

System simulation for VLIW based systems

Luc Michel, **Nicolas Fournel** and Frédéric Pétrot

Tima Laboratory



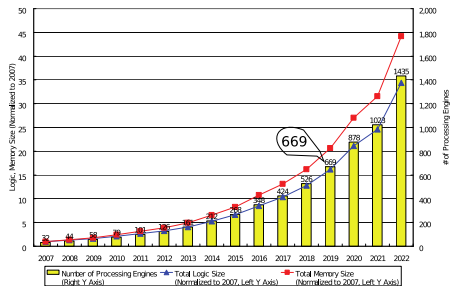
- **Introduction**
 - Context & motivations
- **Technology**
 - DBT
 - VLIW architectures
- **VLIW DBT**
 - Algorithm
 - Implementation
- **Conclusion**

Plan

- **Introduction**
 - Context & motivations
- **Technology**
 - DBT
 - VLIW architectures
- **VLIW DBT**
 - Algorithm
 - Implementation
- **Conclusion**

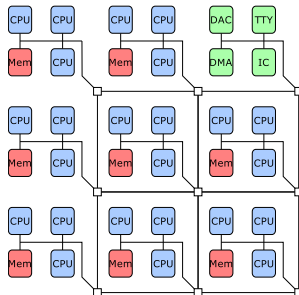
MPSoC Perspectives

ITRS Roadmap for the number of cores in consumer devices



Characteristics of future SoCs architecture

- ▶ Massively parallel
- ▶ Shared memory
- ▶ More and more homogeneous
- ▶ VLIW architectures



Simulation Context

Motivations

- ▶ Design space exploration and Early Software Development
- ▶ Goal: optimize chips for performances (time and energy)

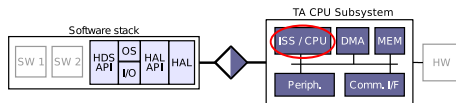
Difficulties

- ▶ Complexity reduces simulation performances (number of processors)
- ▶ Precision influences simulation speed (trade-off)
- ▶ Simulation speed is however a great concern

Previously Proposed solution

Dynamic Binary Translation based ISS.

- ▶ Pros: fast and precise
- ▶ Cons: complex development



Plan

- **Introduction**
 - Context & motivations
- **Technology**
 - DBT
 - VLIW architectures
- **VLIW DBT**
 - Algorithm
 - Implementation
- **Conclusion**

Dynamic Binary Translation (DBT)

Characteristics

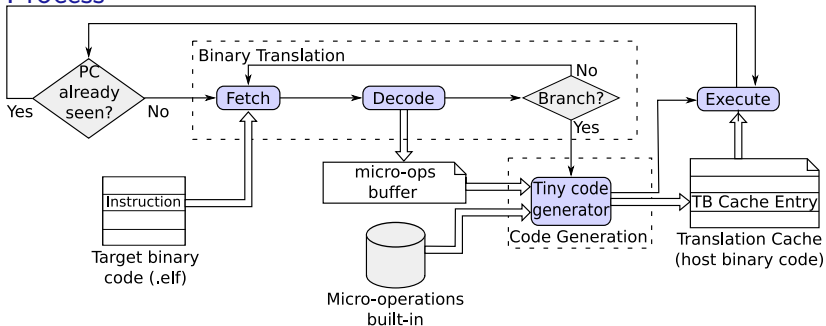
- ▶ Two step process:
 - Translation: from target binary to IR
 - Code generation: from IR to host binary
- ▶ Translation unit: translation block

History

- ▶ Originally developed for backward compatibility
- ▶ Largely used in virtualization today
- ▶ VLIW DBT has not been handled as of today:
 - effort for scalar to scalar DBT (classical situation)
 - scalar to VLIW examples (IA32 to IA64)

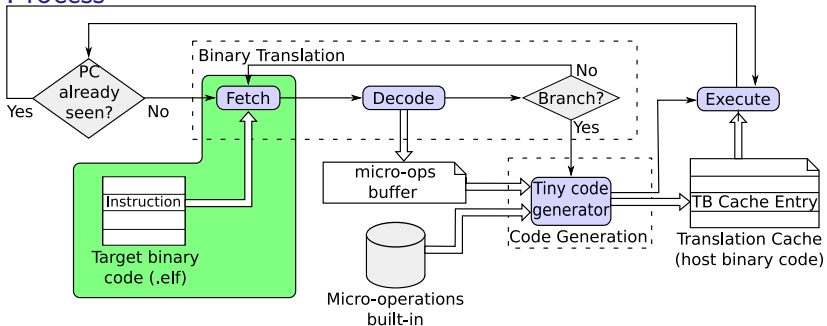
Dynamic Binary Translation Process

Process



Dynamic Binary Translation Process

Process

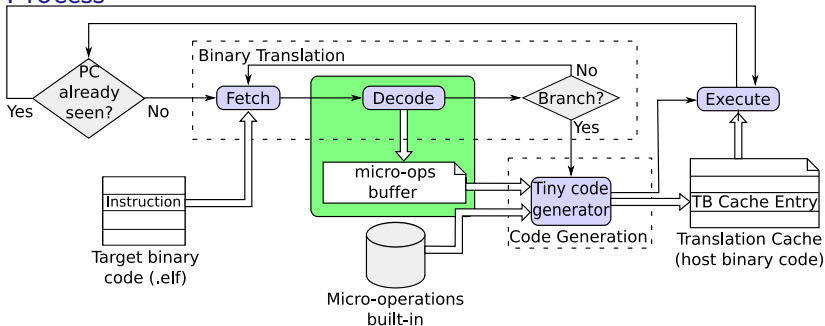


Code generation example

18 target_instrX

Dynamic Binary Translation Process

Process



Code generation example

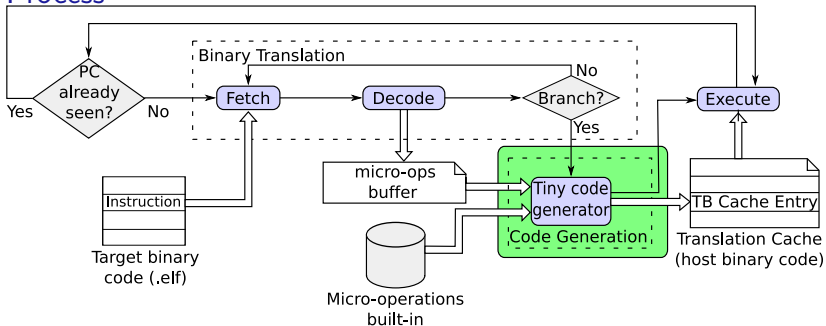
18 target_instrX

micro-op1_instrX

micro-op2_instrX

Dynamic Binary Translation Process

Process



Code generation example

18 target_instrX

micro-op1_instrX

host_instr1_micro-op1_instrX

host_instr2_micro-op1_instrX

host_instr3_micro-op1_instrX

micro-op2_instrX

host_instr1_micro-op2_instrX

Similarities with Compilation Process

Translation Example (ARM to x86)

18 and r2,r1, r5

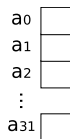
```
mov_i32 tmp0, r5
mov_i32 tmp1, r1
and_i32 tmp0, tmp0, tmp1
mov_i32 r2, tmp0
```

```
mov 0x05(ebp), %esi
mov 0x04(ebp), %ebx
and %esi, %ebx
mov %ebx, 0x08(%ebp)
```

DBT Aspects

- ▶ Target registers are in memory (array): variables
- ▶ Allocation of host registers for operations

Target Registers View



VLIW Architecture

Explicit instruction level parallelism

- ▶ Execution packets: instructions executed in parallel,
- ▶ Denoted with “||” symbol.

VLIW Example

Assembly code example

```
    add    .L1    a0, a1, a2
||  sub    .D1    a3, a2, a1
||  mpy32  .M2    b5, b3, b3

    add    .L2    b2, b3, b3
||  ldw    .D1T1  *(a0), a1
```

VLIW Architecture (2)

No stalls nor bypasses in pipelines

But some instructions take time...

- ▶ Compilers and assembly programmers have to take care,
- ▶ Need to wait some cycles before result actually available,
- ▶ Depends on the instruction latency (or delay slot).

Assembly code example (C6X)

```
mpy32  .M1    a5, a2, a3
add     .M1    a1, a3, a6 ; 1
add     .M1    a2, a3, a7 ; 2
add     .M1    a5, a3, a8 ; 3
add     .M1    a9, a3, a9 ; result of mpy32 in a3
```

VLIW Architecture (3)

Same rule applies for the branches

The delay slot of branches is 5 (C6X)

```
b .S1      0xbeef      ; branch instruction
add .L1     a1, a2, a3  ; 1
nop                ; 2
sub .L1     a2, a4, a1  ; 3
nop                ; 4
nop                ; 5
add .D1     a3, a2, a1  ; branch taken here
                        ; (add not executed)
```

VLIW Architecture (4)

The execution pipeline is never flushed

Case of the instructions in the branch delay slot

- ▶ They are executed normally,
- ▶ If they don't finish before the jump, they continue after it.

Example

```
b      .S1 label1      ; branch instruction
nop    2              ; no operation for 2 cycles
mpy32  .M1 a1, a2, a3 ; delay slot: 3
nop    2              ; no operation for 2 cycles
add    .D1 a3, a2, a1 ; branch taken here
...
label1: add .L1 a3, a5, a6 ; old value of a3 used
add    .L1 a3, a7, a8 ; new value of a3 used
```

Plan

- **Introduction**
 - Context & motivations
- **Technology**
 - DBT
 - VLIW architectures
- **VLIW DBT**
 - Algorithm
 - Implementation
- **Conclusion**

Naive Scalar DBT of VLIW code

Operations

Translate all instructions into QEMU IR instructions.

VLIW Code example

```
add   .L1 a0, a1, a2 ; 1st cycle
|| sub .D1 a1, a4, a6
|| mpy32 .M1 a2, a1, a3 ; lat 3

sub   .D1 a6, a3, a3 ; 2nd cycle

add   .L1 a2, a3, a4 ; 3rd cycle

nop                                     ; 4th cycle

add   .L1 a2, a3, a5 ; 5th cycle
```

QEMU IR Translation

```
add_i32    a2, a0, a1
sub_i32    a6, a1, a4
mul_i32    a3, a2, a1

sub_i32    a3, a6, a3

add_i32    a4, a2, a3

add_i32    a5, a2, a3
```

The parallel execution problem

Symptom

Source value overwritten due to sequential execution

VLIW Code example

```

add   .L1 a0, a1, a2 ; 1st cycle
|| sub .D1 a1, a4, a6
|| mpy32 .M1 a2, a1, a3 ; lat 3

sub   .D1 a6, a3, a3 ; 2nd cycle

add   .L1 a2, a3, a4 ; 3rd cycle

nop                                     ; 4th cycle

add   .L1 a2, a3, a5 ; 5th cycle

```

QEMU IR Translation

```

add_i32  a2, a0, a1
sub_i32  a6, a1, a4
mul_i32  a3, a2, a1

sub_i32  a3, a6, a3

add_i32  a4, a2, a3

add_i32  a5, a2, a3

```

Replication of registers

Solution

Use a Single Register Assignment (close to compilation SSA concept)

VLIW Code example

```

add    .L1 a0, a1, a2 ; 1st cycle
|| sub  .D1 a1, a4, a6
|| mpy32 .M1 a2, a1, a3 ; lat 3

sub    .D1 a6, a3, a3 ; 2nd cycle

add    .L1 a2, a3, a4 ; 3rd cycle

nop                                ; 4th cycle

add    .L1 a2, a3, a5 ; 5th cycle

```

QEMU IR Translation

```

add_i32  a2_1, a0_0, a1_0
sub_i32  a6_1, a1_0, a4_0
mul_i32  a3_1, a2_0, a1_0

sub_i32  a3_2, a6_1, a3_1

add_i32  a4_1, a2_1, a3_2

add_i32  a5_1, a2_0, a3_2

```

The delay slot problem

Symptom

Delayed write back are taken into account too early

VLIW Code example

```

add    .L1 a0, a1, a2 ; 1st cycle
|| sub  .D1 a1, a4, a6
|| mpy32 .M1 a2, a1, a3 ; lat 3

sub    .D1 a6, a3, a3 ; 2nd cycle

add    .L1 a2, a3, a4 ; 3rd cycle

nop                                ; 4th cycle

add    .L1 a2, a3, a5 ; 5th cycle

```

QEMU IR Translation

```

add_i32    a2_1, a0_0, a1_0
sub_i32    a6_1, a1_0, a4_0
mul_i32    a3_1, a2_0, a1_0

sub_i32    a3_2, a6_1, a3_1

add_i32    a4_1, a2_1, a3_2

add_i32    a5_1, a2_0, a3_2

```

Delay slot handling

Solution

Keep previous instance of the register until it is updated (delayed).

VLIW Code example

```

add   .L1 a0, a1, a2 ; 1st cycle
|| sub .D1 a1, a4, a6
|| mpy32 .M1 a2, a1, a3 ; lat 3

sub   .D1 a6, a3, a3 ; 2nd cycle

add   .L1 a2, a3, a4 ; 3rd cycle

nop                                     ; 4th cycle

add   .L1 a2, a3, a5 ; 5th cycle

```

QEMU IR Translation

```

add_i32    a2_1, a0_0, a1_0
sub_i32    a6_1, a1_0, a4_0
mul_i32    a3_1, a2_0, a1_0

sub_i32    a3_2, a6_1, a3_0

add_i32    a4_1, a2_1, a3_2

add_i32    a5_1, a2_0, a3_1

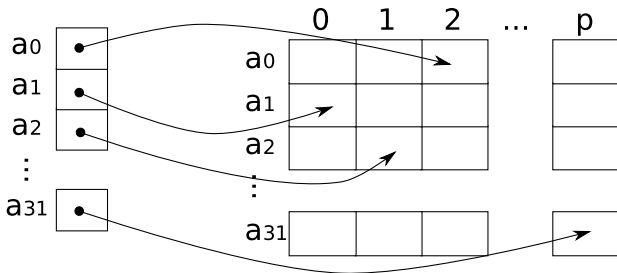
```

Implementation of the solution

Don't allocate/free registers all the time

- ▶ The simulator contains a pool of registers (p replicates),
- ▶ Allocation of replicated registers when needed,
- ▶ Well sized pool, always enough registers.

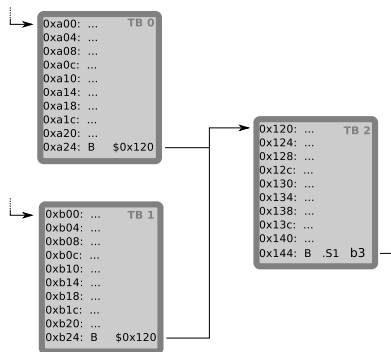
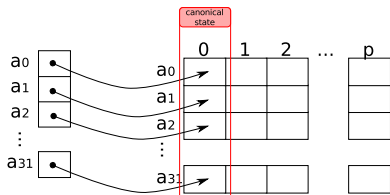
Target Registers View



Independence between translation blocks

Canonical state between TBs

- ▶ To ensure TB independence,
- ▶ Need to leave a TB in the canonical state,
- ▶ The next TB starts in the canonical state.



More difficulties

Conditional execution

- ▶ Nop()-ified if condition not verified.
- ▶ Register replication problem
- ▶ Simplified version of SSA ϕ -function

Branch delay slots

- ▶ Instructions after branch instruction have to be executed
- ▶ Split branch handling and forward place the end of Translation block handling

Branch in branch delay slot

Branch can be hidden in branch delay slots

Plan

- **Introduction**
 - Context & motivations
- **Technology**
 - DBT
 - VLIW architectures
- **VLIW DBT**
 - Algorithm
 - Implementation
- **Conclusion**

Conclusion

Proposition

- ▶ Dynamic Binary Translation of VLIW architecture on Scalar
- ▶ **Mathematically proven as correct**

Promising results

- ▶ Significant simulation speed-up
- ▶ Optimizations are still possible

Thank you!

Do you have any questions?