Software Pipelining for the Pegasus IR

Cody Hartwig
chartwig@cs.cmu.edu

Elie Krevat
ekrevat@cs.cmu.edu

Abstract

Modern processors, especially VLIW processors, often have the ability to execute multiple instructions simultaneously. Taking advantage of this capability is crucial for high performance software applications. Software pipelining is a technique designed to increase the level of parallelism in loops. We propose a new approach to software pipelining based on direct manipulations of control flow graphs in *Pegasus*: an intermediate representation used by the CASH compiler. In this paper, we describe the design and implementation of our software pipelining algorithm. Additionally, we provide a detailed analysis of the metrics and heuristics used by our algorithm in the context of a simple code example.

1 Introduction

Modern VLIW architectures can schedule multiple instructions at once, but they are constrained by data and control dependencies that limit the opportunity for parallel execution. True data dependencies occur when an instruction depends on the result of a previous instruction. Other data dependencies occur when two operations write to the same variable, or an input variable to an instruction is written to by a later instruction. Control dependencies occur when predicated instructions are conditionally executed. Software compilers use parallelization techniques to work around these dependencies and exploit as much instruction level parallelism as possible from a given program.

Software pipelining is a highly effective technique to increase the level of available parallelism in the body of a loop by restructuring the code to overlap operations from different iterations. By overlapping iterations, there are more instructions available for scheduling and better opportunities to schedule instructions in parallel. Since the code in a loop may be executed many times over, even a small improvement in instruction level parallelism can lead to a significant performance improvement.

Software pipelining in general has been the source of much research, and we cover a brief classification and survey of the most popular techniques in Section 2. Our approach differs from previous research because we apply our algorithms in the context of *Pegasus*: an intermediate representation used by the CASH compiler [4, 5]. The CASH compiler translates programs written in C into implementations of hardware components. Pegasus was designed to support spatial computation, so operations in a program correspond to actual hardware operations, and a Pegasus graph models both the data flow and control flow of a program. In Pegasus, basic blocks in the control flow graph are combined into hyperblocks that represent units of speculative work. So while previous approaches to software pipelining use a loop body of instructions, our approach makes use of hyperblocks, operators, and a representation that reveals dependencies in a control flow graph. By implementing this approach in Pegasus and not the generated assembly code, we abstract away lower-level resource constraints that are handled in the later stages of compilation.

To implement software pipelining in Pegasus, we propose a localized and iterative approach that pipelines operations one at a time. Our approach computes operation outputs for future loop iterations in the current iteration. Pipelining an operation consists of moving that operation from the hyperblock of a loop body into the hyperblock's pre-header, and the data-flow for values before and after executing that operation are fed into the loop hyperblock. Then each loop iteration uses the value of the operation already computed, either in the pre-header or during a previous iteration, and computes the operation value for a future iteration. This approach is analogous to preparing temporary variables of future iterations to make the loop body schedule more efficient.

An operation is a candidate to be pipelined if it matches a number of possible patterns, described fully in Section 4. Pattern matching in Pegasus is a simple local decision, since patterns depend only on the type of operation and the source of its inputs. In our current implementation, because we do not create an epilogue, operations must also be side-effect free (e.g., loads may be pipelined but not stores). Our approach also chooses operations to pipeline that are on the most expensive paths from the beginning to the end of a hyperblock. This heuristic for choosing the next operation to pipeline tends to decouple the more expensive operations from longer path dependencies, so after software pipelining more operations are scheduled in parallel.

While the potential benefit of software pipelining is substantial, possible negative side effects are increased register pressure and wasted speculative operations. The increase in register pressure can come from computing instructions from multiple iterations at once, and can result in register spilling. Wasted speculative operations can occur when extra instructions are computed to prepare a very tightly pipelined loop that has a control flow which executes the loop only a very few times or not at all. If not handled correctly, these side effects can eliminate the benefit of software pipelining, and even do more harm than good. Since scheduling with resource constraints is a well known NP-hard problem [7], heuristics are generally used to avoid the worst of these situations, and a feedback approach between the different stages of the compiler can provide better hints as to the most effective strategies. For example, a less aggressive software pipelining strategy should be implemented in response to register spilling. We do not explicitly implement such a feedback loop, but this is an area for future work that is fully compatible with our approach.

2 Related Work

Many algorithms exist to perform software pipelining, and using a classification developed by Allan et al. [3] these algorithms generally perform either kernel recognition or modulo scheduling. Percolation scheduling [10] is an additional approach with a more localized decision process that doesn't fit exactly into either of the previous classifications, although its concepts of primitive transformations are combined with loop unrolling in Aiken's Perfect Pipelining kernel recognition algorithm [2].

Kernel recognition techniques assume the schedule for loop iterations is fixed and unroll the loop some n number of times, choosing a value of n that reveals enough instructions to improve the instruction level parallelism without creating too much code size expansion. A pattern recognition stage then identifies a repeating kernel from the unrolled loop that can be scheduled efficiently. A well known example of this technique is Aiken and Nicolau's Perfect Pipelining [1, 2].

Alternatively, modulo scheduling techniques focus on creating a schedule from one iteration of a loop that can be repeated without violating any resource and precedence constraints. A minimum initiation interval is calculated for the minimum number of instructions required to separate repeated iterations of the schedule. If the scheduler fails to find a schedule with the minimum initiation interval, it will increment this interval and iterate the same process. Examples of this technique include Lam's hierarchical reduction method that handle conditional statements on VLIW machines [9] and Rau's Iterative Modulo Scheduling [13, 15]. Rau also discusses how register pressure and allocation strategies are affected by his approach, but specifically avoids the problem of what to do when there are not enough available registers [14].

Percolation scheduling applies many atomic program transformations to a parallel execution control-flow graph based on a number of guidance rules and heuristics (including information from data-dependency analysis). The nodes in a parallel execution graph contain many operations, and operations are moved between nodes if there are no dependency constraints [6, 10, 11, 12].

At a basic level, Percolation Scheduling may appear similar to our approach, however, the transformations of Percolation Scheduling are actually very different because they change the order of independent operations. Since the Pegasus graph encapsulates both data-flow and control-flow information, the ordering of a series of operators in a Pegasus hyperblock must always be respected, since operations that appear later in the ordering depend on the results of earlier operations. The parallel execution graph used in Percolation Scheduling does not have these desirable dependence properties built into the graphical structure. Also, Percolation Scheduling produces code explosions by visiting nodes on every global control path between moves, while our pattern matching algorithm makes use of localized decisions.

3 Approach

We propose implementing software pipelining through direct manipulation of Pegasus graphs. The primary goal of this approach is to reduce data dependencies between operations in an effort to increase the opportunity for instruction level parallelism.

At a high level, we implement software pipelining by moving operations between iterations of a loop. For example, if a value is loaded from memory in a loop body, we can move that load to the previous loop iteration. In this way, the loaded value is available immediately at the beginning of new iterations and uses of it are not required to wait for a load delay. Meanwhile, the current iteration will execute the load that will be used by the next iteration. This effectively decouples the dependency between the load and its uses. This method also requires adding an instance of the operation to the loop preheader. The output of this operation is used in the first loop iteration. It's important to note that since the loop preheader will always execute under our implementation, it's important that we only move operations that are side-effect free.

Moving operations between loop iterations requires that we address several challenges. First, we need an algorithm that can correctly move operations between iterations of a loop. Second, we must choose a set of operations that we can pipeline without introducing incorrectness. Third, we must choose a priority for each instruction that can be pipelined. Lastly, we need a heuristic to decide when we have achieved the optimal pipelined graph. In other words, once an operation is successfully moved, we need a way to determine if we should move another.

To pipeline a specific Pegasus circuit, we apply an iterative algorithm that moves one operation per iteration. The first step of this algorithm is to mark all the operations that are possible to move. As stated earlier, in order for an operation to be a candidate, it must be side-effect free. Additionally, in order to simplify the algorithm, an operation must have only constants and *mu*s as its parents in the graph. Once the possible operations are marked, we use a heuristic to compute the cost of the most expensive path through each operation to the end of the loop body. Then we choose the operation with the most expensive path to pipeline. This method allows us to reduce the cost of the most expensive path through a loop body. In other words, it reduces the data dependencies for the loop body.

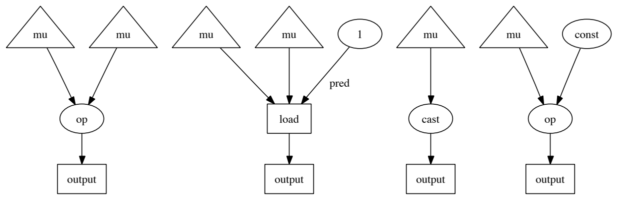


Figure 1: Patterns recognized by software pipelining include side-effect free operations with inputs that are *mu*s or constants.

4 Design

In this section we describe the specific algorithm we have designed to implement software pipelining in Pegasus as well as show a simple example execution of this algorithm.

4.1 The Algorithm

We have designed an iterative algorithm. For each iteration of the algorithm, we make a list of operations available for pipelining, choose a specific operation from these, and move the selected operation. We then evaluate whether another iteration should be executed.

Available Operations The first step of our algorithm is to find operations that are candidates to be pipelined. In order to be a candidate, an operation must meet several requirements. As described later, the selected operation will be moved one iteration back. This implies that the operation in question will execute at least one extra time. Therefore, the operation chosen must be side-effect free. Additionally, our algorithm requires that only operations at the beginning of a loop can be moved. Since Pegasus is a graph representation, these requirements are recognized through a pattern matching scheme. In order to be eligible, an operation must be either an arithmetic operations, a load, or a cast. The operations must have only *mu*s or constants as inputs. Additionally, loads must have a constant predicate input. Figure 1 shows these patterns in a Pegasus representation.

Choosing an operation Once we have a list of possible operations to pipeline, we must choose the optimal operation to move next. Since our goal is to decrease data dependencies, we choose an operation based on this information. For each operation that can be moved, we calculate the most expensive path between the operation and an *eta*. This method reduces the length of the most expensive path and thus, is likely to reduce the dependencies of the loop body.

Moving an operation Moving an operation is a general process that follows a series of steps. These steps are shown in Figure 2.

The first step is to add the operation to the loop preheader. This added operation is responsible for feeding the first iteration of the loop. We create a new *eta* in the preheader for the output of this value. This *eta* is matched to a new *mu/eta* pair in the loop body. These represent the temporary variable introduced.

Next, we alter the uses of the selected operation to take their value from the temporary variable introduced in Step 1.

Next, the output of the selected operation is modified to feed the temporary variable created. This represents performing the operation that will be used in a future iteration.

Finally, we change the inputs of the selected operation to the value they would have in the next iteration. This is done by simply connecting them directly before the *eta*s that feed the original inputs.

This method describes how operations with 2 inputs and one output are moved. It should be noted that loads are operations with 3 inputs and 2 outputs. These operations are handled identically, except they introduce 2 temporary variables: one for the output value and one for the output token.

How many operations to move Once an operation is moved, we need to decide whether to move another operation or to terminate the software pipelining process. There are many heuristics that can be applied here. The important concern is to balance the benefit of software pipelining with its cost. Moving operations decreases the dependencies between operations in any given iteration of a loop body. This allows greater instruction level parallelism in the generated schedules. However, this is not without cost. Moving operations requires the introduction of temporary variables to carry values between loops. These temporaries will increase register pressure and, at an extreme, could introduce register spill. This can eventually result in schedules that are actually worse. Therefore, we try to choose an heuristic that maximizes the potential for parallelism without introducing register spill.

Our currently implemented algorithm uses two metrics for this calculation. First, we calculate the length of the most expensive path through the current loop body. The most expensive path is defined as the longest dependency chain through the loop, where each link of the chain is weighted by the cost of the operations it's attached to. When this most expensive chain falls below a certain limit, we stop the pipelining algorithm. Second, we keep track of how many operations we have moved. The more operations that are moved, the more temporary variables will be required to keep track of these values. Therefore, we limit the maximum number of operations we move in order to prevent increased register spill.

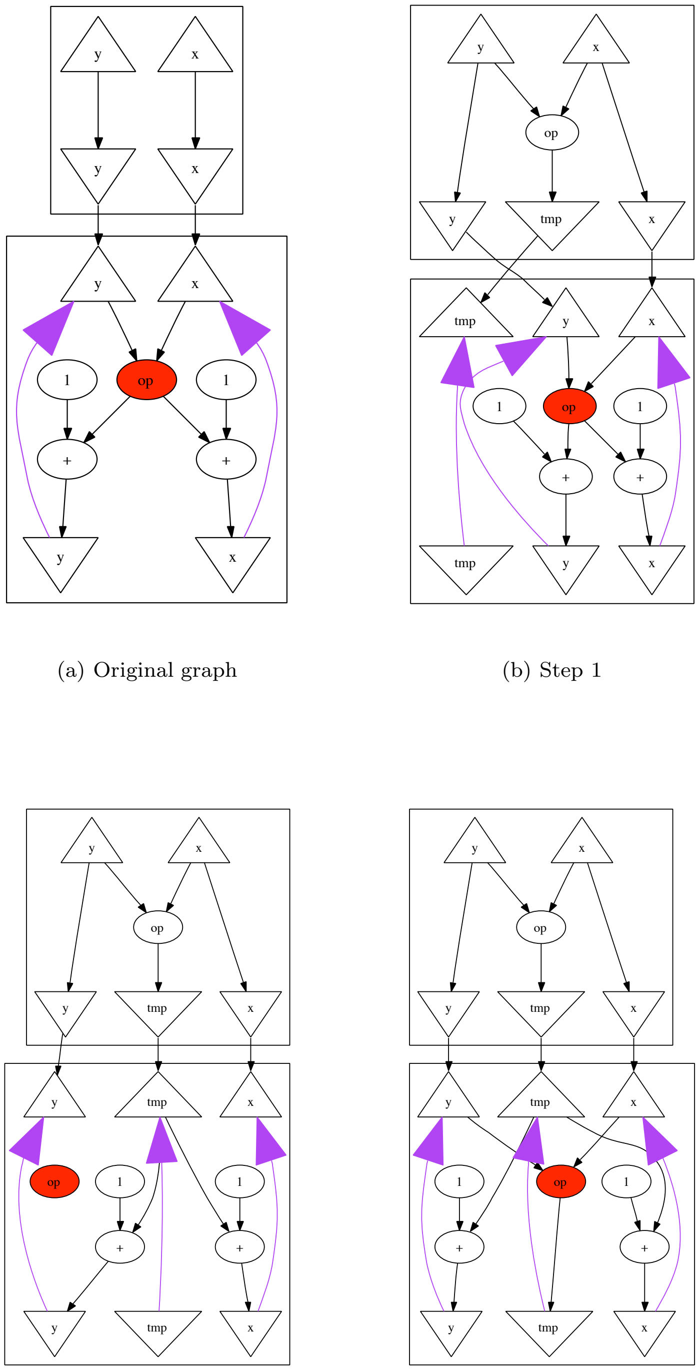
4.2 Example

This example demonstrates a concrete execution of our algorithm. Consider the following code:

```
int i = 0;
char a[100];
while (i < 100) {
  a[i] = 2 * a[i];
  i++;
}
```

Intuitively, we can see in this example that the fundamental operations are a load, store, multiply, and 2 additions. In addition, we can see a dependency chain between an add, a load, a multiply, and a store. This means that all these operations will be forced to execute in series. This is more easily seen in the Pegasus graph shown in Figure 3(a). When our algorithm analyzes this graph, it will find two operations that are available to move (these operations are shaded in the graph). In order to choose which operation to move, we examine the cost of paths they are on. We will calculate that the cost of the operation summing 'a' and 'i' is 14, while the cost of the other operation is 1. Therefore, we will select the first addition to move. We then move this operation using the algorithm described above. The resulting graph is shown in Figure 3(b).

The algorithm will now decide whether to continue and move another operation or to terminate. After observing that the most costly path is still quite long, and that only one operation has been moved thus far, the algorithm will choose to move another operation. This time the choice is between the two shaded operations in Figure 3(b). Clearly the load is on the longest path. At this point, the load is moved. Figure 3(c) shows the resulting graph. At this point we



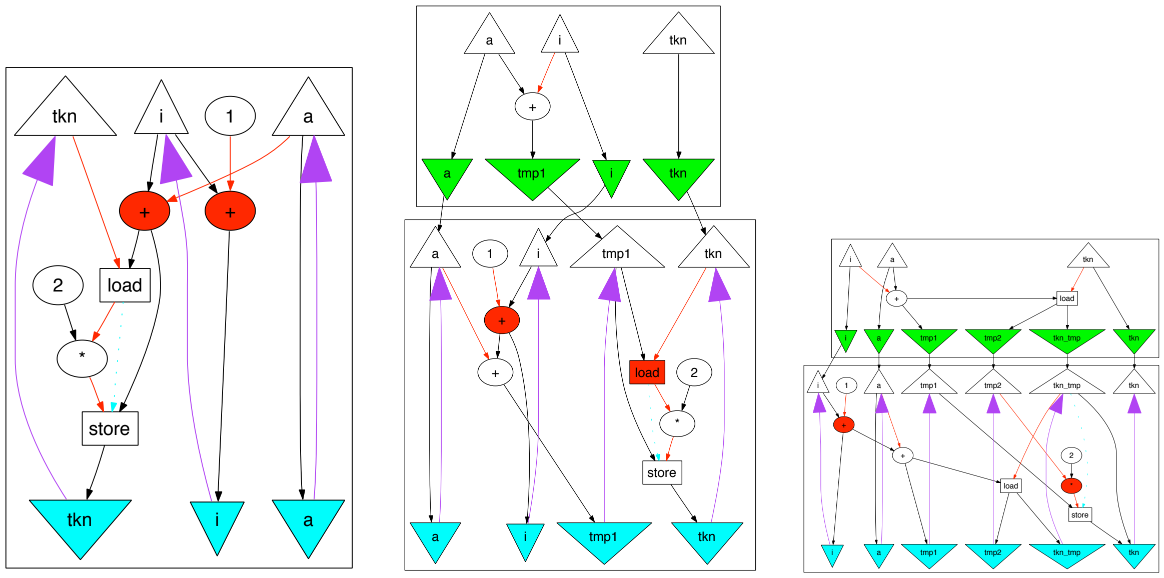
(a) Original graph

(b) Step 1

(c) Step 2

(d) Step 3

Figure 2: Steps to move operation ‘op’



(a) Original graph

(b) Iteration 1

(c) Iteration 2

Figure 3: A simple example from Section 4.2 to demonstrate the process of moving an arithmetic operation and a load.

can see that the load and store are no longer dependent on each other and can therefore be scheduled in parallel. In reality, our algorithm would continue to pipeline in this case, but this example ends here.

5 Implementation

We have implemented our software pipelining algorithm in the `c2dil` framework. Currently, the software pipelining stage runs before other optimizations and only runs once. In other words, although other optimizations are iterated in alternation, software pipelining will not be repeated once it completes the first time. This is not a restriction of our method, but it has made it easier to measure and analyze our results.

We have implemented the software pipelining functionality entirely in the files `ssaopt.cc`, `procedure.h`, `procedure.cc`, and `c2dil.cc`. Our algorithm is implemented in approximately 1000 lines of C++ code.

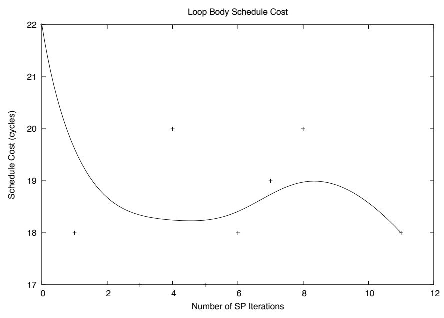
To facilitate debugging and inspection of the optimizations, we create two `dot` files. `before_sp` shows the graph immediately before software pipelining is performed. `after_sp` shows the graph immediately after software pipelining is performed. Additionally, output is printed to standard output reflecting which operations are available for pipelining at each step as well as the estimated path cost for each operation and the operation chosen to move for that iteration.

We found that although it is rather difficult to determine correctness by comparing the first graph to the last graph, it's quite simple to verify the correctness of a graph by comparing it to a graph that only differs by one moved operation. In this way, the correctness of a series of graphs can be verified in a sort of pseudo-inductive method.

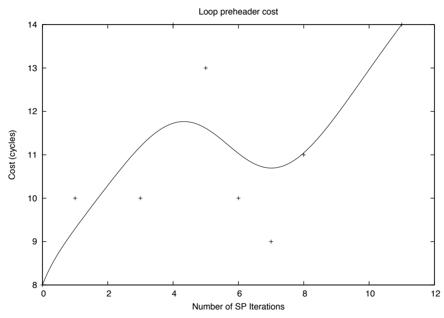
We have added an option to `c2dil` called `nosp` which will disable software pipelining for the given run. In this case, the software pipelining function is still called, but returns before any changes are made.

6 Evaluation

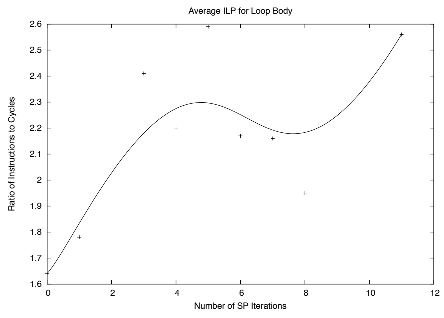
We evaluated our software pipelining algorithm for Pegasus by compiling a few small sample programs to demonstrate the potential benefit of our approach. Programs that have dependent loads and stores are generally prime candidates for achieving performance improvements via software-pipelining tech-



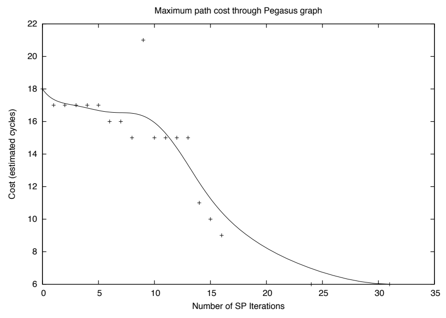
(a) Loop body cost



(b) Loop preheader cost



(c) Average ILP



(d) Path cost

Figure 4: Evaluation results for repeated software pipelining iterations on a moving average function.

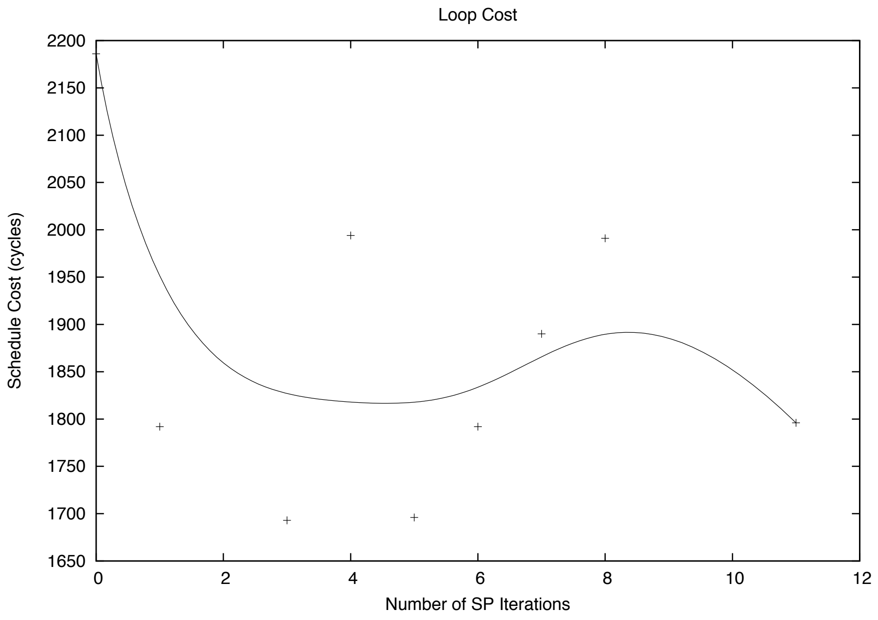


Figure 5: Approximate function costs for $n = 99$ iterations of the moving average function, calculated after each successive pipelining stage.

niques. Therefore, we present one such particular example in this section: a method to calculate a moving average of a set of data points.

```
void moving_avg(int *a)
{
  int i = 1;
  while (i < 100) {
    int t1 = a[i-1];
    int t2 = a[i];
    a[i] = (t1+t2)/2;
    i++;
  }
}
```

Figure 6: Sample code to compute a moving average of a 100 element array.

The code for our moving average function is shown in Figure 6. This function averages each element of an array with the previous element, and the result is written back in place to update the original value with the averaged value. In its unpipelined form, the store at line 7 has a true data dependency on the result of the two loads immediately before it. Additionally, there is a loop dependency between every value except the first one in the array (`a[i]`) and the value written just before it in the previous iteration (`a[i-1]`).

The goal of our software pipelining approach is to make available the right set of instructions to schedule between dependencies so that the actual loop body can perform more work in parallel. In order to evaluate our approach we analyze the Pegasus graph and the final assembly code produced after each incremental pipeline, where operations are chosen for pipelining according to the heuristics described in Section 7. Specifically, we focus on four metrics.

1. *The cost of the loop body*, as measured by the number of VLIW instructions.

2. *The cost of the loop pre-header*, as measured by the number of VLIW instructions.

3. *The average instruction-level parallelism (ILP)*, as measured by the average number of operations in every VLIW instruction of the loop body

4. *The most expensive path of the loop*, discovered by a depth-first search starting at each *mu* in a hyperblock and ending at the *eta*s.

We expect that a good final schedule will have a smaller loop body cost since the loop body pattern will be more efficient. Some of these costs will be shifted into the loop pre-header, but since the pre-header is only executed once, compared to the loop body which is executed a possibly large number of times, we can tolerate a fair amount of pre-header code expansion and still achieve better performance.

Figure 4(a) and Figure 4(b) graph the loop body and pre-header costs after each operation is pipelined. The data appears to have a good amount of variability but the smooth bezier trend lines show a pattern of generally increasing pre-header costs offset by decreasing loop body costs. Through inspection we see that pipelining either 3, 5, or 11 operations leads to relatively better loop body performance than other schedules. A feedback approach with these metrics would prefer to stop pipelining after 3 iterations of our algorithm, since at about the same efficiency of the loop body we prefer smaller pre-header costs and less pipelining stages to limit the amount of register pressure.

The average instruction-level parallelism demonstrates how much work is actually being performed in parallel, so higher levels of parallelism generally correspond with more efficient code. Perhaps more importantly, the average ILP also reveals how much room there is for improvement, since there are a limited number of execution units that can possibly be scheduled in parallel. It is the job of the scheduler to convert a higher ILP into an actual savings on loop body instructions.

Figure 4(c) graphs the average instruction-level parallelism of the loop body, and once again identifies iterations 3, 5, and 11 as good stopping points. As expected, the positive trend line supports the notion that software pipelining tends to improve the number of operations scheduled in parallel.

The most-expensive path metric is used as a heuristic in our approach because it can reveal many glaring dependencies between operations, which makes the operations on the most expensive path better candidates for software pipelining. This metric can also be calculated locally during the software pipelining stage without requiring a feedback loop from other stages of the compiler.

Figure 4(d) graphs the most expensive paths remaining in the loop hyperblock after each iteration of our pipelining algorithm. We can see a steady decline in the most expensive path lengths which level off at a cost of 6, which is equivalent to a graph with a load that is the only operation between an *eta* and a *mu*.

We calculate the approximate runtime cost of a loop function as follows:

$$Cost(fnc) = Cost(Preheader) + n * Cost(LoopBody)$$

In this equation, n is equal to the number of iterations of the loop, and the cost of the pre-header and loop body are the two metrics measured earlier, corresponding to the number of VLIW instructions in the final assembly code.

In the moving average example, the initial function cost is 2186 VLIW instruction, calculated for the value $n = 99$. Figure 6 graphs the resulting cost of the function after every iteration of our software pipelining algorithm. As we discovered earlier through inspection, 3 iterations yield the best performance with an approximate cost of 1796 VLIW instructions. This represents an approximate improvement of 18% over the non-pipelined version, and demonstrates the effectiveness of a feedback loop between the results of the scheduler and the aggressiveness of the software pipelining stage.

The complete Pegasus graphs for the moving average functions are available online, showing the state of the graph before any software pipelining and after 3 and 11 iterations of our algorithm [8]. Software pipelining can make the data flow of these graphs much more complex to follow, as is apparent in the actual Pegasus graph output.

7 Discussion

While designing our software pipelining algorithm, we have discovered several areas for improvement. These improvements center around heuristics used at different stages of the pipelining process. In this section, we discuss several of these heuristics as well as possible improvements. Here we also discuss limitations of our current implementation.

Optimization Heuristics There are two main heuristics used in our algorithm. Since we use an iterative approach that moves one operation per iteration, it becomes necessary to prioritize instructions available for pipelining and determine when further pipelining will no longer be helpful.

In our current implementation, we choose which operation to pipeline by the length of data paths through the Pegasus circuit. This approach has worked well in our experiments. It's easy to understand and simple to implement. However, it's not without flaw. It's primary shortcoming is that it doesn't take advantage of any extra information in the system. For example, once we finish pipelining and start register allocation we will have spill information available. Therefore, an improved approach might use this information about spill to reorder operation priority in future pipelining attempts.

The second challenge of our algorithm is to decide when the optimal amount of pipelining has been done. Our experiments show that at a certain point pipelining will decrease performance significantly. Therefore, our algorithm needs to find the optimal point. The most obvious solution which we have discussed in Section 6 is to iterate pipelining, moving more operations each time to find a trend and choose the optimal pipeline. However, this approach is highly dependant on the previously discussed heuristics for choosing operation priority. Since operations are currently always chosen deterministically in the same order, we may miss orderings that would improve the schedule greatly.

In both of these cases, as with most compiler optimizations, a feedback approach is extremely useful. Software pipelining should run and then allow scheduling. The register allocation and scheduling attempt should feed information back to the pipelining stage to more intelligently select operations and the number of iterations in the next attempt. As scheduling attempts progress, this method might converge a bit closer to the optimal schedule than our current approach.

Software pipelining with no epilogue Traditional software pipelining adds both a prologue and an epilogue to loop bodies. In these cases, the prologue is responsible for filling the pipeline, the loop body executes on a full pipeline, and the epilogue is responsible for emptying the pipeline. However, in our implementation, we add only a prologue to loops. By copying operations into the prologue, we calculate values in iterations before they are used. This is a convenient solution, but there are two costs that are worth mentioning. First, we can only pipeline side-effect free operations. This is because operations that are pipelined will be executed for at least one extra iteration of the loop. Ordinarily the epilogue would have a version of the loop body that didn't contain these operations in an effort to avoid this phenomenon. This also implies that operations such as loads could access values in unintended memory locations, when pre-loading for the next iteration during the last loop iterations. Therefore, this operation requires a memory model that doesn't cause an exception when this occurs. Second, software pipelining without an epilogue is slightly less efficient. In traditional software pipelining, the epilogue would be a copy of the loop with all the operations in the prologue removed. However, in our approach, the last iterations of the loop can be thought of as repeating the operations in the prologue. Therefore, compared to a software pipelining strategy with an epilogue, we incur an additional cost for each operation that we add to the preheader. In practice, this has very little impact because the loop body itself will dominate execution. We feel that this cost is justified in the simplification it allows in the code.

Limitations In our current implementation, we have used several test cases to verify that our algorithm operates correctly. In each of these cases we have analyzed the output graphs to see that the pipelining operation has proceeded as we expect. In all cases, we believe the graphs to be correct. However, in some cases, the scheduler emits ASM that doesn't correctly reflect the graph. In these cases we have seen various errors, such as PC corruption or invalid output. These errors have prevented us from using additional data points in Figure 6, and is specifically responsible for the lack of ASM metric data points at iterations 2, 9, and 10. In discussions with Tim and Mahim, we have concluded the problem is most likely in the scheduler. Unfortunately, under the time constraints of this semester, we were unable to further investigate the scheduler.

8 Conclusion

We have proposed and implemented a software pipeling algorithm to be used directly on the Pegasus intermediate representation. Our approach to software pipelining recognizes most regular patterns of operations as pipelineable via our iterative pattern matching algorithm. We have also tested this algorithm on a variety of test cases and presented the analysis of one particular moving average example in this paper. In our analysis, we showed that our algorithm offers an approximate 18% improvement over the optimizations that currently exist in `c2dil`.

We conclude that software pipelining in Pegasus is a viable and worthwhile optimization. While we have identified several opportunities for future improvements in Section 7, we believe that our algorithm provides a significant performance improvement with our current heuristics. By using the metrics evaluated in Section 6, extending our algorithm to implement a more extensive feedback-based approach has the potential for even better performance.

9 Acknowledgements

We would like to thank Seth Goldstein, Tim Callahan, and Mahim Mishra for all of their help in developing our iterative software pipelining approach and learning the intricacies of `Pegasus` and the `c2dil` framework.

References

[1] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN88 Conference on Programming Languages Design and Implementation*, pages 308–317, June 1988.

[2] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *European Symposium on Programming*, pages 221–235, 1988.

[3] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[4] M. Budiu and S. Goldstein. Optimizing memory accesses for spatial computation, 2003.

[5] M. Budiu and S. C. Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, Pittburgh, PA 15213 - USA, April 2002.

[6] K. Ebcioglu and A. Nicolau. A global resource-constrained parallelization technique. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 154–163, New York, NY, USA, 1989. ACM Press.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[8] C. Hartwig and E. Krevat. Pegasus graphs for pipelined moving average example. `http://www.cs.cmu.edu/~chartwig/15-745`.

[9] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328, New York, NY, USA, 1988. ACM Press.

[10] A. Nicolau. Percolation scheduling: A parallel compilation technique. Technical report, Ithaca, NY, USA, 1985.

[11] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *ICPP*, pages 614–618, 1985.

[12] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II - Software, pages II–120–II–124, Boca Raton, FL, 1993. CRC Press.

[13] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th International Symposium on Microarchitecture (MICRO-27)*, pages 63–74, December 1994.

[14] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 283–299, New York, NY, USA, 1992. ACM Press.

[15] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 158–169, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.