X-Mem: Extensible Memory Micro-benchmarking Tool

1.0

Generated by Doxygen 1.8.8

Wed Sep 24 2014 14:55:40

Contents

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 xmem Namespace Reference

The namespace of The Lean Mean C++ Option Parser.

Namespaces

- benchmark
- common
- config
- power
- thread
- timers

5.1.1 Detailed Description

The namespace of The Lean Mean C++ Option Parser.

5.2 xmem::benchmark Namespace Reference

Namespaces

- benchmark_kernels

Classes

- class Benchmark

  Flexible abstract class for any memory benchmark.
- class BenchmarkManager

  Manages running all benchmarks at a high level.
- class LatencyBenchmark

  A type of benchmark that measures memory latency via random pointer chasing. TODO: loaded latency tests.
- class ThroughputBenchmark

  A type of benchmark that measures memory throughput either via sequential, strided sequential, or random access patterns.
- class ThroughputBenchmarkWorker

  Helper multithreading-friendly class to do the core throughput benchmark.

5.3 xmem::benchmark::benchmark_kernels Namespace Reference

Functions

- int32_t dummy_chasePointers (uintptr_t *, uintptr_t **, size_t len)

  Mimics the __chasePointers() method but doesn't do the memory accesses.
- int32_t 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 dummy_empty (void *, void *)

  Does nothing. Used for measuring the time it takes just to call a benchmark routine via function pointer.
- int32_t 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 dummy_forwSequentialLoop_Word64 (void *start_address, void *end_address)

  Used for measuring the time spent doing everything in forward sequential Word 64 loops except for the memory access itself.
- int32_t dummy_forwSequentialLoop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward sequential Word 128 loops except for the memory access itself.
- int32_t dummy_forwSequentialLoop_Word256 (void *start_address, void *end_address)

  Used for measuring the time spent doing everything in forward sequential Word 256 loops except for the memory access itself.
- int32_t 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 dummy_revSequentialLoop_Word64 (void *start_address, void *end_address)

  Used for measuring the time spent doing everything in reverse sequential Word 64 loops except for the memory access itself.
- int32_t dummy_revSequentialLoop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse sequential Word 128 loops except for the memory access itself.
- int32_t dummy_revSequentialLoop_Word256 (void *start_address, void *end_address)

  Used for measuring the time spent doing everything in reverse sequential Word 256 loops except for the memory access itself.
- int32_t dummy_forwStride2Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 2-strided Word 32 loops except for the memory access itself.
- int32_t dummy_forwStride2Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 2-strided Word 64 loops except for the memory access itself.
- int32_t dummy_forwStride2Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 2-strided Word 128 loops except for the memory access itself.
- int32_t dummy_forwStride2Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 2-strided Word 256 loops except for the memory access itself.
- int32_t dummy_revStride2Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 32 loops except for the memory access itself.
- int32_t dummy_revStride2Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 64 loops except for the memory access itself.
- int32_t dummy_revStride2Loop_Word128 (void *start_address, void *end_address)

TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 128 loops except for the memory access itself.

- int32_t dummy_revStride2Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 256 loops except for the memory access itself.

- int32_t dummy_forwStride4Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 4-strided Word 32 loops except for the memory access itself.

- int32_t dummy_forwStride4Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 4-strided Word 64 loops except for the memory access itself.

- int32_t dummy_forwStride4Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 4-strided Word 128 loops except for the memory access itself.

- int32_t dummy_forwStride4Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 4-strided Word 256 loops except for the memory access itself.

- int32_t dummy_revStride4Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 32 loops except for the memory access itself.

- int32_t dummy_revStride4Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 64 loops except for the memory access itself.

- int32_t dummy_revStride4Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 128 loops except for the memory access itself.

- int32_t dummy_revStride4Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 256 loops except for the memory access itself.

- int32_t dummy_forwStride8Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 8-strided Word 32 loops except for the memory access itself.

- int32_t dummy_forwStride8Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 8-strided Word 64 loops except for the memory access itself.

- int32_t dummy_forwStride8Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 8-strided Word 128 loops except for the memory access itself.

- int32_t dummy_forwStride8Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 8-strided Word 256 loops except for the memory access itself.

- int32_t dummy_revStride8Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 32 loops except for the memory access itself.

- int32_t dummy_revStride8Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 64 loops except for the memory access itself.

- int32_t dummy_revStride8Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 128 loops except for the memory access itself.

- int32_t dummy_revStride8Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 256 loops except for the memory access itself.

- int32_t dummy_forwStride16Loop_Word32 (void *start_address, void *end_address)

TODO. Used for measuring the time spent doing everything in forward 16-strided Word 32 loops except for the memory access itself.

- int32_t dummy_forwStride16Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 16-strided Word 64 loops except for the memory access itself.
- int32_t dummy_forwStride16Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 16-strided Word 128 loops except for the memory access itself.
- int32_t dummy_forwStride16Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 16-strided Word 256 loops except for the memory access itself.
- int32_t dummy_revStride16Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 32 loops except for the memory access itself.
- int32_t dummy_revStride16Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 64 loops except for the memory access itself.
- int32_t dummy_revStride16Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 128 loops except for the memory access itself.
- int32_t dummy_revStride16Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 256 loops except for the memory access itself.
- int32_t dummy_randomLoop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in random Word 32 loops except for the memory access itself.
- int32_t dummy_randomLoop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in random Word 64 loops except for the memory access itself.
- int32_t dummy_randomLoop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in random Word 128 loops except for the memory access itself.
- int32_t dummy_randomLoop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in random Word 256 loops except for the memory access itself.
- int32_t forwSequentialRead_Word32 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, reading in 32-bit chunks.
- int32_t forwSequentialRead_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, reading in 64-bit chunks.
- int32_t forwSequentialRead_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory forward sequentially, reading in 128-bit chunks.
- int32_t forwSequentialRead_Word256 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, reading in 256-bit chunks.
- int32_t revSequentialRead_Word32 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, reading in 32-bit chunks.
- int32_t revSequentialRead_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, reading in 64-bit chunks.
- int32_t revSequentialRead_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory reverse sequentially, reading in 128-bit chunks.
- int32_t revSequentialRead_Word256 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, reading in 256-bit chunks.
- int32_t forwSequentialWrite_Word32 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, writing in 32-bit chunks.

- int32_t forwSequentialWrite_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, writing in 64-bit chunks.
- int32_t forwSequentialWrite_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory forward sequentially, writing in 128-bit chunks.
- int32_t forwSequentialWrite_Word256 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, writing in 256-bit chunks.
- int32_t revSequentialWrite_Word32 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, writing in 32-bit chunks.
- int32_t revSequentialWrite_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, writing in 64-bit chunks.
- int32_t revSequentialWrite_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory reverse sequentially, writing in 128-bit chunks.
- int32_t revSequentialWrite_Word256 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, writing in 256-bit chunks.
- int32_t 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 forwStride2Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 2, reading in 64-bit chunks.
- int32_t forwStride2Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 2, reading in 128-bit chunks.
- int32_t forwStride2Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 2, reading in 256-bit chunks.
- int32_t 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 revStride2Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 2, reading in 64-bit chunks.
- int32_t revStride2Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 2, reading in 128-bit chunks.
- int32_t revStride2Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 2, reading in 256-bit chunks.
- int32_t 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 forwStride2Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 2, writing in 64-bit chunks.
- int32_t forwStride2Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 2, writing in 128-bit chunks.
- int32_t forwStride2Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 2, writing in 256-bit chunks.
- int32_t 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 revStride2Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 2, writing in 64-bit chunks.
- int32_t revStride2Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 2, writing in 128-bit chunks.
- int32_t revStride2Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 2, writing in 256-bit chunks.
- int32_t 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 forwStride4Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 4, reading in 64-bit chunks.
- int32_t forwStride4Read_Word128 (void *start_address, void *end_address)

TODO. Walks over the allocated memory in forward strides of size 4, reading in 128-bit chunks.

- int32_t forwStride4Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 4, reading in 256-bit chunks.
- int32_t 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 revStride4Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 4, reading in 64-bit chunks.
- int32_t revStride4Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 4, reading in 128-bit chunks.
- int32_t revStride4Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 4, reading in 256-bit chunks.
- int32_t 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 forwStride4Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 4, writing in 64-bit chunks.
- int32_t forwStride4Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 4, writing in 128-bit chunks.
- int32_t forwStride4Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 4, writing in 256-bit chunks.
- int32_t 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 revStride4Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 4, writing in 64-bit chunks.
- int32_t revStride4Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 4, writing in 128-bit chunks.
- int32_t revStride4Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 4, writing in 256-bit chunks.
- int32_t 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 forwStride8Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 8, reading in 64-bit chunks.
- int32_t forwStride8Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 8, reading in 128-bit chunks.
- int32_t forwStride8Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 8, reading in 256-bit chunks.
- int32_t 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 revStride8Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 8, reading in 64-bit chunks.
- int32_t revStride8Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 8, reading in 128-bit chunks.
- int32_t revStride8Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 8, reading in 256-bit chunks.
- int32_t 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 forwStride8Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 8, writing in 64-bit chunks.
- int32_t forwStride8Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 8, writing in 128-bit chunks.
- int32_t forwStride8Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 8, writing in 256-bit chunks.

- int32_t 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 revStride8Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 8, writing in 64-bit chunks.
- int32_t revStride8Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 8, writing in 128-bit chunks.
- int32_t revStride8Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 8, writing in 256-bit chunks.
- int32_t 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 forwStride16Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 16, reading in 64-bit chunks.
- int32_t forwStride16Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 16, reading in 128-bit chunks.
- int32_t forwStride16Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 16, reading in 256-bit chunks.
- int32_t 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 revStride16Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 16, reading in 64-bit chunks.
- int32_t revStride16Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 16, reading in 128-bit chunks.
- int32_t revStride16Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 16, reading in 256-bit chunks.
- int32_t 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 forwStride16Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 16, writing in 64-bit chunks.
- int32_t forwStride16Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 16, writing in 128-bit chunks.
- int32_t forwStride16Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 16, writing in 256-bit chunks.
- int32_t 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 revStride16Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 16, writing in 64-bit chunks.
- int32_t revStride16Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 16, writing in 128-bit chunks.
- int32_t revStride16Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 16, writing in 256-bit chunks.
- int32_t randomRead_Word32 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, reading in 32-bit chunks.
- int32_t randomRead_Word64 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, reading in 64-bit chunks.
- int32_t randomRead_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, reading in 128-bit chunks.
- int32_t randomRead_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, reading in 256-bit chunks.
- int32_t randomWrite_Word32 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, writing in 32-bit chunks.
- int32_t randomWrite_Word64 (void *start_address, void *end_address)

TODO. Walks over the allocated memory in random order, writing in 64-bit chunks.

- int32_t randomWrite_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, writing in 128-bit chunks.

- int32_t randomWrite_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, writing in 256-bit chunks.

5.3.1 Function Documentation

5.3.1.1 int32_t xmem::benchmark::benchmark_kernels::chasePointers (uintptr_t * *first_address*, uintptr_t ** *last_touched_address*, size_t *len*)

Walks over the allocated memory in random order by chasing pointers.

Parameters

num_pointers	The total number of pointer dereferences to make.

Returns

Undefined.

5.3.1.2 int32_t xmem::benchmark::benchmark_kernels::dummy_chasePointers (uintptr_t * , uintptr_t ** , size_t *len*)

Mimics the __chasePointers() method but doesn't do the memory accesses.

Returns

Undefined.

5.3.1.3 int32_t xmem::benchmark::benchmark_kernels::dummy_empty (void * , void *)

Does nothing. Used for measuring the time it takes just to call a benchmark routine via function pointer.

Returns

Undefined.

5.3.1.4 int32_t xmem::benchmark::benchmark_kernels::dummy_forwSequentialLoop_Word128 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward sequential Word 128 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.5 int32_t xmem::benchmark::benchmark_kernels::dummy_forwSequentialLoop_Word256 (void * *start_address,* void * *end_address*)

Used for measuring the time spent doing everything in forward sequential Word 256 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.6 int32_t xmem::benchmark::benchmark_kernels::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.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.7 int32_t xmem::benchmark::benchmark_kernels::dummy_forwSequentialLoop_Word64 (void * *start_address*, void * *end_address*)

Used for measuring the time spent doing everything in forward sequential Word 64 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.8 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride16Loop_Word128 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 16-strided Word 128 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.9 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride16Loop_Word256 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 16-strided Word 256 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.10 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride16Loop_Word32 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 16-strided Word 32 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.11 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride16Loop_Word64 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 16-strided Word 64 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.12 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride2Loop_Word128 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 2-strided Word 128 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.13 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride2Loop_Word256 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 2-strided Word 256 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.14 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride2Loop_Word32 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 2-strided Word 32 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.15 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride2Loop_Word64 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 2-strided Word 64 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.16 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride4Loop_Word128 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 4-strided Word 128 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.17 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride4Loop_Word256 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 4-strided Word 256 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.18 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride4Loop_Word32 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 4-strided Word 32 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.19 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride4Loop_Word64 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 4-strided Word 64 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.20 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride8Loop_Word128 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 8-strided Word 128 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.21 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride8Loop_Word256 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 8-strided Word 256 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.22 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride8Loop_Word32 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 8-strided Word 32 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.23 int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride8Loop_Word64 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in forward 8-strided Word 64 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.24 int32_t xmem::benchmark::benchmark_kernels::dummy_randomLoop_Word128 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in random Word 128 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.25 int32_t xmem::benchmark::benchmark_kernels::dummy_randomLoop_Word256 (void * *start_address,* void * *end_address*)

TODO. Used for measuring the time spent doing everything in random Word 256 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.26 int32_t xmem::benchmark::benchmark_kernels::dummy_randomLoop_Word32 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in random Word 32 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.27 int32_t xmem::benchmark::benchmark_kernels::dummy_randomLoop_Word64 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in random Word 64 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.28 int32_t xmem::benchmark::benchmark_kernels::dummy_revSequentialLoop_Word128 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse sequential Word 128 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.29 int32_t xmem::benchmark::benchmark_kernels::dummy_revSequentialLoop_Word256 (void * *start_address,* void * *end_address*)

Used for measuring the time spent doing everything in reverse sequential Word 256 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.30 int32_t xmem::benchmark::benchmark_kernels::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.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.31 int32_t xmem::benchmark::benchmark_kernels::dummy_revSequentialLoop_Word64 (void * *start_address*, void * *end_address*)

Used for measuring the time spent doing everything in reverse sequential Word 64 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.32 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride16Loop_Word128 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 128 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.33 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride16Loop_Word256 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 256 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.34 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride16Loop_Word32 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 32 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.35 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride16Loop_Word64 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 64 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.36 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride2Loop_Word128 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 128 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.37 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride2Loop_Word256 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 256 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.38 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride2Loop_Word32 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 32 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.39 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride2Loop_Word64 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 64 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.40 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride4Loop_Word128 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 128 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.41 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride4Loop_Word256 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 256 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.42 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride4Loop_Word32 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 32 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.43 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride4Loop_Word64 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 64 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.44 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride8Loop_Word128 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 128 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.45 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride8Loop_Word256 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 256 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.46 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride8Loop_Word32 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 32 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.47 int32_t xmem::benchmark::benchmark_kernels::dummy_revStride8Loop_Word64 (void * *start_address*, void * *end_address*)

TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 64 loops except for the memory access itself.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.48 int32_t xmem::benchmark::benchmark_kernels::forwSequentialRead_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory forward sequentially, reading in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.49 int32_t xmem::benchmark::benchmark_kernels::forwSequentialRead_Word256 (void * *start_address*, void * *end_address*)

Walks over the allocated memory forward sequentially, reading in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.50 int32_t xmem::benchmark::benchmark_kernels::forwSequentialRead_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory forward sequentially, reading in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.51 int32_t xmem::benchmark::benchmark_kernels::forwSequentialRead_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory forward sequentially, reading in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.52 int32_t xmem::benchmark::benchmark_kernels::forwSequentialWrite_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory forward sequentially, writing in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.53 int32_t xmem::benchmark::benchmark_kernels::forwSequentialWrite_Word256 (void * *start_address*, void * *end_address*)

Walks over the allocated memory forward sequentially, writing in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.54 int32_t xmem::benchmark::benchmark_kernels::forwSequentialWrite_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory forward sequentially, writing in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.55 int32_t xmem::benchmark::benchmark_kernels::forwSequentialWrite_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory forward sequentially, writing in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.56 int32_t xmem::benchmark::benchmark_kernels::forwStride16Read_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 16, reading in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.57 int32_t xmem::benchmark::benchmark_kernels::forwStride16Read_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 16, reading in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.58 int32_t xmem::benchmark::benchmark_kernels::forwStride16Read_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 16, reading in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.59 int32_t xmem::benchmark::benchmark_kernels::forwStride16Read_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 16, reading in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.60 int32_t xmem::benchmark::benchmark_kernels::forwStride16Write_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 16, writing in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.61 int32_t xmem::benchmark::benchmark_kernels::forwStride16Write_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 16, writing in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.62 int32_t xmem::benchmark::benchmark_kernels::forwStride16Write_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 16, writing in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.63 int32_t xmem::benchmark::benchmark_kernels::forwStride16Write_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 16, writing in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.64 int32_t xmem::benchmark::benchmark_kernels::forwStride2Read_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 2, reading in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.65 int32_t xmem::benchmark::benchmark_kernels::forwStride2Read_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 2, reading in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.66 int32_t xmem::benchmark::benchmark_kernels::forwStride2Read_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 2, reading in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.67 int32_t xmem::benchmark::benchmark_kernels::forwStride2Read_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 2, reading in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.68 int32_t xmem::benchmark::benchmark_kernels::forwStride2Write_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 2, writing in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.69 int32_t xmem::benchmark::benchmark_kernels::forwStride2Write_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 2, writing in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.70 int32_t xmem::benchmark::benchmark_kernels::forwStride2Write_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 2, writing in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.71 int32_t xmem::benchmark::benchmark_kernels::forwStride2Write_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 2, writing in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.72 int32_t xmem::benchmark::benchmark_kernels::forwStride4Read_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 4, reading in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.73 int32_t xmem::benchmark::benchmark_kernels::forwStride4Read_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 4, reading in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.74 int32_t xmem::benchmark::benchmark_kernels::forwStride4Read_Word32 (void * *start_address*, void * *end_address*)

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

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.75 int32_t xmem::benchmark::benchmark_kernels::forwStride4Read_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 4, reading in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.76 int32_t xmem::benchmark::benchmark_kernels::forwStride4Write_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 4, writing in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.77 int32_t xmem::benchmark::benchmark_kernels::forwStride4Write_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 4, writing in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.78 int32_t xmem::benchmark::benchmark_kernels::forwStride4Write_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 4, writing in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.79 int32_t xmem::benchmark::benchmark_kernels::forwStride4Write_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 4, writing in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.80 int32_t xmem::benchmark::benchmark_kernels::forwStride8Read_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 8, reading in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.81 int32_t xmem::benchmark::benchmark_kernels::forwStride8Read_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 8, reading in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.82 int32_t xmem::benchmark::benchmark_kernels::forwStride8Read_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 8, reading in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.83 int32_t xmem::benchmark::benchmark_kernels::forwStride8Read_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 8, reading in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.84 int32_t xmem::benchmark::benchmark_kernels::forwStride8Write_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 8, writing in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.85 int32_t xmem::benchmark::benchmark_kernels::forwStride8Write_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in forward strides of size 8, writing in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.86 int32_t xmem::benchmark::benchmark_kernels::forwStride8Write_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 8, writing in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

5.3.1.87 int32_t xmem::benchmark::benchmark_kernels::forwStride8Write_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in forward strides of size 8, writing in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.88 int32_t xmem::benchmark::benchmark_kernels::randomRead_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in random order, reading in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.89 int32_t xmem::benchmark::benchmark_kernels::randomRead_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in random order, reading in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.90 int32_t xmem::benchmark::benchmark_kernels::randomRead_Word32 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in random order, reading in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.91 int32_t xmem::benchmark::benchmark_kernels::randomRead_Word64 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in random order, reading in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.92 int32_t xmem::benchmark::benchmark_kernels::randomWrite_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in random order, writing in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

5.3.1.93 int32_t xmem::benchmark::benchmark_kernels::randomWrite_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in random order, writing in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.94 int32_t xmem::benchmark::benchmark_kernels::randomWrite_Word32 (void * *start_address,* void * *end_address*)

TODO. Walks over the allocated memory in random order, writing in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.95 int32_t xmem::benchmark::benchmark_kernels::randomWrite_Word64 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in random order, writing in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.96 int32_t xmem::benchmark::benchmark_kernels::revSequentialRead_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory reverse sequentially, reading in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.97 int32_t xmem::benchmark::benchmark_kernels::revSequentialRead_Word256 (void * *start_address*, void * *end_address*)

Walks over the allocated memory reverse sequentially, reading in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.98 int32_t xmem::benchmark::benchmark_kernels::revSequentialRead_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory reverse sequentially, reading in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.99 int32_t xmem::benchmark::benchmark_kernels::revSequentialRead_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory reverse sequentially, reading in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.100 int32_t xmem::benchmark::benchmark_kernels::revSequentialWrite_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory reverse sequentially, writing in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.101 int32_t xmem::benchmark::benchmark_kernels::revSequentialWrite_Word256 (void * *start_address*, void * *end_address*)

Walks over the allocated memory reverse sequentially, writing in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.102 int32_t xmem::benchmark::benchmark_kernels::revSequentialWrite_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory reverse sequentially, writing in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.103 int32_t xmem::benchmark::benchmark_kernels::revSequentialWrite_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory reverse sequentially, writing in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.104 int32_t xmem::benchmark::benchmark_kernels::revStride16Read_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 16, reading in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.105 int32_t xmem::benchmark::benchmark_kernels::revStride16Read_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 16, reading in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.106 int32_t xmem::benchmark::benchmark_kernels::revStride16Read_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 16, reading in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.107 int32_t xmem::benchmark::benchmark_kernels::revStride16Read_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 16, reading in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.108 int32_t xmem::benchmark::benchmark_kernels::revStride16Write_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 16, writing in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

5.3.1.109 int32_t xmem::benchmark::benchmark_kernels::revStride16Write_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 16, writing in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.110 int32_t xmem::benchmark::benchmark_kernels::revStride16Write_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 16, writing in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.111 int32_t xmem::benchmark::benchmark_kernels::revStride16Write_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 16, writing in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.112 int32_t xmem::benchmark::benchmark_kernels::revStride2Read_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 2, reading in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.113 int32_t xmem::benchmark::benchmark_kernels::revStride2Read_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 2, reading in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.114 int32_t xmem::benchmark::benchmark_kernels::revStride2Read_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 2, reading in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.115 int32_t xmem::benchmark::benchmark_kernels::revStride2Read_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 2, reading in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.116 int32_t xmem::benchmark::benchmark_kernels::revStride2Write_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 2, writing in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.117 int32_t xmem::benchmark::benchmark_kernels::revStride2Write_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 2, writing in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.118 int32_t xmem::benchmark::benchmark_kernels::revStride2Write_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 2, writing in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.119 int32_t xmem::benchmark::benchmark_kernels::revStride2Write_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 2, writing in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.120 int32_t xmem::benchmark::benchmark_kernels::revStride4Read_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 4, reading in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.121 int32_t xmem::benchmark::benchmark_kernels::revStride4Read_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 4, reading in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.122 int32_t xmem::benchmark::benchmark_kernels::revStride4Read_Word32 (void * *start_address*, void * *end_address*)

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

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.123 int32_t xmem::benchmark::benchmark_kernels::revStride4Read_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 4, reading in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.124 int32_t xmem::benchmark::benchmark_kernels::revStride4Write_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 4, writing in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.125 int32_t xmem::benchmark::benchmark_kernels::revStride4Write_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 4, writing in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.126 int32_t xmem::benchmark::benchmark_kernels::revStride4Write_Word32 (void * *start_address*, void * *end_address*)

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

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.127 int32_t xmem::benchmark::benchmark_kernels::revStride4Write_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 4, writing in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.128 int32_t xmem::benchmark::benchmark_kernels::revStride8Read_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 8, reading in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.129 int32_t xmem::benchmark::benchmark_kernels::revStride8Read_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 8, reading in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.130 int32_t xmem::benchmark::benchmark_kernels::revStride8Read_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 8, reading in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.131 int32_t xmem::benchmark::benchmark_kernels::revStride8Read_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 8, reading in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.132 int32_t xmem::benchmark::benchmark_kernels::revStride8Write_Word128 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 8, writing in 128-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.133 int32_t xmem::benchmark::benchmark_kernels::revStride8Write_Word256 (void * *start_address*, void * *end_address*)

TODO. Walks over the allocated memory in reverse strides of size 8, writing in 256-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.134 int32_t xmem::benchmark::benchmark_kernels::revStride8Write_Word32 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 8, writing in 32-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.3.1.135 int32_t xmem::benchmark::benchmark_kernels::revStride8Write_Word64 (void * *start_address*, void * *end_address*)

Walks over the allocated memory in reverse strides of size 8, writing in 64-bit chunks.

Parameters

start_address	The beginning of the memory region of interest.
end_address	The end of the memory region of interest.

Returns

Undefined.

5.4 xmem::common Namespace Reference

Namespaces

- win

Typedefs

- typedef uint32_t Word32_t

Enumerations

- enum pattern_mode_t { SEQUENTIAL, 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 batterns are broadly categorized by reads and writes.
- enum chunk_size_t { NUM_CHUNK_SIZES }

  Legal memory read/write chunk sizes in bits.

Functions

- void print_welcome_message ()

  Prints a basic welcome message to the console with useful information.
- void print_types_report ()

  Prints the various C/C++ types to the console for this machine.
- void print_compile_time_options ()

  Prints compile-time option information to the console.
- void test_timers ()

  Tests any enabled timers and outputs results to the console for sanity checking.

- void 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 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 unlock_thread_to_numa_node ()

  Clears the affinity of the calling thread to any given NUMA node.
- bool lock_thread_to_cpu (uint32_t cpu_id)

  Sets the affinity of the calling thread to a given logical CPU.
- bool unlock_thread_to_cpu ()

  Clears the affinity of the calling thread to any given logical CPU.
- int32_t 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 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 config_page_size ()

  Queries the page sizes from the system and sets relevant global variables.

Variables

- size_t g_page_size
- size_t g_large_page_size
- uint32_t g_num_nodes
- uint32_t g_num_logical_cpus
- uint32_t g_num_physical_packages
- uint32_t g_starting_test_index
- uint32_t g_test_index

5.4.1 Typedef Documentation

5.4.1.1 typedef uint32_t xmem::common::Word32_t

5.4.2 Enumeration Type Documentation

5.4.2.1 enum xmem::common::chunk_size_t

Legal memory read/write chunk sizes in bits.

Enumerator

NUM_CHUNK_SIZES

5.4.2.2 enum xmem::common::pattern_mode_t

Memory access patterns are broadly categorized by sequential or random-access.

Enumerator

SEQUENTIAL

NUM_PATTERN_MODES

5.4.2.3 enum xmem::common::rw_mode_t

Memory access batterns are broadly categorized by reads and writes.

Enumerator

READ

WRITE

NUM_RW_MODES

5.4.3 Function Documentation

5.4.3.1 size_t xmem::common::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.

Parameters

working_set_size_KB	The working set size of the memory in KB.

Returns

The number of passes to use.

5.4.3.2 bool xmem::common::config_page_size ( )

Queries the page sizes from the system and sets relevant global variables.

Returns

False if the default value has to be used because the appropriate values could not be queried successfully from the OS.

5.4.3.3 int32_t xmem::common::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.

Parameters

numa_node	The NUMA node of interest.
cpu_in_node	The Nth logical CPU in the node.

Returns

The Nth logical CPU ID in the specified NUMA node. If none is found, returns -1.

5.4.3.4 bool xmem::common::lock_thread_to_cpu (uint32_t *cpu_id*)

Sets the affinity of the calling thread to a given logical CPU.

Parameters

cpu_id	The logical CPU identifier to lock the thread to.

Returns

True on success.

5.4.3.5 bool xmem::common::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.

Parameters

numa_node	The NUMA node number to select a CPU from.

Returns

True on success.

5.4.3.6 void xmem::common::print_compile_time_options ( )

Prints compile-time option information to the console.

5.4.3.7 void xmem::common::print_types_report ( )

Prints the various C/C++ types to the console for this machine.

5.4.3.8 void xmem::common::print_welcome_message ( )

Prints a basic welcome message to the console with useful information.

5.4.3.9 void xmem::common::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.

5.4.3.10 void xmem::common::test_timers ( )

Tests any enabled timers and outputs results to the console for sanity checking.

5.4.3.11 bool xmem::common::unlock_thread_to_cpu ( )

Clears the affinity of the calling thread to any given logical CPU.

Returns

True on success.

5.4.3.12 bool xmem::common::unlock_thread_to_numa_node ( )

Clears the affinity of the calling thread to any given NUMA node.

Returns

True on success.

5.4.4 Variable Documentation

5.4.4.1 size_t xmem::common::g_large_page_size

Large page size on the system, in bytes.

5.4.4.2 uint32_t xmem::common::g_num_logical_cpus

Number of logical CPU cores in the system.

5.4.4.3 uint32_t xmem::common::g_num_nodes

Number of NUMA nodes in the system.

5.4.4.4 uint32_t xmem::common::g_num_physical_packages

Number of physical CPU packages in the system. Generally this is the same as number of NUMA nodes, unless UMA emulation is done in hardware.

5.4.4.5 size_t xmem::common::g_page_size

Default page size on the system, in bytes.

5.4.4.6 uint32_t xmem::common::g_starting_test_index

Numeric identifier for the first benchmark test.

5.4.4.7 uint32_t xmem::common::g_test_index

Numeric identifier for the current benchmark test.

5.5 xmem::common::win Namespace Reference

Namespaces

- third_party

5.6 xmem::common::win::third_party Namespace Reference

Classes

- class CPdhQuery

A third-party class for querying performance counter data from Windows.

5.7 xmem::config Namespace Reference

Namespaces

- third_party

Classes

- class Configurator

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

Enumerations

- enum optionIndex {
  UNKNOWN, HELP, MEAS_LATENCY, MEAS_THROUGHPUT,
  WORKING_SET_SIZE, ITERATIONS, BASE_TEST_INDEX, OUTPUT_FILE }

  Enumerates all possible types of command-line options.

Variables

- const third_party::Descriptor usage []

  Command-line option descriptors as needed by stuff in <config/third_party/optionparser.h>. This is basically the help message content.

5.7.1 Enumeration Type Documentation

5.7.1.1 enum xmem::config::optionIndex

Enumerates all possible types of command-line options.

Enumerator

UNKNOWN

HELP

MEAS_LATENCY

MEAS_THROUGHPUT

WORKING_SET_SIZE

ITERATIONS

BASE_TEST_INDEX

OUTPUT_FILE

5.7.2 Variable Documentation

5.7.2.1 const third_party::Descriptor xmem::config::usage[]

Initial value:

```
= {
            { UNKNOWN, 0, "", "", third_party::Arg::None, "USAGE: xmem [options]\n\n"
            "Options:" },
            { HELP, 0, "h", "help", third_party::Arg::None, "    -h, --help    \tPrint usage and exit."
       },
            { MEAS_LATENCY, 0, "l", "latency", third_party::Arg::None, "    -l, --latency    \t
      Measure memory latency" },
            { MEAS_THROUGHPUT, 0, "t", "throughput", third_party::Arg::None, "    -t,
       --throughput    \tMeasure memory throughput" },
            { WORKING_SET_SIZE, 0, "w", "working_set_size",
      third_party::MyArg::PositiveInteger, "    -w, --working_set_size    \tWorking set size in KB. This must be a multiple of
            { ITERATIONS, 0, "n", "iterations", third_party::MyArg::PositiveInteger, "    -n,
       --iterations    \tIterations per benchmark test" },
            { BASE_TEST_INDEX, 0, "i", "base_test_index",
      third_party::MyArg::NonnegativeInteger, "    -i, --base_test_index    \tNumerical index of the first benchmark, for trac
       },
            { OUTPUT_FILE, 0, "f", "output_file", third_party::MyArg::Required, "    -f,
       --output_file    \tOutput filename to use. If not specified, no output file generated." },
            { UNKNOWN, 0, "", "", third_party::Arg::None, "\nExamples:\n"
            "    xmem --help\n"
            "    xmem -h\n"
            "    xmem -t\n"
            "    xmem -t --latency -n10 -w=524288 -f results.csv -i 101\n"
            },
            { 0, 0, 0, 0, 0, 0 }
        }
```

Command-line option descriptors as needed by stuff in <config/third_party/optionparser.h>. This is basically the help message content.

5.8 xmem::config::third_party Namespace Reference

Classes

- struct Arg

  Functions for checking the validity of option arguments.
- struct Descriptor

  Describes an option, its help text (usage) and how it should be parsed.
- class ExampleArg
- class MyArg
- class Option

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

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

  Determines the minimum lengths of the buffer and options arrays used for Parser.

Typedefs

- typedef ArgStatus(∗ 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 { ARG_NONE, ARG_OK, ARG_IGNORE, ARG_ILLEGAL }

  Possible results when checking if an argument is valid for a certain option.

Functions

- template<typename OStream >
  void 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 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 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 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 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)

5.8.1 Typedef Documentation

5.8.1.1 typedef ArgStatus(* xmem::config::third_party::CheckArg)(const Option &option, bool msg)

Signature of functions that check if an argument is valid for a certain type of option.

Every Option has such a function assigned in its Descriptor.

```
Descriptor usage[] = { {UNKNOWN, 0, "", "", Arg::None, ""}, ... };
```

A CheckArg function has the following signature:

```
ArgStatus CheckArg(const Option& option, bool msg);
```

It is used to check if a potential argument would be acceptable for the option. It will even be called if there is no argument. In that case `option.arg` will be `NULL`.

If `msg` is `true` and the function determines that an argument is not acceptable and that this is a fatal error, it should output a message to the user before returning ARG_ILLEGAL. If `msg` is `false` the function should remain silent (or you will get duplicate messages).

See ArgStatus for the meaning of the return values.

While you can provide your own functions, often the following pre-defined checks (which never return ARG_ILLEGAL) will suffice:

- `Arg::None` For options that don't take an argument: Returns ARG_NONE.
- `Arg::Optional` Returns ARG_OK if the argument is attached and ARG_IGNORE otherwise.

5.8.2 Enumeration Type Documentation

5.8.2.1 enum xmem::config::third_party::ArgStatus

Possible results when checking if an argument is valid for a certain option.

In the case that no argument is provided for an option that takes an optional argument, return codes `ARG_OK` and `ARG_IGNORE` are equivalent.

Enumerator

ARG_NONE The option does not take an argument.

ARG_OK The argument is acceptable for the option.

ARG_IGNORE The argument is not acceptable but that's non-fatal because the option's argument is optional.

ARG_ILLEGAL The argument is not acceptable and that's fatal.

5.8.3 Function Documentation

5.8.3.1 template<typename OStream > void xmem::config::third_party::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.

printUsage() takes the `help` texts of a Descriptor[] array and formats them into a usage message, wrapping lines to achieve the desired output width.

Table formatting:

Aside from plain strings which are simply line-wrapped, the usage may contain tables. Tables are used to align elements in the output.

```
// Without a table. The explanatory texts are not aligned.
-c, --create  |Creates something.
-k, --kill  |Destroys something.

// With table formatting. The explanatory texts are aligned.
-c, --create  |Creates something.
-k, --kill    |Destroys something.
```

Table formatting removes the need to pad help texts manually with spaces to achieve alignment. To create a table, simply insert \t (tab) characters to separate the cells within a row.

```
const option::Descriptor usage[] = {
{..., "-c, --create  \tCreates something." },
{..., "-k, --kill  \tDestroys something." }, ...
```

Note that you must include the minimum amount of space desired between cells yourself. Table formatting will insert further spaces as needed to achieve alignment.

You can insert line breaks within cells by using \v (vertical tab).

```
const option::Descriptor usage[] = {
{..., "-c,\v--create  \tCreates\vsomething." },
{..., "-k,\v--kill  \tDestroys\vsomething." }, ...

// results in

-c,       Creates
--create  something.
-k,       Destroys
--kill    something.
```

You can mix lines that do not use \t or \v with those that do. The plain lines will not mess up the table layout. Alignment of the table columns will be maintained even across these interjections.

```
const option::Descriptor usage[] = {
{..., "-c, --create  \tCreates something." },
{..., "---------------------------------" },
{..., "-k, --kill  \tDestroys something." }, ...

// results in

-c, --create  Creates something.
---------------------------------
-k, --kill    Destroys something.
```

You can have multiple tables within the same usage whose columns are aligned independently. Simply insert a dummy Descriptor with `help==0`.

```
const option::Descriptor usage[] = {
{..., "Long options:" },
{..., "--very-long-option  \tDoes something long." },
{..., "--ultra-super-mega-long-option  \tTakes forever to complete." },
{..., 0 }, // ---------- table break -----------
{..., "Short options:" },
{..., "-s  \tShort." },
{..., "-q  \tQuick." }, ...

// results in

Long options:
--very-long-option              Does something long.
--ultra-super-mega-long-option  Takes forever to complete.
Short options:
-s  Short.
-q  Quick.

// Without the table break it would be

Long options:
--very-long-option              Does something long.
--ultra-super-mega-long-option  Takes forever to complete.
Short options:
-s                              Short.
-q                              Quick.
```

Output methods:

Because TheLeanMeanC++Option parser is freestanding, you have to provide the means for output in the first argument(s) to printUsage(). Because printUsage() is implemented as a set of template functions, you have great flexibility in your choice of output method. The following example demonstrates typical uses. Anything that's similar enough will work.

```
#include <unistd.h>  // write()
#include <iostream>  // cout
#include <sstream>   // ostringstream
#include <cstdio>    // fwrite()
using namespace std;

void my_write(const char* str, int size) {
  fwrite(str, size, 1, stdout);
}

struct MyWriter {
  void write(const char* buf, size_t size) const {
     fwrite(str, size, 1, stdout);
  }
};

struct MyWriteFunctor {
  void operator()(const char* buf, size_t size) {
     fwrite(str, size, 1, stdout);
  }
};
...
printUsage(my_write, usage);    // custom write function
printUsage(MyWriter(), usage);  // temporary of a custom class
MyWriter writer;
printUsage(writer, usage);      // custom class object
MyWriteFunctor wfunctor;
printUsage(&wfunctor, usage);   // custom functor
printUsage(write, 1, usage);    // write() to file descriptor 1
printUsage(cout, usage);        // an ostream&
printUsage(fwrite, stdout, usage);  // fwrite() to stdout
ostringstream sstr;
printUsage(sstr, usage);        // an ostringstream&
```

Notes:

- the `write()` method of a class that is to be passed as a temporary as `MyWriter()` is in the example, must be a `const` method, because temporary objects are passed as const reference. This only applies to temporary objects that are created and destroyed in the same statement. If you create an object like `writer` in the example, this restriction does not apply.
- a functor like `MyWriteFunctor` in the example must be passed as a pointer. This differs from the way functors are passed to e.g. the STL algorithms.

- All printUsage() templates are tiny wrappers around a shared non-template implementation. So there's no penalty for using different versions in the same program.
- printUsage() always interprets Descriptor::help as UTF-8 and always produces UTF-8-encoded output. If your system uses a different charset, you must do your own conversion. You may also need to change the font of the console to see non-ASCII characters properly. This is particularly true for Windows.
- **Security warning:** Do not insert untrusted strings (such as user-supplied arguments) into the usage. printUsage() has no protection against malicious UTF-8 sequences.

Parameters

prn	The output method to use. See the examples above.
usage	the Descriptor[] array whose `help` texts will be formatted.
width	the maximum number of characters per output line. Note that this number is in actual characters, not bytes. printUsage() supports UTF-8 in `help` and will count multi-byte UTF-8 sequences properly. Asian wide characters are counted as 2 characters.
last_column_min_percent	(0-100) The minimum percentage of `width` that should be available for the last column (which typically contains the textual explanation of an option). If less space is available, the last column will be printed on its own line, indented according to `last_column_own_line_max_percent`.
last_column_own_line_max_percent	(0-100) If the last column is printed on its own line due to less than `last_column_min_percent` of the width being available, then only `last_column_own_line_max_percent` of the extra line(s) will be used for the last column's text. This ensures an indentation. See example below.

```
// width=20, last_column_min_percent=50 (i.e. last col. min. width=10)
--3456789 1234567890
          1234567890

// width=20, last_column_min_percent=75 (i.e. last col. min. width=15)
// last_column_own_line_max_percent=75
--3456789
     123456789012345
     67890

// width=20, last_column_min_percent=75 (i.e. last col. min. width=15)
// last_column_own_line_max_percent=33 (i.e. max. 5)
--3456789
               12345
               67890
               12345
               67890
```

5.8.3.2 template<typename Function > void xmem::config::third_party::printUsage (Function * *prn*, const Descriptor *usage[]*, int *width* = 80, int *last_column_min_percent* = 50, int *last_column_own_line_max_percent* = 75)

5.8.3.3 template<typename Temporary > void xmem::config::third_party::printUsage (const Temporary & *prn*, const Descriptor *usage[]*, int *width* = 80, int *last_column_min_percent* = 50, int *last_column_own_line_max_percent* = 75)

5.8.3.4 template<typename Syscall > void xmem::config::third_party::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)

5.8.3.5 template<typename Function , typename Stream > void xmem::config::third_party::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)

5.9 xmem::power Namespace Reference

Classes

- class NativeDRAMPowerReader

A class for measuring socket-level DRAM power from the OS performance counter interface.

- class PowerReader

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

5.10 xmem::thread Namespace Reference

Classes

- class Runnable

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

- class Thread

  a nice wrapped thread interface independent of particular OS API

5.11 xmem::timers Namespace Reference

Namespaces

- win

Classes

- class Timer

  This class abstracts a simple high resolution stopwatch timer. WARNING: these objects are NOT thread safe.

5.12 xmem::timers::win Namespace Reference

Classes

- class QPCTimer

  This class implements a simple high resolution stopwatch timer based on Windows' QueryPerformanceCounter API. WARNING: these objects are NOT thread safe.

Functions

- uint64_t get_qpc_time ()

  Query the QPC timer.

5.12.1 Function Documentation

5.12.1.1 uint64_t xmem::timers::win::get_qpc_time ( )

Query the QPC timer.

Returns

The current QPC timer tick.

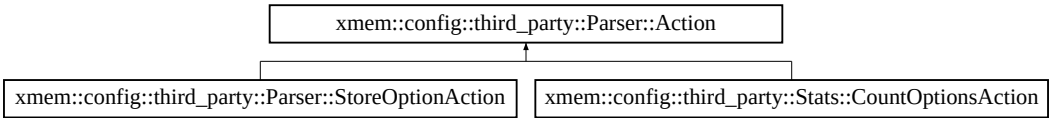
Chapter 6

Class Documentation

6.1 xmem::config::third_party::Parser::Action Struct Reference

`#include <optionparser.h>`

Inheritance diagram for xmem::config::third_party::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.

6.1.1 Member Function Documentation

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

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

Parameters

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

Returns

`false` iff a fatal error has occurred.

Reimplemented in xmem::config::third_party::Parser::StoreOptionAction.

6.1.1.2 virtual bool xmem::config::third_party::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 occured and the parse should be aborted.

Reimplemented in xmem::config::third_party::Parser::StoreOptionAction, and xmem::config::third_party::Stats::↩ CountOptionsAction.

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

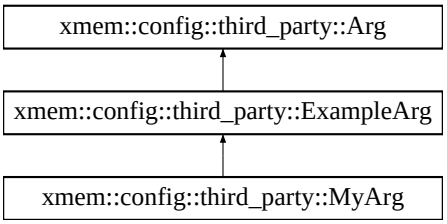
- src/config/third_party/optionparser.h

6.2 xmem::config::third_party::Arg Struct Reference

Functions for checking the validity of option arguments.

`#include <optionparser.h>`

Inheritance diagram for xmem::config::third_party::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.

6.2.1 Detailed Description

Functions for checking the validity of option arguments.

Every Option has such a function assigned in its Descriptor.

```
Descriptor usage[] = { {UNKNOWN, 0, "", "", Arg::None, ""}, ... };
```

A CheckArg function has the following signature:

```
ArgStatus CheckArg(const Option& option, bool msg);
```

It is used to check if a potential argument would be acceptable for the option. It will even be called if there is no argument. In that case `option.arg` will be `NULL`.

If `msg` is `true` and the function determines that an argument is not acceptable and that this is a fatal error, it should output a message to the user before returning ARG_ILLEGAL. If `msg` is `false` the function should remain silent (or you will get duplicate messages).

See ArgStatus for the meaning of the return values.

While you can provide your own functions, often the following pre-defined checks (which never return ARG_ILLEGAL) will suffice:

- `Arg::None` For options that don't take an argument: Returns ARG_NONE.
- `Arg::Optional` Returns ARG_OK if the argument is attached and ARG_IGNORE otherwise.

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;
  }
};
```

6.2.2 Member Function Documentation

6.2.2.1 static ArgStatus xmem::config::third_party::Arg::None (const Option & , bool) [inline], [static]

For options that don't take an argument: Returns ARG_NONE.

6.2.2.2 static ArgStatus xmem::config::third_party::Arg::Optional (const Option & *option,* bool) [inline], [static]

Returns ARG_OK if the argument is attached and ARG_IGNORE otherwise.

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

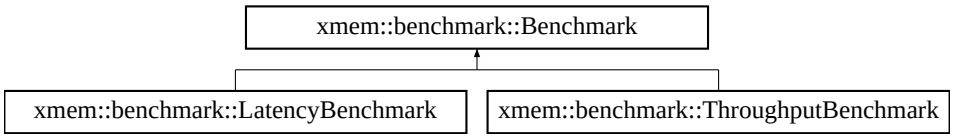
- src/config/third_party/optionparser.h

6.3 xmem::benchmark::Benchmark Class Reference

Flexible abstract class for any memory benchmark.

`#include <Benchmark.h>`

Inheritance diagram for xmem::benchmark::Benchmark:



Public Member Functions

- Benchmark (void *mem_array, size_t len, uint32_t iterations, xmem::common::chunk_size_t chunk_size, uint32_t cpu_node, uint32_t mem_node, uint32_t num_worker_threads, std::string name, xmem::timers::Timer *timer, std::vector< xmem::power::PowerReader * > dram_power_readers)

  Constructor.
- ~Benchmark ()

  Destructor.
- virtual bool run ()=0

  Runs the benchmark.
- virtual void report_benchmark_info ()=0

  Reports benchmark configuration details to the console.
- virtual void report_results ()=0

  Reports results to the console.
- void report_power_results ()

  Reports power measurement results to the console.
- bool isValid ()

  Checks to see that the object is in a valid state.
- bool hasRun ()

  Checks to see if the benchmark has run.
- double getMetricOnIter (uint32_t iter)

  Extracts the metric of interest for a given iteration. Units are interpreted by the inheriting class.
- double getAverageMetric ()

  Gets the average benchmark metric across all iterations.
- double getAverageDRAMPower (uint32_t socket_id)

  Gets the average DRAM power over the benchmark.
- double getPeakDRAMPower (uint32_t socket_id)

  Gets the peak DRAM power over the benchmark.
- size_t getLen ()

  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 ()

  Gets the number of iterations for this benchmark.
- xmem::common::chunk_size_t getChunkSize ()

  Gets the width of memory access used in this benchmark.
- uint32_t getCPUNode ()

  Gets the CPU NUMA node used in this benchmark.
- uint32_t getMemNode ()

Gets the memory NUMA node used in this benchmark.

- uint32_t getNumThreads ()

  Gets the number of worker threads used in this benchmark.

- std::string getName ()

  Gets the human-friendly name of this benchmark.

Protected Member Functions

- 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
- xmem::common::chunk_size_t _chunk_size
- size_t * _indices
- uint32_t _cpu_node
- uint32_t _mem_node
- uint32_t _num_worker_threads
- xmem::timers::Timer * _timer
- std::vector < xmem::power::PowerReader * > _dram_power_readers
- std::vector < xmem::thread::Thread * > _dram_power_threads
- std::vector< double > _average_dram_power_socket
- std::vector< double > _peak_dram_power_socket
- bool _hasRun
- std::vector< double > _metricOnIter
- double _averageMetric
- std::string _name
- bool _obj_valid
- bool _warning

6.3.1 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.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 Benchmark::Benchmark (void * *mem_array*, size_t *len*, uint32_t *iterations*, xmem::common::chunk_size_t *chunk_size*, uint32_t *cpu_node*, uint32_t *mem_node*, uint32_t *num_worker_threads*, std::string *name*, xmem::timers::Timer * *timer*, std::vector< xmem::power::PowerReader * > *dram_power_readers*)

Constructor.

Parameters

mem_array	a pointer to a contiguous chunk of memory that has been allocated for benchmarking among the worker threads. This should be aligned to a 256-bit boundary and should be the working set size times number of threads large.
len	Length of the raw_mem_array in bytes. This should be a multiple of 4 KB pages.
iterations	Number of iterations to do of the complete benchmark, to average out results.
passes_per_↩ iteration	Number of passes to do in each iteration, to ensure timed section of code is "long enough".
chunk_size	encoded size of an individual memory access.
cpu_node	the logical CPU NUMA node to use for the benchmark
mem_node	the logical memory NUMA node used in the benchmark
num_worker_↩ threads	number of worker threads to use in the benchmark
name	name of the benchmark to use when reporting
timer	pointer to an existing Timer
dram_power_↩ readers	vector of pointers to PowerReader objects for measuring DRAM power

6.3.2.2 Benchmark::~Benchmark ( )

Destructor.

6.3.3 Member Function Documentation

6.3.3.1 bool Benchmark::_start_power_threads ( ) [protected]

Starts the DRAM power measurement threads.

Returns

true on success

6.3.3.2 bool Benchmark::_stop_power_threads ( ) [protected]

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

Returns

true on success

6.3.3.3 double Benchmark::getAverageDRAMPower (uint32_t *socket_id*)

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).

6.3.3.4 double Benchmark::getAverageMetric ( )

Gets the average benchmark metric across all iterations.

Returns

The average metric.

6.3.3.5 xmem::common::chunk_size_t Benchmark::getChunkSize ( )

Gets the width of memory access used in this benchmark.

Returns

The chunk size for this benchmark.

6.3.3.6 uint32_t Benchmark::getCPUNode ( )

Gets the CPU NUMA node used in this benchmark.

Returns

The NUMA CPU node used in this benchmark.

6.3.3.7 uint32_t Benchmark::getIterations ( )

Gets the number of iterations for this benchmark.

Returns

The number of iterations for this benchmark.

6.3.3.8 size_t Benchmark::getLen ( )

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.

6.3.3.9 uint32_t Benchmark::getMemNode ( )

Gets the memory NUMA node used in this benchmark.

Returns

The NUMA memory node used in this benchmark.

6.3.3.10 double Benchmark::getMetricOnIter (uint32_t *iter*)

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

Parameters

iter	Iteration to extract.

Returns

The metric on the iteration specified by the input.

6.3.3.11 std::string Benchmark::getName ( )

Gets the human-friendly name of this benchmark.

Returns

The benchmark test name.

6.3.3.12 uint32_t Benchmark::getNumThreads ( )

Gets the number of worker threads used in this benchmark.

Returns

The number of worker threads used in this benchmark.

6.3.3.13 double Benchmark::getPeakDRAMPower (uint32_t *socket_id*)

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).

6.3.3.14 bool Benchmark::hasRun ( )

Checks to see if the benchmark has run.

Returns

True if run() has already completed successfully.

6.3.3.15 bool Benchmark::isValid ( )

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

Returns

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

6.3.3.16 virtual void xmem::benchmark::Benchmark::report_benchmark_info ( ) [pure virtual]

Reports benchmark configuration details to the console.

Implemented in xmem::benchmark::ThroughputBenchmark, and xmem::benchmark::LatencyBenchmark.

6.3.3.17 void Benchmark::report_power_results ( )

Reports power measurement results to the console.

6.3.3.18 virtual void xmem::benchmark::Benchmark::report_results ( ) [pure virtual]

Reports results to the console.

Implemented in xmem::benchmark::ThroughputBenchmark, and xmem::benchmark::LatencyBenchmark.

6.3.3.19 virtual bool xmem::benchmark::Benchmark::run ( ) [pure virtual]

Runs the benchmark.

Returns

true on benchmark success

Implemented in xmem::benchmark::ThroughputBenchmark, and xmem::benchmark::LatencyBenchmark.

6.3.4 Member Data Documentation

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

The average DRAM power in this benchmark, per socket.

6.3.4.2 double xmem::benchmark::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.

6.3.4.3 xmem::common::chunk_size_t xmem::benchmark::Benchmark::_chunk_size [protected]

Chunk size of memory accesses in this benchmark. TODO: Move this to ThroughputBenchmark.h, as it does not apply in all situations, e.g. in LatencyBenchmark.

6.3.4.4 uint32_t xmem::benchmark::Benchmark::_cpu_node [protected]

The CPU NUMA node used in this benchmark.

6.3.4.5 std::vector<xmem::power::PowerReader*> xmem::benchmark::Benchmark::_dram_power_readers [protected]

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

6.3.4.6 std::vector<xmem::thread::Thread*> xmem::benchmark::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.

6.3.4.7 bool xmem::benchmark::Benchmark::_hasRun [protected]

Indicates whether the benchmark has run.

6.3.4.8 size_t* xmem::benchmark::Benchmark::_indices [protected]

Pointer to a list of indices. This is for indirect memory addressing. Currently unused. TODO: Remove this entirely?

6.3.4.9 uint32_t xmem::benchmark::Benchmark::_iterations [protected]

Number of iterations used in this benchmark.

6.3.4.10 size_t xmem::benchmark::Benchmark::_len [protected]

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

6.3.4.11 void* xmem::benchmark::Benchmark::_mem_array [protected]

Pointer to the memory region to use in this benchmark.

6.3.4.12 uint32_t xmem::benchmark::Benchmark::_mem_node [protected]

The memory NUMA node used in this benchmark.

6.3.4.13 std::vector<double> xmem::benchmark::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.

6.3.4.14 std::string xmem::benchmark::Benchmark::_name [protected]

Name of this benchmark.

6.3.4.15 uint32_t xmem::benchmark::Benchmark::_num_worker_threads [protected]

The number of worker threads used in this benchmark.

6.3.4.16 bool xmem::benchmark::Benchmark::_obj_valid [protected]

Indicates whether this benchmark object is valid.

6.3.4.17 std::vector<double> xmem::benchmark::Benchmark::_peak_dram_power_socket [protected]

The peak DRAM power in this benchmark, per socket.

6.3.4.18 xmem::timers::Timer* xmem::benchmark::Benchmark::_timer [protected]

The reference timer for this benchmark. TODO: Remove this. It isn't thread safe anyway, so workers don't use it.

6.3.4.19 bool xmem::benchmark::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/benchmark/Benchmark.h
- src/benchmark/Benchmark.cpp

6.4 xmem::benchmark::BenchmarkManager Class Reference

Manages running all benchmarks at a high level.

`#include <BenchmarkManager.h>`

Public Member Functions

- BenchmarkManager (size_t working_set_size, uint32_t iterations_per_benchmark, bool output_to_file, std::string results_filename)

  Constructor.
- ~BenchmarkManager ()

  Destructor.
- bool runAll ()

  Runs all possible benchmark configurations.
- bool runThroughputBenchmarks ()

  Runs the throughput benchmarks.
- bool runLatencyBenchmarks ()

  Runs the latency benchmark.

6.4.1 Detailed Description

Manages running all benchmarks at a high level.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 BenchmarkManager::BenchmarkManager (size_t *working_set_size*, uint32_t *iterations_per_benchmark*, bool *output_to_file*, std::string *results_filename*)

Constructor.

Parameters

working_set_size	Total memory to test in bytes on each NUMA node. The BenchmarkManager will try to allocate them by itself.
iterations_per_benchmark	Number of passes to run for each individual benchmark.
output_to_file	If true, write to file specified by results_filename.
results_filename	Filename to write results to if output_to_file is true.

6.4.2.2 BenchmarkManager::~BenchmarkManager ( )

Destructor.

6.4.3 Member Function Documentation

6.4.3.1 bool BenchmarkManager::runAll ( )

Runs all possible benchmark configurations.

Returns

True on success.

6.4.3.2 bool BenchmarkManager::runLatencyBenchmarks ( )

Runs the latency benchmark.

Returns

True on benchmarking success.

6.4.3.3 bool BenchmarkManager::runThroughputBenchmarks ( )

Runs the throughput benchmarks.

Returns

True on benchmarking success.

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

- src/benchmark/BenchmarkManager.h
- src/benchmark/BenchmarkManager.cpp

6.5 xmem::common::win::third_party::CPdhQuery::CException Class Reference

`#include <win_CPdhQuery.h>`

Public Member Functions

- CException (std::tstring const &errorMsg)
- std::tstring What () const

6.5.1 Constructor & Destructor Documentation

6.5.1.1 xmem::common::win::third_party::CPdhQuery::CException::CException (std::tstring const & *errorMsg*) [inline]

6.5.2 Member Function Documentation

6.5.2.1 std::tstring xmem::common::win::third_party::CPdhQuery::CException::What ( ) const [inline]

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

- src/common/win/third_party/win_CPdhQuery.h

6.6 xmem::config::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.
- Configurator (bool runLatency, bool runThroughput, size_t working_set_size, uint32_t iterations_per_test, std::string filename, bool use_output_file)

  Specialized constructor for when you don't want to get config from input, and you want to pass it in directly.
- int configureFromInput (int argc, char *argv[])

  Configures the tool based on user's command-line inputs.
- bool latencyTestSelected ()

  Indicates if the latency test has been selected.
- bool throughputTestSelected ()

  Indicates if the throughput test has been selected.
- size_t getWorkingSetSize ()

  Gets the working set size in bytes for each worker thread, if applicable.
- uint32_t getIterationsPerTest ()

  Gets the number of iterations that should be run of each benchmark.
- std::string getOutputFilename ()

  Gets the output filename to use, if applicable.
- bool useOutputFile ()

  Determines whether to generate an output CSV file.

6.6.1 Detailed Description

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

6.6.2 Constructor & Destructor Documentation

6.6.2.1 Configurator::Configurator ( )

Default constructor. A default configuration is set. You will want to run configureFromInput() most likely.

6.6.2.2 Configurator::Configurator (bool *runLatency*, bool *runThroughput*, size_t *working_set_size*, uint32_t *iterations_per_test*, std::string *filename*, bool *use_output_file*)

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

Parameters

runLatency	Indicates latency benchmarks should be run.
runThroughput	Indicates throughput benchmarks should be run.
working_set_size	The total size of memory to test in all benchmarks, in bytes. This MUST be a multiple of 4KB pages.

iterations_per_↩ test	For each unique benchmark test, this is the number of times to repeat it.
filename	Output filename to use.
use_output_file	If true, use the provided output filename.

6.6.3 Member Function Documentation

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

Configures the tool based on user's command-line inputs.

Parameters

argc	The argc from main().
argv	The argv from main().

Returns

0 on success.

6.6.3.2 uint32_t xmem::config::Configurator::getIterationsPerTest ( ) [inline]

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

Returns

The iterations for each test.

6.6.3.3 std::string xmem::config::Configurator::getOutputFilename ( ) [inline]

Gets the output filename to use, if applicable.

Returns

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

6.6.3.4 size_t xmem::config::Configurator::getWorkingSetSize ( ) [inline]

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

Returns

The working set size in bytes.

6.6.3.5 bool xmem::config::Configurator::latencyTestSelected ( ) [inline]

Indicates if the latency test has been selected.

Returns

True if the latency test has been selected to run.

6.6.3.6 bool xmem::config::Configurator::throughputTestSelected ( ) [inline]

Indicates if the throughput test has been selected.

Returns

True if the throughput test has been selected to run.

6.6.3.7 bool xmem::config::Configurator::useOutputFile ( ) [inline]

Determines whether to generate an output CSV file.

Returns

True if an output file should be used.

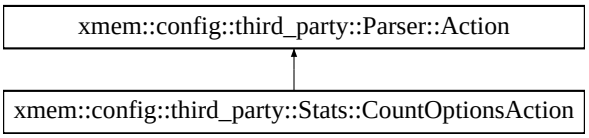
The documentation for this class was generated from the following files:

- src/config/Configurator.h
- src/config/Configurator.cpp

6.7 xmem::config::third_party::Stats::CountOptionsAction Class Reference

`#include <optionparser.h>`

Inheritance diagram for xmem::config::third_party::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.

6.7.1 Constructor & Destructor Documentation

6.7.1.1 xmem::config::third_party::Stats::CountOptionsAction::CountOptionsAction (unsigned * *buffer_max_*) [inline]

Creates a new CountOptionsAction that will increase `*buffer_max_` for each parsed Option.

6.7.2 Member Function Documentation

6.7.2.1 bool xmem::config::third_party::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 occured and the parse should be aborted.

Reimplemented from xmem::config::third_party::Parser::Action.

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

- src/config/third_party/optionparser.h

6.8 xmem::common::win::third_party::CPdhQuery Class Reference

A third-party class for querying performance counter data from Windows.

`#include <win_CPdhQuery.h>`

Classes

- class CException

Public Member Functions

- CPdhQuery (std::tstring const &counterPath)

  Constructor.
- ~CPdhQuery ()

  Destructor. The counter and query handle will be closed.
- std::map< std::tstring, double > CollectQueryData ()

  Collect all the data since the last sampling period.

6.8.1 Detailed Description

A third-party class for querying performance counter data from Windows.

Source: `http://askldjd.wordpress.com/2011/01/05/a-pdh-helper-class-cpdhquery/`, retrieved September 2014. Lightly modified and wrapped code to work here, like namespaces and some comments.

Author

Alan Ning

6.8.2 Constructor & Destructor Documentation

6.8.2.1 xmem::common::win::third_party::CPdhQuery::CPdhQuery (std::tstring const & *counterPath*) [inline], [explicit]

Constructor.

6.8.2.2 xmem::common::win::third_party::CPdhQuery::~CPdhQuery ( ) [inline]

Destructor. The counter and query handle will be closed.

6.8.3 Member Function Documentation

6.8.3.1 std::map<std::tstring, double> xmem::common::win::third_party::CPdhQuery::CollectQueryData ( ) [inline]

Collect all the data since the last sampling period.

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

- src/common/win/third_party/win_CPdhQuery.h

6.9 xmem::config::third_party::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.

6.9.1 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
  }
  , ...
};
```

6.9.2 Member Data Documentation

6.9.2.1 const CheckArg xmem::config::third_party::Descriptor::check_arg

For each option that matches shortopt 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.

6.9.2.2 const char* xmem::config::third_party::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 shortopt 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.

6.9.2.3 const unsigned xmem::config::third_party::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.

6.9.2.4 const char* const xmem::config::third_party::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 shortopt 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 shortopt 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.

6.9.2.5 const char* const xmem::config::third_party::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.

6.9.2.6 const int xmem::config::third_party::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.

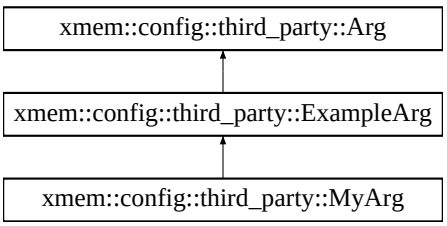
The documentation for this struct was generated from the following file:

- src/config/third_party/optionparser.h

6.10 xmem::config::third_party::ExampleArg Class Reference

`#include <ExampleArg.h>`

Inheritance diagram for xmem::config::third_party::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)

6.10.1 Member Function Documentation

6.10.1.1 static ArgStatus xmem::config::third_party::ExampleArg::NonEmpty (const Option & *option,* bool *msg*) [inline], [static]

6.10.1.2 static void xmem::config::third_party::ExampleArg::printError (const char * *msg1,* const Option & *opt,* const char * *msg2*) [inline], [static]

6.10.1.3 static ArgStatus xmem::config::third_party::ExampleArg::Required (const Option & *option,* bool *msg*) [inline], [static]

6.10.1.4 static ArgStatus xmem::config::third_party::ExampleArg::Unknown (const Option & *option,* bool *msg*) [inline], [static]

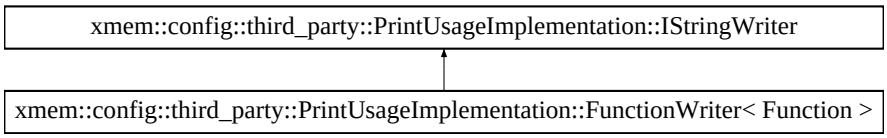
The documentation for this class was generated from the following file:

- src/config/third_party/ExampleArg.h

6.11 xmem::config::third_party::PrintUsageImplementation::FunctionWriter< Function > Struct Template Reference

`#include <optionparser.h>`

Inheritance diagram for xmem::config::third_party::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

6.11.1 Constructor & Destructor Documentation

6.11.1.1 template<typename Function> xmem::config::third_party::PrintUsageImplementation::FunctionWriter< Function >::FunctionWriter (Function * *w*) [inline]

6.11.2 Member Function Documentation

6.11.2.1 template<typename Function> virtual void xmem::config::third_party::PrintUsageImplementation::FunctionWriter< Function >::operator() (const char * , int) [inline], [virtual]

Writes the given number of chars beginning at the given pointer somewhere.

Reimplemented from xmem::config::third_party::PrintUsageImplementation::IStringWriter.

6.11.3 Member Data Documentation

6.11.3.1 template<typename Function> Function* xmem::config::third_party::PrintUsageImplementation::FunctionWriter< Function >::write

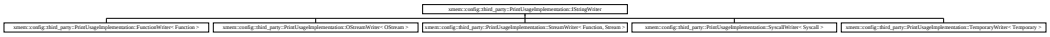
The documentation for this struct was generated from the following file:

- src/config/third_party/optionparser.h

6.12 xmem::config::third_party::PrintUsageImplementation::IStringWriter Struct Reference

`#include <optionparser.h>`

Inheritance diagram for xmem::config::third_party::PrintUsageImplementation::IStringWriter:



Public Member Functions

- virtual void operator() (const char *, int)

  Writes the given number of chars beginning at the given pointer somewhere.

6.12.1 Member Function Documentation

6.12.1.1 virtual void xmem::config::third_party::PrintUsageImplementation::IStringWriter::operator() (const char * , int) [inline], [virtual]

Writes the given number of chars beginning at the given pointer somewhere.

Reimplemented in xmem::config::third_party::PrintUsageImplementation::StreamWriter< Function, Stream >, xmem::config::third_party::PrintUsageImplementation::SyscallWriter< Syscall >, xmem::config::third_party::PrintUsageImplementation::TemporaryWriter< Temporary >, xmem::config::third_party::PrintUsageImplementation::OStreamWriter< OStream >, and xmem::config::third_party::PrintUsageImplementation::FunctionWriter< Function >.

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

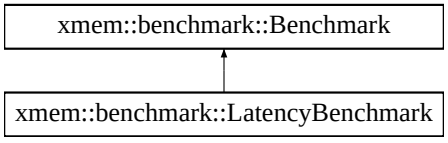
- src/config/third_party/optionparser.h

6.13 xmem::benchmark::LatencyBenchmark Class Reference

A type of benchmark that measures memory latency via random pointer chasing. TODO: loaded latency tests.

`#include <LatencyBenchmark.h>`

Inheritance diagram for xmem::benchmark::LatencyBenchmark:



Public Types

- typedef int32_t(* LatencyBenchFunction)(uintptr_t *, uintptr_t **, size_t)

Public Member Functions

- LatencyBenchmark (void *mem_array, size_t len, uint32_t iterations, uint32_t cpu_node, uint32_t mem_node, uint32_t num_worker_threads, std::string name, xmem::timers::Timer *timer, std::vector< xmem::power::↩ PowerReader * > dram_power_readers)
- virtual bool run ()

  Runs the benchmark.
- virtual void report_benchmark_info ()

  Outputs the benchmark configuration to the console.
- virtual void report_results ()

  Outputs a report of the benchmark results to the console if run() returned true.

Additional Inherited Members

6.13.1 Detailed Description

A type of benchmark that measures memory latency via random pointer chasing. TODO: loaded latency tests.

6.13.2 Member Typedef Documentation

6.13.2.1 typedef int32_t(* xmem::benchmark::LatencyBenchmark::LatencyBenchFunction)(uintptr_t *, uintptr_t **, size_t)

6.13.3 Constructor & Destructor Documentation

6.13.3.1 LatencyBenchmark::LatencyBenchmark (void * *mem_array*, size_t *len*, uint32_t *iterations*, uint32_t *cpu_node*, uint32_t *mem_node*, uint32_t *num_worker_threads*, std::string *name*, **xmem::timers::Timer** * *timer*, std::vector< **xmem::power::PowerReader** * > *dram_power_readers*)

Constructor.

Parameters

mem_array	a pointer to a contiguous chunk of memory that has been allocated for benchmarking among the worker threads. This should be aligned to a 256-bit boundary and should be the working set size times number of threads large.
len	Length of the raw_mem_array in bytes. This should be a multiple of 4 KB pages.
iterations	Number of iterations (passes) to do of the complete benchmark.
cpu_node	the logical CPU NUMA node to use in the benchmark
mem_node	the logical memory NUMA node used in the benchmark
num_worker_↩ threads	number of worker threads to use in the benchmark
name	name of the benchmark to use when reporting
timer	pointer to an existing Timer object
dram_power_↩ readers	vector of pointers to PowerReader objects for measuring DRAM power

6.13.4 Member Function Documentation

6.13.4.1 void LatencyBenchmark::report_benchmark_info ( ) [virtual]

Outputs the benchmark configuration to the console.

Implements xmem::benchmark::Benchmark.

6.13.4.2 void LatencyBenchmark::report_results ( ) [virtual]

Outputs a report of the benchmark results to the console if run() returned true.

Implements xmem::benchmark::Benchmark.

6.13.4.3 bool LatencyBenchmark::run ( ) [virtual]

Runs the benchmark.

Returns

True on success.

Implements xmem::benchmark::Benchmark.

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

- src/benchmark/LatencyBenchmark.h
- src/benchmark/LatencyBenchmark.cpp

6.14 xmem::config::third_party::PrintUsageImplementation::LinePartIterator Class Reference

`#include <optionparser.h>`

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.

6.14.1 Constructor & Destructor Documentation

6.14.1.1 xmem::config::third_party::PrintUsageImplementation::LinePartIterator::LinePartIterator (const Descriptor *usage[]*) [inline]

Creates an iterator for `usage`.

6.14.2 Member Function Documentation

6.14.2.1 int xmem::config::third_party::PrintUsageImplementation::LinePartIterator::column ( ) [inline]

Returns the index (counting from 0) of the column in which the part pointed to by data() is located.

6.14.2.2 const char* xmem::config::third_party::PrintUsageImplementation::LinePartIterator::data ( ) [inline]

Returns the current part of the iteration.

6.14.2.3 int xmem::config::third_party::PrintUsageImplementation::LinePartIterator::length ( ) [inline]

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

6.14.2.4 int xmem::config::third_party::PrintUsageImplementation::LinePartIterator::line ( ) [inline]

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

6.14.2.5 bool xmem::config::third_party::PrintUsageImplementation::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

false	if moving to next part failed because no further part exists.

See LinePartIterator for details about the iteration.

6.14.2.6 bool xmem::config::third_party::PrintUsageImplementation::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

false	if moving to next row failed because no further row exists.

6.14.2.7 bool xmem::config::third_party::PrintUsageImplementation::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

false	if moving to next table failed because no further table exists.

6.14.2.8 void xmem::config::third_party::PrintUsageImplementation::LinePartIterator::restartRow ( ) [inline]

Reset iteration to the beginning of the current row.

6.14.2.9 void xmem::config::third_party::PrintUsageImplementation::LinePartIterator::restartTable ( ) [inline]

Reset iteration to the beginning of the current table.

6.14.2.10 int xmem::config::third_party::PrintUsageImplementation::LinePartIterator::screenLength ( ) [inline]

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

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

- src/config/third_party/optionparser.h

6.15 xmem::config::third_party::PrintUsageImplementation::LineWrapper Class Reference

`#include <optionparser.h>`

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.).

6.15.1 Constructor & Destructor Documentation

6.15.1.1 xmem::config::third_party::PrintUsageImplementation::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.

6.15.2 Member Function Documentation

6.15.2.1 void xmem::config::third_party::PrintUsageImplementation::LineWrapper::flush (IStringWriter & *write*) [inline]

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).

6.15.2.2 void xmem::config::third_party::PrintUsageImplementation::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

write	where to write the data.
data	the new chunk of data to write.
len	the length of the chunk of data to write.

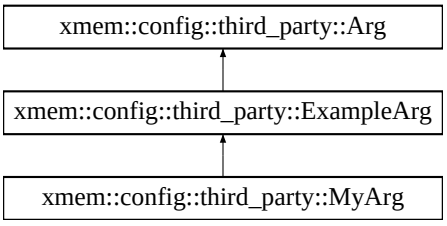
The documentation for this class was generated from the following file:

- src/config/third_party/optionparser.h

6.16 xmem::config::third_party::MyArg Class Reference

`#include <MyArg.h>`

Inheritance diagram for xmem::config::third_party::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.

6.16.1 Member Function Documentation

6.16.1.1 static ArgStatus xmem::config::third_party::MyArg::Integer (const Option & *option,* bool *msg*) [inline], [static]

Checks an option that it is an integer.

6.16.1.2 static ArgStatus xmem::config::third_party::MyArg::NonnegativeInteger (const Option & *option,* bool *msg*) [inline], [static]

Checks an option that it is a nonnegative integer.

6.16.1.3 static ArgStatus xmem::config::third_party::MyArg::PositiveInteger (const Option & *option,* bool *msg*) [inline], [static]

Checks an option that it is a positive integer.

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

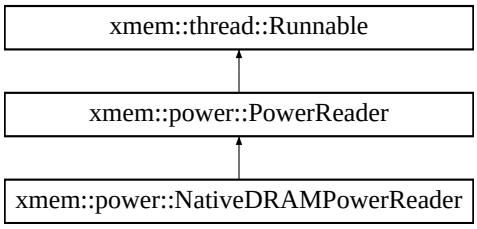
- src/config/third_party/MyArg.h

6.17 xmem::power::NativeDRAMPowerReader Class Reference

A class for measuring socket-level DRAM power from the OS performance counter interface.

`#include <NativeDRAMPowerReader.h>`

Inheritance diagram for xmem::power::NativeDRAMPowerReader:



Public Member Functions

- NativeDRAMPowerReader (uint32_t counter_cpu_index, double sampling_period, double power_units, std::string name, int32_t cpu_affinity)

  Constructor.
- ~NativeDRAMPowerReader ()

  Destructor.
- virtual void run ()

  Starts measuring power at the rate implied by the sampling_period passed in the constructor. Terminates when stop() is called.

Additional Inherited Members

6.17.1 Detailed Description

A class for measuring socket-level DRAM power from the OS performance counter interface.

6.17.2 Constructor & Destructor Documentation

6.17.2.1 NativeDRAMPowerReader::NativeDRAMPowerReader (uint32_t *counter_cpu_index,* double *sampling_period,* double *power_units,* std::string *name,* int32_t *cpu_affinity*)

Constructor.

Parameters

counter_cpu_index	Which CPU's DRAM power counter to sample. A single hardware counter might be shared across different CPUs.
sampling_period	The time between power samples in seconds.
power_units	The power units for each sample in watts.
cpu_affinity	The CPU affinity for this object's run() method for any thread that calls it. If negative, no affinity preference.

6.17.2.2 NativeDRAMPowerReader::~NativeDRAMPowerReader ( )

Destructor.

6.17.3 Member Function Documentation

6.17.3.1 void NativeDRAMPowerReader::run ( ) [virtual]

Starts measuring power at the rate implied by the sampling_period passed in the constructor. Terminates when stop() is called.

Implements xmem::power::PowerReader.

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

- src/power/NativeDRAMPowerReader.h
- src/power/NativeDRAMPowerReader.cpp

6.18 xmem::config::third_party::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.

6.18.1 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; ...
```

6.18.2 Constructor & Destructor Documentation

6.18.2.1 xmem::config::third_party::Option::Option ( ) [inline]

Creates a new Option that is a one-element linked list and has NULL desc, name, arg and namelen.

6.18.2.2 xmem::config::third_party::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.

6.18.2.3 xmem::config::third_party::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.

6.18.3 Member Function Documentation

6.18.3.1 void xmem::config::third_party::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.

6.18.3.2 int xmem::config::third_party::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.

6.18.3.3 Option* xmem::config::third_party::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.

6.18.3.4 int xmem::config::third_party::Option::index ( ) const [inline]

Returns Descriptor::index of this Option's Descriptor, or -1 if this Option is invalid (unused).

6.18.3.5 bool xmem::config::third_party::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.

6.18.3.6 bool xmem::config::third_party::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.

6.18.3.7 Option* xmem::config::third_party::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.

6.18.3.8 Option* xmem::config::third_party::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.

6.18.3.9 Option* xmem::config::third_party::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.

6.18.3.10 xmem::config::third_party::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; ...
```

6.18.3.11 xmem::config::third_party::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; ...
```

6.18.3.12 void xmem::config::third_party::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.

6.18.3.13 Option* xmem::config::third_party::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.

6.18.3.14 Option* xmem::config::third_party::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.

6.18.3.15 int xmem::config::third_party::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 !
}
```

6.18.4 Member Data Documentation

6.18.4.1 const char* xmem::config::third_party::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.

6.18.4.2 const Descriptor* xmem::config::third_party::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*()` .

6.18.4.3 const char* xmem::config::third_party::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.

6.18.4.4 int xmem::config::third_party::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.

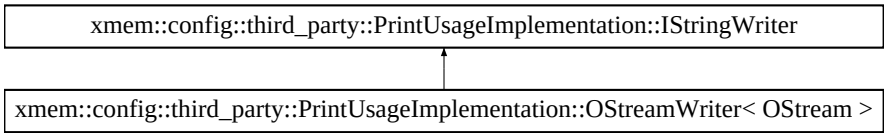
The documentation for this class was generated from the following file:

- src/config/third_party/optionparser.h

6.19 xmem::config::third_party::PrintUsageImplementation::OStreamWriter< OStream > Struct Template Reference

`#include <optionparser.h>`

Inheritance diagram for xmem::config::third_party::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

6.19.1 Constructor & Destructor Documentation

6.19.1.1 template<typename OStream> xmem::config::third_party::PrintUsageImplementation::OStreamWriter< OStream >::OStreamWriter (OStream & *o*) [inline]

6.19.2 Member Function Documentation

6.19.2.1 template<typename OStream> virtual void xmem::config::third_party::PrintUsageImplementation::OStreamWriter< OStream >::operator() (const char * , int) [inline], [virtual]

Writes the given number of chars beginning at the given pointer somewhere.

Reimplemented from xmem::config::third_party::PrintUsageImplementation::IStringWriter.

6.19.3 Member Data Documentation

6.19.3.1 template<typename OStream> OStream& xmem::config::third_party::PrintUsageImplementation::OStreamWriter< OStream >::ostream

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

- src/config/third_party/optionparser.h

6.20 xmem::config::third_party::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

6.20.1 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])
  ...
```

6.20.2 Constructor & Destructor Documentation

6.20.2.1 xmem::config::third_party::Parser::Parser ( ) [inline]

Creates a new Parser.

6.20.2.2 xmem::config::third_party::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

gnu	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 `gnu` flag will have no further effect on this argument vector. So it is enough to pass `gnu==true` when creating Stats.
usage	Array of Descriptor objects that describe the options to support. The last entry of this array must have 0 in all fields.
argc	The number of elements from `argv` that are to be parsed. If you pass -1, the number will be determined automatically. In that case the `argv` list must end with a NULL pointer.
argv	The arguments to be parsed. If you pass -1 as `argc` the last pointer in the `argv` list must be NULL to mark the end.
options	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 `usage` *PLUS* ONE.
buffer	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 `bufmax` to limit the number of options stored.

min_abbr_len	Passing a value `min_abbr_len > 0` enables abbreviated long options. The parser will match a prefix of a long option as if it was the full long option (e.g. `-foob=10` will be interpreted as if it was `-foobar=10`), as long as the prefix has at least `min_abbr_len` characters (not counting the `-`) and is unambiguous. Be careful if combining `min_abbr_len=1` with `single_minus_longopt=true` because the ambiguity check does not consider short options and abbreviated single minus long options will take precedence over short options.
single_minus_longopt	Passing `true` 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. `-file` would be interpreted as `-file` and not as `-f -i -l -e` (assuming a long option named "`file`" exists).
bufmax	The greatest index in the `buffer`[] array that parse() will write to is `bufmax-1`. 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 `bufmax` 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()].

6.20.2.3 xmem::config::third_party::Parser::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`) `[inline]`

Parser(...) with non-const argv.

6.20.2.4 xmem::config::third_party::Parser::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`) `[inline]`

POSIX Parser(...) (gnu==false).

6.20.2.5 xmem::config::third_party::Parser::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`) `[inline]`

POSIX Parser(...) (gnu==false) with non-const argv.

6.20.3 Member Function Documentation

6.20.3.1 bool xmem::config::third_party::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);
```

6.20.3.2 const char* xmem::config::third_party::Parser::nonOption (int *i*) [inline]

Returns **nonOptions()[i]** (*without* checking if i is in range!).

6.20.3.3 const char** xmem::config::third_party::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().

6.20.3.4 int xmem::config::third_party::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.

6.20.3.5 int xmem::config::third_party::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).

6.20.3.6 void xmem::config::third_party::Parser::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`) [inline]

Parses the given argument vector.

Parameters

gnu	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 `gnu` flag will have no further effect on this argument vector. So it is enough to pass `gnu==true` when creating Stats.
usage	Array of Descriptor objects that describe the options to support. The last entry of this array must have 0 in all fields.
argc	The number of elements from `argv` that are to be parsed. If you pass -1, the number will be determined automatically. In that case the `argv` list must end with a NULL pointer.
argv	The arguments to be parsed. If you pass -1 as `argc` the last pointer in the `argv` list must be NULL to mark the end.
options	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 `usage` *PLUS* ONE.
buffer	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 `bufmax` to limit the number of options stored.
min_abbr_len	Passing a value `min_abbr_len > 0` enables abbreviated long options. The parser will match a prefix of a long option as if it was the full long option (e.g. `-foob=10` will be interpreted as if it was `-foobar=10`), as long as the prefix has at least `min_abbr_len` characters (not counting the `-`) and is unambiguous. Be careful if combining `min_abbr_len=1` with `single_minus_longopt=true` because the ambiguity check does not consider short options and abbreviated single minus long options will take precedence over short options.
single_minus_longopt	Passing `true` 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. `-file` would be interpreted as `-file` and not as `-f -i -l -e` (assuming a long option named "`file`" exists).
bufmax	The greatest index in the `buffer`[] array that parse() will write to is `bufmax-1`. 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 `bufmax` 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()].

6.20.3.7 void xmem::config::third_party::Parser::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`) `[inline]`

parse() with non-const argv.

6.20.3.8 void xmem::config::third_party::Parser::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) [inline]

POSIX parse() (gnu==false).

6.20.3.9 void xmem::config::third_party::Parser::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) [inline]

POSIX parse() (gnu==false) with non-const argv.

6.20.4 Friends And Related Function Documentation

6.20.4.1 friend struct Stats [friend]

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

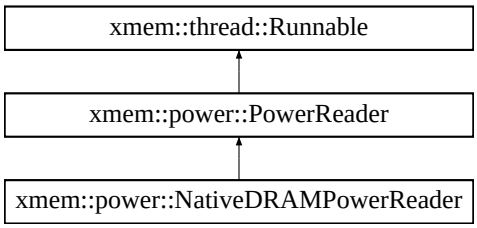
- src/config/third_party/optionparser.h

6.21 xmem::power::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::power::PowerReader:



Public Member Functions

- PowerReader (double 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.
- double 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

- double _sampling_period
- double _power_units
- std::string _name
- bool _stop_signal
- std::vector< double > _power_trace
- double _average_power
- double _peak_power
- size_t _num_samples
- int32_t _cpu_affinity

Additional Inherited Members

6.21.1 Detailed Description

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

6.21.2 Constructor & Destructor Documentation

6.21.2.1 PowerReader::PowerReader (double *sampling_period*, double *power_units*, std::string *name*, int32_t *cpu_affinity*)

Constructor.

Parameters

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

6.21.2.2 PowerReader::~PowerReader ( )

Destructor.

6.21.3 Member Function Documentation

6.21.3.1 bool PowerReader::calculateMetrics ( )

Calculates the relevant metrics.

Returns

True on success.

6.21.3.2 bool PowerReader::clear ( )

Clears the stored power data.

Returns

True on success.

6.21.3.3 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.

6.21.3.4 double PowerReader::getAveragePower ( )

Gets the average power.

Returns

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

6.21.3.5 double PowerReader::getLastSample ( )

Gets the last sample.

Returns

The last power sample measured.

6.21.3.6 size_t PowerReader::getNumSamples ( )

Gets the number of samples collected.

Returns

Number of samples collected.

6.21.3.7 double PowerReader::getPeakPower ( )

Gets the peak power.

Returns

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

6.21.3.8 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.

6.21.3.9 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.

6.21.3.10 double PowerReader::getSamplingPeriod ( )

Gets the sampling period.

Returns

The sampling period of the measurements in seconds.

6.21.3.11 std::string PowerReader::name ( )

Gets the name of this object.

Returns

The human-friendly name of this PowerReader.

6.21.3.12 virtual void xmem::power::PowerReader::run ( ) [pure virtual]

Starts measuring power at the rate implied by the sampling_period passed in the constructor. Call stop() to indicate to stop measuring.

Implements xmem::thread::Runnable.

Implemented in xmem::power::NativeDRAMPowerReader.

6.21.3.13 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.

6.21.4 Member Data Documentation

6.21.4.1 double xmem::power::PowerReader::_average_power [protected]

The average power.

6.21.4.2 int32_t xmem::power::PowerReader::_cpu_affinity [protected]

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

6.21.4.3 std::string xmem::power::PowerReader::_name [protected]

Name of this object.

6.21.4.4 size_t xmem::power::PowerReader::_num_samples [protected]

The number of samples collected.

6.21.4.5 double xmem::power::PowerReader::_peak_power [protected]

The peak power observed.

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

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

6.21.4.7 double xmem::power::PowerReader::_power_units [protected]

Power units in watts.

6.21.4.8 double xmem::power::PowerReader::_sampling_period [protected]

Power sampling period in seconds.

6.21.4.9 bool xmem::power::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/power/PowerReader.h
- src/power/PowerReader.cpp

6.22 xmem::config::third_party::PrintUsageImplementation Struct Reference

`#include <optionparser.h>`

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)

6.22.1 Member Function Documentation

6.22.1.1 static void xmem::config::third_party::PrintUsageImplementation::indent (IStringWriter & *write,* int & *x,* int *want_x*) [inline], [static]

6.22.1.2 static bool xmem::config::third_party::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......
```

6.22.1.3 static void xmem::config::third_party::PrintUsageImplementation::printUsage (IStringWriter & *write,* const Descriptor *usage[],* int *width* = 80, int *last_column_min_percent* = 50, int *last_column_own_line_max_percent* = 75) [inline], [static]

6.22.1.4 static void xmem::config::third_party::PrintUsageImplementation::upmax (int & *i1,* int *i2*) [inline], [static]

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

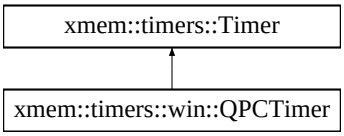
- src/config/third_party/optionparser.h

6.23 xmem::timers::win::QPCTimer Class Reference

This class implements a simple high resolution stopwatch timer based on Windows' QueryPerformanceCounter API. WARNING: these objects are NOT thread safe.

`#include <QPCTimer.h>`

Inheritance diagram for xmem::timers::win::QPCTimer:



Public Member Functions

- QPCTimer ()

  Constructor. This may take a noticeable amount of time.
- virtual void start ()

  Starts the timer.
- virtual uint64_t stop ()

  Stops the timer.

Additional Inherited Members

6.23.1 Detailed Description

This class implements a simple high resolution stopwatch timer based on Windows' QueryPerformanceCounter API. WARNING: these objects are NOT thread safe.

6.23.2 Constructor & Destructor Documentation

6.23.2.1 QPCTimer::QPCTimer ( )

Constructor. This may take a noticeable amount of time.

6.23.3 Member Function Documentation

6.23.3.1 void QPCTimer::start ( ) [virtual]

Starts the timer.

Implements xmem::timers::Timer.

6.23.3.2 uint64_t QPCTimer::stop ( ) [virtual]

Stops the timer.

Returns

Elapsed time since last start() call in ticks.

Implements xmem::timers::Timer.

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

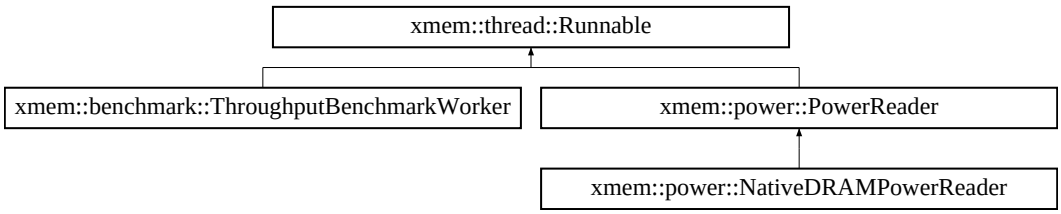
- src/timers/win/QPCTimer.h
- src/timers/win/QPCTimer.cpp

6.24 xmem::thread::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::thread::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.

6.24.1 Detailed Description

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

6.24.2 Constructor & Destructor Documentation

6.24.2.1 Runnable::Runnable ( )

Constructor.

6.24.2.2 Runnable::~Runnable ( )

Destructor.

6.24.3 Member Function Documentation

6.24.3.1 bool Runnable::_acquireLock (int32_t *timeout*) [protected]

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

Parameters

timeout	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.

6.24.3.2 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).

6.24.3.3 virtual void xmem::thread::Runnable::run ( ) [pure virtual]

Does some "work". Pure virtual method that any derived class must implement in a thread-safe manner.

Implemented in xmem::benchmark::ThroughputBenchmarkWorker, xmem::power::PowerReader, and xmem::power::NativeDRAMPowerReader.

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

- src/thread/Runnable.h
- src/thread/Runnable.cpp

6.25 xmem::config::third_party::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.

6.25.1 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.

6.25.2 Constructor & Destructor Documentation

6.25.2.1 xmem::config::third_party::Stats::Stats ( ) [inline]

Creates a Stats object with counts set to 1 (for the sentinel element).

6.25.2.2 xmem::config::third_party::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.

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

Stats(...) with non-const argv.

6.25.2.4 xmem::config::third_party::Stats::Stats (const Descriptor *usage[]*, int *argc*, const char ** *argv*, int *min_abbr_len* = 0, bool *single_minus_longopt* = false) [inline]

POSIX Stats(...) (gnu==false).

6.25.2.5 xmem::config::third_party::Stats::Stats (const Descriptor *usage[]*, int *argc*, char ** *argv*, int *min_abbr_len* = 0, bool *single_minus_longopt* = false) [inline]

POSIX Stats(...) (gnu==false) with non-const argv.

6.25.3 Member Function Documentation

6.25.3.1 void xmem::config::third_party::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.

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

add() with non-const argv.

6.25.3.3 void xmem::config::third_party::Stats::add (const Descriptor *usage[]*, int *argc*, const char ** *argv*, int *min_abbr_len* = 0, bool *single_minus_longopt* = false) [inline]

POSIX add() (gnu==false).

6.25.3.4 void xmem::config::third_party::Stats::add (const Descriptor *usage[]*, int *argc*, char ** *argv*, int *min_abbr_len* = 0, bool *single_minus_longopt* = false) [inline]

POSIX add() (gnu==false) with non-const argv.

6.25.4 Member Data Documentation

6.25.4.1 unsigned xmem::config::third_party::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.

6.25.4.2 unsigned xmem::config::third_party::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.

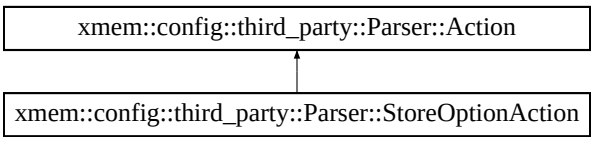
The documentation for this struct was generated from the following file:

- src/config/third_party/optionparser.h

6.26 xmem::config::third_party::Parser::StoreOptionAction Class Reference

`#include <optionparser.h>`

Inheritance diagram for xmem::config::third_party::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.

6.26.1 Constructor & Destructor Documentation

6.26.1.1 xmem::config::third_party::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

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

6.26.2 Member Function Documentation

6.26.2.1 bool xmem::config::third_party::Parser::StoreOptionAction::finished (int *numargs*, const char ** *args*) [inline], [virtual]

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

Parameters

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

Returns

`false` iff a fatal error has occurred.

Reimplemented from xmem::config::third_party::Parser::Action.

6.26.2.2 bool xmem::config::third_party::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 occured and the parse should be aborted.

Reimplemented from xmem::config::third_party::Parser::Action.

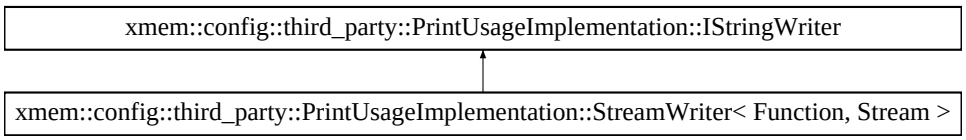
The documentation for this class was generated from the following file:

- src/config/third_party/optionparser.h

6.27 xmem::config::third_party::PrintUsageImplementation::StreamWriter< Function, Stream > Struct Template Reference

`#include <optionparser.h>`

Inheritance diagram for xmem::config::third_party::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

6.27.1 Constructor & Destructor Documentation

6.27.1.1 template<typename Function, typename Stream> xmem::config::third_party::PrintUsageImplementation::StreamWriter< Function, Stream >::StreamWriter (Function * w, Stream * s) [inline]

6.27.2 Member Function Documentation

6.27.2.1 template<typename Function, typename Stream> virtual void xmem::config::third_party::PrintUsageImplementation::StreamWriter< Function, Stream >::operator() (const char * , int) [inline], [virtual]

Writes the given number of chars beginning at the given pointer somewhere.

Reimplemented from xmem::config::third_party::PrintUsageImplementation::IStringWriter.

6.27.3 Member Data Documentation

6.27.3.1 template<typename Function, typename Stream> Function* xmem::config::third_party::PrintUsageImplementation::StreamWriter< Function, Stream >::fwrite

6.27.3.2 template<typename Function, typename Stream> Stream* xmem::config::third_party::PrintUsageImplementation::StreamWriter< Function, Stream >::stream

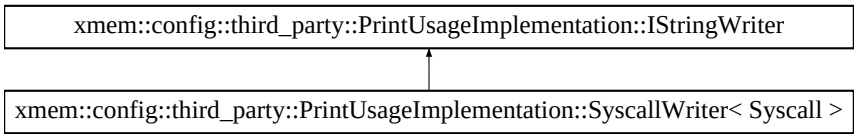
The documentation for this struct was generated from the following file:

- src/config/third_party/optionparser.h

6.28 xmem::config::third_party::PrintUsageImplementation::SyscallWriter< Syscall > Struct Template Reference

`#include <optionparser.h>`

Inheritance diagram for xmem::config::third_party::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

6.28.1 Constructor & Destructor Documentation

6.28.1.1 template<typename Syscall> xmem::config::third_party::PrintUsageImplementation::SyscallWriter< Syscall >::SyscallWriter (Syscall * *w*, int *f*) [inline]

6.28.2 Member Function Documentation

6.28.2.1 template<typename Syscall> virtual void xmem::config::third_party::PrintUsageImplementation::SyscallWriter< Syscall >::operator() (const char * , int) [inline], [virtual]

Writes the given number of chars beginning at the given pointer somewhere.

Reimplemented from xmem::config::third_party::PrintUsageImplementation::IStringWriter.

6.28.3 Member Data Documentation

6.28.3.1 template<typename Syscall> int xmem::config::third_party::PrintUsageImplementation::SyscallWriter< Syscall >::fd

6.28.3.2 template<typename Syscall> Syscall* xmem::config::third_party::PrintUsageImplementation::SyscallWriter< Syscall >::write

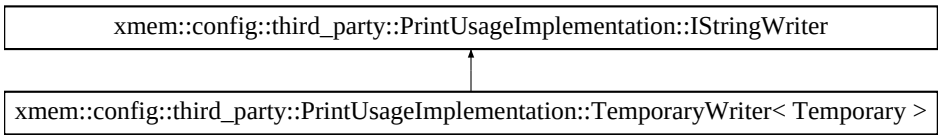
The documentation for this struct was generated from the following file:

- src/config/third_party/optionparser.h

6.29 xmem::config::third_party::PrintUsageImplementation::TemporaryWriter< Temporary > Struct Template Reference

`#include <optionparser.h>`

Inheritance diagram for xmem::config::third_party::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

6.29.1 Constructor & Destructor Documentation

6.29.1.1 template<typename Temporary> xmem::config::third_party::PrintUsageImplementation::TemporaryWriter< Temporary >::TemporaryWriter (const Temporary & *u*) [inline]

6.29.2 Member Function Documentation

6.29.2.1 template<typename Temporary> virtual void xmem::config::third_party::PrintUsageImplementation::TemporaryWriter< Temporary >::operator() (const char *, int) [inline], [virtual]

Writes the given number of chars beginning at the given pointer somewhere.

Reimplemented from xmem::config::third_party::PrintUsageImplementation::IStringWriter.

6.29.3 Member Data Documentation

6.29.3.1 template<typename Temporary> const Temporary& xmem::config::third_party::PrintUsageImplementation::TemporaryWriter< Temporary >::userstream

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

- src/config/third_party/optionparser.h

6.30 xmem::thread::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 ()
- bool start ()
- bool create_and_start ()
- bool join (int32_t timeout)
- bool cancel ()
- int32_t getExitCode ()
- bool started ()
- bool completed ()
- bool validTarget ()
- bool created ()
- bool isThreadSuspended ()
- bool isThreadRunning ()
- Runnable * getTarget ()

6.30.1 Detailed Description

a nice wrapped thread interface independent of particular OS API

6.30.2 Constructor & Destructor Documentation

6.30.2.1 Thread::Thread (Runnable * target)

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

Parameters

target	The target object to do some work with in a new thread.

6.30.2.2 Thread::~Thread ( )

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

6.30.3 Member Function Documentation

6.30.3.1 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.

6.30.3.2 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.

6.30.3.3 bool Thread::create ( )

Creates the thread if the target Runnable is valid, but does not start running it.

Returns

true if the thread was successfully created.

6.30.3.4 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.

6.30.3.5 bool Thread::created ( )

Returns

true if the thread has been created successfully.

6.30.3.6 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.

6.30.3.7 Runnable * Thread::getTarget ( )

Returns

a pointer to the target Runnable object

6.30.3.8 bool Thread::isThreadRunning ( )

Returns

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

6.30.3.9 bool Thread::isThreadSuspended ( )

Returns

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

6.30.3.10 bool Thread::join (int32_t *timeout*)

Blocks the calling thread until the worker thread managed by this object terminates. If the worker thread has already terminated, returns immediately. If the worker has not yet started, returns immediately.

Parameters

timeout	timeout in milliseconds to wait for the thread. If 0, does not wait at all. If negative, waits indefinitely.

Returns

true if the worker thread terminated successfully, false otherwise or if join() was not called legally.

6.30.3.11 bool Thread::start ( )

Starts the thread immediately if the thread has been created. This invokes the run() method in the Runnable target that was passed in the constructor.

Returns

true if the thread was successfully started.

6.30.3.12 bool Thread::started ( )

Returns

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

6.30.3.13 bool Thread::validTarget ( )

Returns

true if the Runnable target is valid.

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

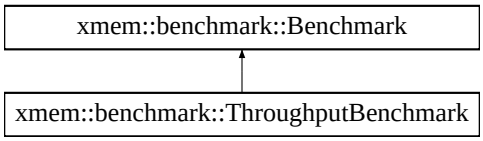
- src/thread/Thread.h
- src/thread/Thread.cpp

6.31 xmem::benchmark::ThroughputBenchmark Class Reference

A type of benchmark that measures memory throughput either via sequential, strided sequential, or random access patterns.

`#include <ThroughputBenchmark.h>`

Inheritance diagram for xmem::benchmark::ThroughputBenchmark:



Public Types

- typedef int32_t(∗ ThroughputBenchFunction)(void ∗, void ∗)

Public Member Functions

- ThroughputBenchmark (void ∗mem_array, size_t len, uint32_t iterations, xmem::common::chunk_size_t chunk_size, uint32_t cpu_node, uint32_t mem_node, uint32_t num_worker_threads, std::string name, xmem::timers::Timer ∗timer, std::vector< xmem::power::PowerReader ∗ > dram_power_readers, int64_t stride_size, xmem::common::pattern_mode_t pattern_mode, xmem::common::rw_mode_t rw_mode)

  Constructor.
- virtual bool run ()

  Runs the benchmark.
- virtual void report_benchmark_info ()

  Reports benchmark configuration details to the console.
- virtual void report_results ()

  Reports results to the console.
- int64_t getStrideSize ()

  Gets the stride size for this benchmark.
- xmem::common::pattern_mode_t getPatternMode ()

  Gets the pattern mode for this benchmark.
- xmem::common::rw_mode_t getRWMode ()

  Gets the read/write mode for this benchmark.

Additional Inherited Members

6.31.1 Detailed Description

A type of benchmark that measures memory throughput either via sequential, strided sequential, or random access patterns.

6.31.2 Member Typedef Documentation

6.31.2.1 typedef int32_t(* xmem::benchmark::ThroughputBenchmark::ThroughputBenchFunction)(void *, void *)

6.31.3 Constructor & Destructor Documentation

6.31.3.1 ThroughputBenchmark::ThroughputBenchmark (void * *mem_array*, size_t *len*, uint32_t *iterations*, xmem::common::chunk_size_t *chunk_size*, uint32_t *cpu_node*, uint32_t *mem_node*, uint32_t *num_worker_threads*, std::string *name*, xmem::timers::Timer * *timer*, std::vector< xmem::power::PowerReader * > *dram_power_readers*, int64_t *stride_size*, xmem::common::pattern_mode_t *pattern_mode*, xmem::common::rw_mode_t *rw_mode*)

Constructor.

Parameters

mem_array	a pointer to a contiguous chunk of memory that has been allocated for benchmarking among the worker threads. This should be aligned to a 256-bit boundary and should be the working set size times number of threads large.
len	Length of the raw_mem_array in bytes. This should be a multiple of 4 KB pages.
iterations	Number of iterations (passes) to do of the complete benchmark.
chunk_size	encoded size of an individual memory access.
cpu_node	the logical CPU NUMA node to use in the benchmark
mem_node	the logical memory NUMA node used in the benchmark
num_worker_threads	number of worker threads to use in the benchmark
name	name of the benchmark to use when reporting
timer	pointer to an existing Timer object
dram_power_readers	vector of pointers to PowerReader objects for measuring DRAM power
stride_size	For sequential access patterns, the stride counted in chunks. Negative values mean reverse access pattern. A stride of 1 is purely sequential.
pattern_mode	Indicates sequential or random access.
rw_mode	Indicates reads or writes. TODO: allow for a mixture

6.31.4 Member Function Documentation

6.31.4.1 pattern_mode_t ThroughputBenchmark::getPatternMode ( )

Gets the pattern mode for this benchmark.

Returns

The pattern mode enumerator.

6.31.4.2 rw_mode_t ThroughputBenchmark::getRWMode ( )

Gets the read/write mode for this benchmark.

Returns

The read/write mix mode.

6.31.4.3 int64_t ThroughputBenchmark::getStrideSize ( )

Gets the stride size for this benchmark.

Returns

The stride size in chunks.

6.31.4.4 void ThroughputBenchmark::report_benchmark_info ( ) [virtual]

Reports benchmark configuration details to the console.

Implements xmem::benchmark::Benchmark.

6.31.4.5 void ThroughputBenchmark::report_results ( ) [virtual]

Reports results to the console.

Implements xmem::benchmark::Benchmark.

6.31.4.6 bool ThroughputBenchmark::run ( ) [virtual]

Runs the benchmark.

Returns

true on success

Implements xmem::benchmark::Benchmark.

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

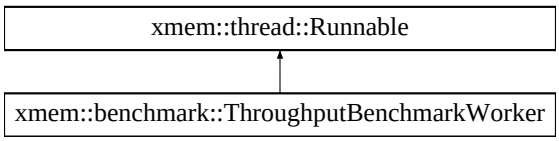
- src/benchmark/ThroughputBenchmark.h
- src/benchmark/ThroughputBenchmark.cpp

6.32 xmem::benchmark::ThroughputBenchmarkWorker Class Reference

Helper multithreading-friendly class to do the core throughput benchmark.

`#include <ThroughputBenchmarkWorker.h>`

Inheritance diagram for xmem::benchmark::ThroughputBenchmarkWorker:



Public Types

- typedef int32_t(∗ BenchFunction)(void ∗, void ∗)

Public Member Functions

- ThroughputBenchmarkWorker (void *mem_array, size_t len, BenchFunction bench_fptr, BenchFunction dummy_fptr, uint32_t cpu_affinity)

  Constructor.
- ~ThroughputBenchmarkWorker ()

  Destructor.
- virtual void run ()

  Thread-safe worker method.
- size_t getLen ()

  Gets the length of the memory region used by this worker.
- uint64_t getBytesPerPass ()

  Gets the number of bytes used in each pass of the benchmark by this worker.
- uint64_t getPasses ()

  Gets the number of passes for this worker.
- uint64_t getElapsedTicks ()

  Gets the elapsed ticks for this worker on the core benchmark kernel.
- uint64_t getElapsedDummyTicks ()

  Gets the elapsed ticks for this worker on the dummy version of the core benchmark kernel.
- uint64_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.

Additional Inherited Members

6.32.1 Detailed Description

Helper multithreading-friendly class to do the core throughput benchmark.

6.32.2 Member Typedef Documentation

6.32.2.1 typedef int32_t(* xmem::benchmark::ThroughputBenchmarkWorker::BenchFunction)(void *, void *)

6.32.3 Constructor & Destructor Documentation

6.32.3.1 ThroughputBenchmarkWorker::ThroughputBenchmarkWorker (void * *mem_array*, size_t *len*, BenchFunction *bench_fptr*, BenchFunction *dummy_fptr*, uint32_t *cpu_affinity*)

Constructor.

Parameters

mem_array	Pointer to the memory region to use by this worker.
len	Length of the memory region to use by this worker.
bench_fptr	Pointer to the core benchmark kernel to use.
dummy_fptr	Pointer to the dummy version of the core benchmark kernel to use.
cpu_affinity	Logical CPU identifier to lock this worker's thread to.

6.32.3.2 ThroughputBenchmarkWorker::~ThroughputBenchmarkWorker ( )

Destructor.

6.32.4 Member Function Documentation

6.32.4.1 uint64_t ThroughputBenchmarkWorker::getAdjustedTicks ( )

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

Returns

The adjusted ticks for this worker.

6.32.4.2 uint64_t ThroughputBenchmarkWorker::getBytesPerPass ( )

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

Returns

Number of bytes in each pass.

6.32.4.3 uint64_t ThroughputBenchmarkWorker::getElapsedDummyTicks ( )

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

Returns

The number of elapsed dummy ticks.

6.32.4.4 uint64_t ThroughputBenchmarkWorker::getElapsedTicks ( )

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

Returns

The number of elapsed ticks.

6.32.4.5 size_t ThroughputBenchmarkWorker::getLen ( )

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

Returns

Length of memory region in bytes.

6.32.4.6 uint64_t ThroughputBenchmarkWorker::getPasses ( )

Gets the number of passes for this worker.

Returns

The number of passes.

6.32.4.7 bool ThroughputBenchmarkWorker::hadWarning ( )

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

Returns

True if the worker's results had a warning.

6.32.4.8 void ThroughputBenchmarkWorker::run ( ) [virtual]

Thread-safe worker method.

Implements xmem::thread::Runnable.

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

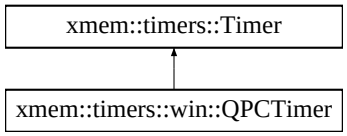
- src/benchmark/ThroughputBenchmarkWorker.h
- src/benchmark/ThroughputBenchmarkWorker.cpp

6.33 xmem::timers::Timer Class Reference

This class abstracts a simple high resolution stopwatch timer. WARNING: these objects are NOT thread safe.

`#include <Timer.h>`

Inheritance diagram for xmem::timers::Timer:



Public Member Functions

- Timer ()

  Constructor. This may take a noticeable amount of time.
- virtual void start ()=0

  Starts the timer.
- virtual uint64_t stop ()=0

  Stops the timer.
- double stop_in_ns ()

  Stops the timer.
- uint64_t get_ticks_per_sec ()

  Gets ticks per second for this timer.
- double get_ns_per_tick ()

  Gets nanoseconds per tick for this timer.

Protected Attributes

- uint64_t _ticks_per_sec
- double _ns_per_tick

6.33.1 Detailed Description

This class abstracts a simple high resolution stopwatch timer. WARNING: these objects are NOT thread safe.

6.33.2 Constructor & Destructor Documentation

6.33.2.1 Timer::Timer ( )

Constructor. This may take a noticeable amount of time.

6.33.3 Member Function Documentation

6.33.3.1 double Timer::get_ns_per_tick ( )

Gets nanoseconds per tick for this timer.

Returns

the number of nanoseconds per tick

6.33.3.2 uint64_t Timer::get_ticks_per_sec ( )

Gets ticks per second for this timer.

Returns

The reported number of ticks per second.

6.33.3.3 virtual void xmem::timers::Timer::start ( ) [pure virtual]

Starts the timer.

Implemented in xmem::timers::win::QPCTimer.

6.33.3.4 virtual uint64_t xmem::timers::Timer::stop ( ) [pure virtual]

Stops the timer.

Returns

Elapsed time since last start() call in ticks.

Implemented in xmem::timers::win::QPCTimer.

6.33.3.5 double Timer::stop_in_ns ( )

Stops the timer.

Returns

Elapsed time since last start() call in nanoseconds.

6.33.4 Member Data Documentation

6.33.4.1 double xmem::timers::Timer::_ns_per_tick [protected]

Nanoseconds per tick for this timer.

6.33.4.2 uint64_t xmem::timers::Timer::_ticks_per_sec [protected]

Ticks per second for this timer.

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

- src/timers/Timer.h
- src/timers/Timer.cpp

Chapter 7

File Documentation

7.1 src/benchmark/Benchmark.cpp File Reference

Implementation file for the Benchmark class.

```
#include "Benchmark.h"
#include <common/common.h>
#include <power/PowerReader.h>
#include <cstdint>
#include <iostream>
#include <vector>
```

7.1.1 Detailed Description

Implementation file for the Benchmark class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.2 src/benchmark/Benchmark.h File Reference

Header file for the Benchmark class.

```
#include <common/common.h>
#include <timers/Timer.h>
#include <power/PowerReader.h>
#include <thread/Thread.h>
#include <thread/Runnable.h>
#include <cstdint>
#include <string>
#include <vector>
```

Classes

- class xmem::benchmark::Benchmark

  Flexible abstract class for any memory benchmark.

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::benchmark

7.2.1 Detailed Description

Header file for the Benchmark class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.3 src/benchmark/benchmark_kernels/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/common.h>
```

Functions

- int asm_forwSequentialRead_Word256 (Word256_t *first_word, Word256_t *last_word)
- int asm_revSequentialRead_Word256 (Word256_t *last_word, Word256_t *first_word)
- int asm_forwSequentialWrite_Word256 (Word256_t *first_word, Word256_t *last_word)
- int asm_revSequentialWrite_Word256 (Word256_t *last_word, Word256_t *first_word)
- int asm_dummy_forwSequentialLoop_Word256 (Word256_t *first_word, Word256_t *last_word)
- int asm_dummy_revSequentialLoop_Word256 (Word256_t *first_word, Word256_t *last_word)

7.3.1 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!

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.3.2 Function Documentation

7.3.2.1 int asm_dummy_forwSequentialLoop_Word256 (Word256_t * *first_word*, Word256_t * *last_word*)

7.3.2.2 int asm_dummy_revSequentialLoop_Word256 (Word256_t * *first_word*, Word256_t * *last_word*)

7.3.2.3 int asm_forwSequentialRead_Word256 (Word256_t * *first_word,* Word256_t * *last_word*)

7.3.2.4 int asm_forwSequentialWrite_Word256 (Word256_t * *first_word,* Word256_t * *last_word*)

7.3.2.5 int asm_revSequentialRead_Word256 (Word256_t * *last_word,* Word256_t * *first_word*)

7.3.2.6 int asm_revSequentialWrite_Word256 (Word256_t * *last_word,* Word256_t * *first_word*)

7.4 src/benchmark/benchmark_kernels/benchmark_kernels.h File Reference

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

```
#include <cstdint>
```

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::benchmark
- xmem::benchmark::benchmark_kernels

Functions

- int32_t xmem::benchmark::benchmark_kernels::dummy_chasePointers (uintptr_t *, uintptr_t **, size_t len)

  Mimics the __chasePointers() method but doesn't do the memory accesses.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::dummy_forwSequentialLoop_Word64 (void *start_address, void *end_address)

  Used for measuring the time spent doing everything in forward sequential Word 64 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwSequentialLoop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward sequential Word 128 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwSequentialLoop_Word256 (void *start_address, void *end_address)

  Used for measuring the time spent doing everything in forward sequential Word 256 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::dummy_revSequentialLoop_Word64 (void *start_address, void *end_address)

Used for measuring the time spent doing everything in reverse sequential Word 64 loops except for the memory access itself.

- int32_t xmem::benchmark::benchmark_kernels::dummy_revSequentialLoop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse sequential Word 128 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revSequentialLoop_Word256 (void *start_address, void *end_address)

  Used for measuring the time spent doing everything in reverse sequential Word 256 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride2Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 2-strided Word 32 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride2Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 2-strided Word 64 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride2Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 2-strided Word 128 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride2Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 2-strided Word 256 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride2Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 32 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride2Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 64 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride2Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 128 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride2Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 2-strided Word 256 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride4Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 4-strided Word 32 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride4Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 4-strided Word 64 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride4Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 4-strided Word 128 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride4Loop_Word256 (void *start_address, void *end_address)

TODO. Used for measuring the time spent doing everything in forward 4-strided Word 256 loops except for the memory access itself.

- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride4Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 32 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride4Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 64 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride4Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 128 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride4Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 4-strided Word 256 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride8Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 8-strided Word 32 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride8Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 8-strided Word 64 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride8Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 8-strided Word 128 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride8Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 8-strided Word 256 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride8Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 32 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride8Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 64 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride8Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 128 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride8Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 8-strided Word 256 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride16Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 16-strided Word 32 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride16Loop_Word64 (void *start_address, void *end_address)

TODO. Used for measuring the time spent doing everything in forward 16-strided Word 64 loops except for the memory access itself.

- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride16Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 16-strided Word 128 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_forwStride16Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in forward 16-strided Word 256 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride16Loop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 32 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride16Loop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 64 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride16Loop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 128 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_revStride16Loop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in reverse 16-strided Word 256 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_randomLoop_Word32 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in random Word 32 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_randomLoop_Word64 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in random Word 64 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_randomLoop_Word128 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in random Word 128 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::dummy_randomLoop_Word256 (void *start_address, void *end_address)

  TODO. Used for measuring the time spent doing everything in random Word 256 loops except for the memory access itself.
- int32_t xmem::benchmark::benchmark_kernels::forwSequentialRead_Word32 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, reading in 32-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwSequentialRead_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, reading in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwSequentialRead_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory forward sequentially, reading in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwSequentialRead_Word256 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, reading in 256-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::revSequentialRead_Word32 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, reading in 32-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revSequentialRead_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, reading in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revSequentialRead_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory reverse sequentially, reading in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revSequentialRead_Word256 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, reading in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwSequentialWrite_Word32 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, writing in 32-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwSequentialWrite_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, writing in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwSequentialWrite_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory forward sequentially, writing in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwSequentialWrite_Word256 (void *start_address, void *end_address)

  Walks over the allocated memory forward sequentially, writing in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revSequentialWrite_Word32 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, writing in 32-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revSequentialWrite_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, writing in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revSequentialWrite_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory reverse sequentially, writing in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revSequentialWrite_Word256 (void *start_address, void *end_address)

  Walks over the allocated memory reverse sequentially, writing in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::forwStride2Read_Word64 (void *start_address, void *end↩_address)

  Walks over the allocated memory in forward strides of size 2, reading in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwStride2Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 2, reading in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwStride2Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 2, reading in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::revStride2Read_Word64 (void *start_address, void *end↩_address)

Walks over the allocated memory in reverse strides of size 2, reading in 64-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::revStride2Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 2, reading in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revStride2Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 2, reading in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::forwStride2Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 2, writing in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwStride2Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 2, writing in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwStride2Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 2, writing in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::revStride2Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 2, writing in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revStride2Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 2, writing in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revStride2Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 2, writing in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::forwStride4Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 4, reading in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwStride4Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 4, reading in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwStride4Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 4, reading in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::revStride4Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 4, reading in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revStride4Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 4, reading in 128-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::revStride4Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 4, reading in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::forwStride4Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 4, writing in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwStride4Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 4, writing in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwStride4Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 4, writing in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::revStride4Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 4, writing in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revStride4Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 4, writing in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revStride4Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 4, writing in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::forwStride8Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 8, reading in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwStride8Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 8, reading in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwStride8Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 8, reading in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::revStride8Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 8, reading in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revStride8Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 8, reading in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revStride8Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 8, reading in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::forwStride8Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 8, writing in 64-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::forwStride8Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 8, writing in 128-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::forwStride8Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 8, writing in 256-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::revStride8Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 8, writing in 64-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::revStride8Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 8, writing in 128-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::revStride8Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 8, writing in 256-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::forwStride16Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 16, reading in 64-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::forwStride16Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 16, reading in 128-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::forwStride16Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 16, reading in 256-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::revStride16Read_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 16, reading in 64-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::revStride16Read_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 16, reading in 128-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::revStride16Read_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 16, reading in 256-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::forwStride16Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in forward strides of size 16, writing in 64-bit chunks.

- int32_t xmem::benchmark::benchmark_kernels::forwStride16Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 16, writing in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::forwStride16Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in forward strides of size 16, writing in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::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::benchmark::benchmark_kernels::revStride16Write_Word64 (void *start_address, void *end_address)

  Walks over the allocated memory in reverse strides of size 16, writing in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revStride16Write_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 16, writing in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::revStride16Write_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in reverse strides of size 16, writing in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::randomRead_Word32 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, reading in 32-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::randomRead_Word64 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, reading in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::randomRead_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, reading in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::randomRead_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, reading in 256-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::randomWrite_Word32 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, writing in 32-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::randomWrite_Word64 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, writing in 64-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::randomWrite_Word128 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, writing in 128-bit chunks.
- int32_t xmem::benchmark::benchmark_kernels::randomWrite_Word256 (void *start_address, void *end_address)

  TODO. Walks over the allocated memory in random order, writing in 256-bit chunks.

7.4.1 Detailed Description

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

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.5 src/benchmark/BenchmarkManager.cpp File Reference

Implementation file for the BenchmarkManager class.

```
#include "BenchmarkManager.h"
#include <common/common.h>
#include <common/win/third_party/win_common_third_party.h>
#include <cstdint>
#include <stdlib.h>
#include <iostream>
#include <sstream>
#include <assert.h>
```

7.5.1 Detailed Description

Implementation file for the BenchmarkManager class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.6 src/benchmark/BenchmarkManager.h File Reference

Header file for the BenchmarkManager class.

```
#include <common/common.h>
#include <timers/Timer.h>
#include <power/NativeDRAMPowerReader.h>
#include "Benchmark.h"
#include "ThroughputBenchmark.h"
#include "LatencyBenchmark.h"
#include <cstdint>
#include <vector>
#include <fstream>
```

Classes

- class xmem::benchmark::BenchmarkManager

  Manages running all benchmarks at a high level.

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::benchmark

7.6.1 Detailed Description

Header file for the BenchmarkManager class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.7 src/benchmark/LatencyBenchmark.cpp File Reference

Implementation file for the LatencyBenchmark class.

```
#include "LatencyBenchmark.h"
#include <common/common.h>
#include <benchmark/benchmark_kernels/benchmark_kernels.h>
#include <iostream>
#include <algorithm>
#include <random>
#include <assert.h>
#include <time.h>
```

7.7.1 Detailed Description

Implementation file for the LatencyBenchmark class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.8 src/benchmark/LatencyBenchmark.h File Reference

Header file for the LatencyBenchmark class.

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

Classes

- class xmem::benchmark::LatencyBenchmark

  A type of benchmark that measures memory latency via random pointer chasing. TODO: loaded latency tests.

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::benchmark

7.8.1 Detailed Description

Header file for the LatencyBenchmark class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.9 src/benchmark/ThroughputBenchmark.cpp File Reference

Implementation file for the ThroughputBenchmark class.

```
#include "ThroughputBenchmark.h"
#include <common/common.h>
#include "ThroughputBenchmarkWorker.h"
#include <benchmark/benchmark_kernels/benchmark_kernels.h>
#include <thread/Thread.h>
#include <thread/Runnable.h>
#include <iostream>
#include <algorithm>
#include <assert.h>
#include <random>
#include <time.h>
```

7.9.1 Detailed Description

Implementation file for the ThroughputBenchmark class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.10 src/benchmark/ThroughputBenchmark.h File Reference

Header file for the ThroughputBenchmark class.

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

Classes

- class xmem::benchmark::ThroughputBenchmark

  A type of benchmark that measures memory throughput either via sequential, strided sequential, or random access patterns.

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::benchmark

7.10.1 Detailed Description

Header file for the ThroughputBenchmark class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.11 src/benchmark/ThroughputBenchmarkWorker.cpp File Reference

Implementation file for the ThroughputBenchmarkWorker class.

```
#include "ThroughputBenchmarkWorker.h"
#include <benchmark/benchmark_kernels/benchmark_kernels.h>
#include <iostream>
#include <common/common.h>
```

7.11.1 Detailed Description

Implementation file for the ThroughputBenchmarkWorker class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.12 src/benchmark/ThroughputBenchmarkWorker.h File Reference

Header file for the ThroughputBenchmarkWorker class.

```
#include <thread/Runnable.h>
#include <common/common.h>
#include <cstdint>
```

Classes

- class xmem::benchmark::ThroughputBenchmarkWorker

  Helper multithreading-friendly class to do the core throughput benchmark.

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::benchmark

7.12.1 Detailed Description

Header file for the ThroughputBenchmarkWorker class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.13 src/common/common.cpp File Reference

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

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

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::common

Variables

- size_t xmem::common::g_page_size
- size_t xmem::common::g_large_page_size
- uint32_t xmem::common::g_num_nodes
- uint32_t xmem::common::g_num_logical_cpus
- uint32_t xmem::common::g_num_physical_packages
- uint32_t xmem::common::g_starting_test_index
- uint32_t xmem::common::g_test_index

7.13.1 Detailed Description

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

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.14 src/common/common.h File Reference

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

```
#include <cstdint>
```

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::common

Macros

- #define VERSION 1.0
- #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 GB_4 4294967296
- #define DEFAULT_PAGE_SIZE 4096
- #define DEFAULT_WORKING_SET_SIZE DEFAULT_PAGE_SIZE
- #define DEFAULT_NUM_CPUS 1

- #define DEFAULT_NUM_NODES 1
- #define DEFAULT_THREAD_JOIN_TIMEOUT 600000
- #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 VERBOSE
- #define USE_ALL_NUMA_NODES
- #define MULTITHREADING_ENABLE
- #define USE_TSC_TIMER
- #define USE_LARGE_PAGES
- #define USE_TIME_BASED_BENCHMARKS
- #define BENCHMARK_DURATION_SEC 4
- #define THROUGHPUT_BENCHMARK_BYTES_PER_PASS 4096
- #define USE_THROUGHPUT_SEQUENTIAL_PATTERN
- #define USE_THROUGHPUT_FORW_STRIDE_1
- #define USE_THROUGHPUT_READS
- #define USE_THROUGHPUT_WRITES
- #define USE_LATENCY_BENCHMARK_RANDOM_SHUFFLE_PATTERN
- #define POWER_SAMPLING_PERIOD_SEC 1

Typedefs

- typedef uint32_t xmem::common::Word32_t

Enumerations

- enum xmem::common::pattern_mode_t { xmem::common::SEQUENTIAL, xmem::common::NUM_PATTERN_MODES }

  Memory access patterns are broadly categorized by sequential or random-access.
- enum xmem::common::rw_mode_t { xmem::common::READ, xmem::common::WRITE, xmem::common::NUM_RW_MODES }

  Memory access batterns are broadly categorized by reads and writes.
- enum xmem::common::chunk_size_t { xmem::common::NUM_CHUNK_SIZES }

  Legal memory read/write chunk sizes in bits.

Functions

- void xmem::common::print_welcome_message ()

  Prints a basic welcome message to the console with useful information.
- void xmem::common::print_types_report ()

  Prints the various C/C++ types to the console for this machine.
- void xmem::common::print_compile_time_options ()

  Prints compile-time option information to the console.
- void xmem::common::test_timers ()

  Tests any enabled timers and outputs results to the console for sanity checking.
- void xmem::common::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::common::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::common::unlock_thread_to_numa_node ()

  Clears the affinity of the calling thread to any given NUMA node.
- bool xmem::common::lock_thread_to_cpu (uint32_t cpu_id)

  Sets the affinity of the calling thread to a given logical CPU.
- bool xmem::common::unlock_thread_to_cpu ()

  Clears the affinity of the calling thread to any given logical CPU.
- int32_t xmem::common::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::common::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::common::config_page_size ()

  Queries the page sizes from the system and sets relevant global variables.

7.14.1 Detailed Description

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

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.14.2 Macro Definition Documentation

7.14.2.1 #define BENCHMARK_DURATION_SEC 4

RECOMMENDED VALUE: At least 2. Number of seconds to run in each benchmark.

7.14.2.2 #define DEFAULT_NUM_CPUS 1

Default number of logical CPU cores.

7.14.2.3 #define DEFAULT_NUM_NODES 1

Default number of NUMA nodes.

7.14.2.4 #define DEFAULT_PAGE_SIZE 4096

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

7.14.2.5 #define DEFAULT_THREAD_JOIN_TIMEOUT 600000

Default number of milliseconds to wait for a thread to join. Negative values mean indefinite wait.

7.14.2.6 #define DEFAULT_WORKING_SET_SIZE DEFAULT_PAGE_SIZE

Default working set size in bytes.

7.14.2.7 #define GB 1073741824

7.14.2.8 #define GB_4 4294967296

7.14.2.9 #define KB 1024

7.14.2.10 #define LATENCY_BENCHMARK_UNROLL_LENGTH 512

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

7.14.2.11 #define MB 1048576

7.14.2.12 #define MB_16 16777216

7.14.2.13 #define MB_256 268435456

7.14.2.14 #define MB_4 4194304

7.14.2.15 #define MB_512 536870912

7.14.2.16 #define MB_64 67108864

7.14.2.17 #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.

7.14.2.18 #define MULTITHREADING_ENABLE

RECOMMENDED ENABLED. Use multiple threads for benchmarks wherever applicable. Note that power measurement is always done with multiple threads separate from the benchmarking threads, regardless if this option is set or not.

7.14.2.19 #define POWER_SAMPLING_PERIOD_SEC 1

RECOMMENDED VALUE: 1. Sampling period in seconds for all power measurement mechanisms.

7.14.2.20 #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.

7.14.2.21 #define UNROLL1024(*x*) UNROLL512(x) UNROLL512(x)

7.14.2.22 #define UNROLL128(*x*) UNROLL64(x) UNROLL64(x)

7.14.2.23 #define UNROLL16(*x*) UNROLL8(x) UNROLL8(x)

7.14.2.24 #define UNROLL16384(*x*) UNROLL8192(x) UNROLL8192(x)

7.14.2.25 #define UNROLL2(*x*) x x

7.14.2.26 #define UNROLL2048(*x*) UNROLL1024(x) UNROLL1024(x)

7.14.2.27 #define UNROLL256(*x*) UNROLL128(x) UNROLL128(x)

7.14.2.28 #define UNROLL32(*x*) UNROLL16(x) UNROLL16(x)

7.14.2.29 #define UNROLL32768(*x*) UNROLL16384(x) UNROLL16384(x)

7.14.2.30 #define UNROLL4(*x*) UNROLL2(x) UNROLL2(x)

7.14.2.31 #define UNROLL4096(*x*) UNROLL2048(x) UNROLL2048(x)

7.14.2.32 #define UNROLL512(*x*) UNROLL256(x) UNROLL256(x)

7.14.2.33 #define UNROLL64(*x*) UNROLL32(x) UNROLL32(x)

7.14.2.34 #define UNROLL65536(*x*) UNROLL32768(x) UNROLL32768(x)

7.14.2.35 #define UNROLL8(*x*) UNROLL4(x) UNROLL4(x)

7.14.2.36 #define UNROLL8192(*x*) UNROLL4096(x) UNROLL4096(x)

7.14.2.37 #define USE_ALL_NUMA_NODES

RECOMMENDED ENABLED. Test all NUMA node combinations for CPU and memory. If disabled, only node 0 is used for both CPU and memory.

7.14.2.38 #define USE_LARGE_PAGES

RECOMMENDED ENABLED. Allocate memory using large pages rather than small normal pages. In general, this is highly recommended, as the TLB can skew benchmark results for DRAM.

7.14.2.39 #define USE_LATENCY_BENCHMARK_RANDOM_SHUFFLE_PATTERN

RECOMMENDED ENABLED. In latency benchmarks, generate the pointer chasing pattern using a random shuffle, which has a chance of creating small cycles. Much faster to run but strictly less correct. O(N)

7.14.2.40 #define USE_THROUGHPUT_FORW_STRIDE_1

RECOMMENDED ENABLED. In throughput benchmarks with sequential pattern, do forward strides of 1 chunk (forward sequential).

7.14.2.41 #define USE_THROUGHPUT_READS

RECOMMENDED ENABLED. In throughput benchmarks, read from memory.

7.14.2.42 #define USE_THROUGHPUT_SEQUENTIAL_PATTERN

RECOMMENDED ENABLED. Run the sequential family pattern of ThroughputBenchmarks.

7.14.2.43 #define USE_THROUGHPUT_WRITES

RECOMMENDED ENABLED. In throughput benchmarks, write to memory.

7.14.2.44 #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).

7.14.2.45 #define USE_TSC_TIMER

RECOMMENDED DISABLED. Use the Intel Time Stamp Counter native hardware timer. Only use this if you know what you are doing.

7.14.2.46 #define VERBOSE

Increases console output information detail by a lot.

7.14.2.47 #define VERSION 1.0

7.15 src/common/win/third_party/win_common_third_party.cpp File Reference

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

```
#include "win_common_third_party.h"
#include <common/common.h>
#include <common/win/win_common.h>
#include <iostream>
#include <malloc.h>
#include <stdio.h>
#include <tchar.h>
```

7.15.1 Detailed Description

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

7.16 src/common/win/third_party/win_common_third_party.h File Reference

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

7.16.1 Detailed Description

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

7.17 src/common/win/third_party/win_CPdhQuery.h File Reference

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

```
#include "win_common_third_party.h"
#include <windows.h>
#include <pdh.h>
#include <pdhmsg.h>
#include <string>
#include <map>
#include <sstream>
#include <vector>
#include <tchar.h>
#include <iostream>
```

Classes

- class xmem::common::win::third_party::CPdhQuery

  A third-party class for querying performance counter data from Windows.
- class xmem::common::win::third_party::CPdhQuery::CException

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::common
- xmem::common::win
- xmem::common::win::third_party

7.17.1 Detailed Description

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

7.18 src/common/win/win_common.cpp File Reference

Implementation file for some common Windows helper stuff.

```
#include <common/common.h>
#include "win_common.h"
#include <windows.h>
```

7.18.1 Detailed Description

Implementation file for some common Windows helper stuff.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.19 src/common/win/win_common.h File Reference

Header file for some common Windows helper stuff.

7.19.1 Detailed Description

Header file for some common Windows helper stuff.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.20 src/config/Configurator.cpp File Reference

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

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

7.20.1 Detailed Description

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

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.21 src/config/Configurator.h File Reference

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

```
#include <common/common.h>
#include "third_party/optionparser.h"
#include "third_party/MyArg.h"
#include <cstdint>
#include <string>
```

Classes

- class xmem::config::Configurator

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

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::config

Enumerations

- enum xmem::config::optionIndex { xmem::config::UNKNOWN, xmem::config::HELP, xmem::config::MEAS_LATENCY, xmem::config::MEAS_↩ THROUGHPUT, xmem::config::WORKING_SET_SIZE, xmem::config::ITERATIONS, xmem::config::BASE_TEST_INDEX, xmem::config::OUTPUT_FILE }

  Enumerates all possible types of command-line options.

Variables

- const third_party::Descriptor xmem::config::usage []

  Command-line option descriptors as needed by stuff in <config/third_party/optionparser.h>. This is basically the help message content.

7.21.1 Detailed Description

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

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.22 src/config/third_party/ExampleArg.h File Reference

Slightly-modified third-party code related to OptionParser.

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

Classes

- class xmem::config::third_party::ExampleArg

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::config
- xmem::config::third_party

7.22.1 Detailed Description

Slightly-modified third-party code related to OptionParser.

7.23 src/config/third_party/MyArg.h File Reference

Extensions to third-party optionparser-related code.

```
#include <cstdint>
#include <stdio.h>
#include "ExampleArg.h"
```

Classes

- class xmem::config::third_party::MyArg

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::config
- xmem::config::third_party

7.23.1 Detailed Description

Extensions to third-party optionparser-related code.

Author

Mark Gottscho Email: mgottscho@ucla.edu, t-magott@microsoft.com Summer 2014 Microsoft Research Intern

7.24 src/config/third_party/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::config::third_party::Descriptor

  Describes an option, its help text (usage) and how it should be parsed.
- class xmem::config::third_party::Option

  A parsed option from the command line together with its argument if it has one.
- struct xmem::config::third_party::Arg

  Functions for checking the validity of option arguments.
- struct xmem::config::third_party::Stats

  Determines the minimum lengths of the buffer and options arrays used for Parser.
- class xmem::config::third_party::Parser

  Checks argument vectors for validity and parses them into data structures that are easier to work with.
- struct xmem::config::third_party::Parser::Action

- class xmem::config::third_party::Stats::CountOptionsAction
- class xmem::config::third_party::Parser::StoreOptionAction
- struct xmem::config::third_party::PrintUsageImplementation
- struct xmem::config::third_party::PrintUsageImplementation::IStringWriter
- struct xmem::config::third_party::PrintUsageImplementation::FunctionWriter< Function >
- struct xmem::config::third_party::PrintUsageImplementation::OStreamWriter< OStream >
- struct xmem::config::third_party::PrintUsageImplementation::TemporaryWriter< Temporary >
- struct xmem::config::third_party::PrintUsageImplementation::SyscallWriter< Syscall >
- struct xmem::config::third_party::PrintUsageImplementation::StreamWriter< Function, Stream >
- class xmem::config::third_party::PrintUsageImplementation::LinePartIterator
- class xmem::config::third_party::PrintUsageImplementation::LineWrapper

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::config
- xmem::config::third_party

Typedefs

- typedef ArgStatus(* xmem::config::third_party::CheckArg)(const Option &option, bool msg)

  Signature of functions that check if an argument is valid for a certain type of option.

Enumerations

- enum xmem::config::third_party::ArgStatus { xmem::config::third_party::ARG_NONE, xmem::config::third_party::ARG_OK, xmem::config::third_party::ARG_IGNORE, xmem::config::third_party::ARG_ILLEGAL }

  Possible results when checking if an argument is valid for a certain option.

Functions

- template<typename OStream >
  void xmem::config::third_party::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::config::third_party::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::config::third_party::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::config::third_party::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::config::third_party::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)

7.24.1 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.

7.25 src/main.cpp File Reference

main entry point to the tool

```
#include <common/common.h>
#include <common/win/win_common.h>
#include <common/win/third_party/win_common_third_party.h>
#include <config/Configurator.h>
#include <benchmark/BenchmarkManager.h>
#include <iostream>
```

Functions

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

  The main entry point to the program.

7.25.1 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.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.25.2 Function Documentation

7.25.2.1 int main (int *argc*, char * *argv[]*)

The main entry point to the program.

7.26 src/power/NativeDRAMPowerReader.cpp File Reference

Implementation file for the NativeDRAMPowerReader class.

```
#include <power/NativeDRAMPowerReader.h>
#include <common/common.h>
#include <cstdint>
#include <vector>
```

7.26.1 Detailed Description

Implementation file for the NativeDRAMPowerReader class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.27 src/power/NativeDRAMPowerReader.h File Reference

Header file for the NativeDRAMPowerReader class.

```
#include <common/common.h>
#include <power/PowerReader.h>
#include <thread/Runnable.h>
#include <common/win/third_party/win_CPdhQuery.h>
#include <cstdint>
#include <vector>
#include <string>
```

Classes

- class xmem::power::NativeDRAMPowerReader

  A class for measuring socket-level DRAM power from the OS performance counter interface.

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::power

7.27.1 Detailed Description

Header file for the NativeDRAMPowerReader class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.28 src/power/PowerReader.cpp File Reference

Implementation file for the PowerReader class.

```
#include "PowerReader.h"
#include <common/common.h>
#include <cstdint>
#include <vector>
#include <iostream>
```

7.28.1 Detailed Description

Implementation file for the PowerReader class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.29 src/power/PowerReader.h File Reference

Header file for the PowerReader class.

```
#include <common/common.h>
#include <thread/Runnable.h>
#include <cstdint>
#include <vector>
```

Classes

- class xmem::power::PowerReader

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

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::power

7.29.1 Detailed Description

Header file for the PowerReader class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.30 src/thread/Runnable.cpp File Reference

Implementation file for the Runnable class.

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

7.30.1 Detailed Description

Implementation file for the Runnable class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.31 src/thread/Runnable.h File Reference

Header file for the Runnable class.

```
#include <cstdint>
```

Classes

- class xmem::thread::Runnable

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

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::thread

7.31.1 Detailed Description

Header file for the Runnable class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.32 src/thread/Thread.cpp File Reference

Implementation file for the Thread class.

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

7.32.1 Detailed Description

Implementation file for the Thread class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.33 src/thread/Thread.h File Reference

Header file for the Thread class.

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

Classes

- class xmem::thread::Thread

  a nice wrapped thread interface independent of particular OS API

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::thread

7.33.1 Detailed Description

Header file for the Thread class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.34 src/timers/Timer.cpp File Reference

Implementation file for the Timer class.

```
#include "Timer.h"
```

7.34.1 Detailed Description

Implementation file for the Timer class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.35 src/timers/Timer.h File Reference

Header file for the Timer class.

```
#include <cstdint>
```

Classes

- class xmem::timers::Timer

  This class abstracts a simple high resolution stopwatch timer. WARNING: these objects are NOT thread safe.

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::timers

7.35.1 Detailed Description

Header file for the Timer class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.36 src/timers/win/QPCTimer.cpp File Reference

Header file for the QPCTimer class.

```
#include "QPCTimer.h"
```

7.36.1 Detailed Description

Header file for the QPCTimer class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.37 src/timers/win/QPCTimer.h File Reference

Header file for the QPCTimer class.

```
#include <timers/Timer.h>
#include <cstdint>
```

Classes

- class xmem::timers::win::QPCTimer

  This class implements a simple high resolution stopwatch timer based on Windows' QueryPerformanceCounter API. WARNING: these objects are NOT thread safe.

Namespaces

- xmem

  The namespace of The Lean Mean C++ Option Parser.
- xmem::timers
- xmem::timers::win

Functions

- uint64_t xmem::timers::win::get_qpc_time ()

  Query the QPC timer.

7.37.1 Detailed Description

Header file for the QPCTimer class.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.38 src/timers/x86_64/TSCTimer.cpp File Reference

Implementation file for the TSCTimer class as well as some C-style functions for working with the TSC timer hardware directly.

```
#include "TSCTimer.h"
```

7.38.1 Detailed Description

Implementation file for the TSCTimer class as well as some C-style functions for working with the TSC timer hardware directly.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation

7.39 src/timers/x86_64/TSCTimer.h File Reference

Header file for the TSCTimer class as well as some C-style functions for working with the TSC timer hardware directly.

```
#include <common/common.h>
#include <timers/Timer.h>
#include <cstdint>
```

7.39.1 Detailed Description

Header file for the TSCTimer class as well as some C-style functions for working with the TSC timer hardware directly.

Author

Mark Gottscho Email: mgottscho@ucla.edu (C) 2014 Microsoft Corporation