Implementing Mips Cache Model in Qemu

Concise Table of Contents

Table of Contents

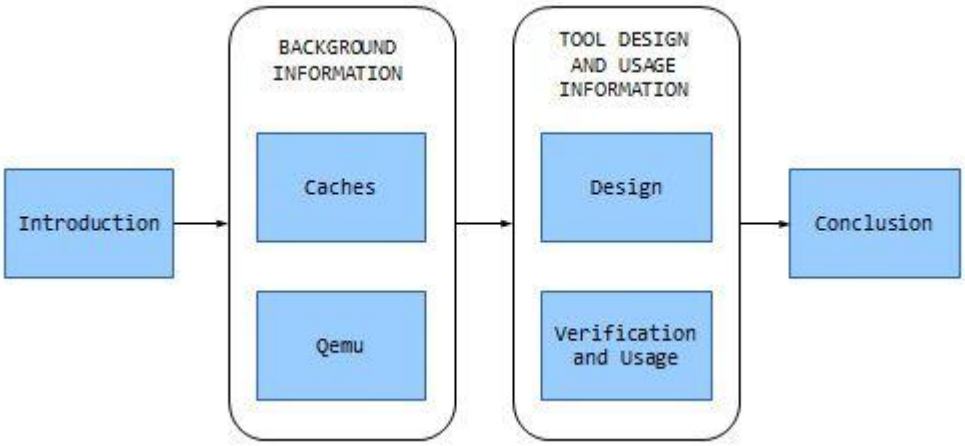
Introduction

Caches are a critical part of today's microprocessors. The purpose of this document is to describe features, design, usage, and possible future extensions of a tool developed to model cache behavior in Mips microprocessors.

The tool itself is based on a widely-used microprocessor and system emulator called Qemu. Qemu is modified so that it can be configured to offer insight into Mips microprocessor cache behavior during a scenario of choice. This means that cache-related bottlenecks in an application can be pinpointed by using this tool, and then attainable application improvements tested. New cache designs also can be evaluated by this tool.

This document is organized as follows: First several sections of this document discuss main relevant technical background facts, like basics of cache solutions in contemporary microprocessors, metrics used to evaluate cache operation, and internal organization and extensibility of Qemu. Subsequent sections describe design and implementation of the tool itself, its usage, techniques used for its verification, and also its limitations. And, finally, the last section hints some of possibilities for further development.

Document organization is also described in following picture:



Caches in Contemporary Microprocessors

Generally, a *cache* is a certain amount of storage for holding a limited number of copies of data items that are more permanently stored in some other place that is harder/slower to access than the cache itself.

Computer systems use many kinds of cache: web caches, network caches, database caches, disk drive caches, microprocessor caches. This document and the tool, however, deal only with cache used in microprocessors. Following sections explain basic principles of operation of caches. They also define vocabulary that is necessary for understanding other sections of the document.

Limits of Microprocessor Speed

Transistor density and performance of microprocessor cores improved over time in last decades in accordance with Moore Low. However, another significant phenomenon emerged: the rate of speed increase of memory (both internal and external) is much less than the rate of speed increase of microprocessor cores. Since computer systems inevitably need to deal with data from external storage, this affects system performance significantly. This problem is often called “memory wall”. It could be said that caches are designed solely in order to alleviate the problem of “memory wall”.

Locality Principle

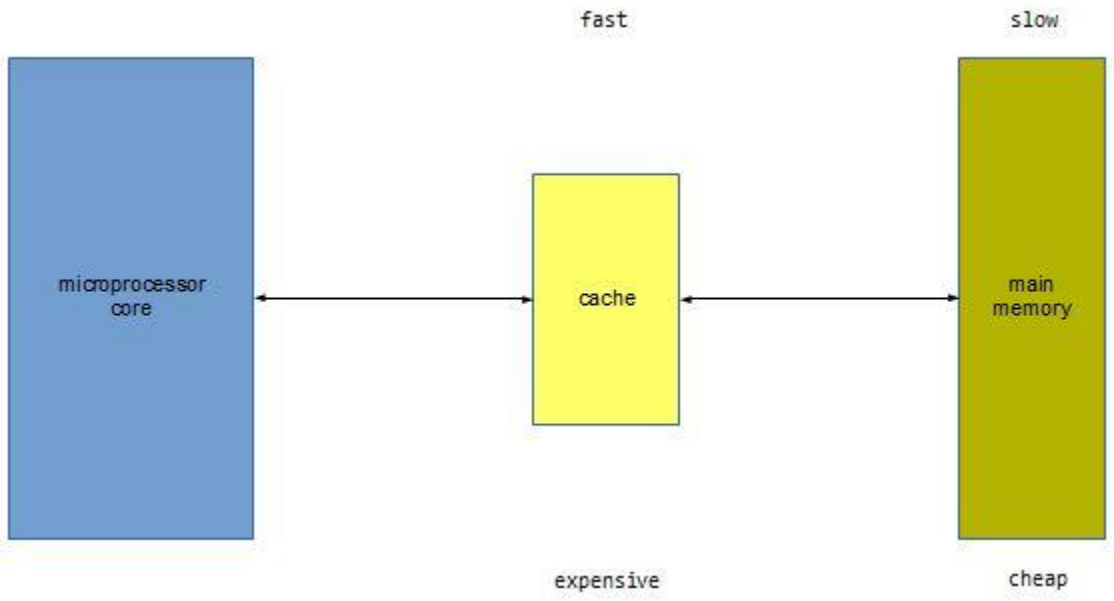
It is well established that execution of software complies with the principle of locality, in particular temporal locality and spatial locality.

Principle of temporal locality states that if certain data is needed during execution, it is likely that the same data will be needed at some point of time in near future.

Principle of spatial locality states that if certain data is needed during execution, it is likely that the data that is placed close to it will also be needed at some point of time in near future.

Basic Idea Underlining Cache

Basic idea of cache is that any application most of the time accesses relatively small subset of its address space during its execution. On top of that, some of such memory subsets are accessed repeatedly. Therefore, if a fast memory – cache – is placed between the main memory and the core, the data access can be organized so that it is performed most of the time from cache, therefore significantly improving overall performance of the application.



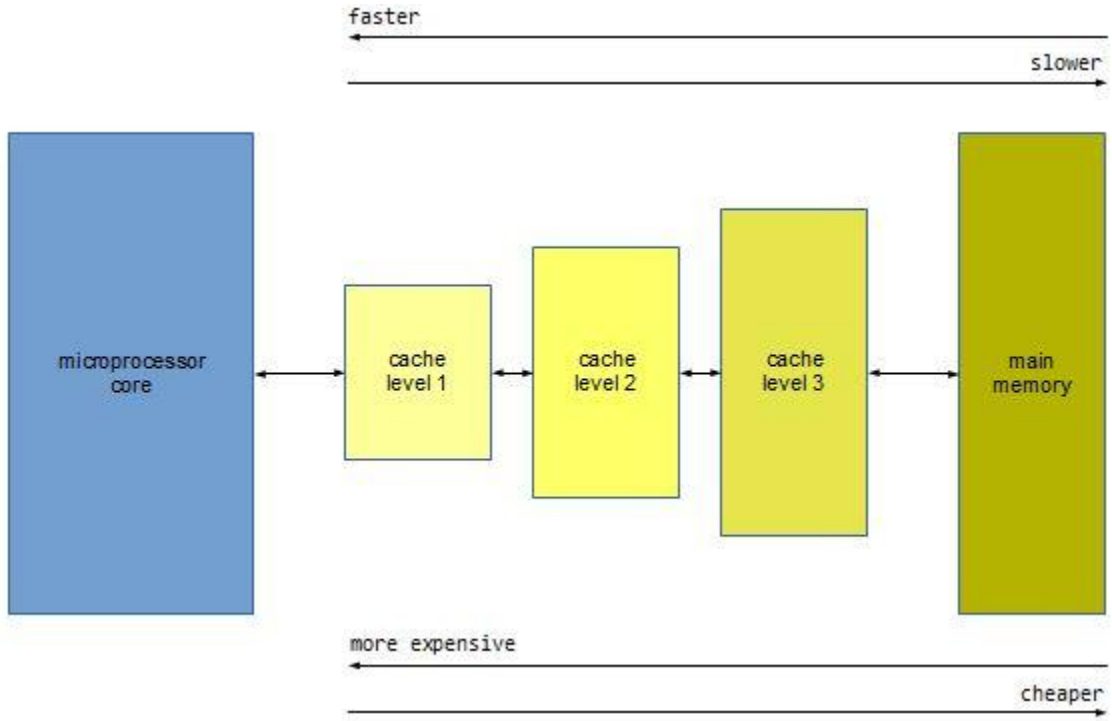
Some basic terms related to this method of storing data in caches are defined below.

Hit occurs if data appears in the cache on access attempt. This will result in data transfer at maximum speed. *Hit time* is time to access data if found in the cache. *Hit rate* is a ratio of hits and all memory accesses.

Miss occurs if data is not in the cache on access attempt. This will result in extra delay, since data must be accessed in slower memory outside of cache. *Miss penalty* is time to access data if not found in the cache. *Miss rate* is a ratio of misses and all memory accesses.

Hierarchy

The same logic can be applied further for data access between core and cache, and therefore it makes sense to organize cache in several layers, as pictured in the following figure.



This organization is called *multi-level cache*.

The fastest cache level is always the closest to the processor. Access time for a cache level is, as a rule, inversely proportional to its cost, and, for that reason, in all practical implementations, the fastest and closest to core section is the smallest, and each subsequent level is larger, slower, and cheaper.

Caches that are closer to the processor core are referred as *lower-level caches* (i.e. the first-level cache is the *lowest-level cache*). *Higher-level caches* and *highest-level cache* are defined in the same fashion. Highest-level cache is also called *last-level cache (LLC)*.

Hit rate, hit time, miss rate, and miss penalty (mentioned in previous section) are defined individually for each cache level, and also for the cache system as a whole.

Hit time for the lowest level-cache is typically one or two machine cycles. Miss penalty for the highest level-cache is nowadays typically several hundred machine cycles.

Inclusiveness

Multi-level caches use different approaches regarding the question of whether the data stored in lower-level caches must also be present in higher-level caches.

In *inclusive cache hierarchies*, every level of the cache hierarchy is a subset of the next higher level. This means that data is often placed in more than one place in the whole cache system. Although such organization obviously decreases effective capacity of the cache, it has some advantages in multi-core systems.

In *exclusive cache hierarchies*, on the other hand, data is placed in at most one level of the cache hierarchy. This maximizes cache capacity.

In *non-inclusive cache hierarchies*, data in lower-level caches may be also in higher-level caches but is not required to.

Private and Shared Access

In multicore processors, each core usually has its own lowest-level cache, while other levels are common to all cores. The rationale for such organization is as follows: lowest level cache must be very close to the core so that hit time remains one or several machine cycles; higher-level caches are shared to increase their in scenarios where only some of the cores do memory-intensive tasks.

Levels that are accessed by only one core are called *private caches*, while others are called *shared caches*.

Cache Lines and Memory Blocks

From the point of view of programs at machine instruction level, contemporary microprocessors access data in the chunks of words, which are 32-bit or 64-bit structures. However, as a rule, cache data is organized in larger chunks, called *cache lines*. When data is transferred between cache and main memory, or between cache levels, it is always done with granularity of cache lines. It usually doesn't impose significant performance penalty in comparison to transferring data with smaller granularity. It is also important that such organization also allows exploitation of spatial locality.

Each cache line is intended to contain one contiguous portion of main memory, and the start address of such portions is multiple of cache line size. Such portions of main memory are called *memory blocks*.

Data, Tag, and Status Arrays

In order to access data in the cache, a mechanism must exist for keeping track of main memory address of each cache line. This means that cache must contain information on location of data residing in it, on top of the data itself. Such information is stored in *tag array*, and data is said to be stored in *data array*. Additionally, most of the cache implementations for various reasons need several flags for each line – the most frequent example of such flags are valid flag and dirty flag. Area where such flags are stored for each cache line is called *status array*.

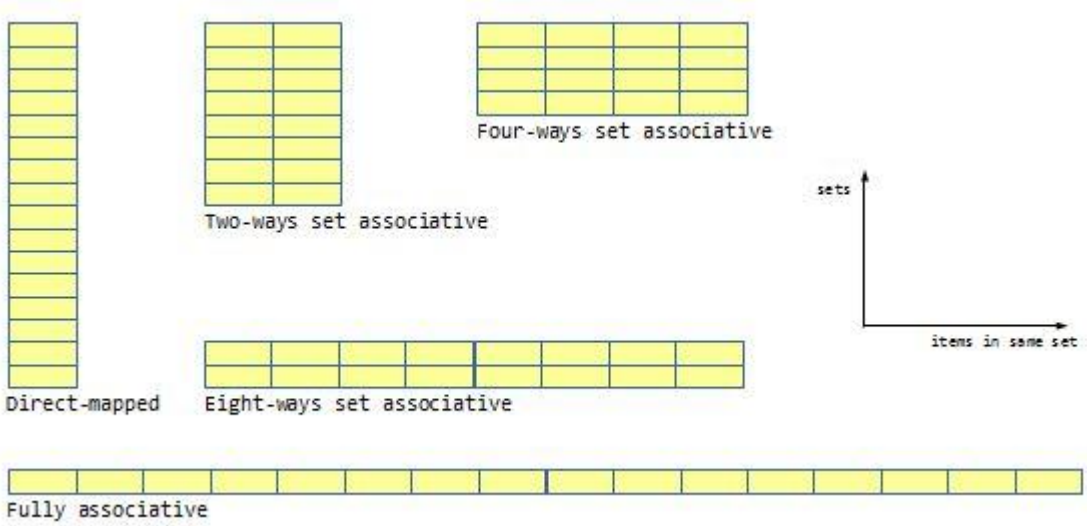
Cache Associativity

It is of course possible to implement a cache where each cache line can hold a copy of any memory block. This is called a *fully associative cache*. Although such organization is the simplest at high logical levels, it has some implementation drawbacks. To access a cache line the processor core has to compare the tags of each and every cache line with the tag of the requested address. The tag would be comprised of the entire part of the address which is not the offset into the cache line.

In *direct-mapped cache*, each memory block can be stored in only one cache line. The mapping is usually determined by address of memory block modulo cache line size (modulo is here mathematical operation "remainder of division a/b"). This organization allows some simplification in implementation over fully associative cache. However, this also means that memory blocks can compete for their place in the cache in the same cache line.

Set-associative cache could be viewed as something between fully-associative and direct-mapped caches. A number of cache lines are organized into *sets*. A memory block can be placed only in one set (determined by memory block address), but it can be placed in any of cache lines in this set. The number of cache lines in a set is also called number of *ways*. A way can be understood as all cache lines from all sets that are in particular position with respect to the set that contain particular cache line.

The following picture illustrates cache organization with respect to associativity.



Write Policies

For a case when a data need to be written to memory, the system doesn't need to write the data to the main memory instantaneously. So, there are two main approaches, or policies, to this scenario:

- *write-through policy*;

- *write-back policy*.

For write-through caches, write is done synchronously both to the cache and to the main memory. The write-through cache is actually the simplest way to implement cache coherency.

The write-back policy is more sophisticated. Here the processor does not immediately write the modified cache line back to main memory. Instead, the data is written to the cache line, and the cache line is only marked as dirty. When the cache line is dropped from the cache at some point in the future the dirty bit will instruct the processor to write the data back at that time instead of just discarding the content.

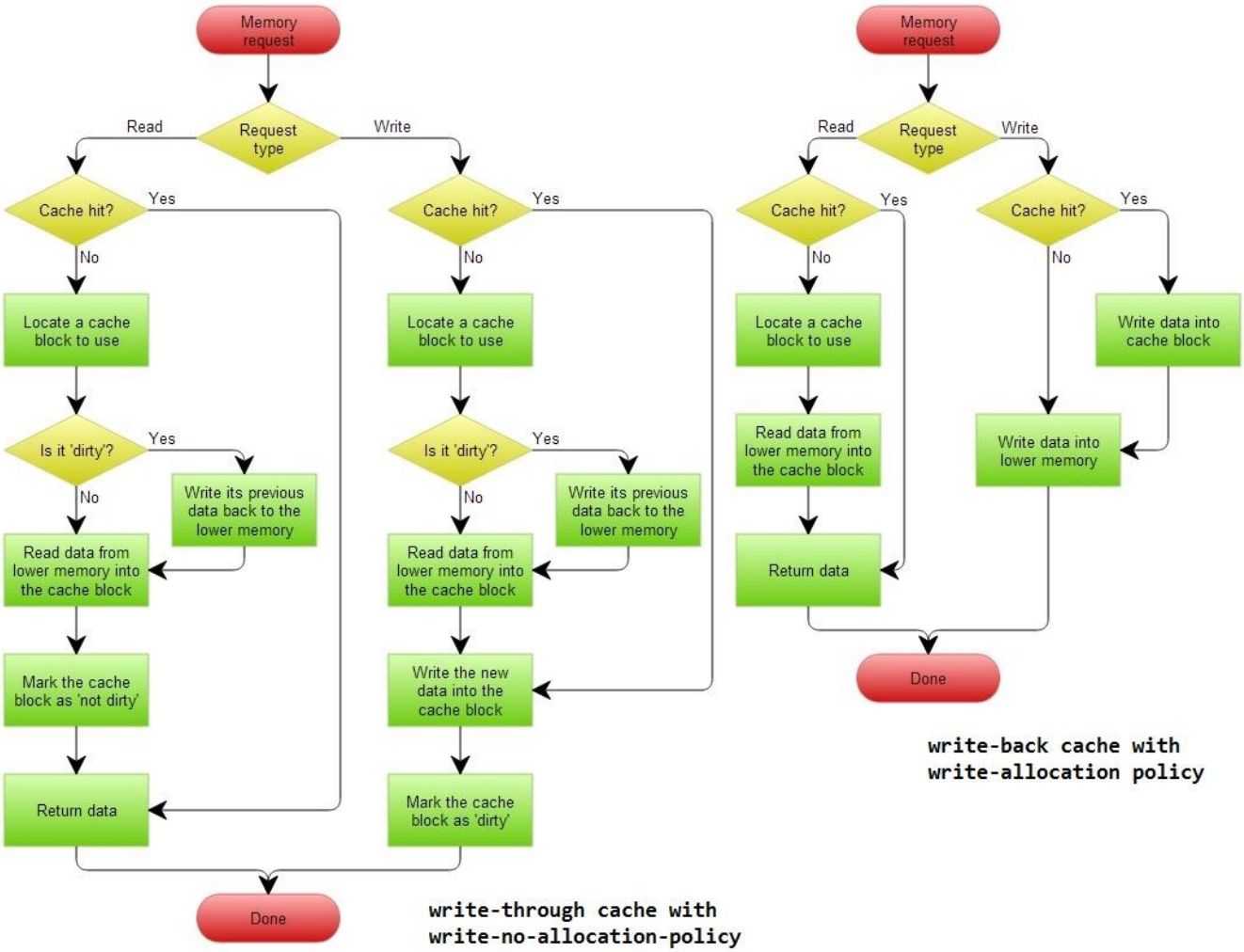
Write Miss Policies

A situation may arise if the data that needs to be written is not in the cache. The question is if such data should be brought in caches during such scenario, or in main memory, or perhaps in both. There are two policies regarding this issue:

- *write-allocat policy*;
- *write-no-allocate policy*.

The write-allocate cache will write data to the cache. The write-no-allocate cache will write data in main memory, and will not update caches. Note that in the former case, there must be some additional mechanism to inform subsequent accesses to the memory location in question that data in the cache may be different from the data in main memory. This is usually done by using dirty flag.

Following picture illustrates what happens logically on data read and data write in two combinations of write policies described in previous sections:



Reads vs. Writes

In general, reads get more priority over writes in cache organizations. Writes can be most of the time deferred, and in such cases microprocessor cores can most of the time continue execution even if write to main memory is not completed. Writes are usually performed in on-chip buffers called *write buffers*.

Replacement policies

For set-associative and fully-associative caches, sets contain more than one cache line, and a choice needs to be made when evicting existing data from the cache and bringing in new data to replace it. The performance impact is highly dependent upon the characteristics of the application, but still some replacement options are generally better than others. Popular options include:

- *First In, First Out (FIFO) replacement policy* evicts the oldest line to have been brought into the set or cache.

- *Last In, First Out (LIFO) replacement policy* evicts the youngest line to have been brought into the set or cache.
- *Least Recently Used (LRU) replacement policy* evicts the least recently referenced line in the set or cache
- *Random replacement policy* evicts a random line from the set or cache.

Memory translation

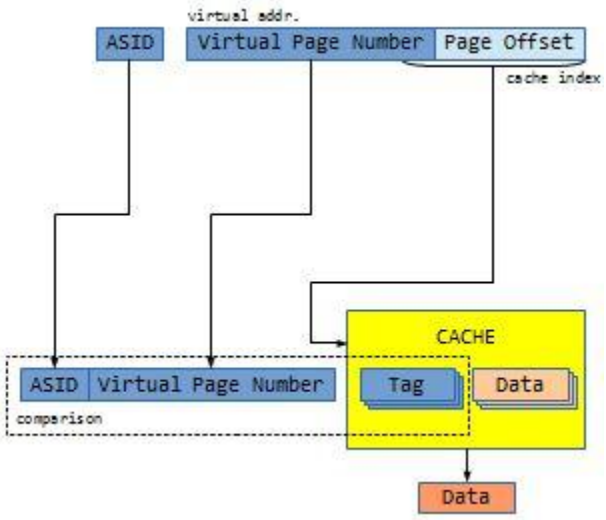
In modern computer systems, cache must translate virtual address to physical address. The information on this mapping is stored in page tables, which are stored in main memory. If main memory is always looked up during such process, this would have severe consequences on performance. Processors use additional cache for this translations, and these caches are *called translation lookaside buffers*, or *TLBs*. TLB is basically a page table cache. Details of TLB design options are beyond the scope of this document.

Virtual Memory and Cache Indexing and Tagging

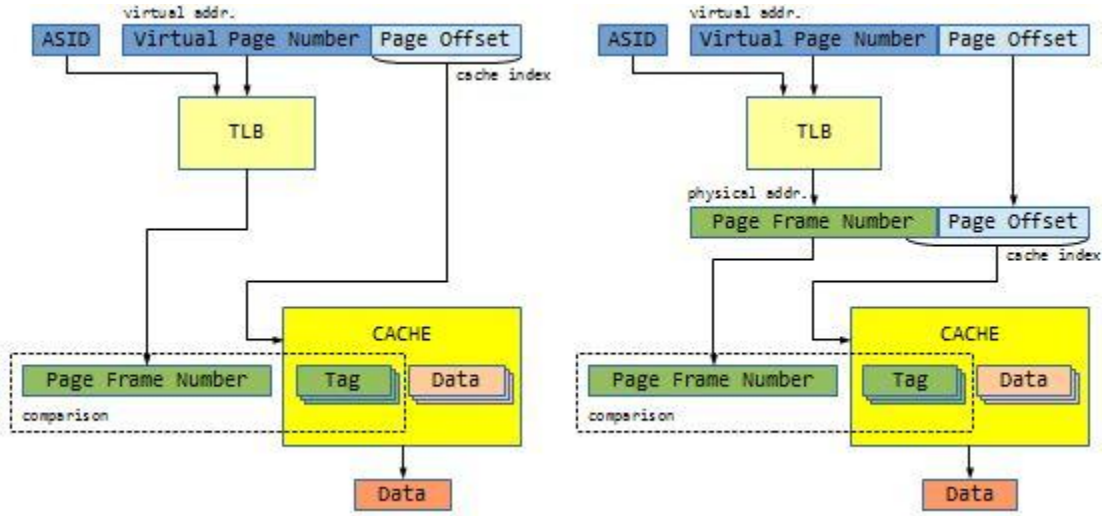
Most modern computer systems use virtual memory: they divide the physical memory into pages and allocate them to different processes, and also giving each process a contiguous address space, called *virtual memory* address space. Applications deal only with *virtual addresses*. This means that read and store instructions in complied application will contain virtual addresses, and cache must be organized to address this fact.

There are different possibilities as to which address to use for the index and the tag of a cache:

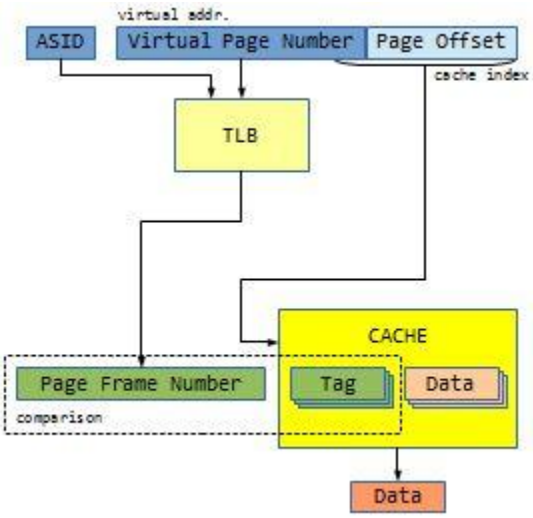
Virtually indexed, virtually tagged caches use the virtual address for both the index and the tag, which can lead to faster caches than other approaches because no address translation is required. However, there are a number of drawbacks. After every process switch, the virtual addresses refer to different physical addresses, and that requires the cache to be flushed. Apart from that, two different virtual addresses can refer to the same physical address; this is also called “aliasing". In such a case, the same data can be cached twice in different locations. Moreover, virtual-to-physical address mappings can change, which requires the cache to be flushed as well. Following figure illustrates this cache configuration:



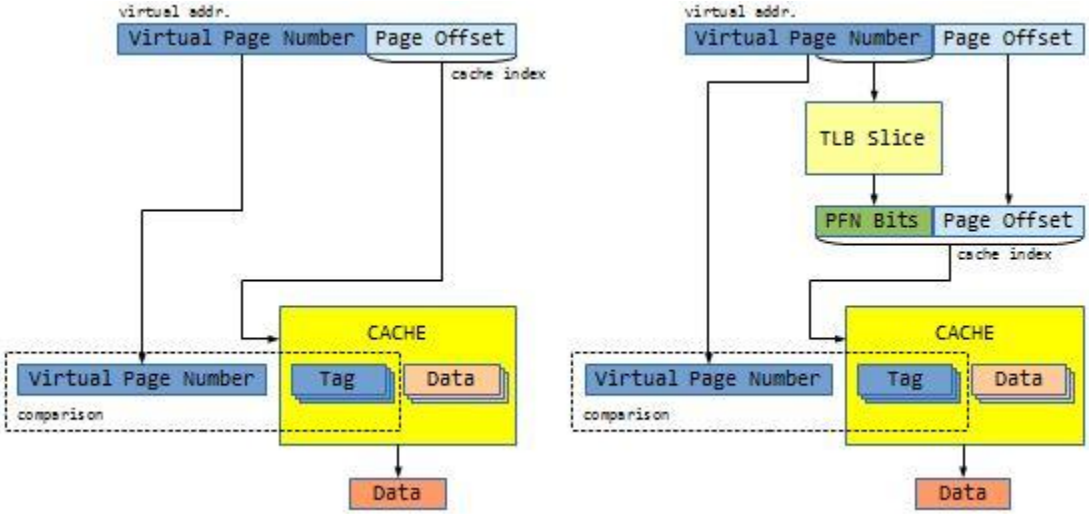
Physically indexed, physically tagged caches use the physical address for both the index and the tag. While this approach avoids the problems just mentioned, it is much slower as every cache access requires expensive address translations. Following picture illustrates this cache configuration, for two essentially different cases – on the left it the case where cache index is entirely contained in page offset, and on the right is the case where cache index has more bits that page offset:



Virtually indexed, physically tagged caches combine the advantages of the two other approaches. They use the part of the address that is identical for both the virtual and physical address (i.e. a part of the page offset) to index the cache. While checking the cache for this index, the remaining bits of the virtual address can be translated, and thus the physical address can be used as tag. This approach is, however, limited to smaller caches.



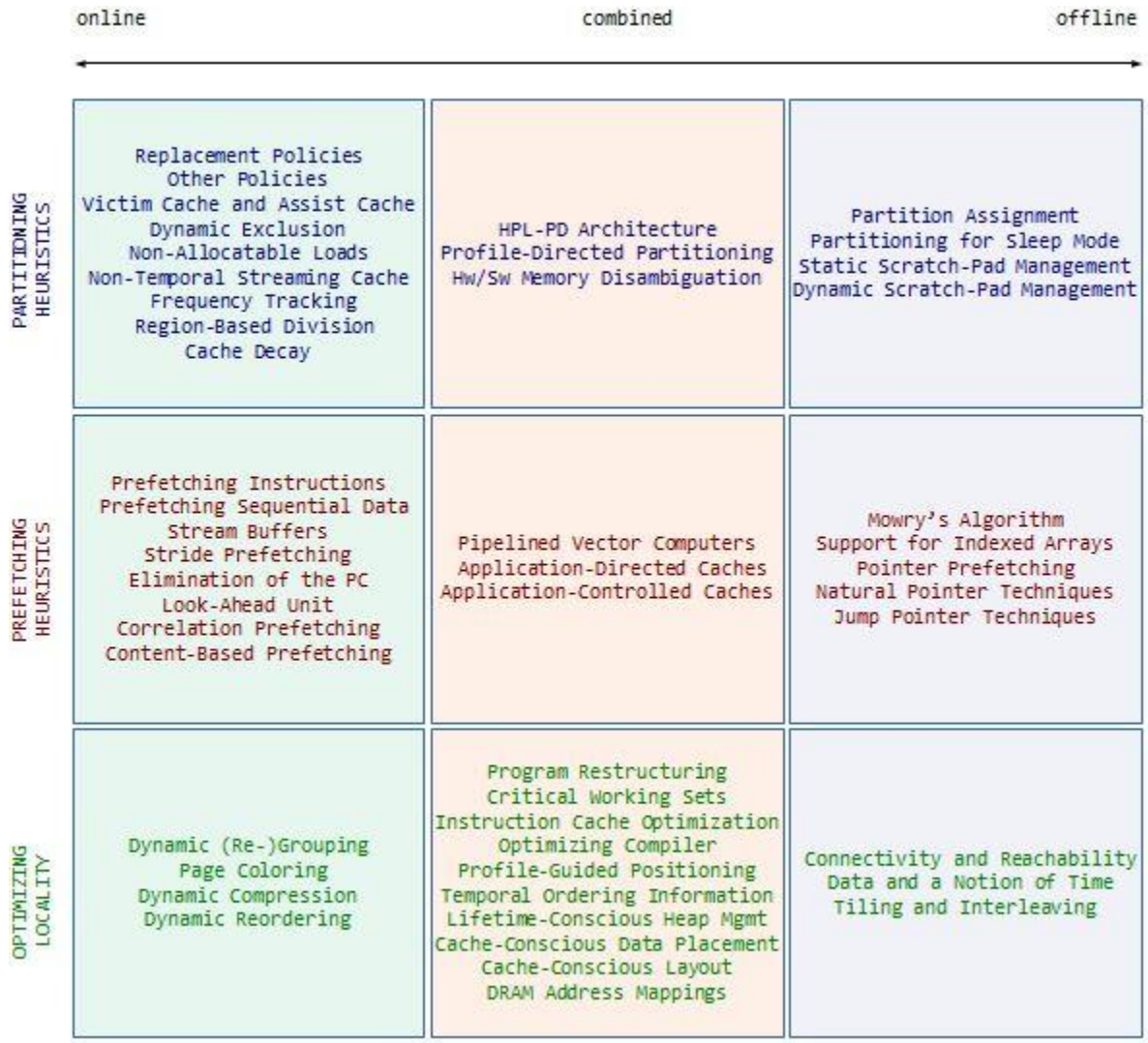
Physically indexed, virtually tagged caches organization is rarely used, and is illustrated on the following figure (there are two essentially different cases, depending on the number of bits required for page offset and cache index):



Modern processors often use virtually indexed, physically tagged first-level caches and physically indexed physically tagged higher-level caches.

Cache Content Management in General

Few key cache design options were presented in preceding sections. However, there are numerous other important cache design techniques developed in last several decades. Of course, it is beyond the scope of this document to describe all of them. Instead, following diagram tries to organize most important of these techniques in several logical groups:



The diagram itself is based on categorization found in Chapter 3 of the book "Memory Systems: Cache, DRAM, Disk" by Jacob at al. One can find a brief description for each technique in this book, and also pointers to relevant papers, where correspondent ideas are further explained and examined.

Quantifying Cache performance

AMAT – Average Memory Access Time

Average memory access time (AMAT) is defined as:

Average memory access time = Hit time + Miss rate × Miss penalty

MPKI – Misses Per 1,000 Instructions

Misses Per 1,000 Instructions (MPKI) represents number of misses (for particular cache level or cache system as a whole) divided by thousands of executed instructions.

CPI – Cycles Per Instruction

Cycles Per Instruction (CPI) is average number of clock cycles that is needed for one instruction execution.

IPC – Instructions Per Cycle

Instruction Per Cycle (IPC) is average number of instructions executed for each clock cycle.

Some Mips-Specific Cache Features

MIPS microprocessors usually have two-level cache with separate instruction and data caches at the lowest level. Also, MIPS instruction set includes certain instructions that enable application to directly manipulate cache. Those instructions are: CACHE, SYNC, etc. Details on these instructions and cache organization in general are available in MIPS product guides.

Qemu – Starting Point for the Tool

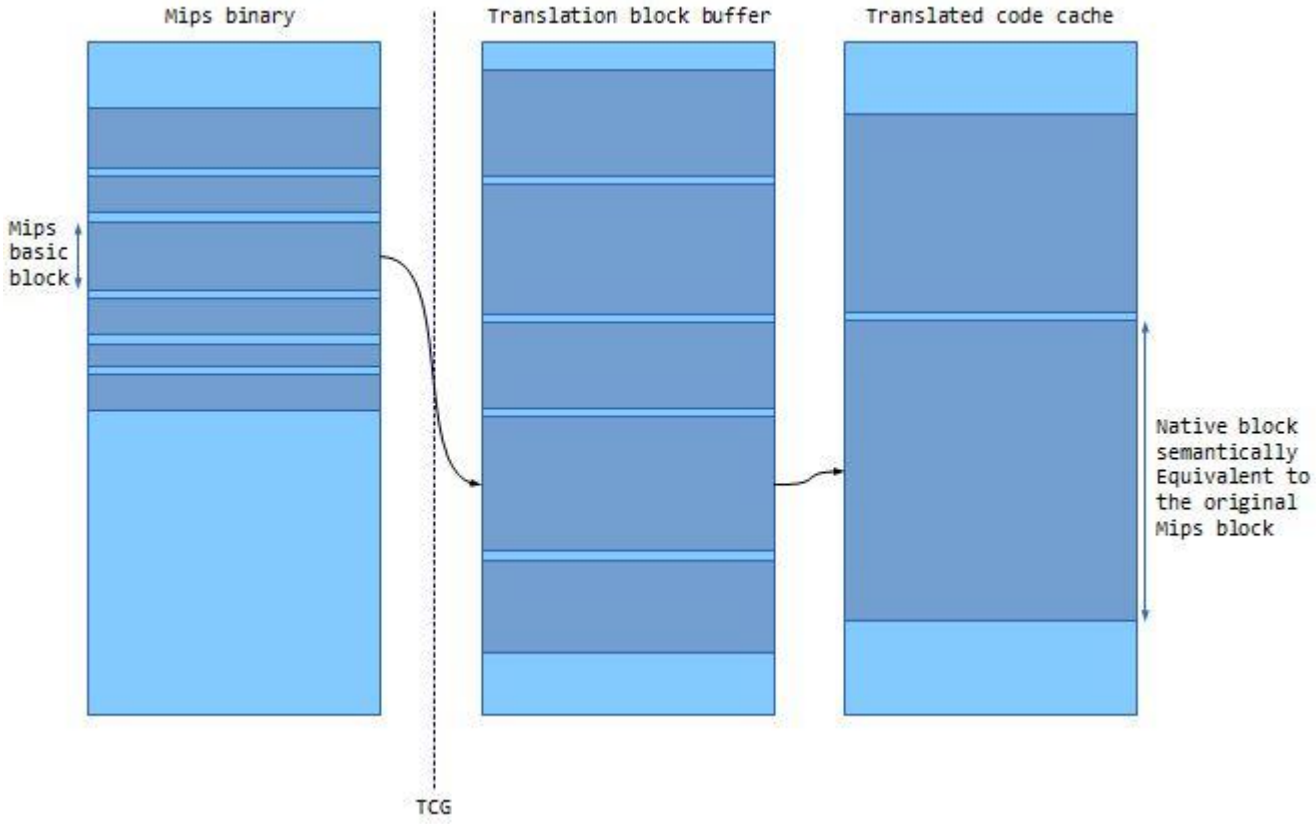
QEMU is an open-source processor and system emulator.

Qemu Basic Terms

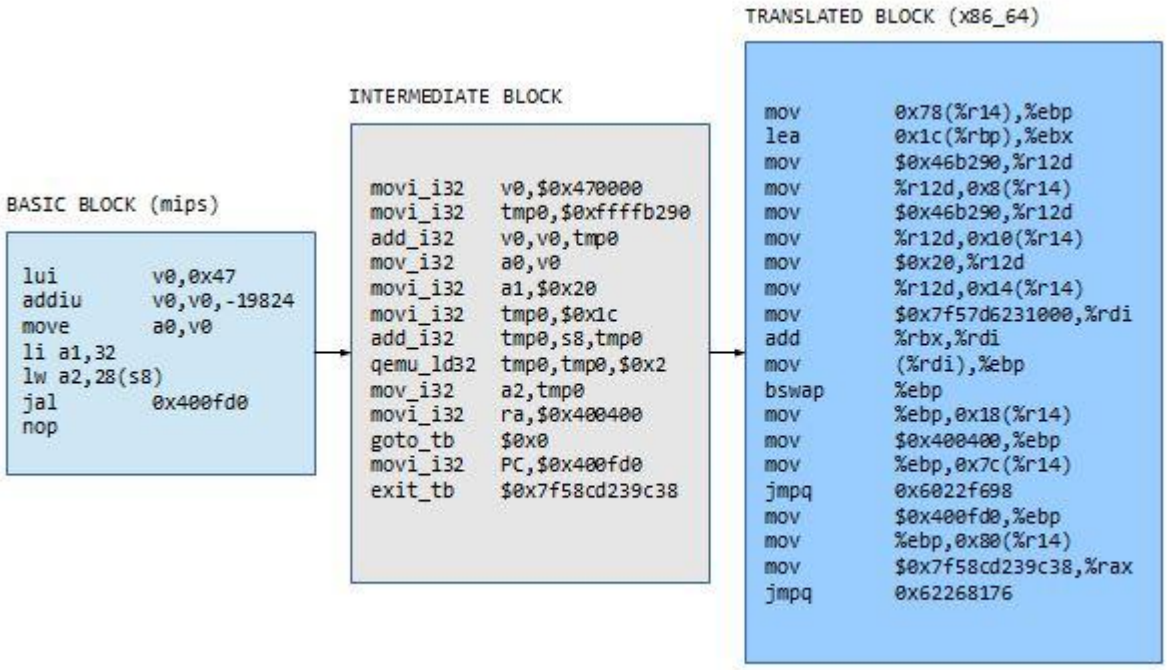
The system that is simulated is called *guest*. The machine where the simulation is run is called *host*.

With respect to granularity, Qemu divides original code in chunks that are portions of code from a certain location to the first branch or jump instruction. These chunks are called *basic blocks*.

Qemu translates basic blocks to an intermediate language via Tiny Code Generator – TCG. The intermediate language looks like other machine languages, with somewhat simplified instruction set. This is illustrated in the following figure.



Following picture illustrates an example of a basic block that is first translated to intermediate code, and it turn to host code (in this case, it is x86_64 code):



System and User Mode

Qemu can emulate computer system in two modes:

System mode refers to complete system emulation.

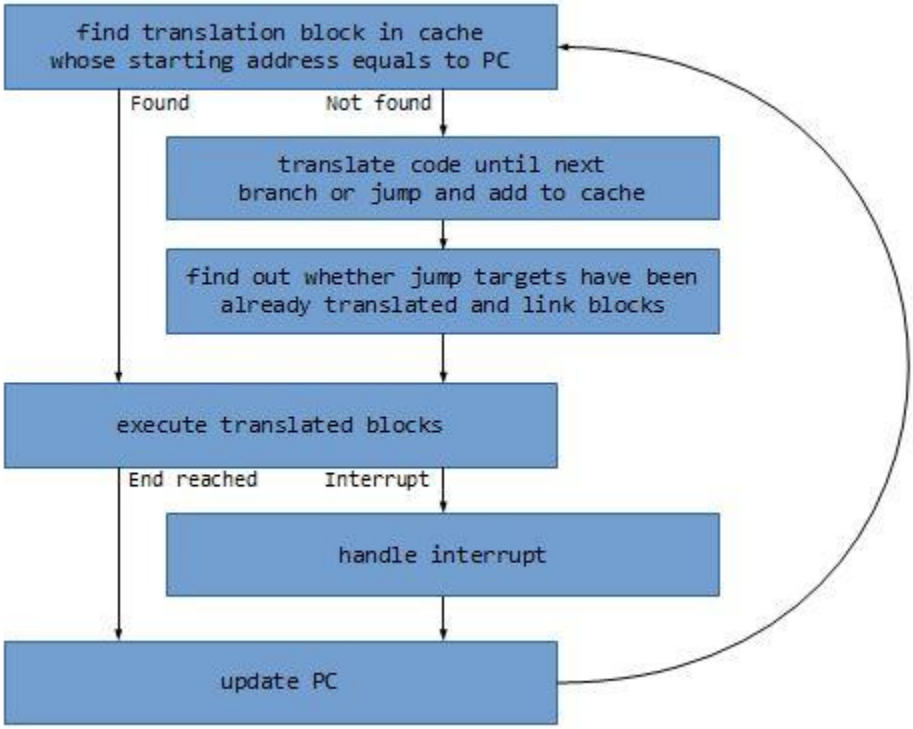
User mode offers possibility of executing individual application. The applications must be Linux user-mode command-line application.

The tool described in this document uses system mode only.

Basics of Qemu CPU Emulation

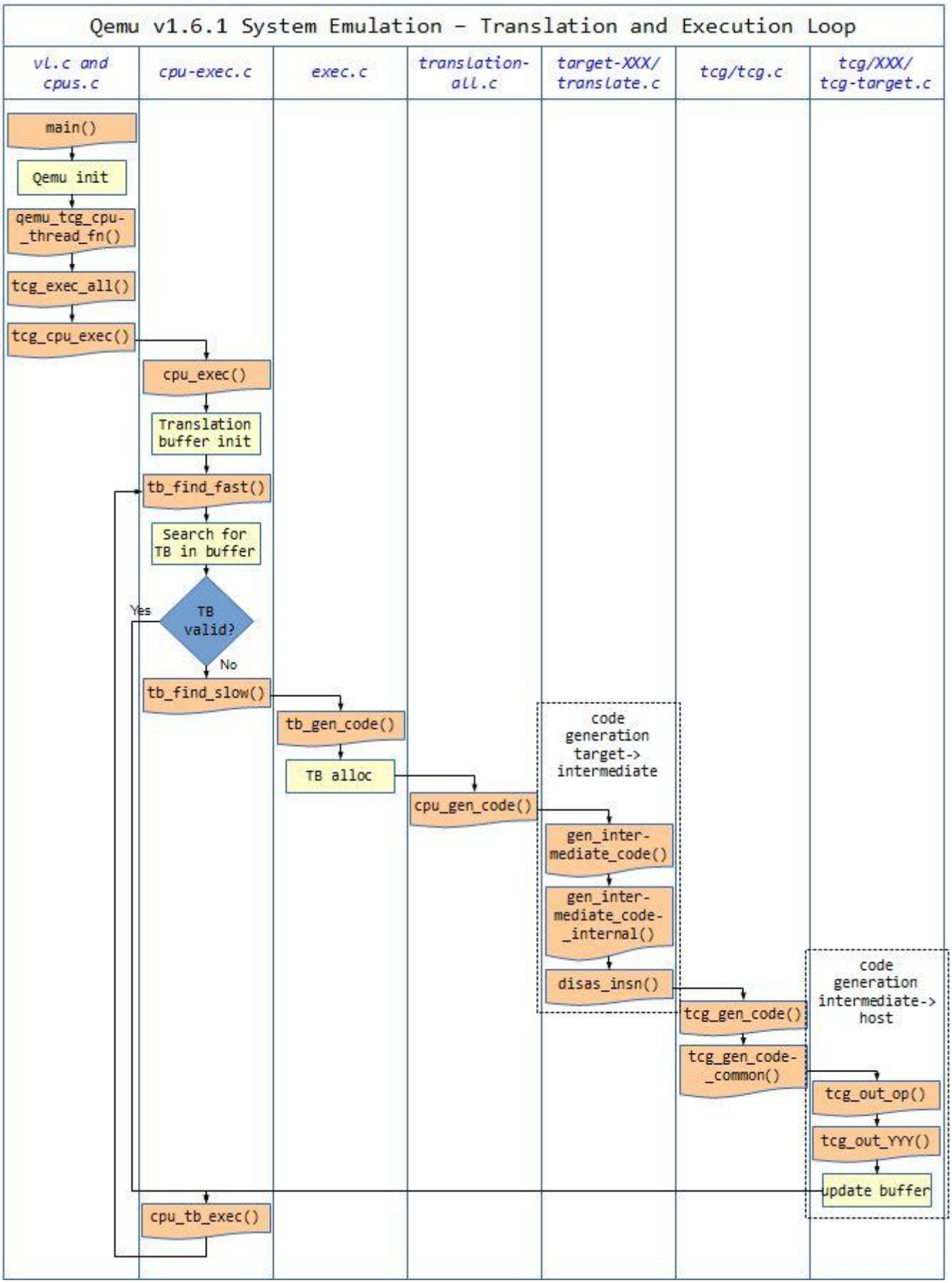
Qemu's emulation strategy is dynamic translation of basic-blocks from the emulated binary. No interpretation of instructions means that the simulation is temporarily halted whilst the translation is taking place, but as the sections of code being translated are small this pause is typically unnoticeable to the user. Once translated the native code remains stored in a static buffer in memory, negating the need for future translations so long as the buffer is not emptied.

To improve the execution speed further, Qemu's direct block chaining strategy allows native code to jump directly to the following native block. This avoids the need to return to the dispatch loop, where intensive tasks such as interrupt and exception processing take place.



Details of Qemu CPU Emulation

Following figure represents more detailed (but still simplified) view of main execution loop of Qemu.


Qemu v1.6.1 System Emulation – Translation and Execution Loop
vl.c and cpus.c
cpu-exec.c
exec.c
translation-all.c
target-XXX/ translate.c
tcg/tcg.c
tcg/XXX/ tcg-target.c
main()
Qemu init
qemu_tcg_cpu-_thread_fn()
tcg_exec_all()
tcg_cpu_exec()
cpu_exec()
Translation buffer init
tb_find_fast()
Search for TB in buffer
TB valid?
Yes
No
tb_find_slow()
tb_gen_code()
TB alloc
cpu_gen_code()
code generation target-> intermediate
gen_inter-mediate_code()
gen_inter-mediate_code-_internal()
disas_insn()
tcg_gen_code()
tcg_gen_code-_common()
code generation intermediate-> host
tcg_out_op()
tcg_out_YYY()
update buffer
cpu_tb_exec()

Guest Memory Implementation

During startup, Qemu allocates contiguous portion of host memory for the needs of guest emulation.

Guest load and store instructions typically contain guest virtual addresses. Qemu translate those addresses to guest physical addresses, that are than translated to host virtual addresses, and translated host instructions see only them, host virtual addresses.

Design of Mips Cache Model

Main Idea

Cache model tools generally are used in following contexts:

- Evaluating existing cache performance,
- Tuning applications to better use the cache, and
- New cache designs.

This tool is designed so that can be used in each of these areas.

The tool itself is a modified version of Qemu, known general-purpose system emulator/virtualizer. It is based on similar tool already developed prior to 2010 in MIPS.

Implemented Functionality

Following functionality is currently implemented within this tool:

- Ability to start system mode Qemu emulation for MIPS processor that produces statistics on cache usage and instruction counts.
- Ability to define cache configuration that needs to be simulated.
- Ability to control starting, stopping, and reporting of cache and instruction statistics.

Data Structures

The most important new data structure is mips_cache_t. It contains fields for cache configuration, and also various statistics counters and other variables needed for cache simulation. Each object of type mips_cache_t corresponds to one cache level.

Interface with Core Qemu Modules

The interface is following:

- Initialization:
  - vl.c
  - target-mips/cpu.c
- Command line handling:
  - vl.c
  - target-mips/cpu.c
- Hot key support:
  - ui/sdl.c
- Core simulation:
  - target-mips/op_helper.c

Changes by source code file

Source code file	Description of changes
include/qemu/log.h	Declaration of constants, variables, and functions related to logging.
include/qemu/option.h	Declaration of structure cache_model_t and variables used for handling command line options.
qemu-log.c	Definition of functions related to logging.
qemu-options.hx	Additional options and help messages for Qemu command line.
target-mips/cache_defs.h	Declaration of constants, variables, structures, and functions related to core cache model.
target-mips/cpu.c	Invocation of cache model initialization.
target-mips/cpu.h	Extension of CPUMIPSTLBContext, TCState, and CPUMIPSState structures.
target-mips/helper.c	A function changed from static to non-static.
target-mips/helper.h	Declaration of additional helpers.
target-mips/op_helper.c	Core functionality of cache simulation.
target-mips/translate.c	Individual instruction instrumentation and instruction count stats.
ui/sdl.c	Control via hot keys.
vl.c	Initialization.

Limitations

- TLB simulation is implemented as so called JTLB (Joint TLB), meaning that it is not possible to configure simulation to handle systems with multiple TLBs.
- Simulation works in Qemu system mode only.
- It is not possible to simulate processors with more than one core.

Usage and Verification of Mips Cache Model

This part of the document describes practical steps needed to use the tool. Also, various methods for its verification are described. These verifications can also be used as starting point for various real-life application of the tool.

Build

Before attempting to build the tool, it is desirable to make sure that your system is capable of building Qemu (without any changes required by this tool). Once you confirm that, follow this procedure to build the cache model tool:

Step 1. Clone Qemu source

```
$>  git clone git://git.qemu-project.org/qemu.git qemu-v1.6.1
```

Step 2. Checkout version v1.6.1:

```
$> cd qemu-v1.6.1
$> git checkout v1.6.1
```

Step 3. Apply the cache model patch (accompanying this documentation):

```
$> patch -p1 < qemu-v1.6.1-cache-model.patch
```

Step 4. Configure the build:

```
$> ./configure --target-list=mipsel-softmmu --enable-sdl --disable-werror
```

Step 5. Build:

```
$> make -j4
```

Configuration and Control

Extended Qemu Command Line

Once one built the tool, and designed and prepared your set of cache tests/benchmarks, one need to specify a cache model options that will be used during simulation. This operation is performed by using a new Qemu command line options "*-cache" and "-inscount".* Any basic cache parameter which exists on real systems (such as size, number of ways, sets, cache line size, collision victim selection algorithm, type of cache, level, …) can be configured.

Two Additional CLI Options

To see the list of all available Qemu command line options, type following line in the terminal:

```
$> qemu-system-mipsel –help
<.....................>
<.....................>
<.....................>
<.....................>
<.....................>
<... description of other Qemu switches ...>
<.....................>

-msg timestamp[=on|off]
                change the format of messages
                on|off controls leading timestamps (default:on)
-cache lNmodel[,lNparam][,LNparam=value]
                Enable cache simulation.
                Use this option for applications profiling, and gather some statistical
                information about cache utilization.
                This option impacts performance significantly.
                Type -cache ? for available options and usage
-inscount true[,icountlog=PATH] : Enable instruction count statistics.
                PATH should point to an existing directory.
                A file named icount.log will be created.
```

Last two options are new, added by the tool.

Syntax of Additional CLI Options

One can use following command to get help on the syntax of two new CLI options:

```
$> qemu-system-mipsel -cache ?
Cache model simulation configuration

Available parameters (comma separated values) :
       l1imodel      : Enable L1 instruction cache
```

```
        l1iways=VAL    : Set number of ways.
                         Default(4)
        l1isets=VAL    : Set number of sets.
                         Default(256)
        l1ilength=VAL  : Set line size in bytes.
                         Default(32)
        l1ialg=VAL     : Victim selection algorithm.
                         Type -cache l1ialg=? for available options.
                         Default(lru)
        l1irdal        : Allocate line on Read operation. Default.
        l1iwral        : Allocate line on Write operation.
        l1itype=VAL    : Cache type.
                         Type -cache l1itype=? for available options.
                         Default(VIPT)
        l1ienops       : Enable cache,pref,synci instruction simulation.
                         Not enabled by default.
-------------------------------------------------------------------
        l1dmodel       : Enable L1 data cache
        l1dways=VAL    : Set number of ways.
                         Default(4)
        l1dsets=VAL    : Set number of sets.
                         Default(256)
        l1dlength=VAL  : Set line size in bytes.
                         Default(32)
        l1dalg=VAL     : Victim selection algorithm.
                         Type -cache l1dalg=? for available options.
                         Default(lru)
        l1drdal        : Allocate line on Read operation. Default.
        l1dwral        : Allocate line on Write operation.
        l1dwback       : If this option is set, write back policy is used,
                         otherwise write through is used. Default.
        l1dtype=VAL    : Cache type.
                         Type -cache l1dtype=? for available options.
                         Default(VIPT)
        l1denops       : Enable cache,pref,synci instruction simulation.
                         Not enabled by default.
-------------------------------------------------------------------
        l2model        : Enable L2 unified cache
        l2ways=VAL     : Set number of ways.
                         Default(8)
        l2sets=VAL     : Set number of sets.
                         Default(2048)
        l2length=VAL   : Set line size in bytes.
                         Default(32)
        l2alg=VAL      : Victim selection algorithm.
                         Type -cache l2alg=? for available options.
                         Default(lru)
        l2rdal         : Allocate line on Read operation. Default.
        l2wral         : Allocate line on Write operation.
        l2wback        : If this option is set, write back policy is used,
                         otherwise write through is used. Default.
        l2type=VAL     : Cache type.
                         Type -cache l2type=? for available options.
                         Default(VIPT)
        l2enops        : Enable cache,pref,synci instruction simulation.
                         Not enabled by default.
-------------------------------------------------------------------
        l3model        : Enable L3 unified cache
        l3ways=VAL     : Set number of ways.
                         Default(8)
        l3sets=VAL     : Set number of sets.
                         Default(4096)
        l3length=VAL   : Set line size in bytes.
                         Default(32)
```

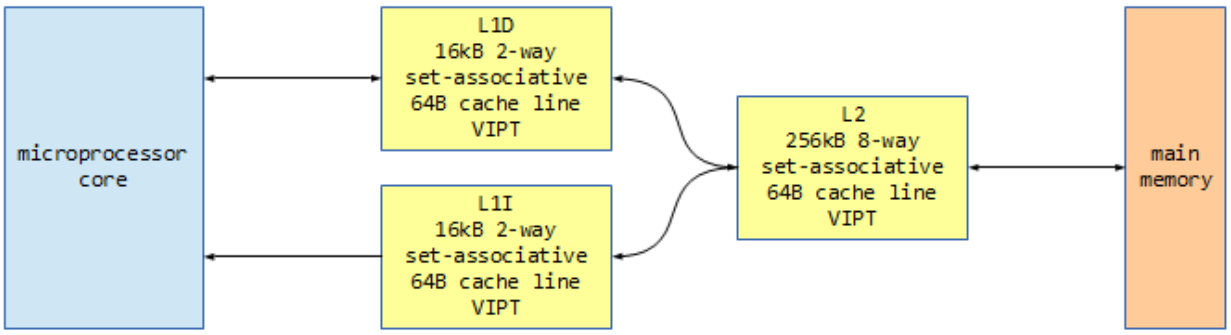
```
l3alg=VAL      : Victim selection algorithm.
                 Type -cache l3alg=? for available options.
                 Default(lru)
l3rdal         : Allocate line on Read operation. Default.
l3wral         : Allocate line on Write operation.
l3wback        : If this option is set, write back policy is used,
                 otherwise write through is used. Default.
l3type=VAL     : Cache type.
                 Type -cache l3type=? for available options.
                 Default(VIPT)
l3enops        : Enable cache,pref,synci instruction simulation.
                 Not enabled by default.


cachelog=PATH  : Create cache.log to a user defined PATH.
                 Default(/tmp/cache.log)
```

Example of usage :

```
-cache l1imodel,l1dmodel,l1dwback,l2model,l2wback
```

Example 1

Let's say that the following cache organization needs to be modeled:



In order to do that, one needs to start Qemu in this way:

```
$> qemu-system-mipsel –cache
l1imodel,l1iways=2,l1ilength=64,l1isets=128,l1itype=VIPT,l1dmodel,l1dways=2,l1dlength=64,l1dsets=128
,l1dtype=VIPT,l1dwback,l2model,l2ways=8,l2length=64,l2sets=512,l2type=VIPT,l2wback
<OTHER_QEMU_OPTIONS>
```

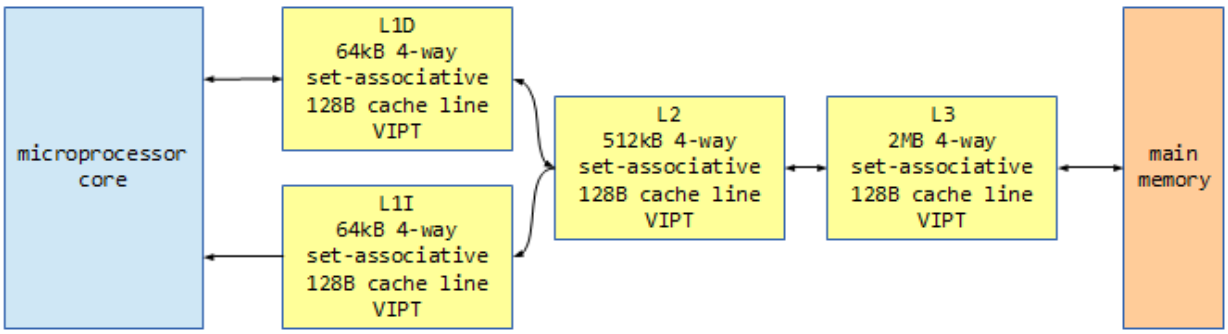
As an additional confirmation, the following will be the output in the terminal on Qemu startup:

```
Name     : L1-icache
Ways     : 2
Sets     : 128
Line     : 64
Size     : 16384
Flags    :
           VIPT     - Virtually indexed, physically tagged
```

```
            LRU      - Least recently used replacement, per set
            RDAL     - Allocate on read miss
            PASS_K1  - Uncached access to KSEG1
Mem       :
Next      : L2-cache

Name      : L1-dcache
Ways      : 2
Sets      : 128
Line      : 64
Size      : 16384
Flags     :
            VIPT     - Virtually indexed, physically tagged
            LRU      - Least recently used replacement, per set
            RDAL     - Allocate on read miss
            WBACK    - Write back - valid line set dirty
            PASS_K1  - Uncached access to KSEG1
Mem       :
Next      : L2-cache

Name      : L2-cache
Ways      : 8
Sets      : 512
Line      : 64
Size      : 262144
Flags     :
            VIPT     - Virtually indexed, physically tagged
            LRU      - Least recently used replacement, per set
            RDAL     - Allocate on read miss
            WBACK    - Write back - valid line set dirty
            PASS_K1  - Uncached access to KSEG1
Mem       :
Next      : subsystem
```

Example 2

Let's suppose that three-level cache is now to be modeled:



Command line should look like this now:

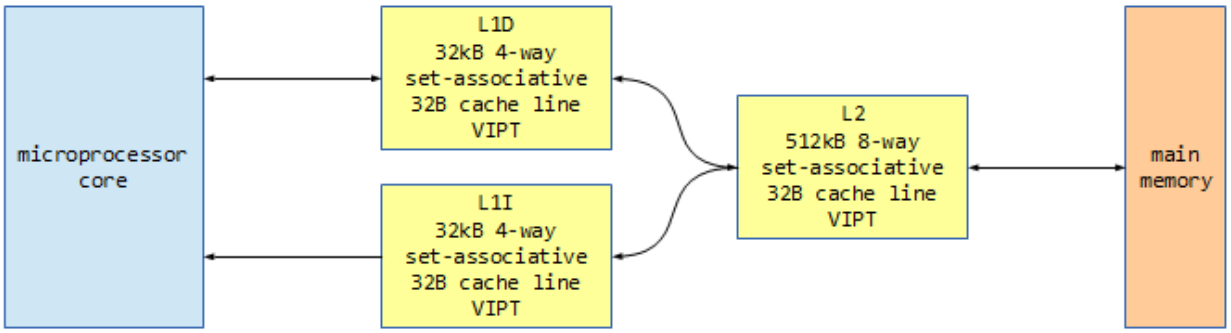
```
$> qemu-system-mipsel -cache
l1imodel,l1iways=4,l1ilength=128,l1isets=128,l1itype=VIPT,l1dmodel,l1dways=4,l1dlength=128,l1dsets=1
```

```
28,l1dtype=VIPT,l1dwback,l2model,l2ways=4,l2length=128,l2sets=1024,l2type=VIPT,l2wback,l3model,l3way
s=4,l3length=128,l3sets=4096,l3type=VIPT,l3wback <OTHER_QEMU_OPTIONS>
```

Confirmation similar to the one from previous example will be displayed in the terminal.

Default Configuration

If most of the cache simulation-related parameters are omitted, this organization will be in effect:



This is achieved by following simple command line:

```
$> qemu-system-mipsel -cache l1imodel,l1dmodel,l2model <OTHER_QEMU_OPTIONS>
```

Controlling the cache simulation

Following the example of some existing profiling tools like *perf, OProfile,* similar control applications have been implemented.

Start/Stop Simulation via Control app

Sim-control (Accompanying this documentation) – This application can be used to start/stop cache model simulation at desired time by evoking it with appropriate switches:

```
sim-control --start
sim-control --stop
```

This approach is good when writing scripts which will execute some list of tests:

```
$> sim-control --start
$> ./test1
$> ./test2
$> sim-control --stop
```

Start/Stop Simulation Via Wrapper App

Sim-wrapper (Accompanying this documentation) – This application is useful when you want to run simulation over one executable during its lifetime. The wrapper application starts cache model simulation, runs the given executable and waits for it to finish, after which it stops the simulation.

$> sim-wrapper /bin/echo “Hello World”

Start/Stop Simulation Via Keyboard Hot-Keys

Hot-key combinations have been instrumented to control simulation flow. This allows the user to filter out some parts of execution from the simulation. For example, application boot-up process is not of interest, so you can start your application (like browser), and after entering the address of some benchmark site, you start the simulation by pressing a defined key combination on the keyboard.

Key combinations are:

- [CTRL + '] - This key combination starts the simulation.
- [CTRL + ;] - This key combination stops the simulation.

Notification about simulation status can be seen in the terminal where you started Qemu.

NOTE: Your keyboard must be grabbed by Qemu in order for this to work, it is not possible via remote view/control applications, for example VNC.

Gathering simulation results

You do not need some special tool to view the results of cache model simulation, it is in plain text format and is stored on the host where you run Qemu, by default in the file **/tmp/cache.log.N** unless you change this setting via cachelog parameter (described in the CLI section). N specifies the current simulation iteration.

Executed instructions counters

Sometimes there is a need to see some statistics about number of specific instructions executed, for example to see if some application is memory or cpu intensive.

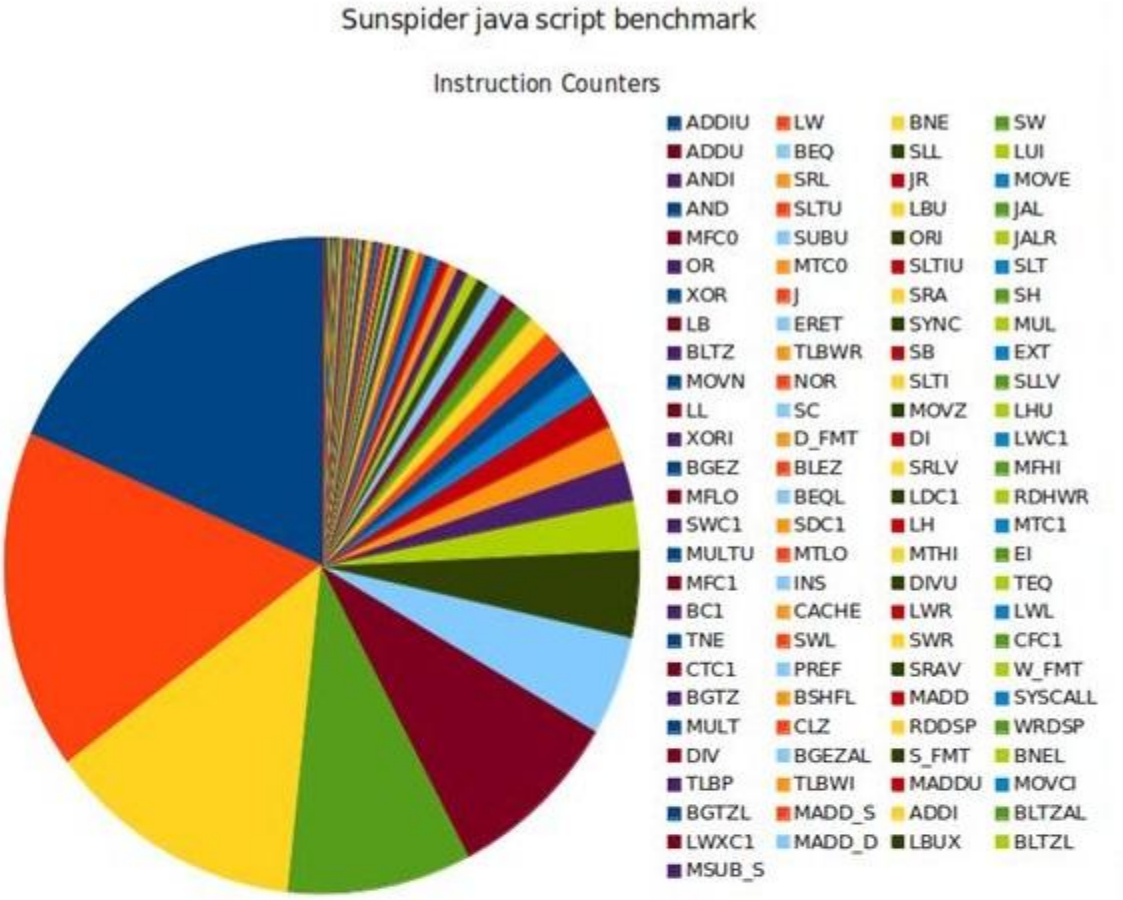
You can activate this option separately through command line, by using switch “-icount”.

```
-inscount true[,icountlog=PATH] : Enable instruction count statistics.
                PATH should point to an existing directory.
                A file named icount.log will be created.
```

Once you've enabled instruction counting feature, you control it the same way as you would control cache simulation, using hot keys, sim-control or sim-wrapper application.Results will be written by default in a file named /tmp/**icount.log.N** unless you change the destination by using *icountlog* switch.

Sunspider java script benchmark example:

Sunspider java script benchmark



Instruction	Count
ADDIU	2167904105
LW	1978284709
BNE	1533534273
SW	1102917029
ADDU	1046133696
BEQ	576936392
SLL	504548034
LUI	285608868
ANDI	228793555
SRL	215335781
JR	209682971
MOVE	154292054
AND	151053300
SLTU	137208656
LBU	122966193
JAL	115772127
MFC0	105179670
SUBU	94992244
ORI	70499927
JALR	67872075
OR	65916988
MTC0	56218993
SLTIU	56134714
SLT	48340878
XOR	44196777
J	41496466
SRA	32706304
SH	29838528
LB	28934837
ERET	28249979
SYNC	27336419
MUL	25209089
BLTZ	21689162
TLBWR	21416278
SB	19781706
EXT	19384290
MOVN	18488613
NOR	17888500
SLTI	17769270
SLLV	17626774
LL	17209250
SC	17085611
MOVZ	16706104
LHU	14768933
XORI	14131963
D_FMT	12604259
DI	11995375
LWC1	11814342
BGEZ	9669045
BLEZ	9291842
SRLV	8966094
MFHI	8825905
MFLO	7564788
BEQL	7503387
LDC1	7201153
RDHWR	5507183
SWC1	4438273
SDC1	4382460
LH	3861610
MTC1	3623770
MULTU	3575412
MTLO	3112853
MTHI	3040019
EI	2979790
MFC1	2585776
INS	2563580
DIVU	2291518
TEQ	2186463
BC1	2076210
CACHE	2041393
LWR	1968476
LWL	1968240
TNE	1952002
SWL	1781444
SWR	1765671
CFC1	1619652
CTC1	1614165
PREF	1607064
SRAV	1417088
W_FMT	1384661
BGTZ	1337560
BSHFL	1063166
MADD	960522
SYSCALL	889141
MULT	868660
CLZ	853632
RDDSP	564999
WRDSP	564978
DIV	233280
BGEZAL	230620
S_FMT	129492
BNEL	112987
TLBP	47567
TLBWI	26271
MADDU	24150
MOVCI	23838
BGTZL	1313
MADD_S	835
ADDI	831
BLTZAL	710
LWXC1	445
MADD_D	339
LBUX	66
BLTZL	24
MSUB_S	17
Sum	11756758481

Verification Using DineroIV

http://pages.cs.wisc.edu/~markhill/DineroIV/
http://www.cc.gatech.edu/~bader/COURSES/UNM/ece438/dinero/d4.pdf

Dinero IV is a cache simulator for memory reference traces.

- subroutine-callable interface in addition to trace-reading program
- simulation of multi-level caches
- simulation of dissimilar I and D caches
- good performance, especially for highly associative caches
- classification of compulsory, capacity, and conflict misses
- support for multiple input formats
- cleaned up and modernized code, improved portability

Dinero is an offline cache simulation tool. It's simple, configurable and easy to use. It can analyze memory traces generated in specific format and output the cache statistic results for the given configuration.

It was used as first level of verification for cache model simulation implemented in Qemu. It was good for validating the algorithms used for victim selection on cache collision events as well for overall results.

The great disadvantage of Dinero is lack of scalability. It can be used only for small examples, because memory traces can be very large in size.

Because of that, for more precise and reliable verification, oprofile/perf tools were used. Those tools make use of hardware performance counters implemented in processors and the results gatherd are more trustworthy.

Verification Using Hardware Performance Counters

The best method of validation for software simulations is correlation with data gathered from real hardware system. Therefore, implementation of cache model in Qemu, should be verified in the same way.

Qemu cache model configuration

Cache model should be configured to match the cache parameters on the available hardware.

Hardware description

CPU :

```
system type           : MIPS Malta
processor             : 0
cpu model             : MIPS 74Kc V4.9
BogoMIPS              : 15928.13
wait instruction      : yes
microsecond timers    : yes
tlb_entries           : 32
extra interrupt vector       : yes
hardware watchpoint   : yes, count: 4, address/irw mask: [0x0000, 0x0000, 0x0000, 0x0000]
ASEs implemented      : mips16 dsp
shadow register sets  : 1
kscratch registers    : 0
core                  : 0
VCED exceptions               : not available
VCEI exceptions               : not available
```

Operating system

Linux debian 3.2.39 #5 PREEMPT Fri Oct 4 13:11:07 CEST 2013 mips GNU/Linux

Cache:

Primary instruction cache 32kB, VIPT, 4-way, linesize 32 bytes.
Primary data cache 32kB, 4-way, PIPT, no aliases, linesize 32 bytes
MIPS secondary cache 512kB, 8-way, linesize 32 bytes.
Qemu emulation description:

CPU :

```
system type           : MIPS Malta
processor             : 0
cpu model             : MIPS 74Kc V0.0  FPU V0.0
BogoMIPS              : 6536.90
wait instruction      : yes
microsecond timers    : yes
tlb_entries           : 16
extra interrupt vector       : yes
hardware watchpoint   : yes, count: 1, address/irw mask: [0x0ff8]
ASEs implemented      : mips16 dsp
shadow register sets  : 1
kscratch registers    : 0
core                  : 0
VCED exceptions               : not available
VCEI exceptions               : not available
```

Cache model

```
Name     : L1-icache
Ways     : 4
Sets     : 256
Line     : 32
Size     : 32768
```

```
Flags    :
            VIPT     - Virtually indexed, physically tagged
            LRU      - Least recently used replacement, per set
            RDAL     - Allocate on read miss
            EN_OPS   - Enable cache and prefetch operations
            PASS_K1  - Uncached access to KSEG1
Mem      :
Next     : L2-cache

Name     : L1-dcache
Ways     : 4
Sets     : 256
Line     : 32
Size     : 32768
Flags    :
            PIPT     - Physically indexed, physically tagged
            LRU      - Least recently used replacement, per set
            RDAL     - Allocate on read miss
            WBACK    - Write back - valid line set dirty
            EN_OPS   - Enable cache and prefetch operations
            PASS_K1  - Uncached access to KSEG1
Mem      :
Next     : L2-cache

Name     : L2-cache
Ways     : 8
Sets     : 2048
Line     : 32
Size     : 524288
Flags    :
            VIPT     - Virtually indexed, physically tagged
            LRU      - Least recently used replacement, per set
            RDAL     - Allocate on read miss
            WBACK    - Write back - valid line set dirty
            EN_OPS   - Enable cache and prefetch operations
            PASS_K1  - Uncached access to KSEG1
Mem      :
Next     : subsystem
```

Test case selection

There are many studies about how matrix operations impact performance and how should they be optimized to utilize cache in the best way possible. Examples can be found on the Internet and are very well documented:

http://www.aqualab.cs.northwestern.edu/classes/EECS213/eecs-213-s10/lectures/11-CacheMem.pdf

IJK matrix multiplication algorithms are the best choice for verification of Qemu cache model simulation, because the results of cache utilization can be easily calculated and are known in advance. The main goal of these algorithms is to make use of spatial and temporal data locality in such a way that the number of access to the main memory is brought to the minimum.

Defining correlation data parameters

When examining the optimal cache utilization on real system, performance is the best parameter to measure (cpu cycles, execution time). Emulation in Qemu is not ideal, for example, timers are not precise, because they depend on the host configuration on which the emulation is run. So it is necessary to define another parameter which can overcome these flaws.

Cache performance is tightly connected with parameters like Global miss rate.

global_miss_rate = (main_memory_loads + main_memory_stores) /
(L1I-cache_loads + L1D-cache_loads + L1D-cache_stores)

It defines the relationship between the number of accesses to main memory (which is the most expensive access) and the total number of cache accesses.

Gathering the data

Now that we have chosen the parameter with which the verification will be done, it is necessary to collect the data needed to calculate global miss rate on the hardware. *Perf* tool was used for this purpose :

https://perf.wiki.kernel.org/index.php/Tutorial

This tool gathers information about certain hardware events through Performance Counters. Events of our interest are:

L1I-cache-loads
L1D-cache-loads
L1D-cache-stores

L2-cache-load-misses
L2-cache-store-misses

L1D-cache-writebacks

Limitatons of perf tool

It is not possible to collect the information about exact number of main memory accesses on real hardware, but we can use the following approximation:

main_memory_access = (main_memory_loads + main_memory_stores) = (L2-cache-load-misses + L2-cache-store-misses + L1D-cache-writebacks)

Presenting the results

The best way to compare the results of testing from both platforms is by graphical representation. We want to see the behavior of cache across iterations through different MM algorithms and sizes of matrices. That would give us a picture of how global miss rate changes and where is the line of threshold. Both platforms (hardware/Qemu) should show very similar results.

The graph on the next page shows that comparison. Considering the introduced approximations and limitations of perf tool, the results show a high degree of correlation. Data from both platforms converge to expected values, for example, *bijk* (Block Matrix Multiply) algorithm shows the best results both in Qemu and on Hardware, which is expected. That algorithm uses blocks of matrix data whose size is aligned to match cache data line and therefore must give optimal results.

http://suif.stanford.edu/papers/lam-asplos91.pdf

Consider the example of matrix multiplication for matrices of size NxN:

```
for i:=1 to N do
    for k:= 1 to N do
        r = X[i,k]; /* register allocated */
        for j:=1 to N do
            Z[i,j] += r* Y[k,j];
```

Figure 1(a) shows the data access pattern of this code. The same element X[i,k] is used by all iterations of the innermost loop; it can be register allocated and is fetched from memory only once. Assuming that the matrix is organized in row major order, the innermost loop of this code accesses consecutive data in the Y and Z matrices, and thus utilizes the cache prefetch mechanism fully. The same row of Z accessed in an innermost loop is reused in the next iteration of the middle loop, and the same row of Y is reused in the otftermost loop. Whether the data remains in the cache at the time of reuse depends on the size of the cache. Unless the cache is large enough to hold at least one N x N matrix, the data Y would have been displaced before reuse. If the cache cannot hold even one row of the data then Z data in the cache cannot be reused. In the worst case, $2N^3 + N^2$ words of data need to be read from memory in N^3 iterations. The high ratio of memory fetches to numerical operations can significantly slow down the machine.

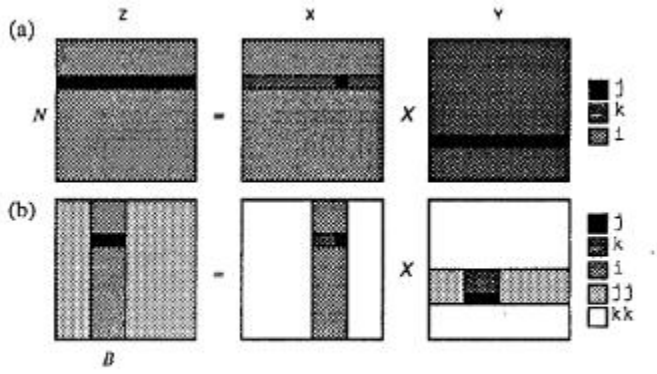
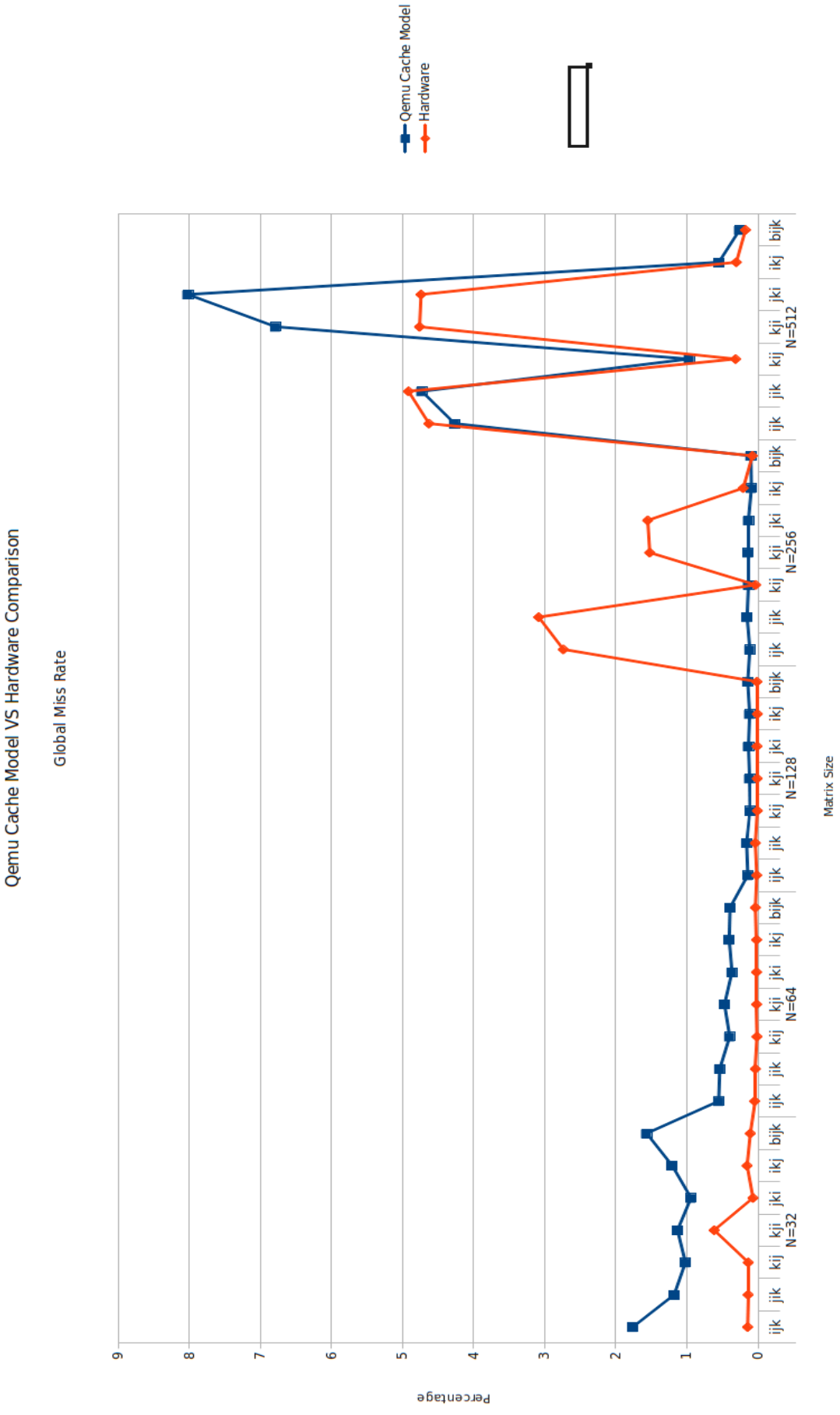


Figure 1: Data access pattern in (a) unblocked and (b) blocked matrix multiplication.


Qemu Cache Model VS Hardware Comparison
Global Miss Rate
Qemu Cache Model
Hardware
Percentage
Matrix Size
N=32
N=64
N=128
N=256
N=512
ijk jik kij kji jki ikj bijk ijk jik kij kji jki ikj bijk ijk jik kij kji jki ikj bijk ijk jik kij kji jki ikj bijk ijk jik kij kji jki ikj bijk

Possibilities for Future Extensions

- Complete TLB simulation.
- Breaking cache misses to compulsory, conflict, and capacity cache misses.
- Branch predictor simulation.
- Cache simulation in Qemu user-mode.
- Keeping track of cache statistics per code line.
- Comparisons Mips/Arm/Intel.
- Investigating current compiler optimizations.
- Investigating proposals for compiler improvement.
- Optimization of existing applications.
- Multi-core processor cache simulations.
- Investigating proposals for cache sections of a microprocessors design.
- Porting to Android emulator and/or FirefoxOS emulator.

Conclusion

The tool presented can be used for MIPS cache simulation. There are many areas where it can be applied.

Index