TERMA
ALLIES IN INNOVATION

T-EMU 2.0: The Next Generation LLVM Based Micro-Processor Emulator

Dr. Mattias Holm <maho@terma.com>

- **Outline**
  - **Emulation Primer**
  - **T-EMU 2.0**
    - **History**
    - **Architectural Overview**
    - **TableGen and LLVM use**
  - **Future Directions**
  - **Final remarks of LLVM use**

Emulation

- **Instruction level simulation of a CPU**
- **Executes the emulated *target's* instruction set in a *virtual machine* running on a *host*.**
- **Simulates memory and MMU**
- **Simulates peripheral devices and/or provides a way to integrate external devices**
- **Performance measured in MIPS (millions of emulated instructions per second)**

Emulation: Parts of an Emulator

- **Instruction decoder**
  - **One for the interpreter**
  - **One for the binary translator**
  - **One for the assembler living down the lane...**
- **Instruction semantics**
  - **One routine per instruction**
    - **May be in variants (e.g. arithmetic instructions with %g0 as destination)**
  - **Binary translator and interpreter need different types of instruction descriptions...**
    - **Write two... or rather not.**
    - **Write one and #ifdef yourself around issues**
    - **Write one and transform it to the relevant format**

Emulation: Interpretation

- **Decode dispatch**
  - **Main loop:**
    - `instr = fetch(mem, cpu->pc)    ;; Fetch instruction`
    - `impl = decode(instr)           ;; Decode instruction`
    - `impl(cpu, mem)                 ;; Execute instruction, can be indirect branch to label....`
- **Threaded Interpreter**
  - **Instructions embed the main loop, i.e. threads it along the instructions**
    - **Avoids the return to main loop instruction**

Emulation: Binary Translation

- **Roughly the same as a JITter**
  - **Similar optimisations apply**
- **Translates blocks of target code to native host code on the fly**
- **Can combine with interpreters**
  - **Common approach in JavaScript engines**
- **Basic block is often related to target code, not host code...**
- **Basic block chaining embeds the emulation loop (similar to interpreter threading)**
- **Implementation**
  - **C function emitting code directly (no optimisations except simple ones (e.g. proper instruction selection))**
    - **Very fast at code generation time**
  - **Data driven: emulator intermediate used to emit machine code after transformations (e.g. LLVM IR)**

Emulation

- **Common Implementation Languages**
  - **Assembler**
    - **Can fine tune**
    - **Not portable**
  - **C**
    - **Usually not fast enough for interpretation (except when threading code...)**
    - **Can implement dynamic code generator reasonably efficiently**
  - **Custom languages / DSLs**
    - **Portable (depending on DSL compiler)**
    - **High performance**
    - **Easy to maintain but may need significant resources for in-house maintenance.**
- **T-EMU 2 use the LLVM toolchain**
  - **TableGen for instruction decoders**
  - **LLVM Assembler for instruction semantics (embedded in TableGen files)**

State of the Art

- **Binary translators**
  - **OVPSim**
  - **Windriver Simics (~350 MIPS)**
  - **QEMU (partially GPL → no use in certain industries)**
- **Interpretation (SPARC emulators)**
  - **TSIM (~60 MIPS)**
  - **ESOC Emulator (65 MIPS no MMU, 25 MIPS with MMU)**
  - **T-EMU 2.0...**
- **Others**
  - **Countless of game console emulators etc**

T-EMU 2.0: The Terma Emulator

- **T-EMU 1:**
  - **Derivation of ESOC Emulator Suite 1.11**
  - **Formed the baseline for the work on ESOC Emulator Suite 2.0**
  - **Written in EMMA: The Extensible Meta-Macro Assembler (embedded assembler, using Ada as host language)**
  - **Emulates**
    - **MIL-STD-1750A/B**
    - **SPARCv8 (ERC32, LEON2, LEON3)**
- **T-EMU 2:**
  - **Complete rewrite**
  - **Using modern C++11 and LLVM**
  - **LLVM compiler tools are used extensively**
  - **Interpreted, but ready to upgrade with binary translation capabilities**
  - **Significant work spent on defining a device modelling APIs**
    - **Can easily be wrapped for scripting languages (e.g. prototype your device model in Python) or SMP2 (an ESA standard for simulation models)**
  - **Can emulate multi-core processors**
  - **Emulates SPARCv8 (ERC32, LEON2, LEON3, LEON4)**

T-EMU 2.0: The Terma Emulator

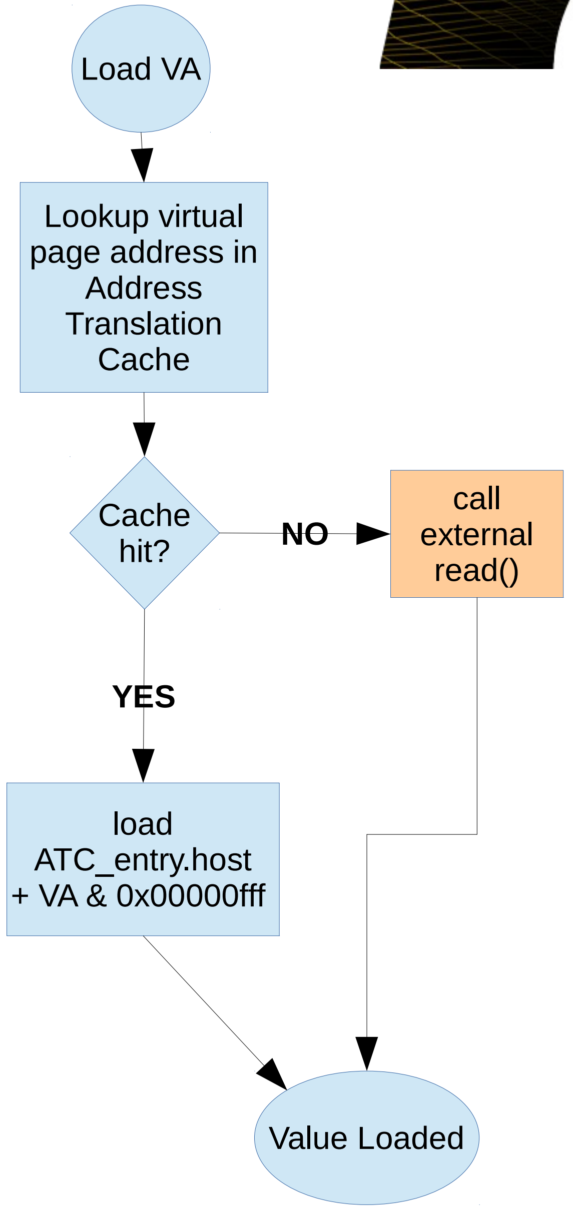
- **Library based design**
  - **Easy to integrate in simulators**
  - **Public stable API is C (i.e. can integrate with just about anything).**
- **Command Line Interface**
  - **Assisting with emulator and model development and integration**
  - **Embedded / on-board software development (e.g. unit tests)**

T-EMU 2.0: Architecture and Internals

- **Emulator Cores:**
  - **Written in TableGen and LLVM assembler**
  - **(Operational) decode-dispatch cores transformed to threaded code automatically using custom LLVM transformation passes.**
  - **TableGen data combines: instruction decoders, instruction semantics and assembler syntax in a transformable format**
  - **Multi-core support**
- **Emulator Shell**
  - **Implemented using the T-EMU 2.0 object system APIs**
  - **Integrates auto-generated assemblers and disassemblers generated from TableGen data.**
  - **High level interfaces**
    - **Interrupt interface, memory interface, etc**

T-EMU 2.0: Memory Emulation

- **Each processor has a memory space attached to it:**
  - **Memory space decodes addresses**
- **N-level page table for identifying memory mapped objects**
  - **memory**
  - **devices**
- **Unified interface for memory and devices:**
  - **Memory Access Interface**
  - **Zero-overhead for MMU due to address translation cache**
- **Memory attributes**
  - **breakpoint, watchpoint read + write, upset, faulty, user 1,2,3**


Load VA
Lookup virtual page address in Address Translation Cache
Cache hit?
NO
call external read()
YES
load ATC_entry.host + VA & 0x00000fff
Value Loaded

T-EMU 2.0: Device Models

MMIO Models Implement the MemAccessIface:

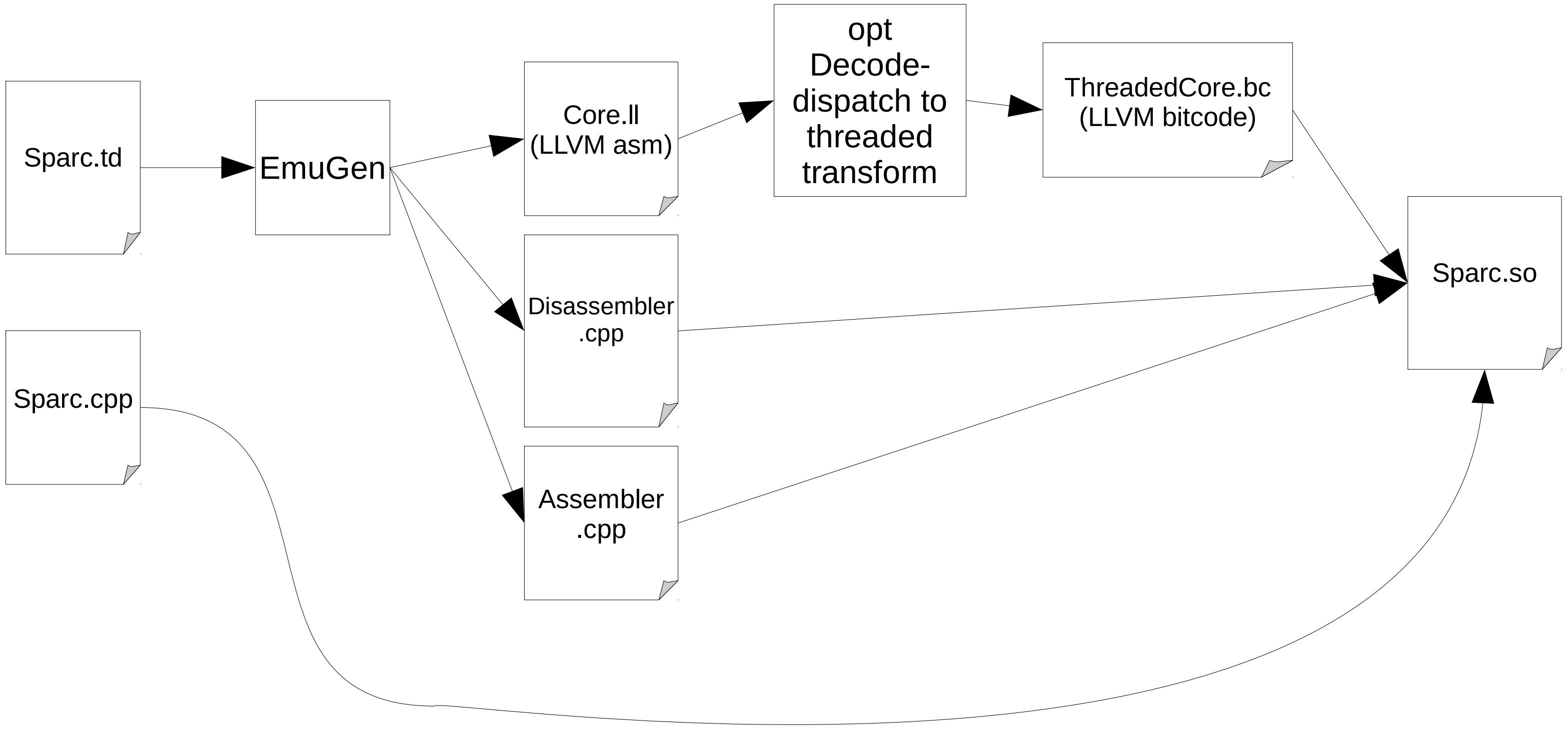
```
typedef struct temu_MemAccessIface {
  void (*fetch)(void *Obj, temu_MemTransaction *Mt);
  void (*read)(void *Obj, temu_MemTransaction *Mt);
  void (*write)(void *Obj, temu_MemTransaction *Mt);
} temu_MemAccessIface;
```

The functions take a potiner to a MemTransaction object (which is constructed by the core):

```
typedef struct temu_MemTransaction {
  uint64_t Va; // Virtual addr
  uint64_t Pa; // Physical addr
  uint64_t Value; // Out or in value
  uint8_t Size; // Log size of access

  uint64_t Offset; // Pa – Dev Start
  void *Initiator; // CPU pointer
  void *Page; // Out (for ATC)
  uint64_t Cycles; // Out (cost of op)
} temu_MemTransaction;
```

T-EMU 2.0: Compilation Pipeline



TableGen Based Capabilities

- **Generate multiple instruction decoders**
  - **Switch based (C or LLVM ASM / IR)**
  - **Table based (nested tables or single table)**
  - **Can quickly pick the best one for the task and experiment**
    - **Assemblers use switch based decoders**
    - **Interpreter use single table decoder**
- **Generates decode-dispatch emulator core in LLVM assembler**
- **Generates assembler and disassembler from instruction descriptions.**
- **Simplified maintenance due to code block concatenation and multi-classes used to e.g. provide single definition for both reg-reg and reg-imm operations.**

LLVM Transformations

- **Decode dispatch core has one function per instruction (it is operational using an emulator loop implemented in C).**
  - **Decode table identifies functions**
- **LLVM pass creates a single "emulate" function**
  - **One label per instruction**
  - **One call to semantics for the instruction**
  - **Fetch, decode and indirect branch after call**
  - **Semantics are inlined into the single emulate function**
  - **Decode table transformed to an indirect branch target table**
- **Emulator intrinsics:**
  - **All state accesses and modifications done through emulator intrinsics**
  - **E.g. call @emu.getReg(cpu_t *cpu, i5 %rs1)**
  - **We can easily change the way we access registers (different alternatives for emulating SPARC register windows and similar) e.g:**
    - **Indirect access through pointer array (nice in an interpreter)**
    - **First and last window synchronisation on save, restore and %psr updates (nice in a binary translator)**

T-EMU 2.0: TableGen CPU Descriptions

```
multiclass ri_inst_alu<bits<2> op, bits<6> op3, string asm, code sem> {
  def rr : fmt3_1<op, op3> {
    let AsmStr = asm # “ {rs1:gpr}, {rs2:gpr}, {rd:gpr}”;
    let Semantics = [{
      %r1 = call i32 @emu.getReg(%cpu_t* %cpu, i5 %rs1)
      %r2 = call i32 @emu.getReg(%cpu_t* %cpu, i5 %rs2)
    }] # sem # [{
      call void @emu.setReg(%cpu_t* %cpu, i5 %rd, i32 %res)
    }]
  }
  def ri : fmt3_2<op, op3> {
    let AsmStr = asm # “ {rs1:gpr}, {simm13}, {rd:gpr}”;
    ...
  }
}
defm add : ri_inst_alu <0b10, 0b1010101, “add”, [{
    %res = add i32 %r1, %r2
}]>;
```

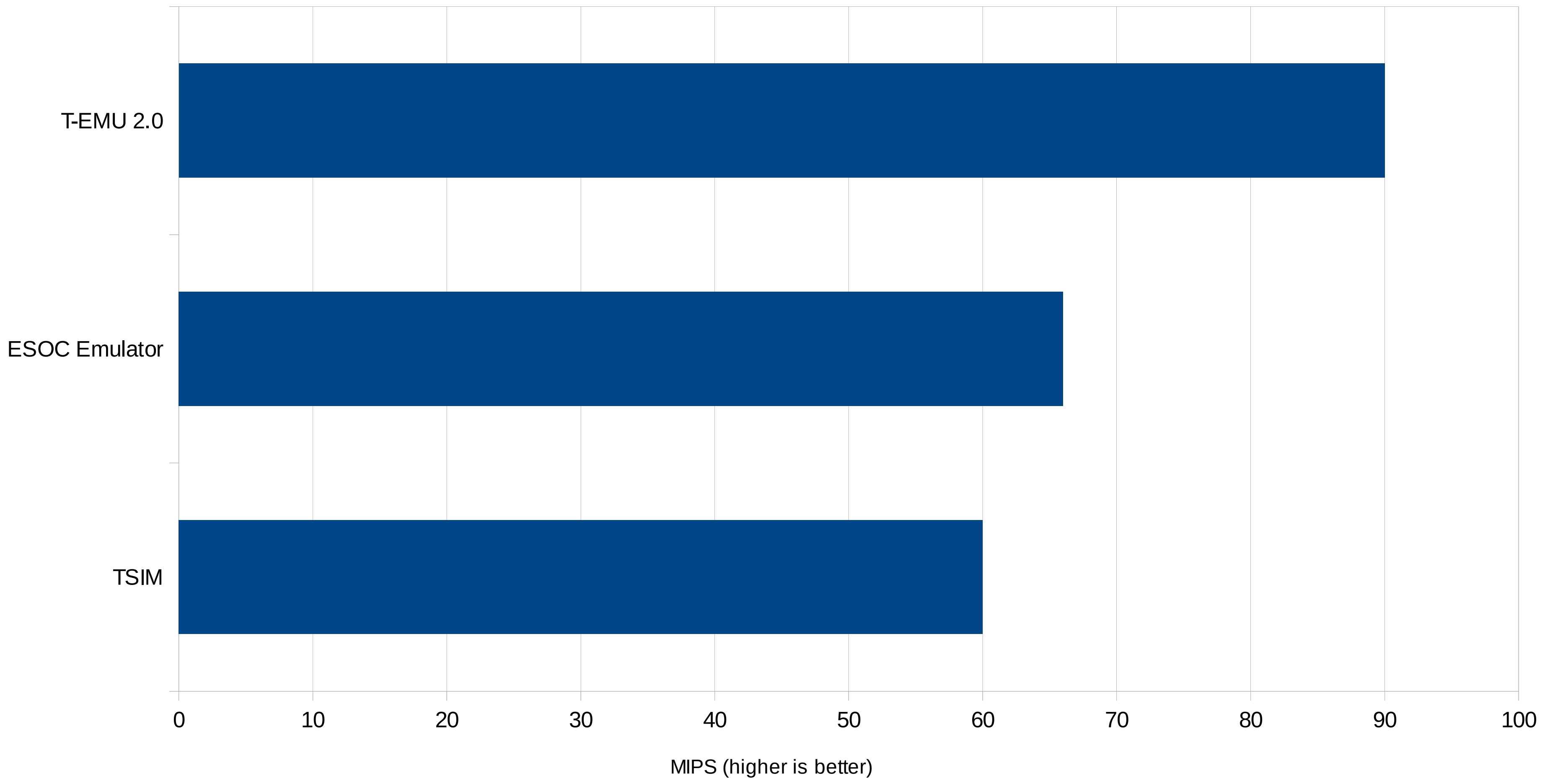
LLVM Transformation

```
def void @add_rr(%cpu_t* %cpu, i32 %inst) {
unpack:
  %rs1 = call i5 @emu.unpack.i5(i32 %inst, i32 14)
  %rs2 = call i5 @emu.unpack.i5(i32 %inst, i32 0)
  %rd = call i5 @emu.unpack.i5(i32 %inst, i32 25)
  br label %semantics
semantics:
  %r1 = call i32 @emu.getReg(%cpu_t* %cpu, i5 %rs1)
  %r2 = call i32 @emu.getReg(%cpu_t* %cpu, i5 %rs2)
  %res = add i32 %r1, %r2
  call void @emu.setReg(%cpu_t* %cpu, i5 %rd, i32 %res)
  ret void
}
```

LLVM Transformations

```
;; Note: grossly simplified (no step updates,
;; missing hundereds of instructions etc)
def @emulate(%cpu_t* %cpu, i64 %steps) {
entry:
  %pc = call i32 @emu.getPc(%cpu_t* %cpu)
  %inst = call i32 @emu.fetch(%cpu_t* %cpu, i32 %pc)
  %dest = call i8* @emu.decode(i32 %inst)
  indirectbr %dest
add_rr:
  %inst0 = phi i32 entry %inst...
  call void @add_rr(%cpu_t* %cpu, %inst0)
  %pc0 = call i32 @emu.normalIncPc(%cpu_t* %cpu)
  %inst1 = call i32 @emu.fetch(%cpu_t* %cpu, i32 %pc0)
  %dest1 = call i8* @emu.decode(i32 %inst0)
  indirectbr %dest1
}
```

Current Interpreted Emulator Performance



- 3.5 GHz x86-64
- ESOC Emu numbers are for the stock ESOC Emu configuration without MMU. Current ongoing optimisation work.
- TSIM numbers from http://www.gaisler.com/
- Anything above 50 MIPS is high performance for an interpreted emulator

T-EMU: General Future Directions

- **Binary translation (>300 MIPS)**
- **Additional architectures (ARM, PowerPC, MIPS etc)**
- **Direct support for more ways for device modelling:**
  - **SMP2**
  - **System-C**
  - **VHDL**
- **Bigger model library:**
  - **Provide models for all common spacecraft processors and peripherals**

T-EMU: Directions with LLVM

- **Binary translation**
  - **Instruction semantics already binary translation friendly**
  - **Binary translation specific decoders can be generated**
  - **LLVM format can be transformed to:**
    - **Direct code emitting functions:**
      - **Code emission will be fast**
    - **Pre-generated instructions implementations for memcopy-based code emission:**
      - **Code must obviously be PIC**
      - **Code emission will be very fast**
      - **Stiching of code blocks is tricky**
    - **LLVM or IR templates for LLVM based JIT**
      - **Code emission will be “slow”**
      - **Can use optimisations (emitted code will be fast)**
      - **Likely slower than we want in the standard case**
      - **MC-JIT can probably be used.**
      - **One LLVM function per extended basic blocks (e.g. the SCCs formed by standard emulated basic blocks with indirect and absolute branches (in the target code) as terminators).**
    - **Note: we probably want a multi-tier JITter (see e.g. the WebKit JavaScript engine).**

Bumps on the Road

- **TableGen is not really well documented:**
  - **Several semantic issues are best figured out by using it**
  - **The existing documentation and a LLVM dev meeting video tutorial helps**
  - **Read the source…**
- **Writing in LLVM assembler:**
  - **Hard to debug**
  - **No way to include files (M4 or CPP to the rescue)**
  - **No way to define named constants (M4 or CPP to the rescue again)**
  - **It wasn't really intended for this, so we are not complaining… :)**

LLVM and T-EMU 2

- **LLVM is useful in an emulator for two reasons**
  - **TableGen (really powerful)**
  - **IR and the transformation passes**
- **LLVM is not just for compilers and programming language designers**
- **Enabled the rapid development of a new high-performance and *hackable* emulator in a short time**
- **Ensures we can extend the new emulator with binary translation without rewriting the instruction definitions.**
- **Domain specific code transformations are very, very powerful. LLVM transformation toolchain is not just for standard compiler optimisations**
- **Unique use of LLVM (we think)**
  - **Related work exists (e.g. LLVM as optimisation of QEMU)**

Questions?

http://t-emu.terma.com/

PoCs:
- Technical: Dr. Mattias Holm <maho@terma.com>
- Sales: Roger M. Patrick <rmp@terma.com>

Further Reading

- **Dynamically Translating x86 to LLVM using QEMU: *Vitaly Chipounov and George Candea*, 2010**
- **Using the LLVM Compiler Infrastructure for Optimised Asynchronous Dynamic Translation in QEMU, *Andrew Jeffery*, 2009**
- **LnQ: Building High Performance Dynamic Binary Translators with Existing Compiler Backends, *Chun-Chen Hsu et.al*, 2011**