The Quantum Computing Company™

Measuring Computation Time on D-Wave Systems

09-1107A-E

CONTENTS

CHAPTER

ONE

ABOUT THIS DOCUMENT

1.1 Intended Audience

This document is for users of the D-Wave™ quantum computer system who want to better understand and leverage the physical implementation of the quantum processing unit (QPU) architecture.

1.2 Scope

This document describes the computation process, in the context of system *timing*, on D-Wave quantum computers. It explains the overall service time that is allocated to a problem, describes how the QPU is accessed within that period, and lists the timing-related fields that are available in the Solver API (SAPI)—including those that allow for user control.

1.3 Related Documents

See also the following related documents, all of which are available on the Qubist web user interface:

- *Technical Description of the D-Wave Quantum Processing Unit*
- *Postprocessing Methods on D-Wave Systems*
- C, MATLAB, and Python developer guides, which describe the application programming interfaces (APIs) used for programming a D-Wave system

In addition, QPU specifications are available per system; contact D-Wave at dwsupport@dwavesys.com for this information.

CHAPTER

TWO

D-WAVE TIMING OVERVIEW

Fig. 2.1 shows a simplified diagram of the computation workflow on a D-Wave system. Each problem consists of a single input together with parameters specifying, for example, the number of reads and whether postprocessing (PP) should be invoked. A problem is sent across a network to the SAPI server and joins a queue. Queued problems are assigned to workers, which can run in parallel. A worker prepares the problem for the quantum processing unit (QPU) and optionally for postprocessing (PP), sends the problem to the QPU queue, receives results, and bundles the results with additional information about the problem (such as runtime information).

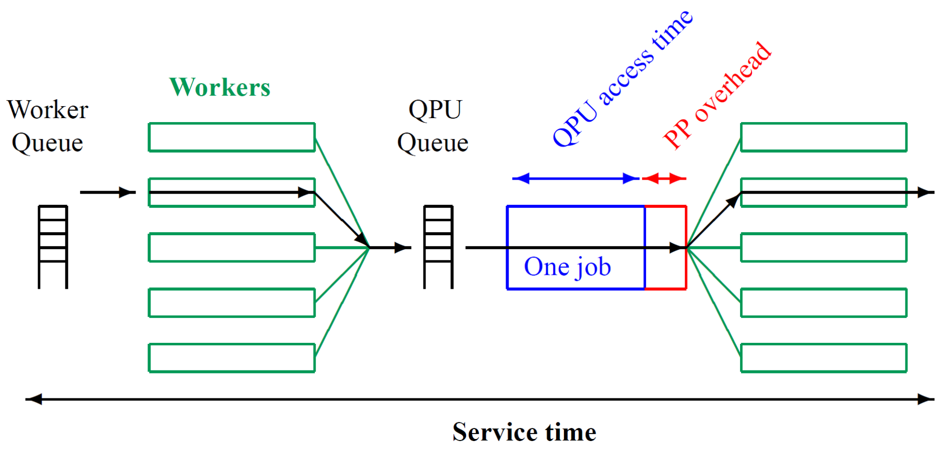

Fig. 2.1: Overview of computation workflow on a D-Wave system.

The total time for a problem to pass through the system is the service time. Note that service time includes wait time in two queues, which can vary according to the number of problems and workers active at any moment. The QPU handles problems one at a time; the time for one problem to move through the QPU is the QPU access time.

The timing fields and values discussed here are returned by the result() and status() functions of a submitted problem object. *QPU Access Time* gives a simple Python example invoking these functions. The timing keywords described in this document are identical in all supported programming languages. For more information about programming and timing structures, see the Solver API and documentation available on the Qubist web user interface.

QPU Access Time describes time components and values related to QPU access time, which is returned by the result() function. *Service Time* describes timer values related to service time, returned by the status() function, and postprocessing time, returned by the result() function.

2.1 Sources of Timer Errors

Although wall clock timers on modern computer systems typically report time in units of nanoseconds, those times may only be accurate to within a few microseconds. Furthermore, as Fig. 2.2 illustrates, running a given code block several times can yield different runtime measurements, even though the instruction execution sequence does not change. Runtime distributions with occasional large outliers, as seen here, are not unusual. Timing variations are caused by noise from the operating system due to such things as process scheduling, memory management, power management, and the runtime environment (e.g., garbage collection).

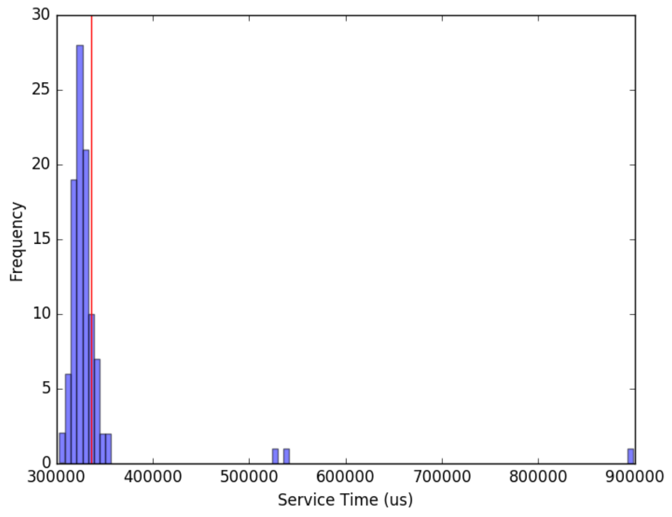

Fig. 2.2: Histogram of 100 measurements of service time using a wall clock timer. As shown here, wall clock timers in modern computer systems can be unreliable, occasionally returning unusually high values. As a result, mean runtimes tend to overestimate typical runtimes. In this example, the mean time of 336.5 ms (red line) is higher than 75 percent of the measurements. One measurement is about 3 times higher and two measurements are almost 2 times higher than typical.

For these reasons, mean reported runtimes can often be higher than typical runtimes: for example in Fig. 2.2, the mean time of 336.45 ms is higher than 75 percent of the measured runtimes. In this context the smallest time recorded for a single process is considered the most accurate, because noise from outside sources can only increase elapsed time.[1] Because system activity increases with the number of active problems, the most accurate times for a single process are obtained by measuring on an otherwise empty system.

Timer latency may impose additional measurement errors, since querying the system clock can add microseconds to elapsed time. To reduce the impact of timer latency, a given code block may be measured inside a loop that iterates many times, with running time calculated as the average time per iteration. Because of system noise and timer latency, component times measured one way may not add up to total times measured another way; see *[Bryant2010]* for details. These sources of timer error are present on all computer systems, including D-Wave platforms. Normal timer variation as described here may occasionally yield atypical and imprecise results; also, one expects wall clock times to vary with the particular system configuration and with system load.

[1] A more common practice in computational research is to report an alternative measurement called CPU time, which is intended to filter out operating system noise. However, CPU timers are only accurate to tens of milliseconds, and CPU times are not available for QPU time measurements. For consistency, we use wall clock times throughout.

CHAPTER

THREE

QPU ACCESS TIME

3.1 Timing Data Returned by SAPI

Below is a sample of Python code for accessing timing data returned by SAPI. Timing values are returned in the timing object and the status object. The timing object referenced on line 10 is a Python dictionary containing (key, value) pairs. The keys match keywords discussed in this section.

```
01 from dwave_sapi2.local import local_connection
02 from dwave_sapi2.core import async_solve_ising, await_completion
03 import datetime

04 h,J = build_hamiltonian()
05 solver = local_connection.get_solver('example_solver')
06 submitted_problem = async_solve_ising(solver, h, J, num_reads=100)
07 await_completion([submitted_problem], min_done=2, timeout=1.0)

08 # QPU and PP times
09 result = submitted_problem.result()
10 timing = result['timing']

11 # Service time
12 time_format = "%Y-%m-%dT%H:%M:%S.%fZ"
13 status = submitted_problem.status()
14 start_time = datetime.datetime.strptime(status ['time_received'], time_format)
15 finish_time = datetime.datetime.strptime(status['time_solved'], time_format)
16 service_time = finish_time - start_time
```

3.2 Timing-Related Keywords in SAPI

The following keywords are deprecated in SAPI 2.4 are unavailable in subsequent SAPI releases.

Table 3.1: Deprecated Keywords

Deprecated Keyword	New Keyword
total_real_time[1]	qpu_access_time
run_time_chip	qpu_sampling_time
anneal_time_per_run	qpu_anneal_time_per_sample
readout_time_per_run	qpu_readout_time_per_sample

[1] Also called EHART.

Instead, use the new keywords shown in the table below. These keywords are identical in all programming languages supported by SAPI.

Table 3.2: Timing Values Supported as of SAPI Release 2.4

Value	Keyword in SAPI	Meaning
T	qpu_access_time	Total time in QPU
T_p	qpu_programming_time	Total time to program the QPU[2]
Δ		Additional low-level operations.
R		Number of reads/samples
T_s	qpu_sampling_time	Total time for R samples
T_a	qpu_anneal_time_per_sample	Time for one anneal
T_r	qpu_readout_time_per_sample	Time for one read
T_d	qpu_delay_time_per_sample	Delay time between anneals[3]

3.3 User-Specified Timing Parameters

The following user-specified parameters affect timing:

- `num_reads`—Number of states (output solutions) to read from the solver per programming cycle.
- `annealing_time`—Duration, in microseconds, of quantum annealing time. This value populates the `qpu_anneal_time_per_sample` field returned.
- `programming_thermalization`—Number of microseconds to wait after programming the QPU to allow it to cool back to base temperature; i.e., *post-programming thermalization time*. Values lower than the default speed up solving at the expense of solution quality. This value contributes to the total T_p.
- `readout_thermalization`—Number of microseconds to wait after each state is read from the QPU to allow it to cool back to base temperature; i.e., *post-readout thermalization time*. This optional value contributes to the `qpu_delay_time_per_sample` value returned.

Note: While still supported in SAPI Release 2.10, the `readout_thermalization` parameter is deprecated and will eventually be removed from the API. Plan code updates accordingly.

[2] The user-specified value, programming_thermalization, affects T_p. However, even if programming_thermalization is zero, T_p is several milliseconds for other operations.

[3] The time returned in the qpu_delay_time_per_sample field is equal to a constant plus the user-specified value, readout_thermalization.

3.4 Breakdown of QPU Access Time

As illustrated in Fig 3.1, the time T to process a single problem, called qpu_access_time in SAPI, is broken into a one-time initialization step to program the QPU; initialization also includes a small amount of time (denoted Δ) for low-level operations that are not reported separately. Initialization is followed by a sampling step used to generate R sample solutions to the input.

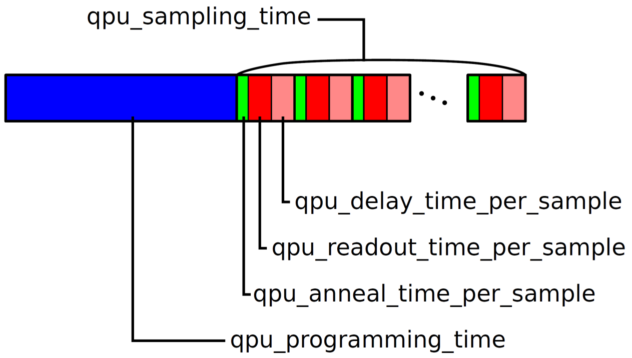
qpu_sampling_time
qpu_delay_time_per_sample
qpu_readout_time_per_sample
qpu_anneal_time_per_sample
qpu_programming_time

Fig. 3.1: QPU access time has two parts, programming time (blue) and sampling time (multicolor); a small amount of additional time (Δ) for low-level operations is not shown. Sampling time is the time for some number R of samples, where each sample time is divided into anneal time (green), readout time (red), and delay time (pink).

Total time for one sample is further broken into anneal, readout, and thermalization times. Note that these component times are calculated as average times per sample. Possible rounding errors mean that the sum of these times may not match the total sampling time reported. Thus, these times relate as follows.

$$\begin{aligned} T &= T_p + \Delta + T_s \\ T_s/R &\approx T_a + T_r + T_d \end{aligned}$$

3.5 Upper Limit on User-Specified Timing-Related Parameters

The D-Wave system limits your ability to submit long-running problems to prevent you from inadvertently monopolizing QPU time. This limit varies by D-Wave product model.

The limit is calculated according to the following formula:

$$Duration = ((P_1 + P_2) * P_3) + P_4 \qquad (3.1)$$

where P_1, P_2, P_3, and P_4 are the values specified for the annealing_time, readout_thermalization, num_reads, and programming_thermalization parameters, respectively.

If you attempt to submit a problem that exceeds the limit for your system, an error is returned showing the values in microseconds. For example:

```
ERROR: Problem run duration is too long: 4010000 > 3000000
```

Note that it is possible to specify values that fall within the permitted ranges for each individual parameter, yet together cause the problem to surpass the limit.

CHAPTER

FOUR

SERVICE TIME

Apart from QPU access time, the total service time can be broken into:

- Postprocessing (PP) time
- Any additional time required by the worker before and after QPU access
- Wait time in queues

Postprocessing time is the only one of these three components that is reported separately. For more details about postprocessing and how it is handled in the timing structure, see *Postprocessing Methods on D-Wave Systems*, available on the Qubist web user interface.

4.1 Postprocessing Time

The user has the option to apply one or more postprocessing utilities to the raw results returned by the QPU. As shown in Fig. 4.1, postprocessing (red) works in tandem with sampling (blue), so that the computation times overlap. In this diagram, time intervals associated with small batches of samples are marked by vertical blue lines. Samples from the QPU are returned in batches like these; each batch is postprocessed before the next batch is ready.

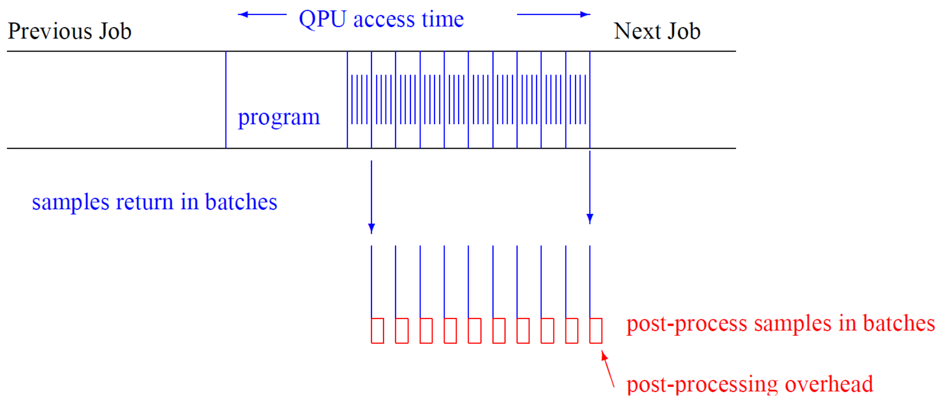

Fig. 4.1: Relationship of QPU time to postprocessing time. A sequence of problem enters the QPU (previous, current, next). Postprocessing for the current problem (red boxes) takes place concurrently with the sampling step, and is applied to batches of samples as they are returned by the QPU (blue slices). The additional overhead time for postprocessing is never more than the time to postprocess the last batch, and postprocessing does not impose any delay on starting the next problem.

Postprocessing overhead does not impose any delay to QPU access for the next problem, because it takes place concurrently with the next problem's programming time.

The system returns two associated timing values, as shown in the table below. Referring to Fig. 4.1, `total_post_processing_time` is the sum of all times in the red boxes, while `post_processing_overhead` is the extra time needed (a single red box) to process the last batch. This latter time together with `qpu_access_time` contributes to overall service time.

Note: Even if no postprocessing is run on a problem, the returned `post_processing_overhead` value is non-zero. This is because the process to compute the final energies occurs after results are returned and therefore becomes part of the postprocessing overhead.

Keyword in SAPI	Meaning
total_post_processing_time	Total time for postprocessing
post_processing_overhead_time	Added for the last batch

4.2 Service Time Per Problem

Service time is defined as the difference between time-in and time-out for each problem, as shown in the table.

Keyword in SAPI	Meaning
time_received	When problem arrives
time_solved	When bundled results are available

As mentioned in the introduction, service time for a single problem depends on the system load; that is, how many other problems are present at a given time. During periods of heavy load, wait time in the two queues may contribute to increased service times. D-Wave has no control over system load under normal operating conditions. Therefore, it is not possible to guarantee that service time targets can be met. Service time measurements described in other D-Wave documents are intended only to give a rough idea of the range of experience that might be found under varying conditions. Service times are typically measured under two different scenarios:

- A sequential workflow test measures service time for batches of problems to move through the system, assuming that they arrive sequentially and are solved by a single worker, with no contention for the QPU.
- A concurrent workflow test measures service times for problems that are served in parallel by a number of workers. Under this scenario the wait time in the first queue decreases as workers are added, but the wait time for QPU access increases due to contention.

BIBLIOGRAPHY

[Bryant2010] Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective (2nd Edition)*, Pearson, 2010.