1 Introduction

Software pipelining is an optimization technique used to speed up the execution of loops. A compiler performing the optimization reorders instructions within a loop in order to minimize latencies and avoid wasting instruction cycles. The optimization parallels the out-of-order execution paradigm used by microprocessors, except that instruction reordering is done at the software level, i.e. by the compiler.

Traditional software pipelining exploits a single processor’s parallelism capabilities to execute several instructions at once. A related technique, known as Decoupled Software Pipelining (DSWP) [3], instead exploits multicore or multi-processor machines’ parallelism capabilities to execute loop instructions in several threads, which can communicate with one another. Rather than leaving the onus on the programmer to write parallel programs, DSWP automatically transforms sequential programs into parallel ones. DSWP aims to increase efficiency in execution and offer latency tolerance by producing non-speculative, decoupled threads.

In this project, we provide an implementation of DSWP for the LLVM compiler framework. In particular, we provide the DSWP code as an optimization pass that can be run on loops in the LLVM intermediate representation.

2 Background on DSWP/DSWP+

DSWP seeks to decrease inter-core communication and resource requirements, thereby reducing the overhead of parallelization. The algorithm works by building a dependence graph for loops, assigning the strongly connected components of that graph to threads, and passing data between threads to pipeline execution of the loop. We describe these steps in further detail below.

The first step of the algorithm is to build a program dependence graph, where an edge from one instruction to another indicates a dependence of the latter on the former. Dependences can be of two types: data dependence or control dependence. Data dependences refer to loads, stores, or allocations in common registers or memory locations. Control dependences refer to conditional branch or switch instructions, in which the outcome of the condition can determine whether another particular instruction will be executed.

Once the dependence graph is built, the algorithm partitions the nodes into strongly connected components (SCCs). Thus, the result can be viewed as a graph in which nodes are groups of instructions, and edges indicate a dependence between instructions belonging to the two groups of
the corresponding nodes. The resulting graph has the advantage of being a directed acyclic graph (DAG), which will be critical for proper communication between threads in subsequent steps.

Next, the algorithm assigns the various SCCs to different threads. An assignment is considered valid if a dependence of SCC $i$ on SCC $j$ implies that $j$ is assigned to a lower-numbered thread than that of $i$, or possibly the same thread as $i$. Since dependences will entail communication between respective threads, the validity criterion ensures that communication can only pass along one direction, i.e. from lower-numbered threads to higher-numbered threads.

Finally, synchronization instructions are added between threads, and the appropriate produce and consume instructions are added between threads to communicate data and control synchronization.

Depending on the function of the loop, there may be only a few strongly connected components in the dependence graph, hence limiting the performance benefits that can be realized by DSWP. In fact, the speedup obtainable by DSWP is limited by the portion of the work performed by the slowest SCC; if one SCC does half the work of the loop, the DSWP-transformed loop will be no more than twice as fast as the original loop, no matter how many threads are used.

It is often the case, however, that the individual components derived by DSWP are more amenable to other automatic parallelization techniques. For example, one of the components may take the form of a DOALL loop. In this case, we can simply split that component amongst multiple threads, each handling different iterations of the component. This is the idea behind PS-DSWP [4], which uses DSWP to transform the code into components, some of which may then be optimized by DOALL transformations.

In fact, the DSWP optimization can enable a wider variety of parallelization techniques than just DOALL transformations. In DSWP+ [2], the authors use DSWP to enable DOALL, LOCAL-WRITE [1], and SpecDOALL [5] transformations. This allows for more parallelization on a wider variety of programs.

3 Previous Work

The original authors implemented DSWP in the IMPACT compiler system [3]; later implementations used the VELOCITY compiler [4]. Despite several requests on the LLVM mailing lists, however, there is no available implementation of DSWP in the popular LLVM optimization architecture. And, in fact, there appears to be no implementation of a complete DSWP+ system at all. For empirical results in the paper on DSWP+ [2], the authors instead manually transformed code according to the algorithm of the optimization.

In a previous project for this course, [6] began an implementation of DSWP as an LLVM pass. We originally proposed to extend their implementation to incorporate the other components of DSWP+. Unfortunately, however, the DSWP implementation of [6] was substantially incomplete and contained many bugs. We will describe some of the drawbacks of their implementation that we have modified in the next section.

4 Implementation Details

We will now describe our implementation of DSWP, including some of the design decisions and challenges we faced for each stage, along with differences from the implementation of [6].
Dependence Graph Extraction

The first step of DSWP is to determine the dependence relationships amongst the instructions of a loop.

Data dependences through registers are straightforward in LLVM: we simply follow the def-use chains. We only need to worry about true dependences here.

Data dependences via memory are, as in any general compiler, substantially more complicated to identify, and rely on alias analyses. We use LLVM’s built-in MemoryDependenceAnalysis pass for this purpose.

DSWP requires a slightly more complex definition of control dependences than is usual, because it must also include “loop-carried” control dependences. There dependences are present when an instruction determines whether another instruction is executed on this iteration, whereas classical control dependences are present only if the former instruction can decide whether the latter is ever executed. In order to calculate these, we use the method described in [3] of finding control dependencies in the graph where one loop iteration has been peeled off. The idea is to create two copies of the loop, where the first copy indicates the peeled off iteration, and add control flow edges between the two copies whenever an instruction in the loop is dependent upon by an instruction in the following iteration of the loop. Standard control dependency analysis is then run on the modified graph. In particular, in this analysis, node \( w \) is control dependent on node \( v \) if (1) \( w \) is not a post-dominator of \( v \) and (2) there is some path from \( v \) to \( w \) in the control flow graph such that all nodes between \( v \) and \( w \) are post-dominated by \( w \). After obtaining the control dependency analysis on the “peeled” graph, corresponding nodes in the two loop copies are coalesced to obtain the final control dependency graph of the loop.

In performing the standard control dependency analysis on the “peeled” graph, we use LLVM’s built-in PostDominatorTree class to construct a post-dominator tree, after which we run a simple tree walk procedure.

[6] used a simpler heuristic for control dependencies that was insufficient in practice.

Strongly Connected Components

Given the program dependence graph, finding SCCs is fairly simple; we implement it using both a forward and a backward depth-first search.

Thread Partitioning

Once the SCCs have been extracted, we estimate the latency of each one by adding up estimates for each instruction in the component. Here, we use a constant estimate for the latency of function calls, which can of course vary widely and so are a source of potential errors in the thread assignment process. Future implementations could use LLVM’s facilities to estimate the amount of time that a given function will take.

When each SCC’s latency has been estimated, we use a heuristic for the partitioning problem (which is NP-complete): we simply schedule components whose predecessors have already been scheduled in decreasing order of cost. Once a thread passes the mean latency needed per thread, we consider it complete and move on to the next thread.

[6] used this approach, but incorrectly scheduled components when any of their parents had been scheduled, rather than requiring all of their parents to be complete. This frequently yielded incorrect schedules that lead to profoundly bad performance in practice.
The original DSWP paper [3] used this algorithm, but with one difference: in the event of a tie, our code chooses an arbitrary instruction, whereas the paper says to break ties according to which choice will result in the fewest outgoing dependencies. This seems like it would help in selecting better thread partitions and should certainly be implemented in the future.

**Code Splitting**

This stage involves copying the relevant components of the loop to the worker threads and rewriting them to refer to in-loop values. Those worker threads also need to load their “inputs” (the variables that are live at the beginning of the loop) and the “outputs” (the variables defined within the loop that are live after it is complete).

In our implementation, as originally established by [6], the workers run in separate threads via the pthreads system. Arguments are passed in by giving the thread a pointer to a structure containing each input variable; the struct also contains memory locations for the output variables to be placed after the thread has finished its work.

Though conceptually simple, this stage required a significant amount of work in practice; the implementation of [6], in addition to containing many bugs, did not allow for outputs at all, and did not have any handling of LLVM’s phi nodes. The system also originally passed parameters by casting them to 64-bit integers and placing them in an array; this approach is both less efficient than the struct approach we have switched to and conceptually less clean.

**Synchronization**

This final stage inserts the “flows” where worker threads communicate with one another.

In [3], the authors used DSWP in a simulator where the CPU was equipped with dedicated hardware for intercore communication. Zhao and Hahnenberg [6], by contrast, wrote a simple library in C using the pthreads APIs to support queues, which block only when full (if pushing onto the queue) or empty (if popping from it), and which manage their accesses via locks.

For register data dependencies, these queues pass the value of registers across threads. For all other types of access, they pass no data, but enforce correct ordering of accesses by pushing and popping empty values.

Unfortunately, these queues also cast their data to 64-bit integers. Since each queue only ever stores data of a single type, we would like to change the system to create the runtime queues dynamically based on argument types, as we do the structs for worker thread arguments. This would decrease overhead by removing the need for casts, make the system conceptually simpler, save memory, increase speed, and eliminate the need to include a separate runtime library when running the optimized code.

The implementation of [6] also had some substantial bugs here. In particular, it inserted produce instructions in the location where values were defined but consume instructions where the value would be used, so if a value is used on only some iterations, the producers and consumers become out of sync and the program quickly deadlocks when the queue fills. They handled control dependencies for branch instructions in a way that frequently resulted in deadlocks, as well.

We have fixed these bugs, and as far as we know our synchronization is correct. It is, however, somewhat too conservative: with control dependencies in particular we use many redundant queues. Ottoni et al. [3] suggest that such redundant flows can be eliminated, but do not give an algorithm
for doing so; it will be important to do this to reduce the overhead of DSWP is this code is to be used.

5 Experiments

We show experiments on a program implementing a solution to Project Euler\(^1\) problem 44, which deals with pentagonal numbers: those that can be written as \(n(3n - 1)/2\) for some natural number \(n\). The problem is to find the minimum distance between two pentagonal numbers whose sum and difference is itself pentagonal. We run our code on a solution in C with a liberal license, found online\(^2\). The solution simply examines all pairs of the first 10,000 pentagonal numbers, checks if their sum and difference are each pentagonal, and maintains the lowest distance between two such numbers.

In a straightforward timing experiment on a machine with a 2.8 GHz Intel Core 2 Duo processor with the runtime queue library configured to have a maximum size of 32 elements, we found that the original code run under clang’s \(-O2\) optimization level ran in 1.41 seconds of elapsed time (1.38s user, 0.02s system, 100 \%CPU). When we ran DSWP on that code (after running \(-O2\)), the resulting code ran in 2 minutes and 40 seconds (77s user, 71.16s system, 92 \%CPU). Even though this code was running two threads, each was on average less than half loaded; the remainder of the time is spent waiting for locks.

Because we have not implemented the removal of redundant synchronization flows, the output of DSWP had many channels that were essentially the same thing repeated one immediately after another. We also tested a version where we manually reduced the channels to contain no straightforward redundancies. In that case, interestingly, the resulting code ran in 4 minutes (20s user, 103s system, 50 \%CPU). It thus spent significantly more time waiting for locks than did the unmodified DSWP code, but also executed many fewer userspace instructions (presumably overhead related to the communication). Why this happens is a mystery to us.

Figure 1 shows the number of elements in one of the data dependence channels during the first section of execution, corresponding to roughly 0.2 seconds. The times that are “poking out” of the top are when the producer is blocking because the queue is full; the equivalent points on the bottom are when the consumer is blocking because the queue is empty. We can see that one thread or the other is usually blocking on this particular queue; hence the poor performance.

6 Lessons Learned

In the equivalent section of their project report, Zhao and Hahnenberg [6] noted that academic papers do not always give very explicit descriptions of steps taken in their implementations. This is certainly true of some aspects of [3]; for example, the algorithm for collapsing redundant flows seems important but is barely mentioned in the paper.

We have also learned, however, that grad students’ course projects do not necessarily implement as much as they seem to claim. After reading the report of [6], it seemed to us that it would be a mostly-complete DSWP implementation that only needed a little bit of further work to be entirely usable, and that we could then go on to extending it to DSWP+ and beyond. This turned out not to

\(^1\)http://projecteuler.net
\(^2\)https://github.com/eagletmt/project-euler-c/blob/master/40-49/problem44.c; we marked the two helper functions as inline.
be the case. After much work, we finally have a mostly-working implementation of most of DSWP, with a few missing parts and poor performance. It might take the equivalent amount of work of another course project to get this implementation up to excellent performance, and another to get a good implementation of DSWP+ going. One should also note that, in the literature, the bulk of speed improvements occurred after performing the DSWP+ extensions (e.g., DOALL, LOCALWRITE), whereas with pure DSWP, the performance was only slightly better (though, at the very least, on par with the unoptimized code). Thus, it is reasonable to expect that appreciable gains will occur only with a full implementation of DSWP+, though an improvement of our DSWP implementation that yields performance similar to that of the unoptimized single-threaded code would be a tangible first step in that direction.

7 Future Work

The most obvious problem with this work is its poor performance. We believe that this is due to a combination of the high overhead of our queuing system, redundant synchronizations, and a weak scheduling heuristic; each of those areas is feasible to improve. For example, for the scheduling heuristic we can both implement the full heuristic of the original paper (incorporating the number of introduced outgoing dependencies), as well as investigating other extensions that take into account a more nuanced view of the cost of a particular schedule. It would also be very helpful to understand why the less-redundant DSWP code runs so much slower (in terms of wall time) than the more-redundant DSWP code.

There are also many miscellaneous areas of the code that could use improvement; it certainly needs some packaging work before it would really be distributable as an LLVM pass for general use.

Of course, DSWP in itself does not tend to see very impressive improvements; where it really shines is when the extensions (DOALL, LOCALWRITE, and the like) are applied to the components. A quality implementation of DSWP+ would both potentially help users improve their code and help in research related to DSWP+-style techniques.
Once some of the issues above are addressed, it would also be important to evaluate the code on a more thorough benchmark.

References


