The Memory Performance Micro-Benchmark Suite
User's Guide

Douglas M. Pase
doug.pase@emc.com

Introduction

The Memory Performance Micro-Benchmark Suite (MBS) is a collection of benchmarks that measure a computer's performance at a fundamental level. The suite contains benchmarks for measuring transition time into and out of the operating system kernel, the time to copy data from one location in memory to another, memory latency, memory bandwidth, the latency of synchronization and computational kernels. Measurements of this type must reflect the variety of architectures that exist today. In particular, support for NUMA systems with deep cache hierarchies and multiple cores in each domain considerably complicates benchmark design. This suite addresses those considerations in a flexible manner. It allows the user to control the size and location of data for a test, to test the performance of each level of cache, local memory, remote memory, with or without TLB misses. Furthermore, the operation can be measured under loaded or unloaded conditions. In every case performance is measured with a high-precision, cycle accurate timer based on the processor time-stamp counter. The accuracy of the benchmark can by be controlled by specifying the duration of the measurement or the number of iterations.

These benchmarks are collected into a suite for two reasons. First, the benchmarks are related in that they all measure similar fundamental properties of computer systems, mostly centered around memory or computational performance of the system. They benefit from having similar controls for similar operations, and from being measured in similar ways. Second, these benchmarks are most valuable when they provide measurements repeated over a variety of conditions. For example, reporting that the memory latency of a system is *x* nanoseconds provides some information, but showing how the latency varies over the cache hierarchy, local memory with and without TLB hits, and remote memory, is much more informative. It gives the reader clues as to how to exploit the resources of the system to create high performance codes.

The Benchmarks

The suite contains the following benchmarks.

getpid - This benchmark measures the kernel transition time through repeated calls to the Linux function `getpid()`. This function is the simplest of all the Linux system calls. It enters the kernel, chases one pointer (`current->ppid`) and returns that value to the user. As such, the vast majority of the time for this call is spent in the transition into and out of the kernel.

memcpy - This benchmark measures the rate at which data can be copied from one block of memory to another. It supports a variety of memory copy routines, including the libc routines `bcopy()`, `bzero()`, `memcpy()`, `memmove()` and `memset()`. In addition, it also supports several hand-coded copy routines, including a naive copy and copy routines that use hand-unrolled loops and non-temporal (non-caching) loads and stores. It allows the user to specify the size of the data to be copied, as well as the size and placement of the pool it is to be copied from and the placement of the pool it is to be copied to.

pchase - *pchase* is a benchmark that measures memory latency within a system. It does this by creating a chain of pointers on start-up, then repeatedly chasing those pointers to the end. This approach is more accurate than other software methods of measuring memory latency because the system has no means of parallelizing the operation or prefetching data. The processor cannot begin to fetch the next address until the previous address has arrived. The benchmark can measure memory loading by controlling the number of pointer de-references within a single thread, or the number of threads, or both.

stream - This is the popular STREAM benchmark originally created by John McCalpin at the University of Virginia (http://www.cs.virginia.edu/stream/). It measures memory bandwidth for four common floating-point kernels, namely *copy, scale, add* and *triad*. All four kernels treat all data as double-precision floating-point values. The copy operation is a vector copy, or `a[*]=b[*]`; scale is a vector copy operation mixed with a scalar multiply, or `a[*]=s*b[*]`; add is a vector add operation, or `a[*]=b[*]+c[*]`; and triad is a vector add mixed with a scalar multiplication, or `a[*]=b[*]+s*c[*]`. This benchmark is included because it is well known, but it is also highly sensitive to choice of compiler and compiler options and difficult to get right. Its results also tend to be more optimistic than other benchmarks, because of details in the way the benchmark is defined.

sha - This benchmark measures the performance of a selection of the 160-bit SHA-1 and 256-bit SHA-2 hash functions. The benchmark allows the user to specify the size of message to be hashed and the size and placement of pool the message is taken from. The SHA-1 code also makes use of an internal memory copy routine, and the user is allowed to specify which memory copy routine is used.

sync-c - This benchmark is experimental, a work in progress. It measures the time to access and release a single synchronization resource under contention. The benchmark controls the number of threads to be used. The synchronization object is always placed on domain 0. All threads wait at a barrier before accessing the resource.

sync-u - This benchmark measures the time to access and release synchronization resources when *not* under contention. The benchmark allows the user to control the number and placement of synchronization objects and threads. The benchmark supports POSIX thread spin locks, mutexes and reader/writer locks, plus 32-bit and 64-bit cmpxchg (x86 compare-and-exchange) instructions and small collection of simple hand coded operations built from cmpxchg instructions.

getpid

Getpid is the simplest of benchmarks. It accepts no command line arguments. It first calibrates the time stamp counter, then it performs as many `getpid()` operations as it can in one second. It reports the total number of operations, the amount of time it spent and the average time in nanoseconds to complete one operation.

memcpy

As mentioned above, *memcpy* measures the throughput of various memory data copy operations. The data is copied from one block of memory, the source, to another block, the destination. Source and destination blocks are selected from separate pools, and the locations of each pool may be indicated separately.

The supported operations are as follows:

1. *bcopy*, the `bcopy()` memory copy routine from libc (deprecated).
2. *memcpy*, the `memcpy()` memory copy routine from libc.
3. *memmove*, the `memmove()` memory copy routine from libc.
4. *copy_tt*, a copy routine that does 64-bit aligned word copies, with temporal (caching) loads and stores.
5. *copy_tn*, a copy routine that does 64-bit aligned word copies, with temporal (caching) loads and non-temporal (non-caching) stores.
6. *xtta*, a copy routine that uses the `movdqa` instruction to move aligned 128-bit words through `xmm` registers to perform temporal transfers of data.
7. *xttu*, a copy routine that uses the `movdqu` instruction to move unaligned 128-bit words through `xmm` registers to perform temporal transfers of data.
8. *xtn*, a copy routine that uses `movdqa` instructions to perform a temporal transfer of 128-bit words into `xmm` registers, and `movntdq` instructions to perform a non-temporal transfer back to memory.
9. *xnt*, a copy routine that uses `movntdqa` instructions to perform a non-temporal transfer of 128-bit words into `xmm` registers, and `movdqa` instructions to perform a temporal transfer back to memory.
10. *xnn*, a copy routine that uses `movntdqa` instructions to perform a non-temporal transfer of 128-bit words into `xmm` registers, and `movntdq` instructions to perform a non-temporal transfer back to memory.
11. *bzero*, the `bzero()` memory zeroing routine from libc (deprecated).
12. *memset*, the `memset()` memory initialization routine from libc.
13. *naivezero*, a simple and intuitive hand-coded byte zeroing routine.

This benchmark supports many command line options for controlling benchmark behavior, as follows.

-L This option (*--library*) selects which facilities are to be used internally to map processors and data to NUMA domains. *-L numa* specifies that *libnuma* will be used. *-L sched* specifies that `sched_setaffinity()` will be used. When *-L sched* is specified, the *-d* option must also be used to specify the mapping of threads to NUMA domains.

-d This option (*--domains*) describes the mapping of threads to NUMA domains. It may only be used in conjunction with the *-L sched* option. The syntax for this option is *-d <list>*, where *<list>* is a comma separated list of NUMA domains where each corresponding

logical processor core resides. Logical cores and NUMA domains start at zero and count upwards. For example, a list of 0,0,0,0,1,1,1,1 indicates that there are eight logical cores and two NUMA domains in the system - eight cores because there are eight values in the list, two NUMA domains because the highest value in the list is 1. In this example, cores 0 through 3 all map to domain 0, and cores 4 through 7 map to domain 1.

-a This option (--*access*) describes the pattern used to select blocks to copy. Three values are supported for this option, namely *random*, *forward*, and *backward*. *Random* indicates the blocks to be copied are selected randomly within the pool and is the default when no selection is given. As their names suggest, *forward* and *backward* select blocks sequentially in the indicated direction. In every case the order in which blocks will be selected is determined once, prior to the execution of the timed portion of the benchmark, and used throughout the remainder of the benchmark. It is not changed dynamically while the performance is being measured.

-c This option (--*copy*) gives the size of the block to be copied. The syntax is *-c <number>* where *<number>* is a positive octal, decimal or hexadecimal integer. Numbers follow the standard C/C++ format, that is, octal numbers are preceded by 0 and fall in the range of 0 to 7, decimal numbers begin with a digit other than 0, and hexadecimal numbers are preceded by 0x and fall in the range of 0 through 9 or A through F. Additionally, decimal numbers can be followed by the postfix characters K, M, G, T, meaning the number is to be scaled by 2^{10}, 2^{20}, 2^{30} or 2^{40}, respectively. The default value for this parameter is 4K, or 4096 bytes.

-e This option (--*experiments*) gives the number of experiments to be run. The syntax for this option is *-e <number>* where *<number>* is a positive octal, decimal or hexadecimal integer, as above. When this value is greater than 1, only the *best* result of all experiments is reported. The default value is 1.

-i This option (--*iterations*) gives the number of iterations to be used in each experiment. The duration of any experiment may be specified either using the *-i* or *-s* option, but not both. The former specifies the number of iterations to use, the latter specifies the number of seconds each experiment is to run. The syntax for this option is *-i <number>* where *<number>* is a positive octal, decimal or hexadecimal integer, as above. The default is to run for 1 second, as with the *-s* option described below.

-m This option (--*method*) describes the method used to copy the bytes from memory. The syntax for this option is *-m <method>* where *<method>* is one of *bcopy*, *memcpy*, *memmove*, *copy_tt*, *copy_tn*, *xtta*, *xttu*, *xtn*, *xnt*, *xnn*, *bzero*, *memset*, or *naivezero*. Each of these methods is described in more detail above.

-n This option (--*numa*) describes the placement of data relative to the thread that operates on it. Each thread uses two pools of memory, one for the source of the copy, one for the destination (or target). So, each thread must have three NUMA domains specified: the domain where the thread resides, the domain where the source pool resides and the

domain where the destination pool resides. This option allows the user to specify each of these values in several ways.

First, *-n local* specifies that threads will be allocated round robin among domains, and the source and destination pools will be local to, or reside where, the thread resides.

Second, *-n gen <thrd> <src> <dest>* specifies that the mapping for the thread, source and destination pools will be according to the expressions *thrd*, *src* and *dest*, respectively. Each expression gives the domain ID as a value relative to the thread number. Threads and domains are both numbered consecutively starting at zero. A constant value, such as 0 or 1 means that the thread or pool is to be affinitized to that domain, in this example, 0 or 1. So a specification of *-n gen 0 0 0* would map all threads, source pools and destination pools to domain 0. Expressions may also be a constant with a leading '+' or '-', which indicates the domain is the thread ID plus (or minus) the constant value modulo the number of domains in the system. For example, consider a system with only two NUMA domains, 0 and 1. In this case an expression such as +1 would indicate the domain would be the thread ID plus 1 modulo 2. Independent of the number of domains within a system, one commonly used specification is *-n gen +0 +0 +0* for distributing all threads evenly across the domains and placing the pools locally (the same as *-n local* above). Another useful specification is *-n gen +0 +1 +0* for distributing all threads evenly, as before, but placing the source pool remotely. In other words, it measures the throughput of copying a remote source to a local destination.

Third, a much greater degree of control can be expressed using the *map* specification. The *map* syntax is *-n map <map>*. For this specification, *<map>* is a list of thread domain specifications for each thread and pool, separated by semicolons. A thread domain specification uses the form *t:s,d* where *t* is the domain where the thread resides, *s* is the domain where the source pool resides, and *d* is the domain where the destination pool resides. Only constant integers are allowed for the values *t*, *s* and *d*. For example, the specification *-n map 0:0,0;1:1,1;2:2,2;3:3,3* indicates there are four threads (because there are four thread domain specifications). Thread 0 is affinitized to domain 0, thread 1 is on domain 1, thread 2 is on domain 2 and thread 3 is on domain 3. Furthermore, the source and destination pools happen to be mapped to the same domain where the thread resides. If a thread or pool domain exceeds the maximum domain it is wrapped around using a modulo function.

-o This option (*--output*) describes the output format. Possible values are *-o table*, *-o csv*, *-o hdr* and *-o both*. *Table* reports the results in a format that is easy for humans to read, but inconvenient to process by machine. *Csv* reports the results in a comma separated value format that is convenient for machines to read and process, but inconvenient for humans. *Hdr* prints only the header for *csv* format and exits. *Both* prints both the header and the values in *csv* format. The default value is *table*.

-p This option (*--pool*) describes the size of each of the source and destination pools in bytes. The syntax is *-p <number>* where *<number>* is an octal, decimal or hexadecimal number

as described above. The default value is 4M (2^{22}) bytes per pool. The choice of pool size determines whether the copy operation takes place in some level of cache or in memory.

-s This option (*--seconds*) describes the number of seconds each experiment will run. The syntax is *-s <seconds>* where *<seconds>* is a positive decimal number with an optional decimal fraction. For example, *-s 1.5* represents an experiment duration of 1.5 seconds. This option may not be used with the *-i* (*--iterations*) option. The default value is 1 second.

-t This option (*--threads*) describes the number of threads to be used. The syntax for this option is *-t <threads>* where *<threads>* is a positive integer representing the number of threads to be used. This option may be used with the *-n gen* or *-n local* options but not the *-n map* option.

The output of this benchmark is somewhat complex, although most of the values are intended to specify input parameters in order to allow the result to be reproduced. The output fields are as follows.

copy method - The routine used to copy data in memory from the *-m* option, e.g., `bcopy()` or `memcpy()`.

copy size - The size of the copy buffer as specified by the *-c* option.

pool size - The size of each source and destination pool as specified by the *-p* option.

test size - The total amount of memory allocated into pools by the entire test. It is the number of threads times the size of each pool times the number of pools per thread (2).

copies per thread - The number of copy operations per thread per iteration.

number of threads - The number of threads used in the test, as specified by the *-n* or *-t* options.

iterations - The number of iterations as specified by the *-i* option or as measured if *-s* was used. Note that when the number of threads is greater than one and *iterations* is measured, there is some small chance that the number of iterations measured by one thread will be slightly different than the number of iterations measured by another thread. This is resolved by using the number of iterations reported by thread 0. For tests of sufficient length any inaccuracies this introduces will be negligible.

experiments - The number of experiments as specified by the *-e* option.

numa placement - The suboption used with *-n* to specify the mapping of threads and pools to NUMA domains.

numa domains - The number of NUMA domains within the system as computed from the *-d* option or as measured if *-L numa* was used.

domain map - The precise mapping of threads and pools to domains. This mapping is complete and can be used with the *-n map* option to recreate the original domain mapping.

total copies - The total number of copy operations performed by all threads over all iterations. This is also equal to *copies per thread* x *number of threads* x *iterations*.

total bytes copied - The total number of bytes copied in all copy operations from all threads. This is also equal to *total copies* multiplied by *copy size*.

elapsed time (seconds) - Total elapsed (wall clock) time in seconds over the measured portion of the test. Whether the *-s* option or the *-i* option is used, this time is the measured time. It may be slightly longer that the number of seconds specified by *-s* because the test only completes on whole iterations. Time is measured in time stamp counter ticks and converted to seconds.

elapsed time (timer ticks) - Total elapsed (wall clock) time in timer ticks over the measured portion of the test. Time is measured and used in timer ticks, that is, units of the time stamp counter. These units are usually processor clocks with an accuracy ranging from about 1 nanosecond to about 16 nanoseconds, depending on the architecture and compiler.

clock resolution (ns) - The minimum accuracy in nanoseconds of the routine used to measure time in the system.

read bandwidth - The measured throughput (not bandwidth), in megabytes (2^{20} bytes) per second, used to read data from memory. This value should be the total bytes copied divided by the elapsed time and should match the *write bandwidth* field for memory copy operations such as *bcopy* and equal zero for memory initialization operations such as *bzero*.

write bandwidth - The measured throughput (not bandwidth), in megabytes (2^{20} bytes) per second, used to write data to memory. This value should be the total bytes copied divided by the elapsed time.

pchase

The *pchase* benchmark measures the latency of memory references. It does so by creating a chain of pointers that are randomly scattered throughout a pool of memory, then repeatedly following those pointers in a timed loop. The reason for using a chain of pointers instead of something else is that the processor cannot begin to fetch the next reference to memory until the current reference is completely resolved. In that way, prefetch engines with deep queues and wide paths to memory, features that improve bandwidth but not latency, do not distort the latency measurement.

This benchmark is useful for measuring both unloaded and loaded latency. Unloaded latency is measured when there is only a single memory reference in the system at any time during the benchmark run. It gives the best possible latency throughout the system because there is no other activity to compete or interfere with its completion. It is useful but also, to some degree at least, an unrealistic measure, because servers are used as throughput engines, loaded down with as much work at a time as they can productively do.

pchase is able to measure loaded latency in several ways. The first, and generally most useful method, is to load the system with multiple threads, each chasing pointer chains. The user may control the size and placement of each thread and chain to carefully measure whatever path, or combination of paths, through the system is desired. A second way to measure loaded latency is to increase the number of chains being referenced concurrently within a single thread. Of course, a combination of the two can also be used.

Several additional factors influence the behavior and performance of a system. Perhaps the best known is the cache hierarchy, its cache line size, depth (number of levels of cache), associativity and size of each level. *pchase* provides options to describe the size of a cache line, which is used internally to avoid fetching data from the same cache line more than once. Setting the chain size to fit within some level of cache allows the user to test the latency of that level.

Another factor is the translation look-aside buffer, or TLB. This feature provides a cache of sorts for the virtual address translation mechanism of the memory management unit, or MMU. To illustrate, when the processor presents a virtual address to the MMU, the MMU must somehow convert the virtual address to a physical address. When the mapping is cached, the MMU can respond immediately with the corresponding physical address. When it is not cached, the MMU must make multiple references to memory (typically 2 for 32-bit architectures and 4 for 64-bit architectures) to complete the conversion. The chain size can be adjusted to statistically assure references that always hit or always miss the TLB.

A third factor is the DRAM page. Memory DIMMs are typically made from one or more DRAM chips, each of which is organized into what are called DRAM pages. Memory controllers can use these DRAM pages to speed up access to memory. In order to read or write a location in memory, the DRAM page containing that location must first be opened, accessed, and eventually closed. But a DIMM can only support a small number of pages being open at a time, so the memory controller must implement a DRAM page policy. Typical policies are DRAM page open, where the controller leaves the page open until forced to close it due to competing references, and DRAM page closed, where the controller opens the page, makes the access and immediately closes the page. Because opening and closing pages takes time, page open policies are useful when many references take place within a page. Page closed policies are more useful when references within a short period of time are generally scattered across many pages.

Of course, whether memory is placed within a local or remote NUMA domain also affects performance, and options are supported to control the mapping of chains to domains.

With this description of memory in mind, this is how *pchase* chains are constructed. When a chain is allocated it is first partitioned into cache lines. Only one pointer within a chain is ever placed within a given cache line, and two chains never overlap. This guarantees a cache line is only used once during a chain traversal. Next, the chain is partitioned into pages according to the specified page size. An unused page is selected at random from the chain, then cache lines are selected at random within the selected page. As each cache line is selected it is linked into the pointer chain.

This continues until all cache lines within a page have been selected, after which the next page is selected at random. In this way, memory references always cover a page first before moving to the next page. When this behavior is not desired, the page size can be set to equal the chain size.

This benchmark supports many command line options for controlling benchmark behavior, as follows.

-L This option (--*library*) selects which facilities are to be used internally to map processors and data to NUMA domains. *-L numa* specifies that *libnuma* will be used. *-L sched* specifies that `sched_setaffinity()` will be used. When *-L sched* is specified, the *-d* option must also be used to specify the mapping of threads to NUMA domains.

-d This option (--*domains*) describes the mapping of threads to NUMA domains. It may only be used in conjunction with the *-L sched* option. The syntax for this option is *-d <list>*, where *<list>* is a comma separated list of NUMA domains where each corresponding logical processor core resides. Logical cores and NUMA domains start at zero and count upwards. For example, a list of 0,0,0,0,1,1,1,1 indicates that there are eight logical cores and two NUMA domains in the system - eight cores because there are eight values in the list, two NUMA domains because the highest value in the list is 1. In this example, cores 0 through 3 all map to domain 0, and cores 4 through 7 map to domain 1.

-a This option (--*access*) describes the pattern used to select pages during the page selection process and cache lines during the line selection process. Three values are supported for this option, namely *random*, *forward*, and *backward*. *Random* indicates the pages and cache lines are selected randomly within the chain and is the default when no selection is given. Its syntax is *-a random*. As their names suggest, *forward* and *backward* select pages and cache lines sequentially in the indicated direction. With these two selections a stride is also required. Their syntax is *-a forward <stride>* and *-a backward <stride>*, where *<stride>* is a small positive integer. In every case the order in which cache lines will be selected is determined once, prior to the execution of the timed portion of the benchmark, and used throughout the remainder of the benchmark. It is not changed dynamically while the performance is being measured.

-c This option (--*chain*) gives the size of the pool of memory from which the pointer chains are taken. The syntax is *-c <number>* where *<number>* is a positive octal, decimal or hexadecimal integer. Numbers follow the standard C/C++ format, that is, octal numbers are preceded by 0 and fall in the range of 0 to 7, decimal numbers begin with a digit other than 0, and hexadecimal numbers are preceded by 0x and fall in the range of 0 through 9 or A through F. Additionally, decimal numbers can be followed by the postfix characters K, M, G, T, meaning the number is to be scaled by 2^{10}, 2^{20}, 2^{30} or 2^{40}, respectively. The default value for this parameter is 16MB, or 2^{24} bytes.

-e This option (--*experiments*) gives the number of experiments to be run. The syntax for this option is *-e <number>* where *<number>* is a positive octal, decimal or hexadecimal

integer, as above. When this value is greater than 1, only the *best* result of all experiments is reported. The default value is 1.

-i This option (*--iterations*) gives the number of iterations to be used in each experiment. The duration of any experiment may be specified either using the *-i* or *-s* option, but not both. The former specifies the number of iterations to use, the latter specifies the number of seconds each experiment is to run. The syntax for this option is *-i <number>* where *<number>* is a positive octal, decimal or hexadecimal integer, as above. The default is to run for 1 second, as with the *-s* option described below.

-l This option (*--line*) describes the size of a cache line, in bytes, within the system. The syntax for this option is *-l <size>* where *<size>* is a small positive octal, decimal or hexadecimal number.

-n This option (*--numa*) describes the placement of data relative to the thread that operates on it. Each thread uses two pools of memory, one for the source of the copy, one for the destination (or target). So, each thread must have three NUMA domains specified: the domain where the thread resides, the domain where the source pool resides and the domain where the destination pool resides. This option allows the user to specify each of these values in several ways.

First, *-n local* specifies that threads will be allocated round robin among domains, and the pointer chains will be local to, or reside where, the thread resides.

Second, *-n gen <thread> <chain>* specifies that the mapping for the thread and pointer chain will be according to the expressions *thread* and *chain*, respectively. Each expression gives the domain ID as a value relative to the thread number. Threads and domains are both numbered consecutively starting at zero. A constant value, such as 0 or 1 means that the thread or chain is to be affinitized to that domain, in this example, 0 or 1. So a specification of *-n gen 0 0* would map all threads and pointer chains to domain 0. Expressions may also be a constant with a leading '+' or '-', which indicates the domain is the thread ID plus (or minus) the constant value modulo the number of domains in the system. For example, consider a system with only two NUMA domains, 0 and 1. In this case an expression such as +1 would indicate the domain would be the thread ID plus 1 modulo 2. Independent of the number of domains within a system, one commonly used specification is *-n gen +0 +0* for distributing all threads evenly across the domains and placing the chains locally (the same as *-n local* above). Another useful specification is *-n gen +0 +1* for distributing all threads evenly, as before, but placing the pointer chain remotely. In other words, it measures the latency of chasing remote pointers. Note that this option, *-n gen*, only supports a single pointer chain per thread. If more than one chain per thread is needed, another option must be used.

Third, a much greater degree of control can be expressed using the *map* specification. The *map* syntax is *-n map <map>*. For this specification, *<map>* is a list of thread domain specifications for each thread and pointer chain, separated by semicolons. A thread domain specification uses the form t_i:c_{i0},c_{i1},c_{i2},...,c_{ik} where t_i is the domain where thread *i* resides and c_{ij} is the domain where the j^{th}chain of thread *i* resides. Only constant integers are allowed for the values *t* and *c*. For example, the specification *-n map 0:0;1:1;2:2;3:3* indicates there are four threads (because there are four thread domain specifications) and one chain per thread. Thread 0 is affinitized to domain 0, thread 1 is on domain 1, thread 2 is on domain 2 and thread 3 is on domain 3. Furthermore, the pointer chains happen to be mapped to the same domain where the thread resides. If a thread or chain domain exceeds the maximum domain it is wrapped around using a modulo function. Also, the number of chains must be the same for all threads.

Two additional suboptions, *-n xor <mask>* and *-n add <offset>*, exist but have been deprecated. The former performs an exclusive-or operation with the thread ID and *<mask>* to form the domain. The latter adds an offset to the thread ID to do the same. Both operations are modulo the number of NUMA domains.

-o This option (--*output*) describes the output format. Possible values are *-o table*, *-o csv*, *-o hdr* and *-o both*. *Table* reports the results in a format that is easy for humans to read, but inconvenient to process by machine. *Csv* reports the results in a comma separated value format that is convenient for machines to read and process, but inconvenient for humans. *Hdr* prints only the header for *csv* format and exits. *Both* prints both the header and the values in *csv* format. The default value is *table*.

-p This option (--*page*) describes the size, in bytes, of a page. The syntax is *-p <number>* where *<number>* is an octal, decimal or hexadecimal number as described above. The default page size is 4K (4096) bytes. The choice of page size can be used to explore DRAM page policies or TLB performance and size.

-r This option (--*references*) describes the number of chains to be used per thread. When one chain is used, the benchmark kernel code is simply "`a=*a;`". With two chains the code is expanded to "`a=*a; b=*b;`", and so forth. Current superscalar processors will perform such independent loads concurrently. This option may be used with *-n local* but not with *-n map* or *-n gen*.

-s This option (--*seconds*) describes the number of seconds each experiment will run. The syntax is *-s <seconds>* where *<seconds>* is a positive decimal number with an optional decimal fraction. For example, *-s 1.5* represents an experiment duration of 1.5 seconds. This option may not be used with the *-i* (--*iterations*) option. The default value is 1 second.

-t This option (--*threads*) describes the number of threads to be used. The syntax for this option is *-t <threads>* where *<threads>* is a positive integer representing the number of threads to be used. This option may be used with the *-n gen* or *-n local* options but not the *-n map* option.

This benchmark has many output values, but most of the values are intended to specify input parameters in order to allow the result to be reproduced. The output fields are as follows.

pointer size - The size of the pointer used for this benchmark. For 32-bit architectures it is four bytes. Four 64-bit architectures it is eight bytes.

cache line size - The cache line size as it is used within the benchmark. This value comes from the *-l* command line option and is not taken directly from the system. The default value is 64 bytes.

page size - The page size as it is used within the system. This value comes from the *-p* command line option. All cache lines within a page are exhausted before selecting a cache line within another page. This feature can be used to measure the performance impact of DRAM pages, page sizes within an operating system or TLB coverage. The default size is 4K (4096) bytes.

chain size - The amount of memory, in bytes, covered by a single pointer chain. This value comes from the *-c* command line option.

thread size - The amount of memory, in bytes, covered by all pointer chains within a single thread. It is equal to the number of chains per thread (*-r*) multiplied by the chain size (*-c*). It reflects the amount of data that a thread will use as its working set and therefore what a core will try to keep in its local cache.

test size - The total chain space, in bytes, covered by all chains within all threads. It is equal to the number of threads (*-t*) multiplied by the number of chains per thread (*-r*) multiplied by the chain size (*-c*). In systems with a single top level shared cache, such as some single processor socket systems, this reflects the working set of the benchmark and therefore the amount of data the benchmark is attempting to retain in its top level cache.

chains per thread - The number of chains being referenced concurrently by a thread. This value comes from the *-r* command line option.

number of threads - The total number of concurrent threads executing the benchmark. This value comes from the *-t* command line option.

iterations - The number of iterations as specified by the *-i* option or as measured if *-s* was used. Note that when the number of threads is greater than one and *iterations* is measured, there is some small chance that the number of iterations measured by one thread will be slightly different than the number of iterations measured by another thread. This is resolved by using the number of iterations reported by thread 0. For tests of sufficient length any inaccuracies this introduces will be negligible.

experiments - The number of experiments as specified by the *-e* option.

access pattern - The suboption used with *-a* to specify the manner in which pointers were allocated to chains. It is one of *random, forward* or *backward*.

stride - The stride used for selecting the next chain. If access (-*a*) was *forward* or *backward* the value is a positive integer. The value is 1 if the access was *random*.

numa placement - The suboption used with -*n* to specify the mapping of threads and chains to NUMA domains.

offset or mask - The additional value used with the deprecated -*n* suboptions *xor* and *add*.

numa domains - The number of NUMA domains within the system as computed from the -*d* option or as measured if -*L numa* was used.

domain map - The precise mapping of threads and pools to domains. This mapping is complete and can be used with the -*n map* option to recreate the original domain mapping.

operations per chain - The number of pointer links, or address loads, in a single chain. When the value of the -*l* option correctly reflects the cache line size of the system, this value also reflects the number of cache line loads per chain.

total operations - The total number of address loads per iteration performed in the benchmark. It is equal to *operations per chain* multiplied by *chains per thread* multiplied by *number of threads*.

elapsed time - Total elapsed (wall clock) time in seconds over the measured portion of the test. Whether the -*s* option or the -*i* option is used, this time is the measured time. It may be slightly longer that the number of seconds specified by -*s* because the test only completes on whole iterations. Time is measured in time stamp counter ticks and converted to seconds.

elapsed time - Total elapsed (wall clock) time in timer ticks over the measured portion of the test. Time is measured and used in timer ticks, that is, units of the time stamp counter. These units are usually processor clocks with an accuracy ranging from about 1 nanosecond to about 16 nanoseconds, depending on the architecture and compiler.

clock resolution - The minimum accuracy of the routine used to measure time in the system.

memory latency - The average time in nanoseconds to traverse a single link in the pointer chain. It is computed as *elapsed time* divided by (*iterations* multiplied by *chain size*), and scaled appropriately.

memory bandwidth - The throughput (effective bandwidth) of the system for this type of access. If access (-*a*) is *random*, it will reflect the random throughput of the system. It is computed as (*chain size* multiplied by *iterations* multiplied by *chains per thread* multiplied by *number of threads* multiplied by *cache line size*) divided by *elapsed time* and scaled to reflect megabytes (10^6) per second.

This value is only accurate when the *-l* value accurately reflects the effective cache line size. For *x86*, the cache line size is 32 bytes. For *x86-64*, the cache line size is 64 bytes. Other architectures may have other sizes of cache lines. Furthermore, on some architectures certain BIOS settings, such as second sector prefetch, can change the effective cache line size and render this value inaccurate.

sha

This benchmark measures the performance of a selection of the Secure Hash Algorithm (SHA) hash functions. It uses a highly optimized version of the 160-bit SHA-1 hash function and several implementations, each optimized to varying degrees, of the 256-bit SHA-2 function. The benchmark allows the user to specify the size of message to be hashed and the size and placement of pools the messages are taken from. The SHA code also makes use of an internal memory copy routine, and the user is allowed to specify which memory copy routine is used.

The full list of hash routines is as follows:
1. *sha1*, a highly optimized version of the 160-bit SHA-1 hash function.
2. *sha256g*, a version of the 256-bit SHA-2 hash function that is designed to be vectorizable with a high-performance vectorizing compiler. Its performance is low for non-vectorizing compilers.
3. *sha256s*, a scalar, partially optimized version of the 256-bit SHA-2 hash function. Substitutions of the x86 ROR instruction for the SHA ROTR (rotate right) operation are included with this version, but otherwise this implementation is straght from the U.S. Department of Commerce algorithm description.
4. *sha256u*, a scalar, optimized version of the 256-bit SHA-2 hash function. Substitutions of the x86 ROR instruction for the SHA ROTR (rotate right) operation are included with this version and the main computational loop has been unrolled.
5. *sha256v*, a vector, partially optimized version of the 256-bit SHA-2 hash function. The data types have been changed from `uint32_t` to `v4si` and appropriate vector operations have been inserted where needed, but this version is otherwise very similar to *sha256s*.
6. *sha256x*, a vectorized and optimized version of the 256-bit SHA-2 hash function. Data types have been changed from `uint32_t` to `v4si` and appropriate vector operations have been inserted where needed, and the main computational loop has been unrolled.

The interface is such that four buffers per thread may be hashed simultaneously to achieve the maximum performance from the processor core. All messages and message pools are the same size for this benchmark, but four separate domain locations must be specified. A list of pointers to each message buffer is generated and the list is ordered or randomized according to the access option (*-a*).

Once the timer is started, each thread walks through its own list, hashing every message in the list. At the end of the list, the number of iterations (*-i*) or the elapsed time (*-s*) is checked and if not complete, the list is walked again. The timer is started after a barrier and a barrier is executed just before the final timer completes, but there is no barrier before or after intermediate tests for completion.

-L This option (--*library*) selects which facilities are to be used internally to map processors and data to NUMA domains. *-L numa* specifies that *libnuma* will be used. *-L sched* specifies that `sched_setaffinity()` will be used. When *-L sched* is specified, the *-d* option must also be used to specify the mapping of threads to NUMA domains.

-d This option (--*domains*) describes the mapping of threads to NUMA domains. It may only be used in conjunction with the *-L sched* option. The syntax for this option is *-d <list>*, where *<list>* is a comma separated list of NUMA domains where each corresponding logical processor core resides. Logical cores and NUMA domains start at zero and count upwards. For example, a list of 0,0,0,0,1,1,1,1 indicates that there are eight logical cores and two NUMA domains in the system - eight cores because there are eight values in the list, two NUMA domains because the highest value in the list is 1. In this example, cores 0 through 3 all map to domain 0, and cores 4 through 7 map to domain 1.

-a This option (--*access*) describes the pattern used to select blocks to digest. Three values are supported for this option, namely *random*, *forward*, and *backward*. *Random* indicates the blocks are selected randomly within the pool and is the default when no selection is given. As their names suggest, *forward* and *backward* select blocks sequentially in the indicated direction. In every case the order in which blocks will be selected is determined once, prior to the execution of the timed portion of the benchmark, and used throughout the remainder of the benchmark. It is not changed dynamically while the performance is being measured.

-b This option (--*bytes*) describes the number of bytes in the digest to be hashed. The syntax for this option is *-b <number>* where *<number>* is a positive octal, decimal or hexadecimal integer. Numbers follow the standard C/C++ format, that is, octal numbers are preceded by 0 and fall in the range of 0 to 7, decimal numbers begin with a digit other than 0, and hexadecimal numbers are preceded by 0x and fall in the range of 0 through 9 or A through F. Additionally, decimal numbers can be followed by the postfix characters K, M, G, T, meaning the number is to be scaled by 2^{10}, 2^{20}, 2^{30} or 2^{40}, respectively. The default value for this parameter is 64 bytes.

-c This option (--*copy*) specifies the memory copy to be used internally within the hash routine. The syntax for this option is *-c memcpy* or *-c copy_tt* or *-c copy_tn*, where *memcpy* specifies the libc `memcpy()` routine, *copy_tt* specifies a hand coded routine using temporal (caching) loads and stores, and *copy_tn* specifies a hand coded routine using temporal loads and non-temporal (non-caching) stores.

-e This option (--*experiments*) gives the number of experiments to be run. The syntax for this option is *-e <number>* where *<number>* is a positive octal, decimal or hexadecimal integer, as above. When this value is greater than 1, only the *best* result of all experiments is reported. The default value is 1.

-i This option (--*iterations*) gives the number of iterations to be used in each experiment. The duration of any experiment may be specified either using the *-i* or *-s* option, but not both. The former specifies the number of iterations to use, the latter specifies the number

of seconds each experiment is to run. The syntax for this option is *-i <number>* where *<number>* is a positive octal, decimal or hexadecimal integer, as above. The default is to run for 1 second, as with the *-s* option described below.

-m This option (*--method*) specifies the method to be used to digest the message block. The syntax for this option is *-m sha1*. Currently only the 160-bit SHA1 algorithm is supported.

-n This option (*--numa*) describes the placement of data relative to the thread that operates on it. Each thread uses one pool of memory for the synchronization objects it will access. So, each thread must have two NUMA domains specified: the domain where the thread resides and the domain where the synchronization object pool resides. This option allows the user to specify each of these values in several ways.

First, *-n local* specifies that threads will be allocated round robin among domains and the synchronization pools will be local to, or reside where, the threads reside.

Second, *-n gen <thread> <h0> <h1> <h2> <h3>* specifies that the mapping for the thread and message pools will be according to the expressions *thread, h0, h1, h2* and *h3*, respectively. Each expression gives the domain ID as a value relative to the thread number. Threads and domains are both numbered consecutively starting at zero. A constant value, such as 0 or 1 means that the thread or pool is to be affinitized to that domain, in this example, 0 or 1. So a specification of *-n gen 0 0 0 0 0* would map all threads and message pools to domain 0. Expressions may also be a constant with a leading '+' or '-', which indicates the domain is the thread ID plus (or minus) the constant value modulo the number of domains in the system. For example, consider a system with only two NUMA domains, 0 and 1. In this case an expression such as +1 would indicate the domain would be the thread ID plus 1 modulo 2. Independent of the number of domains within a system, one commonly used specification is *-n gen +0 +0 +0 +0 +0* for distributing all threads evenly across the domains and placing the pools locally (the same as *-n local* above). Another useful specification is *-n gen +0 +1 +1 +1 +1* for distributing all threads evenly, as before, but placing the message pools remotely. In other words, it measures the throughput of hashing remote messages.

Third, a much greater degree of control can be expressed using the *map* specification. The *map* syntax is *-n map <map>*. For this specification, *<map>* is a list of thread domain specifications for each thread and pool, separated by semicolons. A thread domain specification uses the form *t:h0,h1,h2,h3* where *t* is the domain where the thread resides and *h* is the domain where a message pool resides. Only constant integers are allowed for the values *t* and *h*. The specification *-n map 0:0,0,0,0;1:1,1,1,1;2:2,2,2,2;3:3,3,3,3* indicates there are four threads (because there are four thread domain specifications). Thread 0 is affinitized to domain 0, thread 1 is on domain 1, thread 2 is on domain 2 and thread 3 is on domain 3. Furthermore, the message pools happen to be mapped to the same domain where the thread resides. If a thread or pool domain exceeds the maximum domain it is wrapped around using a modulo function.

-o This option (*--output*) describes the output format. Possible values are *-o table*, *-o csv*, *-o hdr* and *-o both*. *Table* reports the results in a format that is easy for humans to read, but inconvenient to process by machine. *Csv* reports the results in a comma separated value format that is convenient for machines to read and process, but inconvenient for humans. *Hdr* prints only the header for *csv* format and exits. *Both* prints both the header and the values in *csv* format. The default value is *table*.

-p This option (*--pool*) specifies the number of bytes to be allocated to each message pool. The syntax for this option is *-p <number>* where *<number>* is a positive octal, decimal or hexadecimal integer, as above. The default value is 4M (2^{22}) bytes.

-s This option (*--seconds*) describes the number of seconds each experiment will run. The syntax is *-s <seconds>* where *<seconds>* is a positive decimal number with an optional decimal fraction. For example, *-s 1.5* represents an experiment duration of 1.5 seconds. This option may not be used with the *-i* (*--iterations*) option. The default value is 1 second.

-t This option (*--threads*) describes the number of threads to be used. The syntax for this option is *-t <threads>* where *<threads>* is a positive integer representing the number of threads to be used. This option may be used with the *-n gen* or *-n local* options but not the *-n map* option.

There are many output values for this benchmark. They are as follows:

hash method - The method used to hash each message, as specified by the *-m* option.

copy method - The internal copy routine used by the hash method, as specified by the *-c* option.

hash size (bytes) - The size in bytes of the message to be hashed, as specified by the *-b* option.

pool size (bytes) - The actual size in bytes of each pool from which messages are taken. This value is specified by the *-p* option, but the actual value may be rounded up if the size specified is not an integer multiple of the hash size (*-b*).

test size (bytes) - The total number of bytes allocated to message pools for all threads in this benchmark. It is computed as *pool size* multiplied by *number of threads* multiplied by *pools per thread*.

hashes per pool - The number of message buffers, or hashes to be performed, per message pool. This is computed as *pool size* divided by *hash size*.

pools per thread - The number of message pools per thread.

hashes/thread/iteration - The number of hash operations per thread per iteration. This is computed as *hashes per pool* multiplied by *pools per thread*.

number of threads - The number of threads used in the test, as specified by the *-n* or *-t* options.

iterations - The number of iterations as specified by the *-i* option or as measured if *-s* was used. Note that when the number of threads is greater than one and *iterations* is measured, there is some small chance that the number of iterations measured by one thread will be slightly different than the number of iterations measured by another thread. This is resolved by using the number of iterations reported by thread 0. For tests of sufficient length any inaccuracies this introduces will be negligible.

experiments - The number of experiments as specified by the *-e* option.

numa placement - The suboption for *-n* used to generate the distribution of threads and message pools.

numa domains - The number of NUMA domains within the system as computed from the *-d* option or as measured if *-L numa* was used.

domain map - The precise mapping of threads and pools to domains. This mapping is complete and can be used with the *-n map* option to recreate the original domain mapping.

total hashes - The total number of hash operations completed during the experiment. It is computed as *iterations* multiplied by *hashes per pool* multiplied by *pools per thread* multiplied by *number of threads*.

total bytes hashed - The total number of bytes hashed during the experiment. It is computed as *iterations* multiplied by *hashes per pool* multiplied by *pools per thread* multiplied by *number of threads* multiplied by *hash size*.

elapsed time (seconds) - Total elapsed (wall clock) time in seconds over the measured portion of the test. Whether the *-s* option or the *-i* option is used, this time is the measured time. It may be slightly longer that the number of seconds specified by *-s* because the test only completes on whole iterations. Time is measured in time stamp counter ticks and converted to seconds.

elapsed time (timer ticks) - Total elapsed (wall clock) time in timer ticks over the measured portion of the test. Time is measured and used in timer ticks, that is, units of the time stamp counter. These units are usually processor clocks with an accuracy ranging from about 1 nanosecond to about 16 nanoseconds, depending on the architecture and compiler.

clock resolution (ns) - The minimum accuracy in nanoseconds of the routine used to measure time in the system.

hash bandwidth (MiB/s) - The rate at which hashes are computed, in megabytes (2^{20}) per second. It is computed as *total bytes hashed* divided by *elapsed time* and scaled to reflect megabytes per second.

stream

The *stream* benchmark measures memory throughput using four common computational kernels. It was originally created by John McCalpin, then a professor at the University of Virginia. The home page for the *stream* benchmark is http://www.cs.virginia.edu/stream/.

The four kernels are *copy*, *scale*, *add* and *triad*. Each kernel treats all data as double-precision floating-point values. The *copy* operation is a vector copy, or `a[*]=b[*]`; *scale* is a vector copy operation mixed with a scalar multiply, or `a[*]=s*b[*]`; *add* is a vector add operation, or `a[*]=b[*]+c[*]`; and *triad* is a vector add combined with a scalar multiplication, or `a[*]=b[*]+s*c[*]`.

This benchmark uses a few coding tricks to get the most performance out of memory and, as a result, is highly sensitive to choice of compiler and compiler options. Specifically, all arrays used are defined as global variables within the scope of the timed routine. Having a constant base address and bounds for the arrays allows the compiler to eliminate a few load operations within the inner loop that could not be avoided if the array were passed in as a function parameter. Knowing the size of the array also allows the compiler to use non-temporal (non-caching) data stores. Other compiler optimizations, such as common subexpression elimination and invariant expression migration shorten the path between loads, so they also have a significant impact on performance. For this reason, the choice of compiler and compiler optimizations has a marked effect on the results.

This benchmark relies heavily on OpenMP as its implementation to exploit parallelism within the system. OpenMP creates a pool of threads upon start-up and parks them in such a way that they can be activated quickly for loops that have exploitable parallelism. The threads wait at a barrier until the loop is ready for execution, then return to a parked state when there is no more work to be done. This is important because the loop that forms the computational kernel is not considered complete by OpenMP or the benchmark until the final thread completes. Because of load balancing it means that the benchmark may run faster when the number of threads is a multiple of the number of domains in the system.

For example, consider a system with two domains and two cores per domain. Running the benchmark with two threads gives each domain half of the total available work. Running it with three threads gives one domain one thread, or one third of the work, and the other domain two threads, or two thirds of the work. Even though there are enough cores in each domain to support the computational part of the load, if the system is limited by the memory throughput the domain with one thread will complete its work before the other domain and have to sit idle while it waits. This illustrates how increasing the number of threads may not always increase the performance of this benchmark even when there is an abundance of cores available to execute it.

There are no command line arguments to this benchmark, but there is one environment variable, `OMP_NUM_THREADS`, that controls the number of threads OpenMP will use during execution.

The benchmark runs each operation (*copy*, *scale*, *add*, *triad*) ten times. It reports the rate in megabytes (10^6 bytes) per second for the best time for that operation. It also reports the minimum time, the maximum time and the root mean square (or square root of the sum of the squares) of the times for each operation.

Rate - The *best* rate at which memory was transferred, in megabytes (10^6 bytes) per second, for each operation. This rate includes both data loads and stores.

RMS time - The root mean square time, or the square root of the sum of the squares of the times for each operation.

Min time - The minimum time for each operation.

Max time - The maximum time for each operation.

sync-c

This benchmark is experimental, a work in progress. It measures the time to access and release a single synchronization resource under contention. The benchmark controls the number of threads to be used. The synchronization object is always placed on domain 0. All threads wait at a barrier before accessing the resource.

One reason the benchmark is experimental is that the first thread out of the barrier often has an enormous advantage over all other threads and can execute several hundred thousand operations before the next thread begins to compete for the lock or other object. The second and all subsequent threads have more stable results. At the moment that advantage must be manually discounted to make the benchmark results more meaningful.

The full list of synchronization objects/routines is as follows:
1. *spin*, a POSIX spin lock
2. *mutex*, a POSIX mutex (sleeping lock)
3. *reader*, a POSIX reader/writer lock in reader mode
4. *writer*, a POSIX reader/writer lock in writer mode
5. *cmpxchg32*, an x86 32-bit cmpxchg (compare-and-exchange) instruction
6. *cmpxchg64*, an x86 64-bit cmpxchg instruction
7. *cmpxspin*, a simple hand-coded spin lock built from x86 cmpxchg instructions
8. *fairspin*, a simple hand-coded fair spin lock built from x86 cmpxchg instructions
9. *atomic32*, a simple 32-bit atomic increment function built from x86 cmpxchg instructions
10. *atomic64*, a simple 64-bit atomic increment function built from x86 cmpxchg instructions

This benchmark supports many options to control various aspects of its inner workings. They are:

-*L* This option (--*library*) selects which facilities are to be used internally to map processors and data to NUMA domains. -*L numa* specifies that *libnuma* will be used. -*L sched* specifies that `sched_setaffinity()` will be used. When -*L sched* is specified, the -*d* option must also be used to specify the mapping of threads to NUMA domains.

-d This option (--*domains*) describes the mapping of threads to NUMA domains. It may only be used in conjunction with the *-L sched* option. The syntax for this option is *-d <list>*, where <*list*> is a comma separated list of NUMA domains where each corresponding logical processor core resides. Logical cores and NUMA domains start at zero and count upwards. For example, a list of 0,0,0,0,1,1,1,1 indicates that there are eight logical cores and two NUMA domains in the system - eight cores because there are eight values in the list, two NUMA domains because the highest value in the list is 1. In this example, cores 0 through 3 all map to domain 0, and cores 4 through 7 map to domain 1.

-e This option (--*experiments*) gives the number of experiments to be run. The syntax for this option is *-e* <*number*> where <*number*> is a positive octal, decimal or hexadecimal integer. Numbers follow the standard C/C++ format, that is, octal numbers are preceded by 0 and fall in the range of 0 to 7, decimal numbers begin with a digit other than 0, and hexadecimal numbers are preceded by 0x and fall in the range of 0 through 9 or A through F. Additionally, decimal numbers can be followed by the postfix characters K, M, G, T, meaning the number is to be scaled by 2^{10}, 2^{20}, 2^{30} or 2^{40}, respectively. When this value is greater than 1, only the *best* result of all experiments is reported. The default value is 1.

-i This option (--*iterations*) gives the number of iterations to be used in each experiment. The duration of any experiment may be specified either using the *-i* or *-s* option, but not both. The former specifies the number of iterations to use, the latter specifies the number of seconds each experiment is to run. The syntax for this option is *-i* <*number*> where <*number*> is a positive octal, decimal or hexadecimal integer, as above. The default is to run for 1 second, as with the *-s* option described below.

-m This option (--*method*) describes the type of synchronization object to be used. The full list of possible synchronization objects is listed above. The default type is *spin*, which is a POSIX spin lock.

-n This option (--*numa*) describes the placement of threads within the system. Each thread accesses only a single synchronization object that is allocated on domain 0. So, each thread must have only one NUMA domain specified: the domain where the thread resides. This option allows the user to specify each of these values in several ways.

First, *-n local* specifies that threads will be allocated round robin among domains.

Second, *-n gen* <*thread*> specifies that the mapping for the thread will be according to the expression *thread*. The expression gives the domain ID as a value relative to the thread number. Threads and domains are both numbered consecutively starting at zero. A constant value, such as 0 or 1 means that the thread or pool is to be affinitized to that domain, in this example, 0 or 1. So a specification of *-n gen 0* would map all threads to domain 0, local to the object. Similarly, a specification of *-n gen 1* would map all threads to domain 1, or remote to the object. Expressions may also be a constant with a leading '+' or '-', which indicates the domain is the thread ID plus (or minus) the constant value modulo the number of domains in the system. For example, consider a system with only two NUMA domains, 0 and 1. In this case an expression such as +1 would indicate the domain would be the thread ID plus 1 modulo 2. Independent of the number of domains within a system, one commonly used specification is *-n gen +0* for distributing all threads evenly across the domains and placing the pools locally (the same as *-n local* above).

Third, a much greater degree of control can be expressed using the *map* specification. The *map* syntax is *-n map <map>*. For this specification, *<map>* is a list of thread domain specifications for each thread, separated by semicolons. A thread domain specification uses the single value *t* where *t* is the domain where the thread resides. Only constant integers are allowed for the values *t*. For example, the specification *-n map 0;1;2;3* indicates there are four threads (because there are four thread domain specifications). Thread 0 is affinitized to domain 0, thread 1 is on domain 1, thread 2 is on domain 2 and thread 3 is on domain 3. If a thread domain exceeds the maximum domain it is wrapped around using a modulo function.

-o This option (*--output*) describes the output format. Possible values are *-o table*, *-o csv*, *-o hdr* and *-o both*. *Table* reports the results in a format that is easy for humans to read, but inconvenient to process by machine. *Csv* reports the results in a comma separated value format that is convenient for machines to read and process, but inconvenient for humans. *Hdr* prints only the header for *csv* format and exits. *Both* prints both the header and the values in *csv* format. The default value is *table*.

-s This option (*--seconds*) describes the number of seconds each experiment will run. The syntax is *-s <seconds>* where *<seconds>* is a positive decimal number with an optional decimal fraction. For example, *-s 1.5* represents an experiment duration of 1.5 seconds. This option may not be used with the *-i* (*--iterations*) option. The default value is 1 second.

-t This option (*--threads*) describes the number of threads to be used. The syntax for this option is *-t <threads>* where *<threads>* is a positive integer representing the number of threads to be used. This option may be used with the *-n gen* or *-n local* options but not the *-n map* option.

-z This option (*--delay*) describes the number of processor clock ticks a lock object is to be held before being released, or the number of processor clock ticks to delay after performing an atomic update or *cmpxchg* instruction before accessing the object again.

Output for this benchmark is as follows.

op type - The type of synchronization object, as specified by the *-m* option.

number of threads - The number of threads used in the test, as specified by the *-n* or *-t* options.

experiments - The number of experiments as specified by the *-e* option.

numa domains - The number of NUMA domains within the system as computed from the *-d* option or as measured if *-L numa* was used.

domain map - The precise mapping of threads and pools to domains. This mapping is complete and can be used with the *-n map* option to recreate the original domain mapping.

delay (ticks) - The number of processor ticks to delay while holding a lock or after performing an atomic operation or *cmpxchg* instruction, as specified by the *-z* option.

elapsed time (seconds) - Total elapsed (wall clock) time in seconds over the measured portion of the test. Whether the *-s* option or the *-i* option is used, this time is the measured time. It may be slightly longer that the number of seconds specified by *-s* because the test only completes on whole iterations. Time is measured in time stamp counter ticks and converted to seconds.

elapsed time (timer ticks) - Total elapsed (wall clock) time in timer ticks over the measured portion of the test. Time is measured and used in timer ticks, that is, units of the time stamp counter. These units are usually processor clocks with an accuracy ranging from about 1 nanosecond to about 16 nanoseconds, depending on the architecture and compiler.

clock resolution (ns) - The minimum accuracy in nanoseconds of the routine used to measure time in the system.

min iterations - The least number of synchronization operations measured by any thread.

avg iters per thread - The average number of synchronization operations measured over all threads.

max iterations - The most number of synchronization operations measured by any thread. As mentioned above, the first thread out of the initial barrier often has a significant advantage and is able to lock and unlock the object several hundred thousand times before the second thread begins to compete.

total iterations - The total number of synchronization operations completed by all threads during this experiment.

op bandwidth (Mops/s) - The rate in millions (10^6) of ops per second at which synchronization operations are completed. It is computed as *total iterations* divided by *elapsed time*.

sync-u

This benchmark measures the time to access and release synchronization resources when *not* under contention. The benchmark allows the user to control the type, number and placement of synchronization objects and threads. Each thread builds a list of pointers to its own collection of synchronization objects, aligned on the size of boundary specified. For example, alignment on a 64-byte boundary means the address modulo 64 will be zero for every object in the list. The list of pointers is then randomized to prevent the cache controller from knowing from where to prefetch the next object.

Once the timer is started, each thread walks through its own list, accessing every object in the list. If the objects are locks, the object is locked and unlocked, cmpxchg instructions are executed exactly once, atomic increment functions are executed once, etc. At the end of the list, the number of iterations (*-i*) or the elapsed time (*-s*) is checked and if not complete, the list is walked again. The timer is started after a barrier and a barrier is executed just before the final timer completes, but there is no barrier before or after intermediate tests for completion.

The full list of synchronization objects/routines is as follows:

1. *spin*, a POSIX spin lock
2. *mutex*, a POSIX mutex (sleeping lock)
3. *reader*, a POSIX reader/writer lock in reader mode
4. *writer*, a POSIX reader/writer lock in writer mode
5. *cmpxchg32*, an x86 32-bit cmpxchg (compare-and-exchange) instruction
6. *cmpxchg64*, an x86 64-bit cmpxchg instruction
7. *cmpxspin*, a simple hand-coded spin lock built from x86 cmpxchg instructions
8. *fairspin*, a simple hand-coded fair spin lock built from x86 cmpxchg instructions
9. *atomic32*, a simple 32-bit atomic increment function built from x86 cmpxchg instructions
10. *atomic64*, a simple 64-bit atomic increment function built from x86 cmpxchg instructions

This benchmark supports many command line options for controlling benchmark behavior, as follows.

-L This option (*--library*) selects which facilities are to be used internally to map processors and data to NUMA domains. *-L numa* specifies that *libnuma* will be used. *-L sched* specifies that `sched_setaffinity()` will be used. When *-L sched* is specified, the *-d* option must also be used to specify the mapping of threads to NUMA domains.

-d This option (*--domains*) describes the mapping of threads to NUMA domains. It may only be used in conjunction with the *-L sched* option. The syntax for this option is *-d <list>*, where *<list>* is a comma separated list of NUMA domains where each corresponding logical processor core resides. Logical cores and NUMA domains start at zero and count upwards. For example, a list of 0,0,0,0,1,1,1,1 indicates that there are eight logical cores and two NUMA domains in the system - eight cores because there are eight values in the list, two NUMA domains because the highest value in the list is 1. In this example, cores 0 through 3 all map to domain 0, and cores 4 through 7 map to domain 1.

-a This option (*--align*) describes the alignment of each synchronization object. The syntax for this option is *-a <number>* where *<number>* is a positive octal, decimal or hexadecimal integer. Numbers follow the standard C/C++ format, that is, octal numbers are preceded by 0 and fall in the range of 0 to 7, decimal numbers begin with a digit other than 0, and hexadecimal numbers are preceded by 0x and fall in the range of 0 through 9 or A through F. Additionally, decimal numbers can be followed by the postfix characters K, M, G, T, meaning the number is to be scaled by 2^{10}, 2^{20}, 2^{30} or 2^{40}, respectively. The default value for this parameter is 64 bytes.

-e This option (--*experiments*) gives the number of experiments to be run. The syntax for this option is *-e <number>* where *<number>* is a positive octal, decimal or hexadecimal integer, as above. When this value is greater than 1, only the *best* result of all experiments is reported. The default value is 1.

-i This option (--*iterations*) gives the number of iterations to be used in each experiment. The duration of any experiment may be specified either using the *-i* or *-s* option, but not both. The former specifies the number of iterations to use, the latter specifies the number of seconds each experiment is to run. The syntax for this option is *-i <number>* where *<number>* is a positive octal, decimal or hexadecimal integer, as above. The default is to run for 1 second, as with the *-s* option described below.

-k This option (--*sync*) describes the number of synchronization objects to be used per thread. The syntax for this option is *-k <number>* where *<number>* is a positive octal, decimal or hexadecimal integer, as described above. Each iteration of an experiment will walk through all synchronization objects in a pool in a random order. The order is determined once, before the timed portion of the benchmark begins, and used throughout the rest of the benchmark. The default value for this option is 4M (2^{22}) objects per thread.

-m This option (--*method*) describes the type of synchronization object to be used. The full list of possible synchronization objects is listed above. The default type is *spin*, which is a POSIX spin lock.

-n This option (--*numa*) describes the placement of data relative to the thread that operates on it. Each thread uses one pool of memory for the synchronization objects it will access. So, each thread must have two NUMA domains specified: the domain where the thread resides and the domain where the synchronization object pool resides. This option allows the user to specify each of these values in several ways.

First, *-n local* specifies that threads will be allocated round robin among domains and the synchronization pools will be local to, or reside where, the threads reside.

Second, *-n gen <thread> <sync>* specifies that the mapping for the thread and synchronization object pools will be according to the expressions *thread* and *sync*, respectively. Each expression gives the domain ID as a value relative to the thread number. Threads and domains are both numbered consecutively starting at zero. A constant value, such as 0 or 1 means that the thread or pool is to be affinitized to that domain, in this example, 0 or 1. So a specification of *-n gen 0 0* would map all threads and synchronization object pools to domain 0. Expressions may also be a constant with a leading '+' or '-', which indicates the domain is the thread ID plus (or minus) the constant value modulo the number of domains in the system. For example, consider a system with only two NUMA domains, 0 and 1. In this case an expression such as +1 would indicate the domain would be the thread ID plus 1 modulo 2. Independent of the number of domains within a system, one commonly used specification is *-n gen +0 +0* for distributing all threads evenly across the domains and placing the pools locally (the same as *-n local* above). Another useful specification is *-n gen +0 +1* for distributing all threads evenly, as before, but placing the synchronization object pool remotely. In other words, it measures the throughput of accessing a remote lock or atomic variable.

Third, a much greater degree of control can be expressed using the *map* specification. The *map* syntax is *-n map <map>*. For this specification, *<map>* is a list of thread domain specifications for each thread and pool, separated by semicolons. A thread domain specification uses the form *t:s* where *t* is the domain where the thread resides and *s* is the domain where the synchronization object pool resides. Only constant integers are allowed for the values *t* and *s*. For example, the specification *-n map 0:0;1:1;2:2;3:3* indicates there are four threads (because there are four thread domain specifications). Thread 0 is affinitized to domain 0, thread 1 is on domain 1, thread 2 is on domain 2 and thread 3 is on domain 3. Furthermore, the synchronization pools happen to be mapped to the same domain where the thread resides. If a thread or pool domain exceeds the maximum domain it is wrapped around using a modulo function.

-o This option (*--output*) describes the output format. Possible values are *-o table*, *-o csv*, *-o hdr* and *-o both*. *Table* reports the results in a format that is easy for humans to read, but inconvenient to process by machine. *Csv* reports the results in a comma separated value format that is convenient for machines to read and process, but inconvenient for humans. *Hdr* prints only the header for *csv* format and exits. *Both* prints both the header and the values in *csv* format. The default value is *table*.

-s This option (*--seconds*) describes the number of seconds each experiment will run. The syntax is *-s <seconds>* where *<seconds>* is a positive decimal number with an optional decimal fraction. For example, *-s 1.5* represents an experiment duration of 1.5 seconds. This option may not be used with the *-i* (*--iterations*) option. The default value is 1 second.

-t This option (*--threads*) describes the number of threads to be used. The syntax for this option is *-t <threads>* where *<threads>* is a positive integer representing the number of threads to be used. This option may be used with the *-n gen* or *-n local* options but not the *-n map* option.

This benchmark reports many output values, but most of the values are intended to specify input parameters in order to allow the result to be reproduced. The output fields are as follows.

op type - The type of synchronization object, as specified by the *-m* option.

bytes allocated - The total number of bytes allocated per thread. This is calculated as the *ops per thread* (*-k*) multiplied by the actual size of the object, rounded up to the next multiple of the alignment size (*-a*).

ops per thread - The number of synchronization objects per thread as specified by the *-k* option.

number of threads - The number of threads used in the test, as specified by the *-n* or *-t* options.

iterations - The number of iterations as specified by the *-i* option or as measured if *-s* was used. Note that when the number of threads is greater than one and *iterations* is measured, there is some small chance that the number of iterations measured by one thread will be slightly different than the number of iterations measured by another thread. This is resolved by using the number of iterations reported by thread 0. For tests of sufficient length any inaccuracies this introduces will be negligible.

experiments - The number of experiments as specified by the *-e* option.

total ops - The total number of operations in the benchmark. This is computed as *ops per thread* multiplied by *number of threads* multiplied by *iterations*.

numa domains - The number of NUMA domains within the system as computed from the *-d* option or as measured if *-L numa* was used.

placement - The suboption used with *-n* to specify the mapping of threads and pools to NUMA domains.

domain map - The precise mapping of threads and pools to domains. This mapping is complete and can be used with the *-n map* option to recreate the original domain mapping.

elapsed time (seconds) - Total elapsed (wall clock) time in seconds over the measured portion of the test. Whether the *-s* option or the *-i* option is used, this time is the measured time. It may be slightly longer that the number of seconds specified by *-s* because the test only completes on whole iterations. Time is measured in time stamp counter ticks and converted to seconds.

elapsed time (timer ticks) - Total elapsed (wall clock) time in timer ticks over the measured portion of the test. Time is measured and used in timer ticks, that is, units of the time stamp counter. These units are usually processor clocks with an accuracy ranging from about 1 nanosecond to about 16 nanoseconds, depending on the architecture and compiler.

clock resolution (ns) - The minimum accuracy in nanoseconds of the routine used to measure time in the system.

op bandwidth (Mops/s) - The rate at which synchronization operations are performed. This is calculated as *ops per thread* multiplied by *number of threads* multiplied by *iterations* divided by *elapsed time* and scaled to give millions (10^6) of ops per second.

op latency (ns) - The average latency to complete a single synchronization operation. Locks include both the lock and unlock operations in the time. Atomic and compare-and-swap operations include only the single operation. This value is computed as *elapsed time* divided by the product of *ops per thread* and *iterations*, scaled to nanoseconds.

Appendix A: Building The Benchmarks

Each of the benchmarks can be built individually by changing directories to its source directory and typing *make*. In addition, there is a build script, called *build.sh*, at the top directory. This script will build all benchmarks with a reasonable set of compiler options and remove the object files.

Most benchmarks are written in C++ but portions of some benchmarks are also written in C. The choice of compiler can be managed using environment variables. Environment variables that affect compilation are as follows:

CC - Controls the selection of C compiler. The default is *gcc*.

CFLAGS - Controls the C compiler options. The default is *-O3 -m64* for most benchmarks. A similar set of compiler options for the Intel C/C++ compilers, is *-O3 -xSSE4.2 -fast*. The *stream* benchmark uses *-O3 -fopenmp -m64* for *gcc* and *-O3 -xSSE4.2 -fast -openmp* for the Intel C compiler *icc*.

CXX - Controls the selection of C++ compiler. The default is *g++*.

CXXFLAGS - Controls the C++ compiler options. As for *gcc*, the default is *-O3 -m64*. A useful set of compiler options for the Intel C++ compiler *icpc* is *-O3 -xSSE4.2 -fast*.

EXE - Controls the location of the executable. This can be used to create multiple versions of the executable, as in the following example.

```
% make CXX=”icpc” CXXFLAGS=”-O3” EXE=”getpid64_icc”
```

LIB - Controls the library options. This option sometimes interferes with an external variable used by the Intel compiler, and may need to be reset externally. An example of that would be

```
% make CXX=”icpc” CXXFLAGS=”-O3 -fast -xSSE4.2” LIB=””
```

As mentioned at the beginning of this section, a build script, *build.sh*, exists in the top directory. Each of these environment variables can be used with the build script as well. Typical Bash Shell syntax would be something like the following.

```
% CXX=”icpc” CXXFLAGS=”-O3 -fast -xSSE4.2” build.sh
```

A final note. Every source file contains a static variable containing an identifying string starting with “\0@ID”. This string also contains the date the file was compiles, a SHA256 hash of the source file, the name of the compiler, compiler version, options used to compile and name of the source file.

Appendix B: Running The Benchmarks

Benchmarks can be run individually using the flags described in previous sections, or they can be run as a suite using the script *run.sh*. This script has its own syntax and its own set of defaults. A simple set of examples can be found in the script *suite.sh*, which contains the following:

```
time nice --20 ./run.sh -getpid
time nice --20 ./run.sh -sha
time nice --20 ./run.sh -stream
time nice --20 ./run.sh -memcpy
time nice --20 ./run.sh -pchase
time nice --20 ./run.sh -sync-u
time nice --20 ./run.sh -sync-c
```

Of course, the Bash *time* command returns the elapsed, CPU and system times and the *nice* command elevates the benchmark priority.

These run scripts run the indicated benchmark while varying the input parameters to see how performance changes. The kinds of changes typically monitored are memory size to capture cache and memory performance, the number of threads to see how performance changes with different loading, the number of NUMA domains used, and whether the memory is local or remote.

Run.sh Usage

```
./run.sh
    [-getpid]
    [-memcpy [bcopy] [memcpy] [memmove] [naive]
        [copy_tt] [copy_tn] [xttu] [xtta] [xtn]
        [xnt] [xnn] [bzero] [memset] [naivezero]]
    [-pchase]
    [-stream]
    [-sha [sha1] [sha256g] [sha256s] [sha256u] [sha256v]
        [sha256x]]
    [-sync-c [spin] [mutex] [reader] [writer]
        [cmpxchg32] [cmpxchg64] [cmpxspin] [fairspin]
        [atomic32] [atomic64]]
    [-sync-u [spin] [mutex] [reader] [writer]
        [cmpxchg32] [cmpxchg64] [cmpxspin] [fairspin]
        [atomic32] [atomic64]]
    [-grain {1x|2x|4x}]
    [-scp [<user>@<host>:<dir>]]
```

Any of the benchmarks can be selected individually or in any combination with others. Four of the benchmarks, *memcpy*, *sha*, *sync-c* and *sync-u*, also have sub-parts that can be selected individually.

In addition to the benchmarks, there is an option to automatically copy the data files to another user/system/directory using the *scp* command as each sequence of runs completes.

Another option, *-grain*, determines the granularity of the benchmark runs. A value of *4x* runs the scripts with four times the granularity as a value of *1x*. The default value is *4x*.

When no command line arguments are given, it is as if the following were used:

```
./run.sh -getpid -memcpy memcpy copy_tn -pchase -sha sha1
sha256u sha256x -stream -sync-u spin mutex reader writer cmpx-
spin cmpxchg32 cmpxchg64
```

Environment Variables

Several environment variables can also be used to affect benchmark behavior. They are:

outdir - specifies the target location that *scp* will use to copy benchmark results as they become available. The syntax of the value of *outdir* is *user@host*:*directory.*

host - the host name of the system running the benchmarks. It is used as part of the name of the output file containing the results. When host is not defined with a value, the script attempts to determine the value automatically by using the output of the *hostname* command.

domains - specifies the NUMA domains where each of the system logical cores reside. This is used to map threads to domains using the Linux `sched_setaffinity()` function, which is in turn used to map virtual addresses to NUMA domains. An alternative, supported by the benchmarks but not the scripts, is to use the libnuma functions `numa_run_on_node()` and `numa_set_membind()`. The syntax of this variable is *d0,d1,...,dk* where *d0* is the domain of core 0, *d1* is the domain of core 1, etc. When no value for this variable is provided, the scripts examine the contents of the directory `/sys/devices/system/node/` to determine the mapping of cores to domains and set the value.

threads - specifies the number of possible threads (logical cores) in the system. When not specified by the user, the scripts examine the contents of the /sys/devices/system/node/ directory to determine its value.

dom_cnt - specifies the number of domains (domain count) in the system. When not specified by the user, the scripts examine the contents of the /sys/devices/system/node/ directory to determine its value.

rep - specifies the number of times a benchmark is to be repeated. For example, *rep=5* means the benchmark will be repeated five times and there will be five results reported. The default value is 1.

Variables Specific to *memcpy*

exp - specifies the number of times a benchmark is to be repeated, similar to *rep*. However, for *exp* only the run with the best performance is reported.

maxpool - specifies the largest memory pool size, in kilobytes, to be used. Source and target copy buffers are selected randomly from a pool of memory. The size of the pool determines whether the data is copied from L1, L2, or L3 cache, or from memory. The default is 262144, which represents 2^{28} bytes

Variables Specific to *pchase*

exp - specifies the number of times a benchmark is to be repeated, similar to *rep*. However, for *exp* only the run with the best performance is reported.

maxpool - specifies the largest memory pool size, in kilobytes, to be used. Pointers are selected randomly from a pool of memory. The size of the pool determines whether the pointer is in L1, L2, or L3 cache, or in memory. The default is 262144, which represents 2^{28} bytes.

minpool - specifies the smallest memory pool size in kilobytes, similar to the variable *maxpool*. The default is 8, which represents 2^{13} bytes.

access - specifies the type of pointer access to be used. The default is to use random access. Other values include *forward* and *backward*.

Variables Specific to *sync-u*

exp - specifies the number of times a benchmark is to be repeated, similar to *rep*. However, for *exp* only the run with the best performance is reported.

maxpool - specifies the largest memory pool size, in synchronization objects, one per alignment unit, to be used. The default alignment unit is 64 bytes, or one cache line. The default pool size is 8M (2^{23}) synchronization objects. (The benchmark reports the amount of memory used.)

minpool - specifies the smallest memory pool size, similar to *maxpool*. The default value is 128 synchronization objects.