OFFIS Transportation
INSTITUTE FOR
INFORMATION TECHNOLOGY

A Workload Extraction Framework for Software Performance Model Generation

Verkehr
Transportation

Philipp Ittershagen,
Philipp A. Hartmann, Kim Grüttner,
Wolfgang Nebel
philipp.ittershagen@offis.de
OFFIS—Institute for Information Technology
R&D Division Transportation

09.12.2014
WP2/WP4 Meeting
Oldenburg

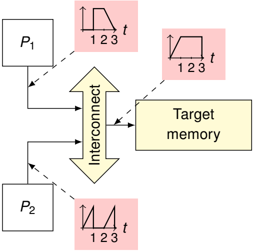
1 Motivation

Introduction

- MPSoC: high **feature density** through **shared resource** usage
    - e.g. shared memory, interconnect, peripherals, . . .

Software performance behaviour estimation challenging:

- mutual interference from **shared resources**
    - also caused by implicit **memory usage**
- delayed **local** execution times on each P_i
- design decision **impact** only visible in an **advanced step** of the design process
    - Needed: Instruction-Set Simulator, fully configured virtual platform, complete target toolchain, . . .



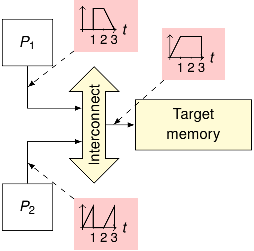
1 Motivation

Introduction

- MPSoC: high **feature density** through **shared resource** usage
  - e.g. shared memory, interconnect, peripherals, . . .

Software performance behaviour estimation challenging:

- mutual interference from **shared resources**
  - also caused by implicit **memory usage**
- delayed **local** execution times on each P_i
- design decision **impact** only visible in an **advanced step** of the design process
  - Needed: Instruction-Set Simulator, fully configured virtual platform, complete target toolchain, . . .



How to provide an early estimation of software performance behaviour on the application level?

2 Key Observations on Performance Analysis

Introduction

- shared resource contention often approximated by **traffic generators**
  - no application-specific **control flow modelling**
- **precise functional** behaviour is of **limited** interest for mapping, interference, or throughput analysis
  - i.e. DCT algorithm operating on memory regions: resource usage **mostly independent** of image content
  - distinction between **control** and **data flow** basic blocks
- enabling an **early** performance estimation helps in performing **key design decisions**

Goal: provide automatic **workload extraction** of an application's **performance characteristics** for early estimations in the design flow.

3 Idea of this Approach

Introduction

1. During compilation, **classify** control- and data flow-related basic blocks

3 Idea of this Approach

Introduction

1. During compilation, **classify** control- and data flow-related basic blocks
2. Derive a software performance model by **static analysis**

   of the application’s performance characteristics of the target instructions (captured in **characterization vectors**)

   - local **execution delay** and **memory access** patterns

3 Idea of this Approach

Introduction

1. During compilation, **classify** control- and data flow-related basic blocks
2. Derive a software performance model by **static analysis**

   of the application's performance characteristics of the target instructions (captured in **characterization vectors**)

   - local **execution delay** and **memory access** patterns
3. Annotate **characterized** target software behaviour

   back to target-independent, intermediate representation

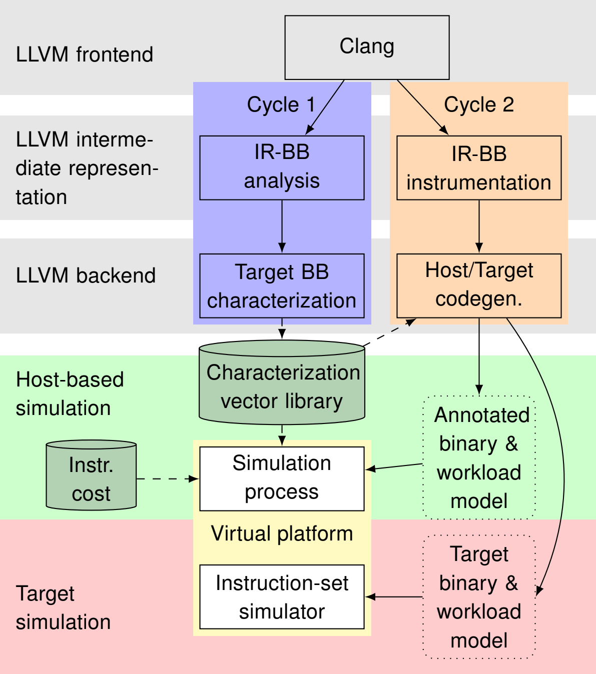
3 Idea of this Approach

Introduction

1. During compilation, **classify** control- and data flow-related basic blocks
2. Derive a software performance model by **static analysis**

   of the application's performance characteristics of the target instructions (captured in **characterization vectors**)
   - local **execution delay** and **memory access** patterns
3. Annotate **characterized** target software behaviour

   back to target-independent, intermediate representation
4. Code generation process
   1. Create an annotated workload model suitable for a **native** simulation
      - **functional behaviour** of data flow basic blocks may even be **omitted** in native simulation
   2. Test model accuracy: generate **target workload model** from characterization information
      - **replace** data flow basic blocks with their workload representation by generating **target code** from the characterization vectors


LLVM frontend
Clang
Cycle 1
Cycle 2
LLVM intermediate representation
IR-BB analysis
IR-BB instrumentation
LLVM backend
Target BB characterization
Host/Target codegen.
Host-based simulation
Characterization vector library
Annotated binary & workload model
Instr. cost
Simulation process
Virtual platform
Target simulation
Instruction-set simulator
Target binary & workload model

5 Running Example: Lifetime of a Basic Block

Software Performance Model Approach

Compiler input:
C code

```
int arr[6];

void main()
{
  arr[0] = arr[1]
         + arr[2];


  arr[3] = arr[4]
         + arr[5];
}
```

During Compilation:
Intermediate Representation

```
%0 = load i32* ; ...
%1 = load i32* ; ...
%add = add nsw i32 %0, %1
store i32 %add, i32* ; ...
%2 = load i32* ; ...
%3 = load i32* ; ...
%add1 = add nsw i32 %2, %3
store i32 %add1, i32* ; ...
ret void
```

Compiler Output:
Target Instructions

```
ldr r0, [pc + 36]
ldr r0, [r0]
ldr r1, [r0 + 4]
ldr r2, [r0 + 8]
add r1, r1, r2
str r1, [r0]
ldr r1, [r0 + 16]
ldr r2, [r0 + 20]
add r1, r1, r2
str r1, [r0 + 12]
bx  lr
```

6 Characterization and Annotation Process

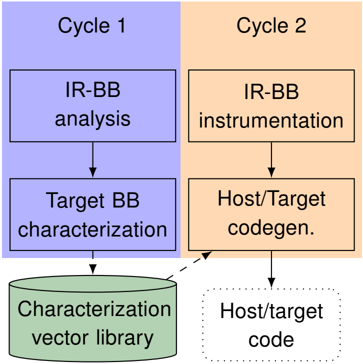
Software Performance Model Approach

Cycle 1

1. **analyse** IR basic block dependencies
   - mark basic blocks containing **no control flow** logic
2. **characterize** target instructions of all basic blocks
   - create **characterization vector** library

Cycle 2

1. **instrumentation**: annotate characterization vectors on intermediate representation
2. output **target** or **native** binary
   - host/target **binary**
   - host/target **workload model** (data flow basic blocks replaced/removed)



7 Characterization Vectors

Software Performance Model Approach

A **characterization vector** . . .

- characterizes a **set** of instructions,
- contains information about **temporal properties** of each instruction,
  - **cost factor**, always 1 in this case
- classifies instructions according to their **memory access behaviour** (read=R, write=W, nop=N).

ARM Target Code	V_0
`ldr r0, [pc + 36]`	$(1, R)$
`ldr r0, [r0]`	$(1, R)$
`ldr r1, [r0 + 4]`	$(1, R)$
`ldr r2, [r0 + 8]`	$(1, R)$
`add r1, r1, r2`	$(1, N)$
`str r1, [r0]`	$(1, W)$
`ldr r1, [r0 + 16]`	$(1, R)$
`ldr r2, [r0 + 20]`	$(1, R)$
`add r1, r1, r2`	$(1, N)$
`str r1, [r0 + 12]`	$(1, W)$
`bx  lr`	$(1, N)$

8 Instrumentation

Software Performance Model Approach

We have:

- a **characterization vector** for each target basic block,
- the target basic block relations to **IR basic blocks**.

We can now insert an **annotation** with a characterization vector identifier to **each IR basic block**.

Annotated IR basic block

```
%0 = load i32* ; ...
%1 = load i32* ; ...
%add = add nsw i32 %0, %1
store i32 %add, i32* ; ...
%2 = load i32* ; ...
%3 = load i32* ; ...
%add1 = add nsw i32 %2, %3
store i32 %add1, i32* ; ...
call void @__bb_mark(i32 0) ; V_0
ret void
```

9 Host Binaries and Workload Model Generation

Software Performance Model Approach

Use annotations in **code generation** step:

- **native** code generation
  for host-based simulation
  - update the **processor simulation model** using characterization vector information
  - annotation compiled to **function call**
  - link with **wrapper** implementing function call

Native code and model update

```
mov     $0x0,%eax # BB id 0
mov     0x4,%ecx
# [...]
add     0x14,%ecx
mov     %ecx,0xc
movl    $0x0,(%esp)
mov     %eax,-0x4(%ebp)
call    3a # call __bb_main(0)
add     $0x8,%esp
pop     %ebp
ret
```

9 Host Binaries and Workload Model Generation

Software Performance Model Approach

Use annotations in **code generation** step:

- **native** code generation for host-based simulation
  - update the **processor simulation model** using characterization vector information
  - annotation compiled to **function call**
  - link with **wrapper** implementing function call
- host-based **workload model**
  - replace **data flow basic block** with annotation
  - leave out data flow, just update model

Update processor model only

```
push    %ebp
mov     %esp,%ebp
sub     $0x8,%esp
movl    $0x0,(%esp)
call    e # call __bb_main(V_0)
add     $0x8,%esp
pop     %ebp
ret
```

10 Target Binaries and Workload Model Generation

Software Performance Model Approach

Use annotations in **code generation** step:

- **target** code generation for target (ISS-based) simulation
  - generate **original target binary** by ignoring annotations during code generation process

V_0	**Target code**
$(1, R)$	`ldr r0, [pc + 36]`
$(1, R)$	`ldr r0, [r0]`
$(1, R)$	`ldr r1, [r0 + 4]`
$(1, R)$	`ldr r2, [r0 + 8]`
$(1, N)$	`add r1, r1, r2`
$(1, W)$	`str r1, [r0]`
$(1, R)$	`ldr r1, [r0 + 16]`
$(1, R)$	`ldr r2, [r0 + 20]`
$(1, N)$	`add r1, r1, r2`
$(1, W)$	`str r1, [r0 + 12]`
$(1, N)$	`bx  lr`

10 Target Binaries and Workload Model Generation

Software Performance Model Approach

Use annotations in **code generation** step:

- **target** code generation for target (ISS-based) simulation
  - generate **original target binary** by ignoring annotations during code generation process
- target **workload model**
  - replace **data flow basic block** with characterization vector **representation**
  - generate **target** instructions according to characterization vector content

Target workload model

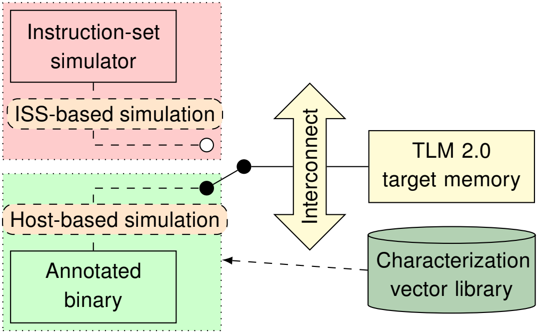
V_0	$V_i \rightarrow Instr$
$(1, R)$	`ldr r1, [r0]`
$(1, R)$	`ldr r1, [r0]`
$(1, R)$	`ldr r1, [r0]`
$(1, R)$	`ldr r1, [r0]`
$(1, N)$	`mov r1, r1`
$(1, W)$	`str r1, [r0]`
$(1, R)$	`ldr r1, [r0]`
$(1, R)$	`ldr r1, [r0]`
$(1, N)$	`mov r1, r1`
$(1, W)$	`str r1, [r0]`
$(1, N)$	`mov r1, r1`

11 Setup

Evaluation

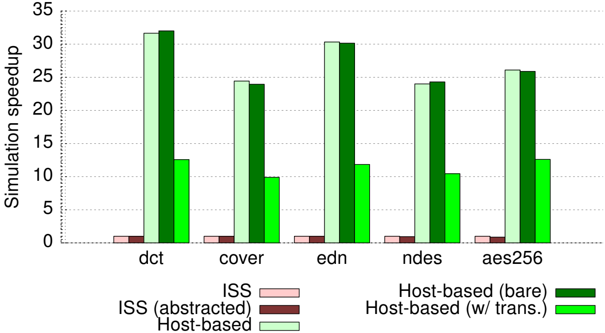
Virtual platform configuration:

- ISS **replaceable** with wrapper for host-based simulation binary



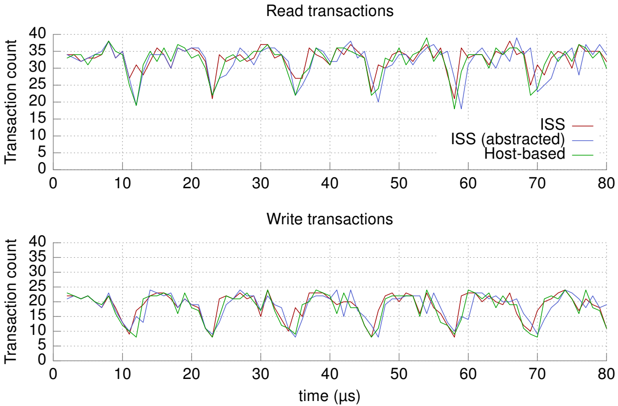
12 Simulation Speed-up

Evaluation



- **ISS (abstracted):** target binary (with generated target instructions)
- **Host-based (bare):** native simulation (without data flow blocks)
- **Host-based with trans.:** native simulation including bus model

13 Example Results: DCT
Evaluation



- **ISS (abstracted):** target binary (with generated target instructions)
- **Host-based:** native simulation

14 More results: Host-based Workload Model vs. Original ISS Behaviour

Evaluation

Workload model with characterization vectors vs. original binary trace from ISS.

	Simulated time		Generated transactions	
Example	Abs. (ns)	Dev. (%)	Abs.	Dev. (%)
`dct`	231 930.050	<0.01	11 587	−2.74
`edn`	1 450 110.050	−3.21	84 949	−1.93
`cover`	51 000.050	−0.02	3096	−0.23
`ndes`	1 110 198.100	−0.02	68 623	−5.76
`aes256`	581 730.050	<0.01	24 165	−0.92

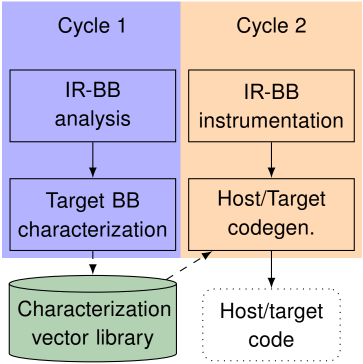
15 Limitations

Evaluation

- target support depends on LLVM **backend**
  - supports x86, AMD64, PowerPC (64), ARM, Thumb, SPARC, Alpha, MIPS, System Z, Xcore
  - support for Microblaze dropped due to missing maintainer
- characterization vector currently does not contain information about **memory addresses**
  - cache models currently not supported
- optimizations for **characterization** phase currently not handled
  - need $1 - n$ mapping from IR to target basic blocks
  - however, output for optimized **annotated** IR possible

16 Conclusion

- source code characterization on a basic block-level considering both **local execution** times and **shared resource** usage patterns
- native simulation with annotated binary
  - without data flow basic block functionality
- ISS-based simulation with target binary
  - replace **data flow basic blocks** with characterization vector representation
- characterization vectors are a way to statically account **temporal** as well as **memory usage behaviour**



17 Literature

Ittershagen, P., Hartmann, P. A., Grüttner, K., and Nebel, W.
A Workload Extraction Framework for Software Performance Generation
Proceedings of the 7th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO'15), Amsterdam, The Netherlands