn Process Networks (KPNs)

- Graph representation of applications
  - Processes **communicate only** over FIFO buffers
  - Good model for streaming applications
  - Good match for signal processing & multi-media

- Stereo digital audio filter

```
# include "PnTransform.h"
# include "PnVPUTG.h"
# include "PnVPUMap.h"
# include "clang/AST/ASTContext.h"
# include "PnStreamFactory.h"

using namespace clang;

void clang::PnTransform(TransformTarget transformTarget, bool traces, const std::string& strMappingFileName, const ASTContext& CtX, Sema& S, const llvm::sys::Path& BasePath) {
    assert(transformTarget != TransformInvalid);
    TranslationUnitDecl* D = CtX.getTranslationUnitDecl();
    PnTransform SizeOf(D, S);
    PnTransform Task(D, S);
    switch (transformTarget) {
    case TransformSystemC:
        case TransformVPUTG:
        case TransformVPUMap:
            PnCopying(D, S);
            break;
        default:
    }
    PnTransform Ptreads(D, S, traces);
    ErasePnPDef(Ds(D);
    break;
    case TransformSystemC:
        PrintForSystemC(D, S, traces, streamFactory);
        ErasePnPDef(Ds(D);
        break;
    case TransformVPUTG:
        PrintForVPUTG(D, S, streamFactory);
        ErasePnPDef(Ds(D);
        break;
    case TransformVPUMap:
        PrintForVPUMap(D, S, strMappingFileName, streamFactory);
        ErasePnPDef(Ds(D);
        break;
    case TransformInvalid:
        assert(false);
        break;
    }
}
```

```
src
fft_l
filter_l
ifft_l
sink
fft_r
filter_r
ifft_r
```
Language: C for process networks

- **FIFO Channels**
  ```c
  typedef struct { int i; double d; } my_struct_t;
  __PNchannel my_struct_t S;
  __PNchannel int A = {1, 2, 3}; /* Initialization */
  __PNchannel short C[2], D[2], F[2], G[2];
  ```

- **Processes & networks**
  ```c
  __PNkpn AudioAmp __PNin(short A[2]) __PNout(short B[2])
  __PNparam(short boost){
    while (1)
      __PNin(A) __PNout(B) {
        for (int i = 0; i < 2; i++)
          B[i] = A[i]*boost;
      }
  __PNprocess Amp1 = AudioAmp __PNin(C) __PNout(F) __PNparam(3);
  __PNprocess Amp2 = AudioAmp __PNin(D) __PNout(G) __PNparam(10);
  ```
Architecture model

- System model including:
  - Topology, interconnect, memories
  - Computation: cost tables (as backup)
  - Communication: cost function (no contention)
- Example: Texas Instruments Keystone

```
<Platform>
  <Processors List="dsp0 dsp1 dsp2 dsp3 dsp4 dsp5 dsp6 dsp7"/>
  <Memories List="local_mem_dsp0 L2 local_mem_dsp1 L2 local_mem_dsp2 L2 local_mem_dsp3 L2 local_mem_dsp4 L2 local_mem_dsp5 L2 local_mem_dsp6 L2">
    <CommPrimitives List="IPCII_SL2 IPCII_DDR EDMA3_SL2 EDMA3_DDR EDMA4_DDR EDMA5_DDR"/>
  </Memories>
</Platform>
```

```
<Processor Name="dsp0" CoreRef="DSPC66"/>
```

```
<Processor Name="dsp1" CoreRef="DSPC66"/>
```

```
<Processor Name="dsp7" CoreRef="DSPC66"/>
```

```
<Memory>
  <LocalMemory Name="local_mem_dsp0_L2" Size="524288" BaseAddress_hex="00800000" ProcessorRef="dsp0"/>
</Memory>
```

```
<Core Name="DSPC66" CoreType="DSPC66" Category="DSP">
<MultiTaskingInfo MaxNumberOfTasks="1">
  <ContextSwitchInfo StoreTime="1000" LoadTime="1000"/>
  <SchedulingPolicies List="FIFO PriorityBased"/>
</MultiTaskingInfo>
```

```
<CostTable>
  <Operation Name="Load">
    <VariableType Name="Char"> <Cost>1</Cost> </VariableType>
    <VariableType Name="Double"> </VariableType>
```

[Oden13]
Architecture model: Communication

- Piecewise curve-fitting from measurements
Architecture model: Communication (2)

- Models for Network on Chips (NoC)
- Channels can be mapped to
  - Local scratchpad (producer or consumer)
  - Global SDRAM

Cost model & measurement

![Graph showing cost model and measurement](image-url)
Constraints

- **Timing constraints**
  - Process throughput
  - Latencies along paths
  - Time triggering
- **Mapping constraints**
  - Processes to processors
  - Channels to primitives
- **Platform constraints**
  - Subset of resources (processors or memories)
  - Utilization
Algorithmic description

- **Extended application specification**
  - Selected processes are algorithmic kernels with **algorithmic parameters**

- **Extended platform model**
  - SW/HW accelerated kernels and their **implementation parameters**
Outline

- Motivation
- Input specs
- Analysis and synthesis
- Code generation and evaluation
- Summary
Analysis and synthesis: Overview

CPN application
Architecture model
Non-functional specification

Analysis: Instrumentation, profiling, tracing
Sequential performance estimation
Time-annotated traces
Mapping and scheduling
Parallel perf. estimation
Increase resources

Mapping configuration

Increase resources
Tracing: Dealing with dynamic behavior

- KPNs do not have firing semantics
- **White model of processes**: source code analysis and tracing
- Tracing: instrumentation, token logging and event recording

```plaintext
... for (; i < x; i++) {
    write(&c2);
    f1(...);
    read(&cl);
    f2(...);
    read(&cl);
    ...
```
Sequential performance estimation

- Fine-grained: Sometimes within code basic-blocks
- IR-level instrumentation
  - Cost tables for different architectures
  - Execution count in between events
- Advanced: Emulate effect of target compilers and back annotate to IR
Sequential performance estimation (2)

- Processor models

Execution counts, branch stats and execution traces

[Gense14]
Sequential performance estimation (3)

- Abstract models for compiler emulation
  - Resources (functional units, register banks)
  - Operations (pipeline effects, SIMD, addressing, predicated exec.)
  - SW-related costs (calling convention, register spilling, C-lib calls)

Processor Model

<table>
<thead>
<tr>
<th>FU₁</th>
<th>FU₂</th>
<th>FU₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>ASR</td>
<td>LD</td>
</tr>
<tr>
<td>SUB</td>
<td>LSL</td>
<td>ST</td>
</tr>
<tr>
<td>MUL</td>
<td>LSR</td>
<td>CMP</td>
</tr>
<tr>
<td>OR</td>
<td>ZEX</td>
<td>BR</td>
</tr>
<tr>
<td>RF₁(16,32)</td>
<td>RF₂(8,32)</td>
<td></td>
</tr>
</tbody>
</table>

Conventions
- Prologue: \(1+2 \times X_{\text{arg}}\)
- Epilogue: 4
- Branch\(_{\text{overh}}\): 3

External library costs
- malloc: \(9+0.3 \times N_{\text{size}}\)
- fsqrt: 235

© J. Castrillon. DREAM Seminar, Oct. 2015
Parallel performance estimation

- Discrete event simulator to evaluate a solution
  - Replay traces according to mapping
  - Extract costs from architecture file (NoC modeling, context switches, communication)

Mapping configuration

Trace Replay Module (TRM)

Architecture model

Context switch

Blocked time

Grantt Chart, platform utilization, channel profiles, ...

Time-annotated traces

© J. Castrillon. DREAM Seminar, Oct. 2015
Trace-based synthesis

- Synthesis based on code and trace analysis (using simple heuristics)
  - Mapping of processes and channels
  - Scheduling policies
  - Buffer sizing
Event traces can be represented as large dependence graphs.
Trace-based algorithms (2)

- Event traces can be represented as large dependence graphs
- Possible to reason about
  - Channel sizes and memory allocation
  - Mapping and scheduling onto heterogeneous processors

\[ size(chan. 2) = 2, \text{ size(chan. 1,3) }= 1 \]
Dealing with heterogeneity: group-based mapping (GBM)

1) Initialize: All to all
2) Select element: Trace graph critical path
3) Reduce group
4) Assess & propagate
5) Quasi-homogeneous

[Castrill12]
Mapping for HW accelerators

- Not only mapping but also configuration
  - Match algorithmic parameters with implementation parameters
  - Adjust synchronization and communication protocols

Algorithm library

Platform model + characterization of special components

[Castrill10, Castrill11]

N: Algorithmic actors
F: Existing implementation in target platform
Multiple traces

- Different input $\rightarrow$ different behavior (traces)
  - Characterize behaviors and impact on mapping performance

[Goens15]
Multiple traces (2)

- Different input $\rightarrow$ different behavior (traces)
  - Characterize behaviors and impact on mapping performance
Multiple traces (3)

- Behavior difference as metric in trace/history monoid

Random KPNs: Slow down w.r.t. optimum vs. trace distance
→ No correlation

JPEG application: Observed trace groups
Increasing resources

- Add resources to the synthesis until constraints are met

- Add processors and memories
  - Easy for homogeneous platforms
  - Non trivial for heterogeneous platforms
Increasing resources: Exploit symmetries

- Identify mapping **equivalent classes** due to HW symmetries

- Do not evaluate equivalent mappings
- Reduce search space when adding resources
- Multiple traces (revisit)
  - Random traces: **5 out of 83** classes account for **50%** of all optimal mappings
- Application to multi-application analysis

[Goens15]
Outline

- Motivation
- Input specs
- Analysis and synthesis
- **Code generation and evaluation**
- Summary
- Take mapping configuration and generate code accordingly
- From architecture model: APIs, configuration parameters, …
- Sample targets: experimental heterogenous systems, TI (Keysonte, TDA3x), Parallela/Ephiphany, ARM-based (Exynos, Snapdragon)
Debugging with virtual platforms

- Interactive debugging
  - Get snapshots of the system state
  - Full system stop
  - Track progress irrespective of mapping

Internal state of the MPSoC schedule: assigned, blocked and running tasks

[Castrillon11]
Debugging with virtual platforms (2)

- Deterministic replay and automatic bug exploration

[Diagram showing high-level events, monitor, application, MPSoC, oracle, controller, iterative bug exploration, and control actions]

[Murillo14]
Evaluation and results

- Virtual platforms: SystemC models of full systems
  - Explore heterogeneous architectures
  - Easier to integrated state-of-the-art accelerators
  - Configurable accuracy

- Real platforms for validation
  - Speedup on commercial platforms
  - Code generation against vendor stacks
Example: multi-media applications

- Platform: 2 RISCs, 4 VLIW, 7 Memories
Example: multi-media applications (2)

- Dealing with real-time constraints

![Graphs showing Iterative Mapping](image)

- Tool: ~1 min. for LP-AF, ~7 min. for MJPEG
- Sim.: ~6 days for LP-AF, ~24 for MJPEG

~10 min. ~10 days (~x10^3)
Example: With HW acceleration

- Application: MIMO OFDM receiver
- Hardware
  - Platform 1: Baseline software
  - Platform 2: Optimized software
  - Platform 3: Optimized SW + HW

Achieved rate @ 100 MHz

<table>
<thead>
<tr>
<th>Rate (bps)</th>
<th>1) bsp1 (sw, unoptimized)</th>
<th>2) bsp2 (sw, optimized)</th>
<th>3) bsp3 (hw)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7680</td>
<td>128000</td>
<td>1758241,758</td>
</tr>
</tbody>
</table>

Library 1: SW implementations
Library 2: SW optimized
Library 3: SW+HW accel.
Manual vs. Automatic: TI Keystone

- **Image processing**
- **Audio filtering application**
- **LTE digital receiver**

[Aguilar14]
TRM vs. Actual execution: TI Keystone

- Makespan: LP-AF
  - Time (Mcycles) vs. Number of Cores
  - Makespan error < 20% (Avg. 7%)

- Makespan: JPEG
  - Time (Mcycles) vs. Number of Cores
  - Makespan error < 20% (Avg. 7%)

- Makespan: MJPEG
  - Time (Mcycles) vs. Number of Cores
  - Makespan error < 20% (Avg. 7%)
  - Speedup error < 8% (Avg. 5%)
Outline

- Motivation
- Input specs
- Analysis and synthesis
- Code generation and evaluation
- Summary
Summary

- Tool flow for mapping KPN applications
  - CPN language: a language for KPNs close to C
  - Analysis and synthesis based on traces
  - Extensions for: multiple traces and algorithmic descriptions
  - Backends for multiple platforms (bus and NoC-based)

- Current and future work
  - More on static code analysis
  - Continue on multiple-traces and symmetries
  - Applying to other domains (server applications)
References


Thanks!
Questions?