deqdip

A x86_64 Cache Dequeue Protocol that provides
a consistent deterministic baseline for performance characterisation tasks
including support for Elixir (Erlang)

marcel@aexa.uk

x86_64 Linux Release
(Dequeue D-Cache I-Cache P-Cache)

v1.01

Elixir & Erlang Multicore scalability background

The process model in Elixir follows that of Erlang which provides threads with a reduced process context containing at least a private stack. Stacks can be minimal and their treatment in software is well understood. Erlang then has N process schedulers per N cores of the CPU. The VM runs in one traditional process hardware context and performs all Elixir (Erlang) process scheduling for these stack based lightweight processes. This is a technique much used to create Transaction Processing Monitors for OLTP systems, notably on UNIX Tuxedo as a commercial project and Pinewood as a Labs-only project. OLTP systems have many of the desired design parameters including demanding scalability integrity and availability characteristics that the Erlang design centre also pursued.

The other major design win in this area of Elixir (Erlang) thread/core scalability comes from the decision to use Functional programming. This is advantageous in that all memory is by default private. At a fundamental level one can say there is no data sharing in Erlang or Elixir. All data is passed around on private stacks. This reduces the dependence on requiring performance degrading global locks and decreases likelihood of associated hard to debug problems such as race conditions that arise from the use of such locks. Functional programming in a large concurrent and parallel processor typical in modern x86_64 processors provides a large lever that effortlessly solves many of the problems that the representative corpus that is the last 20 years in language design and system software design consistently failed to respond to. Whilst CPUs rapidly moved forward with threading technology going from 2 hardware threads of execution circa 1990 to 256 hardware threads of execution in 2020, both languages and system software design responded to this by doing very little or nothing (Ex. pthreads already existed in 1990). This is why Elixir & Erlang has emerged recently as a premiere technology solution for scaleable core performance

It is inevitable that Elixir (Erlang) will be selected for evaluation purposes and subsequent performance characterisation, either in the evaluation of different algorithmic approaches taken in Elixir (Erlang) itself or when comparing a system using a comparable but different implementation approach.

Having experience in developing languages and system software it is often the case that performance characterisation can sometimes be useful in informing and evaluating some of the critical design decisions in terms of the software engineering undertaken.

This provided the motivation for me to develop the `deqdip` dequeue cache policy, the purpose of which is to provide a consistent and deterministic baseline for comparative performance characterization tasks, a mandatory requirement to having confidence in performance data.

High level design decisions

There are various ways to reliably produce performance characterisation data. For Intel CPUs this is a simple matter of reading out the CPUs high resolution microsecond timers. In fact Elixir already has a timer API to do just that:

```
{ usec, :ok } = :timer.tc( elixir-statement )
```

A popular approach to using this :timer facility is in the same way wall-clock calendar timing information is read., that is, being sufficient to just call the function and obtain timing data.

This is not only misguided but completely erroneous, not least in the case of timing metrics taken from high resolution microsecond timers.

Intel CPUs are some of the most heavily caching CPUs ever developed. The disparity between doing a read from cache and doing a read out of memory whilst executing the code under analysis can be hugely significant. Exactly how much process state is being cached during the analysis period on a multicore processor is a complete unknown and almost certainly never the same on two separate runs.

This problem can be approached in one of two ways, either by reaching a known position with the caches warm or taking the caches back to a zero state. Since there is no instrumentation in the Intel CPU to reach a known warm state trying to synthesize one is problematic. The effect of zeroing out the cache may be unacceptable or even impossible, however such analysis is usually done on for-purpose test systems and at most may cause some global latency whilst the CPU faults-in required state to cache. What we gain from this approach is a consistently deterministic state highly suitable for performance characterization tasks at the cost of slower overall program performance.

Intel has adopted many ways to cache (literally hide) process and program state to enhance CPU efficiency for example.

- Maintaining a Data Cache (DCache).
- Maintaining an Instruction Cache (ICache).
- Additional Ad-hoc physical memory caching metadata kept in MTTR registers describing caching behaviour for discrete memory ranges.
- Additional Ad-hoc virtual memory caching metadata kept in PAT tables again describing caching behaviour for discrete memory ranges including setting device hardware as uncached but also enabling large memory sections for very large database operation.

The first three of these can be dequeued (zero'd) with some control bits in a Processor Register and using the WBINVD instruction. This code sequence must execute in an elevated privilege mode. The simplest way to do that on Linux is to write a Kernel loadable module. It is also easy in UNIX to signal execution of the code in our Kernel module via the IOCTL mechanism.

The MTTR tables are cleared similarly, however Linux no longer uses MTTR tables, using PAT tables instead. This means the MTTR clear-down sequence can be ignored.

Currently PAT tables are used for I/O devices and GPUs. and will not affect `deqdip` objectives when running in a test system. This means they can safely be ignored.

when the decache code executes it will do so in the context of the current running thread and for the Multicore situation we have to arrange the code instead to run on every node (Intel: logical core) in the system. The kernel has effective support for this and avoids adding unnecessary complexity to the design.

Yet another often overlooked factor is still contributing significant nondeterminism and that is the Linux filesystem Page cache. The best approach is to either also decache the filesystem or mount the filesystem cacheless. A decache is a temporary action and allows the system to reheat the cache whilst a cacheless mount does not allow the possibility of ever reattaining the benefit of running from cache. Linux already has a standard way to flush this cache which we will use.

The only way we can access our C Kernel mode implementation for the decache is via a technique called a NIF file/function. This is a shared library (.so) that gets loaded with the ErlangVM during process startup. The NIF will be used to steer a dequeue cache request from an Elixir function call through the NIF to the actual decache sequence in the Kernel Loadable module.

The way Erlang interacts with a shared C/Linux Library is broadley speaking incompatible with the overall design of Erlang. It's VM runs in a single hardware process context. It caches all kinds of state including temporal metadata and a map of what the distributed environment looks like. At some point Erlang has to hand off control from the Erlang process to the NIF. At this point metadata state is no longer being updated or stored for later processing. This means the longer the time elapsed executing code in the NIF thread the greater the drift in the Erlang VM metadata, potentially reaching a point where it is not possible to recover or synthesize missing metadata leading to erratic or complete VM/System failure.

In this NIF latency matter it is only the decache of the filesystem Page-Cache that presents an opportunity for breaching latency guidelines. A way to mitigate this is to flush the filesystem Page-Cache before starting the Elixir (Erlang) VM. This means any subsequent decache issued by the NIF will run with minimal latency.

In fact, close analysis also reveals a similar situation involving the I&D Caches could develop and the call to dequeue the I&D Cache may vary greatly on different runs adding undesirable latency to the actual performance measurement timings. This necessitates taking the decache algorithm and formulating a decache protocol. Arranging two different I&D decache's will guarantee the second and final call returns in a more consistently deterministic manner suitable for comparative performance analysis.

There would be little point in executing the same decache algorithm twice back-to-back since the net overall effect observable at user level would appear to be almost exactly the same in the 1-Call design. The obvious solution to that is to introduce an arbitrary delay between the two calls. This solves two problems. Firstly because x86_64 is such a heavily cached Architecture dequeuing some of these large caches introduces more indeterminism by way of introducing jitter at the microarchitecture level. This delay will allow the system to recover to a stable state and allow the launch of the second more deterministic cache dequeue.

Then the cache protocol becomes this:

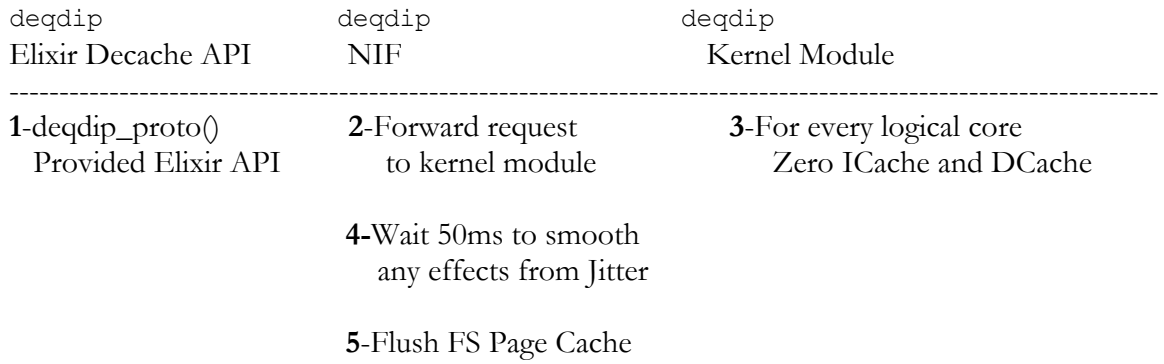

Figure 1 The Deqdip Dequeue Protocol

Unfortunately there are no set deterministic guidelines for how long a thread of execution should run in a NIF other than reducing it as much as possible. In order to keep the functionality-matrix as manageable as possible a decision was taken to keep the FS page cache flush in the NIF opting for a more integral design rather than a fragmentary one.

It is anticipated that there may be cases or users wanting to relocate that functionality due to concerns of NIF-ErlangVM latency. I would advise in this case to use the following technique as an alternative.

$(flush FS page cache in the normal 'Linux admin' way)
$(start ErlangVM and application)
Elixir/Erlang>Perform performance characterisation task/s

This way the overhead of the dequeue fs page cache in the NIF can be acceptably resolved and provides better overall response in terms of NIF to Erlang VM latency.

Validation

There are at least two good Cache-Bound test cases to use in Validation, either significant amounts of Matrix Math or equally significant amounts of String character data favoured because it is simpler to source and vary test data loads.

Method

To prove the system is being properly decached with `deqdip` it needs to perform a string search on a nonsignificant memory/cache bound input with caches warmed by completing two test runs back to back and comparing that with the cold cache `deqdip` case. To simplify the test setup `deqdip` was also implemented in a C utility program `deqdipctl` which can be paired with UNIX `sort`, both easily run from the command line.

The UNIX `sort` utility can be used to sort input files of strings. If the input file is large enough it will perform disk accesses to create temporary working files. Since we want to avoid anything doing disk accesses and keep everything memory/cache bound as much as possible there is a way in which sort can be invoked to avoid disk accesses simply by an argument to increase its default memory pool (`-S`) and giving it a temporary file specification (`-T`) that it is non existent.

Having one sort thread active however isn't much of a test for validating `deqdip` and fortunately we can also easily vary the number of cores sort runs on with an additional (`parallel=N`) argument.

```
$time sort --parallel=<n> -S <n>[K|M|G] -T /0 -o /dev/null
```

This will execute a memory/cache bound string sort avoiding any disk access other than to load the initial data and run 'sort' on 4 cores.

A wide variety of pre-made input string data can be found here

`dumps.wikimedia.orf/enwiki/latest/`

The tests were run on Linux (v5.6.6) using 25Mb of string data. The number of cores to use during testing was decided during preliminaries which indicated distinct non-linear scalability for this particular workload on the 2 Vs 4 core case. Testing then concentrated on warm Vs cold caches for both the 1 core and 2 core cases.

Deqdip 1-Core Test Results

1 Core	Real	User	System
Caches Pre-Warmed	**0m0.046s**	0m0.008s	0m0.038s
Deqdip Decache Cold	**0m0.061s**	0m0.021s	0m0.040s

This shows a `deqdip` decache produced the desired slower results across the board.

Deqdip 2-Core Test Results

2 Core	Real	User	System
Caches Pre-Warmed	**0m0.340s**	0.020s	0.044s
Deqdip Decache Cold	**0m0.719s**	0.025s	0.041s

Again after `deqdip` decache the test produces the desired slower results. Note the metric for system time on a 2-Core cold cache is 3/1000ths of a second faster versus a 2-Core warm cache whereas in both cases for the 1-Core runs they were predictably almost similar. This may just be an attribute of the way Linux sort works and is not a matter of subject.

Conclusion

A Decache protocol was designed, developed, implemented and validated and provides a convenient way to carry out performance characterisation tasks. In principle it trades slower overall run times for greater deterministic predictability.

In addition a C user level API and an Elixir (Erlang) API have been provided.

The `deqdip` package available on my github user account `cyanotype1` provides the following components

- A Decache protocol for x86_64 processors running Linux implemented via:-
- A Linux Kernel Loadable Module
- A Linux System device file
- An Erlang NIF dynamic library to steer Elixir API requests to the Kernel Module
- An Elixir API `deqdipctl_proto()/0` to call the decache protocol