CoEx: Novel Profiling-Based Algorithm/Architecture Co-Exploration for ASIP Design

Juan Eusse, Christopher Williams, Rainer Leupers
Chair for Software for Systems on Silicon (SSS)

ReCoSoC 2013, Darmstadt July 10th, 2013
1. Why do we need ASIP oriented profiling? (I)

- In a world of changing standards, how to keep the right amount of flexibility while being efficient?

Source: T.Noll, RWTH Aachen
1. Why do we need ASIP oriented profiling? (II)

- **Architecture Description Languages (e.g. LISA) -based tools can:**
  - Generate the SW environment (assembler, linker, simulator, compiler)
  - Generate HDL descriptions
- **Profiling has remained the entry point to all ADL-based methodologies**

![Diagram showing the profiling process](image-url)
Input specification comes as “high-level” C/C++ code
- Usually directly from the algorithm designer

Profiling used only to detect application “hotspots”
- SLP tools are intended for GP program analysis
- Emulation-Based is more accurate but cannot be reused
- ISS/HW based requires the existence of a target processor architecture
Main Goals:

- Profile at source level, using different granularities controlled by the designer (profiling scenarios)
- Retain execution speed inherent to native execution
- Allow the greatest possible flexibility while keeping target independence
- Generate information to ease algorithmic exploration and architecture customization
Available profiling configurations related to the ASIP design stage

**Algorithmic Exploration**

1. Hotspot detection
2. Common sub-case optimizations
3. Memory usage optimization
4. Numerical transformations

**Architecture Customization**

1. Instruction set design
2. Data path construction and sizing
3. Custom memory architectures/hierarchies
4. Specialized HW (branch predictors, ZOL, AGUs)

**Profiling configuration**
Standalone Multi-Grained SLP based on LLVM code instrumentation

- Granularity of the *profiling scenario* is configured by the designer

- Generated profiling information is independent of the target architecture
4. CoEx implementation (II)

void merge_lines(char cond) {
    float *sPtr = (float*)malloc(2*sizeof(float));
    if(cond) sPtr[index] = val; else sPtr[index+1] = val;
    free(sPtr);
}

entry:
<xml version="1.0" encoding="utf-8"?>
call void (i8*,...) @PROF_LSInitLocalInfo (...)?
<profConfig>
%call = call i8* @PROF_Malloc(i64 8)
%tobool = icmp ne i8 %1, 0
br i1 %tobool, label %if.then, label %if else
</profConfig>
<fnBbSt enabled="true"/>
<traceOutput filename="trace.txt"/>
if.then:
call void @PROF_FNProcessFunctionEntry(...)
%call1 = call float @fn1()
call void @PROF_FNProcessFunctionExit(...)
store float %call1, float* %arrayidx
br label %if.end
if.else:
call void @PROF_Mem( ...)
store float %5, float* %arrayidx2
br label %if.end
if.end:
call void @free(i8* %8)
ret %12
if.end:
call void @PROF_Free(i8* %9)
ret %12

Pure LLVM IR
Instrumented LLVM IR
4. CoEx implementation (III)

- **Static File:**
  - Language dependent information

- **Dynamic File:**
  - Application execution extracted information

- **General Trace**
  - Functions, basic blocks, memory

- **Value Trace:**
  - Individual value traces

---

Size and type of output depends on the **profiling scenario** configuration.
4. CoEx implementation (IV)

- Profiling results visualization:
  - Intuitive navigation through the profiling results
  - Linking/highlighting of the application source code
4. CoEx implementation (IV)

- Pre-architectural performance estimation

Abstract Processor Model (APM)

instruction Set

Functional Units (FUs)

Library calls cost calibration

Instruction Latencies

FU characteristics

IR - Instruction mapping

List Scheduler

LLVM IR

entry

for.cond

LLVM IR Instructions

BB Cost

Exec.Count

Branch Penalties

Performance Estimate

Profiling Information

- Exec Ct: -

Branch Info

True: -

False: -

- Exec Ct: 25

Branch Info

Unconditional

- Exec Ct: 422

Branch Info

True: 80%

False: 20%

- Exec Ct: -

Branch Info

True: -

False: -

- Exec Ct: -

Branch Info

True: -

False: -

Instruction Set

ADD

LD

NOP

NOP

MUL

CALL

NOP

NOP

ADD

MUL

NOP

NOP

ORI

Instruction Latencies

FU characteristics

IR - Instruction mapping

Library calls cost calibration

Pre-architectural performance estimation

30x465
4. CoEx implementation (IV)
5. Evaluation: Execution Overhead (I)

- **Instrumentation Overhead:**
  - Generated *profiling scenarios* for AES, JPEG, ADPCM, FFT(iFFT), Blowfish, Susan from DSPStone and EEMBC.
  - Two non-optimized applications considered:
    - Audio filter application
    - Planar marker detection for augmented reality – case study
  - *Profiling scenarios* tuned to match existing SLP analyses
  - Native execution time is the baseline for overhead calculations

Overhead compared with gprof, gcov, callgrind (function call), leak-check (dynamic memory), massif (stack/heap), dhat (memory accesses), bbv (basic block tracing)

*ISS-CA has no equivalent configuration within CoEx*

© Juan Eusse – 2013
ReCoSoC 2013
Customization of the PD_RISC processor for an AR application

Detect black-and-white 2-Dimensional markers in an image
- Input specification consists on ~2900 lines of C code
- Function pointers, recursion, SP floating-point, dynamic memory management heavily used

Algorithm steps:
1. Divide the image into 40x40px regions
2. Detect pixels with strong magnitude changes
3. Detect which belong to straight lines
4. Merge compatible lines (super-lines)
5. Extend super lines until corners
6. Keep lines that have corners
7. Build line chains
8. Detect markers
Profiling Scenario 1: Function/Basic Block/Timing analysis (no trace)

- Light-weight profiling (low execution overhead)
- Steps (3) and (4) of the algorithm consume 29% and 40% total execution time

<table>
<thead>
<tr>
<th></th>
<th>Memory Address</th>
<th>Load-Store</th>
<th>Integer Ops.</th>
<th>Floating Point Ops.</th>
<th>Function Execution Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line check</td>
<td>11</td>
<td>70</td>
<td>3</td>
<td>25</td>
<td>693600</td>
</tr>
<tr>
<td>2x1 Vector Normalization</td>
<td>12</td>
<td>50</td>
<td>2</td>
<td>10</td>
<td>734044</td>
</tr>
<tr>
<td>2x1 Dot Product</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>3</td>
<td>2264043</td>
</tr>
<tr>
<td>Square Root</td>
<td>0</td>
<td>17</td>
<td>2</td>
<td>5</td>
<td>1073440</td>
</tr>
<tr>
<td>2x1 Vector Length</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>1099245</td>
</tr>
</tbody>
</table>

- 10% in calls to malloc/free
Profiling Scenario 2: Function/Basic Block profiling (stack/heap trace)
- Observed initial/final frame memory (de)allocation
- Closer look revealed repetitive (de)allocation
- Trace examination enabled:
  - Static memory and memory pool sizing

13% of overall execution speedup
No architectural customization
Profiling Scenario 3: Hotspot input/output value trace
- Traced hotspots from profiling scenario 1
- Assumed a 32bit fixed point word
- Explored MSE for different quantization schemes (using Matlab)

- Replaced floats by Q21.10 fixed point
- 27% further speedup
- Still no architectural customization
Profiling Scenario 4: Function/Basic Block/Memory Access profiling (Fn/BB traces enabled)

- Exploration of the generated information through the GUI
- Architecture customization only done using fusion-type instructions:
  - Fixed point support for the ALU
  - SIMD addition, subtraction and multiplication
  - Dot product for 2x1 vectors
  - Reciprocal square root approximation

6x combined speedup achieved in only two days of design time
### 4. Case Study: Planar-Marker detection for AR (V)

- **Pre-architectural performance estimation of case study results**
  - Estimation performed after each successive algorithm/architecture iteration
  - Accuracy metric based on CA simulation results from ISS

<table>
<thead>
<tr>
<th>Application/Architecture Revision</th>
<th>ISS-CA Cycles</th>
<th>Estimated Cycles</th>
<th>Error (%)</th>
<th>ISS Time (sec)</th>
<th>Estimation Time (sec)</th>
<th>Estimation/Simulation Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input specification + PD RISC (Base)</td>
<td>3705186373</td>
<td>2970991784</td>
<td>-19.82</td>
<td>4147</td>
<td>1.23</td>
<td>3371</td>
</tr>
<tr>
<td>Static Memory + PD RISC (Base)</td>
<td>3403357531</td>
<td>2688236170</td>
<td>-21.01</td>
<td>3762</td>
<td>1.21</td>
<td>3109</td>
</tr>
<tr>
<td>Fixed Point + PD RISC (Base)</td>
<td>2658942738</td>
<td>2238013034</td>
<td>-15.83</td>
<td>2991</td>
<td>1.22</td>
<td>2471</td>
</tr>
<tr>
<td>Fixed Point + PD RISC (Fixed +Vector)</td>
<td>1670310514</td>
<td>1365812907</td>
<td>-18.23</td>
<td>2948</td>
<td>1.25</td>
<td>2358</td>
</tr>
<tr>
<td>Fixed Point + PD RISC (Square Reciprocal approx.)</td>
<td>622717072</td>
<td>514052942</td>
<td>-17.45</td>
<td>2991</td>
<td>1.24</td>
<td>2412</td>
</tr>
</tbody>
</table>
We propose Multi-Grained Profiling, which combines granularity levels according to the ASIP design stage to ease \textit{algorithmic exploration, application optimization} and \textit{architecture exploration}.

We have implemented an MGP-enabled profiling tool (CoEx) to test the validity of the approach.

Although the execution overhead regarding native execution is considerable, the amount of generated information and the possibility of re-using it for other analyses (i.e. performance estimation) compensates such overhead.

A GUI has been developed to help the designer in the analysis of the generated profiling information.
Pre-architectural performance estimation of early architectural decisions has been also explored, obtaining fairly accurate results without the need for application simulation on an ISS.

In the case study we have shown that by using CoEx, a designer can grasp the inner workings of an application specification in a time efficient manner.

Furthermore, we were able to customize the PD_RISC processor in just two days design time to detect planar markers in 2D images, obtaining 6x performance gains.

Future work will explore more in depth performance estimation based on abstract processor models, in order to get more accurate results.
Questions?

Thank you!