

X-Mem

2.1.13

Generated by Doxygen 1.8.6

Wed Apr 22 2015 18:52:58

Contents

1	README	1
2	Hierarchical Index	7
2.1	Class Hierarchy	7
3	Class Index	9
3.1	Class List	9
4	File Index	11
4.1	File List	11
5	Class Documentation	13
5.1	xmem::Parser::Action Struct Reference	13
5.1.1	Member Function Documentation	13
5.1.1.1	finished	13
5.1.1.2	perform	13
5.2	xmem::Arg Struct Reference	14
5.2.1	Detailed Description	14
5.3	xmem::Benchmark Class Reference	15
5.3.1	Detailed Description	17
5.3.2	Constructor & Destructor Documentation	17
5.3.2.1	Benchmark	17
5.3.3	Member Function Documentation	17
5.3.3.1	_run_core	17
5.3.3.2	_start_power_threads	18
5.3.3.3	_stop_power_threads	18
5.3.3.4	getAverageDRAMPower	18
5.3.3.5	getAverageMetric	18
5.3.3.6	getChunkSize	18
5.3.3.7	getCPUNode	18
5.3.3.8	getIterations	19
5.3.3.9	getLen	19
5.3.3.10	getMemNode	19

5.3.3.11	getMetricOnlter	19
5.3.3.12	getMetricUnits	19
5.3.3.13	getName	19
5.3.3.14	getNumThreads	20
5.3.3.15	getPatternMode	20
5.3.3.16	getPeakDRAMPower	20
5.3.3.17	getRWMode	20
5.3.3.18	getStrideSize	20
5.3.3.19	hasRun	20
5.3.3.20	isValid	20
5.3.3.21	run	21
5.3.4	Member Data Documentation	21
5.3.4.1	_average_dram_power_socket	21
5.3.4.2	_averageMetric	21
5.3.4.3	_chunk_size	21
5.3.4.4	_cpu_node	21
5.3.4.5	_dram_power_readers	21
5.3.4.6	_dram_power_threads	21
5.3.4.7	_hasRun	21
5.3.4.8	_iterations	21
5.3.4.9	_len	21
5.3.4.10	_mem_array	22
5.3.4.11	_mem_node	22
5.3.4.12	_metricOnlter	22
5.3.4.13	_metricUnits	22
5.3.4.14	_name	22
5.3.4.15	_num_worker_threads	22
5.3.4.16	_obj_valid	22
5.3.4.17	_pattern_mode	22
5.3.4.18	_peak_dram_power_socket	22
5.3.4.19	_rw_mode	22
5.3.4.20	_stride_size	22
5.3.4.21	_warning	23
5.4	xmem::BenchmarkManager Class Reference	23
5.4.1	Detailed Description	23
5.4.2	Constructor & Destructor Documentation	23
5.4.2.1	BenchmarkManager	23
5.4.3	Member Function Documentation	23
5.4.3.1	runAll	23
5.4.3.2	runLatencyBenchmarks	24

5.4.3.3	runThroughputBenchmarks	24
5.5	xmem::Configurator Class Reference	24
5.5.1	Detailed Description	25
5.5.2	Member Function Documentation	25
5.5.2.1	configureFromInput	25
5.5.2.2	extensionsEnabled	27
5.5.2.3	getIterationsPerTest	27
5.5.2.4	getNumWorkerThreads	27
5.5.2.5	getOutputFilename	27
5.5.2.6	getStartingTestIndex	27
5.5.2.7	getWorkingSetSizePerThread	27
5.5.2.8	isNUMAEnabled	28
5.5.2.9	latencyTestSelected	28
5.5.2.10	runExtDelayInjectedLoadedLatencyBenchmark	28
5.5.2.11	setUseOutputFile	28
5.5.2.12	throughputTestSelected	28
5.5.2.13	useChunk32b	28
5.5.2.14	useLargePages	28
5.5.2.15	useOutputFile	29
5.5.2.16	useRandomAccessPattern	29
5.5.2.17	useReads	29
5.5.2.18	useSequentialAccessPattern	29
5.5.2.19	useStrideN1	29
5.5.2.20	useStrideN16	29
5.5.2.21	useStrideN2	30
5.5.2.22	useStrideN4	30
5.5.2.23	useStrideN8	30
5.5.2.24	useStrideP1	30
5.5.2.25	useStrideP16	30
5.5.2.26	useStrideP2	30
5.5.2.27	useStrideP4	30
5.5.2.28	useStrideP8	31
5.5.2.29	useWrites	31
5.5.2.30	verboseMode	31
5.6	xmem::Stats::CountOptionsAction Class Reference	31
5.6.1	Constructor & Destructor Documentation	31
5.6.1.1	CountOptionsAction	31
5.6.2	Member Function Documentation	32
5.6.2.1	perform	32
5.7	xmem::Descriptor Struct Reference	32

5.7.1	Detailed Description	32
5.7.2	Member Data Documentation	33
5.7.2.1	check_arg	33
5.7.2.2	help	33
5.7.2.3	index	33
5.7.2.4	longopt	33
5.7.2.5	shortopt	34
5.7.2.6	type	34
5.8	xmem::ExampleArg Class Reference	34
5.9	xmem::PrintUsagImplementation::FunctionWriter< Function > Struct Template Reference	35
5.10	xmem::PrintUsagImplementation::IStringWriter Struct Reference	35
5.11	xmem::LatencyBenchmark Class Reference	35
5.11.1	Detailed Description	36
5.11.2	Member Function Documentation	36
5.11.2.1	_run_core	36
5.11.2.2	getAvgLoadMetric	37
5.11.2.3	getLoadMetricOnIter	37
5.11.3	Member Data Documentation	37
5.11.3.1	_averageLoadMetric	37
5.11.3.2	_loadMetricOnIter	37
5.12	xmem::LatencyWorker Class Reference	37
5.12.1	Detailed Description	38
5.12.2	Constructor & Destructor Documentation	38
5.12.2.1	LatencyWorker	38
5.13	xmem::PrintUsagImplementation::LinePartIterator Class Reference	38
5.13.1	Member Function Documentation	39
5.13.1.1	next	39
5.13.1.2	nextRow	39
5.13.1.3	nextTable	39
5.14	xmem::PrintUsagImplementation::LineWrapper Class Reference	39
5.14.1	Constructor & Destructor Documentation	40
5.14.1.1	LineWrapper	40
5.14.2	Member Function Documentation	40
5.14.2.1	process	40
5.15	xmem::LoadWorker Class Reference	40
5.15.1	Detailed Description	41
5.15.2	Constructor & Destructor Documentation	41
5.15.2.1	LoadWorker	41
5.15.2.2	LoadWorker	41
5.16	xmem::MemoryWorker Class Reference	41

5.16.1 Detailed Description	42
5.16.2 Constructor & Destructor Documentation	43
5.16.2.1 MemoryWorker	43
5.16.3 Member Function Documentation	44
5.16.3.1 getAdjustedTicks	44
5.16.3.2 getBytesPerPass	44
5.16.3.3 getElapsedDummyTicks	44
5.16.3.4 getElapsedTicks	44
5.16.3.5 getLen	44
5.16.3.6 getPasses	45
5.16.3.7 hadWarning	45
5.16.4 Member Data Documentation	45
5.16.4.1 _adjusted_ticks	45
5.16.4.2 _bytes_per_pass	45
5.16.4.3 _completed	45
5.16.4.4 _cpu_affinity	45
5.16.4.5 _elapsed_dummy_ticks	45
5.16.4.6 _elapsed_ticks	45
5.16.4.7 _len	45
5.16.4.8 _mem_array	45
5.16.4.9 _passes	46
5.16.4.10 _warning	46
5.17 xmem::MyArg Class Reference	46
5.18 xmem::Option Class Reference	46
5.18.1 Detailed Description	48
5.18.2 Constructor & Destructor Documentation	48
5.18.2.1 Option	48
5.18.2.2 Option	48
5.18.3 Member Function Documentation	48
5.18.3.1 append	48
5.18.3.2 count	48
5.18.3.3 first	49
5.18.3.4 isFirst	49
5.18.3.5 isLast	49
5.18.3.6 last	49
5.18.3.7 next	49
5.18.3.8 nextwrap	49
5.18.3.9 operator const Option *	50
5.18.3.10 operator Option *	50
5.18.3.11 operator=	50

5.18.3.12 prev	50
5.18.3.13 prevwrap	50
5.18.3.14 type	50
5.18.4 Member Data Documentation	51
5.18.4.1 arg	51
5.18.4.2 desc	51
5.18.4.3 name	51
5.18.4.4 namelen	51
5.19 xmem::PrintUsageImplementation::OStreamWriter< OStream > Struct Template Reference	52
5.20 xmem::Parser Class Reference	52
5.20.1 Detailed Description	53
5.20.2 Constructor & Destructor Documentation	54
5.20.2.1 Parser	54
5.20.3 Member Function Documentation	55
5.20.3.1 error	55
5.20.3.2 nonOptions	55
5.20.3.3 nonOptionsCount	55
5.20.3.4 optionsCount	55
5.20.3.5 parse	56
5.21 xmem::PowerReader Class Reference	57
5.21.1 Detailed Description	59
5.21.2 Constructor & Destructor Documentation	59
5.21.2.1 PowerReader	59
5.21.3 Member Function Documentation	59
5.21.3.1 calculateMetrics	59
5.21.3.2 clear	59
5.21.3.3 clear_and_reset	59
5.21.3.4 getAveragePower	60
5.21.3.5 getLastSample	60
5.21.3.6 getNumSamples	60
5.21.3.7 getPeakPower	60
5.21.3.8 getPowerTrace	60
5.21.3.9 getPowerUnits	60
5.21.3.10 getSamplingPeriod	61
5.21.3.11 name	61
5.21.3.12 stop	61
5.21.4 Member Data Documentation	61
5.21.4.1 _average_power	61
5.21.4.2 _cpu_affinity	61
5.21.4.3 _name	61

5.21.4.4	_num_samples	61
5.21.4.5	_peak_power	61
5.21.4.6	_power_trace	61
5.21.4.7	_power_units	62
5.21.4.8	_sampling_period	62
5.21.4.9	_stop_signal	62
5.22	xmem::PrintUsagelImplementation Struct Reference	62
5.22.1	Member Function Documentation	62
5.22.1.1	isWideChar	62
5.23	xmem::Runnable Class Reference	63
5.23.1	Detailed Description	63
5.23.2	Member Function Documentation	64
5.23.2.1	_acquireLock	64
5.23.2.2	_releaseLock	65
5.24	xmem::Stats Struct Reference	65
5.24.1	Detailed Description	66
5.24.2	Constructor & Destructor Documentation	66
5.24.2.1	Stats	66
5.24.3	Member Function Documentation	66
5.24.3.1	add	66
5.24.4	Member Data Documentation	67
5.24.4.1	buffer_max	67
5.24.4.2	options_max	67
5.25	xmem::Parser::StoreOptionAction Class Reference	67
5.25.1	Constructor & Destructor Documentation	67
5.25.1.1	StoreOptionAction	67
5.25.2	Member Function Documentation	68
5.25.2.1	finished	68
5.25.2.2	perform	68
5.26	xmem::PrintUsagelImplementation::StreamWriter< Function, Stream > Struct Template Reference	68
5.27	xmem::PrintUsagelImplementation::SyscallWriter< Syscall > Struct Template Reference	69
5.28	xmem::PrintUsagelImplementation::TemporaryWriter< Temporary > Struct Template Reference	69
5.29	xmem::Thread Class Reference	70
5.29.1	Detailed Description	70
5.29.2	Constructor & Destructor Documentation	70
5.29.2.1	Thread	70
5.29.2.2	~Thread	70
5.29.3	Member Function Documentation	71
5.29.3.1	cancel	71
5.29.3.2	completed	71

5.29.3.3	create_and_start	71
5.29.3.4	created	71
5.29.3.5	getExitCode	71
5.29.3.6	getTarget	71
5.29.3.7	isThreadRunning	71
5.29.3.8	isThreadSuspended	72
5.29.3.9	join	72
5.29.3.10	started	72
5.29.3.11	validTarget	72
5.30	xmem::ThroughputBenchmark Class Reference	72
5.30.1	Detailed Description	73
5.30.2	Member Function Documentation	73
5.30.2.1	_run_core	73
5.31	xmem::Timer Class Reference	73
5.31.1	Detailed Description	73
5.31.2	Member Function Documentation	74
5.31.2.1	get_ns_per_tick	74
5.31.2.2	get_ticks_per_ms	74
5.31.3	Member Data Documentation	74
5.31.3.1	_ns_per_tick	74
5.31.3.2	_ticks_per_ms	74
6	File Documentation	75
6.1	src/Benchmark.cpp File Reference	75
6.1.1	Detailed Description	75
6.2	src/benchmark_kernels.cpp File Reference	75
6.2.1	Detailed Description	75
6.3	src/BenchmarkManager.cpp File Reference	76
6.3.1	Detailed Description	76
6.4	src/common.cpp File Reference	76
6.4.1	Detailed Description	76
6.5	src/Configurator.cpp File Reference	77
6.5.1	Detailed Description	77
6.6	src/ext/DelayInjectedLoadedLatencyBenchmark/delay_injected_benchmark_kernels.cpp File Reference	77
6.6.1	Detailed Description	77
6.7	src/ext/DelayInjectedLoadedLatencyBenchmark/DelayInjectedLoadedLatencyBenchmark.cpp File Reference	77
6.7.1	Detailed Description	77
6.8	src/include/Benchmark.h File Reference	78
6.8.1	Detailed Description	78

6.9	src/include/benchmark_kernels.h File Reference	78
6.9.1	Detailed Description	80
6.10	src/include/BenchmarkManager.h File Reference	80
6.10.1	Detailed Description	81
6.11	src/include/common.h File Reference	81
6.11.1	Detailed Description	83
6.11.2	Macro Definition Documentation	83
6.11.2.1	BENCHMARK_DURATION_MS	83
6.11.2.2	DEFAULT_LARGE_PAGE_SIZE	83
6.11.2.3	DEFAULT_NUM_L1_CACHES	83
6.11.2.4	DEFAULT_NUM_L2_CACHES	83
6.11.2.5	DEFAULT_NUM_L3_CACHES	83
6.11.2.6	DEFAULT_NUM_L4_CACHES	83
6.11.2.7	DEFAULT_NUM_LOGICAL_CPUS	84
6.11.2.8	DEFAULT_NUM_NODES	84
6.11.2.9	DEFAULT_NUM_PHYSICAL_CPUS	84
6.11.2.10	DEFAULT_NUM_PHYSICAL_PACKAGES	84
6.11.2.11	DEFAULT_NUM_WORKER_THREADS	84
6.11.2.12	DEFAULT_PAGE_SIZE	84
6.11.2.13	DEFAULT_WORKING_SET_SIZE_PER_THREAD	84
6.11.2.14	EXT_DELAY_INJECTED_LOADED_LATENCY_BENCHMARK	84
6.11.2.15	LATENCY_BENCHMARK_UNROLL_LENGTH	84
6.11.2.16	MIN_ELAPSED_TICKS	84
6.11.2.17	POWER_SAMPLING_PERIOD_MS	84
6.11.2.18	THROUGHPUT_BENCHMARK_BYTES_PER_PASS	85
6.11.2.19	USE_OS_TIMER	85
6.11.2.20	USE_TIME_BASED_BENCHMARKS	85
6.12	src/include/Configurator.h File Reference	85
6.12.1	Detailed Description	85
6.13	src/include/ExampleArg.h File Reference	86
6.13.1	Detailed Description	86
6.14	src/include/ext/DelayInjectedLoadedLatencyBenchmark/delay_injected_benchmark_kernels.h File Reference	86
6.14.1	Detailed Description	89
6.15	src/include/ext/DelayInjectedLoadedLatencyBenchmark/DelayInjectedLoadedLatencyBenchmark.h File Reference	89
6.15.1	Detailed Description	89
6.16	src/include/LatencyBenchmark.h File Reference	89
6.16.1	Detailed Description	89
6.17	src/include/LatencyWorker.h File Reference	89

6.17.1 Detailed Description	90
6.18 src/include/LoadWorker.h File Reference	90
6.18.1 Detailed Description	90
6.19 src/include/MemoryWorker.h File Reference	90
6.19.1 Detailed Description	90
6.20 src/include/MyArg.h File Reference	90
6.20.1 Detailed Description	91
6.21 src/include/optionparser.h File Reference	91
6.21.1 Detailed Description	92
6.22 src/include/PowerReader.h File Reference	94
6.22.1 Detailed Description	95
6.23 src/include/Runnable.h File Reference	95
6.23.1 Detailed Description	95
6.24 src/include/Thread.h File Reference	95
6.24.1 Detailed Description	95
6.25 src/include/ThroughputBenchmark.h File Reference	95
6.25.1 Detailed Description	96
6.26 src/include/Timer.h File Reference	96
6.26.1 Detailed Description	96
6.27 src/include/win/win_common_third_party.h File Reference	96
6.27.1 Detailed Description	96
6.28 src/include/win/win_CPdhQuery.h File Reference	96
6.28.1 Detailed Description	96
6.29 src/include/win/WindowsDRAMPowerReader.h File Reference	96
6.29.1 Detailed Description	97
6.30 src/LatencyBenchmark.cpp File Reference	97
6.30.1 Detailed Description	97
6.31 src/LatencyWorker.cpp File Reference	97
6.31.1 Detailed Description	97
6.32 src/LoadWorker.cpp File Reference	97
6.32.1 Detailed Description	97
6.33 src/main.cpp File Reference	98
6.33.1 Detailed Description	98
6.34 src/MemoryWorker.cpp File Reference	98
6.34.1 Detailed Description	98
6.35 src/PowerReader.cpp File Reference	98
6.35.1 Detailed Description	98
6.36 src/Runnable.cpp File Reference	99
6.36.1 Detailed Description	99
6.37 src/Thread.cpp File Reference	99

6.37.1 Detailed Description	99
6.38 src/ThroughputBenchmark.cpp File Reference	99
6.38.1 Detailed Description	99
6.39 src/Timer.cpp File Reference	100
6.39.1 Detailed Description	100
6.40 src/win/win_common_third_party.cpp File Reference	100
6.40.1 Detailed Description	100
6.41 src/win/WindowsDRAMPowerReader.cpp File Reference	100
6.41.1 Detailed Description	100
Index	101

Chapter 1

README

X-Mem: Extensible Memory Benchmarking Tool v2.1.13

The flexible open-source research tool for characterizing memory hierarchy throughput, latency, and power.

Originally authored by Mark Gottscho (Email: mgotttscho@ucla.edu) as a Summer 2014 intern at Microsoft Research, Redmond, WA.

This project is under active development. Stay tuned for more updates.

PROJECT REVISION DATE: April 22, 2015.

LICENSE

The MIT License (MIT)

Copyright (c) 2015 Microsoft

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

FEATURES

This tool is provided as open source with the hope of being useful to the broader research and development community. Here are some of X-Mem's features.

Flexibility: Easy reconfiguration for different combinations of tests

- Working sets in increments of 4KB, allowing cache up to main memory-level benchmarking
- NUMA support
- Multi-threading support
- Large page support

Extensibility: modularity via C++ object-oriented principles

- Supports rapid addition of new benchmark kernel routines
- Example: stream triad algorithm, impact of false sharing, etc. are possible with minor changes

Cross-platform: Currently implemented for Windows and GNU/Linux on x86-64, x86-64 with AVX extensions CPUs

- Designed to allow straightforward porting to other operating systems and ISAs
- 32-bit x86 port under development
- ARM port under development

Memory throughput:

- Accurate measurement of sustained memory throughput to all levels of cache and memory
- Regular access patterns: forward & reverse sequential as well as strides of 2, 4, 8, and 16 words
- Random access patterns
- Read and write
- 32, 64, 128, 256-bit width memory instructions where applicable on each architecture

Memory latency:

- Accurate measurement of round-trip memory latency to all levels of cache and memory
- Loaded and unloaded latency via use of multithreaded load generation

Memory power:

- Support custom power instrumentation through a simple interface that end-users can implement
- Can collect DRAM power via custom driver exposed in Windows performance counter API

Documentation:

- Extensive Doxygen source code comments, PDF manual, HTML

INCLUDED EXTENSIONS (under src/include/ext and src/ext directories):

- Loaded latency benchmark variant with load delays inserted as nop instructions between memory instructions. This is done for 64 and 256-bit chunks on x86-64 with AVX extensions, forward sequential read load threads only at the moment.

For feature requests, please refer to the contact information at the end of this README.

RUNTIME PREREQUISITES

There are a few runtime prerequisites in order for the software to run correctly.

HARDWARE:

- Intel x86-64 CPU with optional support for AVX extensions. AMD CPUs should also work although this has not been tested.
- COMING SOON: Intel 32-bit x86 CPU

- COMING SOON: ARMv7 CPUs

WINDOWS:

- Microsoft Windows 8.0 64-bit or later, Server 2012 or later.
- Microsoft Visual C++ 2013 Redistributables (64-bit)
- COMING SOON: Windows 32-bit
- You MAY need Administrator privileges, in order to:
 - Use large pages, if the `--large_pages` option is selected (see USAGE, below)
 - The first time you use `--large_pages` on a given Windows machine, you may need to ensure that your Windows user account has the necessary rights to allow lockable memory pages. To do this on Windows 8, run `gpedit.msc` → Local Computer Policy → Computer Configuration → Windows Settings → Security Settings → Local Policies → User Rights Assignment → Add your username to "Lock pages in memory". Then log out and then log back in.
 - Use the PowerReader interface, depending on end-user implementation
 - Elevate thread priority and pin threads to logical CPUs for improved performance and benchmarking consistency

GNU/LINUX:

- GNU utilities with support for C++11. Tested with gcc 4.8.2 on Ubuntu 14.04 LTS for x86-64 CPU.
- `libhugetlbfs`. You can obtain it at <http://libhugetlbfs.sourceforge.net>. On Ubuntu systems, you can install using "sudo apt-get install libhugetlbfs0".
- Potentially, administrator privileges, if you plan to use the `--large_pages` option.
 - During runtime, if the `--large_pages` option is selected, you may need to first manually ensure that large pages are available from the OS. This can be done by running "hugeadm --pool-list". It is recommended to set minimum pool to 1GB (in order to measure DRAM effectively). If needed, this can be done by running "hugeadm --pool-pages-min 2MB:512". Alternatively, run the `linux_setup_runtime_hugetlbfs.sh` script that is provided with X-Mem.

INSTALLATION

The only file that is needed to run on Windows is `xmem-win.exe`, and `xmem-linux` on GNU/Linux. It has no other dependencies aside from the system prerequisites listed above.

USAGE

USAGE: `xmem [options]`

Options: `-a`, `--all` Run all possible benchmark modes and settings supported by X-Mem. This will override any other relevant user inputs. Note that X-Mem may run for a long time. `-c`, `--chunk_size` A chunk size in bits to use for load traffic-generating threads used in throughput and loaded latency benchmarks. A chunk is the size of each memory access in a benchmark. Allowed values: 32 64 128 and 256 (platform dependent). Note that some chunk sizes may not be supported on all hardware. 32-bit chunks are not compatible with random-access patterns on 64-bit machines; these combinations of settings will be skipped if they occur. DEFAULT: 64 on 64-bit systems, 32 on 32-bit systems. `-e`, `--extension` Run an X-Mem extension defined by the user at build time. The integer argument specifies a single unique extension. This option may be included multiple times. Note that the extension behavior may or may not depend on the other X-Mem options as its semantics are defined by the extension author. `-f`, `--output_file` Generate an output file in CSV format using the given filename. `-h`, `--help` Print X-Mem usage and exit. `-i`, `--base_test_index` Base index for the first benchmark to run. This option is provided for user convenience in enumerating benchmark tests across several subsequent runs of X-Mem. DEFAULT: 1 `-j`, `--num_worker_threads` Number of worker threads to use in benchmarks. This may not exceed the number

of logical CPUs in the system. For throughput benchmarks, this is the number of independent load-generating threads. For latency benchmarks, this is the number of independent load-generating threads plus one latency measurement thread. In latency benchmarks, 1 worker thread indicates no loading is applied. DEFAULT: 1 -l, -latency Unloaded or loaded latency benchmarking mode. If 1 thread is used, unloaded latency is measured using 64-bit random reads. Otherwise, 1 thread is always dedicated to the 64-bit random read latency measurement, and remaining threads are used for load traffic generation using access patterns, chunk sizes, etc. specified by other arguments. See the throughput option for more information on load traffic generation. -n, -iterations Iterations per benchmark. Multiple independent iterations may be performed on each benchmark setting to ensure consistent results. DEFAULT: 1 -r, -random_access Use a random access pattern for load traffic-generating threads used in throughput and loaded latency benchmarks. -s, -sequential_access Use a sequential and/or strided access pattern for load traffic generating-threads used in throughput and loaded latency benchmarks. -t, -throughput Throughput benchmarking mode. Aggregate throughput is measured across all worker threads. Each load traffic-generating worker in a particular benchmark runs an identical kernel. Multiple distinct benchmarks may be run depending on the specified benchmark settings (e.g., aggregated 64-bit and 256-bit sequential read throughput using strides of 1 and -8 chunks). -u, -ignore_numa Force uniform memory access (UMA) mode. This only has an effect in non-uniform memory access (NUMA) systems. Limits benchmarking to CPU and memory NUMA node 0 instead of all intra-node and inter-node combinations. This mode can be useful in situations where the user is not interested in cross-node effects or node asymmetry. This option may also be required if large pages are desired on GNU/Linux systems due to lack of NUMA support in current versions of hugetlbfs. See the large_pages option. -v, -verbose Verbose mode increases the level of detail in X-Mem console reporting. -w, -working_set_size Working set size per worker thread in KB. This must be a multiple of 4KB. In all benchmarks, each worker thread works on its own "private" region of memory. For example, 4-thread throughput benchmarking with a working set size of 4 KB might result in measuring the aggregate throughput of four L1 caches corresponding to four physical cores, with no data sharing between threads. Similarly, an 8-thread loaded latency benchmark with a working set size of 64 MB would use 512 MB of memory in total for benchmarking, with no data sharing between threads. This would result in performance measurement of the shared DRAM physical interface, the shared L3 cache, etc. -L, -large_pages Use large pages. This might enable better memory performance by reducing the translation-lookaside buffer (TLB) bottleneck. However, this is not supported on all systems. On GNU/Linux, you need hugetlbfs support with pre-reserved huge pages prior to running X-Mem. On GNU/Linux, you also must use the ignore_numa option, as hugetlbfs is not NUMA-aware at this time. -R, -reads Use memory read-based patterns in load traffic-generating threads. -W, -writes Use memory write-based patterns in load traffic-generating threads. -S, -stride_size A stride size to use for load traffic-generating threads, specified in powers-of-two multiples of the chunk size(s). Allowed values: 1, -1, 2, -2, 4, -4, 8, -8, 16, -16. Positive indicates the forward direction (increasing addresses), while negative indicates the reverse direction.

If a given option is not specified, X-Mem defaults will be used where appropriate.

===== EXAMPLE USAGE =====

Print X-Mem usage message and exit. If -help or -h is specified, benchmarks will not run regardless of other options.

```
xmem --help
xmem -h
```

Run unloaded latency benchmarks with 5 iterations of each distinct benchmark setting. The chunk size of 32 bits and sequential access pattern options will be ignored as they only apply to load traffic-generating threads, which are unused here as the default number of worker threads is 1. Console reporting will be verbose.

```
xmem -l --verbose -n5 --chunk_size=32 -s
```

Run throughput and loaded latency benchmarks on a per-thread working set size of 512 MB for a grand total of 1 GB of memory space. Use chunk sizes of 32 and 256 bits for load traffic-generating threads, and ignore NUMA effects. Number the first benchmark test starting at 101 both in console reporting and CSV file output (results.csv).

```
xmem -t --latency -w524288 -f results.csv -c32 -c256 -i 101 -u -j2
```

Run 3 iterations of throughput and loaded latency on a working set of 128 KB per thread. Use 4 worker threads in total. For load traffic-generating threads, use all combinations of read and write memory accesses, random-access patterns, forward sequential, and strided patterns of size -4 and -16 chunks. Ignore NUMA effects in the system and

use large pages. Finally, increase verbosity of console output. Finally, run custom user extensions with indices 0 and 1. Note that these may not necessarily obey other input parameters as their behavior is defined by the extension author.

```
xmem -w128 -n3 -j4 -l -t --extension=0 -e1 -s -S1 -S-4 -r -S16 -R -W -u
```

-L -v

Run EVERYTHING and dump results to file.

```
xmem -a -v -ftest.csv
```

Have fun! =]

BUILDING FROM SOURCE

After you have set the desired compile-time options, build the source. On Windows, running build-win.bat should suffice. On GNU/Linux, run build-linux.sh. Each takes a single argument specifying the target architecture.

If you customize your build, make sure you use the "Release" mode for your OS/compiler. Do not include debug capabilities as it can dramatically affect performance of the benchmarks, leading to pessimistic results.

BUILD PREREQUISITES

There are a few software build prerequisites, depending on your platform.

WINDOWS:

- Any version of Visual Studio 2013 64-bit (also known as version 12.0).
- Python 2.7. You can obtain it at <http://www.python.org>.
- SCons build system. You can obtain it at <http://www.scons.org>. Build tested with SCons 2.3.4.

GNU/LINUX:

- gcc with support for the C++11 standard. Tested with gcc version 4.8.2 on Ubuntu 14.04 LTS for x86-64.
- Python 2.7. You can obtain it at <http://www.python.org>. On Ubuntu systems, you can install using "sudo apt-get install python2.7". You may need some other Python 2.7 packages as well.
- SCons build system. You can obtain it at <http://www.scons.org>. On Ubuntu systems, you can install using "sudo apt-get install scons". Build tested with SCons 2.3.4.
- Kernel support for large (huge) pages. This support can be verified on your Linux installation by running "grep hugetlbfs /proc/filesystems". If you do not have huge page support in your kernel, you can build a kernel with the appropriate options switched on: "CONFIG_HUGETLB_PAGE" and "CONFIG_HUGETLBFS".
- libhugetlbfs. This is used for allocating large (huge) pages if the `-large_pages` runtime option is selected. You can obtain it at <http://libhugetlbfs.sourceforge.net>. On Ubuntu systems, you can install using "sudo apt-get install libhugetlbfs-dev".

DOCUMENTATION BUILD PREREQUISITES

The following tools are only needed for automatically regenerating source code documentation with HTML and PDF.

WINDOWS:

- doxygen tool. You can obtain it at <http://www.stack.nl/~dimitri/doxygen>.

- LaTeX distribution. You can get a Windows distribution at <http://www.miktex.org>.
- make for Windows. You can obtain it at <http://gnuwin32.sourceforge.net/packages/make.-htm>. You will have to manually add it to your Windows path.

GNU/LINUX:

- doxygen tool. You can obtain it at <http://www.stack.nl/~dimitri/doxygen>. On Ubuntu systems, you can install with "sudo apt-get install doxygen".
- LaTeX distribution. On Ubuntu systems, LaTeX distributed with doxygen should actually be sufficient. You can install with "sudo apt-get install doxygen-latex".
- make. This should be included on any GNU/Linux system.

SOURCE CODE DOCUMENTATION

The tool comes with built-in Doxygen comments in the source code, which can be used to generate both HTML and LaTeX → PDF documentation. Documentation is maintained under the doc/ subdirectory. To build documentation after modifying the source, run build-docs-win.bat on Windows, or build-docs-linux.sh on GNU/Linux systems. Note that Doxygen and a LaTeX distribution must be installed on the system.

VERSION CONTROL

This project is under version control using git. Its master repository is hosted at <https://github.com/Microsoft/X-Mem.git>. There is also another fork at <https://github.com/nanocad-lab/X--Mem.git>, which generally mirrors Microsoft's repository.

CONTACT, FEEDBACK, AND BUG REPORTS

For questions, comments, criticism, bug reports, and other feedback for this software, please contact Mark Gottscho via email at mgottscho@ucla.edu or via web at <http://www.seas.ucla.edu/~gottscho>.

ACKNOWLEDGMENT

Mark Gottscho would like to thank Dr. Mohammed Shoaib of Microsoft Research and Dr. Sriram Govindan of Microsoft for their mentorship in the creation of this software. Further thanks to Dr. Bikash Sharma, Mark Santaniello, Mike Andrewartha, and Laura Caulfield of Microsoft for their contributions, feedback, and assistance. Thank you as well to Dr. Jie Liu of Microsoft Research, Dr. Badriddine Khessib and Dr. Kushagra Vaid of Microsoft, and Prof. Puneet Gupta of UCLA for giving me the opportunity to create this work. Finally, Mark would like to thank Dr. Fedor Pikus of Mentor Graphics for teaching him some useful HPC programming techniques.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

xmem::Parser::Action	13
xmem::Parser::StoreOptionAction	67
xmem::Stats::CountOptionsAction	31
xmem::Arg	14
xmem::ExampleArg	34
xmem::MyArg	46
xmem::Benchmark	15
xmem::LatencyBenchmark	35
xmem::ThroughputBenchmark	72
xmem::BenchmarkManager	23
xmem::Configurator	24
xmem::Descriptor	32
xmem::PrintUsageImplementation::IStringWriter	35
xmem::PrintUsageImplementation::FunctionWriter< Function >	35
xmem::PrintUsageImplementation::OStreamWriter< OStream >	52
xmem::PrintUsageImplementation::StreamWriter< Function, Stream >	68
xmem::PrintUsageImplementation::SyscallWriter< Syscall >	69
xmem::PrintUsageImplementation::TemporaryWriter< Temporary >	69
xmem::PrintUsageImplementation::LinePartIterator	38
xmem::PrintUsageImplementation::LineWrapper	39
xmem::Option	46
xmem::Parser	52
xmem::PrintUsageImplementation	62
xmem::Runnable	63
xmem::MemoryWorker	41
xmem::LatencyWorker	37
xmem::LoadWorker	40
xmem::PowerReader	57
xmem::Stats	65
xmem::Thread	70
xmem::Timer	73

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

xmem::Parser::Action	13
xmem::Arg	
Functions for checking the validity of option arguments	14
xmem::Benchmark	
Flexible abstract class for any memory benchmark	15
xmem::BenchmarkManager	
Manages running all benchmarks at a high level	23
xmem::Configurator	
Handles all user input interpretation and generates the necessary flags for running benchmarks	24
xmem::Stats::CountOptionsAction	31
xmem::Descriptor	
Describes an option, its help text (usage) and how it should be parsed	32
xmem::ExampleArg	34
xmem::PrintUsagImplementation::FunctionWriter< Function >	35
xmem::PrintUsagImplementation::IStringWriter	35
xmem::LatencyBenchmark	
A type of benchmark that measures memory latency via random pointer chasing. Loading may be provided with separate threads which access memory as quickly as possible using given access patterns	35
xmem::LatencyWorker	
Multithreading-friendly class to do memory loading	37
xmem::PrintUsagImplementation::LinePartlterator	38
xmem::PrintUsagImplementation::LineWrapper	39
xmem::LoadWorker	
Multithreading-friendly class to do memory loading	40
xmem::MemoryWorker	
Multithreading-friendly class to run memory access kernels	41
xmem::MyArg	46
xmem::Option	
A parsed option from the command line together with its argument if it has one	46
xmem::PrintUsagImplementation::OStreamWriter< OStream >	52
xmem::Parser	
Checks argument vectors for validity and parses them into data structures that are easier to work with	52
xmem::PowerReader	
An abstract base class for measuring power from an arbitrary source. This class is runnable using a worker thread	57
xmem::PrintUsagImplementation	62

xmem::Runnable	
A base class for any object that implements a thread-safe run() function for use by Thread objects	63
xmem::Stats	
Determines the minimum lengths of the buffer and options arrays used for Parser	65
xmem::Parser::StoreOptionAction	67
xmem::PrintUsageImplementation::StreamWriter< Function, Stream >	68
xmem::PrintUsageImplementation::SyscallWriter< Syscall >	69
xmem::PrintUsageImplementation::TemporaryWriter< Temporary >	69
xmem::Thread	
Nice wrapped thread interface independent of particular OS API	70
xmem::ThroughputBenchmark	
A type of benchmark that measures memory throughput	72
xmem::Timer	
This class abstracts some characteristics of simple high resolution stopwatch timer. However, due to the inability or complexity of abstracting shared hardware timers, this class does not actually provide start and stop functions	73

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

src/ Benchmark.cpp	Implementation file for the Benchmark class	75
src/ benchmark_kernels.cpp	Implementation file for benchmark kernel functions for doing the actual work we care about. :) .	75
src/ BenchmarkManager.cpp	Implementation file for the BenchmarkManager class	76
src/ common.cpp	Implementation file for common preprocessor definitions, macros, functions, and global constants	76
src/ Configurator.cpp	Implementation file for the Configurator class and some helper data structures	77
src/ LatencyBenchmark.cpp	Implementation file for the LatencyBenchmark class	97
src/ LatencyWorker.cpp	Implementation file for the LatencyWorker class	97
src/ LoadWorker.cpp	Implementation file for the LoadWorker class	97
src/ main.cpp	Main entry point to the tool	98
src/ MemoryWorker.cpp	Implementation file for the MemoryWorker class	98
src/ PowerReader.cpp	Implementation file for the PowerReader class	98
src/ Runnable.cpp	Implementation file for the Runnable class	99
src/ Thread.cpp	Implementation file for the Thread class	99
src/ ThroughputBenchmark.cpp	Implementation file for the ThroughputBenchmark class	99
src/ Timer.cpp	Implementation file for the Timer class	100
src/ext/DelayInjectedLoadedLatencyBenchmark/ delay_injected_benchmark_kernels.cpp	Implementation file for benchmark kernel functions for the delay-injected loaded latency benchmark	77
src/ext/DelayInjectedLoadedLatencyBenchmark/ DelayInjectedLoadedLatencyBenchmark.cpp	Implementation file for the DelayInjectedLatencyBenchmark class	77
src/include/ Benchmark.h	Header file for the Benchmark class	78

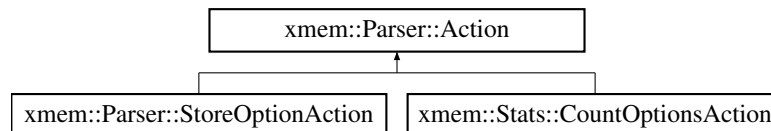
src/include/ benchmark_kernels.h	Header file for benchmark kernel functions for doing the actual work we care about. :)	78
src/include/ BenchmarkManager.h	Header file for the BenchmarkManager class	80
src/include/ build_datetime.h	??
src/include/ common.h	Header file for common preprocessor definitions, macros, functions, and global constants . . .	81
src/include/ Configurator.h	Header file for the Configurator class and some helper data structures	85
src/include/ ExampleArg.h	Slightly-modified third-party code related to OptionParser	86
src/include/ LatencyBenchmark.h	Header file for the LatencyBenchmark class	89
src/include/ LatencyWorker.h	Header file for the LatencyWorker class	89
src/include/ LoadWorker.h	Header file for the LoadWorker class	90
src/include/ MemoryWorker.h	Header file for the MemoryWorker class	90
src/include/ MyArg.h	Extensions to third-party optionparser-related code	90
src/include/ optionparser.h	This is the only file required to use The Lean Mean C++ Option Parser. Just #include it and you're set	91
src/include/ PowerReader.h	Header file for the PowerReader class	94
src/include/ Runnable.h	Header file for the Runnable class	95
src/include/ Thread.h	Header file for the Thread class	95
src/include/ ThroughputBenchmark.h	Header file for the ThroughputBenchmark class	95
src/include/ Timer.h	Header file for the Timer class	96
src/include/ext/DelayInjectedLoadedLatencyBenchmark/ delay_injected_benchmark_kernels.h	Header file for benchmark kernel functions with integrated delays for doing the actual work we care about. :)	86
src/include/ext/DelayInjectedLoadedLatencyBenchmark/ DelayInjectedLoadedLatencyBenchmark.h	Header file for the DelayInjectedLoadedLatencyBenchmark class	89
src/include/win/ win_common_third_party.h	Header file for some third-party helper code for working with Windows APIs	96
src/include/win/ win_CPdhQuery.h	Header and implementation file for some third-party code for measuring Windows OS-exposed performance counters	96
src/include/win/ WindowsDRAMPowerReader.h	Header file for the WindowsDRAMPowerReader class	96
src/win/ win_common_third_party.cpp	Implementation file for some third-party helper code for working with Windows APIs	100
src/win/ WindowsDRAMPowerReader.cpp	Implementation file for the WindowsDRAMPowerReader class	100

Chapter 5

Class Documentation

5.1 xmem::Parser::Action Struct Reference

Inheritance diagram for xmem::Parser::Action:



Public Member Functions

- virtual bool [perform](#) ([Option](#) &)
Called by `Parser::workhorse()` for each [Option](#) that has been successfully parsed (including unknown options if they have a [Descriptor](#) whose [Descriptor::check_arg](#) does not return `ARG_ILLEGAL`).
- virtual bool [finished](#) (int numargs, const char **args)
Called by `Parser::workhorse()` after finishing the parse.

5.1.1 Member Function Documentation

5.1.1.1 virtual bool xmem::Parser::Action::finished (int numargs, const char ** args) [inline],[virtual]

Called by `Parser::workhorse()` after finishing the parse.

Parameters

<i>numargs</i>	the number of non-option arguments remaining
<i>args</i>	pointer to the first remaining non-option argument (if numargs > 0).

Returns

`false` iff a fatal error has occurred.

Reimplemented in [xmem::Parser::StoreOptionAction](#).

5.1.1.2 virtual bool xmem::Parser::Action::perform ([Option](#) &) [inline],[virtual]

Called by `Parser::workhorse()` for each [Option](#) that has been successfully parsed (including unknown options if they have a [Descriptor](#) whose [Descriptor::check_arg](#) does not return `ARG_ILLEGAL`).

Returns `false` iff a fatal error has occurred and the parse should be aborted.

Reimplemented in [xmem::Parser::StoreOptionAction](#), and [xmem::Stats::CountOptionsAction](#).

The documentation for this struct was generated from the following file:

- [src/include/optionparser.h](#)

5.2 xmem::Arg Struct Reference

Functions for checking the validity of option arguments.

```
#include <optionparser.h>
```

Inheritance diagram for `xmem::Arg`:



Static Public Member Functions

- static ArgStatus [None](#) (const [Option](#) &, bool)
For options that don't take an argument: Returns ARG_NONE.
- static ArgStatus [Optional](#) (const [Option](#) &option, bool)
Returns ARG_OK if the argument is attached and ARG_IGNORE otherwise.

5.2.1 Detailed Description

Functions for checking the validity of option arguments.

The following example code can serve as starting place for writing your own more complex CheckArg functions:

```

struct Arg: public option::Arg
{
    static void printError(const char* msg1, const option::Option& opt, const char* msg2)
    {
        fprintf(stderr, "ERROR: %s", msg1);
        fwrite(opt.name, opt.namelen, 1, stderr);
        fprintf(stderr, "%s", msg2);
    }

    static option::ArgStatus Unknown(const option::Option& option, bool msg)
    {
        if (msg) printError("Unknown option '", option, "'\n");
        return option::ARG_ILLEGAL;
    }

    static option::ArgStatus Required(const option::Option& option, bool msg)
    {
        if (option.arg != 0)
            return option::ARG_OK;

        if (msg) printError("Option '", option, "' requires an argument\n");
        return option::ARG_ILLEGAL;
    }

    static option::ArgStatus NonEmpty(const option::Option& option, bool msg)
    {
        if (option.arg != 0 && option.arg[0] != 0)
            return option::ARG_OK;
    }
}
  
```

```

    if (msg) printError("Option '", option, "' requires a non-empty argument\n");
    return option::ARG_ILLEGAL;
}

static option::ArgStatus Numeric(const option::Option& option, bool msg)
{
    char* endptr = 0;
    if (option.arg != 0 && strtol(option.arg, &endptr, 10){});
    if (endptr != option.arg && *endptr == 0)
        return option::ARG_OK;

    if (msg) printError("Option '", option, "' requires a numeric argument\n");
    return option::ARG_ILLEGAL;
}
};

```

The documentation for this struct was generated from the following file:

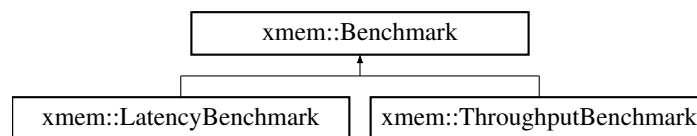
- [src/include/optionparser.h](#)

5.3 xmem::Benchmark Class Reference

Flexible abstract class for any memory benchmark.

```
#include <Benchmark.h>
```

Inheritance diagram for xmem::Benchmark:



Public Member Functions

- [Benchmark](#) (void *mem_array, size_t len, uint32_t iterations, uint32_t num_worker_threads, uint32_t mem_node, uint32_t cpu_node, pattern_mode_t pattern_mode, rw_mode_t rw_mode, chunk_size_t chunk_size, int32_t stride_size, std::vector< [PowerReader](#) * > dram_power_readers, std::string metricUnits, std::string name)
Constructor.
- virtual [~Benchmark](#) ()
Destructor.
- bool [run](#) ()
Runs the benchmark.
- void [print_benchmark_header](#) () const
Prints a header piece of information describing the benchmark to the console.
- virtual void [report_benchmark_info](#) () const
Reports benchmark configuration details to the console.
- virtual void [report_results](#) () const
Reports results to the console.
- bool [isValid](#) () const
Checks to see that the object is in a valid state.
- bool [hasRun](#) () const
Checks to see if the benchmark has run.
- double [getMetricOnIter](#) (uint32_t iter) const
Extracts the metric of interest for a given iteration. Units are interpreted by the inheriting class.

- double [getAverageMetric](#) () const
Gets the average benchmark metric across all iterations.
- std::string [getMetricUnits](#) () const
Gets the units of the metric for this benchmark.
- double [getAverageDRAMPower](#) (uint32_t socket_id) const
Gets the average DRAM power over the benchmark.
- double [getPeakDRAMPower](#) (uint32_t socket_id) const
Gets the peak DRAM power over the benchmark.
- size_t [getLen](#) () const
Gets the length of the memory region in bytes. This is not necessarily the "working set size" depending on multi-threading configuration.
- uint32_t [getIterations](#) () const
Gets the number of iterations for this benchmark.
- chunk_size_t [getChunkSize](#) () const
Gets the width of memory access used in this benchmark.
- int32_t [getStrideSize](#) () const
Gets the stride size for this benchmark.
- uint32_t [getCPUNode](#) () const
Gets the CPU NUMA node used in this benchmark.
- uint32_t [getMemNode](#) () const
Gets the memory NUMA node used in this benchmark.
- uint32_t [getNumThreads](#) () const
Gets the number of worker threads used in this benchmark.
- std::string [getName](#) () const
Gets the human-friendly name of this benchmark.
- pattern_mode_t [getPatternMode](#) () const
Gets the pattern mode for this benchmark.
- rw_mode_t [getRWMode](#) () const
Gets the read/write mode for this benchmark.

Protected Member Functions

- virtual bool [_run_core](#) ()=0
The core benchmark function.
- bool [_start_power_threads](#) ()
Starts the DRAM power measurement threads.
- bool [_stop_power_threads](#) ()
Stops the DRAM power measurement threads. This is a blocking call.

Protected Attributes

- void * [_mem_array](#)
- size_t [_len](#)
- uint32_t [_iterations](#)
- uint32_t [_num_worker_threads](#)
- uint32_t [_mem_node](#)
- uint32_t [_cpu_node](#)
- pattern_mode_t [_pattern_mode](#)
- rw_mode_t [_rw_mode](#)
- chunk_size_t [_chunk_size](#)
- int32_t [_stride_size](#)

- `std::vector< PowerReader * > _dram_power_readers`
- `std::vector< Thread * > _dram_power_threads`
- `std::vector< double > _metricOnIter`
- `double _averageMetric`
- `std::string _metricUnits`
- `std::vector< double > _average_dram_power_socket`
- `std::vector< double > _peak_dram_power_socket`
- `std::string _name`
- `bool _obj_valid`
- `bool _hasRun`
- `bool _warning`

5.3.1 Detailed Description

Flexible abstract class for any memory benchmark.

This class provides a generic interface for interacting with a benchmark. All benchmarks should be derived from this class.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 `Benchmark::Benchmark (void * mem_array, size_t len, uint32_t iterations, uint32_t num_worker_threads, uint32_t mem_node, uint32_t cpu_node, pattern_mode_t pattern_mode, rw_mode_t rw_mode, chunk_size_t chunk_size, int32_t stride_size, std::vector< PowerReader * > dram_power_readers, std::string metricUnits, std::string name)`

Constructor.

Parameters

<i>mem_array</i>	A pointer to a contiguous chunk of memory that has been allocated for benchmarking among potentially several worker threads. This should be aligned to a 256-bit boundary.
<i>len</i>	Length of <i>mem_array</i> in bytes. This must be a multiple of 4 KB and should be at least the per-thread working set size times the number of worker threads.
<i>iterations</i>	Number of iterations of the complete benchmark. Used to average results and provide a measure of consistency and reproducibility.
<i>passes_per_iteration</i>	Number of passes to do in each iteration, to ensure timed section of code is "long enough".
<i>num_worker_threads</i>	The number of worker threads to use in the benchmark.
<i>mem_node</i>	The logical memory NUMA node used in the benchmark.
<i>cpu_node</i>	The logical CPU NUMA node to use for the benchmark.
<i>pattern_mode</i>	This indicates the general type of access pattern used, e.g. sequential or random.
<i>rw_mode</i>	This indicates the general type of read/write mix used, e.g. pure reads or pure writes.
<i>chunk_size</i>	Size of an individual memory access for load-generating worker threads.
<i>stride_size</i>	For sequential access patterns, this is the address distance between successive accesses, counted in chunks. Negative values indicate a reversed access pattern. A stride of +/-1 is purely sequential.
<i>dram_power_readers</i>	A group of PowerReader objects for measuring DRAM power.
<i>name</i>	The name of the benchmark to use when reporting to console.

5.3.3 Member Function Documentation

5.3.3.1 `virtual bool xmem::Benchmark::run_core ()` `[protected]`, `[pure virtual]`

The core benchmark function.

Returns

True on success.

Implemented in [xmem::LatencyBenchmark](#), and [xmem::ThroughputBenchmark](#).

5.3.3.2 bool Benchmark::_start_power_threads () [protected]

Starts the DRAM power measurement threads.

Returns

True on success.

5.3.3.3 bool Benchmark::_stop_power_threads () [protected]

Stops the DRAM power measurement threads. This is a blocking call.

Returns

True on success.

5.3.3.4 double Benchmark::getAverageDRAMPower (uint32_t socket_id) const

Gets the average DRAM power over the benchmark.

Returns

The average DRAM power for a given socket in watts, or 0 if the data does not exist (power was unable to be collected or the benchmark has not run).

5.3.3.5 double Benchmark::getAverageMetric () const

Gets the average benchmark metric across all iterations.

Returns

The average metric.

5.3.3.6 chunk_size_t Benchmark::getChunkSize () const

Gets the width of memory access used in this benchmark.

Returns

The chunk size for this benchmark.

5.3.3.7 uint32_t Benchmark::getCPUNode () const

Gets the CPU NUMA node used in this benchmark.

Returns

The NUMA CPU node used in this benchmark.

5.3.3.8 uint32_t Benchmark::getIterations () const

Gets the number of iterations for this benchmark.

Returns

The number of iterations for this benchmark.

5.3.3.9 size_t Benchmark::getLen () const

Gets the length of the memory region in bytes. This is not necessarily the "working set size" depending on multi-threading configuration.

Returns

Length of the memory region in bytes.

5.3.3.10 uint32_t Benchmark::getMemNode () const

Gets the memory NUMA node used in this benchmark.

Returns

The NUMA memory node used in this benchmark.

5.3.3.11 double Benchmark::getMetricOnIter (uint32_t iter) const

Extracts the metric of interest for a given iteration. Units are interpreted by the inheriting class.

Parameters

<i>iter</i>	Iteration to extract.
-------------	-----------------------

Returns

The metric on the iteration specified by the input.

5.3.3.12 std::string Benchmark::getMetricUnits () const

Gets the units of the metric for this benchmark.

Returns

A string representing the units for printing to console and file.

5.3.3.13 std::string Benchmark::getName () const

Gets the human-friendly name of this benchmark.

Returns

The benchmark test name.

5.3.3.14 `uint32_t Benchmark::getNumThreads () const`

Gets the number of worker threads used in this benchmark.

Returns

The number of worker threads used in this benchmark.

5.3.3.15 `pattern_mode_t Benchmark::getPatternMode () const`

Gets the pattern mode for this benchmark.

Returns

The pattern mode enumerator.

5.3.3.16 `double Benchmark::getPeakDRAMPower (uint32_t socket_id) const`

Gets the peak DRAM power over the benchmark.

Returns

The peak DRAM power for a given socket in watts, or 0 if the data does not exist (power was unable to be collected or the benchmark has not run).

5.3.3.17 `rw_mode_t Benchmark::getRWMode () const`

Gets the read/write mode for this benchmark.

Returns

The read/write mix mode.

5.3.3.18 `int32_t Benchmark::getStrideSize () const`

Gets the stride size for this benchmark.

Returns

The stride size in chunks.

5.3.3.19 `bool Benchmark::hasRun () const`

Checks to see if the benchmark has run.

Returns

True if `run()` has already completed successfully.

5.3.3.20 `bool Benchmark::isValid () const`

Checks to see that the object is in a valid state.

Returns

True if the object was constructed correctly and can be used.

5.3.3.21 bool Benchmark::run ()

Runs the benchmark.

Returns

True on benchmark success

5.3.4 Member Data Documentation

5.3.4.1 std::vector<double> xmem::Benchmark::_average_dram_power_socket [protected]

The average DRAM power in this benchmark, per socket.

5.3.4.2 double xmem::Benchmark::_averageMetric [protected]

Average metric over all iterations. Unit-less because any benchmark can set this metric as needed. It is up to the descendant class to interpret units.

5.3.4.3 chunk_size_t xmem::Benchmark::_chunk_size [protected]

Chunk size of memory accesses in this benchmark.

5.3.4.4 uint32_t xmem::Benchmark::_cpu_node [protected]

The CPU NUMA node used in this benchmark.

5.3.4.5 std::vector<PowerReader*> xmem::Benchmark::_dram_power_readers [protected]

The power reading objects for measuring DRAM power on a per-socket basis during the benchmark.

5.3.4.6 std::vector<Thread*> xmem::Benchmark::_dram_power_threads [protected]

The power reading threads for measuring DRAM power on a per-socket basis during the benchmark. These work with the DRAM power readers. Although they are worker threads, they are not counted as the "official" benchmarking worker threads.

5.3.4.7 bool xmem::Benchmark::_hasRun [protected]

Indicates whether the benchmark has run.

5.3.4.8 uint32_t xmem::Benchmark::_iterations [protected]

Number of iterations used in this benchmark.

5.3.4.9 size_t xmem::Benchmark::_len [protected]

Length of the memory region in bytes. This is not the working set size per thread!

5.3.4.10 `void* xmem::Benchmark::_mem_array` [protected]

Pointer to the memory region to use in this benchmark.

5.3.4.11 `uint32_t xmem::Benchmark::_mem_node` [protected]

The memory NUMA node used in this benchmark.

5.3.4.12 `std::vector<double> xmem::Benchmark::_metricOnIter` [protected]

Metrics for each iteration of the benchmark. Unit-less because any benchmark can set this metric as needed. It is up to the descendant class to interpret units.

5.3.4.13 `std::string xmem::Benchmark::_metricUnits` [protected]

String representing the units of measurement for the metric.

5.3.4.14 `std::string xmem::Benchmark::_name` [protected]

Name of this benchmark.

5.3.4.15 `uint32_t xmem::Benchmark::_num_worker_threads` [protected]

The number of worker threads used in this benchmark.

5.3.4.16 `bool xmem::Benchmark::_obj_valid` [protected]

Indicates whether this benchmark object is valid.

5.3.4.17 `pattern_mode_t xmem::Benchmark::_pattern_mode` [protected]

Access pattern mode.

5.3.4.18 `std::vector<double> xmem::Benchmark::_peak_dram_power_socket` [protected]

The peak DRAM power in this benchmark, per socket.

5.3.4.19 `rw_mode_t xmem::Benchmark::_rw_mode` [protected]

Read/write mode.

5.3.4.20 `int32_t xmem::Benchmark::_stride_size` [protected]

Stride size in chunks for sequential pattern mode only.

5.3.4.21 bool xmem::Benchmark::_warning [protected]

Indicates whether the benchmarks results might be clearly questionable/inaccurate/incorrect due to a variety of factors.

The documentation for this class was generated from the following files:

- src/include/Benchmark.h
- src/Benchmark.cpp

5.4 xmem::BenchmarkManager Class Reference

Manages running all benchmarks at a high level.

```
#include <BenchmarkManager.h>
```

Public Member Functions

- [BenchmarkManager](#) ([Configurator](#) &config)
Constructor.
- [~BenchmarkManager](#) ()
Destructor.
- bool [runAll](#) ()
Runs all benchmark configurations (does not include extensions).
- bool [runThroughputBenchmarks](#) ()
Runs the throughput benchmarks.
- bool [runLatencyBenchmarks](#) ()
Runs the latency benchmark.

5.4.1 Detailed Description

Manages running all benchmarks at a high level.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 BenchmarkManager::BenchmarkManager (Configurator & config)

Constructor.

Parameters

<i>config</i>	The configuration object containing run-time options for this X-Mem execution instance.
---------------	---

5.4.3 Member Function Documentation

5.4.3.1 bool BenchmarkManager::runAll ()

Runs all benchmark configurations (does not include extensions).

Returns

True on success.

5.4.3.2 bool BenchmarkManager::runLatencyBenchmarks ()

Runs the latency benchmark.

Returns

True on benchmarking success.

5.4.3.3 bool BenchmarkManager::runThroughputBenchmarks ()

Runs the throughput benchmarks.

Returns

True on benchmarking success.

The documentation for this class was generated from the following files:

- src/include/BenchmarkManager.h
- src/BenchmarkManager.cpp

5.5 xmem::Configurator Class Reference

Handles all user input interpretation and generates the necessary flags for running benchmarks.

```
#include <Configurator.h>
```

Public Member Functions

- [Configurator](#) ()
Default constructor. A default configuration is set. You will want to run [configureFromInput\(\)](#) most likely.
- [int32_t configureFromInput](#) (int argc, char *argv[])
Specialized constructor for when you don't want to get config from input, and you want to pass it in directly.
- [bool extensionsEnabled](#) () const
Determines whether user extensions are enabled.
- [bool runExtDelayInjectedLoadedLatencyBenchmark](#) () const
If included at compile-time, determines whether the delay-injected loaded latency benchmark extension should be run.
- [bool latencyTestSelected](#) () const
Indicates if the latency test has been selected.
- [bool throughputTestSelected](#) () const
Indicates if the throughput test has been selected.
- [size_t getWorkingSetSizePerThread](#) () const
Gets the working set size in bytes for each worker thread, if applicable.
- [bool useChunk32b](#) () const
Determines if chunk size of 32 bits should be used in relevant benchmarks.
- [bool isNUMAEnabled](#) () const
Determines if the benchmarks should test for all CPU/memory NUMA combinations.
- [uint32_t getIterationsPerTest](#) () const
Gets the number of iterations that should be run of each benchmark.
- [bool useRandomAccessPattern](#) () const
Determines if throughput benchmarks should use a random access pattern.

- bool [useSequentialAccessPattern](#) () const
Determines if throughput benchmarks should use a sequential access pattern.
- uint32_t [getNumWorkerThreads](#) () const
Gets the number of worker threads to use.
- uint32_t [getStartingTestIndex](#) () const
Gets the numerical index of the first benchmark for CSV output purposes.
- std::string [getOutputFilename](#) () const
Gets the output filename to use, if applicable.
- bool [useOutputFile](#) () const
Determines whether to generate an output CSV file.
- void [setUseOutputFile](#) (bool use)
Changes whether an output file should be used.
- bool [verboseMode](#) () const
Determines whether X-Mem is in verbose mode.
- bool [useLargePages](#) () const
Determines whether X-Mem should use large pages.
- bool [useReads](#) () const
Determines whether reads should be used in throughput benchmarks.
- bool [useWrites](#) () const
Determines whether writes should be used in throughput benchmarks.
- bool [useStrideP1](#) () const
Determines if a stride of +1 should be used in relevant benchmarks.
- bool [useStrideN1](#) () const
Determines if a stride of -1 should be used in relevant benchmarks.
- bool [useStrideP2](#) () const
Determines if a stride of +2 should be used in relevant benchmarks.
- bool [useStrideN2](#) () const
Determines if a stride of -2 should be used in relevant benchmarks.
- bool [useStrideP4](#) () const
Determines if a stride of +4 should be used in relevant benchmarks.
- bool [useStrideN4](#) () const
Determines if a stride of -4 should be used in relevant benchmarks.
- bool [useStrideP8](#) () const
Determines if a stride of +8 should be used in relevant benchmarks.
- bool [useStrideN8](#) () const
Determines if a stride of -8 should be used in relevant benchmarks.
- bool [useStrideP16](#) () const
Determines if a stride of +16 should be used in relevant benchmarks.
- bool [useStrideN16](#) () const
Determines if a stride of -16 should be used in relevant benchmarks.

5.5.1 Detailed Description

Handles all user input interpretation and generates the necessary flags for running benchmarks.

5.5.2 Member Function Documentation

5.5.2.1 int32_t Configurator::configureFromInput (int argc, char * argv[])

Specialized constructor for when you don't want to get config from input, and you want to pass it in directly.

Parameters

<i>runExtensions</i>	Indicates if user-defined code should be run in addition to other standard functionality.
<i>run_ext_delay_injected_loaded_latency_benchmark</i>	If true, and this extension is included at compile-time, run the delay-injected loaded latency benchmark extension.
<i>run_ext_stream_benchmark</i>	If true, and this extension is included at compile-time, run the STREAM-like benchmark extension.
<i>runLatency</i>	Indicates latency benchmarks should be run.
<i>runThroughput</i>	Indicates throughput benchmarks should be run.
<i>working_set_size_per_thread</i>	The total size of memory to test in all benchmarks, in bytes, per thread. This MUST be a multiple of 4KB pages.
<i>num_worker_threads</i>	The number of threads to use in throughput benchmarks, loaded latency benchmarks, and stress tests.
<i>use_chunk_32b</i>	If true, include 32-bit chunks for relevant benchmarks.
<i>use_chunk_64b</i>	If true, include 64-bit chunks for relevant benchmarks.
<i>use_chunk_128b</i>	If true, include 128-bit chunks for relevant benchmarks.
<i>use_chunk_256b</i>	If true, include 256-bit chunks for relevant benchmarks.
<i>numa_enable</i>	If true, then test all combinations of CPU/memory NUMA nodes.
<i>iterations_per_test</i>	For each unique benchmark test, this is the number of times to repeat it.
<i>use_random_access_pattern</i>	If true, use random-access patterns in throughput benchmarks.
<i>use_sequential_access_pattern</i>	If true, use sequential-access patterns in throughput benchmarks.
<i>starting_test_index</i>	Numerical index to use for the first test. This is an aid for end-user interpreting and post-processing of result CSV file, if relevant.
<i>filename</i>	Output filename to use.
<i>use_output_file</i>	If true, use the provided output filename.
<i>verbose</i>	If true, then X-Mem should be more verbose in its console reporting.
<i>use_large_pages</i>	If true, then X-Mem will attempt to force usage of large pages.
<i>use_reads</i>	If true, then throughput benchmarks should use reads.
<i>use_writes</i>	If true, then throughput benchmarks should use writes.
<i>use_stride_p1</i>	If true, include stride of +1 for relevant benchmarks.
<i>use_stride_n1</i>	If true, include stride of -1 for relevant benchmarks.
<i>use_stride_p2</i>	If true, include stride of +2 for relevant benchmarks.
<i>use_stride_n2</i>	If true, include stride of -2 for relevant benchmarks.
<i>use_stride_p4</i>	If true, include stride of +4 for relevant benchmarks.
<i>use_stride_n4</i>	If true, include stride of -4 for relevant benchmarks.
<i>use_stride_p8</i>	If true, include stride of +8 for relevant benchmarks.
<i>use_stride_n8</i>	If true, include stride of -8 for relevant benchmarks.
<i>use_stride_p16</i>	If true, include stride of +16 for relevant benchmarks.
<i>use_stride_n16</i>	If true, include stride of -16 for relevant benchmarks. Configures the tool based on user's command-line inputs.

<i>argc</i>	The argc from main() .
<i>argv</i>	The argv from main() .

Returns

0 on success.

5.5.2.2 `bool xmem::Configurator::extensionsEnabled () const [inline]`

Determines whether user extensions are enabled.

Returns

True if extensions are enabled.

5.5.2.3 `uint32_t xmem::Configurator::getIterationsPerTest () const [inline]`

Gets the number of iterations that should be run of each benchmark.

Returns

The iterations for each test.

5.5.2.4 `uint32_t xmem::Configurator::getNumWorkerThreads () const [inline]`

Gets the number of worker threads to use.

Returns

The number of worker threads.

5.5.2.5 `std::string xmem::Configurator::getOutputFilename () const [inline]`

Gets the output filename to use, if applicable.

Returns

The output filename to use if [useOutputFile\(\)](#) returns true. Otherwise return value is "".

5.5.2.6 `uint32_t xmem::Configurator::getStartingTestIndex () const [inline]`

Gets the numerical index of the first benchmark for CSV output purposes.

Returns

The starting benchmark index.

5.5.2.7 `size_t xmem::Configurator::getWorkingSetSizePerThread () const [inline]`

Gets the working set size in bytes for each worker thread, if applicable.

Returns

The working set size in bytes.

5.5.2.8 `bool xmem::Configurator::isNUMAEnabled () const [inline]`

Determines if the benchmarks should test for all CPU/memory NUMA combinations.

Returns

True if all NUMA nodes should be tested.

5.5.2.9 `bool xmem::Configurator::latencyTestSelected () const [inline]`

Indicates if the latency test has been selected.

Returns

True if the latency test has been selected to run.

5.5.2.10 `bool xmem::Configurator::runExtDelayInjectedLoadedLatencyBenchmark () const [inline]`

If included at compile-time, determines whether the delay-injected loaded latency benchmark extension should be run.

Returns

True if it should be run.

5.5.2.11 `void xmem::Configurator::setUseOutputFile (bool use) [inline]`

Changes whether an output file should be used.

Parameters

<i>use</i>	If true, then use the output file.
------------	------------------------------------

5.5.2.12 `bool xmem::Configurator::throughputTestSelected () const [inline]`

Indicates if the throughput test has been selected.

Returns

True if the throughput test has been selected to run.

5.5.2.13 `bool xmem::Configurator::useChunk32b () const [inline]`

Determines if chunk size of 32 bits should be used in relevant benchmarks.

Returns

True if 32-bit chunks should be used.

5.5.2.14 `bool xmem::Configurator::useLargePages () const [inline]`

Determines whether X-Mem should use large pages.

Parameters

<i>True</i>	if large pages should be used.
-------------	--------------------------------

5.5.2.15 `bool xmem::Configurator::useOutputFile () const [inline]`

Determines whether to generate an output CSV file.

Returns

True if an output file should be used.

5.5.2.16 `bool xmem::Configurator::useRandomAccessPattern () const [inline]`

Determines if throughput benchmarks should use a random access pattern.

Returns

True if random access should be used.

5.5.2.17 `bool xmem::Configurator::useReads () const [inline]`

Determines whether reads should be used in throughput benchmarks.

Returns

True if reads should be used.

5.5.2.18 `bool xmem::Configurator::useSequentialAccessPattern () const [inline]`

Determines if throughput benchmarks should use a sequential access pattern.

Returns

True if sequential access should be used.

5.5.2.19 `bool xmem::Configurator::useStrideN1 () const [inline]`

Determines if a stride of -1 should be used in relevant benchmarks.

Returns

True if a stride of -1 should be used.

5.5.2.20 `bool xmem::Configurator::useStrideN16 () const [inline]`

Determines if a stride of -16 should be used in relevant benchmarks.

Returns

True if a stride of -16 should be used.

5.5.2.21 `bool xmem::Configurator::useStrideN2 () const [inline]`

Determines if a stride of -2 should be used in relevant benchmarks.

Returns

True if a stride of -2 should be used.

5.5.2.22 `bool xmem::Configurator::useStrideN4 () const [inline]`

Determines if a stride of -4 should be used in relevant benchmarks.

Returns

True if a stride of -4 should be used.

5.5.2.23 `bool xmem::Configurator::useStrideN8 () const [inline]`

Determines if a stride of -8 should be used in relevant benchmarks.

Returns

True if a stride of -8 should be used.

5.5.2.24 `bool xmem::Configurator::useStrideP1 () const [inline]`

Determines if a stride of +1 should be used in relevant benchmarks.

Returns

True if a stride of +1 should be used.

5.5.2.25 `bool xmem::Configurator::useStrideP16 () const [inline]`

Determines if a stride of +16 should be used in relevant benchmarks.

Returns

True if a stride of +16 should be used.

5.5.2.26 `bool xmem::Configurator::useStrideP2 () const [inline]`

Determines if a stride of +2 should be used in relevant benchmarks.

Returns

True if a stride of +2 should be used.

5.5.2.27 `bool xmem::Configurator::useStrideP4 () const [inline]`

Determines if a stride of +4 should be used in relevant benchmarks.

Returns

True if a stride of +4 should be used.

5.5.2.28 `bool xmem::Configurator::useStrideP8 () const [inline]`

Determines if a stride of +8 should be used in relevant benchmarks.

Returns

True if a stride of +8 should be used.

5.5.2.29 `bool xmem::Configurator::useWrites () const [inline]`

Determines whether writes should be used in throughput benchmarks.

Returns

True if writes should be used.

5.5.2.30 `bool xmem::Configurator::verboseMode () const [inline]`

Determines whether X-Mem is in verbose mode.

Returns

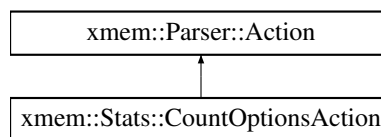
True if verbose mode is enabled.

The documentation for this class was generated from the following files:

- [src/include/Configurator.h](#)
- [src/Configurator.cpp](#)

5.6 xmem::Stats::CountOptionsAction Class Reference

Inheritance diagram for xmem::Stats::CountOptionsAction:



Public Member Functions

- [CountOptionsAction](#) (unsigned *buffer_max_)
- bool [perform](#) ([Option](#) &)

Called by [Parser::workhorse\(\)](#) for each [Option](#) that has been successfully parsed (including unknown options if they have a [Descriptor](#) whose [Descriptor::check_arg](#) does not return `ARG_ILLEGAL`).

5.6.1 Constructor & Destructor Documentation

5.6.1.1 `xmem::Stats::CountOptionsAction::CountOptionsAction (unsigned * buffer_max_) [inline]`

Creates a new [CountOptionsAction](#) that will increase *buffer_max_ for each parsed [Option](#).

5.6.2 Member Function Documentation

5.6.2.1 `bool xmem::Stats::CountOptionsAction::perform (Option &) [inline],[virtual]`

Called by `Parser::workhorse()` for each [Option](#) that has been successfully parsed (including unknown options if they have a [Descriptor](#) whose [Descriptor::check_arg](#) does not return `ARG_ILLEGAL`).

Returns `false` iff a fatal error has occurred and the parse should be aborted.

Reimplemented from [xmem::Parser::Action](#).

The documentation for this class was generated from the following file:

- [src/include/optionparser.h](#)

5.7 xmem::Descriptor Struct Reference

Describes an option, its help text (usage) and how it should be parsed.

```
#include <optionparser.h>
```

Public Attributes

- `const unsigned` [index](#)
Index of this option's linked list in the array filled in by the parser.
- `const int` [type](#)
Used to distinguish between options with the same [index](#). See [index](#) for details.
- `const char *const` [shortopt](#)
Each char in this string will be accepted as a short option character.
- `const char *const` [longopt](#)
The long option name (without the leading -).
- `const CheckArg` [check_arg](#)
For each option that matches [shortopt](#) or [longopt](#) this function will be called to check a potential argument to the option.
- `const char *` [help](#)
The usage text associated with the options in this [Descriptor](#).

5.7.1 Detailed Description

Describes an option, its help text (usage) and how it should be parsed.

The main input when constructing an `option::Parser` is an array of `Descriptors`.

Example:

```
enum OptionIndex {CREATE, ...};
enum OptionType {DISABLE, ENABLE, OTHER};

const option::Descriptor usage[] = {
    { CREATE,                                // index
      OTHER,                                // type
      "c",                                  // shortopt
      "create",                              // longopt
      Arg::None,                             // check_arg
      "--create Tells the program to create something." // help
    }, ...
};
```

5.7.2 Member Data Documentation

5.7.2.1 const CheckArg xmem::Descriptor::check_arg

For each option that matches [shorpt](#) or [longopt](#) this function will be called to check a potential argument to the option.

This function will be called even if there is no potential argument. In that case it will be passed `NULL` as `arg` parameter. Do not confuse this with the empty string.

See `CheckArg` for more information.

5.7.2.2 const char* xmem::Descriptor::help

The usage text associated with the options in this [Descriptor](#).

You can use `option::printUsage()` to format your usage message based on the `help` texts. You can use dummy Descriptors where [shorpt](#) and [longopt](#) are both the empty string to add text to the usage that is not related to a specific option.

See `option::printUsage()` for special formatting characters you can use in `help` to get a column layout.

Attention

Must be UTF-8-encoded. If your compiler supports C++11 you can use the `"u8"` prefix to make sure string literals are properly encoded.

5.7.2.3 const unsigned xmem::Descriptor::index

Index of this option's linked list in the array filled in by the parser.

Command line options whose Descriptors have the same index will end up in the same linked list in the order in which they appear on the command line. If you have multiple long option aliases that refer to the same option, give their descriptors the same `index`.

If you have options that mean exactly opposite things (e.g. `-enable-foo` and `-disable-foo`), you should also give them the same `index`, but distinguish them through different values for `type`. That way they end up in the same list and you can just take the last element of the list and use its type. This way you get the usual behaviour where switches later on the command line override earlier ones without having to code it manually.

Tip:

Use an enum rather than plain ints for better readability, as shown in the example at [Descriptor](#).

5.7.2.4 const char* const xmem::Descriptor::longopt

The long option name (without the leading `-`).

If this [Descriptor](#) should not have a long option name, use the empty string `""`. `NULL` is not permitted here!

While [shorpt](#) allows multiple short option characters, each [Descriptor](#) can have only a single long option name. If you have multiple long option names referring to the same option use separate Descriptors that have the same `index` and `type`. You may repeat short option characters in such an alias [Descriptor](#) but there's no need to.

Dummy Descriptors:

You can use dummy Descriptors with an empty string for both [shorpt](#) and [longopt](#) to add text to the usage that is not related to a specific option. See [help](#). The first dummy [Descriptor](#) will be used for unknown options (see below).

Unknown Option Descriptor:

The first dummy [Descriptor](#) in the list of Descriptors, whose [shortopt](#) and [longopt](#) are both the empty string, will be used as the [Descriptor](#) for unknown options. An unknown option is a string in the argument vector that is not a lone minus '-' but starts with a minus character and does not match any [Descriptor's shortopt](#) or [longopt](#). Note that the dummy descriptor's [check_arg](#) function *will* be called and its return value will be evaluated as usual. I.e. if it returns ARG_ILLEGAL the parsing will be aborted with `Parser::error()==true`. if [check_arg](#) does not return ARG_ILLEGAL the descriptor's [index](#) *will* be used to pick the linked list into which to put the unknown option.

If there is no dummy descriptor, unknown options will be dropped silently.

5.7.2.5 const char* const xmem::Descriptor::shortopt

Each char in this string will be accepted as a short option character.

The string must not include the minus character '-' or you'll get undefined behaviour.

If this [Descriptor](#) should not have short option characters, use the empty string "". NULL is not permitted here!

See [longopt](#) for more information.

5.7.2.6 const int xmem::Descriptor::type

Used to distinguish between options with the same [index](#). See [index](#) for details.

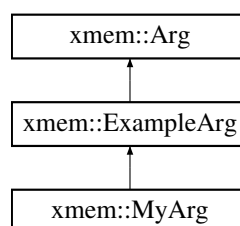
It is recommended that you use an enum rather than a plain int to make your code more readable.

The documentation for this struct was generated from the following file:

- [src/include/optionparser.h](#)

5.8 xmem::ExampleArg Class Reference

Inheritance diagram for xmem::ExampleArg:

**Static Public Member Functions**

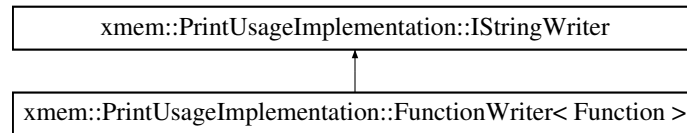
- static void **printError** (const char *msg1, const [Option](#) &opt, const char *msg2)
- static ArgStatus **Unknown** (const [Option](#) &option, bool msg)
- static ArgStatus **Required** (const [Option](#) &option, bool msg)
- static ArgStatus **NonEmpty** (const [Option](#) &option, bool msg)

The documentation for this class was generated from the following file:

- [src/include/ExampleArg.h](#)

5.9 xmem::PrintUsageImplementation::FunctionWriter< Function > Struct Template Reference

Inheritance diagram for xmem::PrintUsageImplementation::FunctionWriter< Function >:



Public Member Functions

- virtual void [operator\(\)](#) (const char *str, int size)
Writes the given number of chars beginning at the given pointer somewhere.
- **FunctionWriter** (Function *w)

Public Attributes

- Function * **write**

The documentation for this struct was generated from the following file:

- [src/include/optionparser.h](#)

5.10 xmem::PrintUsageImplementation::IStringWriter Struct Reference

Inheritance diagram for xmem::PrintUsageImplementation::IStringWriter:



Public Member Functions

- virtual void [operator\(\)](#) (const char *, int)
Writes the given number of chars beginning at the given pointer somewhere.

The documentation for this struct was generated from the following file:

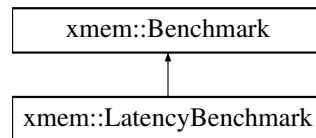
- [src/include/optionparser.h](#)

5.11 xmem::LatencyBenchmark Class Reference

A type of benchmark that measures memory latency via random pointer chasing. Loading may be provided with separate threads which access memory as quickly as possible using given access patterns.

```
#include <LatencyBenchmark.h>
```

Inheritance diagram for xmem::LatencyBenchmark:



Public Member Functions

- [LatencyBenchmark](#) (void *mem_array, size_t len, uint32_t iterations, uint32_t num_worker_threads, uint32_t mem_node, uint32_t cpu_node, pattern_mode_t pattern_mode, rw_mode_t rw_mode, chunk_size_t chunk_size, int32_t stride_size, std::vector< [PowerReader](#) * > dram_power_readers, std::string name)
Constructor. Parameters are passed directly to the [Benchmark](#) constructor. See [Benchmark](#) class documentation for parameter semantics.
- virtual [~LatencyBenchmark](#) ()
Destructor.
- double [getLoadMetricOnIter](#) (uint32_t iter) const
Get the average load throughput in MB/sec that was imposed on the latency measurement during the given iteration.
- double [getAvgLoadMetric](#) () const
Get the overall average load throughput in MB/sec that was imposed on the latency measurement.
- virtual void [report_benchmark_info](#) () const
Reports benchmark configuration details to the console.
- virtual void [report_results](#) () const
Reports results to the console.

Protected Member Functions

- virtual bool [_run_core](#) ()
The core benchmark function.

Protected Attributes

- std::vector< double > [_loadMetricOnIter](#)
- double [_averageLoadMetric](#)

5.11.1 Detailed Description

A type of benchmark that measures memory latency via random pointer chasing. Loading may be provided with separate threads which access memory as quickly as possible using given access patterns.

5.11.2 Member Function Documentation

5.11.2.1 bool LatencyBenchmark::_run_core () [protected], [virtual]

The core benchmark function.

Returns

True on success.

Implements [xmem::Benchmark](#).

5.11.2.2 double LatencyBenchmark::getAvgLoadMetric () const

Get the overall average load throughput in MB/sec that was imposed on the latency measurement.

Returns

The average throughput in MB/sec.

5.11.2.3 double LatencyBenchmark::getLoadMetricOnIter (uint32_t iter) const

Get the average load throughput in MB/sec that was imposed on the latency measurement during the given iteration.
iter The iteration of interest.

Returns

The average throughput in MB/sec.

5.11.3 Member Data Documentation

5.11.3.1 double xmem::LatencyBenchmark::_averageLoadMetric [protected]

The average load throughput in MB/sec that was imposed on the latency measurement.

5.11.3.2 std::vector<double> xmem::LatencyBenchmark::_loadMetricOnIter [protected]

Load metrics for each iteration of the benchmark. This is in MB/s.

The documentation for this class was generated from the following files:

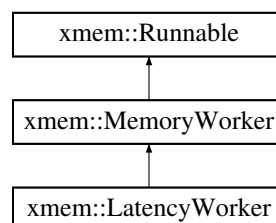
- [src/include/LatencyBenchmark.h](#)
- [src/LatencyBenchmark.cpp](#)

5.12 xmem::LatencyWorker Class Reference

Multithreading-friendly class to do memory loading.

```
#include <LatencyWorker.h>
```

Inheritance diagram for xmem::LatencyWorker:



Public Member Functions

- [LatencyWorker](#) (void *mem_array, size_t len, RandomFunction kernel_fptr, RandomFunction kernel_dummy_fptr, int32_t cpu_affinity)
Constructor for sequential-access patterns.

- virtual [~LatencyWorker](#) ()
Destructor.
- virtual void [run](#) ()
Thread-safe worker method.

Additional Inherited Members

5.12.1 Detailed Description

Multithreading-friendly class to do memory loading.

5.12.2 Constructor & Destructor Documentation

5.12.2.1 LatencyWorker::LatencyWorker (void * *mem_array*, size_t *len*, RandomFunction *kernel_fptr*, RandomFunction *kernel_dummy_fptr*, int32_t *cpu_affinity*)

Constructor for sequential-access patterns.

Parameters

<i>mem_array</i>	Pointer to the memory region to use by this worker.
<i>len</i>	Length of the memory region to use by this worker.
<i>kernel_fptr</i>	Pointer to the sequential core benchmark kernel to use.
<i>kernel_dummy_fptr</i>	Pointer to the sequential dummy version of the core benchmark kernel to use.
<i>cpu_affinity</i>	Logical CPU identifier to lock this worker's thread to.

The documentation for this class was generated from the following files:

- src/include/[LatencyWorker.h](#)
- src/[LatencyWorker.cpp](#)

5.13 xmem::PrintUsageImplementation::LinePartIterator Class Reference

Public Member Functions

- [LinePartIterator](#) (const [Descriptor](#) *usage*[])
*Creates an iterator for *usage*.*
- bool [nextTable](#) ()
Moves iteration to the next table (if any). Has to be called once on a new [LinePartIterator](#) to move to the 1st table.
- void [restartTable](#) ()
Reset iteration to the beginning of the current table.
- bool [nextRow](#) ()
Moves iteration to the next row (if any). Has to be called once after each call to [nextTable\(\)](#) to move to the 1st row of the table.
- void [restartRow](#) ()
Reset iteration to the beginning of the current row.
- bool [next](#) ()
Moves iteration to the next part (if any). Has to be called once after each call to [nextRow\(\)](#) to move to the 1st part of the row.
- int [column](#) ()
Returns the index (counting from 0) of the column in which the part pointed to by [data\(\)](#) is located.
- int [line](#) ()

Returns the index (counting from 0) of the line within the current column this part belongs to.

- int [length](#) ()

Returns the length of the part pointed to by [data\(\)](#) in raw chars (not UTF-8 characters).

- int [screenLength](#) ()

Returns the width in screen columns of the part pointed to by [data\(\)](#). Takes multi-byte UTF-8 sequences and wide characters into account.

- const char * [data](#) ()

Returns the current part of the iteration.

5.13.1 Member Function Documentation

5.13.1.1 bool xmem::PrintUsagImplementation::LinePartIterator::next () [inline]

Moves iteration to the next part (if any). Has to be called once after each call to [nextRow\(\)](#) to move to the 1st part of the row.

Return values

<i>false</i>	if moving to next part failed because no further part exists.
--------------	---

See [LinePartIterator](#) for details about the iteration.

5.13.1.2 bool xmem::PrintUsagImplementation::LinePartIterator::nextRow () [inline]

Moves iteration to the next row (if any). Has to be called once after each call to [nextTable\(\)](#) to move to the 1st row of the table.

Return values

<i>false</i>	if moving to next row failed because no further row exists.
--------------	---

5.13.1.3 bool xmem::PrintUsagImplementation::LinePartIterator::nextTable () [inline]

Moves iteration to the next table (if any). Has to be called once on a new [LinePartIterator](#) to move to the 1st table.

Return values

<i>false</i>	if moving to next table failed because no further table exists.
--------------	---

The documentation for this class was generated from the following file:

- [src/include/optionparser.h](#)

5.14 xmem::PrintUsagImplementation::LineWrapper Class Reference

Public Member Functions

- void [flush](#) (IStringWriter &write)

Writes out all remaining data from the [LineWrapper](#) using `write`. Unlike [process\(\)](#) this method indents all lines including the first and will output a `\n` at the end (but only if something has been written).

- void [process](#) (IStringWriter &write, const char *data, int len)

Process, wrap and output the next piece of data.

- [LineWrapper](#) (int x1, int x2)

Constructs a [LineWrapper](#) that wraps its output to fit into screen columns `x1` (incl.) to `x2` (excl.).

5.14.1 Constructor & Destructor Documentation

5.14.1.1 `xmem::PrintUsagImplementation::LineWrapper::LineWrapper (int x1, int x2) [inline]`

Constructs a [LineWrapper](#) that wraps its output to fit into screen columns `x1` (incl.) to `x2` (excl.).

`x1` gives the indentation [LineWrapper](#) uses if it needs to indent.

5.14.2 Member Function Documentation

5.14.2.1 `void xmem::PrintUsagImplementation::LineWrapper::process (IStringWriter & write, const char * data, int len) [inline]`

Process, wrap and output the next piece of data.

[process\(\)](#) will output at least one line of output. This is not necessarily the `data` passed in. It may be data queued from a prior call to [process\(\)](#). If the internal buffer is full, more than 1 line will be output.

[process\(\)](#) assumes that the a proper amount of indentation has already been output. It won't write any further indentation before the 1st line. If more than 1 line is written due to buffer constraints, the lines following the first will be indented by this method, though.

No `\n` is written by this method after the last line that is written.

Parameters

<i>write</i>	where to write the data.
<i>data</i>	the new chunk of data to write.
<i>len</i>	the length of the chunk of data to write.

The documentation for this class was generated from the following file:

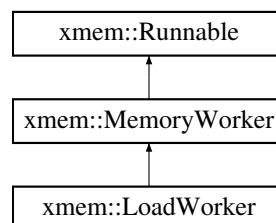
- `src/include/optionparser.h`

5.15 xmem::LoadWorker Class Reference

Multithreading-friendly class to do memory loading.

```
#include <LoadWorker.h>
```

Inheritance diagram for `xmem::LoadWorker`:



Public Member Functions

- [LoadWorker](#) (void *mem_array, size_t len, SequentialFunction kernel_fptr, SequentialFunction kernel_dummy_fptr, int32_t cpu_affinity)
Constructor for sequential-access patterns.
- [LoadWorker](#) (void *mem_array, size_t len, RandomFunction kernel_fptr, RandomFunction kernel_dummy_fptr, int32_t cpu_affinity)

Constructor for random-access patterns.

- virtual [~LoadWorker](#) ()

Destructor.

- virtual void [run](#) ()

Thread-safe worker method.

Additional Inherited Members

5.15.1 Detailed Description

Multithreading-friendly class to do memory loading.

5.15.2 Constructor & Destructor Documentation

5.15.2.1 LoadWorker::LoadWorker (void * *mem_array*, size_t *len*, SequentialFunction *kernel_fptr*, SequentialFunction *kernel_dummy_fptr*, int32_t *cpu_affinity*)

Constructor for sequential-access patterns.

Parameters

<i>mem_array</i>	Pointer to the memory region to use by this worker.
<i>len</i>	Length of the memory region to use by this worker.
<i>kernel_fptr</i>	Pointer to the sequential core benchmark kernel to use.
<i>kernel_dummy_fptr</i>	Pointer to the sequential dummy version of the core benchmark kernel to use.
<i>cpu_affinity</i>	Logical CPU identifier to lock this worker's thread to.

5.15.2.2 LoadWorker::LoadWorker (void * *mem_array*, size_t *len*, RandomFunction *kernel_fptr*, RandomFunction *kernel_dummy_fptr*, int32_t *cpu_affinity*)

Constructor for random-access patterns.

Parameters

<i>mem_array</i>	Pointer to the memory region to use by this worker.
<i>len</i>	Length of the memory region to use by this worker.
<i>kernel_fptr</i>	Pointer to the random core benchmark kernel to use.
<i>kernel_dummy_fptr</i>	Pointer to the random dummy version of the core benchmark kernel to use.
<i>cpu_affinity</i>	Logical CPU identifier to lock this worker's thread to.

The documentation for this class was generated from the following files:

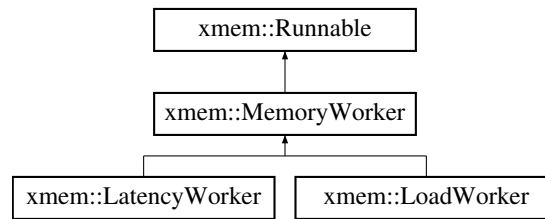
- src/include/[LoadWorker.h](#)
- src/[LoadWorker.cpp](#)

5.16 xmem::MemoryWorker Class Reference

Multithreading-friendly class to run memory access kernels.

```
#include <MemoryWorker.h>
```

Inheritance diagram for xmem::MemoryWorker:



Public Member Functions

- [MemoryWorker](#) (void *mem_array, size_t len, int32_t cpu_affinity)
Constructor.
- virtual [~MemoryWorker](#) ()
Destructor.
- virtual void [run](#) ()=0
Thread-safe worker method.
- size_t [getLen](#) ()
Gets the length of the memory region used by this worker.
- uint32_t [getBytesPerPass](#) ()
Gets the number of bytes used in each pass of the benchmark by this worker.
- uint32_t [getPasses](#) ()
Gets the number of passes for this worker.
- tick_t [getElapsedTicks](#) ()
Gets the elapsed ticks for this worker on the core benchmark kernel.
- tick_t [getElapsedDummyTicks](#) ()
Gets the elapsed ticks for this worker on the dummy version of the core benchmark kernel.
- tick_t [getAdjustedTicks](#) ()
Gets the adjusted ticks for this worker. This is elapsed ticks minus elapsed dummy ticks.
- bool [hadWarning](#) ()
Indicates whether worker's results may be questionable/inaccurate/invalid.

Protected Attributes

- void * [_mem_array](#)
- size_t [_len](#)
- int32_t [_cpu_affinity](#)
- uint32_t [_bytes_per_pass](#)
- uint32_t [_passes](#)
- tick_t [_elapsed_ticks](#)
- tick_t [_elapsed_dummy_ticks](#)
- tick_t [_adjusted_ticks](#)
- bool [_warning](#)
- bool [_completed](#)

Additional Inherited Members

5.16.1 Detailed Description

Multithreading-friendly class to run memory access kernels.

5.16.2 Constructor & Destructor Documentation

5.16.2.1 MemoryWorker::MemoryWorker (void * *mem_array*, size_t *len*, int32_t *cpu_affinity*)

Constructor.

Parameters

<i>mem_array</i>	Pointer to the memory region to use by this worker.
<i>len</i>	Length of the memory region to use by this worker.
<i>passes_per_iteration</i>	for size-based benchmarking, this is the number of passes to execute in a single benchmark iteration.
<i>cpu_affinity</i>	Logical CPU identifier to lock this worker's thread to.

5.16.3 Member Function Documentation

5.16.3.1 `tick_t MemoryWorker::getAdjustedTicks ()`

Gets the adjusted ticks for this worker. This is elapsed ticks minus elapsed dummy ticks.

Returns

The adjusted ticks for this worker.

5.16.3.2 `uint32_t MemoryWorker::getBytesPerPass ()`

Gets the number of bytes used in each pass of the benchmark by this worker.

Returns

Number of bytes in each pass.

5.16.3.3 `tick_t MemoryWorker::getElapsedDummyTicks ()`

Gets the elapsed ticks for this worker on the dummy version of the core benchmark kernel.

Returns

The number of elapsed dummy ticks.

5.16.3.4 `tick_t MemoryWorker::getElapsedTicks ()`

Gets the elapsed ticks for this worker on the core benchmark kernel.

Returns

The number of elapsed ticks.

5.16.3.5 `size_t MemoryWorker::getLen ()`

Gets the length of the memory region used by this worker.

Returns

Length of memory region in bytes.

5.16.3.6 uint32_t MemoryWorker::getPasses ()

Gets the number of passes for this worker.

Returns

The number of passes.

5.16.3.7 bool MemoryWorker::hadWarning ()

Indicates whether worker's results may be questionable/inaccurate/invalid.

Returns

True if the worker's results had a warning.

5.16.4 Member Data Documentation

5.16.4.1 tick_t xmem::MemoryWorker::_adjusted_ticks [protected]

Elapsed ticks minus dummy elapsed ticks.

5.16.4.2 uint32_t xmem::MemoryWorker::_bytes_per_pass [protected]

Number of bytes accessed in each kernel pass.

5.16.4.3 bool xmem::MemoryWorker::_completed [protected]

If true, worker completed.

5.16.4.4 int32_t xmem::MemoryWorker::_cpu_affinity [protected]

The logical CPU affinity for this worker.

5.16.4.5 tick_t xmem::MemoryWorker::_elapsed_dummy_ticks [protected]

Total elapsed ticks on the dummy kernel routine.

5.16.4.6 tick_t xmem::MemoryWorker::_elapsed_ticks [protected]

Total elapsed ticks on the kernel routine.

5.16.4.7 size_t xmem::MemoryWorker::_len [protected]

The length of the memory region for this worker.

5.16.4.8 void* xmem::MemoryWorker::_mem_array [protected]

The memory region for this worker.

5.16.4.9 uint32_t xmem::MemoryWorker::_passes [protected]

Number of passes.

5.16.4.10 bool xmem::MemoryWorker::_warning [protected]

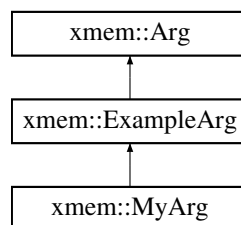
If true, results may be suspect.

The documentation for this class was generated from the following files:

- src/include/[MemoryWorker.h](#)
- src/[MemoryWorker.cpp](#)

5.17 xmem::MyArg Class Reference

Inheritance diagram for xmem::MyArg:



Static Public Member Functions

- static ArgStatus [Integer](#) (const [Option](#) &option, bool msg)
Checks an option that it is an integer.
- static ArgStatus [NonnegativeInteger](#) (const [Option](#) &option, bool msg)
Checks an option that it is a nonnegative integer.
- static ArgStatus [PositiveInteger](#) (const [Option](#) &option, bool msg)
Checks an option that it is a positive integer.

The documentation for this class was generated from the following file:

- src/include/[MyArg.h](#)

5.18 xmem::Option Class Reference

A parsed option from the command line together with its argument if it has one.

```
#include <optionparser.h>
```

Public Member Functions

- int [type](#) () const
Returns [Descriptor::type](#) of this [Option](#)'s [Descriptor](#), or 0 if this [Option](#) is invalid (unused).
- int [index](#) () const
Returns [Descriptor::index](#) of this [Option](#)'s [Descriptor](#), or -1 if this [Option](#) is invalid (unused).

- `int count ()`
Returns the number of times this `Option` (or others with the same `Descriptor::index`) occurs in the argument vector.
- `bool isFirst () const`
Returns true iff this is the first element of the linked list.
- `bool isLast () const`
Returns true iff this is the last element of the linked list.
- `Option * first ()`
Returns a pointer to the first element of the linked list.
- `Option * last ()`
Returns a pointer to the last element of the linked list.
- `Option * prev ()`
Returns a pointer to the previous element of the linked list or NULL if called on `first()`.
- `Option * prevwrap ()`
Returns a pointer to the previous element of the linked list with wrap-around from `first()` to `last()`.
- `Option * next ()`
Returns a pointer to the next element of the linked list or NULL if called on `last()`.
- `Option * nextwrap ()`
Returns a pointer to the next element of the linked list with wrap-around from `last()` to `first()`.
- `void append (Option *new_last)`
Makes `new_last` the new `last()` by chaining it into the list after `last()`.
- `operator const Option * () const`
Casts from `Option` to `const Option*` but only if this `Option` is valid.
- `operator Option * ()`
Casts from `Option` to `Option*` but only if this `Option` is valid.
- `Option ()`
Creates a new `Option` that is a one-element linked list and has NULL `desc`, `name`, `arg` and `namelen`.
- `Option (const Descriptor *desc_, const char *name_, const char *arg_)`
Creates a new `Option` that is a one-element linked list and has the given values for `desc`, `name` and `arg`.
- `void operator= (const Option &orig)`
Makes `*this` a copy of `orig` except for the linked list pointers.
- `Option (const Option &orig)`
Makes `*this` a copy of `orig` except for the linked list pointers.

Public Attributes

- `const Descriptor * desc`
Pointer to this `Option`'s `Descriptor`.
- `const char * name`
The name of the option as used on the command line.
- `const char * arg`
Pointer to this `Option`'s argument (if any).
- `int namelen`
The length of the option `name`.

5.18.1 Detailed Description

A parsed option from the command line together with its argument if it has one.

The [Parser](#) chains all parsed options with the same [Descriptor::index](#) together to form a linked list. This allows you to easily implement all of the common ways of handling repeated options and enable/disable pairs.

- Test for presence of a switch in the argument vector:

```
if ( options[QUIET] ) ...
```

- Evaluate `--enable-foo/--disable-foo` pair where the last one used wins:

```
if ( options[FOO].last()->type() == DISABLE ) ...
```

- Cumulative option (`-v` verbose, `-vv` more verbose, `-vvv` even more verbose):

```
int verbosity = options[VERBOSE].count();
```

- Iterate over all `--file=<fname>` arguments:

```
for (Option* opt = options[FILE]; opt; opt = opt->next())
    fname = opt->arg; ...
```

5.18.2 Constructor & Destructor Documentation

5.18.2.1 `xmem::Option::Option (const Descriptor * desc_, const char * name_, const char * arg_) [inline]`

Creates a new [Option](#) that is a one-element linked list and has the given values for [desc](#), [name](#) and [arg](#).

If `name_` points at a character other than `'-'` it will be assumed to refer to a short option and [namelen](#) will be set to 1. Otherwise the length will extend to the first `'='` character or the string's 0-terminator.

5.18.2.2 `xmem::Option::Option (const Option & orig) [inline]`

Makes `*this` a copy of `orig` except for the linked list pointers.

After this operation `*this` will be a one-element linked list.

5.18.3 Member Function Documentation

5.18.3.1 `void xmem::Option::append (Option * new_last) [inline]`

Makes `new_last` the new [last\(\)](#) by chaining it into the list after [last\(\)](#).

It doesn't matter which element you call [append\(\)](#) on. The new element will always be appended to [last\(\)](#).

Attention

`new_last` must not yet be part of a list, or that list will become corrupted, because this method does not unchain `new_last` from an existing list.

5.18.3.2 `int xmem::Option::count () [inline]`

Returns the number of times this [Option](#) (or others with the same [Descriptor::index](#)) occurs in the argument vector.

This corresponds to the number of elements in the linked list this [Option](#) is part of. It doesn't matter on which element you call [count\(\)](#). The return value is always the same.

Use this to implement cumulative options, such as `-v`, `-vv`, `-vvv` for different verbosity levels.

Returns 0 when called for an unused/invalid option.

5.18.3.3 `Option* xmem::Option::first () [inline]`

Returns a pointer to the first element of the linked list.

Use this when you want the first occurrence of an option on the command line to take precedence. Note that this is not the way most programs handle options. You should probably be using `last()` instead.

Note

This method may be called on an unused/invalid option and will return a pointer to the option itself.

5.18.3.4 `bool xmem::Option::isFirst () const [inline]`

Returns true iff this is the first element of the linked list.

The first element in the linked list is the first option on the command line that has the respective `Descriptor::index` value.

Returns true for an unused/invalid option.

5.18.3.5 `bool xmem::Option::isLast () const [inline]`

Returns true iff this is the last element of the linked list.

The last element in the linked list is the last option on the command line that has the respective `Descriptor::index` value.

Returns true for an unused/invalid option.

5.18.3.6 `Option* xmem::Option::last () [inline]`

Returns a pointer to the last element of the linked list.

Use this when you want the last occurrence of an option on the command line to take precedence. This is the most common way of handling conflicting options.

Note

This method may be called on an unused/invalid option and will return a pointer to the option itself.

Tip:

If you have options with opposite meanings (e.g. `-enable-foo` and `-disable-foo`), you can assign them the same `Descriptor::index` to get them into the same list. Distinguish them by `Descriptor::type` and all you have to do is check `last() -> type()` to get the state listed last on the command line.

5.18.3.7 `Option* xmem::Option::next () [inline]`

Returns a pointer to the next element of the linked list or NULL if called on `last()`.

If called on `last()` this method returns NULL. Otherwise it will return the option with the same `Descriptor::index` that follows this option on the command line.

5.18.3.8 `Option* xmem::Option::nextwrap () [inline]`

Returns a pointer to the next element of the linked list with wrap-around from `last()` to `first()`.

If called on `last()` this method returns `first()`. Otherwise it will return the option with the same `Descriptor::index` that follows this option on the command line.

5.18.3.9 `xmem::Option::operator const Option * () const` `[inline]`

Casts from `Option` to `const Option*` but only if this `Option` is valid.

If this `Option` is valid (i.e. `desc!=NULL`), returns this. Otherwise returns `NULL`. This allows testing an `Option` directly in an if-clause to see if it is used:

```
if (options[CREATE])
{
    ...
}
```

It also allows you to write loops like this:

```
for (Option* opt = options[FILE]; opt; opt = opt->next())
    fname = opt->arg; ...
```

5.18.3.10 `xmem::Option::operator Option * ()` `[inline]`

Casts from `Option` to `Option*` but only if this `Option` is valid.

If this `Option` is valid (i.e. `desc!=NULL`), returns this. Otherwise returns `NULL`. This allows testing an `Option` directly in an if-clause to see if it is used:

```
if (options[CREATE])
{
    ...
}
```

It also allows you to write loops like this:

```
for (Option* opt = options[FILE]; opt; opt = opt->next())
    fname = opt->arg; ...
```

5.18.3.11 `void xmem::Option::operator= (const Option & orig)` `[inline]`

Makes `*this` a copy of `orig` except for the linked list pointers.

After this operation `*this` will be a one-element linked list.

5.18.3.12 `Option* xmem::Option::prev ()` `[inline]`

Returns a pointer to the previous element of the linked list or `NULL` if called on `first()`.

If called on `first()` this method returns `NULL`. Otherwise it will return the option with the same `Descriptor::index` that precedes this option on the command line.

5.18.3.13 `Option* xmem::Option::prevwrap ()` `[inline]`

Returns a pointer to the previous element of the linked list with wrap-around from `first()` to `last()`.

If called on `first()` this method returns `last()`. Otherwise it will return the option with the same `Descriptor::index` that precedes this option on the command line.

5.18.3.14 `int xmem::Option::type () const` `[inline]`

Returns `Descriptor::type` of this `Option`'s `Descriptor`, or 0 if this `Option` is invalid (unused).

Because this method (and `last()`, too) can be used even on unused Options with `desc==0`, you can (provided you arrange your types properly) switch on `type()` without testing validity first.


```
enum OptionType { UNUSED=0, DISABLED=0, ENABLED=1 };
enum OptionIndex { FOO };
const Descriptor usage[] = {
    { FOO, ENABLED, "", "enable-foo", Arg::None, 0 },
    { FOO, DISABLED, "", "disable-foo", Arg::None, 0 },
    { 0, 0, 0, 0, 0, 0 } };
...
switch(options[FOO].last()->type()) // no validity check required!
{
    case ENABLED: ...
    case DISABLED: ... // UNUSED==DISABLED !
}
```

5.18.4 Member Data Documentation

5.18.4.1 const char* xmem::Option::arg

Pointer to this [Option](#)'s argument (if any).

NULL if this option has no argument. Do not confuse this with the empty string which is a valid argument.

5.18.4.2 const Descriptor* xmem::Option::desc

Pointer to this [Option](#)'s [Descriptor](#).

Remember that the first dummy descriptor (see [Descriptor::longopt](#)) is used for unknown options.

Attention

`desc==NULL` signals that this [Option](#) is unused. This is the default state of elements in the result array. You don't need to test `desc` explicitly. You can simply write something like this:

```
if (options[CREATE])
{
    ...
}
```

This works because of operator `const Option*()` .

5.18.4.3 const char* xmem::Option::name

The name of the option as used on the command line.

The main purpose of this string is to be presented to the user in messages.

In the case of a long option, this is the actual `argv` pointer, i.e. the first character is a '-'. In the case of a short option this points to the option character within the `argv` string.

Note that in the case of a short option group or an attached option argument, this string will contain additional characters following the actual name. Use [namelen](#) to filter out the actual option name only.

5.18.4.4 int xmem::Option::namelen

The length of the option [name](#).

Because [name](#) points into the actual `argv` string, the option name may be followed by more characters (e.g. other short options in the same short option group). This value is the number of bytes (not characters!) that are part of the actual name.

For a short option, this length is always 1. For a long option this length is always at least 2 if single minus long options are permitted and at least 3 if they are disabled.

Note

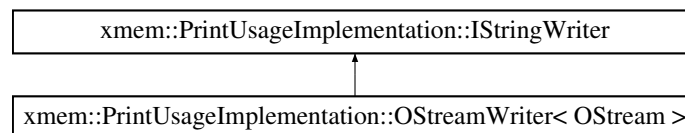
In the pathological case of a minus within a short option group (e.g. `-xf-z`), this length is incorrect, because this case will be misinterpreted as a long option and the name will therefore extend to the string's 0-terminator or a following '=' character if there is one. This is irrelevant for most uses of `name` and `namelen`. If you really need to distinguish the case of a long and a short option, compare `name` to the `argv` pointers. A long option's `name` is always identical to one of them, whereas a short option's is never.

The documentation for this class was generated from the following file:

- `src/include/optionparser.h`

5.19 `xmem::PrintUsageImplementation::OStreamWriter< OStream >` Struct Template Reference

Inheritance diagram for `xmem::PrintUsageImplementation::OStreamWriter< OStream >`:

**Public Member Functions**

- virtual void `operator()` (const char *str, int size)
Writes the given number of chars beginning at the given pointer somewhere.
- **`OStreamWriter`** (`OStream` &o)

Public Attributes

- `OStream` & **`ostream`**

The documentation for this struct was generated from the following file:

- `src/include/optionparser.h`

5.20 `xmem::Parser` Class Reference

Checks argument vectors for validity and parses them into data structures that are easier to work with.

```
#include <optionparser.h>
```

Classes

- struct `Action`
- class `StoreOptionAction`

Public Member Functions

- [Parser](#) ()
Creates a new [Parser](#).
- [Parser](#) (bool gnu, const [Descriptor](#) usage[], int argc, const char **argv, [Option](#) options[], [Option](#) buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)
Creates a new [Parser](#) and immediately parses the given argument vector.
- [Parser](#) (bool gnu, const [Descriptor](#) usage[], int argc, char **argv, [Option](#) options[], [Option](#) buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)
[Parser](#)(...) with non-const argv.
- [Parser](#) (const [Descriptor](#) usage[], int argc, const char **argv, [Option](#) options[], [Option](#) buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)
POSIX [Parser](#)(...) (gnu==false).
- [Parser](#) (const [Descriptor](#) usage[], int argc, char **argv, [Option](#) options[], [Option](#) buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)
POSIX [Parser](#)(...) (gnu==false) with non-const argv.
- void [parse](#) (bool gnu, const [Descriptor](#) usage[], int argc, const char **argv, [Option](#) options[], [Option](#) buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)
Parses the given argument vector.
- void [parse](#) (bool gnu, const [Descriptor](#) usage[], int argc, char **argv, [Option](#) options[], [Option](#) buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)
[parse](#)() with non-const argv.
- void [parse](#) (const [Descriptor](#) usage[], int argc, const char **argv, [Option](#) options[], [Option](#) buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)
POSIX [parse](#)() (gnu==false).
- void [parse](#) (const [Descriptor](#) usage[], int argc, char **argv, [Option](#) options[], [Option](#) buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)
POSIX [parse](#)() (gnu==false) with non-const argv.
- int [optionsCount](#) ()
Returns the number of valid [Option](#) objects in `buffer[]`.
- int [nonOptionsCount](#) ()
Returns the number of non-option arguments that remained at the end of the most recent [parse](#)() that actually encountered non-option arguments.
- const char ** [nonOptions](#) ()
Returns a pointer to an array of non-option arguments (only valid if `nonOptionsCount()` > 0).
- const char * [nonOption](#) (int i)
Returns `nonOptions()` [i] (without checking if i is in range!).
- bool [error](#) ()
Returns `true` if an unrecoverable error occurred while parsing options.

Friends

- struct **Stats**

5.20.1 Detailed Description

Checks argument vectors for validity and parses them into data structures that are easier to work with.

Example:

```
int main(int argc, char* argv[])
{
    argc--=(argc>0); argv+=(argc>0); // skip program name argv[0] if present
    option::Stats stats(usage, argc, argv);
```

```

option::Option options[stats.options_max], buffer[stats.buffer_max];
option::Parser parse(usage, argc, argv, options, buffer);

if (parse.error())
    return 1;

if (options[HELP])
    ...

```

5.20.2 Constructor & Destructor Documentation

5.20.2.1 `xmem::Parser::Parser (bool gnu, const Descriptor usage[], int argc, const char ** argv, Option options[], Option buffer[], int min_abbr_len = 0, bool single_minus_longopt = false, int bufmax = -1)` `[inline]`

Creates a new [Parser](#) and immediately parses the given argument vector.

Parameters

<i>gnu</i>	if true, parse() will not stop at the first non-option argument. Instead it will reorder arguments so that all non-options are at the end. This is the default behaviour of GNU getopt() but is not conforming to POSIX. Note, that once the argument vector has been reordered, the <code>gnu</code> flag will have no further effect on this argument vector. So it is enough to pass <code>gnu==true</code> when creating Stats .
<i>usage</i>	Array of Descriptor objects that describe the options to support. The last entry of this array must have 0 in all fields.
<i>argc</i>	The number of elements from <i>argv</i> that are to be parsed. If you pass -1, the number will be determined automatically. In that case the <i>argv</i> list must end with a NULL pointer.
<i>argv</i>	The arguments to be parsed. If you pass -1 as <i>argc</i> the last pointer in the <i>argv</i> list must be NULL to mark the end.
<i>options</i>	Each entry is the first element of a linked list of Options. Each new option that is parsed will be appended to the list specified by that Option's Descriptor::index . If an entry is not yet used (i.e. the Option is invalid), it will be replaced rather than appended to. The minimum length of this array is the greatest Descriptor::index value that occurs in <i>usage</i> PLUS ONE.
<i>buffer</i>	Each argument that is successfully parsed (including unknown arguments, if they have a Descriptor whose CheckArg does not return ARG_ILLEGAL) will be stored in this array. parse() scans the array for the first invalid entry and begins writing at that index. You can pass <i>bufmax</i> to limit the number of options stored.
<i>min_abbr_len</i>	Passing a value <code>min_abbr_len > 0</code> enables abbreviated long options. The parser will match a prefix of a long option as if it was the full long option (e.g. <code>-foob=10</code> will be interpreted as if it was <code>-foobar=10</code>), as long as the prefix has at least <i>min_abbr_len</i> characters (not counting the <code>-</code>) and is unambiguous. Be careful if combining <code>min_abbr_len=1</code> with <code>single_minus_longopt=true</code> because the ambiguity check does not consider short options and abbreviated single minus long options will take precedence over short options.
<i>single_minus_longopt</i>	Passing <code>true</code> for this option allows long options to begin with a single minus. The double minus form will still be recognized. Note that single minus long options take precedence over short options and short option groups. E.g. <code>-file</code> would be interpreted as <code>-file</code> and not as <code>-f -i -l -e</code> (assuming a long option named "file" exists).
<i>bufmax</i>	The greatest index in the <i>buffer</i> [] array that parse() will write to is <code>bufmax-1</code> . If there are more options, they will be processed (in particular their CheckArg will be called) but not stored. If you used Stats::buffer_max to dimension this array, you can pass -1 (or not pass <i>bufmax</i> at all) which tells parse() that the buffer is "large enough".

Attention

Remember that *options* and *buffer* store [Option objects](#), not pointers. Therefore it is not possible for the same object to be in both arrays. For those options that are found in both *buffer*[] and *options*[] the respective objects are independent copies. And only the objects in *options*[] are properly linked via [Option::next\(\)](#) and [Option::prev\(\)](#). You can iterate over *buffer*[] to process all options in the order they appear in the

argument vector, but if you want access to the other Options with the same [Descriptor::index](#), then you *must* access the linked list via `options[]`. You can get the linked list in options from a buffer object via something like `options[buffer[i].index()]`.

5.20.3 Member Function Documentation

5.20.3.1 `bool xmem::Parser::error () [inline]`

Returns `true` if an unrecoverable error occurred while parsing options.

An illegal argument to an option (i.e. `CheckArg` returns `ARG_ILLEGAL`) is an unrecoverable error that aborts the parse. Unknown options are only an error if their `CheckArg` function returns `ARG_ILLEGAL`. Otherwise they are collected. In that case if you want to exit the program if either an illegal argument or an unknown option has been passed, use code like this

```
if (parser.error() || options[UNKNOWN])
    exit(1);
```

5.20.3.2 `const char** xmem::Parser::nonOptions () [inline]`

Returns a pointer to an array of non-option arguments (only valid if `nonOptionsCount () > 0`).

Note

- `parse()` does not copy arguments, so this pointer points into the actual argument vector as passed to `parse()`.
- As explained at `nonOptionsCount()` this pointer is only changed by `parse()` calls that actually encounter non-option arguments. A `parse()` call that encounters only options, will not change `nonOptions()`.

5.20.3.3 `int xmem::Parser::nonOptionsCount () [inline]`

Returns the number of non-option arguments that remained at the end of the most recent `parse()` that actually encountered non-option arguments.

Note

A `parse()` that does not encounter non-option arguments will leave this value as well as `nonOptions()` undisturbed. This means you can feed the [Parser](#) a default argument vector that contains non-option arguments (e.g. a default filename). Then you feed it the actual arguments from the user. If the user has supplied at least one non-option argument, all of the non-option arguments from the default disappear and are replaced by the user's non-option arguments. However, if the user does not supply any non-option arguments the defaults will still be in effect.

5.20.3.4 `int xmem::Parser::optionsCount () [inline]`

Returns the number of valid [Option](#) objects in `buffer[]`.

Note

- The returned value always reflects the number of Options in the `buffer[]` array used for the most recent call to `parse()`.
- The count (and the `buffer[]`) includes unknown options if they are collected (see [Descriptor::longopt](#)).

5.20.3.5 `void xmem::Parser::parse (bool gnu, const Descriptor usage[], int argc, const char ** argv, Option options[], Option buffer[], int min_abbrev_len = 0, bool single_minus_longopt = false, int bufmax = -1)` `[inline]`

Parses the given argument vector.

Parameters

<i>gnu</i>	if true, parse() will not stop at the first non-option argument. Instead it will reorder arguments so that all non-options are at the end. This is the default behaviour of GNU getopt() but is not conforming to POSIX. Note, that once the argument vector has been reordered, the <code>gnu</code> flag will have no further effect on this argument vector. So it is enough to pass <code>gnu==true</code> when creating Stats .
<i>usage</i>	Array of Descriptor objects that describe the options to support. The last entry of this array must have 0 in all fields.
<i>argc</i>	The number of elements from <code>argv</code> that are to be parsed. If you pass -1, the number will be determined automatically. In that case the <code>argv</code> list must end with a NULL pointer.
<i>argv</i>	The arguments to be parsed. If you pass -1 as <code>argc</code> the last pointer in the <code>argv</code> list must be NULL to mark the end.
<i>options</i>	Each entry is the first element of a linked list of Options. Each new option that is parsed will be appended to the list specified by that Option's Descriptor::index . If an entry is not yet used (i.e. the Option is invalid), it will be replaced rather than appended to. The minimum length of this array is the greatest Descriptor::index value that occurs in <code>usage</code> PLUS ONE.
<i>buffer</i>	Each argument that is successfully parsed (including unknown arguments, if they have a Descriptor whose <code>CheckArg</code> does not return <code>ARG_ILLEGAL</code>) will be stored in this array. parse() scans the array for the first invalid entry and begins writing at that index. You can pass <code>bufmax</code> to limit the number of options stored.
<i>min_abbr_len</i>	Passing a value <code>min_abbr_len > 0</code> enables abbreviated long options. The parser will match a prefix of a long option as if it was the full long option (e.g. <code>-foob=10</code> will be interpreted as if it was <code>-foobar=10</code>), as long as the prefix has at least <code>min_abbr_len</code> characters (not counting the <code>-</code>) and is unambiguous. Be careful if combining <code>min_abbr_len=1</code> with <code>single_minus_longopt=true</code> because the ambiguity check does not consider short options and abbreviated single minus long options will take precedence over short options.
<i>single_minus_longopt</i>	Passing <code>true</code> for this option allows long options to begin with a single minus. The double minus form will still be recognized. Note that single minus long options take precedence over short options and short option groups. E.g. <code>-file</code> would be interpreted as <code>-file</code> and not as <code>-f -i -l -e</code> (assuming a long option named "file" exists).
<i>bufmax</i>	The greatest index in the <code>buffer[]</code> array that parse() will write to is <code>bufmax-1</code> . If there are more options, they will be processed (in particular their <code>CheckArg</code> will be called) but not stored. If you used Stats::buffer_max to dimension this array, you can pass -1 (or not pass <code>bufmax</code> at all) which tells parse() that the buffer is "large enough".

Attention

Remember that `options` and `buffer` store [Option objects](#), not pointers. Therefore it is not possible for the same object to be in both arrays. For those options that are found in both `buffer[]` and `options[]` the respective objects are independent copies. And only the objects in `options[]` are properly linked via [Option::next\(\)](#) and [Option::prev\(\)](#). You can iterate over `buffer[]` to process all options in the order they appear in the argument vector, but if you want access to the other Options with the same [Descriptor::index](#), then you *must* access the linked list via `options[]`. You can get the linked list in options from a buffer object via something like `options[buffer[i].index()]`.

The documentation for this class was generated from the following file:

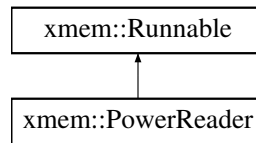
- [src/include/optionparser.h](#)

5.21 xmem::PowerReader Class Reference

An abstract base class for measuring power from an arbitrary source. This class is runnable using a worker thread.

```
#include <PowerReader.h>
```

Inheritance diagram for xmem::PowerReader:



Public Member Functions

- [PowerReader](#) (uint32_t sampling_period, double power_units, std::string [name](#), int32_t cpu_affinity)
Constructor.
- [~PowerReader](#) ()
Destructor.
- virtual void [run](#) ()=0
Starts measuring power at the rate implied by the sampling_period passed in the constructor. Call [stop\(\)](#) to indicate to stop measuring.
- bool [stop](#) ()
Signals to stop measuring power. This is a non-blocking call and return does not indicate the measurement has actually stopped.
- bool [calculateMetrics](#) ()
Calculates the relevant metrics.
- bool [clear](#) ()
Clears the stored power data.
- bool [clear_and_reset](#) ()
Clears the stored power data and resets state so that a new thread can be used with this object.
- std::vector< double > [getPowerTrace](#) ()
Gets the power trace.
- double [getAveragePower](#) ()
Gets the average power.
- double [getPeakPower](#) ()
Gets the peak power.
- double [getLastSample](#) ()
Gets the last sample.
- uint32_t [getSamplingPeriod](#) ()
Gets the sampling period.
- double [getPowerUnits](#) ()
Gets the units of samples in watts.
- size_t [getNumSamples](#) ()
Gets the number of samples collected.
- std::string [name](#) ()
Gets the name of this object.

Protected Attributes

- bool [_stop_signal](#)
- double [_power_units](#)
- std::string [_name](#)
- int32_t [_cpu_affinity](#)
- std::vector< double > [_power_trace](#)

- double [_average_power](#)
- double [_peak_power](#)
- size_t [_num_samples](#)
- uint32_t [_sampling_period](#)

Additional Inherited Members

5.21.1 Detailed Description

An abstract base class for measuring power from an arbitrary source. This class is runnable using a worker thread.

5.21.2 Constructor & Destructor Documentation

5.21.2.1 `PowerReader::PowerReader (uint32_t sampling_period, double power_units, std::string name, int32_t cpu_affinity)`

Constructor.

Parameters

<i>sampling_period</i>	The time between power samples in milliseconds.
<i>power_units</i>	The power units for each sample in watts.
<i>name</i>	The human-friendly name of this object.
<i>cpu_affinity</i>	The logical CPU to be used by the thread calling this object's run() method. If negative, any CPU is OK (no affinity).

5.21.3 Member Function Documentation

5.21.3.1 `bool PowerReader::calculateMetrics ()`

Calculates the relevant metrics.

Returns

True on success.

5.21.3.2 `bool PowerReader::clear ()`

Clears the stored power data.

Returns

True on success.

5.21.3.3 `bool PowerReader::clear_and_reset ()`

Clears the stored power data and resets state so that a new thread can be used with this object.

Returns

True on success.

5.21.3.4 `double PowerReader::getAveragePower ()`

Gets the average power.

Returns

The average power from the measurements. If no data was collected, returns 0.

5.21.3.5 `double PowerReader::getLastSample ()`

Gets the last sample.

Returns

The last power sample measured.

5.21.3.6 `size_t PowerReader::getNumSamples ()`

Gets the number of samples collected.

Returns

Number of samples collected.

5.21.3.7 `double PowerReader::getPeakPower ()`

Gets the peak power.

Returns

The peak power sample from the measurements. If no data was collected, returns 0.

5.21.3.8 `std::vector< double > PowerReader::getPowerTrace ()`

Gets the power trace.

Returns

The measured power trace in a vector. If no data was collected, the vector will be empty.

5.21.3.9 `double PowerReader::getPowerUnits ()`

Gets the units of samples in watts.

Returns

The power units for each measurement sample in watts. For example, if each measurement is in milliwatts, then this returns 1e-3.

5.21.3.10 uint32_t PowerReader::getSamplingPeriod ()

Gets the sampling period.

Returns

The sampling period of the measurements in milliseconds.

5.21.3.11 std::string PowerReader::name ()

Gets the name of this object.

Returns

The human-friendly name of this [PowerReader](#).

5.21.3.12 bool PowerReader::stop ()

Signals to stop measuring power. This is a non-blocking call and return does not indicate the measurement has actually stopped.

Returns

True if it successfully signaled a stop.

5.21.4 Member Data Documentation

5.21.4.1 double xmem::PowerReader::_average_power [protected]

The average power.

5.21.4.2 int32_t xmem::PowerReader::_cpu_affinity [protected]

CPU affinity for any thread using this object's [run\(\)](#) method. If negative, no affinity preference.

5.21.4.3 std::string xmem::PowerReader::_name [protected]

Name of this object.

5.21.4.4 size_t xmem::PowerReader::_num_samples [protected]

The number of samples collected.

5.21.4.5 double xmem::PowerReader::_peak_power [protected]

The peak power observed.

5.21.4.6 std::vector<double> xmem::PowerReader::_power_trace [protected]

The time-ordered list of power samples. The first index is the oldest measurement.

5.21.4.7 `double xmem::PowerReader::_power_units` [protected]

Power units in watts.

5.21.4.8 `uint32_t xmem::PowerReader::_sampling_period` [protected]

Power sampling period in milliseconds.

5.21.4.9 `bool xmem::PowerReader::_stop_signal` [protected]

When true, the `run()` function should finish after the current sample iteration it is working on.

The documentation for this class was generated from the following files:

- `src/include/PowerReader.h`
- `src/PowerReader.cpp`

5.22 `xmem::PrintUsageImplementation` Struct Reference

Classes

- struct `FunctionWriter`
- struct `IStringWriter`
- class `LinePartIterator`
- class `LineWrapper`
- struct `OStreamWriter`
- struct `StreamWriter`
- struct `SyscallWriter`
- struct `TemporaryWriter`

Static Public Member Functions

- static void **upmax** (int &i1, int i2)
- static void **indent** (`IStringWriter` &write, int &x, int want_x)
- static bool **isWideChar** (unsigned ch)
Returns true if ch is the unicode code point of a wide character.
- static void **printUsage** (`IStringWriter` &write, const `Descriptor` usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)

5.22.1 Member Function Documentation

5.22.1.1 `static bool xmem::PrintUsageImplementation::isWideChar (unsigned ch)` [inline],[static]

Returns true if ch is the unicode code point of a wide character.

Note

The following character ranges are treated as wide

```
1100..115F
2329..232A (just 2 characters!)
2E80..A4C6 except for 303F
A960..A97C
AC00..D7FB
F900..FAFF
FE10..FE6B
FF01..FF60
FFE0..FFE6
1B000.....
```

The documentation for this struct was generated from the following file:

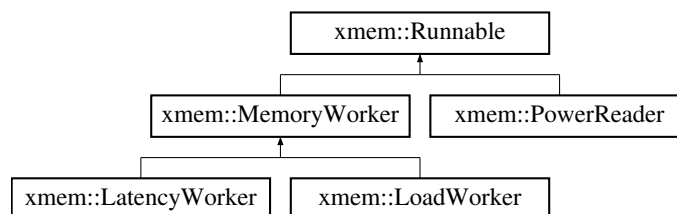
- [src/include/optionparser.h](#)

5.23 xmem::Runnable Class Reference

A base class for any object that implements a thread-safe [run\(\)](#) function for use by [Thread](#) objects.

```
#include <Runnable.h>
```

Inheritance diagram for xmem::Runnable:



Public Member Functions

- [Runnable\(\)](#)
Constructor.
- [~Runnable\(\)](#)
Destructor.
- virtual void [run\(\)](#)=0
Does some "work". Pure virtual method that any derived class must implement in a thread-safe manner.

Protected Member Functions

- bool [_acquireLock](#)(int32_t timeout)
Acquires the object lock to access all object state in thread-safe manner.
- bool [_releaseLock](#)()
Releases the object lock to access all object state in thread-safe manner.

5.23.1 Detailed Description

A base class for any object that implements a thread-safe [run\(\)](#) function for use by [Thread](#) objects.

5.23.2 Member Function Documentation

5.23.2.1 `bool Runnable::_acquireLock (int32_t timeout)` `[protected]`

Acquires the object lock to access all object state in thread-safe manner.

Parameters

<i>timeout</i>	timeout in milliseconds to acquire the lock. If 0, does not wait at all. If negative, waits indefinitely.
----------------	---

Returns

true on success. If not successful, the lock was not acquired, possibly due to a timeout, or the lock might already be held.

5.23.2.2 bool Runnable::_releaseLock() [protected]

Releases the object lock to access all object state in thread-safe manner.

Returns

true on success. If not successful, the lock is either still held or the call was illegal (e.g., releasing a lock that was never acquired).

The documentation for this class was generated from the following files:

- src/include/Runnable.h
- src/Runnable.cpp

5.24 xmem::Stats Struct Reference

Determines the minimum lengths of the buffer and options arrays used for [Parser](#).

```
#include <optionparser.h>
```

Classes

- class [CountOptionsAction](#)

Public Member Functions

- [Stats](#) ()
Creates a [Stats](#) object with counts set to 1 (for the sentinel element).
- [Stats](#) (bool gnu, const [Descriptor](#) usage[], int argc, const char **argv, int min_abbr_len=0, bool single_minus_longopt=false)
*Creates a new [Stats](#) object and immediately updates it for the given *usage* and argument vector. You may pass 0 for *argc* and/or *argv*, if you just want to update [options_max](#).*
- [Stats](#) (bool gnu, const [Descriptor](#) usage[], int argc, char **argv, int min_abbr_len=0, bool single_minus_longopt=false)
[Stats](#)(...) with non-const argv.
- [Stats](#) (const [Descriptor](#) usage[], int argc, const char **argv, int min_abbr_len=0, bool single_minus_longopt=false)
POSIX [Stats](#)(...) (gnu==false).
- [Stats](#) (const [Descriptor](#) usage[], int argc, char **argv, int min_abbr_len=0, bool single_minus_longopt=false)
POSIX [Stats](#)(...) (gnu==false) with non-const argv.
- void [add](#) (bool gnu, const [Descriptor](#) usage[], int argc, const char **argv, int min_abbr_len=0, bool single_minus_longopt=false)

Updates this [Stats](#) object for the given `usage` and argument vector. You may pass 0 for `argc` and/or `argv`, if you just want to update `options_max`.

- void [add](#) (bool `gnu`, const [Descriptor](#) `usage`[], int `argc`, char **`argv`, int `min_abbr_len`=0, bool `single_minus_`-`longopt`=false)

[add\(\)](#) with non-const `argv`.

- void [add](#) (const [Descriptor](#) `usage`[], int `argc`, const char **`argv`, int `min_abbr_len`=0, bool `single_minus_`-`longopt`=false)

POSIX [add\(\)](#) (`gnu`==false).

- void [add](#) (const [Descriptor](#) `usage`[], int `argc`, char **`argv`, int `min_abbr_len`=0, bool `single_minus_`-`longopt`=false)

POSIX [add\(\)](#) (`gnu`==false) with non-const `argv`.

Public Attributes

- unsigned [buffer_max](#)

Number of elements needed for a `buffer`[] array to be used for [parsing](#) the same argument vectors that were fed into this [Stats](#) object.

- unsigned [options_max](#)

Number of elements needed for an `options`[] array to be used for [parsing](#) the same argument vectors that were fed into this [Stats](#) object.

5.24.1 Detailed Description

Determines the minimum lengths of the buffer and options arrays used for [Parser](#).

Because [Parser](#) doesn't use dynamic memory its output arrays have to be pre-allocated. If you don't want to use fixed size arrays (which may turn out too small, causing command line arguments to be dropped), you can use [Stats](#) to determine the correct sizes. [Stats](#) work cumulative. You can first pass in your default options and then the real options and afterwards the counts will reflect the union.

5.24.2 Constructor & Destructor Documentation

- 5.24.2.1 `xmem::Stats::Stats (bool gnu, const Descriptor usage[], int argc, const char ** argv, int min_abbr_len = 0, bool single_minus_longopt = false) [inline]`

Creates a new [Stats](#) object and immediately updates it for the given `usage` and argument vector. You may pass 0 for `argc` and/or `argv`, if you just want to update `options_max`.

Note

The calls to [Stats](#) methods must match the later calls to [Parser](#) methods. See [Parser::parse\(\)](#) for the meaning of the arguments.

5.24.3 Member Function Documentation

- 5.24.3.1 `void xmem::Stats::add (bool gnu, const Descriptor usage[], int argc, const char ** argv, int min_abbr_len = 0, bool single_minus_longopt = false) [inline]`

Updates this [Stats](#) object for the given `usage` and argument vector. You may pass 0 for `argc` and/or `argv`, if you just want to update `options_max`.

Note

The calls to [Stats](#) methods must match the later calls to [Parser](#) methods. See [Parser::parse\(\)](#) for the meaning of the arguments.

5.24.4 Member Data Documentation

5.24.4.1 unsigned xmem::Stats::buffer_max

Number of elements needed for a `buffer[]` array to be used for [parsing](#) the same argument vectors that were fed into this [Stats](#) object.

Note

This number is always 1 greater than the actual number needed, to give you a sentinel element.

5.24.4.2 unsigned xmem::Stats::options_max

Number of elements needed for an `options[]` array to be used for [parsing](#) the same argument vectors that were fed into this [Stats](#) object.

Note

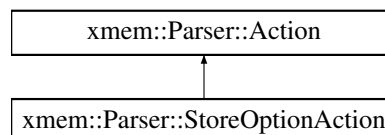
- This number is always 1 greater than the actual number needed, to give you a sentinel element.
- This number depends only on the `usage`, not the argument vectors, because the `options` array needs exactly one slot for each possible [Descriptor::index](#).

The documentation for this struct was generated from the following file:

- `src/include/optionparser.h`

5.25 xmem::Parser::StoreOptionAction Class Reference

Inheritance diagram for `xmem::Parser::StoreOptionAction`:



Public Member Functions

- [StoreOptionAction](#) ([Parser](#) &parser_, [Option](#) options[], [Option](#) buffer[], int bufmax_)
Number of slots in `buffer`. -1 means "large enough".
- bool [perform](#) ([Option](#) &option)
Called by `Parser::workhorse()` for each [Option](#) that has been successfully parsed (including unknown options if they have a [Descriptor](#) whose [Descriptor::check_arg](#) does not return `ARG_ILLEGAL`).
- bool [finished](#) (int numargs, const char **args)
Called by `Parser::workhorse()` after finishing the parse.

5.25.1 Constructor & Destructor Documentation

5.25.1.1 xmem::Parser::StoreOptionAction::StoreOptionAction ([Parser](#) &parser_, [Option](#) options[], [Option](#) buffer[], int bufmax_) [inline]

Number of slots in `buffer`. -1 means "large enough".

Creates a new `StoreOption` action.

Parameters

<i>parser_</i>	the parser whose op_count should be updated.
<i>options_</i>	each Option o is chained into the linked list <code>options_[o.desc->index]</code>
<i>buffer_</i>	each Option is appended to this array as long as there's a free slot.
<i>bufmax_</i>	number of slots in <code>buffer_</code> . -1 means "large enough".

5.25.2 Member Function Documentation

5.25.2.1 `bool xmem::Parser::StoreOptionAction::finished (int numargs, const char ** args) [inline],[virtual]`

Called by `Parser::workhorse()` after finishing the parse.

Parameters

<i>numargs</i>	the number of non-option arguments remaining
<i>args</i>	pointer to the first remaining non-option argument (if <code>numargs > 0</code>).

Returns

`false` iff a fatal error has occurred.

Reimplemented from [xmem::Parser::Action](#).

5.25.2.2 `bool xmem::Parser::StoreOptionAction::perform (Option &) [inline],[virtual]`

Called by `Parser::workhorse()` for each [Option](#) that has been successfully parsed (including unknown options if they have a [Descriptor](#) whose [Descriptor::check_arg](#) does not return `ARG_ILLEGAL`).

Returns `false` iff a fatal error has occurred and the parse should be aborted.

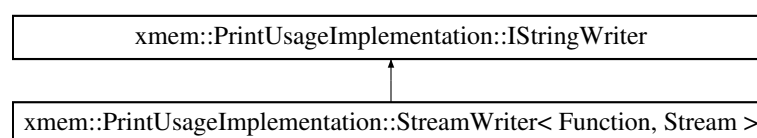
Reimplemented from [xmem::Parser::Action](#).

The documentation for this class was generated from the following file:

- [src/include/optionparser.h](#)

5.26 `xmem::PrintUsageImplementation::StreamWriter< Function, Stream >` Struct Template Reference

Inheritance diagram for `xmem::PrintUsageImplementation::StreamWriter< Function, Stream >`:



Public Member Functions

- virtual void [operator\(\)](#) (const char *str, int size)
Writes the given number of chars beginning at the given pointer somewhere.
- **StreamWriter** (Function *w, Stream *s)

Public Attributes

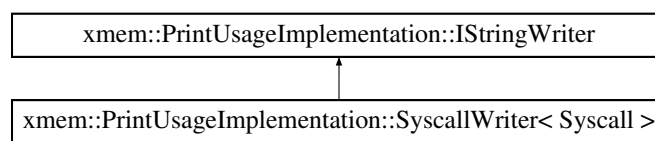
- Function * **fwrite**
- Stream * **stream**

The documentation for this struct was generated from the following file:

- [src/include/optionparser.h](#)

5.27 xmem::PrintUsageImplementation::SyscallWriter< Syscall > Struct Template Reference

Inheritance diagram for xmem::PrintUsageImplementation::SyscallWriter< Syscall >:



Public Member Functions

- virtual void [operator\(\)](#) (const char *str, int size)
Writes the given number of chars beginning at the given pointer somewhere.
- **SyscallWriter** (Syscall *w, int f)

Public Attributes

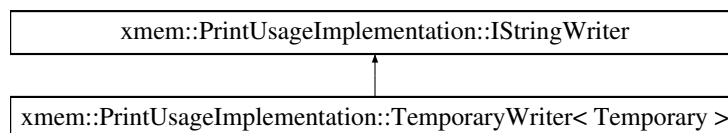
- Syscall * **write**
- int **fd**

The documentation for this struct was generated from the following file:

- [src/include/optionparser.h](#)

5.28 xmem::PrintUsageImplementation::TemporaryWriter< Temporary > Struct Template Reference

Inheritance diagram for xmem::PrintUsageImplementation::TemporaryWriter< Temporary >:



Public Member Functions

- virtual void [operator\(\)](#) (const char *str, int size)
Writes the given number of chars beginning at the given pointer somewhere.
- **TemporaryWriter** (const Temporary &u)

Public Attributes

- const Temporary & **userstream**

The documentation for this struct was generated from the following file:

- [src/include/optionparser.h](#)

5.29 xmem::Thread Class Reference

a nice wrapped thread interface independent of particular OS API

```
#include <Thread.h>
```

Public Member Functions

- [Thread](#) ([Runnable](#) *target)
- [~Thread](#) ()
- bool [create_and_start](#) ()
- bool [join](#) ()
- bool [cancel](#) ()
- int32_t [getExitCode](#) ()
- bool [started](#) ()
- bool [completed](#) ()
- bool [validTarget](#) ()
- bool [created](#) ()
- bool [isThreadSuspended](#) ()
- bool [isThreadRunning](#) ()
- [Runnable](#) * [getTarget](#) ()

5.29.1 Detailed Description

a nice wrapped thread interface independent of particular OS API

5.29.2 Constructor & Destructor Documentation

5.29.2.1 Thread::Thread ([Runnable](#) * target)

Constructor. Does not actually create the real thread or run it.

Parameters

<i>target</i>	The target object to do some work with in a new thread.
---------------	---

5.29.2.2 Thread::~~Thread ()

Destructor. Immediately cancels the thread if it exists. This can be unsafe!

5.29.3 Member Function Documentation

5.29.3.1 `bool Thread::cancel ()`

Cancels the worker thread immediately. This should only be done in emergencies, as it is effectively killed and undefined behavior might occur.

Returns

true if the worker thread was successfully killed.

5.29.3.2 `bool Thread::completed ()`

Returns

true if the thread completed, regardless of the manner in which it terminated. Returns false if it has not been started.

5.29.3.3 `bool Thread::create_and_start ()`

Creates and starts the thread immediately if the target [Runnable](#) is valid. This invokes the `run()` method in the [Runnable](#) target that was passed in the constructor.

Returns

true if the thread was successfully created and started.

5.29.3.4 `bool Thread::created ()`

Returns

true if the thread has been created successfully.

5.29.3.5 `int32_t Thread::getExitCode ()`

Returns

the exit code of the worker thread if it completed. If it did not complete or has not started, returns 0.

5.29.3.6 `Runnable * Thread::getTarget ()`

Returns

a pointer to the target [Runnable](#) object

5.29.3.7 `bool Thread::isThreadRunning ()`

Returns

true if the thread is running. Returns false if the thread has not been created.

5.29.3.8 bool Thread::isThreadSuspended ()

Returns

true if the thread is suspended. Returns false if the thread has not been created.

5.29.3.9 bool Thread::join ()

Blocks the calling thread until the worker thread managed by this object terminates. For simplicity, this does not support a timeout due to pthreads incompatibility with the Windows threading API. If the worker thread has already terminated, returns immediately. If the worker has not yet started, returns immediately.

Returns

true if the worker thread terminated successfully, false otherwise.

5.29.3.10 bool Thread::started ()

Returns

true if the thread has been started, regardless if has completed or not.

5.29.3.11 bool Thread::validTarget ()

Returns

true if the [Runnable](#) target is valid.

The documentation for this class was generated from the following files:

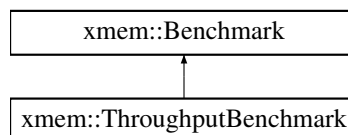
- [src/include/Thread.h](#)
- [src/Thread.cpp](#)

5.30 xmem::ThroughputBenchmark Class Reference

A type of benchmark that measures memory throughput.

```
#include <ThroughputBenchmark.h>
```

Inheritance diagram for xmem::ThroughputBenchmark:



Public Member Functions

- [ThroughputBenchmark](#) (void *mem_array, size_t len, uint32_t iterations, uint32_t num_worker_threads, uint32_t mem_node, uint32_t cpu_node, pattern_mode_t pattern_mode, rw_mode_t rw_mode, chunk_size_t chunk_size, int32_t stride_size, std::vector< [PowerReader](#) * > dram_power_readers, std::string name)
Constructor. Parameters are passed directly to the [Benchmark](#) constructor. See [Benchmark](#) class documentation for parameter semantics.
- virtual [~ThroughputBenchmark](#) ()
Destructor.

Protected Member Functions

- virtual bool [_run_core](#) ()
The core benchmark function.

Additional Inherited Members

5.30.1 Detailed Description

A type of benchmark that measures memory throughput.

5.30.2 Member Function Documentation

5.30.2.1 bool ThroughputBenchmark::_run_core () [protected], [virtual]

The core benchmark function.

Returns

True on success.

Implements [xmem::Benchmark](#).

The documentation for this class was generated from the following files:

- [src/include/ThroughputBenchmark.h](#)
- [src/ThroughputBenchmark.cpp](#)

5.31 xmem::Timer Class Reference

This class abstracts some characteristics of simple high resolution stopwatch timer. However, due to the inability or complexity of abstracting shared hardware timers, this class does not actually provide start and stop functions.

```
#include <Timer.h>
```

Public Member Functions

- [Timer](#) ()
Constructor. This may take a noticeable amount of time.
- tick_t [get_ticks_per_ms](#) ()
Gets ticks per ms for this timer.
- double [get_ns_per_tick](#) ()
Gets nanoseconds per tick for this timer.

Protected Attributes

- tick_t [_ticks_per_ms](#)
- double [_ns_per_tick](#)

5.31.1 Detailed Description

This class abstracts some characteristics of simple high resolution stopwatch timer. However, due to the inability or complexity of abstracting shared hardware timers, this class does not actually provide start and stop functions.

5.31.2 Member Function Documentation

5.31.2.1 `double Timer::get_ns_per_tick ()`

Gets nanoseconds per tick for this timer.

Returns

the number of nanoseconds per tick

5.31.2.2 `tick_t Timer::get_ticks_per_ms ()`

Gets ticks per ms for this timer.

Returns

The reported number of ticks per ms.

5.31.3 Member Data Documentation

5.31.3.1 `double xmem::Timer::_ns_per_tick` [protected]

Nanoseconds per tick for this timer.

5.31.3.2 `tick_t xmem::Timer::_ticks_per_ms` [protected]

Ticks per ms for this timer.

The documentation for this class was generated from the following files:

- [src/include/Timer.h](#)
- [src/Timer.cpp](#)

Chapter 6

File Documentation

6.1 src/Benchmark.cpp File Reference

Implementation file for the Benchmark class.

```
#include <Benchmark.h>
#include <common.h>
#include <benchmark_kernels.h>
#include <PowerReader.h>
#include <cstdint>
#include <iostream>
#include <vector>
#include <time.h>
```

6.1.1 Detailed Description

Implementation file for the Benchmark class.

6.2 src/benchmark_kernels.cpp File Reference

Implementation file for benchmark kernel functions for doing the actual work we care about. :)

```
#include <benchmark_kernels.h>
#include <common.h>
#include <iostream>
#include <random>
#include <algorithm>
#include <time.h>
```

6.2.1 Detailed Description

Implementation file for benchmark kernel functions for doing the actual work we care about. :) Optimization tricks include:

- UNROLL macros to manual loop unrolling. This reduces the relative branch overhead of the loop. We don't want to benchmark loops, we want to benchmark memory! But unrolling too much can hurt code size and instruction locality, potentially decreasing I-cache utilization and causing extra overheads. This is why we allow multiple unroll lengths at compile-time.

- volatile keyword to prevent compiler from optimizing the code and removing instructions that we need. The compiler is too smart for its own good!

6.3 src/BenchmarkManager.cpp File Reference

Implementation file for the BenchmarkManager class.

```
#include <BenchmarkManager.h>
#include <common.h>
#include <Configurator.h>
#include <cstdint>
#include <stdlib.h>
#include <iostream>
#include <sstream>
#include <assert.h>
```

6.3.1 Detailed Description

Implementation file for the BenchmarkManager class.

6.4 src/common.cpp File Reference

Implementation file for common preprocessor definitions, macros, functions, and global constants.

```
#include <common.h>
#include <Timer.h>
#include <iostream>
```

Variables

- bool **xmem::g_verbose** = false
- size_t **xmem::g_page_size**
- size_t **xmem::g_large_page_size**
- uint32_t **xmem::g_num_nodes**
- uint32_t **xmem::g_num_logical_cpus**
- uint32_t **xmem::g_num_physical_cpus**
- uint32_t **xmem::g_num_physical_packages**
- uint32_t **xmem::g_total_l1_caches**
- uint32_t **xmem::g_total_l2_caches**
- uint32_t **xmem::g_total_l3_caches**
- uint32_t **xmem::g_total_l4_caches**
- uint32_t **xmem::g_starting_test_index**
- uint32_t **xmem::g_test_index**
- tick_t **xmem::g_ticks_per_ms**
- double **xmem::g_ns_per_tick**

6.4.1 Detailed Description

Implementation file for common preprocessor definitions, macros, functions, and global constants.

6.5 src/Configurator.cpp File Reference

Implementation file for the Configurator class and some helper data structures.

```
#include <Configurator.h>
#include <common.h>
#include <optionparser.h>
#include <MyArg.h>
#include <cstdint>
#include <iostream>
#include <string>
```

6.5.1 Detailed Description

Implementation file for the Configurator class and some helper data structures.

6.6 src/ext/DelayInjectedLoadedLatencyBenchmark/delay_injected_benchmark_kernels.cpp File Reference

Implementation file for benchmark kernel functions for the delay-injected loaded latency benchmark.

```
#include <delay_injected_benchmark_kernels.h>
#include <common.h>
#include <iostream>
```

6.6.1 Detailed Description

Implementation file for benchmark kernel functions for the delay-injected loaded latency benchmark. Optimization tricks include:

- UNROLL macros to manual loop unrolling. This reduces the relative branch overhead of the loop. We don't want to benchmark loops, we want to benchmark memory! But unrolling too much can hurt code size and instruction locality, potentially decreasing l-cache utilization and causing extra overheads. This is why we allow multiple unroll lengths at compile-time.
- volatile keyword to prevent compiler from optimizing the code and removing instructions that we need. The compiler is too smart for its own good!

6.7 src/ext/DelayInjectedLoadedLatencyBenchmark/DelayInjectedLoadedLatencyBenchmark.cpp File Reference

Implementation file for the DelayInjectedLatencyBenchmark class.

```
#include <common.h>
```

6.7.1 Detailed Description

Implementation file for the DelayInjectedLatencyBenchmark class.

6.8 src/include/Benchmark.h File Reference

Header file for the Benchmark class.

```
#include <common.h>
#include <PowerReader.h>
#include <Thread.h>
#include <Runnable.h>
#include <stdint>
#include <string>
#include <vector>
```

Classes

- class [xmem::Benchmark](#)
Flexible abstract class for any memory benchmark.

6.8.1 Detailed Description

Header file for the Benchmark class.

6.9 src/include/benchmark_kernels.h File Reference

Header file for benchmark kernel functions for doing the actual work we care about. :)

```
#include <common.h>
#include <stdint>
#include <stddef>
```

Typedefs

- typedef int32_t(* **xmem::SequentialFunction**)(void *, void *)
- typedef int32_t(* **xmem::RandomFunction**)(uintptr_t *, uintptr_t **, size_t)

Functions

- bool **xmem::determineSequentialKernel** (rw_mode_t rw_mode, chunk_size_t chunk_size, int32_t stride_size, SequentialFunction *kernel_function, SequentialFunction *dummy_kernel_function)
Determines which sequential memory access kernel to use based on the read/write mode, chunk size, and stride size.
- bool **xmem::determineRandomKernel** (rw_mode_t rw_mode, chunk_size_t chunk_size, RandomFunction *kernel_function, RandomFunction *dummy_kernel_function)
Determines which random memory access kernel to use based on the read/write mode, chunk size, and stride size.
- bool **xmem::buildRandomPointerPermutation** (void *start_address, void *end_address, chunk_size_t chunk_size)
Builds a random chain of pointers within the specified memory region.
- int32_t **xmem::dummy_chasePointers** (uintptr_t *, uintptr_t **, size_t len)
Mimics the chasePointers() method but doesn't do the memory accesses.
- int32_t **xmem::chasePointers** (uintptr_t *first_address, uintptr_t **last_touched_address, size_t len)
Walks over the allocated memory in random order by chasing pointers.

- `int32_t xmem::dummy_empty (void *, void *)`
Does nothing. Used for measuring the time it takes just to call a benchmark routine via function pointer.
- `int32_t xmem::dummy_forwSequentialLoop_Word32 (void *start_address, void *end_address)`
Used for measuring the time spent doing everything in forward sequential Word 32 loops except for the memory access itself.
- `int32_t xmem::dummy_revSequentialLoop_Word32 (void *start_address, void *end_address)`
Used for measuring the time spent doing everything in reverse sequential Word 32 loops except for the memory access itself.
- `int32_t xmem::dummy_forwStride2Loop_Word32 (void *start_address, void *end_address)`
Used for measuring the time spent doing everything in forward 2-strided Word 32 loops except for the memory access itself.
- `int32_t xmem::dummy_revStride2Loop_Word32 (void *start_address, void *end_address)`
Used for measuring the time spent doing everything in reverse 2-strided Word 32 loops except for the memory access itself.
- `int32_t xmem::dummy_forwStride4Loop_Word32 (void *start_address, void *end_address)`
Used for measuring the time spent doing everything in forward 4-strided Word 32 loops except for the memory access itself.
- `int32_t xmem::dummy_revStride4Loop_Word32 (void *start_address, void *end_address)`
Used for measuring the time spent doing everything in reverse 4-strided Word 32 loops except for the memory access itself.
- `int32_t xmem::dummy_forwStride8Loop_Word32 (void *start_address, void *end_address)`
Used for measuring the time spent doing everything in forward 8-strided Word 32 loops except for the memory access itself.
- `int32_t xmem::dummy_revStride8Loop_Word32 (void *start_address, void *end_address)`
Used for measuring the time spent doing everything in reverse 8-strided Word 32 loops except for the memory access itself.
- `int32_t xmem::dummy_forwStride16Loop_Word32 (void *start_address, void *end_address)`
Used for measuring the time spent doing everything in forward 16-strided Word 32 loops except for the memory access itself.
- `int32_t xmem::dummy_revStride16Loop_Word32 (void *start_address, void *end_address)`
Used for measuring the time spent doing everything in reverse 16-strided Word 32 loops except for the memory access itself.
- `int32_t xmem::dummy_randomLoop_Word32 (uintptr_t *, uintptr_t **, size_t len)`
Mimics the randomRead_Word32 and randomWrite_Word32 functions except for the memory accesses.
- `int32_t xmem::forwSequentialRead_Word32 (void *start_address, void *end_address)`
Walks over the allocated memory forward sequentially, reading in 32-bit chunks.
- `int32_t xmem::revSequentialRead_Word32 (void *start_address, void *end_address)`
Walks over the allocated memory reverse sequentially, reading in 32-bit chunks.
- `int32_t xmem::forwSequentialWrite_Word32 (void *start_address, void *end_address)`
Walks over the allocated memory forward sequentially, writing in 32-bit chunks.
- `int32_t xmem::revSequentialWrite_Word32 (void *start_address, void *end_address)`
Walks over the allocated memory reverse sequentially, writing in 32-bit chunks.
- `int32_t xmem::forwStride2Read_Word32 (void *start_address, void *end_address)`
Walks over the allocated memory in forward strides of size 2, reading in 32-bit chunks.
- `int32_t xmem::revStride2Read_Word32 (void *start_address, void *end_address)`
Walks over the allocated memory in reverse strides of size 2, reading in 32-bit chunks.
- `int32_t xmem::forwStride2Write_Word32 (void *start_address, void *end_address)`
Walks over the allocated memory in forward strides of size 2, writing in 32-bit chunks.
- `int32_t xmem::revStride2Write_Word32 (void *start_address, void *end_address)`
Walks over the allocated memory in reverse strides of size 2, writing in 32-bit chunks.
- `int32_t xmem::forwStride4Read_Word32 (void *start_address, void *end_address)`
Walks over the allocated memory in forward strides of size 4, reading in 32-bit chunks.
- `int32_t xmem::revStride4Read_Word32 (void *start_address, void *end_address)`

- Walks over the allocated memory in reverse strides of size 4, reading in 32-bit chunks.*

 - `int32_t xmem::forwStride4Write_Word32 (void *start_address, void *end_address)`

Walks over the allocated memory in forward strides of size 4, writing in 32-bit chunks.
- `int32_t xmem::revStride4Write_Word32 (void *start_address, void *end_address)`

Walks over the allocated memory in reverse strides of size 4, writing in 32-bit chunks.
- `int32_t xmem::forwStride8Read_Word32 (void *start_address, void *end_address)`

Walks over the allocated memory in forward strides of size 8, reading in 32-bit chunks.
- `int32_t xmem::revStride8Read_Word32 (void *start_address, void *end_address)`

Walks over the allocated memory in reverse strides of size 8, reading in 32-bit chunks.
- `int32_t xmem::forwStride8Write_Word32 (void *start_address, void *end_address)`

Walks over the allocated memory in forward strides of size 8, writing in 32-bit chunks.
- `int32_t xmem::revStride8Write_Word32 (void *start_address, void *end_address)`

Walks over the allocated memory in reverse strides of size 8, writing in 32-bit chunks.
- `int32_t xmem::forwStride16Read_Word32 (void *start_address, void *end_address)`

Walks over the allocated memory in forward strides of size 16, reading in 32-bit chunks.
- `int32_t xmem::revStride16Read_Word32 (void *start_address, void *end_address)`

Walks over the allocated memory in reverse strides of size 16, reading in 32-bit chunks.
- `int32_t xmem::forwStride16Write_Word32 (void *start_address, void *end_address)`

Walks over the allocated memory in forward strides of size 16, writing in 32-bit chunks.
- `int32_t xmem::revStride16Write_Word32 (void *start_address, void *end_address)`

Walks over the allocated memory in reverse strides of size 16, writing in 32-bit chunks.
- `int32_t xmem::randomRead_Word32 (uintptr_t *first_address, uintptr_t **last_touched_address, size_t len)`

Walks over the allocated memory in random order by chasing 64-bit pointers.
- `int32_t xmem::randomWrite_Word32 (uintptr_t *first_address, uintptr_t **last_touched_address, size_t len)`

Walks over the allocated memory in random order by chasing 32-bit pointers. A pointer is read and written back with the same value before chasing to the next pointer. Thus, each memory address is a read followed by immediate write operation.

6.9.1 Detailed Description

Header file for benchmark kernel functions for doing the actual work we care about. :)

6.10 src/include/BenchmarkManager.h File Reference

Header file for the BenchmarkManager class.

```
#include <common.h>
#include <Timer.h>
#include <PowerReader.h>
#include <Benchmark.h>
#include <ThroughputBenchmark.h>
#include <LatencyBenchmark.h>
#include <Configurator.h>
#include <cstdint>
#include <vector>
#include <fstream>
```

Classes

- class [xmem::BenchmarkManager](#)
Manages running all benchmarks at a high level.

6.10.1 Detailed Description

Header file for the BenchmarkManager class.

6.11 src/include/common.h File Reference

Header file for common preprocessor definitions, macros, functions, and global constants.

```
#include <stdint>
#include <stddef>
```

Macros

- #define **VERSION** "2.1.13"
- #define **KB** 1024
- #define **MB** 1048576
- #define **MB_4** 4194304
- #define **MB_16** 16777216
- #define **MB_64** 67108864
- #define **MB_256** 268435456
- #define **MB_512** 536870912
- #define **GB** 1073741824
- #define [DEFAULT_PAGE_SIZE](#) 4*KB
- #define [DEFAULT_LARGE_PAGE_SIZE](#) 2*MB
- #define [DEFAULT_WORKING_SET_SIZE_PER_THREAD](#) [DEFAULT_PAGE_SIZE](#)
- #define [DEFAULT_NUM_WORKER_THREADS](#) 1
- #define [DEFAULT_NUM_NODES](#) 1
- #define [DEFAULT_NUM_PHYSICAL_PACKAGES](#) 1
- #define [DEFAULT_NUM_PHYSICAL_CPUS](#) 1
- #define [DEFAULT_NUM_LOGICAL_CPUS](#) 1
- #define [DEFAULT_NUM_L1_CACHES](#) 0
- #define [DEFAULT_NUM_L2_CACHES](#) 0
- #define [DEFAULT_NUM_L3_CACHES](#) 0
- #define [DEFAULT_NUM_L4_CACHES](#) 0
- #define [MIN_ELAPSED_TICKS](#) 10000
- #define **UNROLL2**(x) x x
- #define **UNROLL4**(x) UNROLL2(x) UNROLL2(x)
- #define **UNROLL8**(x) UNROLL4(x) UNROLL4(x)
- #define **UNROLL16**(x) UNROLL8(x) UNROLL8(x)
- #define **UNROLL32**(x) UNROLL16(x) UNROLL16(x)
- #define **UNROLL64**(x) UNROLL32(x) UNROLL32(x)
- #define **UNROLL128**(x) UNROLL64(x) UNROLL64(x)
- #define **UNROLL256**(x) UNROLL128(x) UNROLL128(x)
- #define **UNROLL512**(x) UNROLL256(x) UNROLL256(x)
- #define **UNROLL1024**(x) UNROLL512(x) UNROLL512(x)
- #define **UNROLL2048**(x) UNROLL1024(x) UNROLL1024(x)
- #define **UNROLL4096**(x) UNROLL2048(x) UNROLL2048(x)

- `#define UNROLL8192(x) UNROLL4096(x) UNROLL4096(x)`
- `#define UNROLL16384(x) UNROLL8192(x) UNROLL8192(x)`
- `#define UNROLL32768(x) UNROLL16384(x) UNROLL16384(x)`
- `#define UNROLL65536(x) UNROLL32768(x) UNROLL32768(x)`
- `#define LATENCY_BENCHMARK_UNROLL_LENGTH 512`
- `#define USE_OS_TIMER`
- `#define USE_TIME_BASED_BENCHMARKS`
- `#define BENCHMARK_DURATION_MS 250`
- `#define THROUGHPUT_BENCHMARK_BYTES_PER_PASS 4096`
- `#define POWER_SAMPLING_PERIOD_MS 1000`
- `#define EXT_DELAY_INJECTED_LOADED_LATENCY_BENCHMARK`

Typedefs

- `typedef uint32_t xmem::tick_t`
- `typedef uint32_t xmem::Word32_t`

Enumerations

- `enum pattern_mode_t { SEQUENTIAL, RANDOM, NUM_PATTERN_MODES }`
Memory access patterns are broadly categorized by sequential or random-access.
- `enum rw_mode_t { READ, WRITE, NUM_RW_MODES }`
Memory access patterns are broadly categorized by reads and writes.
- `enum chunk_size_t { CHUNK_32b, NUM_CHUNK_SIZES }`
Legal memory read/write chunk sizes in bits.
- `enum ext_t { EXT_NUM_DELAY_INJECTED_LOADED_LATENCY_BENCHMARK, NUM_EXTENSIONS }`

Functions

- `void xmem::print_welcome_message ()`
Prints a basic welcome message to the console with useful information.
- `void xmem::print_types_report ()`
Prints the various C/C++ types to the console for this machine.
- `void xmem::print_compile_time_options ()`
Prints compile-time option information to the console.
- `void xmem::setup_timer ()`
Initializes the timer and outputs results to the console for sanity checking.
- `void xmem::report_timer ()`
Reports timer info to the console.
- `void xmem::test_thread_affinities ()`
Checks to see if the calling thread can be locked to all logical CPUs in the system, and reports to the console the progress.
- `bool xmem::lock_thread_to_numa_node (uint32_t numa_node)`
Sets the affinity of the calling thread to the lowest numbered logical CPU in the given NUMA node. TODO: Improve this functionality, it is quite limiting.
- `bool xmem::unlock_thread_to_numa_node ()`
Clears the affinity of the calling thread to any given NUMA node.
- `bool xmem::lock_thread_to_cpu (uint32_t cpu_id)`
Sets the affinity of the calling thread to a given logical CPU.
- `bool xmem::unlock_thread_to_cpu ()`
Clears the affinity of the calling thread to any given logical CPU.

- `int32_t xmem::cpu_id_in_numa_node (uint32_t numa_node, uint32_t cpu_in_node)`
Gets the CPU ID for a logical CPU of interest in a particular NUMA node. For example, if `numa_node` is 1 and `cpu_in_node` is 2, and there are 4 logical CPUs per node, then this will give the answer 6 (6th CPU), assuming CPU IDs start at 0.
- `size_t xmem::compute_number_of_passes (size_t working_set_size_KB)`
Computes the number of passes to use for a given working set size in KB, when size-based benchmarking mode is enabled at compile-time. You may want to change this implementation to suit your needs. See the compile-time options in [common.h](#).
- `bool xmem::config_page_size ()`
Queries the page sizes from the system and sets relevant global variables.
- `void xmem::init_globals ()`
Initializes useful global variables.
- `int32_t xmem::query_sys_info ()`
Sets up global variables based on system information at runtime.
- `void xmem::report_sys_info ()`
Reports the system configuration to the console as indicated by global variables.
- `tick_t xmem::start_timer ()`
Query the timer for the start of a timed section of code.
- `tick_t xmem::stop_timer ()`
Query the timer for the end of a timed section of code.

6.11.1 Detailed Description

Header file for common preprocessor definitions, macros, functions, and global constants.

6.11.2 Macro Definition Documentation

6.11.2.1 `#define BENCHMARK_DURATION_MS 250`

RECOMMENDED VALUE: At least 1000. Number of milliseconds to run in each benchmark.

6.11.2.2 `#define DEFAULT_LARGE_PAGE_SIZE 2*MB`

Default platform large page size in bytes. This generally should not be relied on, but is a failsafe.

6.11.2.3 `#define DEFAULT_NUM_L1_CACHES 0`

Default number of L1 caches.

6.11.2.4 `#define DEFAULT_NUM_L2_CACHES 0`

Default number of L2 caches.

6.11.2.5 `#define DEFAULT_NUM_L3_CACHES 0`

Default number of L3 caches.

6.11.2.6 `#define DEFAULT_NUM_L4_CACHES 0`

Default number of L4 caches.

6.11.2.7 #define DEFAULT_NUM_LOGICAL_CPUS 1

Default number of logical CPU cores.

6.11.2.8 #define DEFAULT_NUM_NODES 1

Default number of NUMA nodes.

6.11.2.9 #define DEFAULT_NUM_PHYSICAL_CPUS 1

Default number of physical CPU cores.

6.11.2.10 #define DEFAULT_NUM_PHYSICAL_PACKAGES 1

Default number of physical packages.

6.11.2.11 #define DEFAULT_NUM_WORKER_THREADS 1

Default number of worker threads to use.

6.11.2.12 #define DEFAULT_PAGE_SIZE 4*KB

Default platform page size in bytes. This generally should not be relied on, but is a failsafe.

6.11.2.13 #define DEFAULT_WORKING_SET_SIZE_PER_THREAD DEFAULT_PAGE_SIZE

Default working set size in bytes.

6.11.2.14 #define EXT_DELAY_INJECTED_LOADED_LATENCY_BENCHMARK

RECOMMENDED ENABLED. This allows for a custom extension to X-Mem that performs latency benchmarking with forward sequential 64-bit and 256-bit read-based load threads with variable delays injected in between memory accesses.

6.11.2.15 #define LATENCY_BENCHMARK_UNROLL_LENGTH 512

Number of unrolls in the latency benchmark pointer chasing core function.

6.11.2.16 #define MIN_ELAPSED_TICKS 10000

If any routine measured fewer than this number of ticks its results should be viewed with suspicion. This is because the latency of the timer itself will matter.

6.11.2.17 #define POWER_SAMPLING_PERIOD_MS 1000

RECOMMENDED VALUE: 1000. Sampling period in milliseconds for all power measurement mechanisms.

6.11.2.18 `#define THROUGHPUT_BENCHMARK_BYTES_PER_PASS 4096`

RECOMMENDED VALUE: 4096. Number of bytes read or written per pass of any ThroughputBenchmark. This must be less than or equal to the minimum working set size, which is currently 4 KB.

6.11.2.19 `#define USE_OS_TIMER`

RECOMMENDED ENABLED. If enabled, uses the QPC timer on Windows and the POSIX `clock_gettime()` on GNU/Linux for all timing purposes.

6.11.2.20 `#define USE_TIME_BASED_BENCHMARKS`

RECOMMENDED ENABLED. All benchmarks run for an estimated amount of time, and the figures of merit are computed based on the amount of memory accesses completed in the time limit. This mode has more consistent runtime across different machines, memory performance, and working set sizes, but may have more conservative measurements for differing levels of cache hierarchy (overestimating latency and underestimating throughput).

6.12 src/include/Configurator.h File Reference

Header file for the Configurator class and some helper data structures.

```
#include <common.h>
#include <optionparser.h>
#include <MyArg.h>
#include <stdint>
#include <string>
```

Classes

- class `xmem::Configurator`

Handles all user input interpretation and generates the necessary flags for running benchmarks.

Enumerations

- enum `optionIndex` {
UNKNOWN, ALL, CHUNK_SIZE, EXTENSION,
OUTPUT_FILE, HELP, BASE_TEST_INDEX, NUM_WORKER_THREADS,
MEAS_LATENCY, ITERATIONS, RANDOM_ACCESS_PATTERN, SEQUENTIAL_ACCESS_PATTERN,
MEAS_THROUGHPUT, NUMA_DISABLE, VERBOSE, WORKING_SET_SIZE_PER_THREAD,
USE_LARGE_PAGES, USE_READS, USE_WRITES, STRIDE_SIZE }

Enumerates all possible types of command-line options.

Variables

- const Descriptor `xmem::usage []`

Command-line option descriptors as needed by stuff in [optionparser.h](#). This is basically the help message content.

6.12.1 Detailed Description

Header file for the Configurator class and some helper data structures.

6.13 src/include/ExampleArg.h File Reference

Slightly-modified third-party code related to OptionParser.

```
#include <optionparser.h>
#include <cstdint>
#include <stdio.h>
```

Classes

- class [xmem::ExampleArg](#)

6.13.1 Detailed Description

Slightly-modified third-party code related to OptionParser.

6.14 src/include/ext/DelayInjectedLoadedLatencyBenchmark/delay_injected_benchmark_kernels.h File Reference

Header file for benchmark kernel functions with integrated delays for doing the actual work we care about. :)

```
#include <cstdint>
```

Macros

- **#define my_nop2()** my_nop(); my_nop()
- **#define my_nop4()** my_nop2(); my_nop2()
- **#define my_nop8()** my_nop4(); my_nop4()
- **#define my_nop16()** my_nop8(); my_nop8()
- **#define my_nop32()** my_nop16(); my_nop16()
- **#define my_nop64()** my_nop32(); my_nop32()
- **#define my_nop128()** my_nop64(); my_nop64()
- **#define my_nop256()** my_nop128(); my_nop128()
- **#define my_nop512()** my_nop256(); my_nop256()
- **#define my_nop1024()** my_nop512(); my_nop512()

Functions

- **int32_t xmem::dummy_forwSequentialLoop_Word64_Delay1** (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- **int32_t xmem::dummy_forwSequentialLoop_Word64_Delay2** (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- **int32_t xmem::dummy_forwSequentialLoop_Word64_Delay4** (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- **int32_t xmem::dummy_forwSequentialLoop_Word64_Delay8** (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.

- `int32_t xmem::dummy_forwSequentialLoop_Word64_Delay16` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::dummy_forwSequentialLoop_Word64_Delay32` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::dummy_forwSequentialLoop_Word64_Delay64` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::dummy_forwSequentialLoop_Word64_Delay128` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::dummy_forwSequentialLoop_Word64_Delay256plus` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::dummy_forwSequentialLoop_Word256_Delay1` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::dummy_forwSequentialLoop_Word256_Delay2` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::dummy_forwSequentialLoop_Word256_Delay4` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::dummy_forwSequentialLoop_Word256_Delay8` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::dummy_forwSequentialLoop_Word256_Delay16` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::dummy_forwSequentialLoop_Word256_Delay32` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::dummy_forwSequentialLoop_Word256_Delay64plus` (void *start_address, void *end_address)
Used for measuring the time spent doing everything in delay-injected forward sequential Word 64 loops except for the memory access and delays themselves.
- `int32_t xmem::forwSequentialRead_Word64_Delay1` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 1 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word64_Delay2` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 2 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word64_Delay4` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 4 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word64_Delay8` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 8 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word64_Delay16` (void *start_address, void *end_address)

Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 16 delays (nops) are inserted between memory instructions.

- `int32_t xmem::forwSequentialRead_Word64_Delay32` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 32 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word64_Delay64` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 64 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word64_Delay128` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 128 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word64_Delay256` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 256 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word64_Delay512` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 512 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word64_Delay1024` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 1024 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word256_Delay1` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 1 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word256_Delay2` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 2 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word256_Delay4` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 4 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word256_Delay8` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 8 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word256_Delay16` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 16 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word256_Delay32` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 32 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word256_Delay64` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 64 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word256_Delay128` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 128 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word256_Delay256` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 256 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word256_Delay512` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 512 delays (nops) are inserted between memory instructions.
- `int32_t xmem::forwSequentialRead_Word256_Delay1024` (void *start_address, void *end_address)
Walks over the allocated memory forward sequentially, reading in 64-bit chunks. 1024 delays (nops) are inserted between memory instructions.

6.14.1 Detailed Description

Header file for benchmark kernel functions with integrated delays for doing the actual work we care about. :)

6.15 src/include/ext/DelayInjectedLoadedLatencyBenchmark/DelayInjectedLoadedLatencyBenchmark.h File Reference

Header file for the DelayInjectedLoadedLatencyBenchmark class.

6.15.1 Detailed Description

Header file for the DelayInjectedLoadedLatencyBenchmark class.

6.16 src/include/LatencyBenchmark.h File Reference

Header file for the LatencyBenchmark class.

```
#include <Benchmark.h>
#include <common.h>
#include <cstdint>
#include <string>
```

Classes

- class [xmem::LatencyBenchmark](#)

A type of benchmark that measures memory latency via random pointer chasing. Loading may be provided with separate threads which access memory as quickly as possible using given access patterns.

6.16.1 Detailed Description

Header file for the LatencyBenchmark class.

6.17 src/include/LatencyWorker.h File Reference

Header file for the LatencyWorker class.

```
#include <MemoryWorker.h>
#include <benchmark_kernels.h>
#include <common.h>
```

Classes

- class [xmem::LatencyWorker](#)

Multithreading-friendly class to do memory loading.

6.17.1 Detailed Description

Header file for the LatencyWorker class.

6.18 src/include/LoadWorker.h File Reference

Header file for the LoadWorker class.

```
#include <MemoryWorker.h>
#include <benchmark_kernels.h>
```

Classes

- class [xmem::LoadWorker](#)
Multithreading-friendly class to do memory loading.

6.18.1 Detailed Description

Header file for the LoadWorker class.

6.19 src/include/MemoryWorker.h File Reference

Header file for the MemoryWorker class.

```
#include <common.h>
#include <Runnable.h>
#include <stdint>
```

Classes

- class [xmem::MemoryWorker](#)
Multithreading-friendly class to run memory access kernels.

6.19.1 Detailed Description

Header file for the MemoryWorker class.

6.20 src/include/MyArg.h File Reference

Extensions to third-party optionparser-related code.

```
#include <ExampleArg.h>
#include <stdint>
#include <stdio.h>
#include <stdlib.h>
```


Classes

- class [xmem::MyArg](#)

6.20.1 Detailed Description

Extensions to third-party optionparser-related code.

6.21 src/include/optionparser.h File Reference

This is the only file required to use The Lean Mean C++ Option Parser. Just #include it and you're set.

Classes

- struct [xmem::Descriptor](#)
Describes an option, its help text (usage) and how it should be parsed.
- class [xmem::Option](#)
A parsed option from the command line together with its argument if it has one.
- struct [xmem::Arg](#)
Functions for checking the validity of option arguments.
- struct [xmem::Stats](#)
Determines the minimum lengths of the buffer and options arrays used for [Parser](#).
- class [xmem::Parser](#)
Checks argument vectors for validity and parses them into data structures that are easier to work with.
- struct [xmem::Parser::Action](#)
- class [xmem::Stats::CountOptionsAction](#)
- class [xmem::Parser::StoreOptionAction](#)
- struct [xmem::PrintUsagImplementation](#)
- struct [xmem::PrintUsagImplementation::IStringWriter](#)
- struct [xmem::PrintUsagImplementation::FunctionWriter< Function >](#)
- struct [xmem::PrintUsagImplementation::OStreamWriter< OStream >](#)
- struct [xmem::PrintUsagImplementation::TemporaryWriter< Temporary >](#)
- struct [xmem::PrintUsagImplementation::SyscallWriter< Syscall >](#)
- struct [xmem::PrintUsagImplementation::StreamWriter< Function, Stream >](#)
- class [xmem::PrintUsagImplementation::LinePartIterator](#)
- class [xmem::PrintUsagImplementation::LineWrapper](#)

Typedefs

- typedef ArgStatus(* [xmem::CheckArg](#))(const Option &option, bool msg)
Signature of functions that check if an argument is valid for a certain type of option.

Enumerations

- enum ArgStatus { [xmem::ARG_NONE](#), [xmem::ARG_OK](#), [xmem::ARG_IGNORE](#), [xmem::ARG_ILLEGAL](#) }
Possible results when checking if an argument is valid for a certain option.

Functions

- `template<typename OStream >`
`void xmem::printUsage (OStream &prn, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)`
Outputs a nicely formatted usage string with support for multi-column formatting and line-wrapping.
- `template<typename Function >`
`void xmem::printUsage (Function *prn, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)`
- `template<typename Temporary >`
`void xmem::printUsage (const Temporary &prn, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)`
- `template<typename Syscall >`
`void xmem::printUsage (Syscall *prn, int fd, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)`
- `template<typename Function, typename Stream >`
`void xmem::printUsage (Function *prn, Stream *stream, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)`

6.21.1 Detailed Description

This is the only file required to use The Lean Mean C++ Option Parser. Just `#include` it and you're set. The Lean Mean C++ Option Parser handles the program's command line arguments (`argc`, `argv`). It supports the short and long option formats of `getopt()`, `getopt_long()` and `getopt_long_only()` but has a more convenient interface. The following features set it apart from other option parsers:

Highlights:

- It is a header-only library. Just `#include "optionparser.h"` and you're set.
- It is freestanding. There are no dependencies whatsoever, not even the C or C++ standard library.
- It has a usage message formatter that supports column alignment and line wrapping. This aids localization because it adapts to translated strings that are shorter or longer (even if they contain Asian wide characters).
- Unlike `getopt()` and derivatives it doesn't force you to loop through options sequentially. Instead you can access options directly like this:
 - Test for presence of a switch in the argument vector:


```
if ( options[QUIET] ) ...
```
 - Evaluate `--enable-foo/--disable-foo` pair where the last one used wins:


```
if ( options[FOO].last()->type() == DISABLE ) ...
```
 - Cumulative option (`-v verbose`, `-vv more verbose`, `-vvv even more verbose`):


```
int verbosity = options[VERBOSE].count();
```
 - Iterate over all `--file=<fname>` arguments:


```
for (Option* opt = options[FILE]; opt; opt = opt->next())
    fname = opt->arg; ...
```
 - If you really want to, you can still process all arguments in order:


```
for (int i = 0; i < p.optionsCount(); ++i) {
    Option& opt = buffer[i];
    switch (opt.index()) {
        case HELP: ...
        case VERBOSE: ...
        case FILE:   fname = opt.arg; ...
        case UNKNOWN: ...
    }
}
```

Despite these features the code size remains tiny. It is smaller than `uClibc`'s GNU `getopt()` and just a couple 100 bytes larger than `uClibc`'s SUSv3 `getopt()`.

(This does not include the usage formatter, of course. But you don't have to use that.)

Download:

Tarball with examples and test programs: [optionparser-1.3.tar.gz](#)

Just the header (this is all you really need): [optionparser.h](#)

Changelog:

Version 1.3: Compatible with Microsoft Visual C++.

Version 1.2: Added `Option::namelen` and removed the extraction of short option characters into a special buffer.

Changed `Arg::Optional` to accept arguments if they are attached rather than separate. This is what GNU `getopt()` does and how POSIX recommends utilities should interpret their arguments.

Version 1.1: Optional mode with argument reordering as done by GNU `getopt()`, so that options and non-options can be mixed. See `Parser::parse()`.

Feedback:

Send questions, bug reports, feature requests etc. to: **`optionparser-feedback (a) lists.sourceforge.net`**

Example program:

(Note: `option::*` identifiers are links that take you to their documentation.)

```
#include <iostream>
#include "optionparser.h"

enum optionIndex { UNKNOWN, HELP, PLUS };
const option::Descriptor usage[] =
{
    {UNKNOWN, 0, "", "", option::Arg::None, "USAGE: example [options]\n\n"
     "Options:" },
    {HELP, 0, "", "help", option::Arg::None, " --help \tPrint usage and exit." },
    {PLUS, 0, "p", "plus", option::Arg::None, " --plus, -p \tIncrement count." },
    {UNKNOWN, 0, "", "", option::Arg::None, "\nExamples:\n"
     " example --unknown -- --this_is_no_option\n"
     " example -unk --plus -ppp file1 file2\n" },
    {0,0,0,0,0,0}
};

int main(int argc, char* argv[])
{
    argc-= (argc>0); argv+= (argc>0); // skip program name argv[0] if present
    option::Stats stats(usage, argc, argv);
    option::Option options[stats.options_max], buffer[stats.buffer_max];
    option::Parser parse(usage, argc, argv, options, buffer);

    if (parse.error())
        return 1;

    if (options[HELP] || argc == 0) {
        option::printUsage(std::cout, usage);
        return 0;
    }

    std::cout << "--plus count: " <<
        options[PLUS].count() << "\n";

    for (option::Option* opt = options[UNKNOWN]; opt; opt = opt->next())
        std::cout << "Unknown option: " << opt->name << "\n";

    for (int i = 0; i < parse.nonOptionsCount(); ++i)
        std::cout << "Non-option #" << i << ": " << parse.nonOption(i) << "\n";
}
```

Option syntax:

- The Lean Mean C++ Option Parser follows POSIX `getopt()` conventions and supports GNU-style `getopt_long()` long options as well as Perl-style single-minus long options (`getopt_long_only()`).
- short options have the format `-X` where `X` is any character that fits in a char.
- short options can be grouped, i.e. `-X -Y` is equivalent to `-XY`.
- a short option may take an argument either separate (`-X foo`) or attached (`-Xfoo`). You can make the parser accept the additional format `-X=foo` by registering `X` as a long option (in addition to being a short option) and enabling single-minus long options.
- an argument-taking short option may be grouped if it is the last in the group, e.g. `-ABCXfoo` or `-ABCX foo` (`foo` is the argument to the `-X` option).

- a lone minus character `'-'` is not treated as an option. It is customarily used where a file name is expected to refer to `stdin` or `stdout`.
 - long options have the format `-option-name`.
 - the option-name of a long option can be anything and include any characters. Even `=` characters will work, but don't do that.
 - [optional] long options may be abbreviated as long as the abbreviation is unambiguous. You can set a minimum length for abbreviations.
 - [optional] long options may begin with a single minus. The double minus form is always accepted, too.
 - a long option may take an argument either separate (`-option arg`) or attached (`-option=arg`). In the attached form the equals sign is mandatory.
 - an empty string can be passed as an attached long option argument: `-option-name=`. Note the distinction between an empty string as argument and no argument at all.
 - an empty string is permitted as separate argument to both long and short options.
 - Arguments to both short and long options may start with a `'-'` character. E.g. `-X-X`, `-X -X` or `-long-X=-X`. If `-X` and `-long-X` take an argument, that argument will be `"-X"` in all 3 cases.
 - If using the built-in `Arg::Optional`, optional arguments must be attached.
 - the special option `-` (i.e. without a name) terminates the list of options. Everything that follows is a non-option argument, even if it starts with a `'-'` character. The `-` itself will not appear in the parse results.
 - the first argument that doesn't start with `'-'` or `'-'` and does not belong to a preceding argument-taking option, will terminate the option list and is the first non-option argument. All following command line arguments are treated as non-option arguments, even if they start with `'-'`.
- NOTE: This behaviour is mandated by POSIX, but GNU `getopt()` only honours this if it is explicitly requested (e.g. by setting `POSIXLY_CORRECT`).
- You can enable the GNU behaviour by passing `true` as first argument to e.g. `Parser::parse()`.
- Arguments that look like options (i.e. `'-'` followed by at least 1 character) but aren't, are NOT treated as non-option arguments. They are treated as unknown options and are collected into a list of unknown options for error reporting.
- This means that in order to pass a first non-option argument beginning with the minus character it is required to use the `-` special option, e.g.

```
program -x -- --strange-filename
```

In this example, `--strange-filename` is a non-option argument. If the `-` were omitted, it would be treated as an unknown option.

See `option::Descriptor::longopt` for information on how to collect unknown options.

6.22 src/include/PowerReader.h File Reference

Header file for the `PowerReader` class.

```
#include <common.h>
#include <Runnable.h>
#include <cstdint>
#include <vector>
#include <string>
```

Classes

- class [xmem::PowerReader](#)

An abstract base class for measuring power from an arbitrary source. This class is runnable using a worker thread.

6.22.1 Detailed Description

Header file for the PowerReader class.

6.23 src/include/Runnable.h File Reference

Header file for the Runnable class.

```
#include <cstdint>
```

Classes

- class [xmem::Runnable](#)

A base class for any object that implements a thread-safe [run\(\)](#) function for use by [Thread](#) objects.

6.23.1 Detailed Description

Header file for the Runnable class.

6.24 src/include/Thread.h File Reference

Header file for the Thread class.

```
#include <Runnable.h>  
#include <cstdint>
```

Classes

- class [xmem::Thread](#)

a nice wrapped thread interface independent of particular OS API

6.24.1 Detailed Description

Header file for the Thread class.

6.25 src/include/ThroughputBenchmark.h File Reference

Header file for the ThroughputBenchmark class.

```
#include <Benchmark.h>  
#include <common.h>  
#include <cstdint>  
#include <string>
```

Classes

- class [xmem::ThroughputBenchmark](#)
A type of benchmark that measures memory throughput.

6.25.1 Detailed Description

Header file for the ThroughputBenchmark class.

6.26 src/include/Timer.h File Reference

Header file for the Timer class.

```
#include <common.h>
#include <stdint>
```

Classes

- class [xmem::Timer](#)
This class abstracts some characteristics of simple high resolution stopwatch timer. However, due to the inability or complexity of abstracting shared hardware timers, this class does not actually provide start and stop functions.

6.26.1 Detailed Description

Header file for the Timer class.

6.27 src/include/win/win_common_third_party.h File Reference

Header file for some third-party helper code for working with Windows APIs.

6.27.1 Detailed Description

Header file for some third-party helper code for working with Windows APIs.

6.28 src/include/win/win_CPdhQuery.h File Reference

Header and implementation file for some third-party code for measuring Windows OS-exposed performance counters.

6.28.1 Detailed Description

Header and implementation file for some third-party code for measuring Windows OS-exposed performance counters.

6.29 src/include/win/WindowsDRAMPowerReader.h File Reference

Header file for the WindowsDRAMPowerReader class.

6.29.1 Detailed Description

Header file for the WindowsDRAMPowerReader class.

6.30 src/LatencyBenchmark.cpp File Reference

Implementation file for the LatencyBenchmark class.

```
#include <LatencyBenchmark.h>
#include <common.h>
#include <benchmark_kernels.h>
#include <MemoryWorker.h>
#include <LatencyWorker.h>
#include <LoadWorker.h>
#include <iostream>
#include <random>
#include <assert.h>
#include <time.h>
```

6.30.1 Detailed Description

Implementation file for the LatencyBenchmark class.

6.31 src/LatencyWorker.cpp File Reference

Implementation file for the LatencyWorker class.

```
#include <LatencyWorker.h>
#include <benchmark_kernels.h>
#include <common.h>
#include <iostream>
```

6.31.1 Detailed Description

Implementation file for the LatencyWorker class.

6.32 src/LoadWorker.cpp File Reference

Implementation file for the LoadWorker class.

```
#include <LoadWorker.h>
#include <benchmark_kernels.h>
#include <common.h>
#include <iostream>
```

6.32.1 Detailed Description

Implementation file for the LoadWorker class.

6.33 src/main.cpp File Reference

main entry point to the tool

```
#include <common.h>
#include <build_datetime.h>
#include <Configurator.h>
#include <BenchmarkManager.h>
#include <iostream>
#include <string>
```

Functions

- int `main` (int argc, char *argv[])

The main entry point to the program.

6.33.1 Detailed Description

main entry point to the tool This tool is designed to measure bandwidth and latency of the memory system using several access patterns, strides, and working set sizes. The primary goal is to measure DRAM performance, although it can also measure cache performance depending on the configuration.

6.34 src/MemoryWorker.cpp File Reference

Implementation file for the MemoryWorker class.

```
#include <MemoryWorker.h>
#include <common.h>
```

6.34.1 Detailed Description

Implementation file for the MemoryWorker class.

6.35 src/PowerReader.cpp File Reference

Implementation file for the PowerReader class.

```
#include <PowerReader.h>
#include <common.h>
#include <stdint>
#include <vector>
#include <iostream>
```

6.35.1 Detailed Description

Implementation file for the PowerReader class.

6.36 src/Runnable.cpp File Reference

Implementation file for the Runnable class.

```
#include <Runnable.h>
#include <iostream>
```

Variables

- return **false**

6.36.1 Detailed Description

Implementation file for the Runnable class.

6.37 src/Thread.cpp File Reference

Implementation file for the Thread class.

```
#include <Thread.h>
#include <stdlib.h>
#include <iostream>
```

Variables

- return **false**
- return **true**

6.37.1 Detailed Description

Implementation file for the Thread class.

6.38 src/ThroughputBenchmark.cpp File Reference

Implementation file for the ThroughputBenchmark class.

```
#include <ThroughputBenchmark.h>
#include <common.h>
#include <LoadWorker.h>
#include <Thread.h>
#include <iostream>
#include <assert.h>
#include <time.h>
```

6.38.1 Detailed Description

Implementation file for the ThroughputBenchmark class.

6.39 `src/Timer.cpp` File Reference

Implementation file for the Timer class.

```
#include <Timer.h>
#include <common.h>
```

6.39.1 Detailed Description

Implementation file for the Timer class.

6.40 `src/win/win_common_third_party.cpp` File Reference

Implementation file for some third-party helper code for working with Windows APIs.

6.40.1 Detailed Description

Implementation file for some third-party helper code for working with Windows APIs.

6.41 `src/win/WindowsDRAMPowerReader.cpp` File Reference

Implementation file for the WindowsDRAMPowerReader class.

6.41.1 Detailed Description

Implementation file for the WindowsDRAMPowerReader class.

Index

- ~Thread
 - xmem::Thread, [70](#)
- _acquireLock
 - xmem::Runnable, [64](#)
- _adjusted_ticks
 - xmem::MemoryWorker, [45](#)
- _averageLoadMetric
 - xmem::LatencyBenchmark, [37](#)
- _averageMetric
 - xmem::Benchmark, [21](#)
- _average_dram_power_socket
 - xmem::Benchmark, [21](#)
- _average_power
 - xmem::PowerReader, [61](#)
- _bytes_per_pass
 - xmem::MemoryWorker, [45](#)
- _chunk_size
 - xmem::Benchmark, [21](#)
- _completed
 - xmem::MemoryWorker, [45](#)
- _cpu_affinity
 - xmem::MemoryWorker, [45](#)
 - xmem::PowerReader, [61](#)
- _cpu_node
 - xmem::Benchmark, [21](#)
- _dram_power_readers
 - xmem::Benchmark, [21](#)
- _dram_power_threads
 - xmem::Benchmark, [21](#)
- _elapsed_dummy_ticks
 - xmem::MemoryWorker, [45](#)
- _elapsed_ticks
 - xmem::MemoryWorker, [45](#)
- _hasRun
 - xmem::Benchmark, [21](#)
- _iterations
 - xmem::Benchmark, [21](#)
- _len
 - xmem::Benchmark, [21](#)
 - xmem::MemoryWorker, [45](#)
- _loadMetricOnIter
 - xmem::LatencyBenchmark, [37](#)
- _mem_array
 - xmem::Benchmark, [21](#)
 - xmem::MemoryWorker, [45](#)
- _mem_node
 - xmem::Benchmark, [22](#)
- _metricOnIter
 - xmem::Benchmark, [22](#)
- _metricUnits
 - xmem::Benchmark, [22](#)
- _name
 - xmem::Benchmark, [22](#)
 - xmem::PowerReader, [61](#)
- _ns_per_tick
 - xmem::Timer, [74](#)
- _num_samples
 - xmem::PowerReader, [61](#)
- _num_worker_threads
 - xmem::Benchmark, [22](#)
- _obj_valid
 - xmem::Benchmark, [22](#)
- _passes
 - xmem::MemoryWorker, [45](#)
- _pattern_mode
 - xmem::Benchmark, [22](#)
- _peak_dram_power_socket
 - xmem::Benchmark, [22](#)
- _peak_power
 - xmem::PowerReader, [61](#)
- _power_trace
 - xmem::PowerReader, [61](#)
- _power_units
 - xmem::PowerReader, [61](#)
- _releaseLock
 - xmem::Runnable, [65](#)
- _run_core
 - xmem::Benchmark, [17](#)
 - xmem::LatencyBenchmark, [36](#)
 - xmem::ThroughputBenchmark, [73](#)
- _rw_mode
 - xmem::Benchmark, [22](#)
- _sampling_period
 - xmem::PowerReader, [62](#)
- _start_power_threads
 - xmem::Benchmark, [18](#)
- _stop_power_threads
 - xmem::Benchmark, [18](#)
- _stop_signal
 - xmem::PowerReader, [62](#)
- _stride_size
 - xmem::Benchmark, [22](#)
- _ticks_per_ms
 - xmem::Timer, [74](#)
- _warning
 - xmem::Benchmark, [22](#)
 - xmem::MemoryWorker, [46](#)
- add

- xmem::Stats, 66
- append
 - xmem::Option, 48
- arg
 - xmem::Option, 51
- Benchmark
 - xmem::Benchmark, 17
- BenchmarkManager
 - xmem::BenchmarkManager, 23
- buffer_max
 - xmem::Stats, 67
- calculateMetrics
 - xmem::PowerReader, 59
- cancel
 - xmem::Thread, 71
- check_arg
 - xmem::Descriptor, 33
- clear
 - xmem::PowerReader, 59
- clear_and_reset
 - xmem::PowerReader, 59
- common.h
 - DEFAULT_NUM_NODES, 84
 - DEFAULT_PAGE_SIZE, 84
 - MIN_ELAPSED_TICKS, 84
 - USE_OS_TIMER, 85
- completed
 - xmem::Thread, 71
- configureFromInput
 - xmem::Configurator, 25
- count
 - xmem::Option, 48
- CountOptionsAction
 - xmem::Stats::CountOptionsAction, 31
- create_and_start
 - xmem::Thread, 71
- created
 - xmem::Thread, 71
- DEFAULT_NUM_NODES
 - common.h, 84
- DEFAULT_PAGE_SIZE
 - common.h, 84
- desc
 - xmem::Option, 51
- error
 - xmem::Parser, 55
- extensionsEnabled
 - xmem::Configurator, 27
- finished
 - xmem::Parser::Action, 13
 - xmem::Parser::StoreOptionAction, 68
- first
 - xmem::Option, 48
- get_ns_per_tick
 - xmem::Timer, 74
- get_ticks_per_ms
 - xmem::Timer, 74
- getAdjustedTicks
 - xmem::MemoryWorker, 44
- getAverageDRAMPower
 - xmem::Benchmark, 18
- getAverageMetric
 - xmem::Benchmark, 18
- getAveragePower
 - xmem::PowerReader, 59
- getAvgLoadMetric
 - xmem::LatencyBenchmark, 36
- getBytesPerPass
 - xmem::MemoryWorker, 44
- getCPUNode
 - xmem::Benchmark, 18
- getChunkSize
 - xmem::Benchmark, 18
- getElapsedDummyTicks
 - xmem::MemoryWorker, 44
- getElapsedTicks
 - xmem::MemoryWorker, 44
- getExitCode
 - xmem::Thread, 71
- getIterations
 - xmem::Benchmark, 18
- getIterationsPerTest
 - xmem::Configurator, 27
- getLastSample
 - xmem::PowerReader, 60
- getLen
 - xmem::Benchmark, 19
 - xmem::MemoryWorker, 44
- getLoadMetricOnIter
 - xmem::LatencyBenchmark, 37
- getMemNode
 - xmem::Benchmark, 19
- getMetricOnIter
 - xmem::Benchmark, 19
- getMetricUnits
 - xmem::Benchmark, 19
- getName
 - xmem::Benchmark, 19
- getNumSamples
 - xmem::PowerReader, 60
- getNumThreads
 - xmem::Benchmark, 19
- getNumWorkerThreads
 - xmem::Configurator, 27
- getOutputFilename
 - xmem::Configurator, 27
- getPasses
 - xmem::MemoryWorker, 44
- getPatternMode
 - xmem::Benchmark, 20
- getPeakDRAMPower
 - xmem::Benchmark, 20

- getPeakPower
 - xmem::PowerReader, [60](#)
- getPowerTrace
 - xmem::PowerReader, [60](#)
- getPowerUnits
 - xmem::PowerReader, [60](#)
- getRWMode
 - xmem::Benchmark, [20](#)
- getSamplingPeriod
 - xmem::PowerReader, [60](#)
- getStartingTestIndex
 - xmem::Configurator, [27](#)
- getStrideSize
 - xmem::Benchmark, [20](#)
- getTarget
 - xmem::Thread, [71](#)
- getWorkingSetSizePerThread
 - xmem::Configurator, [27](#)
- hadWarning
 - xmem::MemoryWorker, [45](#)
- hasRun
 - xmem::Benchmark, [20](#)
- help
 - xmem::Descriptor, [33](#)
- index
 - xmem::Descriptor, [33](#)
- isFirst
 - xmem::Option, [49](#)
- isLast
 - xmem::Option, [49](#)
- isNUMAEnabled
 - xmem::Configurator, [27](#)
- isThreadRunning
 - xmem::Thread, [71](#)
- isThreadSuspended
 - xmem::Thread, [71](#)
- isValid
 - xmem::Benchmark, [20](#)
- isWideChar
 - xmem::PrintUsageImplementation, [62](#)
- join
 - xmem::Thread, [72](#)
- last
 - xmem::Option, [49](#)
- latencyTestSelected
 - xmem::Configurator, [28](#)
- LatencyWorker
 - xmem::LatencyWorker, [38](#)
- LineWrapper
 - xmem::PrintUsageImplementation::LineWrapper, [40](#)
- LoadWorker
 - xmem::LoadWorker, [41](#)
- longopt
 - xmem::Descriptor, [33](#)
- MIN_ELAPSED_TICKS
 - common.h, [84](#)
- MemoryWorker
 - xmem::MemoryWorker, [43](#)
- name
 - xmem::Option, [51](#)
 - xmem::PowerReader, [61](#)
- namelen
 - xmem::Option, [51](#)
- next
 - xmem::Option, [49](#)
 - xmem::PrintUsageImplementation::LinePartIterator, [39](#)
- nextRow
 - xmem::PrintUsageImplementation::LinePartIterator, [39](#)
- nextTable
 - xmem::PrintUsageImplementation::LinePartIterator, [39](#)
- nextwrap
 - xmem::Option, [49](#)
- nonOptions
 - xmem::Parser, [55](#)
- nonOptionsCount
 - xmem::Parser, [55](#)
- operator const Option *
 - xmem::Option, [49](#)
- operator Option *
 - xmem::Option, [50](#)
- operator=
 - xmem::Option, [50](#)
- Option
 - xmem::Option, [48](#)
- options_max
 - xmem::Stats, [67](#)
- optionsCount
 - xmem::Parser, [55](#)
- parse
 - xmem::Parser, [55](#)
- Parser
 - xmem::Parser, [54](#)
- perform
 - xmem::Parser::Action, [13](#)
 - xmem::Parser::StoreOptionAction, [68](#)
 - xmem::Stats::CountOptionsAction, [32](#)
- PowerReader
 - xmem::PowerReader, [59](#)
- prev
 - xmem::Option, [50](#)
- prevwrap
 - xmem::Option, [50](#)
- process
 - xmem::PrintUsageImplementation::LineWrapper, [40](#)
- run

- xmem::Benchmark, 20
- runAll
 - xmem::BenchmarkManager, 23
- runExtDelayInjectedLoadedLatencyBenchmark
 - xmem::Configurator, 28
- runLatencyBenchmarks
 - xmem::BenchmarkManager, 23
- runThroughputBenchmarks
 - xmem::BenchmarkManager, 24
- setUseOutputFile
 - xmem::Configurator, 28
- shortopt
 - xmem::Descriptor, 34
- src/Benchmark.cpp, 75
- src/BenchmarkManager.cpp, 76
- src/Configurator.cpp, 77
- src/LatencyBenchmark.cpp, 97
- src/LatencyWorker.cpp, 97
- src/LoadWorker.cpp, 97
- src/MemoryWorker.cpp, 98
- src/PowerReader.cpp, 98
- src/Runnable.cpp, 99
- src/Thread.cpp, 99
- src/ThroughputBenchmark.cpp, 99
- src/Timer.cpp, 100
- src/benchmark_kernels.cpp, 75
- src/common.cpp, 76
- src/ext/DelayInjectedLoadedLatencyBenchmark/Delay-
InjectedLoadedLatencyBenchmark.cpp, 77
- src/ext/DelayInjectedLoadedLatencyBenchmark/delay_
injected_benchmark_kernels.cpp, 77
- src/include/Benchmark.h, 78
- src/include/BenchmarkManager.h, 80
- src/include/Configurator.h, 85
- src/include/ExampleArg.h, 86
- src/include/LatencyBenchmark.h, 89
- src/include/LatencyWorker.h, 89
- src/include/LoadWorker.h, 90
- src/include/MemoryWorker.h, 90
- src/include/MyArg.h, 90
- src/include/PowerReader.h, 94
- src/include/Runnable.h, 95
- src/include/Thread.h, 95
- src/include/ThroughputBenchmark.h, 95
- src/include/Timer.h, 96
- src/include/benchmark_kernels.h, 78
- src/include/common.h, 81
- src/include/ext/DelayInjectedLoadedLatencyBenchmark/-
DelayInjectedLoadedLatencyBenchmark.h, 89
- src/include/ext/DelayInjectedLoadedLatencyBenchmark/delay_
_injected_benchmark_kernels.h, 86
- src/include/optionparser.h, 91
- src/include/win/WindowsDRAMPowerReader.h, 96
- src/include/win/win_CPdhQuery.h, 96
- src/include/win/win_common_third_party.h, 96
- src/main.cpp, 98
- src/win/WindowsDRAMPowerReader.cpp, 100
- src/win/win_common_third_party.cpp, 100
- started
 - xmem::Thread, 72
- Stats
 - xmem::Stats, 66
- stop
 - xmem::PowerReader, 61
- StoreOptionAction
 - xmem::Parser::StoreOptionAction, 67
- Thread
 - xmem::Thread, 70
- throughputTestSelected
 - xmem::Configurator, 28
- type
 - xmem::Descriptor, 34
 - xmem::Option, 50
- USE_OS_TIMER
 - common.h, 85
- useChunk32b
 - xmem::Configurator, 28
- useLargePages
 - xmem::Configurator, 28
- useOutputFile
 - xmem::Configurator, 29
- useRandomAccessPattern
 - xmem::Configurator, 29
- useReads
 - xmem::Configurator, 29
- useSequentialAccessPattern
 - xmem::Configurator, 29
- useStrideN1
 - xmem::Configurator, 29
- useStrideN16
 - xmem::Configurator, 29
- useStrideN2
 - xmem::Configurator, 29
- useStrideN4
 - xmem::Configurator, 30
- useStrideN8
 - xmem::Configurator, 30
- useStrideP1
 - xmem::Configurator, 30
- useStrideP16
 - xmem::Configurator, 30
- useStrideP2
 - xmem::Configurator, 30
- useStrideP4
 - xmem::Configurator, 30
- useStrideP8
 - xmem::Configurator, 30
- useWrites
 - xmem::Configurator, 31
- validTarget
 - xmem::Thread, 72
- verboseMode
 - xmem::Configurator, 31

- xmem::Arg, 14
- xmem::Benchmark, 15
 - _averageMetric, 21
 - _average_dram_power_socket, 21
 - _chunk_size, 21
 - _cpu_node, 21
 - _dram_power_readers, 21
 - _dram_power_threads, 21
 - _hasRun, 21
 - _iterations, 21
 - _len, 21
 - _mem_array, 21
 - _mem_node, 22
 - _metricOnIter, 22
 - _metricUnits, 22
 - _name, 22
 - _num_worker_threads, 22
 - _obj_valid, 22
 - _pattern_mode, 22
 - _peak_dram_power_socket, 22
 - _run_core, 17
 - _rw_mode, 22
 - _start_power_threads, 18
 - _stop_power_threads, 18
 - _stride_size, 22
 - _warning, 22
- Benchmark, 17
- getAverageDRAMPower, 18
- getAverageMetric, 18
- getCPUNode, 18
- getChunkSize, 18
- getIterations, 18
- getLen, 19
- getMemNode, 19
- getMetricOnIter, 19
- getMetricUnits, 19
- getName, 19
- getNumThreads, 19
- getPatternMode, 20
- getPeakDRAMPower, 20
- getRWMode, 20
- getStrideSize, 20
- hasRun, 20
- isValid, 20
- run, 20
- xmem::BenchmarkManager, 23
 - BenchmarkManager, 23
 - runAll, 23
 - runLatencyBenchmarks, 23
 - runThroughputBenchmarks, 24
- xmem::Configurator, 24
 - configureFromInput, 25
 - extensionsEnabled, 27
 - getIterationsPerTest, 27
 - getNumWorkerThreads, 27
 - getOutputFilename, 27
 - getStartingTestIndex, 27
 - getWorkingSetSizePerThread, 27
- isNUMAEnabled, 27
- latencyTestSelected, 28
- runExtDelayInjectedLoadedLatencyBenchmark, 28
- setUseOutputFile, 28
- throughputTestSelected, 28
- useChunk32b, 28
- useLargePages, 28
- useOutputFile, 29
- useRandomAccessPattern, 29
- useReads, 29
- useSequentialAccessPattern, 29
- useStrideN1, 29
- useStrideN16, 29
- useStrideN2, 29
- useStrideN4, 30
- useStrideN8, 30
- useStrideP1, 30
- useStrideP16, 30
- useStrideP2, 30
- useStrideP4, 30
- useStrideP8, 30
- useWrites, 31
- verboseMode, 31
- xmem::Descriptor, 32
 - check_arg, 33
 - help, 33
 - index, 33
 - longopt, 33
 - shortopt, 34
 - type, 34
- xmem::ExampleArg, 34
- xmem::LatencyBenchmark, 35
 - _averageLoadMetric, 37
 - _loadMetricOnIter, 37
 - _run_core, 36
 - getAvgLoadMetric, 36
 - getLoadMetricOnIter, 37
- xmem::LatencyWorker, 37
 - LatencyWorker, 38
- xmem::LoadWorker, 40
 - LoadWorker, 41
- xmem::MemoryWorker, 41
 - _adjusted_ticks, 45
 - _bytes_per_pass, 45
 - _completed, 45
 - _cpu_affinity, 45
 - _elapsed_dummy_ticks, 45
 - _elapsed_ticks, 45
 - _len, 45
 - _mem_array, 45
 - _passes, 45
 - _warning, 46
 - getAdjustedTicks, 44
 - getBytesPerPass, 44
 - getElapsedDummyTicks, 44
 - getElapsedTicks, 44
 - getLen, 44
 - getPasses, 44

- hadWarning, 45
- MemoryWorker, 43
- xmem::MyArg, 46
- xmem::Option, 46
 - append, 48
 - arg, 51
 - count, 48
 - desc, 51
 - first, 48
 - isFirst, 49
 - isLast, 49
 - last, 49
 - name, 51
 - namelen, 51
 - next, 49
 - nextwrap, 49
 - operator const Option *, 49
 - operator Option *, 50
 - operator=, 50
 - Option, 48
 - prev, 50
 - prevwrap, 50
 - type, 50
- xmem::Parser, 52
 - error, 55
 - nonOptions, 55
 - nonOptionsCount, 55
 - optionsCount, 55
 - parse, 55
 - Parser, 54
- xmem::Parser::Action, 13
 - finished, 13
 - perform, 13
- xmem::Parser::StoreOptionAction, 67
 - finished, 68
 - perform, 68
 - StoreOptionAction, 67
- xmem::PowerReader, 57
 - _average_power, 61
 - _cpu_affinity, 61
 - _name, 61
 - _num_samples, 61
 - _peak_power, 61
 - _power_trace, 61
 - _power_units, 61
 - _sampling_period, 62
 - _stop_signal, 62
 - calculateMetrics, 59
 - clear, 59
 - clear_and_reset, 59
 - getAveragePower, 59
 - getLastSample, 60
 - getNumSamples, 60
 - getPeakPower, 60
 - getPowerTrace, 60
 - getPowerUnits, 60
 - getSamplingPeriod, 60
 - name, 61
 - PowerReader, 59
 - stop, 61
- xmem::PrintUsageImplementation, 62
 - isWideChar, 62
- xmem::PrintUsageImplementation::FunctionWriter< Function >, 35
- xmem::PrintUsageImplementation::IStringWriter, 35
- xmem::PrintUsageImplementation::LinePartIterator, 38
 - next, 39
 - nextRow, 39
 - nextTable, 39
- xmem::PrintUsageImplementation::LineWrapper, 39
 - LineWrapper, 40
 - process, 40
- xmem::PrintUsageImplementation::OStreamWriter< O-Stream >, 52
- xmem::PrintUsageImplementation::StreamWriter< Function, Stream >, 68
- xmem::PrintUsageImplementation::SyscallWriter< Syscall >, 69
- xmem::PrintUsageImplementation::TemporaryWriter< Temporary >, 69
- xmem::Runnable, 63
 - _acquireLock, 64
 - _releaseLock, 65
- xmem::Stats, 65
 - add, 66
 - buffer_max, 67
 - options_max, 67
 - Stats, 66
- xmem::Stats::CountOptionsAction, 31
 - CountOptionsAction, 31
 - perform, 32
- xmem::Thread, 70
 - ~Thread, 70
 - cancel, 71
 - completed, 71
 - create_and_start, 71
 - created, 71
 - getExitCode, 71
 - getTarget, 71
 - isThreadRunning, 71
 - isThreadSuspended, 71
 - join, 72
 - started, 72
 - Thread, 70
 - validTarget, 72
- xmem::ThroughputBenchmark, 72
 - _run_core, 73
- xmem::Timer, 73
 - _ns_per_tick, 74
 - _ticks_per_ms, 74
 - get_ns_per_tick, 74
 - get_ticks_per_ms, 74