Blue Gene/Q User Workshop

BG/Q Application Development
Threads & Memory

- Memory Overview
- SMT & Pipeline
- Threading
- Wakeup Unit
- L1Prefecher
- L2 Atomics
- Main Memory
Memory Overview
Memory

- There is a 16KB L1 data-cache and 4KB L1P prefetch cache. The L1P also contains write-back buffers for L1 data. The second-level (L2) cache is 32 MB.

<table>
<thead>
<tr>
<th>Cache</th>
<th>Quantity</th>
<th>Size</th>
<th>Latency</th>
<th>Replacement policy</th>
<th>Other information</th>
<th>Clock domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 instruction cache</td>
<td>18 (1 per processor)</td>
<td>16 KB</td>
<td>3 proc clocks (pclk)</td>
<td>Pseudo least recently used (LRU)</td>
<td>4-way set-associative 64-byte line size</td>
<td>Pclk</td>
</tr>
<tr>
<td>L1 data cache</td>
<td>18 (1 per processor)</td>
<td>16 KB</td>
<td>6 pclk (integer)</td>
<td>Pseudo LRU</td>
<td>8-way set-associative 64-byte line size</td>
<td>Pclk</td>
</tr>
<tr>
<td>L1 prefetch cache</td>
<td>18 (1 per processor)</td>
<td>32 x 128 bytes</td>
<td>24 pclk</td>
<td>Depth stealing and round robin</td>
<td>128-byte line</td>
<td>Pclk / 2</td>
</tr>
<tr>
<td>L2 cache</td>
<td>16</td>
<td>32 MB total</td>
<td>62 pclk</td>
<td>LRU</td>
<td>16-way set-associative 16-way sliced 4 banks per slice 8 sub-banks per slice 128-byte line</td>
<td>Pclk / 2</td>
</tr>
<tr>
<td>Double-data rate (DDR) memory</td>
<td>2</td>
<td>16 GB total</td>
<td>≥ 350 pclk</td>
<td></td>
<td>128-byte line</td>
<td>Pclk x (5 / 8)</td>
</tr>
<tr>
<td>Embedded dynamic random-access memory (eDRAM)</td>
<td>1</td>
<td>256 KB</td>
<td>≥ 80 pclk</td>
<td>Software control</td>
<td>16 bytes wide 8 eDRAM macro-internal bank</td>
<td>Pclk / 2</td>
</tr>
</tbody>
</table>

| Note: a. This value is the quantity on each Blue Gene/Q compute chip. |
| Note: b. The latency value is determined relative to instruction dispatch.
Memory and Caches

Number of cores
Number of threads (total)
L1 Dcache/Icache (each of the 16+1 cores)
L1 private prefetch (L1P)
L1 cache line size
L2 cache line size
Shared L2 cache
Bandwidth L1 to L1P
Bandwidth L1P to L2 (from any core)
Bandwidth L2 to DDR3 (aggregate for 2 channels)
Effective Stream Read (Write) bandwidth to DDR3
Peak Flops
Latency L1 to processor
Latency L1 to L1P
Latency L1 miss to L2 data returned
Latency L1 to main store data returned
Latency for msync (no contention)
Latency for stwcx (no contention)
Network bandwidth (each link)

16+1
64+4 (4 per core)
16kB/16kB (per core)
4kB (32 128B L2 cache lines)
64 bytes
128 bytes
32 MB
16B @ 1.6GHz read, 32 B @ 1.6GHz
32 B read + 12 B write @ 0.8GHz (563 GB/s)
32B r/w @ 1.333GHz (DDR3 1333) 42.656 GB/s
29.5 (27.2) GB/s; 18.4 (17.0) B/c
16*8*1.6GHz= 204.8 gigaFLOP/s
6 processor clocks
24 processor clocks
82 processor clocks
350 processor clocks
50 processor clocks
72 processor clocks
2GB/s send + 2GB/s receive
Memory

- The entire physical memory of a compute node is 16 GB so careful consideration of memory is required when writing applications. Some of that space is allocated for the CNK.
- *Shared memory space is also allocated to the user process at the time the process is created*
- The memory available to the application depends on the number of processes per node. The 16 GB of available memory is partitioned as evenly as possible among the processes on each node.
  - Determined by –ranks-per-node parameter

- The CNK tracks collisions of the stack and heap as the heap is expanded with brk() and mmap() system calls. The CNK and its private data are protected from reads and writes by the user process or threads. The code space of the process is protected from writing by the process or threads. Code and read-only data are shared between the processes that share each node.

- 64-bit memory model. You can use the Linux size command to display the memory size of the program. However, the size command does not provide any information about the runtime memory usage of the stack or heap.
Pipeline and SMT
A2 Core | Pipeline

- **Front-End (7 Stages)**
  - Fetch instructions, predict branches, thread arbitration

- **Back-End (8 Stages)**
  - Register Access and Bypass
  - Instruction Execution
  - L2 Interface

- **Front-End Stages 4-6 are replicated for each hardware thread**
  - Eight instructions per thread buffered in IU4
  - Oldest instruction is decoded and sent to IU5 – dependency calculated
    - **Primary Thread Stall Point** – Missing inputs
    - Only affects the thread
  - IU6 – Hold ready instruction from each thread
    - Scheduler decides which threads will issue
    - One to XU, one to AXU each cycle
    - **Primary Thread Stall Point** – Waiting for issue

- No stalls before IU5
- Flushes effect only instruction from a given thread
  - 32 buffered instruction means flushes before IU4 have little effect
  - Min 16 cycles to empty
  - 4 cycles to add
A2 Core | Why SMT?

- Pipeline is In-Order
  - Every load stalls a thread for the latency to the cache that has be accessed
    - Creates a bubble of idle cycles in the thread
- Out-of-order processors look ahead for an independent instruction to issue to fill bubble
- SMT enables out-of-order like behaviour – **reduces latency**
  - Exploits inherent parallelism in multicore algorithms
    - Often explicitly labelled by the developer
  - This parallelism is often not viewable to out-of-order processor – it can only look a certain amount of instruction ahead
- SMT **enables superscalar** pipeline
  - Without SMT only a single instruction can be issued each cycle
- Note: SMT does not provide increased maximum performance but **increased instruction throughput**
  - 1t and 4t theoretical performance is the same
A2 Core | 4 HW Thread

4 Iterations:
Complete in 4*(3+Latency) cycles
Complete in 6+Latency cycles
Parallelisation via SMT is different to multi-core threading
  - Should consider the two independently i.e. divide by core then for SMT

Since hardware threads are not fully parallel you won't see 4x Xup
  - Xup will depend on ratio of instructions/loads to different cache levels and contention
  - 2-2.5x can be expected.

Threading synchronization and locking is an issue
  - A2 processors contain features to address this (WakeUp Unit)
  - HW threads all access same L1 cache which lessens this problem

OpenMP threading usually breaks loops into chunks
  - Each iteration will have very similar set and order of instructions
  - Means loads between threads will likely overlap

Hardware threads don't:
  - Race ahead of each other – instructions dispatched round-robin
  - Increase contention for execution resources or bandwidth
    - 4t has same total number of instructions as 1t

Hardware threads do contend for certain core resources
  - L1 Cache, Load/Store Queue, L1 Prefetch

Contention can be lessened by choosing loop/task decomposition carefully
  - Ensure threads are working on same data

Threading
**Threads**

- The CNK provides a threading model based on the Native Portable Operating System Interface (POSIX) Thread Library (NPTL) available in the glibc library.
  - default threading package for Linux applications. The NPTL threading package implements the POSIX pthread API.
  - same POSIX pthread API library that Linux uses (-lpthreads) to function on the CNK without special parameters.

- **64 hardware threads, by default 1 user thread per hardware thread.**
  - The kernel scheduler runs on each hardware thread independently. Each local dispatcher handles the dispatching of the software threads assigned to the one hardware thread that it controls. There is no global dispatcher. Therefore, no global locks or blocking conditions are required to manage the dispatching of threads.
Threads

- Hardware Thread Identifiers

<table>
<thead>
<tr>
<th>Processor core ID</th>
<th>Processor thread ID</th>
<th>Processor ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, 1, 2, 3</td>
<td>0, 1, 2, 3</td>
</tr>
<tr>
<td>1</td>
<td>0, 1, 2, 3</td>
<td>4, 5, 6, 7</td>
</tr>
<tr>
<td>2</td>
<td>0, 1, 2, 3</td>
<td>8, 9, 10, 11</td>
</tr>
<tr>
<td>3</td>
<td>0, 1, 2, 3</td>
<td>12, 13, 14, 15</td>
</tr>
<tr>
<td>4</td>
<td>0, 1, 2, 3</td>
<td>16, 17, 18, 19</td>
</tr>
<tr>
<td>5</td>
<td>0, 1, 2, 3</td>
<td>20, 21, 22, 23</td>
</tr>
<tr>
<td>6</td>
<td>0, 1, 2, 3</td>
<td>24, 25, 26, 27</td>
</tr>
<tr>
<td>7</td>
<td>0, 1, 2, 3</td>
<td>28, 29, 30, 31</td>
</tr>
<tr>
<td>8</td>
<td>0, 1, 2, 3</td>
<td>32, 33, 34, 35</td>
</tr>
<tr>
<td>9</td>
<td>0, 1, 2, 3</td>
<td>36, 37, 38, 39</td>
</tr>
<tr>
<td>10</td>
<td>0, 1, 2, 3</td>
<td>40, 41, 42, 43</td>
</tr>
<tr>
<td>11</td>
<td>0, 1, 2, 3</td>
<td>44, 45, 46, 47</td>
</tr>
<tr>
<td>12</td>
<td>0, 1, 2, 3</td>
<td>48, 49, 50, 51</td>
</tr>
<tr>
<td>13</td>
<td>0, 1, 2, 3</td>
<td>52, 53, 54, 55</td>
</tr>
<tr>
<td>14</td>
<td>0, 1, 2, 3</td>
<td>56, 57, 58, 59</td>
</tr>
<tr>
<td>15</td>
<td>0, 1, 2, 3</td>
<td>60, 61, 62, 63</td>
</tr>
<tr>
<td>16</td>
<td>0, 1, 2, 3</td>
<td>64, 65, 66, 67</td>
</tr>
</tbody>
</table>
Process Identifiers & Threads

- The process identifier (PID) is a 4-byte signed number that identifies a process.
- Each process on a compute node has a unique PID.
- The PID value is not unique across compute nodes.
- The PID can be passed to various pthreads, signal APIs, and system calls as a thread group identifier (TGID) for the process.
- The thread identifier (TID) that corresponds to the primary thread of the process is the same value as the PID for the process.

- Some info under /proc (IO nodes) – coming driver will provide more Linux like data
Thread Groups

- Different software components may create a manage threads internally. These threads form a group – ignorant of threading issues in other software components.
- Three typical groups:
  - Application threading implemented directly in pthreads
  - OpenMP
  - PAMI Comm-Threads
- The BG Thread Model is created to handle multiple thread groups mixed in the same code without issues
  - In reality there are some issues you have to consider if your code is using multiple modules
Thread preemption

A pthread is preempted when and only when a pthread with a strictly higher software priority is available to be run on the same hardware thread. This scenario can occur for the following reasons:

- A futex-wait by a higher-priority pthread is satisfied.
- A signal is delivered to a higher-priority pthread.
- A new pthread with a higher software priority is created on, or is migrated to, this hardware thread.
- The software priority of the current pthread is lowered, or the priority of another pthread on the same hardware thread is raised.
- A nanosleep initiated by a higher-priority pthread on the same hardware thread expires.
- A timed futex wait initiated by a higher-priority pthread on the same hardware thread expires.
Hardware thread over-commitment

- More than one pthread can be assigned to a given hardware thread. These additional pthreads are supported by additional kernel thread structures.
- By default, five pthreads can be assigned to one hardware thread. In the Blue Gene/Q threading model, pthreads have absolute affinity to the hardware threads they are associated with.
- This behaviour can be changed using BG_THREADMODEL:
  - Setting this to 1 allows only 1 software thread per hardware thread.
- There is no time-quantum driven preemption of pthreads running on a hardware thread. After a pthread begins to run on a hardware thread, it continues to run until one of the following occurs:
  - The thread calls pthread_yield(), and an equal or higher-priority thread available for dispatch is found.
  - A signal is being delivered to a higher-priority pthread on the same hardware thread.
  - The thread enters a futex wait condition.
  - The thread enters a nanosleep system call.
Threads Scheduling/Affinity

- Scheduler
  - The kernel scheduler runs on each hardware thread independently. Each local dispatcher handles the dispatching of the software threads assigned to the one hardware thread that it controls. **There is no global dispatcher. Therefore, no global locks or blocking conditions are required to manage the dispatching of threads.**

- Affinity
  - When a pthread is created within a process, the CNK must select a hardware thread for the pthread.
  - **two layout algorithms for assigning pthreads to hardware threads** **BG_THREADLAYOUT.**
    - If required, additional layout algorithms can be added.
Threads affinity

- **Breadth-first assignment**
  - Breadth-first is the default thread layout algorithm. This algorithm corresponds to `BG_THREADLAYOUT = 1`. With breadth-first assignment, the hardware thread-selection algorithm progresses across the cores defined within the process before selecting additional threads within a given core – **round robin**
  - Use to have multiple cores with 1t/2t per core

- **Depth-first assignment**
  - This algorithm corresponds to `BG_THREADLAYOUT = 2`. With depth-first assignment, the hardware thread-selection algorithm progresses within each core before moving to another core defined within the process.
  - Use to get optimal placement for 4t per core

- **Thread affinity control**
  - through the `sched_setaffinity()` system call

- **Setting affinity with the pthread attribute** (example page 20 in redbook)
  - `CPU_ZERO(&cpumask);` // initialize the cpu mask
  - `CPU_SET(processorID, &cpumask);`
  - `pthread_attr_setaffinity(&attr, CPU_SETSIZE, &cpumask);`

- **Setting affinity with the system call**
  - `pthread_setaffinity_np( tid, sizeof(mask), &mask )`

- **Enablement Extended thread affinity control**
  - `BG_THREADMODEL` (c.f table D-5 in redbook for more information)
Thread priority control

• There are conditions in which a communication thread might require control only when no other application threads are running. Because of this requirement, communication threads can specify a priority that is lower than any application thread. Conversely, there are situations when a communication thread might need to be the highest priority software thread on the hardware thread. Therefore, communication threads are allowed to set a priority value that is more favored than any application thread priority.

• **Setting priority through the pthread attribute**
  – `pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);` more details page 22

• **Explicit setting of priority**
  – Priority can be set explicitly through the use of the `pthread_setschedparam()` API.
Wakeup Unit
Wakeup Unit

- Each core has a Wakeup Unit (WU)
  - Used in conjunction with the PowerPC *wait instruction*
  - When a hardware thread is in a *wait state*, the hardware thread stops executing
  - WU can send a “wakeup” signal to a hardware thread on detecting certain events
- The aim of the WU is:
  - To reduce performance loss due to threads in a polling loop
    - The polling thread must still use the pipeline to poll i- this wastes cycles that could be used by the other threads on the core
    - The loss is worst if the variable is in L1 (only six cycles latency)
  - To increase the speed at which a thread can be restarted
    - e.g OpenMP threads can be suspended rather than terminated between parallel regions
- **Configurable wakeup conditions:**
  - Wake Address Compare (WAC)
  - Messaging Unit activity
  - Interrupt sources
- Thread guard pages
  - CNK uses the Wakeup Unit to provide memory protection between stack/heap
  - Detects violation, but cannot prevent it
Wake up Unit

- The WU can be programmed by developers via kernel interfaces
  - Headers in /bgsys/drivers/ppcfloor/spi/include/wu/wait.h
- The wait process
  - A thread arms the WU with an address-range to monitor and then is suspended
  - The WU monitors all writes to these addresses
  - When it detects a write it resumes the software thread
  - The thread checks the written data to see if a condition has been met
    - No: Rearm WU and wait again
    - Yes: Continue
- Note: The WU must be armed with a physical address
  - Have to pin the memory corresponding to the address to be monitored using Kernel_CreateMemoryRegion()
  - Compute physical address as Base Physical – (Virtual – Base Virtual)
Fast Thread Wakeup

- The BG/Q OpenMP implementation can utilise the core wakeup unit
  - Lessens overhead in entering parallel regions
  - Lessens overhead in critical regions
- However the wakeup unit does not interact well with multiple software threads per hardware thread
  - A thread suspended by the wakeup unit doesn't relinquish the hardware thread it is bound to
- Can cause deadlock/handing
  - e.g. the master thread relinquishes to another thread which then suspends itself via the WU. The master thread won't run again.
- Turn fast wakeup on using BG_SMP_FASTWAKEUP=yes
  - By default it's off due to the above problems
- If you know there are <= 64 threads in your app you can turn it on
  - Or change BG_THREADMODEL to 1 – only 1 software thread per hw thread
- However you have to be careful mixing PAMI Comm-Threads (asynchronous messaging) and OpenMP with Fast-Wakeup
L1P Prefetching cache
L1 prefetch cache overview

- 1 L1P per core

- The level 1 prefetch (L1p) cache is a module that provides the interface between an A2 core and the rest of the Blue Gene/Q system

- 32 x 128 byte cache structure to identify and prefetch memory access patterns.

- Functions
  - Provides A2 interfaces to the Blue Gene/Q system (store, reload, synchronize, invalidation, ..)
  - 2 prefetching modes
    - Stream prefetch engine with automatically detected and software-hinted streams
      - Contiguous memory access
    - List prefetch engine
      - Non-contiguous but deterministic memory access
L1 prefetch cache overview

- **Streams**
  - Up to **16 concurrent linear streams** of consecutive addresses can be simultaneously prefetched.
  - Linear streams can be automatically identified, hinted using data cache block touch (**dcbt**) instructions, or established optimistically for any miss.
  - Stream underflow (a hit on a line that is being fetched from the switch) triggers a depth increase when adaptation is enabled. Stream replacement and depth stealing lines are selected with a least recently hit algorithm.

- **List**
  - Access patterns can be recorded and reused by a list fetch engine.

- **A2 interface**
  - The L1p cache accepts commands and data from the A2 core at the pclk period.
  - Received commands are queued in a 32-deep lookup queue.
    - eight outstanding data load requests
    - four instruction load requests
    - 20 store requests.
      - Corresponds to the maximum of 16 requests that can be accepted by the switch and an additional four active store commands, not committed to the switch, which the L1p cache can maintain for write combining.
Stream Prefetching

- **Stream** - Access to a number of contiguous memory addresses in sequence
  - L1P detects a stream and prefetches addresses from it

- **Depth**
  - Controls how long the prefetcher assumes the stream is in 128 bytes chunks
    - Typically 2/3

- **Mode**
  - Controls when a stream is established/recognized by the prefetcher
    - **Confirmed**
      - A Stream is confirmed when there are two L1 miss within depth.
      - Once confirmed, the L1p fetches the first depth chunks
      - On a subsequent miss to the next depth chunks the L1P prefetches them
    - **Confirmed or cache touch mode**
      - The prefetcher behaves like confirmed mode. Additionally, a stream will be established if an explicit dcbt (data cache block touch) instruction that results in an L1 cache miss is executed.
    - **Optimistic**
      - Every L1 miss is assumed to be a stream – confirmation is not required
    - **Prefetch Disabled**
      - Subsequent chunks in the stream must be confirmed (2 misses) before they are fetched

- **Adapativity**
  - If adaptivity is on the depth is increased on each subsequent fetch
Stream/linear Prefetch

- **One prefetch buffer for each core**, and if a user wants to change the default behavior, it is important to set the desired parameters on every core.
  - Default policy: “L1P_confirmed_or_dcbt”, generally good
    - begin prefetching once an increasing memory address sequence is detected, or if there is a specific data-cache-block-touch (dcbt) request.
    - three 128-byte lines per stream => 32x128Bytes = ~10 streams/core
    - Can be reduce with 4 threads/core

- The stream prefetch policy and depth can have a effect on application performance, but the effect depends on how many cores and threads are active on the system.

```c
hwthread = Kernel_PhysicalHWThreadID();  // returns 0-3 on each core
if (hwthread == 0) {
  ptr = getenv("L1P_POLICY");
  if (ptr != NULL) {
    if       (strncasecmp(ptr,"opt", 3) == 0) L1P_SetStreamPolicy(L1P_stream_optimistic);  depth increase
    else if  (strncasecmp(ptr,"con", 3) == 0) L1P_SetStreamPolicy(L1P_stream_confirmed); with pragma
    else if  (strncasecmp(ptr,"dcbt",4) == 0) L1P_SetStreamPolicy(L1P_confirmed_or_dcbt);
    else if (strncasecmp(ptr,"dis", 3) == 0) L1P_SetStreamPolicy(L1P_stream_disable);
    else L1P_SetStreamPolicy(L1P_confirmed_or_dcbt);
  }
  ptr = getenv("L1P_DEPTH");
  if (ptr != NULL) {
    stream_depth = atoi(ptr);
    L1P_SetStreamDepth(stream_depth);
  }
}
```
L1P & Thread synchronization

- Each core in a Blue Gene/Q system has a single dedicated linear stream prefetcher. However, all four hardware threads share the same linear stream prefetcher. This sharing can cause an atomicity and ordering problem if multiple threads modify the linear stream prefetcher's configuration registers. The L1p and the SPI do not restrict or block access to the configuration registers. If the application developer is potentially modifying the configuration registers from multiple hardware threads, locking must be added or there might be some non-deterministic choices in the L1p configuration.

- The following strategies can be used for locking:
  - pthread_mutexes
    - Standard POSIX locking primitives. These primitives are very simple and work well in threaded processes.
  - larx/stcx
    - PowerPC load reservation locking mechanism. This mechanism can be used to create a lighter-weight lock than pthread_mutex. Larx/stcx instructions can also be used for multiple processes that have a shared memory region.
  - L2 atomics
    - Using the Blue Gene/Q L2 atomic operations to "take a ticket" with load and increment. The thread blocks until the "now serving" counter matches the ticket. The thread then updates the configuration register and performs a store and add operation to add 1 to the "now serving" counter.
Stream modes

- **Optimistic mode**
  - Assumes that all L1 misses will become streams. In this mode, the L1p immediately starts

- **Confirmed mode**
  - The linear stream prefetcher will wait for at least one additional L1 miss that corresponds to the stream has been detected. Once confirmed, the L1p starts prefetching from the stream.

- **Confirmed or cache touch mode**
  - The prefetcher behaves like confirmed mode. Additionally, a stream will be established if an explicit dcbt (data cache block touch) instruction that results in an L1 cache miss is executed.

- C.f. redbook for more details (L1P prefetcher API descriptions)
L1 Perfect Prefetcher

- Aka Pattern or List Prefetcher
- Usage:
  - First iteration through code:
    • BQC records sequence of memory load accesses
    • Sequence is stored in DDR memory
  - Subsequent iteration through code:
    • BQC loads the sequence and tracks where the code is in the sequence
    • Prefetcher attempts to prefetch memory before it is needed in the sequence
  - Has tolerance for misses in patterns
  - Looks ahead 8 places in a pattern to find a matching address.
  - If no match marks that address as a miss
  - After a certain number of misses it will abandon the stream
- Kernel provides access routines to setup and configure the stream and perfect prefetchers
- Extensive information in updated Redbook
List prefetch

• It is best suited for situations where performance is gated by L1 D-cache load misses, where the miss pattern repeats almost exactly, and where threading is limited to 1 or 2 threads per core.

• Hardware counter data would be useful to provide guidance about which code blocks might benefit, and an estimate for the number of L1 misses – needed to get the right size for read/write lists.

• In experiments so far, the list prefetch feature provides very little gain when using four threads per core. Using four threads per core is pretty effective at increasing throughput when performance is gated by either memory access or floating-point pipelining issues.
List prefetch API

- API: /bgsys/drivers/ppcfloor/spi/include/l1p
  - spi/include/pprefetch.h and spi/include/types.h
- Each hardware thread can independently generate and use a list.
- int L1P_PatternConfigure(uint64_t n);
  - allocates read and write lists: n*128 bytes each
- int L1P_PatternUnconfigure();
  - frees the read and write lists
- int L1P_PatternStart(int record);
  - records a new list if “record” is not zero, else uses previous list
- int L1P_PatternStop();
  - stops recording or playback, switches read/write lists when “record” is not zero
- int L1P_PatternStatus(L1P_Status_t* status);
  - returns bits in the status structure to indicate things like list overflow
Fortran – use wrappers

void l1p_patternconfigure(int * misses) {  // call l1p_patternconfigure(misses)
    L1P_PatternConfigure((uint64_t)*misses);
}

void l1p_patternstart(int * flag) {          // call l1p_patternstart(flag)
    L1P_PatternStart(*flag);
}

void l1p_patternstop(void) {                 // call l1p_patternstop()
    L1P_PatternStop();
}

void l1p_patternstatus(int * status) {    // call l1p_patternstatus(status)
    L1P_Status_t st;                       // finished = ishft(iand(status,8), -3)
    L1P_PatternStatus(&st);                // abandoned = ishft(iand(status,4), -2)
    *status = (int)st.status;              // maximum = ishft(iand(status,2), -1)
}

void l1p_patternunconfigure(void) {       // call l1patternunconfigure()
    L1P_PatternUnconfigure();
}
```c
#include <spi/include/l1p/pprefetch.h>
L1P_Status_t st;

nx = 100; ny = 100; nz = 100;
misses = 2*ny*nz; // expected number of L1 d-cache misses
L1P_PatternConfigure(misses);
for (iter=0; iter<maxiter; iter++) {
    record = 1; // also try record = iter%10
    L1P_PatternStart(record);
    ndx = 0;
    for (k=0; k<nz; k++)
        for (j=0; j<ny; j++) {
            b[ndx] = a[0 + j*nx + k*nx*ny];
            b[ndx+1] = a[1 + j*nx + k*nx*ny];
            ndx += 2;
        }
    L1P_PatternStop();
    L1P_PatternStatus(&st);
    printf("iter %d: maxed=%d, abandoned=%d\n", iter, st.s.maximum, st.s.abandoned);
}
L1P_PatternUnconfigure();
```

Example C – Strided Copy
List prefetch multiple threads example

```plaintext
!$omp parallel
    if (use_L1p) call L1P_PatternConfigure(l1misses)
!$omp end parallel
    do iter = 1, maxiter
        if (mod(iter-1, 4) .eq. 0) then
            record = 1
        else
            record = 0
        end if
        call update_arrays(a, c, delta, nx, ny, nz)
    !$omp parallel
        if (use_L1p) call L1P_PatternStart(record)
    !$omp end parallel
    call copy_halo(a, sbuf, nx, ny, nz)
    !$omp parallel
        if (use_L1p) call L1P_PatternStop()
    !$omp end parallel
    end do
!$omp parallel
    if (use_L1p) call L1P_PatternUnconfigure()
!$omp end parallel
```

sbuf(2*j - 1 + 2*(k-1)*ny) = a(1,j,k)
sbuf(2*j + 2*(k-1)*ny) = a(2,j,k)
List prefetch - results

- Performance measurements for the “copy_halo” routine as a function of threads-per-process, with the list prefetch feature enabled or disabled, using a total of four processes on one node.

<table>
<thead>
<tr>
<th>#threads/process</th>
<th>list enabled: cycles/iteration</th>
<th>list enabled: bandwidth (Bytes/cycle)</th>
<th>list disabled: cycles/iteration</th>
<th>list disabled: bandwidth (Bytes/cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>66.2</td>
<td>11.9</td>
<td>397.8</td>
<td>1.9</td>
</tr>
<tr>
<td>2</td>
<td>48.2</td>
<td>16.3</td>
<td>196.0</td>
<td>3.9</td>
</tr>
<tr>
<td>3</td>
<td>44.8</td>
<td>16.8</td>
<td>133.4</td>
<td>5.8</td>
</tr>
<tr>
<td>4</td>
<td>44.5</td>
<td>16.8</td>
<td>102.8</td>
<td>7.4</td>
</tr>
</tbody>
</table>

- Data for the NAS Parallel Benchmark CG, class B, 16 MPI ranks with the prefetch list enabled and disabled.

<table>
<thead>
<tr>
<th>#threads/core</th>
<th>Time (sec): list enabled</th>
<th>L1P Misses: list enabled</th>
<th>Time (sec): list disabled</th>
<th>L1P Misses: list disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32.19</td>
<td>323,254,145</td>
<td>52.33</td>
<td>23,389,304,136</td>
</tr>
<tr>
<td>4</td>
<td>28.65</td>
<td>10,029,399,930</td>
<td>21.18</td>
<td>26,108,998,844</td>
</tr>
</tbody>
</table>

* For > 1-2thr/core potentially degradation will list prefetch due to bandwidth and L1-D size
List Prefetch Analysis Tool

1. Suggesting the size of the list;
2. Telling whether the list recording should be on or off for each iteration; and
3. Providing related hardware counter events.

=================================================================
1. Instrument the source code manually
   1. `#include "lct.h"
   2. call lctb_init() and lctb_finalize() in the beginning and the end to initialize/finalize the library
   3. call lctb_start() and lctb_stop() at the beginning and the end of the code region of interest
   4. call lctb_pause() and lctb_resume() to skip code regions
2. Compile the source and link the liblct.a library (-llct -lSPI_l1p -lSPI_cnk -lSPI_upci_cnk)
3. Run the job. It is required to have the environment variable
   BG_DEBUGREGSYSCALLSENABLED=1 to enable the customized signal handler
L2 Cache
There are 16 individual caches, or slices. Each cache is assigned to store a unique subset of the physical memory lines. The physical memory addresses that are assigned to each cache slice are static and configurable.
- All core/processes access all slices of the cache

Access is through a full-crossbar switch
- Each slice can be accessed by a core at the same time

The L2 line size is 128 bytes, which is twice the width of an L1 line. L2 slices are set-associative and organized as 1024 sets. Each set has 16-way association.

The main memory is accessed through two on-chip DDR controllers. Each controller manages eight L2 slices.

The primary logic of the L2 caches operates at half the processor clock frequency. Some interface logic operates at lower frequencies. Each L2 slice has a single read data port that is 256 bits wide, a single write data port that is 256 bits wide, and a single request port. This port is shared by all processors through the crossbar switch.

The L2 caches primarily operate as normal, set-associative caches. They also support speculative threads and atomic memory transactions.
The L2 caches serve as the point of coherence for all processors. Therefore, they generate L1 invalidations when required. Because the L2 caches are inclusive of the L1 caches, they can remember which processors might have a valid copy of every line. They can multicast invalidations to only those processors. The L2 caches are also a synchronization point, so they coordinate synchronization (msync), load and reserve (lwarx), and store conditional (stwcx) instructions.

- Separate channels for invalidates, requests and data
Memory - L2 atomic operations

• The BG/Q nodes have support for atomic memory operations in the L2 cache. In some circumstances, atomic memory operations can be more efficient than standard PowerPC `larx/stcx` atomic instructions. The `larx/stcx` instructions require at least two operations for atomicity:
  1. A load with reservation, which brings the data back to a processor general-purpose register (GPR)
  2. A store operation, which pushes out the data

• BIG/Q L2 atomics allow for a single load or store operation to perform a simple arithmetic operation in the L2 cache. This method saves the latency of the load. If the L2 atomic operation is a `store` operation code, the store operation is placed on the queue and the A2 core does not stall.

• the memory regions that contain L2 atomic memory must be predesignated. This predesignation is required because the CNK must create special memory translation entries for L2 atomic memory. Use the following SPI routine to predesignate memory:
  – `uint64_t Kernel_L2AtomicsAllocate(void* atomic_vaddress, size_t length);`
• There are a limited number of memory translation entries.
L2 Atomic Operations Example

```c
#include <spi/include/l2/atomic.h>
#include <hw1/include/common/bgq_alignment.h>

struct L2_LoadIncBounded_t {
    uint64_t i, n;
} lib ALIGN_QUADWORD;

void init()
{
    Kernel_L2AtomicsAllocate(&lib, sizeof(lib));
    L2_AtomicStore(&lib.i, 0);
    L2_AtomicStore(&lib.n, 255);
}

uint64_t obtain_next(int* success)
{
    uint64_t obtained, doneval = 0x8000000000000000;
    obtained = L2_AtomicLoadIncrementBounded(&lib.i);
    *success = (obtained != doneval);
    return obtained;
}

void main()
{
    init();
    #pragma omp parallel
    {
        uint64_t item;
        int more;
        do {
            item=obtain_next(&more);
            if (more) printf("obtained %ld\n", item);
        } while (more);
    }
```
L2 Atomics

- L2 implements atomic operations on every 64-bit word in memory
  - Allows specific memory operations to be performed atomically:
    - Load: Increment, Decrement, Clear (plus variations)
    - Store: Twin, Add, OR, XOR, Max (plus variations)
  - L2 knows the operation when specific shadow physical addresses are read/written

- CNK exposes these L2 atomic physical addresses
  - Applications must pre-register the location of the L2 atomic before access
    - Otherwise segfault
  - Fast barrier implementation using L2 atomics available

- CNK also uses L2 atomics internally
  - Fast performance counters (store w/ add 1)
  - Locking
Atomic Performance results

Barrier speed using different synchronizing hardware

- **atomic: no -invalidates**
- **atomic: invalidates**
- **lwarx/stwcx**
Main Memory
Memory management

• The CNK does not dynamically grow its memory usage over time. The CNK consumes a fixed size of 16 MB out of the 16 GB of memory. Therefore, additional threads, mmaps, system calls, buffers, and so on, do not change internal kernel memory usage. The remainder of the memory (16,368 MB on a 16 GB node) is partitioned for the application.
Memory management – Shared Memory

- **Shared memory**
  - node. The size of shared memory must be specified to the `runjob` command with an environment variable.
    - The **BG_SHARED_MEMSIZE** environment variable specifies the amount of memory to be allocated in MB.
  - Shared memory is allocated with the standard Linux `shm_open()` and `mmap()` methods (POSIX shared memory interface)
  - `shm_open()` and `shm_unlink()` routines access a pseudo-device, `/dev/shm/filename`, which the kernel interprets. Because multiple processes can access or close the shared-memory file, allocation and deallocation are tracked by a simple reference count. Therefore, the processes are not required to coordinate deallocation of the shared memory region.

Example 4-1  Shared memory allocation

```c
fd = shm_open( SHM_FILE, O_RDWR, 0600 );
truncate(fd, MAX_SHARED_SIZE);
shmptr1 = mmap(NULL, MAX_SHARED_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Example 4-2 illustrates shared-memory deallocation.

Example 4-2  Shared memory deallocation

```c
munmap(shmptr1, MAX_SHARED_SIZE);
close(fd);
shm_unlink(SHM_FILE);
```
Memory - **Persistent memory**

- Persistent memory is process memory that retains its contents from job to job.
- To allocate persistent memory, the environment variable `BG_PERSISTMEMSIZE = X` must be specified.
  - “X” represents the number of megabytes to be allocated for use as persistent memory.
- For the persistent memory to be maintained across jobs, all job submissions must specify the same value for the `BG_PERSISTMEMSIZE` variable.
  - The contents of persistent memory can be reinitialized during job startup either by changing the value of `BG_PERSISTMEMSIZE` or by specifying the environment variable `BG_PERSISTMEMRESET=1`.
- The `persist_open()` kernel function supports persistent memory
  - In SPI.