Exploring Compiler Optimizations for Parallel Architectures

Christopher Wood
Introduction to CS Research
caw4567@rit.edu

I. Introduction

In recent years interest in parallel computing has increased substantially. Members of the scientific community have a particularly acute interest in parallel computing because of its ability to process computational problems much faster than sequential computing devices can. As such, parallel programming has become an increasingly popular topic and has fostered research interests focused on the generation of code that runs optimally on parallel architectures.

Currently, there are several obstacles that face this shift from sequential to parallel programming. Firstly, most of the programmers in industry today grew up alongside the concept of sequential programming and are just now being introduced to parallel programming. Unfortunately it would be too time, money, and resource consuming to teach every programmer how to program for parallel architectures. Secondly, parallel programming has long been recognized as more time-consuming, error-prone, and harder to debug than sequential programming [4].

Most importantly, it would be impractical to rewrite existing software for parallel architectures. Therefore, programmers turn to parallel compilers to transform their sequential code into optimal parallel code for a specific architecture. Parallel compilers are very powerful tools because they allow programmers to stick to their sequential programming techniques and still produce quality parallel code that runs optimally on parallel architectures. Unlike sequential compilers that translate a high-level language into machine language, parallel compilers are faced with the added task of converting sequential code into parallel code prior to the high-level to machine code transformation stage. A parallel compiler must handle this task efficiently while trying to optimize for parallelism.

In the parallel paradigm there are several different types of parallelism. However, for the scope of this paper I focus on thread-level parallelism (TLP) and instruction-level parallelism (ILP). There are many techniques used by parallel compilers to maximize these two forms of parallelism, such as loop transformations and instruction scheduling. These two techniques, although small in scope, play a large role in parallel compilers and their ability to translate sequential code effectively.

In order to understand parallel compilers it is important to understand the details of these techniques as they pertain to the different types of parallelism. In this paper I will first discuss loop transformation and instruction scheduling techniques, synthesize the techniques to compare and contrast them, and finally discuss the weaknesses of parallel compilers in hopes of providing jump-off points for future research.

II. Compiler Optimizations

Typical high-level compiler optimization techniques such as loop, data-flow, code-generator, and functional language optimizations are effective for sequential computing. However, these techniques must be altered in the paradigm of parallel computing where each machine architecture can execute multiple instructions simultaneously. From a compiler standpoint, instruction scheduling and loop transformations are two important techniques that must be considered when designing a compiler for a parallel architecture. Research has been done on individual architecture optimization techniques for instruction scheduling and loop transformations and the resulting techniques and experimental results show a lot of promise for the field of parallel compiling techniques.

A. Loop Transformations

Parallel compilers that optimize code with loop transformations often use techniques such as fusion and tiling. Loop fusion is simply the process of creating a single loop nest from a series of loops. Its purpose is to decrease the amount of time between sequential accesses to the same data within a nested loop, which enhances the locality by increasing the chance that the data being accessed will remain in the cache. Fusion also helps reduce synchronization between loops running on parallel machines. Since multiple loops are merged together to form one larger, nested loop the synchronization that once existed between the many separate loops is gone because they are transformed into a single entity. Much like other transformation techniques, loop-fusion is not always legal. Even when it is legal, it can degrade the level of parallelism in the new loop if dependencies emerge between the loops being fused together.

Research in the area of loop-fusion has been conducted to produce techniques that allow the fusion of loops even when loop dependencies are present, maintain parallelism and parallel execution with fused loops, and even eliminate cache conflicts brought about by fused loops. One promising technique to help improve loop-fusion is the shift-and-peel technique, which maintains parallelism and improves locality [9]. However, shift-and-peel transformation is unique in that it does not always produce a single-loop nest and it only requires uniform dependencies between the loops

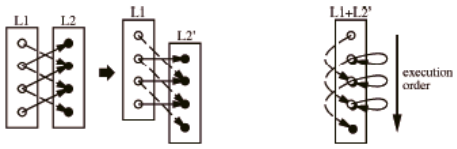

Fig. 1. The shift portion of the shift-and-peel technique is depicted. Iteration spaces are shifted in order to turn cyclic paths into acyclic ones for execution. Figure retrieved from [9].

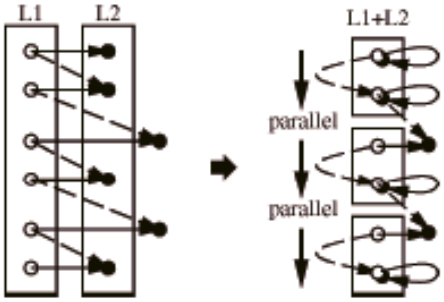

Fig. 2. The peel portion of the shift-and-peel technique is depicted. Loop-carried dependences that cannot be fused together are peeled off the resulting loop and executed outside of the fused loop during the execution stage. Figure retrieved from [9].

to be merged. The shift transformation basically consists of making backward dependences loop-independent in the fused loop by shifting the iteration space with respect to the difference between the dependences as shown in Figure 1. This shift then allows the loops to be legally fused without incurring any penalty from cyclic paths.

The idea behind the peel transformation is to remove serializing dependences resulting from the fusion of loops. This technique consists of pinpointing the iterations from the original loops that become sinks of cross-processor dependences, which are essentially loop-carried dependences from which the sink and source iterations are executed by different processors, and peel them from their respective iteration spaces prior to fusion as shown in Figure 2. These peeled iterations are then executed upon completion of the fused loop, which will ultimately maintain the parallelism of the original loops but also enhance their locality. Results of loop fusion on parallel architectures have been discussed in [9], showing that the speedup ratio of programs such as LL18, calc, and filter on parallel architectures is around approximately 2.0. As the number of processors increases, the speedup gap between un-fused and fused loop speedups widens at a constant rate with loop fusion always outperforming un-fused loops.

One problem with loop-fusion is cache trashing that is caused by conflicts among data items in the cache that consequently cause cache misses. Self-conflicts, which are conflicts that occur between data elements of the same array, and cross-conflicts, which occur between elements of different arrays, are the two types of conflicts that generally cause cache misses. In terms of loop-fusion, cross-conflicts are a more significant cause for concern due to the fact that

```
do i = 0, 3
  do j = 0, 3
    a[i, j] = a[i − 1, j] + a[i − 1, j − 1]
  end do
end do
```

(a) Loop nest 1

```
do ii = 0, 3, 2
  do jj = 0, 3, 2
    do i = ii, min(ii+1, 3)
      do j = jj, min(jj+1, 3)
        a[i, j] = a[i − 1, j] + a[i − 1, j − 1]
      end do
    end do
  end do
end do
```

(b) Tiled loop nest of (a)

Fig. 3. The result of tiling a simple loop nest in (a) is shown in (b). Figure retrieved from [7].

in a large fusion of many loops there are likely to be many different arrays brought together as well. This type of conflict is likely to occur during loop-fusion when portions of the different arrays involved point to the same portion of the cache.

Cache partitioning is a way of avoiding these conflicts and cache trashing by which precise adjustments to the array layout in memory, with some padding or gaps in the cache between each array, is implemented. The cache is essentially partitioned into subsections, each of which is designated for one array in the loop. This way no conflicts will arise because each array acts on its own section in the cache during the loop execution so parallelism is maintained and cache trashing is decreased. The results of cache partitioning in [9] show that as the number of processors increase in a parallel architecture the speedup increases almost at a linear rate. For example, in an architecture with 8 processors it is shown that the speedup ratio was as high as 4.9, which is a remarkably high amount for a simple optimization technique.

Loop tiling is another important form of loop transformation that takes advantage of data reuse and locality in the cache to maintain parallelism and improve locality. Tiling essentially transforms a loop nest of depth n into a loop nest of depth $2n$ where the outer loops control the order of execution of the inner loops within a tile. An example of tiling is shown in Figure 3, where the loop into tiles of size $2x2$ that cover the same iteration space. Fundamentally, the process of loop tiling transforms the original, n-dimensional iteration space loop into n-dimensional tiles of the same size and shape that cover the same iteration space [7]. The challenge with implementing an optimal tiling algorithm is that there are several conditions that affect the performance and legality of a tiled loop. Specifically, loop-carried dependences between iteration points of formed tiles, tile size selection, and distribution of tiles to threads all affect the performance enhancements of tiling [7]. Ensuring the legality of loop tiling should be the more important objective for any parallel compiler and in order to do so they must consider loop-carried data dependences between the iteration points of tiled loops. Based on the tile formation algorithm, acyclic or cyclic chains of tile dependences, which are directly related

to the loop-carried dependences between iteration points of tiles, can form throughout the tiles. Any cyclic chain of tile dependences is considered to be illegal and the compiler must take the appropriate steps to adjust the tile to remove the chain.

The second most important consideration for a loop tiling algorithm is the shape of a tile. The execution time of a tile is dependent on the shape of the tile and the number of iterations included in that tile depending on the architecture. Also, the volume of the tile is proportional to the iteration space of that tile, which implies that the execution time of a tile is heavily dependent on a tile's volume. When trying to select the volume of a specific tile there are ultimately two choices. That is, either single level or multi level tilings. When tiling for single levels of memory in the memory hierarchy it has been shown that the optimal tile shape is achieved by using the largest legal tile slope that results in the longest vertical path of dependent tiles. Quite oppositely, when tiling for multiple levels of memory the optimal tile size is one with a smaller tile slope because more tile dimensions are involved [6]. In a general case, however, the tile shape should be chosen so that the coordinates for the initial and final synchronization points, which are the end points of the longest path of dependent tiles in an iteration space, are equal in as many dimensions as possible. This is difficult to ensure because any increase in tile slope in a dimension consequently decreases the distance between the synchronization points in that dimension. Therefore, the tile slopes in every dimension must be adjusted appropriately so that the synchronization points become as close to each other as possible.

Finally, the distribution of tiles to threads must be addressed by a parallel compiler. The principle of locality must be considered during this stage because the cache plays an important role in the performance of loop transformations. There are two main types of distributions, namely, blocked and cyclic tiling. When working with a parallel, multithreaded architecture it is more advantageous to choose a cyclic tiling scheme because a single tile can be shared by all threads, which allows loop iterations to be split up in a cyclic fashion among the threads. This takes advantage of data reuse and thus improves the overall execution time of the tiled loops. The performance enhancements of loop tiling using a cyclic scheme are quite significant on parallel architectures. For example, on a Simultaneous Multithreading architecture with 8 threads, cyclic parallelization had a speedup ratio of at least 100% for every application tested with [8]. In larger parallel architectures cyclic tiling still proves to be the most effective distribution scheme because the locality of each processor and interprocessor communication are maximized, which are the intended results.

B. Instruction Scheduling

Instruction scheduling is the process of assigning operations to the functional units of a machine during each CPU cycle. The basic problem with instruction scheduling to maximize ILP is organizing operations efficiently so that they fill as many functional units as possible with little horizontal and vertical waste. Over the past several years researchers have attempted to formulate an efficient and optimal solution to the instruction scheduling problem. One notable technique is known as schedule-and-improve, which relied on local code compaction and region selection for instruction scheduling. However, this technique was ultimately flawed because it relied too much on arbitrary decisions when selecting blocks for scheduling. Therefore, researchers began to look for an alternative approach for instruction scheduling and soon came upon the idea of trace scheduling, which has dominated the parallel computing scene ever since it's unveiling in 1979 [1]. Trace scheduling is the process of selecting code from large regions and scheduling the operations inside those regions as if they were in fact their own region. The general algorithm for a region scheduling compiler is as follows:

1. Select a region of as-yet unscheduled, contiguous blocks.
2. Place the selected operations on a data-precedence graph.
3. Schedule the operations.
4. Perform any necessary fixes, such as correcting illegal transformations performed by the scheduling.
5. Repeat step 1 until no unscheduled code remains.

This algorithm is the basis for most region scheduling algorithms and has been adapted and modified since its inception to fit the context of certain architectures. Most of these different algorithms differ in the size of the regions that are selected for scheduling. The three most popular forms of region shapes are traces, superblocks, and treegions, all of which have different shapes of blocks consisting of basic blocks. Tracing is the process of producing regions by selecting basic blocks from the list $B1, B2, ..., Bn$ where each basic block is a predecessor of the next on the list and the code region is cycle free, with the exception of the region being located within a loop [2]. Superblocks are traces with the exception that there cannot be any branches, or side entrances, into the region except for one leading to the first basic block. Superblocks also make use of a region expanding technique known as tail duplication, which allows superblocks to continue through the construction process when a side entrance is found. By enlarging the size of these regions, compilers can effectively increase the amount of ILP for parallel architectures. One more thing to note about traces and superblocks is that they are both linear regions, which means that they do not have any looped control flow in the regions (single control flow). On the other hand, treegions are known as nonlinear regions because they consist of a tree of basic blocks in which the control flow of a program lies. Fundamentally, treegions are formulated by selection basic blocks from the list $B1, B2, ..., Bn$ where each path through the tree yields a superblock and the code is cycle free, with the exception of the region being located within a loop [5].

Once the shape of regions has been decided by an optimiz-

ing compiler its next task is to then formulate those regions and actually construct the schedule. These two features must be compatible in order to achieve maximal performance. Well-selected region shapes should theoretically cover the control-flow of the code region well enough so that it keeps executing along the expected-path that was formed by the scheduler. Quite oppositely, poorly selected regions will not cover the control-flow of the code region properly, which means that instructions from less-likely execution paths will be added to the critical (or most-frequent) executed path that was formed from the scheduler. Since the performance of the scheduler seems to be based upon region shape selection it can be said that it is the most significant aspect of instruction scheduling for optimal ILP.

Parallel compilers also optimize instruction scheduling for maximum TLP. This effect can be implemented in local, loop, and global scopes. Local techniques generally work on basic blocks or traces, loop techniques operate on the loops within code, and global techniques operate on whole programs. Typical multi-threaded parallel architectures utilize local scheduling techniques, or simply local multi-threaded (LMT) scheduling techniques, which essentially improve the code's level of ILP since they operate by scheduling basic blocks or traces. One problem with LMT scheduling is that it does not take advantage of the ability of multi-core architectures to simultaneously follow different execution paths [10]. Global multi-threaded (GMT) scheduling techniques can account for the parallel instruction inadequacies that LMT scheduling techniques entail, especially when dealing with loops that iterate over a large space with very little per block.

The key difference between LMT and GMT scheduling techniques is the necessity of handling control flow of a program. Since GMT scheduling techniques have to handle scheduling of an entire program, program dependence graphs (PDGs) must be constructed in order to represent the data and control flow of a program. PDGs consist of the data dependence graphs that are formulated by LMT scheduling techniques with the addition of control dependence arcs, as shown by Figure 4.

The PDGs are essential to GMT scheduling because they are used to ensure that the data and control dependences are respected while regions of code are executed in parallel from multiple threads. They essentially act as an intermediate representation for multi-threaded code generation and also scheduling decisions. However, one problem inherent with using PDGs as a guide in instruction scheduling arises when dealing with cyclical code. Since the goal of a scheduler is to minimize the longest path through the PDG, finding the longest path is shown to be NP-hard [3]. One approach to transform a cyclic region into an acyclic one is to simply merge inner loops into one single node and disregard all loop-carried dependences.

Another problem with multi-threaded scheduling is synchronization between concurrent threads that need to communicate in order to satisfy the dependences of the PDG. One technique used to improve the communication between

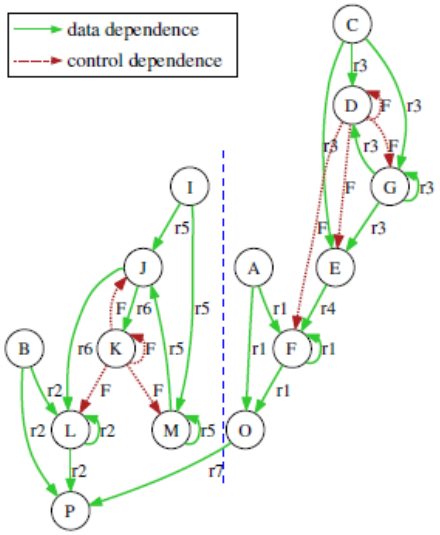

Fig. 4. A sample Program Dependence Graph (PDG). Figure retrieved from [10]

concurrent threads during instruction scheduling is to use a clustering pre-scheduling pass algorithm, which simply clusters nodes from a PDG together that would incur not benefit from being assigned to different threads. The nodes in these clusters are then executed sequentially in the order they were added to the cluster while other nodes that were not added to clusters are concurrently dealt with in parallel across multiple threads, thus improving the level of TLP in a program. Once the cluster pass is complete and the scheduling problem has been reduced to acyclic one, the scheduling decisions are made using an acyclic scheduling algorithm. One such acyclic scheduling algorithm is known as list scheduling, which is simply the process of assigning priorities to nodes in the PDG and schedules each node following a prioritized topological order. Also, a node is scheduled as early as possible so as to satisfy the input dependences for that node that conform to the available resources on the machine. Finally, once the scheduling decisions are complete the final, TLP optimized code is generated.

Optimal instruction scheduling for TLP using GMT scheduling techniques is significantly better than using LMT scheduling techniques. GREMIO, which is a generalized version of Decoupled Software Pipelining (DSWP) in the sense that it can be applied to arbitrary regions of code, was tested to compare the level of parallelism that could be extracted from GMT and LMT techniques [10]. It was shown that less than 2% of the parallelism obtained by GREMIO, which implements GMT scheduling techniques, can be achieved by LMT techniques. Therefore, it can be said that GMT is a superior procedure for scheduling on parallel architectures with multiple threads executing concurrently.

III. Evaluation

Perhaps the most frequently occurring problem that parallel compilers face is synchronization. This is especially

true for loop transformation techniques where the pre-transformation loops operate on a common set of data. For instance, if many individual loops iterate over an array of data and they are fused together, synchronization issues between the individual nested loops will exist and may potentially result in an illegal fusion. If loop-carried dependences emerge from the fusion loops then an infinitely cyclic path can emerge inside the fused loop, which is not ideal and ultimately defeats the purpose of parallelization. Tiling also suffers from a similar condition where created tiles can form illegal cyclic chains of loop-carried dependences.

To solve the problem of synchronization issues these transformation techniques try to avoid the formation of illegal loops. Many algorithms have been introduced to try to maintain parallelism, such as the shift-and-peel technique for loop fusion. The basic idea behind these algorithms is to avoid infinitely cyclic paths within the newly-formed loops by adjusting iterations spaces of the nested loops. Some adjustments may require sections or iterations of the loop to be removed in order to maintain parallelism. While this result is not ideal, it is sometimes the only feasible way to avoid cyclic paths through the newly formed loop. The peeling portion of the shift-and-peel technique is responsible for stripping nested iteration spaces that contain loop-carried dependences and placing them in another, external loop that is executed after the fused loop. Clearly, data dependences between loops are a special consideration for parallel compilers and must be dealt with appropriately in order to maintain parallelism of a program.

Another important consideration that loop transformation techniques must be wary of is cache use and the principle of locality. Loop fusion and tiling techniques increase the level of parallelism or a program or segment of code, but by doing so they also increase the use of the cache. Unlike the problem with synchronization that loop fusion faces, cache thrashing increases when many different sets of data are brought together inside one loop. When parallelism and execution speed are increased by loop transformation techniques, data from the cache is accessed more frequently for use. If iterations that result from the loop transformations rely on different sets of data, cache misses will be much more frequent and the performance enhancements from loop transformations will decrease noticeably.

Theoretically, there are two different approaches that could be taken to avoid this problem. Firstly, the hardware cache size could be changed so that more data can be stored at a time. Secondly, the parallel compiler can optimize the code for cache access. Realistically however, using parallel compilers to solve this problem is the ideal approach. They will often times seek to manage access to the cache to avoid this problem. One example of this is cache partitioning, which is an essential aspect of loop fusion. Based on the potential decrease in performance due to cache misses and thrashing, locality is a significant issue when implementing loop transformations.

From an instruction scheduling standpoint, maintaining TLP and ILP in a parallel program typically cannot be simplified into a single procedure or algorithm. ILP and TLP both incorporate a divide and conquer technique of scheduling regions of code from an entire program but compilers will take different approaches when trying to maximize TLP or ILP. More focus is placed on region shape selection when optimizing ILP because the shape of regions, once fed into the region generation mechanisms of the procedure, are the deciding factor for scheduling operations and are therefore directly related to level of ILP.

In addition, when trying to optimize the ILP for a program it is necessary to decrease the amount of vertical and horizontal waste in each CPU cycle. A large amount of waste implies that parallelism was not achieved because operations that could have been executed simultaneously were instead split up among different cycles and executed almost sequentially. On the other hand a small amount of waste implies that the operations within a parallel program were organized and optimized efficiently so they could be executed in parallel with each other during the same cycle. Therefore, it can be said that optimizing for ILP is essentially a task of decreasing functional unit waste on the CPU.

Parallel compilers that seek to optimize TLP for a program focus more on the program as a whole and the process of assigning regions of scheduled code to certain threads. With the use of PDGs to map out the data and control dependences throughout the program parallel compilers can schedule operations that ensure parallelism and optimal TLP. The use of PDGs is vital to this whole process since the program is being looked at as a whole and not region by region when the scheduling process is performed. Also, since they effectively act as an intermediate representation of the entire program they serve as a way to check the legality of that representation. Cyclic paths are easy to detect when analyzing a PDG and they are also relatively simple to fix.

There are two significant similarities between ILP and TLP optimizing techniques. Namely, ensuring the legality of scheduled regions and the dependence on the parallel architectures. As with the loop transformation techniques, scheduling techniques must also be wary of the legality of their results. When dealing with parallel architectures it is often the case where code adjustments and optimizations will create illegal cyclic paths in the execution of a program. Therefore, parallel compilers should be cautious when optimizing code so as to avoid these cyclic paths during execution.

Generating acyclic execution paths when scheduling regions of code is crucial to ensuring parallelism and legality of the program. Ensuring that every execution path that is scheduling is acyclic is difficult because the parallel architectures that compilers are optimizing for are different in multiple levels. These architecture differences contribute to several of the problems that parallel compilers face when scheduling primarily because it is difficult generalize the scheduling process without knowing about the specific architecture details. For example, trying to maximize TLP without a firm understanding of the number of threads or how the threads communicate with each other on a system

is almost impossible.

ILP and TLP optimization techniques also differ significantly by the scope of the technique.TLP techniques tend to focus on the program as whole, taking into consideration both the control flow and the data dependences that exist between different portions of a program. By analyzing and mapping out the entire program the compiler can schedule regions of code to each thread so that the entire program is optimized on a global thread level. This does not imply that the operations or instructions assigned to every thread are optimized. Rather, it implies that the program instructions are split up among separate threads in an optimal way so that when each thread runs in parallel with the others on a parallel architecture.

Quite differently, ILP techniques focus on specific regions of code within an entire program. These individual regions of code are organized and adjusted so that they operations contained within these regions are assigned to the functional units in the processor. By focusing on individual regions the compiler can also help decrease the amount of waste within the processor, which improves parallelism.

IV. Future Work

The most common problem that parallel compilers face is their dependence on the target architecture. Many of the techniques implemented by parallel compilers depend on certain characteristics or aspects of the target architecture. For instance, instruction scheduling falls victim to this architecture dependence because it must have an understanding of how many functional units are available on a processor and how threads will impact the performance of the optimization. It is theoretically impossible to optimize a program for specific architecture without knowing anything about how that architecture is structured. A general algorithm might be a good approach to try to make a parallel compiler more applicable to a wider set of architectures, but by generalizing the algorithm it will lose the specific details that allow it to make full use of parallel capabilities of a specific architecture.

Instead, it might be a good idea to attempt to abstract the architecture details or optimize a program up to only a certain point. In other words, there could be an intermediate gateway or interpreter that takes code that was compiled using a generic parallel compiler and modifies it appropriately depending on the target architecture it will be executed on. This idea is similar to the Java virtual machines, where Java code is compiled into byte code so that it is portable across multiple platforms. If a parallel compiler could produce similar byte code, with different syntax of course, which could be run on many platforms that have a type of virtual machine then in effect the parallel compiler would be made cross-platform and architecture independent.

Another avenue for future work could be the exploration of priority-based instruction scheduling. When compilers try to maximize ILP or TLP they do so by selecting regions of a program to schedule. Regions are scheduled based on when they should be executed in a sequential environment. However, since regions are executed in parallel with each other where their order does not matter, then perhaps it would be advantageous to prioritize these regions when scheduling. Each region that is selected for scheduling undoubtedly has a unique amount of computational overhead involved. For example, a region might contain a large block of mathematical operations in a loop or simply a single procedure that requires very few instructions. If the parallel regions that contain a large amount of computational overhead are scheduled prior to those that do not then cache thrashing might decrease. This is because the amount of data reads from the cache would be consistently going from a high amount to a low amount based on how computationally intensive a region of code is, which would provide more consistent use of the cache. Perhaps this consistency would thus help improve the overall level of performance.

Another technique that potentially merits future work is the combination of TLP and ILP maximizing techniques into a single algorithm. Regions of code that were distributed among multiple threads could be fed in to ILP maximizing techniques to produce optimized instruction scheduling by decreasing the amount of vertical and horizontal waste. This would essentially be taking the divide and conquer step of instruction scheduling one step further to specialize the regions of code that threads execute.

V. Conclusion

This survey paper discussed, analyzed, and compared different compiler optimization techniques in the context of parallel compilers. Loop transformation and instruction scheduling are two very important and challenging tasks that parallel compilers face. Loop fusion, which is the process of merging individual loops to form one (or several) nested loops, and loop tiling, which is the process of breaking up the iteration spaces of a loop to improve data reuse, are two of the most fundamental loop transformation techniques used by parallel compilers today. Instruction scheduling is the process of optimizing a program to maximize TLP and ILP. It involves selecting regions of code to schedule depending on the type of parallelism trying to obtain.

Although these optimization techniques have been shown to be effective on several different sets of machine architectures, they still encounter synchronization, cyclic path, and legality issues. Several techniques such as the shift-and-peel technique for loop fusion and the use of PDGs for instruction scheduling for maximum TLP have been implemented and proven to be effective against these problems. However, there is still much room for improvement when designing parallel compilers and as we progress into the era of parallel computing researchers will continue to chip away at the monumental challenges parallel compilers face.

References

[1] Fisher J.A. Young C. Faraboschi, P. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE*, 89(11):1638 –1659, Nov 2001.

[2] J.A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, C30(7):478 – 90, 1981/07/. microcode compaction;horizontal microcode;trace scheduling;global compaction;parallel microcode;.

[3] M.R. Garey and D.S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness*. Oxford, UK, 1979//. NP completeness;intractability;problem solving;operations research;computer meta theory;.

[4] Mary Hall, David Padua, and Keshav Pingali. Compiler research: the next 50 years. *Commun. ACM*, 52(2):60–67, 2009.

[5] William A. Havanki, Sanjeev Banerjia, and Thomas M. Conte. Treegion scheduling for wide issue processors. pages 266 – 276, Las Vegas, NV, USA, 1998. Control flow graph (CFG);Instruction scheduling;Treegion scheduling;.

[6] Carter L. Ferrante J. Hogstedt, K. On the parallel execution time of tiled loops. *Parallel and Distributed Systems, IEEE Transactions on*, 14(3):307 – 321, March 2003.

[7] Aso H. Lee, S. Loop-synthesizing transformation for maintaining parallelism and enhancing locality. pages 156 – 163, Oct. 2003.

[8] Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 114–124, Washington, DC, USA, 1997. IEEE Computer Society.

[9] Abdelrahman T.S. Manjikian, N. Fusion of loops for parallelism and locality. *Parallel and Distributed Systems, IEEE Transactions on*, 8(2):193 –209, Feb 1997.

[10] August D.I. Ottoni, G. Global multi-threaded instruction scheduling. pages 56 –68, Dec. 2007.