FTQ/FWQ

Summary Version

1.1

Purpose of Benchmark

The FTQ and FWQ benchmarks measure hardware and software interference or 'noise' on a node from the applications perspective. The benchmarks are used within the Sequoia RFP to a measure compute node hardware and lightweight kernel (LWK) application perceived interference or noise. Sequoia SOW Section 3.2.4 gives the criteria, using the output of the FWQ benchmark measurements to characterize an LWK as meeting the requirement for a "diminutive noise" LWK.

Characteristics of Benchmark

FWQ (fixed work quanta) and FTQ (fixed time quanta) run on each core and hardware thread within a single node via pthreads. FWQ repetitively performs a fixed amount of work (the work quanta), measuring the time necessary to complete that work. FTQ repetitively works for a fixed amount of time (the time quanta), measuring the amount of work that is performed. Both repeat the measurements multiple times (under command line control). For FTQ, the work performed is very simple—incrementing a variable repeatedly. For FWQ there are three work choices. The default is the same as FTQ. FWQ outputs a time file, one per thread. FTQ outputs two files (per thread)—one with the amount of work performed (one per line), the other with the time quantums (one per line).

Due to the fixed work approach of FWQ, the data samples can be used to compute useful statistics (mean, standard deviation and kurtosis) of the scaled noise (sample time minus the minimum work time and scaled by the minimum work time). These statistics then can be used to characterize and categorize the amount of noise in a hardware and software environment. Due to the fixed time quanta approach of FTQ, the work data can be processes with a Fast Fourier Transform (FFT) in order to determine the temporal frequency of software interference. This is an extremely useful tool to find sources of periodic interference such as scheduling intervals, the regular operation of daemons, etc.

Together, FWQ and FTQ are effective benchmarks to measure operating system interruptions or other disturbances of consistency in cycles delivered to the running threads.

Mechanics of Building Benchmark

FTQ/FWQ are simple to build. It has been tested on Linux, MacOSX, and even under Windows XP/Vista via cygwin (although the thread affinity system calls must be removed). There is no configuration step at present, although it is likely that one may emerge if the FTQ/FWQ configuration options become sophisticated enough. The code directory structure is as follows:

`common/` Files common to any benchmark code in here. This currently contains platform independent, high precision timers. The cycle.h file is the redistributable timer header from the FFTW package.

`ftq/` The fixed work quantum and fixed time quantum microbenchmarks. For a reference on this, see the Sottile/Minnich paper at Cluster 2004, or ask Matt for a

copy of this dissertation. The versions included here are the standard sequential version, a threaded version with Pthreads, and an OpenMP version.

`results/` Put your results here.

`octave/` Scripts for analyzing the data.

The only wrinkle in porting FTQ/FWQ to another platform may be the timers. For uncommon systems, one may need to augment the many systems currently configured in the `common/cycle.h` file.

To compile single threaded versions of the benchmarks, type:

```
% make single
```

You should observe two executables being created: ftq and fwq. Both ftq and fwq have been tuned so that granularity work quanta used for sampling is much larger than instruction issue rate fluxuations. For example, by unsetting the parameter MULTIITER in the source files, fwq/ftq uses a single integer increment operation as the work quantum, while the default fwq/ftq source uses 32 increments followed by 31 decrements (effectively equivalent to the single ftq count++). The reason is that one will observe, especially for tiny work quanta, jitter in the data because the majority (approx 70–80%) of the instructions executed per work quantum are actually branch and conditional operations. This causes some level of fluctuation in how long a work quantum takes to execute simply because of the structure of the instruction stream. As the work quantum increases in granularity, the proportion of work quantum instructions to control flow and conditional structures becomes more even, and the fluctuations within the data begin to be due to interference external to FTQ itself (other than apparent cache related, high frequency low level perturbations).

For FWQ it is important that the "fixed work" performed does not make any memory references: especially for systems with non-uniform memory access (NUMA) architectures such as AMD Opteron and upcoming Intel Nehalem based multisocket systems. The default work loop for FWQ, as mentioned above, increments a variable declared "`register long long count`". However, gcc V4.3 (and many earlier versions we have tested) with the default "`-g`" optimization in the makefile does not allocate "`count`" to a register as one would expect, but rather allocates it in a memory location. This introduces variation or fuzz in the fixed work sample runtime due to memory reference latency variations. However, when one turns on "`-O1`" or higher optimization levels, the work loop is eliminated by the compiler optimizer entirely! To get around this problem, we wrote an x86-64 in 64b mode assembly language work loop that only uses registers `and inc, cmp, js` and `nop` instructions. To use this assembly language implementation of the fixed work, include the `-DASMx8664` compile line option. There is a third alternative fixed work loop implementation that is based on a DAXPY vector update operation (of default size 1,024 64b elements). This also suffers from memory latency noise problem, but increases the work to minimize this hardware noise effect. This alternative can be used by including the `–DDAXPY` compile line option.

To compile a version of FWQ/FTQ that runs on a multicore or SMP environment where each thread runs the work loop of the benchmark (resulting in multiple files when run, one data set per thread), you can invoke:

```
% make threads
```

This yields a similar set of executables, except with t_ as a prefix on the executable name, indicating the pthreads version. It is not recommended that you use the threaded versions on a single core.

Note the code uses `sched_setaffinity()` and `pthread_setaffinity_np()` to bind the main process and the threads to cores. If your platform does not support this call, you should replace this section of code with similar functionality.

To compile threaded and single versions of the FTQ/FWQ benchmark, invoke:

```
% make
```

or

```
% make all
```

Mechanics of Running Benchmark

The simplest way to get FWQ/FTQ going is to just try it out with a small amount of work per sample (`-w` option) and small sample count (`-n` option) and observe what the data looks like. Start with the single threaded version and then when the behavior of that is well understood, then try the multi-threaded version. FWQ/FTQ has just a few parameters. Although it might seem strange to have the work quantum granularity as a compile time option, this is done so that the instruction stream is not polluted with conditionals enforcing a run-time determined work quantum.

The parameters are as follows:

`-s` Dump output to the STDOUT
`-o` Prefix for output data file names
`-i` Bits in sampling interval limits (FTQ only). Number of loop iterations is 2^i.
`-w` Bits in work amount (FWQ only). Number of loop iterations is 2^w.
`-n` Number of samples to take
`-h` Usage

Consider a simple run like this:

```
% ./ftq -o testrun -i 20 -n 1000
```

What does this simple run produce? If we look at the current directory, we should see two new files: testrun_counts.dat and testrun_times.dat. These are the output from FTQ.

The testrun_times.dat file contains the end times of each sample as reported by the high precision timers in cycle.h. We should see 1000 lines for the above case, corresponding to the end times for each of the 1000 samples executed. Note that if one subtracts time k from time k+1, one can observe the actual sampling interval that the k+1'th sample required. Note also that the first sample has no start time and is generally thrown out without losing any really valuable information.

The testrun_counts.dat file contains the real data from FTQ. Each of the 1000 entries represents the number of work quanta (in this case, 32 increment operations followed by 31 decrements per work quantum) executed in the time period ending at the corresponding time in the testrun_times.dat file.

The final unexplained argument is `-i 20` or `-w 20`. The sampling intervals that FTQ uses are determined using a bitmask and some simple tricks to deal with long samples that must be compensated for. The `20` is the number of bits that the desired sampling period requires in terms of processor cycles (so even though it may seem large, it isn't). In other words, for the case above, `-i 20`, a single sample period will span at most $(2^{20})-1$ processor cycles. On a 2.8-GHz processor, this corresponds to a sample period of approximately 347 μs or 0.347 ms.

For FWQ the amount of work in core cycles per work loop per sample period on an x86_64 processor is given in the table below. To determine the runtime take the sample work amount (2^{20} in this case) times the number of cycles per work loop times the cycle time of the processor. For the default case, this is 2^{20}x380/2.8e9 = 0.142 sec per sample. A better sample period is near 1 μs. For this case, one should choose w = $\ln_2$(0.001*2.8e9/380) = 12.8 (choose w = 12 or 13).

Compile Option	Compile Optimization	Processor cycles per work loop per sample
-DASMx8664	O1	6.4
Default	-g	380
-DDAXPY	-O3 (with vectorization)	2075

The threaded versions of the code have one additional argument, `-t threads`, which is used to specify the number of threads to be executed. In the case of a multithreaded run, instead of just getting files like ftq_times.dat and ftq_counts.dat, you will get a pair of files for each thread. The thread IDs range from 0 to numthreads−1, so for a two threaded run you will see:

```
ftq_0_counts.dat
ftq_0_times.dat
ftq_1_counts.dat
ftq_1_times.dat
      and
fwq_0_times.dat
fwq_1_times.dat
```

Verification of Results

Using the resulting data from FWQ/FTQ, you can use tools you likely already have on your desktop system to manipulate that data.

With GNU Octave on your desktop system and using the data from testrun_counts.dat, you can run the following sequence of commands.

1. Change the working directory to where the data is, e.g., data_dir.

```
octave:1> cd 'data_dir'
```

2. Add the path to the ftq/Octave directory to the search list. This will allow Octave to find the various .m files needed below.

```
octave:2> addpath('path name to ftq/Octave directory')
```

3. Set up the files to load.

```
octave:3> d=dir('ftq_*_times.dat');
```

4. Load the FTQ time output files, process the data, and print out the statistics.

```
octave:4> [data,xb,sd,k]=analyze('.',d);
Loading data...
fwq_0_times.dat
fwq_1_times.dat
fwq_2_times.dat
fwq_3_times.dat

Computing scaled noise...
Computing kurtosis and skewness values...
Min, Max=
 6715191
 93270283

Mean=
   0.0119197   0.0073124   0.0070419   0.0083376

StdDev=
   0.0166154   0.0121701   0.0080184   0.0091711

Knum=
   1.0678e+03   1.7122e+00   3.5251e-03   7.9641e-01

Kden=
   1.2195e-02   3.5099e-03   6.6140e-04   1.1319e-03

kurtosis=
   8.7565e+04   4.8483e+02   2.3297e+00   7.0061e+02

skewness=

   255.4680    17.0578     1.4652     8.0548
```

5. Plot all the data and use 100 bins for the histograms.

```
octave:5> plotter(data,xb,sd,k,1,100);
```

Two windows should pop up, one with the scaled noise for each output file and one with the histogram of the noise for each output file.

A similar interaction with Octave for loading, analyzing, and plotting the data is:

```
octave:1> cd 'data_dir'                                    Same
octave:2> addpath('path name to ftq/Octave directory')     Same
octave:3> d=dir('fwq_*_times.dat');                  <-- fwq
octave:4> [data,xb,sd,k]=wanalyze('.',d);            <-- wanalyze
octave:5> plotter(data,xb,sd,k,1,100);                     Same
```

Note that a “diminutive noise environment” is one that produces FWQ time samples with small maximum mean of the scaled noise (`xb<1.0e-6`), small maximum standard deviation of scaled noise (`sd<1.0e-3`), and small maximum kurtosis (`k<100`).