BG/Q Application Tuning

memory hierarchy, transactional memory, speculative execution

Dr. Thilo Maurer, IBM Research & Development Böblingen
tmaurer@de.ibm.com

JUQUEEN Porting and Tuning Workshop, Feb 5th 2013
BlueGene/Q Memory Hierarchy

- 8GB memory
- 8GB memory
- 32MB L2 cache
- L1 prefetcher
- 16K L1 Cache
- Power PC core

×17
## Memory and Caches

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>16+1</td>
</tr>
<tr>
<td>Number of threads (total)</td>
<td>64+4 (4 per core)</td>
</tr>
<tr>
<td>L1 Dcache/Icache (each of the 16+1 cores)</td>
<td>16kB/16kB (per core)</td>
</tr>
<tr>
<td>L1 private prefetch (L1P)</td>
<td>4kB (32 128B L2 cache lines)</td>
</tr>
<tr>
<td>L1 cache line size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L2 cache line size</td>
<td>128 bytes</td>
</tr>
<tr>
<td>Shared L2 cache</td>
<td>32 MB</td>
</tr>
<tr>
<td>Bandwidth L1 to L1P</td>
<td>16B @ 1.6GHz read, 32 B @ 1.6GHz</td>
</tr>
<tr>
<td>Bandwidth L1P to L2 (from any core)</td>
<td>32 B read + 12 B write @ 0.8GHz (563 GB/s)</td>
</tr>
<tr>
<td>Bandwidth L2 to DDR3 (aggregate for 2 channels)</td>
<td>32B r/w @ 1.333GHz (DDR3 1333) 42.656 GB/s</td>
</tr>
<tr>
<td>Effective Stream Read (Write) bandwidth to DDR3</td>
<td>29.5 (27.2) GB/s; 18.4 (17.0) B/c</td>
</tr>
<tr>
<td>Peak Flops</td>
<td>16×8×1.6 GHz = 204.8 GF/s</td>
</tr>
<tr>
<td>Latency L1 to processor</td>
<td>6 processor clocks</td>
</tr>
<tr>
<td>Latency L1 to L1P</td>
<td>24 processor clocks</td>
</tr>
<tr>
<td>Latency L1 miss to L2 data returned</td>
<td>72 processor clocks</td>
</tr>
<tr>
<td>Latency L1 to main store data returned</td>
<td>223 processor clocks</td>
</tr>
<tr>
<td>Latency for msync (no contention)</td>
<td>~50 processor clocks</td>
</tr>
<tr>
<td>Latency for stwcx (no contention)</td>
<td>~72 processor clocks</td>
</tr>
<tr>
<td>Network bandwidth (each link)</td>
<td>2GB/s send + 2GB/s receive</td>
</tr>
</tbody>
</table>
Reminder: Recent Thread Synchronisation

➢ when multiple threads access shared data
  ➢ potentially wrong results when (e.g.) data is read while other thread modifies “read-after-write”
➢ Workarounds:
  ➢ Need to have single master (locking)
  ➢ Strict program-flow (barrier)
Transactional Memory

- Mechanism to enable atomic operations on arbitrary set of memory locations
- BG/Q supports hardware transactional memory
  - hardware detects write/read conflicts
  - runtime rolls back on failure
- Benefits
  - replace pessimistic synchronization (locking) with optimistic synchronization
  - may be used to parallelize workload into collaborative but independent tasks on shared data (allowed to commit out-of-order)
Transactional Memory Syntax

- Compiler flag to enable TM pragmas: -qtm
- User needs to specify atomic code blocks using compiler directives
  - Needs to be inside "OMP PARALLEL" or pthread
  - Additional speedup for SAFE_MODE if no irrevocable actions (e.g. IO)

```c
!$omp parallel private(i)
!$omp do
  do i = 1, N
  !TM$ TM_ATOMIC SAFE_MODE
    call code_to_be_atomic(i)
  !TM$ END TM_ATOMIC
  enddo
!$omp end do
!$omp end parallel
```

```c
#pragma omp parallel private(i)
{
  #pragma omp for
    for (int i=0;i<N;i++) {
    #pragma TM_ATOMIC SAFE_MODE
      {
        code_to_be_atomic(i);
      }
  }
}
```
Transactional Memory Limitations & Tuning

- Performance governed by tradeoff
  - (execution time of atomic region) vs. (entry and exit overhead)
  - conflict probability (overhead from rollbacks)
  - nested TM will be flattened
  - number of hardware limitations (next slides)
Transactional Memory Limitations & Tuning

➢ **Number of concurrent transactions**: Each speculative thread obtains a “speculation ID”. There cannot be more than 128 (per node). Requesting more creates processor stalls.

➢ User may need to tune “clean-up interval”:
  ```
  export BG_SPEC_SCRUB_CYCLE=(6-100)
  ```
  many transactions: small values more efficient
  few transactions: large values more efficient
Transactional Memory Limitations & Tuning

➢ **Size of Transactions:** L2-cache\(^1\) is coordination point. For each set 10 of its 16 ways can serve as memory for transactional state.
  ➢ Set of active transactions can not interface (at 128 byte granularity) to more than 20MB of memory
  ➢ Maximum of 10 ways per set usually exhausted earlier

\(^1\)L2 Cache: 32MB capacity, 16-way set-associative, 128B word-size
Transactional Memory Limitations & Tuning

➢ **Size of Transactions:** L2-cache\(^1\) is coordination point. For each set 10 of its 16 ways can serve as memory for transactional state.
  ➢ Set of active transactions can not interface (at 128 byte granularity) to more than 20MB of memory
  ➢ Maximum of 10 ways per set usually exhausted earlier

\(^1\)L2 Cache: 32MB capacity, 16-way set-associative, 128B word-size
Transactional Memory Limitations & Tuning

- **Locality of Transactions**: concurrent transactions may produce conflict although they do not conflict when writing to neighboring addresses
  - Short-running mode (TM only): 8 byte granularity
  - Long-running mode (SE and TM): 64 byte granularity (L1 cache-line)
Transactional Memory Environment Variables

➢ Times to try rollback before entering irrevocable mode (default is 10):
  export TM_MAX_NUM_ROLLBACK=N

➢ Lazy or eager mode to resolve conflicts (default is yes):
  export TM_ENABLE_INTERRUPT_ON_CONFLICT=(yes|no)

➢ Long Running Mode: (default), flush L1 Cache at region begin; L1 caches all reads
  export TM_SHORT_TRANSACTION_MODE=no

➢ Short Running Mode: no Cache flush; but read-after-write needs to resort to L2
  export TM_SHORT_TRANSACTION_MODE=yes

➢ Tune “clean-up interval” if tiny transactions prevalent (default is 20):
  export BG_SPEC_SCRUB_CYCLE=(6-100)

➢ Turn on logs to inspect runtime-gathered statistics for tuning:
  export TM_REPORT_NAME=<filename>
  export TM_REPORT_LOG=(func|summary|all|verbose)
  export TM_REPORT_STAT_ENABLE=(yes|no)
  (see IBM XL Optimization and Programming Guide)
Thread Level Speculation

➢ Similar to Transactional Memory, but
  ➢ on the level of threads
  ➢ commit in-order

➢ Parallelize potentially dependent fragments of serial code
  • runtime creates threads for each speculative section
  • threads run parallel and commit serialized if no conflict
  • on conflict, all threads except current master is rolled back

```c
appdata d;
for (int i=0;i<N;i++) {
    code_fragment(i,&d);
}
{
    code_fragment(0,&d);
    code_fragment(1,&d);
}
```
Thread Level Speculation Syntax

- Enable by compiler flag `-qsmp=speculative`

```c
#pragma speculative for private(i)
for (int i=0;i<N;i++) {
    code_to_be_spec(i);
}

#pragma speculative sections
{
    #pragma speculative section
    { some_code();  }
    #pragma speculative section
    { other_code();  }
}

!SEP$ SPECULATIVE DO PRIVATE(i)
    do i = 1, N
        call code_to_be_spec(i)
    enddo
!SEP$ END SPECULATIVE DO

!SEP$ SPECULATIVE SECTIONS
    call some_code()
!SEP$ SPECULATIVE SECTION
    call other_code()
!SEP$ END SPECULATIVE SECTIONS
```
Thread Level Speculation Limitations & Tuning

➢ Number of concurrent transactions
➢ Size of Transactions
➢ Locality of Transactions
➢ At most 16 processes per node
Thread Level Speculation Limitations & Tuning

➢ Performance governed by trade-off of overhead and conflict probability
  • chunk N iterations to single thread to tune trade-off:
    #pragma speculative for private(i) schedule(static,N)

➢ Times to try rollback before non-speculative (i.e. single-threaded) execution:
  export SE_MAX_NUM_ROLLBACK=N

➢ Turn on logs to inspect runtime-gathered statistics for tuning:
  export SE_REPORT_NAME=<filename>
  export SE_REPORT_LOG=(func|summary|all|verbose)
Further Reading and References

➢ IBM XL C/C++ for BlueGene/Q, V12.1
   IBM XL FORTRAN for Blue Gene/Q, V14.1
   ► Language Reference
   ► Compiler Reference
   ► Optimization and Programming Guide

➢ IBM System Blue Gene Solution Redbooks, e.g.,
   ► Blue Gene/Q Application Development Manual
     http://publib-b.boulder.ibm.com/redpieces/abstracts/sg247948.html
   ► Blue Gene/Q Code Development and Tools Interface
     http://publib-b.boulder.ibm.com/abstracts/redp4659.html

➢ JUQUEEN Documentation

➢ “What Scientific Applications can benefit from Hardware Transactional Memory?”
   M. Schindewolf et al.

➢ “Evaluation of Blue Gene/Q Hardware Support for Transactional Memories”
   A. Wang et al.
Transactional Memory Example – SERIAL

- Code with quasi-random updates of single entries of large array (~19MB)
- Each iteration 2 of 2510000 variables changed: conflict probability ~8e-7,
- Full loop: each variable changed ~4 times

```fortran
integer, parameter :: cs=10, np=5000000
integer, parameter :: nx=(np/cs)
real(DP), allocatable :: x(np), rho(0:1000:nx+1000,5)
integer, allocatable :: charge(np)
real(DP) :: oodx, f1, f2, xa, re
integer :: j1, j2, i, ci

do i=1,np
   xa = x(i)*oodx
   j1 = xa
   j2 = j1+1
   f2 = xa-j1
   f1 = 1.0-f2
   ci = charge(i)
   rho(j1,ci) = rho(j1,ci) + re*f1
   rho(j2,ci) = rho(j2,ci) + re*f2
end do
```
Transactional Memory Example – NAIVE

➢ Naive parallelization with allowed conflicts (rho shared)

```c
 !$OMP PARALLEL private(xa, i, j1, j2, f1, f2, ci) default(shared)
 !$OMP DO
 do i=1,np
   xa = x(i)*oodx
   j1 = xa
   j2 = j1+1
   f2 = xa-j1
   f1 = 1.0-f2
   ci = charge(i)
   rho(j1,ci) = rho(j1,ci) + re*f1
   rho(j2,ci) = rho(j2,ci) + re*f2
 end do
 !$OMP END DO
 !$OMP END PARALLEL
```

Transactional Memory Example – REDUCTION

- Parallelization using OMP REDUCTION (local copies summed after loop)

```c
!$OMP PARALLEL private(xa, i, j1, j2, f1, f2, ci) default(shared)
!$OMP DO REDUCTION(+:rho)
do i=1,np
   xa = x(i)*oodx
   j1 = xa
   j2 = j1+1
   f2 = xa-j1
   f1 = 1.0-f2
   ci = charge(i)
   rho(j1,ci) = rho(j1,ci) + re*f1
   rho(j2,ci) = rho(j2,ci) + re*f2
end do
!$OMP END DO
!$OMP END PARALLEL
```
Transactional Memory Example – Results

The graph shows the execution time (in seconds) as a function of the number of threads. There are two lines on the graph:

- **Red line**: Represents the reduction stage of the transactional memory example.
- **Green line**: Represents the collision stage of the transactional memory example.

As the number of threads increases, the execution time decreases for both stages, indicating improved performance with more concurrent threads.
Parallelization using Atomic Transactions

```c
!$OMP PARALLEL private(xa, i, j1, j2, f1, f2, ci) default(shared)
!$OMP DO
  do i=1,np
    xa = x(i)*oodx
    j1 = xa
    j2 = j1+1
    f2 = xa-j1
    f1 = 1.0-f2
    ci = charge(i)
    !TM$ TM_ATOMIC SAFE_MODE
    rho(j1,ci) = rho(j1,ci) + re*f1
    rho(j2,ci) = rho(j2,ci) + re*f2
    !TM$ END TM_ATOMIC
  end do
!$OMP END DO
!$OMP END PARALLEL
```
Transactional Memory Example – TM Results

The graph illustrates the execution time in seconds for different thread counts and two types of transactions: reduction and collision. The lines represent various thread counts:
- TM SHORT 20
- TM LONG 20

The execution time decreases as the number of threads increases, with the reduction transactions showing a more significant decrease compared to the collision transactions.
Transactional Memory Example – TM Results

The graph shows the execution time in seconds (y-axis) against the number of threads (x-axis). The legend includes the following labels:

- reduction
- collision
- TM SHORT 60
- TM SHORT 20
- TM LONG 60
- TM LONG 20

The execution time decreases as the number of threads increases, indicating improved performance with more parallel processing.
Transactional Memory Example – TM Results

The diagram illustrates the execution time (in seconds) for various transactional memory operations as a function of the number of threads.

- **Reduction**
- **Collision**
- **TM SHORT 60**
- **TM SHORT 20**
- **TM SHORT 6**
- **TM LONG 60**
- **TM LONG 20**
- **TM LONG 6**

The execution time decreases as the number of threads increases, indicating improved performance with increased parallelism.
Transactional Memory Example – Results

Great scaling due to few conflicts

scrubbing “ok”

Overhead

Execution time [secs] vs. # threads
Transactional Memory Example – TM

Parallelization using Atomic Transactions

```c
!$OMP PARALLEL private(xa, i, j1, j2, f1, f2, ci) default(shared)
!$OMP DO
  do i=1,np
    xa = x(i)*oodx
    j1 = xa
    j2 = j1+1
    f2 = xa-j1
    f1 = 1.0-f2
    ci = charge(i)
    !TM$ TM_ATOMIC SAFE_MODE
    rho(j1,ci) = rho(j1,ci) + re*f1
    rho(j2,ci) = rho(j2,ci) + re*f2
    !TM$ END TM_ATOMIC
  end do
!$OMP END DO
!$OMP END PARALLEL
```
Thilo Maurer, IBM R&D Böblingen

Transactional Memory Example – TM

➢ Parallelization using chunking of Atomic Transactions

chunks=np/chunking
!$OMP PARALLEL private(xa, i, j1, j2, f1, f2, ci, i, c) default(shared)
!$OMP DO schedule(static)
do c=1,chunks
  do ii=1,chunking
    i = ii+(c-1)*chunking
    xa = x(i)*oodx
    j1 = xa
    j2 = j1+1
    f2 = xa-j1
    f1 = 1.-f2
    ci = charge(i)
    rho(j1,ci) = rho(j1,ci) + re*f1
    rho(j2,ci) = rho(j2,ci) + re*f2
  end do
!TM$ END TM_ATOMIC
end do
!$OMP END DO
!$OMP END PARALLEL
The figure illustrates the results of a transactional memory example. The graph shows the execution time in seconds (y-axis) against the number of threads (x-axis) with different lines representing different conditions: reduction, collision, and TM SHORT 6.

- **Great scaling due to few conflicts**: This indicates that as the number of threads increases, the execution time decreases, suggesting efficient handling of transactions with fewer conflicts.

- **Overhead**: This term likely refers to the additional time required due to transactional memory operations, which is represented by the upward trend at higher thread counts.

- **Scrubbing “ok”**: This indicates that the scrubbing process is functioning properly, maintaining the integrity of the data.
Transactional Memory Example – Results
short running mode

![Graph showing execution time vs. number of threads for different thread groups. The graph includes lines for reduction, collision, and TM short-6 C1, C2, C4, C8, C16, C32, C64, C128, C256, C512, and C1024.](image)
Transaction Memory Example – Results

long running mode
Transactional Memory – Summary

➢ To make TM work efficiently:
  ➢ think about conflict probabilities
  ➢ trade-off between entry-exit-overhead vs code-size
  ➢ tune the parameters