Vol.2014-EMB-34 No.4
2014/9/17

Cache Simulation for Instruction Set Simulator QEMU

Tran Van Dung*, Ittetsu Taniguchi†, and Hiroyuki Tomiyama†
*Graduate School of Science and Engineering, Ritsumeikan University, 1-1-1 Noji-higashi, Kusatsu, Shiga 525–8577 Japan.
†College of Science and Engineering, Ritsumeikan University, 1-1-1 Noji-higashi, Kusatsu, Shiga 525–8577 Japan.
E-mail: {gr0150pk@ed, i-tanigu@fc, ht@fc}.ritsumei.ac.jp

***Abstract*—In embedded system design, there is an increasing demand for modeling techniques that can provide both accurate measurements of delay and fast simulation speed. Modeling latency effects of a cache can greatly increase accuracy of the simulation and assist developers to optimize their software. Current solutions have not succeeded in balancing three important factors: speed, accuracy and usability. In this research, we created a cache simulation module inside a well-known instruction set simulator QEMU. Our implementation can simulate various cases of cache configuration and obtain every memory access. In full system simulation, speed is kept at around 73 MIPS on a personal host computer which is close to native execution of ARM Cortex-M3 (125 MIPS at 100 MHz). Compared to the widely used cache simulation tool, Valgrind, our simulator is three time faster.**

***Index Terms*—Cache simulation, memory emulation, QEMU, dynamic binary translation.**

I. Introduction

Nowadays, an important part of the computer industry involves embedded systems. Embedded systems as they occur in application domains such as automotive, aeronautical and industrial automation often have to satisfy hard real-time constraints. Timeliness of reactions is absolutely necessary and off-line guarantees have to be derived using safe methods.

Hardware architectures used in such systems now feature caches, deep pipelines, and many kinds of speculation to improve average-case performance. The speed and size are two concerns of embedded systems in the area of memory architecture design. Real-Time embedded systems often have a hard deadline to complete some instructions. In these cases, the speed of memory plays an important role in system performance.

Data within the cache are stored in cache lines. A cache line holds the contents of a contiguous block of main memory. If data requested by the processor are found in a cache line, it is called a cache hit. Otherwise, a cache miss occurs. The contents of the memory block containing the requested word are then fetched from a lower memory layer and copied into a cache line. For this purpose, another data item must typically be replaced.

Cache hits usually take one or two processor cycles, while cache misses take tens of cycles as a penalty of mishandling, so the speed of the memory hierarchy is a key factor in the system. Almost all embedded processors have on-chip instructions and data caches. From the point of view of size, it is critical for battery-operated embedded systems to reduce the amount of power usage.

There are three approaches to cache simulation: source-level simulation, off-line simulation, and on-line simulation. Source code level simulation annotates instrumentation code inside source code to trace memory accesses and simulate cache at run-time. Off-line simulation reads a memory access log generated by other tools, creates a cache model based on a configuration file, and simulates cache behavior. On-line simulation executes software via a system simulator which has a cache model inside to analyze memory accesses and to output cache miss/hit rate.

Our implementation follows the third approach because it helps to balance speed, accuracy and usability. Source-level simulation is fast but it has unavoidable problems tracing all memory accesses. Off-line simulation has difficulty evaluating big applications because memory access logs may be big. It is useful for evaluating various cache configurations for specified programs. On-line simulation, on the other hand, is convenient for evaluating many different applications. Its speed is slower than source-level simulation but its accuracy can be guaranteed.

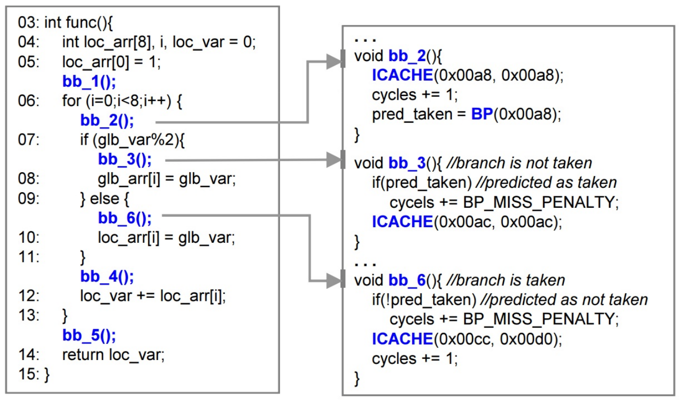
The rest of the paper is organized as follows: In section II, we explain related works of the three approaches. Section III describes background of this research in terms of dynamic binary translation and helper function. In section IV, we give a brief explanation of our methodology. Section V introduces our experiments and results. In section VI, we conclude this paper and give recommendation for the future.

II. Related works

A. Source-level simulation

A source-level model is generated by annotating timing information into application source code and allows for very fast software simulation. Figure 1 gives an example of source code and annotated code.

Zhonglei Wang and Jorg Henkel proposed a novel method to tackle two problems [1]. Firstly, target data addresses cannot be statically resolved during source code instrumentation, so accurate data cache simulation is very difficult at source level. Secondly, cache simulation brings large overhead in simulation performance and therefore cancels the gain of source level simulation. However, they still have difficulty in dealing with pointer aliasing. If a variable is accessed with a pointer that aliases it, manual analysis is needed to find out which variable



```
int func(){
    int loc_arr[8], i, loc_var = 0;
    loc_arr[0] = 1;
    bb_1();
    for (i=0;i<8;i++) {
        bb_2();
        if (glb_var%2){
            bb_3();
            glb_arr[i] = glb_var;
        } else {
            bb_6();
            loc_arr[i] = glb_var;
        }
        bb_4();
        loc_var += loc_arr[i];
    }
    bb_5();
    return loc_var;
}
```

```
...
void bb_2(){
    ICACHE(0x00a8, 0x00a8);
    cycles += 1;
    pred_taken = BP(0x00a8);
}
void bb_3(){ //branch is not taken
    if(pred_taken) //predicted as taken
        cycles += BP_MISS_PENALTY;
    ICACHE(0x00ac, 0x00ac);
}
...
void bb_6(){ //branch is taken
    if(!pred_taken) //predicted as not taken
        cycles += BP_MISS_PENALTY;
    ICACHE(0x00cc, 0x00d0);
    cycles += 1;
}
```

Fig. 1: Example of source level simulation [1]

this pointer points to.

FastVeri [2] converts software code into a virtual CPU model in SystemC. To keep cycle accuracy, FastVeri also back-annotates software code with delays from their instruction and data cache emulation. Also, it is easily connected to external SystemC models, simulators. Their approach, however, is proprietary and not easily extensible towards standard system-level design flows.

B. Off-line simulation

Wei Zang and Ann Gordon-Ross [4] created a novel solution to find a suitable cache configuration for a specified application to meet a predefined miss rate. Their simulator also can simulate multi-level cache hierarchies and achieve 41X speedup compared to the most popular trace-driven cache simulation, Dinero IV [5]. However, accuracy of this method depends on accuracy of memory access log which is not easy to generate and verify. Also, in case of big applications, size of log files may be too big to handle. For example, we utilized QEMU to record memory access during booting ARM Linux and analyzed it by Dinero IV. The size of the log file is 2.5 Gb while analyzing time is 130 seconds.

C. On-line simulation

Valgrind [7] is in essence a virtual machine using just-in-time (JIT) compilation techniques, including dynamic recompilation. Cache evaluation is one of its helpful functionalities to output a cache miss/hit report. However, its accuracy is not good because it doesn't account for cache misses arising from TLB misses, or speculative execution. Also, kernel or process activity is ignored so it is only desirable when considering a single program.

Ardavan Pedram, David Craven, and Andreas Gerstlauer [3] integrated cache simulation into a Transaction Level Modeling (TML) simulator for ARM processor. They achieved high accuracy of cache miss rate and reduced overhead of annotated code. However, the TLM simulator is much slower than our selected one, QEMU.

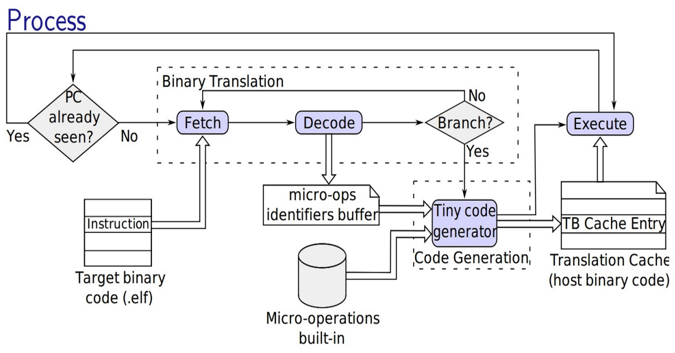


Fig. 2: Dynamic binary translation [9]

III. Background

A. Dynamic binary translation

QEMU is an open-source fast instruction-level CPU emulator [13]. It uses the target CPUs binary code to perform emulation on a host machine. QEMU is extremely flexible; owing to its portable JIT dynamic code generator, it is capable of emulating many different types of CPU targets on many different types of host machines.

Dynamic binary translation (DBT) is the key point to make simulation speed fast and reduce overhead. QEMU divides the target binary code into chunks of code called basic blocks (BBs), using branch instructions as separators. Code generation is performed on a BB basis: when the program counter of the emulated system reaches a specific BB for the first time, the entire BB is translated into equivalent block of host code called translated block (TB). The generated TB is stored in a translation cache (TC), from which it is repeatedly accessed by the host CPU for execution. Figure 2 describes this process completely.

Figure 3 explains implementation of DBT in QEMU source code. *cpu-exec()* is called to execute guest instructions. First of all, it calls *tb_find_fast()* and *tb_find_slow()* to check if guest instructions are in translation cache. If not, *gen_intermediate_code_internal()* is called to translate a basic block to intermediate code. *tcg_gen_code()* continues to translate intermediate code to host code or a TB. This TB is executed by *tcg_qemu_tb_exec()*. If the basic block is translated and stored in translation cache, *tcg_qemu_tb_exec()* executes it without translating.

The use of a TC is the reason why QEMU is so fast; the TC enables the host system to skip code generation for TBs that are already stored in it. As a result, when switching between BBs, QEMU needs to perform code generation about 1% of the time, while nearly 99% of the time it accesses the TB directly from the TC [10].

B. Memory emulation

QEMU uses a softmmu model to speed up translating guest logical addresses to host virtual addresses [12]. Its main idea is storing the offset of guest virtual address to host virtual address in a TLB table. When translating the guest virtual

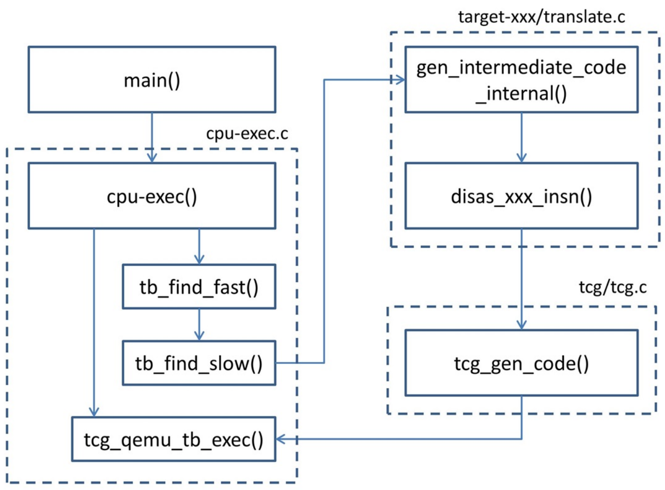


Fig. 3: Implementation of dynamic binary translation in QEMU

address to host virtual address, it will search this TLB table firstly. If there is an entry in the table, then QEMU can add this offset to guest virtual address to get the host virtual address directly. Otherwise, it needs to search the l1_phys_map table and then fills the corresponding entry to the TLB table. This TLB table idea is just like the most traditional hardware TLB.

Moreover, besides helping speed up the process of translating guest virtual address to host virtual address, this softmmu model can speed up the process of dispatching I/O emulation functions according to guest virtual address too. In this case, the idex of I/O emulation functions in io_mem_write/io_mem_read is stored in iotlb.

The softmmu emulation uses C macro to emulate template system. There are several template head files which are included in other files multiple times to generate functions that work for different sized memory and functions to access guest memory with different privileges.

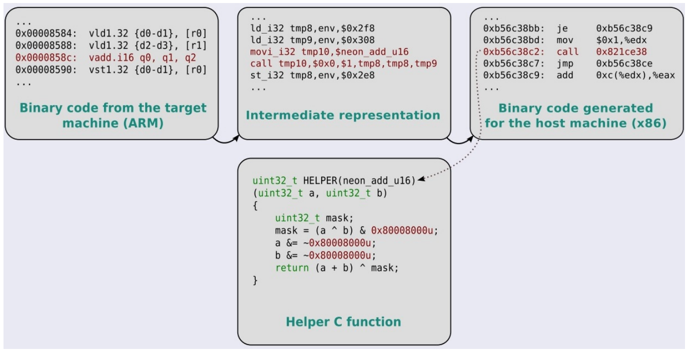
C. Helper function

Helper functions are functions in QEMU which can be called from the translation cache (TC). QEMU uses helper functions to implement uncommon but complex guest instructions, so that they do not have to be implemented entirely as large and complex blocks of code that are compiled at run-time [8]. From these helper functions, callbacks that have been registered by the user's program are called. An example is shown in Figure 4.

For each helper function f to be defined, the first thing to do is to use the macro:

```
DEF_HELPER_n(f, tr, t1, ..., tn);
```

n is the number of operands which are *t1, ..., tn*; *f* is name of the function; and *tr* is return value. This macro generates three pieces of code: (1) the prototype of the helper function helper_f , (2) the op helper function gen_helper_f to be called by DBT to generate the host code to call the helper function,



```
...
0x00008584:  vld1.32 {d0-d1}, [r0]
0x00008588:  vld1.32 {d2-d3}, [r1]
0x0000858c:  vadd.i16 q0, q1, q2
0x00008590:  vst1.32 {d0-d1}, [r0]
...
```

Binary code from the target machine (ARM)

```
...
ld_i32 tmp8,env,$0x2f8
ld_i32 tmp9,env,$0x308
movi_i32 tmp10,$neon_add_u16
call tmp10,$0x0,$1,tmp8,tmp8,tmp9
st_i32 tmp8,env,$0x2e8
...
```

Intermediate representation

```
...
0xb56c38bb:  je     0xb56c38c9
0xb56c38bd:  mov    $0x1,%edx
0xb56c38c2:  call   0x821ce38
0xb56c38c7:  jmp    0xb56c38ce
0xb56c38c9:  add    0xc(%edx),%eax
...
```

Binary code generated for the host machine (x86)

```
uint32_t HELPER(neon_add_u16)
(uint32_t a, uint32_t b)
{
    uint32_t mask;
    mask = (a ^ b) & 0x80008000u;
    a &= ~0x80008000u;
    b &= ~0x80008000u;
    return (a + b) ^ mask;
}
```

Fig. 4: An example of helper functions in QEMU [11]

and (3) the code to register the helper function at run-time for the purpose of debugging. For instance, the macro:

```
DEF_HELPER_2(neon_add_u16, void, i32, i32);
```

will generate the following code:

```
void helper_neon_add_u16 (uint32_t,
    uint32_t);

static inline void
    gen_helper_neon_add_u16(TCGv_i32 arg1,
    TCGv_i32 arg2)
{
  TCGArg args[2];
  int sizemask;
  sizemask = 0;
  args[1 - 1] = GET_TCGV_I32(arg1);
  sizemask |= 0 << 1;
  args[2 - 1] = GET_TCGV_I32(arg2);
  sizemask |= 0 << 2;
  tcg_gen_helperN(helper_neon_add_u16,0,
      sizemask,TCG_CALL_DUMMY_ARG,2, args);
}

tcg_register_helper(helper_fetch_insn, "
    neon_add_u16");
```

The helper function, *helper_neon_add_u16*, is defined as *unit32_t HELPER(neon_add_u16)(unit32_t a, unit32_t b)* in Figure 4. The op helper function is defined by *gen_helper_neon_add_u16*. The code to register the helper function for the purpose of debugging is invoked by *tcg_register_helper*. The helper function will get called by the host code generated by the op helper function defined above and executed together with the host code of each target instruction.

IV. Methodology

A. Instruction cache simulation

To simulate instruction cache, at first, every instruction address must be obtained. Because QEMU loads guest instructions to translate them into intermediate code, instruction address should be traced in this part. In detail, *disas_xxx_insn()*

(xxx is the guest architecture) generates the same thing for each instruction of a basic block. Therefore, we added code here to record instruction addresses to cache simulation function. However, if an instruction is translated, stored in translation cache, and re-executed again, the function *disas_xxx_insn()* will not be called. To tackle this problem, we called a helper function from *disas_xxx_insn()* to transfer an instruction address as below:

```
static void disas_arm_insn(CPUARMState *
    env, DisasContext *s)
{
  unsigned int cond, insn, val, op1, i,
      shift, rm, rs, rn, rd, sh;
  TCGv tmp;
  TCGv tmp2;
  TCGv tmp3;
  TCGv addr;
  TCGv_i64 tmp64;

  // Cache simulation: call icache
      simulation
  wapper_call_icache(s->pc);
  // Cache simulation: end
  insn = arm_ldl_code(env, s->pc, s->
      bswap_code);
  s->pc += 4;
  ...
}

void wapper_call_icache(uint32_t pc)
{
  TCGv tmp = tcg_temp_new_i32();
  tcg_gen_movi_i32(tmp, pc);
  gen_helper_call_icache(tmp);
  tcg_temp_free_i32(tmp);
}
```

This helper function is called every time the mentioned instruction is executed by *tcg_qemu_tb_exec()*. Therefore, our implementation can trace all addresses of executed guest instructions.

The process of instruction cache simulation is illustrated in Figure 5. At the beginning of main loop in vl.c, we modified QEMU to call cache initialization function to initialize cache model in term of size, way, size of line, replacement policy. Our implementation allows users to set these parameters by themselves in order to find the best cache configuration. Options for cache size, way, size of line are not limited while options for replacement policy are round-robin and random. When a memory access occurs, the helper function passes memory accesses to the I-cache simulation function to simulate cache behavior. When users stop QEMU, it outputs a report of cache miss rate.

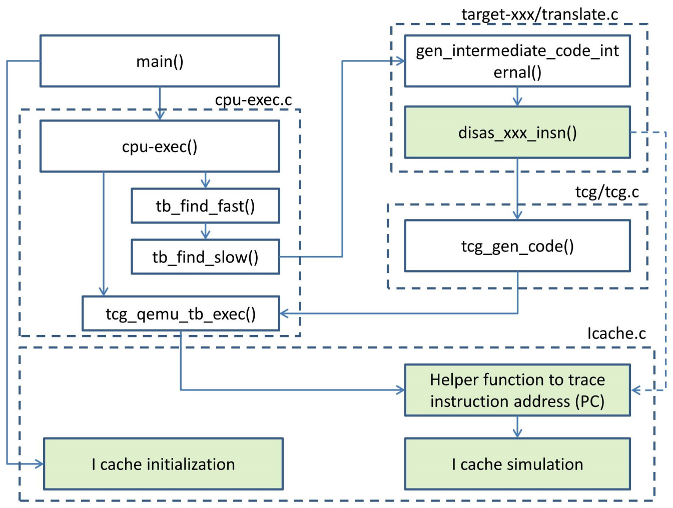


Fig. 5: Implementation of I-cache simulation

B. Data cache simulation

Function *__ldb_mmu/__ldl_mmu/__ldw_mmu* is used to translating the guest virtual address to host virtual address or dispatching guest virtual address to I/O emulation functions. For example, when fetching code from guest memory, the whole code path is as flowing:

```
cpu_exec -> tb_find_fast -> tb_find_slow
-> get_phys_addr_code
-> (if tlb not match) ldub_code(
    softmmu_header.h)
->__ldl_mmu(softmmu_template.h)
->tlb_fill -> cpu_mips_handle_mmu_faul t
-> tlb_set_page -> tlb_set_page_exec
```

We modified these functions to send every memory access to the D-cache simulation function. The description is illustrated in the Figure 6. The implementation of D-cache initialization and D-cache simulation is the same as I-caches ones.

V. Experiments

We measured the speed of our simulator by comparing it to the original QEMU. We booted ARM Linux on both simulators and recorded the booting time and BogoMIPS. BogoMips is an unscientific measurement of CPU speed made by the Linux kernel when it boots to calibrate an internal busy-loop. It is obtained by the command: $cat/proc/cpuinfo as in Figure 7. Based on BogoMIPS, our simulator's speed is decreased fivefold. The booting time shows that our simulator is three times slower than the original QEMU. Despite this speed decrease, users will not experience any inconvenience. On a personal computer, the BogoMIPS of our simulator is around 73 MIPS which is close to native execution of ARM Cortex-M3 (125 MIPS at 100 MHz). The time of booting ARM Linux on the Versatile Express simulation is around 7 seconds.

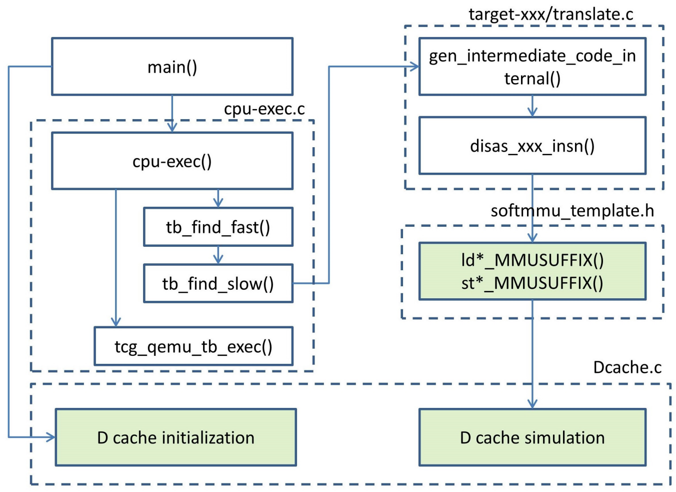
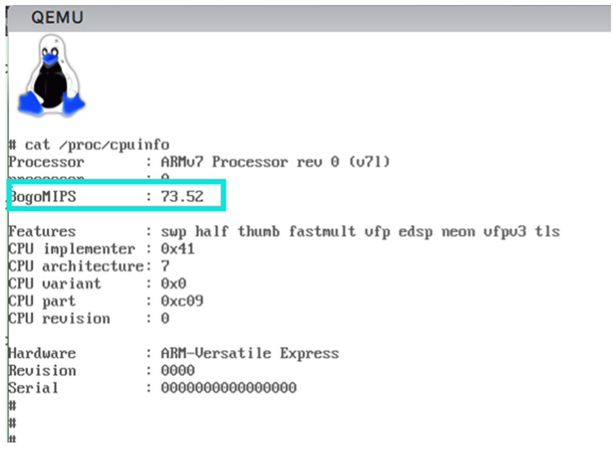


Fig. 6: Implementation of D-cache simulation



```
# cat /proc/cpuinfo
Processor       : ARMv7 Processor rev 0 (v7l)
processor       : 0
BogoMIPS        : 73.52

Features        : swp half thumb fastmult vfp edsp neon vfpv3 tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xc09
CPU revision    : 0

Hardware        : ARM-Versatile Express
Revision        : 0000
Serial          : 0000000000000000
#
#
#
```

Fig. 7: Screen shoot of booting ARM Linux on our simulator

We also compared the speed of our simulator with the widely used Cachegrind which is one of the Valgrind tools. We ported Cachegrind to the ARM platform and ran it on the original QEMU. We measured the cache miss rate of applications including matrix multiply and Jpeg encoder/decoder by both Cachegrind and our simulator. The results obtained show that our simulator is three times faster than Cachegrind.

We evaluated the impact of different cache parameters such as cache size, cache block size, and association on performance. In general, the larger cache capacity, the lower miss-rate, and the better performance [14]. We measured the cache miss rate of booting ARM Linux in many cases of cache size, from 8 Kb to 512 Kb. Our results are shown in Figure 8. Because cache miss rate of I-cache is small, we scaled it 20 times in Figure 8, 9, 10.

Size of cache block has the same impact as cache capacity does. However, bigger block size should reduce the number of blocks which leads to increase the miss rate when reading or writing the content used rarely [14]. In this case, if block size increases over a certain degree, miss rate will rise too. I-cache

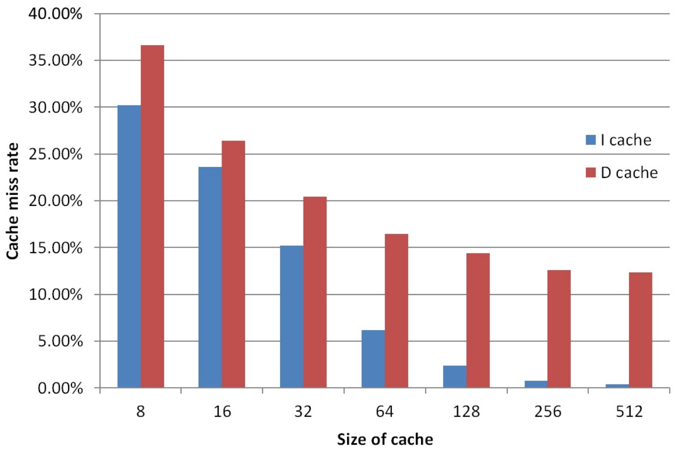


Fig. 8: Cache miss rate when cache size is changed

may have this character because it is used only for reading, not writing. In our experiments, we measured miss rate of booting ARM Linux in many cases of cache block size. Our results are shown in Figure 9. When cache block size increase above 512 Byte, cache miss rate rises too.

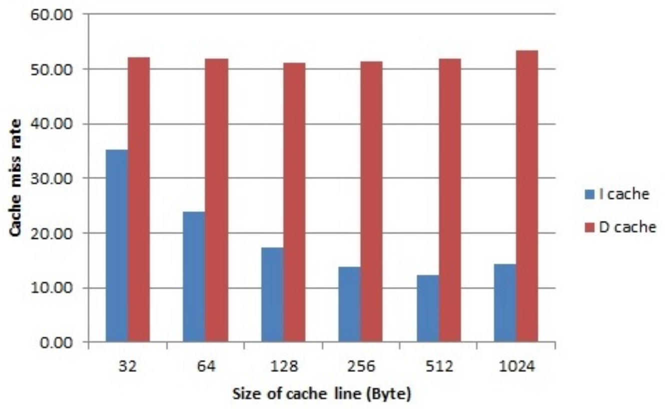


Fig. 9: Cache miss rate when cache block size is changed

Association is the number of memory blocks mirrors to cache. Raise association means that every memory block has more blocks being able to mirror. For example, if association is 2, it means that every memory block has 2 cache blocks to choose to load. In general, raising association could decrease the miss-rate [14]. In our experiments, we measured miss rate of booting ARM Linux when the association is 1, 2, and 4 . Our results are shown in Figure 10.

VI. Conclusion

In this research, we presented the integration of cache simulation into the fast and flexible instruction set simulator, QEMU. Because our methodology can get all instruction addresses of executed instructions and all memory accesses, its accuracy can be guaranteed. We implemented this methodology for ARM architecture and evaluated cache miss rate of several applications. The speed of our simulator is

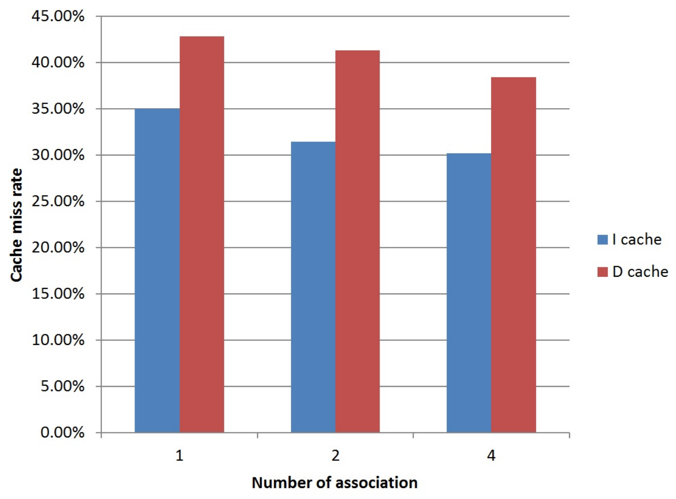


Fig. 10: Cache miss rate when number of association is changed

proven to be practical for users. For future work, we will make QEMU become cycle accurate by integrating a pipeline model with cache simulator. Also, we will extend this research to multi-core architectures to evaluate performance of caches on different cores.

References

[1] Wang Zhonglei and Henkel Jorg, *Fast and Accurate Cache Modeling in Source-Level*. Design, Automation & Test in Europe Conference & Exhibition (DATE'13), 2013.
[2] Araki, D., Ito, N., Shinsha, T., Mori, Y. *High speed hardware/software coverification with cpu model generator from software code*. Technical report, InterDesign Technologies Inc (2006)
[3] Ardavan Pedram, David Craven and Andreas Gerstlauer, *Modeling Cache Effects at the Transaction*. Analysis, Architectures and Modelling of Embedded Systems, Springer Berlin Heidelberg, pp. 89-101.
[4] Wei Zang and Gordon-Ross, *A single-pass Cache Simulation Methodology*. 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).
[5] Dinero IV Trace-Driven Uniprocessor Cache Simulator, *http://pages.cs.wisc.edu/~markhill/DineroIV*.
[6] Hui Kang, Jennifer L. Wong, *vCSIMx86: a Cache Simulation Framework for x86 Virtualization Hosts*. Stony Brook University
[7] Valgrind User Manual, *http://valgrind.org/docs/manual/cg-manual.html*.
[8] Tse-Chen Yeh, Zin-Yuan Lin and Ming-Chao Chiang, *A Novel Technique for Making QEMU an Instruction Set Simulator for Co-simulation with SystemC*. Proceedings of the International MultiConference of Engineers and Computer Scientists 2011.
[9] Marius Gligor, Nicolas Fournel and Frdric Ptrot, *Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation*. Proceeding CODES+ISSS '09 Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, pp. 71-80.
[10] David Thach, Yutaka Tamiya, Shinya Kuwamura and Atsushi Ike, *Fast Cycle Estimation Methodology for Instruction-Level Emulator*. Design, Automation & Test in Europe Conference & Exhibition (DATE'12), 2012.
[11] Luc Michel, Nicolas Fournel and Frdric Ptrot, *QEmu TCG Enhancements for Speeding-up the Emulation of SIMD instructions*. *http://adt.cs.upb.de/quf/quf11/quf2011_12.pdf*
[12] QEMU Internal, *http://vm-kernel.org/blog*.
[13] Fabrice Bellard, *QEMU, a Fast and Portable Dynamic Translator*. Proceedings of USENIX Annual Technical Conference, June 2005.
[14] MA Hai-feng, YAO Nian-min, FAN Hong-bo, *Cache Performance Simulation and Analysis under SimpleScalar Platform*. International Conference on New Trends in Information and Service Science, 2009.