# Automatic Parallelization using TLS on BG/Q

Sriseshan S  
July 25, 2013

## Contents

1. Introduction .................. 2  
2. Infrastructure .............. 2  
3. Methodology .................. 3  
4. Implementation ............... 5  
   4.1 Method 1 ................. 5  
   4.1.1 llvm_bgqse_setup() ........ 6  
   4.1.2 llvm_bgqse_begin() ....... 6  
   4.1.3 llvm_bgqse_end() ........ 6  
   4.2 Method 2 ................ 7  
5. Analysis ..................... 8  
   5.1 Input Dependent Speedup ... 8  
   5.2 Speedup across multiple invocations of a loop ........ 10  
   5.3 Dependence analysis of execution time ................ 10  
6. Future Work .................. 12  
7. Acknowledgements ............ 12  
8. Appendix ..................... 13
1 Introduction

This report summarizes the work done and challenges faced during the 11 week MITACS Globalink research internship at the University of Alberta, Edmonton, Canada.

One of the biggest challenges in parallelization is the necessity of a thorough understanding of source code and of programmer’s support. Thread Level Speculation (TLS) enables compilers to optimistically render parallel code without having to ensure that concurrent threads are actually independent. Recently, a hardware implementation of TLS has been made available by IBM through its BlueGene Q (BG/Q) supercomputer. Hence, the goal of this project was to leverage TLS to achieve speedup of generic serial code through automatic parallelization on BG/Q, using the open source LLVM compiler.

This document includes the following background information and contributions:

• Some features of the hardware and software needed for TLS are described.

• Feasible compiler design strategies are listed with a description of the existing infrastructure.

• A detailed description of the implementation is made.

• Some analysis of the runtime behaviour of TLS is included.

• How the work can be extended in future, is listed.

2 Infrastructure

Thread Level Speculation enables parallelization of code that contains may dependence. The hardware and/or software guarantee(s) correctness of data; therefore, the compiler can optimistically parallelize code without having to conservatively check for dependence. Any given speculative thread can commit results from its speculative execution only after all its earlier threads have committed successfully. This way, if there is a conflict, the newer thread is squashed and re-executed.

To illustrate, consider the example shown in Figure 1. There is a Read-After-Write (RAW) dependency between threads 1 and 3 through variable $a$. In the event where $a$ is pointed to by a pointer $p$ in thread 1 and by a pointer $q$ in thread 3, where $p$ and $q$ point to the same memory location, no existing compiler can statically resolve this, making parallelization impossible. But TLS support makes parallelization a possibility.

Each BG/Q node has 16 cores and they share a common 32M L2 cache. This L2 cache is the point of coherence, and is multiversioned. Therefore, it can store different versions of the same address for different threads, thus enabling rollback of threads when conflicts are detected at runtime. However, it is the role of the compiler to ensure saving execution context before speculative execution because
registers are not buffered. Also, speculative execution on BG/Q is designed to run in jailmode, so that no irrevocable operation is done speculatively. For example, if during speculative execution, there is an access to an I/O device (like \textit{printf}) then a Jail Mode Violation is triggered.

LLVM is a widely used compiler infrastructure that is modular, powerful, yet open source. Middle end \textit{Passes} are written in C++ to perform code analysis and transform, and to provide source level information. A C/C++ port for LLVM on BG/Q is available and is called \textit{bgclang}.

\section{Methodology}

To achieve automatic parallelization using TLS on BG/Q, two approaches were considered:

1. [Standalone] Use LLVM for selecting candidate loops in the code as well as to insert calls to the TLS and POSIX threads runtime.

2. [Dependent] Make use of \textit{pragma} support given by the \textit{bgxlc_r} compiler for TLS and use LLVM to decide candidate loops for parallelization.

With a newly written TLS runtime library (i.e., the first approach,) complete control of the runtime system can be achieved, and, TLS becomes available to
the entire research community, thus making progress more probable. However, this method requires knowledge of or access to BG/Q’s internals so that the correct kernel calls are made. Theoretically, it is possible to reverse engineer this from the assembly code generated by \textit{bgxlc\_r}, and to then mimic it into LLVM. But, it turns out that the TLS runtime in \textit{bgxlc\_r} is very tightly coupled with the \textit{OpenMP} runtime, therefore, rewriting from assembly becomes challenging. Some sample code outlining how calls to the TLS runtime should be made was given, but it was incomplete and did not work for test cases tried. Attempts to complete this implementation eventually failed because of lack of availability of details regarding exact working of TLS on BG/Q. This method is outlined in Figure 2.

The second approach was more of a fallback/backup option as it directly uses \textit{pragma} support of \textit{bgxlc\_r}. Unlike in the first approach, use of LLVM is limited to analysis of code to select candidate loops for TLS and to insert pragmas. To this end, LLVM passes for inserting pragmas and to perform basic static analysis to extract loop sizes and iteration counts were written and tested. However, developing heuristics to select the loops (for which pragmas have to be inserted) is non-trivial, and requires some understanding and analysis of TLS behaviour. Therefore, some experiments to measure dependence of speedup on input and context were done. Also, an experiment to see if a toy program behaves as one would expect (\textit{wrt TLS}) from static analysis was done.

Section 4 outlines implementation details of both these methods. This is followed by a description of preliminary experiments (Section 5) and resulting suggestions for further development (Section 6).
4 Implementation

The description in this section is meant to go hand-in-hand with the source codes (link) available in the appendix (Section 8.)

4.1 Method 1

The best way to explain how calls to the TLS runtime are inserted is with an example.

Consider the following program:

```c
int main() {
    ...
    for (...) {
        <loop_body 1>
    }
    ...
    for (...) {
        <loop_body 2>
    }
    ...
}
```

A simplified version of the transformed code when the above is linked against the TLS library:

```c
int main() {
    ...
    llvm_bgqse_setup();
    for (...) {
        llvm_bgqse_begin();
        <loop_body 1>
        llvm_bgqse_end();
    }
    ...
    for (...) {
        llvm_bgqse_begin();
        <loop_body 2>
        llvm_bgqse_end();
    }
    purge();
    ...
}
```

Also, there is a feature to serialize chunks of iterations and run every $n^{th}$ iteration speculatively. In case of commit failure it falls back to sequential mode. This fallback is done by duplicating the loop body and executing from the
uncommitted iteration. Going back to speculative execution after sequential fallback has not been implemented because, as explained in Section 3, the TLS routines themselves are incomplete.

An LLVM pass is used to achieve the transformation shown above. The pass essentially iterates through all the BasicBlocks of the input program and inserts calls to these functions in the Intermediate Representation (IR.) These calls, however, do not yet parallelize the program, they just enable execution of a given thread speculatively. Therefore, the pthreads library is used to explicitly handle setup, creation and cancelling of concurrent threads. Though there is no constraint on the spawn order, an in-order spawn has been used as it is less likely to cause stalls for in-order commit, which is a TLS constraint (as mentioned in Section 2.)

Calls to the TLS runtime are described next.

4.1.1 llvm_bgqse_setup()
This function is executed once per program. It handles allocation of resources and sets the TLS mode and options to use.

4.1.2 llvm_bgqse_begin()
After successful initialization, this function starts speculative execution whenever called. First, it saves the context by storing the instruction pointer (IP), stack pointer (SP) and the table of contents register (TOC) into a thread local structure. This is done so that the kernel can perform a rollback in case of a conflict. Next, it tries to acquire the next available ID (0-127) for speculative execution of the thread, begins speculative execution and enters jailmode. This function returns a non zero value if any of these steps fail. Upon success, any code that follows this function is executed speculatively.

4.1.3 llvm_bgqse_end()
This function is called just after the loop body finishes executing speculatively. First, it exits jailmode and actually halts the speculative execution. Next, it acquires the ID from the thread local structure but before trying to commit, it waits for the previous thread to commit successfully (unless it is the first thread.) If commit fails, or if the previous thread did not commit successfully, the thread (and all subsequently spawned threads) get(s) invalidated and a non zero value is returned, hinting at sequential fallback. Upon successful commit, the ID is released.

However, there are two things that need to be taken care of:

1. Once llvm_bgqse_begin() returns, all subsequent code is treated to be speculative, including the call to llvm_bgqse_end(). This results in a corrupt call stack. Explicitly inlining these functions is thus essential.
2. In llvm_bgqe_begin(), while saving the registers’ context, it is unclear if storing just the IP, SP and TOC are sufficient. If it is required to store all live-in registers, then this can only be done at runtime, and is likely to cause further TLS overhead.

As mentioned in Section 3, inspite of incorporating the above, this implementation is incomplete and has bugs. Depending upon the program, this either works perfectly, or encounters segmentation fault, or faces continued invalidation of all threads causing sequential execution, or there is invalidation of no thread even in case of conflict; causing incorrect execution. As it looked like further input to understand the underlying system was required, Method 2 was attempted after this.

4.2 Method 2

This method is all about LLVM passes as pragma support of bgxlc_r is used. The role of the passes is to identify candidate loops for TLS and to insert pragma statements before these loops. The loop selection part is essential even in Method 1.

Given a candidate for loop, the goal of the pragma insertion pass is to transform the following:

```c
... 
for (...) { 
... 
```

into:

```c
... 
#pragma speculative for 
for (...) { 
... 
```

When compiled with the option qsm = speculative, bgxlc_r inserts calls like in Method 1 to the TLS runtime (but uses OpenMP for parallelization.) This can be done using LLVM’s RecursiveASTVisitor, which, as the name suggests, recursively visits the abstract syntax tree and simply inserts text just before a for loop, whenever it is encountered. However, even the most basic heuristic of selecting a loop based on number of iterations requires access to the IR, and recovering exact source code from the IR is tricky. Thus, given the filename and line number, it was decided to use a trivial bash script to do pragma insertion.

One static method of selecting a candidate loop for TLS is to set thresholds for loop body size and loop iteration count. While a very big loop body would increase the probability of conflict, a small loop body would make the TLS overhead seem large and a small iteration count would limit parallelism, all of which are suboptimal. An LLVM pass is hence used to obtain the iteration count of all countable loops and an estimate of their loop body sizes. This pass
uses debug information in the IR to retrieve information at the source code level. While this criterion is necessary, it is by no means sufficient as the following two sections show.

5 Analysis

5.1 Input Dependent Speedup

To determine variation in TLS speedup/slowdown of a given program when subject to different inputs, experiments were done on three SPEC2006 benchmarks, viz. fp_lbm, hammer and h264. A bash script was used to insert pragma calls to some of the loops and to insert calls to measure number of cycles using bgpm, which is a C interface to access hardware performance measurement counters on BlueGene. Speedup comparable to openMP speedup was observed wrt fp_lbm but slowdowns were observed for hammer and h264. Each candidate loop was executed speculatively and non-speculatively in separate runs and this was done for every input. The cycles measured during every such execution are for the execution of the corresponding loop alone.
As the speedup varies significantly depending upon the loop, a log boxplot of the absolute value of speedup is shown so that effect of input on speedup (or slowdown) can be seen. The inputs used for \texttt{fp\_lbm} were \textit{ref}, \textit{train} and \textit{test}; those for \texttt{hmmer} were \textit{bombesin}, \textit{leng100}, \textit{nph3} and \textit{retro} and those for \texttt{h264} were \textit{foreman\_ref\_encoder\_baseline}, \textit{foreman\_ref\_encoder\_main}, \textit{foreman\_test\_encoder\_baseline}, \textit{foreman\_train\_encoder\_baseline} and \textit{sss\_encoder\_main}.

While speedup is largely independent of inputs for \texttt{fp\_lbm}, this is not the
case for *hmmer* and *h264*. (It must be noted that a logarithmic scale was used for the boxplots to accommodate the high dynamic range of the values.)

### 5.2 Speedup across multiple invocations of a loop

It is possible that extent of TLS speedup/slowdown of a loop depends not only on the input but also on the calling context. To measure this, each loop invocation was executed speculatively and non-speculatively in separate runs and this was done for every input. However, for *fp_lbm*, it was observed for two of the loops that speedup was largely independent of context and input.

![Graph showing % SE speedup for HandlingOutFlow1 with 3 inputs](image)

### 5.3 Dependence analysis of execution time

In an attempt to measure the overhead caused in TLS due to rollbacks, the following loop was run:

```c
for (i = WINDOW; i < SIZE; i++)
    a[i] = a[i-WINDOW] + 1;
```

The expectation is that the degree of dependence changes with the value of WINDOW, thus causing different conflict probabilities. For example, if WINDOW is 0, then the loop is perfectly independent and for any non-zero value, conflicts are expected. The probability of encountering a conflict depends on the spawn policy of OpenMP (bgxlc_r uses the OpenMP runtime for parallelism), so a sweep from 0 to SIZE/10 was done for WINDOW, with steps of varied granularity. A SIZE of 10000000 was chosen so that when TLS runs long enough, the conflict overhead is not masked.

The execution time varied with WINDOW and this variation was compared against various counter values available viz. number of rollbacks, number of serializations, committed load misses, committed cacheable loads, L1-prefetcher misses, L2 hits, L2 misses, L2 lines stored to main memory and L2 lines loaded.
Table 1: Counter values for number of rollbacks, L1-prefetcher misses, L2 cache hits and misses measured upon TLS execution of a simple loop with varying degrees of dependence from main memory. Some of these values for some WINDOW values are shown in Table [1]. The number of cycles and execution time are directly proportional.

As the number of L2 cache misses is nearly constant across most of the range, the only monotonic relationship observed was that with the number of L2 hits.

The most puzzling column from Table [1] is that the measured number of rollbacks was nearly constant over all the values for WINDOW, despite the notion that different values of WINDOW mean different conflict probabilities. Attempts to ensure that the compiler doesn’t optimize away the dependencies were made by declaring `a` as a pointer instead of an array and by wrapping the above loads and stores into separate functions. Also, the initial values of `a` and the value of WINDOW were randomized so that information about these was available only at runtime. Yet, the measured number of rollbacks remained nearly constant, at ≈586, which is less than 0.01% of SIZE, where SIZE is roughly the total number of iterations. If this was a large value, then it could have been argued that the granularity of the underlying conflict detection in the L2 cache line is too coarse, thus making many false positives. However, as relatively small number of rollbacks were observed, two possible explanations exist:

1. The compiler serialized the loop or unrolled it effectively to hide the dependency.
2. The L2 cache line was evicted prematurely, i.e., before a conflict could be detected.

<table>
<thead>
<tr>
<th>WINDOW</th>
<th>Rollbacks</th>
<th>L1p Misses</th>
<th>L2 Hits</th>
<th>L2 Misses</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>585</td>
<td>81192</td>
<td>22958266</td>
<td>109512803</td>
<td>638504716</td>
</tr>
<tr>
<td>1</td>
<td>585</td>
<td>82795</td>
<td>25878786</td>
<td>109268345</td>
<td>920234122</td>
</tr>
<tr>
<td>2</td>
<td>588</td>
<td>90346</td>
<td>23288451</td>
<td>96260552</td>
<td>849992884</td>
</tr>
<tr>
<td>3</td>
<td>586</td>
<td>4079001</td>
<td>22460165</td>
<td>111945889</td>
<td>636492262</td>
</tr>
<tr>
<td>4</td>
<td>585</td>
<td>4039102</td>
<td>22632073</td>
<td>112311636</td>
<td>633979778</td>
</tr>
<tr>
<td>5</td>
<td>586</td>
<td>4079870</td>
<td>26683489</td>
<td>112933698</td>
<td>977863450</td>
</tr>
<tr>
<td>6</td>
<td>586</td>
<td>4070801</td>
<td>27767357</td>
<td>113041372</td>
<td>1075979998</td>
</tr>
<tr>
<td>7</td>
<td>586</td>
<td>4088353</td>
<td>27578418</td>
<td>113515866</td>
<td>1049858014</td>
</tr>
<tr>
<td>10</td>
<td>586</td>
<td>4081335</td>
<td>26828284</td>
<td>112952959</td>
<td>990203554</td>
</tr>
<tr>
<td>100</td>
<td>585</td>
<td>90071</td>
<td>25947190</td>
<td>108819788</td>
<td>938475532</td>
</tr>
<tr>
<td>1000</td>
<td>586</td>
<td>93373</td>
<td>21983026</td>
<td>110167998</td>
<td>624446560</td>
</tr>
<tr>
<td>10000</td>
<td>588</td>
<td>90806</td>
<td>26079182</td>
<td>104113107</td>
<td>948449626</td>
</tr>
<tr>
<td>100000</td>
<td>587</td>
<td>91694</td>
<td>21691725</td>
<td>78139164</td>
<td>602617072</td>
</tr>
<tr>
<td>1000000</td>
<td>585</td>
<td>90716</td>
<td>23893375</td>
<td>61895535</td>
<td>860489428</td>
</tr>
</tbody>
</table>
The \texttt{-qnounroll} option was tried, but no change in behaviour was observed. Thus, either the compiler chose to spawn very few iterations as parallel TLS threads; thereby serializing the loop to execute in large chunks, or, the spawn policy of OpenMP was suboptimal; causing frequent cache eviction. As this behaviour could not have been available statically, it is suggested that at least one profile run is made on the loop, as is described towards the end of Section 6.

6 Future Work

As and when further input \textit{wrt} BG/Q’s implementation of TLS is available, completion of Method 1 (Section 4.1) is highly desirable. However, irrespective of that, developing heuristics to select candidate loops is itself necessary, interesting and challenging. In the past, various static analysis based criteria have been suggested, including:

- Set thresholds on loop body size and loop iteration count. [Infrastructure to do this has been implemented in this work (Section 4).]
- Unroll small loops to meet the above threshold and to help load balancing across threads. [LLVM has a pass to unroll loops.]
- Set thresholds for data dependence count, misspeculation costs and use value prediction support.

Coupling a weighted scheme of the above with dynamic profiling can be seen as a long-term goal in this regard. One of the reasons why static analysis is not sufficient is evident from Sections 5.1 and 5.3.

Having mentioned that, the experiments outlined in the Section 5 cannot be generalized to form a golden heuristic as program profiles vary significantly. But, to illustrate, given the results obtained in Section 5.1 and Section 5.2, i.e., assuming input dependence and context independence, the following simple heuristic may be used: If there are multiple calls to a loop, execute the loop sequentially the first time, speculatively the second time and measure the speedup. If there is speedup, execute the remaining invocations of the loop speculatively, else, fall back to sequential execution. This way, it is less likely that we encounter the pitfall that was observed in Section 5.3.

7 Acknowledgements

I am extremely grateful to Prof. José Nelson Amaral and Arnaboy Bhattacharyya for their continuous support and guidance throughout this internship; they helped me learn a lot through this project as well as through the weekly CDOL meetings. I would like to thank MITACS Globalink and University of Alberta for enabling me to carry out this internship, and Indian Institute of Technology Madras for allowing me to avail this wonderful opportunity.
8 Appendix

A git repository was used for version control. Access can be granted if required. It is to be noted that printing, usually used as checkpointing, affects TLS run-time significantly, so care needs to be taken.

One version is made available in tarball downloadable from this link:

- The TLS library [Method 1: Section 4.1]
  - Has calls to the TLS runtime outlined into functions
    - src/lib.c
    - include/lib.h
    - bgclang assumed; compile with -O2

- Example of a complete transformed program [Method 1: Section 4.1]
  - Has calls to the TLS library, pthreads and sequential fallback
    - src/test1_tlspthread.c
    - bgclang assumed; compile and link with lib.o from above. (Use -pthread)

- Example build and run script for the above [Method 1: Section 4.1]
  - llvmRun.sh

- LLVM pass to insert function calls [Method 1: Section 4.1]
  - src/TLS.cpp
    - To build: place in a folder in llvm_src/lib/Transforms/; edit Makefile to include this folder; add a Makefile inside this folder similar to other transforms
    - To run: compile source code to .o using -emit-llvm and -c; link .o files using llvm-link to .bc; use opt to generate executable and lli to run it
    - This information is also available for the Hello pass on the LLVM website

- LLVM pass to insert pragmas [Method 2: Section 4.2]
  - src/Pragma.cpp
    - opt generates <input>_pragma.c to be used with bgxlc_r-qsmp=speculative

- LLVM pass to obtain loop info [Method 2: Section 4.2]
  - Prints out a file called loop.info with <file> <line#> <iteration count> <loop body size> for every for loop
- `src/Info.cpp`
- Compile with `-g` before passing to `opt`
- Use `-m2reg` option of `opt`

- Poster that was presented midway through the project, at the symposium held
  - `poster.pdf`