Automatic speculative parallelization of loops using polyhedral dependence analysis

Conference Paper - February 2013
DOI: 10.1145/2446920.2446921

CITATIONS 6

READS 87

2 authors:

Arnamoy Bhattacharyya
University of Toronto
18 PUBLICATIONS 59 CITATIONS

José Nelson Amaral
University of Alberta
163 PUBLICATIONS 1,066 CITATIONS

Some of the authors of this publication are also working on these related projects:

Thread Level Speculation View project
Unified Parallel C Compiler View project

All content following this page was uploaded by José Nelson Amaral on 27 March 2015.

The user has requested enhancement of the downloaded file.
Automatic Speculative Parallelization of Loops Using Polyhedral Dependence Analysis

Arnamoy Bhattacharyya  
University of Alberta  
Edmonton, Canada  
arnamoy@ualberta.ca

José Nelson Amaral  
University of Alberta  
Edmonton, Canada  
jamaral@ualberta.ca

ABSTRACT
Speculative Execution (SE) runs loops in parallel even in the presence of a dependence. Using polyhedral dependence analysis, more speculation candidate loops can be discovered than normal OpenMP parallelization. In this research, a framework is implemented that can automatically perform speculative parallelization of loops using Polly’s [15] polyhedral dependence analysis. The framework uses two different heuristics to find speculation candidates. The first heuristic allows loops with only may dependences to run speculatively in parallel while the second heuristic filters out cold loops and, using profile information, loops with actual run time dependences. The framework is fully automatic. Running SPEC2006 and the PolyBench/C benchmarks on the IBM BlueGene/Q [16] machine shows that the framework is able to discover more parallelization candidates than OpenMP parallelization and achieve better speedup.

Categories and Subject Descriptors
D.1.3 [Concurrent Programming]: [Parallel Programming]

General Terms
Performance

Keywords
Thread Level Speculation, Polyhedral Model, Auto-parallelization

1. INTRODUCTION
With the rapidly growing usage of many-core computers, the need of correct and efficient parallel programs that can fully utilize the available hardware parallelism is also increasing. Auto-parallelization is an ideal choice for the programmers but is very challenging to achieve as the compiler has to prove at compile time the existence or non-existence of dependencies in the code. To generate a better parallel version of the program automatically, the dependence analysis has to be strong. Many static dependence analysis techniques [19, 25, 26] are widely used to find available parallelism in programs. In the polyhedral model [15, 23], each execution instance of a statement is represented as an integer point in a well-defined space called the statement’s polyhedron. By analysing the relation between the statement instances, polyhedral model can reason about the dependences among the program statements. Polyhedral dependence analysis normally operates on SCoPs (Static control parts) that are code fragments with some special features. Polly [15], the polyhedral optimizer in LLVM [20], detects loops with certain properties as SCoPs.
Auto-parallelization of loops is also a widely studied topic. [4, 5, 12, 14]. After the dependence analysis performed by the compiler, a typical auto-parallelizer based on polyhedral model [15] will parallelize SCoPs (loops) of the program only if there is no must and may dependences in the loop (A may dependence is a dependence that may occur at run time but a decision to run the loop in parallel cannot be made at compile time due to insufficient alias information or due to the unknown resolution of conditional statements). Speculative execution (SE) [21, 27] executes program parts in parallel even in the presence of may dependences. In case of a dependence occurring at run time, SE ensures correctness through special (hardware/software) mechanisms of commit and rollback. In this way SE alleviates the compiler’s job of performing a sophisticated dependence analysis to find speculative parallelization candidates. Even if the compiler can’t prove dependence at compile time, a loop can be executed speculatively in parallel. Not all loops should be executed speculatively through SE because there are overheads in SE that can slow down some loops. There is overhead for launching new threads and for rollbacks when speculation fails. For launching new threads, a typical SE system has to store the context of the registers in some speculative buffer so that the system can be restored to a previously consistent state when speculation fails. If dependences frequently occur during SE, the gain from parallelization can be overshadowed by the overhead of rollback and the result is a slowdown of the application. Therefore, a profitable application of SE still needs a dependence analysis that can analyze the different may and must dependences in the loops.
1.1 Research Questions

Given the dependence analyzing power of the polyhedral model and the correctness guaranteed by SE even in the presence of dependences, this research tries to investigate whether we can achieve better performance if loops with may dependences are allowed to execute in parallel speculatively. In summary, this research was conducted to find answers to the following research questions:

1. Can the result from polyhedral dependence analysis be used to find loops that are candidates for speculation in a program?
2. Will relaxing the OpenMP constraint and letting loops with may dependences execute in parallel result in performance improvement?
3. Can the elimination of loops with actual run-time dependence with the help of profiling and cold loops give us better speculation candidates?

The rest of the paper is organized as follows. Section 2 gives background information on the Polyhedral Model and Speculative Execution (SE). Section 3 describes the heuristics that are used by the framework. This section also describes how the heuristics can be modified to find better speculation candidates. Section 4 describes the implementation details of the framework in LLVM. Sections 5 and 6 gives details about the hardware platform and the experimental results. Section 7 presents some of the previous work in the field and Sections 8 and 9 presents the conclusion and the planned future work.

2. BACKGROUND

Most compilers use the Internal Representation (IR) of a program to perform optimizations. IR includes syntax trees, call trees, control-flow graph. In those representations, each statement of a program is represented only once, even if the statement is executed many times (e.g. loops). This type of representation is not sufficient for dependence analysis that takes into consideration dynamic statement instances. Due to compile-time constraints and lack of proper algebraic representation of loops, traditional program transformations can’t adapt the schedule of statement instances of a program. If the data dependences in the loops are non-uniform [22] or if the profitability of a transformation is unpredictable, compilers typically cannot apply loop transformations.

2.1 Polyhedral Compilation

The Polyhedral model offers a flexible and expressive representation for loop nests with statically predictable control flow. Such loop nests in the program are called static control parts (SCoP) [8, 12]; their control and data flow information are represented as the following three components:

- **Iteration domains** - They are used to consider each dynamic instance of a statement through a set of affine inequalities. Each dynamic instance of a statement $S$ is denoted by a pair $(S, i)$ where $i$ is the iteration vector and it contains values of the surrounding loop indices of the statement. If the loop bounds are affine expressions of outer-loop indices and global parameters, then the set of all iteration vectors $i$ for a given statement $S$ can be represented by a polytope $D_s = \{ i | D_s \times (i, g, 1)^T \geq 0 \}$. It is called the iteration domain of the statement $S$, where $g$ is the vector containing all the global parameters of the loop whose dimensionality is $d_g$.

- **Memory access functions** - These functions are used to represent the locations of data the statements are accessing. In SCoPs, memory accesses are normally performed on array references. For each statement $S$ two different sets are defined - $R_s$ and $W_s$ of $(M, f)$ pairs. Each pair represents a reference to a variable $M$ being written or read by a statement $S$. $f$ is the access function that maps iteration vectors in $D_s$ to the memory locations in $M$.

- **Scheduling function** - The set of statement instances that are to be executed dynamically are defined by their iteration domains. But the execution order of each statement instance with respect to other statement instances are not described by this algebraic structure [11]. A scheduling function $S$ is defined for each statement $S$ that maps instances of $S$ to totally ordered multidimensional timestamps (vectors).

Polyhedral dependency analysis uses the above three components to find dependences and apply correct transformations to the SCoPs.

2.2 Speculative Execution

- **Overview** - A common use of SE is to divide a sequential program into parallel thread in a way that does not violate the sequential semantics of the program. There should be an execution order among the threads because SE guarantees correctness. Therefore, there can be two types of threads- predecessor and successor threads. The safe (or non-speculative) thread precedes all speculative threads. There is special hardware (or software) support that checks that no inter-thread dependence is violated. If there is a violation, the incorrect threads are squashed, the system rolls back to a previously stored steady state, and the threads are re-executed. After a number of retries, the threads become sequential.

- **Inter-thread Data Dependence** - Data dependences are typically captured by monitoring the data written and the data read by individual threads. A data dependence violation occurs when a thread writes to a location that has already been read by a successor thread. Dependence violations lead to the squashing of thread, which involves discarding the side effects produced by the thread being squashed.

- **Buffering of States** - Stores performed by a speculative thread generate speculative state that cannot be merged with the committed state of the program because the value may be incorrect. The speculative state is stored separately, typically in the cache of the processor. Speculative writes are not written back to memory until the speculative thread commits. If a dependence violation is detected, the speculative state is
discarded from the cache. Otherwise, when the speculative thread commits, the state is allowed to propagate to memory. When a non-speculative thread finishes execution, the thread commits. Committing informs the rest of the system that the state generated by the task is now part of the safe program state.

- **Data Versioning** - A thread has at most a single version of any given variable. However, different speculative threads running concurrently in the machine may write to the same variable and, as a result, produce different versions of the variable. Such versions must be buffered separately. Moreover, readers must be provided the correct versions. Finally, as threads commit in order, data versions need to be merged with the safe memory state also in order to ensure correctness.

- **Multi-Versioned Caches** - A cache that can hold state from multiple tasks is called multi-versioned [6, 13, 27]. There are two performance reasons why multi-versioned caches are desirable: they avoid processor stalls when there is imbalance between tasks, and they enable lazy commits. In BlueGene/Q, each different version of a memory address can be stored in a different way on the L2 cache. When a write occurs for a speculative thread, the L2 allocates a new way in the corresponding set for the write. A value stored by a speculative write is private to the thread and is not made visible to other threads. The value is made visible to other threads when a thread commits and is discarded upon a thread squashing. In addition, the L2 directory records, for each access, whether the access is read or written, and whether the access is speculative. For speculative accesses, the directory also tracks which thread has read or written the line by recording the speculation ID used by the thread to activate speculation. Tracking this information enables the hardware to detect conflicts among threads and also between speculative and non-speculative thread. [29]

### 2.2.1 SE in BlueGene/Q

The L2 cache in the BG/Q chip is the point of coherence. The L2 cache is divided into 16 slices. Each slice is 16-way set-associative. The cache is a multi-versioned. If more than one thread is modifying the same value, each thread stores different versions of the value in separate ways of a cache so that the value is only visible to one particular thread. The value is made visible to other threads during the time of commit of a transaction and is discarded when a transaction aborts. The hardware uses thread IDs to distinguish between younger (successor) and older (predecessor) threads. The hardware uses unique speculative IDs to associate a memory operation with a thread. BG/Q provides 128 speculative IDs for transactions. If a program runs out of IDs then the start of a new transaction blocks until an ID becomes available [29].

### 3. HEURISTICS

In this section two simple heuristics used to find speculation candidates are described. The first heuristic relaxes the constraints of OpenMP parallelization by allowing loops with *may dependences* to be speculatively executed in parallel. The second heuristic is an extension of the first heuristic to filter out loops with run-time dependence (with the help of profiling) and cold loops (because they are not good candidate for speculation and may cause slowdown).

#### 3.1 Heuristic 1

According to the first heuristic, for being a speculation candidate a SCoP (loop) should have only *may dependences*. The goal of this heuristic is to relax the constraint for OpenMP parallelization (OpenMP does not parallelize loops with *may dependences*) and find more parallelization candidates. The hope is that the *may dependences* will not materialize at run time, thus resulting in speedup.

#### 3.2 Heuristic 2

Heuristic 1 allows loops with *may dependences* to execute in parallel. There can be two cases where the overhead of speculation can still negate the gain from parallelism. In heuristic 2, the two criteria for filtering are based on two different overheads. The first criteria considers the overhead from mispeculation and recovery while the second criteria considers the overhead from thread creation and storing the program state so that the system can be rolled back to a consistent state in case of mispeculation. In this way loops that can not be benefited from SE are filtered.

1. In the loops where *Polly* reports only *may dependences*, the memory accesses are profiled for a training run for some input. If the training run shows that the *may dependences* are materializing into *true dependence* (Here true dependence means dependence occurring at run time, not *read after write* (RAW) dependence) at run time, the loop is not parallelized.

2. If the execution time of the loop is less than some threshold as compared to the total execution time of the program, the loop also is not speculatively parallelized. As in this case, the overhead from speculation can neutralize or worse, negate the gain from parallelism. This threshold is set as 20% of the total execution time because experimental evaluation shows that allowing the speculative execution of smaller loops for the benchmarks leads to slowdown.

### 4. THE FRAMEWORK

The framework is implemented in the LLVM [20] compiler infrastructure. Polyhedral dependence analysis is already implemented as a part of the LLVM project and it is called Polly. [15]

The framework is depicted in Figure 1. First the source-code file to be optimized is compiled with LLVM keeping the debug information in the bitcode (with -g option). The debug information is necessary for source-code instrumentation. Two LLVM passes are run on the generated bitcode. The *printBlocks* pass prints out all the basic blocks in the source file and their corresponding file name and line numbers. Polly’s dependence analysis pass *Polly-dependences* is also run on the bitcode. This pass provides information about the SCoPs and the *may* and *must dependences* in them. This dependence information is extracted from the output and kept into temporary storage. An analyzer program takes the information generated by the above two passes as input,
performs a three-way mapping (SCoPs with basic block information, basic block with file name and line numbers information and SCoPs and dependence information), performs analysis based on which heuristic to use and spits out information about the file names and line numbers where the speculative pragmas to be inserted. A script is run that reads the file name and line numbers and instruments the code accordingly. For the benchmarks tested, the number of loops never exceeds 6000 (the maximum number of candidate loops for speculation are much less than that) and the temporary storage necessary was less than 5 MB.

The instrumented source code is then compiled with the bgxlc compiler specific for the BlueGene/Q machine) compiler to generate the executable that can be run on the BlueGene/Q machine. The framework is fully automatic.

5. EXPERIMENTAL EVALUATION

The framework is implemented in the LLVM [20] compiler infrastructure. Experimentation was performed on two different set of benchmarks - SPEC2006 benchmarks [1] and the PolyBench/C benchmarks [24]. SPEC2006 was chosen because it has been used by other researchers for the evaluation of speculative execution. All the SPEC2006 benchmarks are not reported because some of them don’t run successfully on BlueGene/Q. PolyBench/C benchmarks were chosen because they are suitable benchmarks for polyhedral analysis. Table 1 shows the hardware details of a BlueGene/Q chip. The SPEC2006 benchmarks were run with the ref input and the PolyBench/C benchmarks were run using their default inputs. For calculating the speedups, each benchmark was run 10 times for a given input and the average running time from the 10 runs were taken. Also the 95% confidence interval is shown in the bar chart for the 10 runs.

In Figure 2, most of the SPEC2006 benchmarks achieve a speedup over the optimized sequential version. lbm contains loops with no inter-thread data dependences, but these dependences are not statically provable by the compiler and that’s why they are not parallelized by OpenMP. This benchmark can be greatly benefited by the speculative execution and obtains the highest speedup because these dependences don’t materialize at run time.

6. RESULTS

This section describes the experimental results. First the effect of heuristic 1 on the SPEC2006 and the PolyBench/C benchmarks is described. Then a comparison is made between OpenMP parallelization and the parallelization done by the framework following heuristic 1. After that the effect of heuristic 2 is shown. Effect of different alias analysis on the polyhedral dependence analysis and the scalability of the SE parallelization are also shown.

6.1 Heuristic 1

6.1.1 Spec2006 Benchmarks

In Figure 2, most of the SPEC2006 benchmarks achieve a speedup over the optimized sequential version. lbm contains loops with no inter-thread data dependences, but these dependences are not statically provable by the compiler and that’s why they are not parallelized by OpenMP. This benchmark can be greatly benefited by the speculative execution and obtains the highest speedup because these dependences don’t materialize at run time.

gobmk has many loops with small iteration counts. Small loops are not good candidates for speculative execution because the thread creation overhead negates the impact of parallel execution and we get a slowdown. These loops are later filtered out by heuristic 2.

The speedups reported in Figure 3 indicate that the PolyBench/C programs can be divided into two classes according to the effectiveness of thread-level speculation. Class 1 contains programs 2mm, 3mm, correlation, covariance, doit.

1Later experiments with the highest level (-05) of optimization shows that the performance improvement delivered by the -05 optimization level in the bgxlc compiler is as good as TLS for some benchmarks, while for some benchmarks, TLS has a modest improvement over code optimized with -05. We are continue to study the performance tradeoff between the compiler optimizations at higher levels and TLS.
Table 1: Configuration of a BluGene/Q chip

<table>
<thead>
<tr>
<th>#Processors</th>
<th>17 (16 User and 1 service PowerPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multithreading</td>
<td>4-way Multithreaded</td>
</tr>
<tr>
<td>Clock</td>
<td>1.6GHz</td>
</tr>
<tr>
<td>L1 I/D Cache</td>
<td>16KB/16KB</td>
</tr>
<tr>
<td>Peak Performance</td>
<td>204.8 GFLOPS 55W</td>
</tr>
<tr>
<td>RAM</td>
<td>16 GB DDR3</td>
</tr>
<tr>
<td>Multiversioned Cache</td>
<td>Support for Transactional Memory and Speculative Execution</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Centrally shared, 32 MB</td>
</tr>
<tr>
<td>Chip-to-chip networking</td>
<td>5D Torus topology + external link</td>
</tr>
</tbody>
</table>

Figure 2: Speed up of the instrumented SPEC2006 benchmarks over the optimized sequential version on a 8 node cluster. Most of them gets a speedup, the maximum being lbm due to the presence of a large parallelizable loop. gobmk suffers a slow down because of the presence of many loops with small iteration count. sjeng also offers a slow down because the may dependences actually occur during run time. Dependence profiling can be done to eliminate those candidate loops with may dependences for sjeng.

pen, gemm that achieves speed up on a speculative execution and Class 2, containing gramschmidt, jacobi-2d-imper, lu, ludcmp, seidel experiences a slow down. In the Class 2 PolyBench/C programs the loops parallelized are very small and constitutes a very small portion of the overall program execution (mainly initialization arrays). Therefore the overhead for thread creation in the speculative execution negates the performance achieved from the parallel execution of these loops. Table 2 shows the percentage of the whole program execution time the parallelized loops take for the PolyBench/C benchmarks. For seidel the coverage is only 0.04% and therefore the loops are not good speculation candidates for this benchmark. The cold loops are eliminated by heuristic 2.

6.2 Comparison with OpenMP Parallelization

Heuristic 1 relaxes the constraint of OpenMP parallelization and allows loops with may dependences and no must dependences to speculatively run in parallel too. As seen in Table 3, heuristic 1 was able to find more parallelization candidates than OpenMP parallelization for the PolyBench/C benchmarks. Figure 4 shows the speedup gained from OpenMP parallelization and speculative parallelization for the PolyBench/C benchmarks. For 2mm and 3mm, speculative optimization does not give better speedup because

Table 2: Coverage of the loops parallelized in the PolyBench/C benchmarks. Most of them takes significant portion of the program execution time, but for seidel, the loops parallelized are very cold and therefore the overhead from speculation negates the gain from parallelization.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Percentage of Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2mm</td>
<td>&gt; 99.99</td>
</tr>
<tr>
<td>3mm</td>
<td>&gt; 99.99</td>
</tr>
<tr>
<td>correlation</td>
<td>&gt; 99.99</td>
</tr>
<tr>
<td>covariance</td>
<td>&gt; 99.98</td>
</tr>
<tr>
<td>doitgen</td>
<td>&gt; 99.99</td>
</tr>
<tr>
<td>gemm</td>
<td>&gt; 99.99</td>
</tr>
<tr>
<td>gramschmidt</td>
<td>99.97</td>
</tr>
<tr>
<td>jacobi-2d-imper</td>
<td>&gt; 99.98</td>
</tr>
<tr>
<td>lu</td>
<td>99.96</td>
</tr>
<tr>
<td>ludcmp</td>
<td>99.95</td>
</tr>
<tr>
<td>seidel</td>
<td>0.037</td>
</tr>
</tbody>
</table>

These results confirm the finding of Kim et al. [18]
Figure 3: Speed up of the instrumented PolyBench/C benchmarks over the optimized sequential version on a 12 node cluster. The last five benchmarks suffers a slow down because of launching threads for loops with small iteration count and also parallelizing loops in the innermost level (finer granularity).

Table 3: Number of Loops parallelized by OpenMP parallelism vs Speculative parallelism using heuristic 1. As heuristic 1 allows loops with may dependences and no must dependences to be executed in parallel as well, the heuristic was able to find more parallelizable loops.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>OpenMP</th>
<th>Speculative</th>
</tr>
</thead>
<tbody>
<tr>
<td>2mm</td>
<td>20</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>3mm</td>
<td>27</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>correlation</td>
<td>13</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>covariance</td>
<td>11</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>doitgen</td>
<td>18</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>gemm</td>
<td>13</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>gramschmidt</td>
<td>10</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>jacobi-2d-imper</td>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>lu</td>
<td>8</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>lducmp</td>
<td>12</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>seidel</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

no new loops are discovered by the framework. For the last five benchmarks, OpenMP performs better than speculative optimization because the loops parallelized by heuristic 1 in the benchmarks are pretty small and they take very small portion of the benchmark execution time. Therefore executing them speculatively causes the overhead from SE to negate the gain from parallel execution. The slowdown in these benchmarks were the motivation for heuristic 2. Most of the time the framework outperforms OpenMP for the SPEC2006 benchmarks, as can be seen in Figure 5.

6.3 Heuristic 2

6.3.1 Profiling and Filtering hot loops
To tackle the slowdowns, the reported may dependences...
were profiled to see whether the dependences occur at run time. Also the loops that does not take significant amount of the program execution time were not speculatively parallelized because they negate the gain of parallelization due to thread creation overhead.

The results shown in Figure 6 indicate that the benchmarks that were experiencing slowdown as compared to OpenMP performs equal or better when heuristic 2 is used. Gramschmidt and sjeng performs better than OpenMP as the framework was able to discover more parallel loops than OpenMP via profiling than static dependence analysis. This indicates that by extending heuristic 1, equal or better performance can be achieved as compared to OpenMP.

6.4 Scalability

For measuring the scalability of the speculatively parallelized benchmarks, the instrumented code were run on increasing number of cores. The experimentation using 64 cores was performed on a four node cluster because each node in BG/Q contain 16 cores (each core is 4-way SMP). Figure 7 describes the scalability for the SPEC2006 benchmarks. Speedups flatten out for more than 16 cores for lbm, milc, povray, bzip2. For namd, hmer, sjeng, using more than 16 cores and for mcf, using more than 8 cores achieves a slowdown due to the increasing communication cost between the cores. sphinx3 does not show much variation in performance because the number of loops executed speculatively in parallel are not big enough to scale.

7. RELATED WORK

The literature on auto-parallelization of programs is fairly extensive.

PLUTO [4] is an automatic parallelization tool based on the polyhedral model. PLUTO uses polyhedral model information to perform high-level transformations on affine loop nests. For both coarse-grained parallelism and data locality, PLUTO provides source to source transformation of C programs. The core transformation framework mainly finds affine transformations for efficient tiling and fusion. PLUTO generates OpenMP code from sequential C programs automatically.

Baskaran et al. address the issues on developing a compiler framework for automatic parallelization and performance optimization of affine loop nests for GPGPUs [2]. They also discuss an approach for program transformation for efficient data access using a polyhedral compiler model.

Pouchet et al. goes beyond the restriction of the polyhedral model to statically predictable, loop-based program parts [28]. They remove the limitation, and allow to operate on general data-dependent control-flow by embedding control and exit predicates as first-class citizens of the algebraic representation.

Boulet et al. summarizes the details of earlier parallelization algorithms which used restricted input and were based on weaker dependence analysis than the polyhedral model [5]. According to them, there are two kinds of automatic parallelization techniques based on the polyhedral model (1) scheduling/allocation-based, and (2) partitioning-based. The work of Feautrier [10, 11], Darte-Vivien [7] and Griebl [14] (to some extent) can be considered in the former class, while Lim/Lam’s approach falls into the later class.

Griebl et al. describes an integrated framework that optimizes for both locality and parallelism with space and time tiling [14]. After a finding a schedule for the loop, they treat tiling as a post-processing step. When schedules are used, the framework can tile the inner parallel (space) loops. If
better performance than OpenMP for code generated at optimization level -O0.

In summary, the speculative execution model, that allows us to execute loops in parallel even in the presence of a dependence, can give better performance than OpenMP parallelism if the speculation candidates can be wisely chosen.

9. FUTURE WORK

In heuristic 2, the loops that have actual run time dependence are filtered out. Berube et al. shows that the program’s run time behaviour changes with the change of input to the program [3]. Whether such changes of behaviour also affects the materialization of may dependences remains to be investigated. If for a number of inputs, the loop has a high probability of being independent, the loop can be executed speculatively in parallel. There is ongoing research to capture that dependence behaviour of the loop and modify the heuristic accordingly.

Acknowledgments

I would like to thank Argonne National Laboratory, specially Hal Finkel and Jeff Hammond for giving access to the IBM BlueGene/Q computer and answering questions about the machine and the different tools. I would also like to thank my mentor Matthew Gaudet for guiding me throughout the course of this research.

10. REFERENCES