Decoupled Software Pipelining in LLVM

15-745 Final Project

Fuyao Zhao, Mark Hahnenberg
fuyaoz@cs.cmu.edu, mhahnenb@andrew.cmu.edu

1 Introduction

1.1 Problem

Decoupled software pipelining [5] presents an easy way to automatically extract thread-level parallelism for general loops in any program. The compiler does this by examining the dependences of the loops in a given program,

splitting the instructions of those loops into multiple smaller loops that execute in independent threads, and inserting dependence communication where necessary between these threads so that they remain synchronized. The full process is described below in detail.

1.2 Approach

We chose to implement DSWP using the general-purpose POSIX threading library (pthreads) and the Low Level Virtual Machine (LLVM) compiler infrastructure.

1.3 Related Work

The initial DSWP has been implemented in the context of the IMPACT research compiler using Itanium platform simulator which has customized hardware-level support. Other works of DSWP including [6, 4] also use hardware simulator.

1.4 Contribution

The decision to use LLVM will allow our implementation to be viewed in a context that is more relevant and more widely used than the IMPACT compiler, as LLVM is becoming not only a research but industry standard.

Due to our choice to use pthreads, Our DSWP implementation will also be portable across more platforms than previous implementations since any system that supports both LLVM and POSIX will be able to use our implementation, while former systems were limited to the Itanium platform with customized hardware support.

2 Design

2.1 The Steps of DSWP

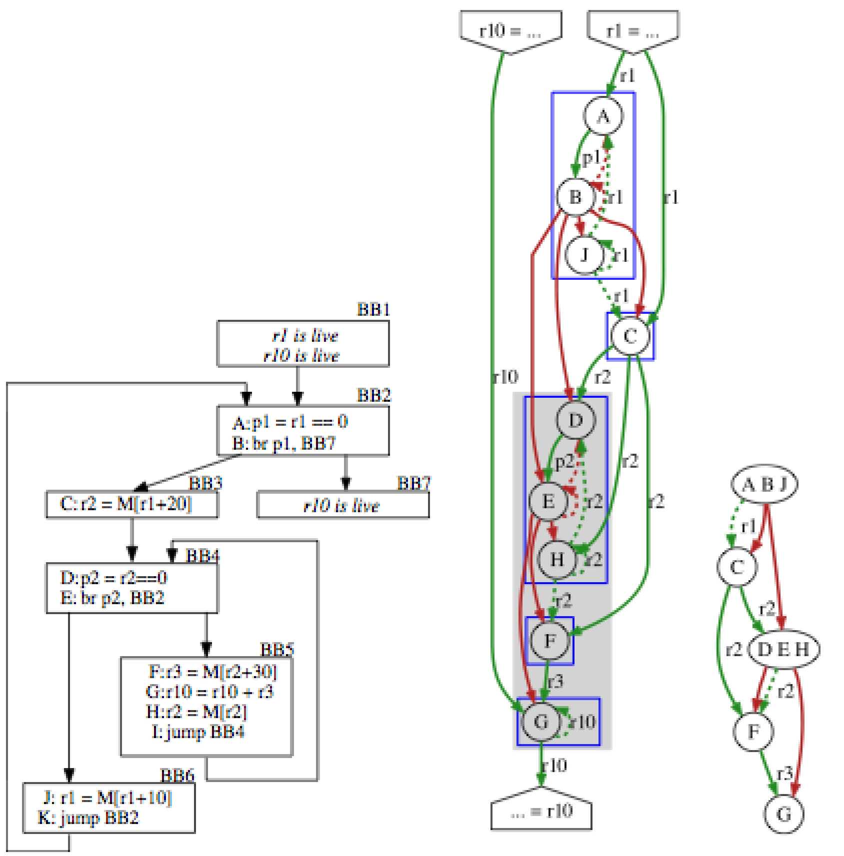
DSWP can be described as a series of steps that first accumulate information about the program, then use that information to the extract the TLP from the program's loops, and finally modifies the program to execute in parallel those portions deemed to be parallelizable.

2.1.1 Build Program Dependence Graph (PDG)

In order to correctly insert dependence flows among the independently running sub-loops that are generated we need to enumerate the various dependences among the instructions in each loop within the program. These dependences can be classified into two categories: data dependences and control dependences. We build the PDG [2] containing this information as our first step as in Figure 1.

This process is different than how it was described in the paper. The paper [3] used a method of peeling off the first iteration of the loop and running normal dependence analysis on that graph and then coalescing the duplicated nodes afterward to fit the original graph. This was difficult in LLVM because of the fact that LLVM is in SSA form, so it would have been unclear as to how to modify the program graph to make this work, a possible solution would be introduce phi node, while our current implementation only support the un-optimized code.

Therefore, we did this in another way. In addition to the normal PDG, essentially we need the header to have a control dependence on any block that branches either to the exit or to the header. We do this simply by checking each branch instruction within the loop to see if it goes back to the loop header. If it does, then we mark the header as depending on that block.



(a) Original code (b) Dep. graph (c) DAG_{SCC}

Figure 1: Illustrates the original program (a), the PDG based on that code (b), and the DAG_{SCC} generated using the PDG.

2.1.2 Find Strongly Connected Components (SCC) and Coalesce into a DAG

In the next step, we take the PDG and find the strongly connected components [1] in it. We do this to ensure that there are no cycles in the resulting graph when we coalesce the nodes, which is necessary to form a pipeline. Each of these SCCs represents cyclic dependences within the loop, so the compiler requires that these remain in the same thread. After we have found the SCCs, we coalesce each of them into single nodes to form a DAG. Refer to Figure 1 for an illustration of this process.

2.1.3 Assign Partitions of DAG to Threads

Our goal for the next step is to partition the nodes of the DAG among a fixed set of threads. A valid partitioning P is such that all of the dependence arcs either are within the same $P_i \in P$ or they go from P_i to P_j such that $i < j$, thus forming a pipeline between the partitions. We want to choose a partitioning among the threads such that we minimize our overall execution time. While this general problem is NP-complete [3], we use the heuristic from the [3] to provide an estimation for a given partition of the DAG. It is basically a greedy algorithm which always tries to add the DAG node that has the largest latency.

However, in this model the communication cost is not considered, which is also the main difference between our implementation and others. Many of the partition results we have examined could be improved if the synchronization costs were also considered. One possible solution would be to give more priority to the nodes not in current partition but with more edges to the nodes in current partition, thereby reducing the amount of necessary inter-thread communication. This idea could be developed further in future works.

2.1.4 Split Partitions into Loops

After we find a valid partitioning of the nodes, we create a new function for each partition and copy in the relevant instructions for those partitions. In the original thread, we send the addresses of these functions to the waiting threads.

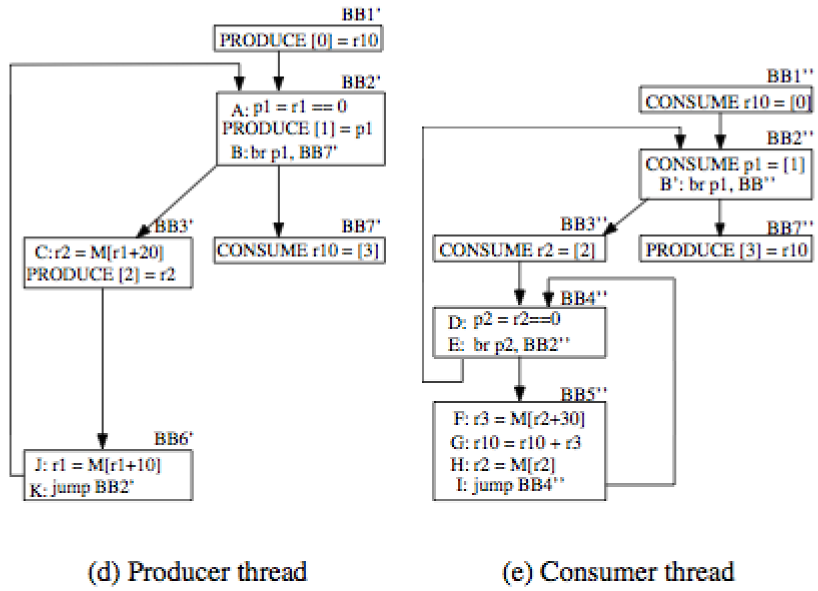


Figure 2: Illustrates the splitting of code across threads and the insertion of synchronization primitives (produce (d) and consume (e)).

2.1.5 Insert Dependence Synchronization

After generating the new functions to be assigned to the separate threads, we need to insert the dependence communication between the partitions.

Since different synchronization queues can be used on consecutive iterations, we must synchronized the threads iteration by iteration. See Figure 2 for an example of code being split across two threads and the resulting produce and consume sites that must be inserted.

2.1.6 Runtime Support

The compiler needs the support of a runtime library that will manipulate the necessary synchronization primitives. A separate synchronization library (simple_sync.c) is linked with the regular source code at compile time. This library, in turn, uses queue.c as its queue implementation. We statically create 256 dependence queues each with a fixed length of 32. If a queue is empty, pop operations will block until a value is pushed into the queue. If a queue is full, push operations will block until space becomes available. The locations where produce and consume instructions are inserted into the program during compilation are simply calls to push and pop respectively.

At the beginning of each loop the new threads are created and each one is sent the address of the function it is to execute. Each dependence that needs to be communicated between threads gets its own queue. Each produce in the thread that generates the dependence is matched by a single consume in the thread that requires that dependence.

3 Experimentation

We built a loadable module for LLVM that can be called as a separate optimization pass. We then ran this pass on some self-written benchmarks. We initially intended to run our optimization on the SPEC2006 benchmarks, however, Our current implementation is limited in dependence analysis so that could not used for large program.

For simple programs, our implementation has significant overhead. For example, an unoptimized loop which calculates the sum from 1 to 10000 takes 0 seconds while our parallelized version takes 4 seconds. This is mainly because when the program is split, one thread increments the loop counter and the other thread calculates the sum, thus incurring a huge synchronization cost.

4 Lessons Learned

Academic papers are not always explicit about the steps they take in their implementations, in order to capture all the necessary detail, you must be more knowledgeable, e.g to read more papers related to the topic, after that you probably could infer what is really going on there.

Additionally, our software engineering skills could be improved, specifically our estimation of project size and our time allocation. This project was a more significant undertaking than we initially thought. (e.g. the code splitting seems to be the simplest part but it take us most time to finish). Currently the project sits at about 2 KLOC and still more could be done.

5 Future Work

We still need to test our implementation more on complex programs, for example, the SPEC2006 benchmarks across multiple platforms, especially on machines with higher numbers of cores. These tests would help us reassure us that our implementation was correct and they would also give us a better idea of how a software-only implementation of DSWP compares to unoptimized code as well as hardware-supported DSWP.

Our dependence analysis is limited in several ways. We need to have a reasonable solution for the dependences that DSWP needs. We also need to consider the dependences after we have phi nodes in the code. Additionally, better alias analysis would help us improve the accuracy of our dependency analysis, reducing the number of dependences in our PDG which in turn would lead to less synchronization overhead.

Since we have a different implementation of thread synchronization, our model for the costs of synchronization could be significantly different from those defined in previous papers. As we mentioned previously, we could investigate these costs further to get more accurate model and better heuristics, thereby improving the quality of our thread partitioning.

The way that we rewrite the program causes a lot of overhead in that we start a group of threads at the beginning of each loop body and then join them at the end of the loop body. We could reduce this initialization overhead by creating all the threads at the beginning of the program or perhaps lazily when we encounter our first loop body.

The number of threads used for this type of pipelining is fixed at compile time. This and past implementations will use all of those threads and no more, even if using fewer or more threads would be more advantageous. One possibility for further development would be to find a way to vary the number of threads used based on the structure of the loop body or perhaps based on static of dynamic profiling.

6 Conclusions

Despite the lack of official benchmark results, we can hypothesize that our software-only implementation of DSWP will not be able to compete with a similar implementation with custom hardware support. However, we have presented a method to extract thread-level parallelism from general loop bodies in a more platform-independent fashion than the previous state-of-the-art.

Therefore, we think that if DSWP is going to be widely used in an industrial setting, the problem of communication overhead must be solved. As we discussed above, we need either improve the thread partitioning significantly or add hardware support, which has shown promise in several academic papers.

References

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[2] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, July 1987.

[3] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.

[5] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

Distribution of Total Credit

Fuyao implement framework of DSWP and component for dependence analysis, SCC/DAG, thread partitioning, loop splitting and synchronization. Mark implemented the runtime support for multithreading based on pthreads, contributed to the synchronization implementation, and managed write-ups and poster.