Global Software Pipelining with Iteration Preselection

David Gregg

Institut für Computersprachen, Technische Universität Wien,
Argentinierstr. 8, A-1040 Wien,
E-mail: dave@complang.tuwien.ac.at
Fax: (+431) 58801-18598

Abstract. Software pipelining loops containing multiple paths is a very difficult problem. Loop shifting offers the possibility of a close to optimal schedule with acceptable code growth. Deciding how often to shift each operation is difficult, and existing heuristics are rather *ad hoc*. We separate loop shifting from scheduling, and present new, non-greedy heuristics. Experimental results show that our approach yields better performance and less code growth.

1 Introduction

Instruction Level Parallelism (ILP) offers the hope of greatly faster computers by automatically overlapping the execution of many machine-level instructions to complete tasks more quickly. An important class of ILP machine is the Very Long Instruction Word computer. These simple machines provide large numbers of execution resources, but require a sophisticated compiler to schedule the instructions. A particularly important scheduling technique is software pipelining, which can produce very compact schedules for loops.

This paper focuses on global software pipelining, that is pipelining loops containing branches. Our approach is based on shifting operations across the loop entry. This technique is used in several important software pipelining algorithms [ME97,NN97,Jai91,DH99]. Most loop shifting algorithms use an iterative approach to software pipelining. These algorithms interleave acyclic scheduling of the loop body, and shifting operations from one iteration to another. Acyclic scheduling has been studied in great detail and a number of good algorithms exist. Loop shifting has been less studied, and most algorithms shift operations, one iteration at a time, and on quite an *ad hoc* basis.

The fundamental problem of loop shifting is how to know which operations should be moved across the loop back edge, and how many times. Algorithms which move operations one iteration at a time are similar to early attempts at global acyclic scheduling which percolated operations from on basic block to the next without any final intended destination. Moving an operation one iteration at a time may create a temporarily worse schedule, that can later be transformed into a better one [Rau94]. The problem is to distinguish between such good moves, and shifts that genuinely make the schedule worse.

Despite the problems of loop shifting, it has one very important advantage — it extends naturally to software pipelining loops containing branches. Loop shifting algorithms can pipeline such loops using existing acyclic scheduling techniques. Most importantly, the initiation interval (II) of the resulting pipeline is not fixed; it can vary depending on the control flow path. Other algorithms can achieve a variable II, but only at the cost of large scale code duplication or scheduling restrictions which may increase the average II. Furthermore, loop shifting naturally allows operations to be scheduled ahead of branches upon which they are control dependent. Many other approaches limit ILP by treating control dependences as data dependences.

This paper deals with software pipelining of integer code using loop shifting. Such code often contains loops with branches and low trip counts. With such loops it is important to both minimize the average II of the pipeline, and to control code growth and other resource usage. With low trip count loops the cost of loop start-up, such as cache misses from an over-large loop prolog, may be just as important as the average II for the loop. Therefore, we concentrate on shifting the loop only enough to make the minimum II attainable. Only after shifting is complete do we apply acyclic scheduling.

The paper is organised as follows. First we describe existing loop shifting techniques. Section 3 describes our novel approach to software pipelining loops containing branches. In section 4 we present experimental results from implementing the algorithm in an ILP compiler. In section 5 we look at other work in this area. And finally we outline some conclusions and open questions for our future research.

2 Shift and Schedule Algorithms

We will first describe shifting algorithms in more detail. These algorithms move operations from inside the loop across the loop back-edge and into the region of code before the start of the loop. This produces a compensation code copy of the moved operation at the end of the loop. Thus operations move from one iteration to another inside the loop. This process is often know as *shifting* the operation.

The most prominent loop shifting algorithms interleave acyclic scheduling and shifting [ME97,NN97,Jai91]. A problem with these algorithms is their rules for choosing which operations to shift. The candidates for shifting are those operations which the acyclic scheduling step has placed toward the start of the acyclic schedule. At each acyclic scheduling stage, operations are scheduled as early as possible. This strategy can work well for operations that are members of the longest dependence chains in the loop, since dependences prevent them being scheduled too early. Other operations may be scheduled far earlier than necessary.

There are a number of problems with shifting operations more often than necessary. First, functional units may be used for speculatively executing operations which could be scheduled less speculatively without delaying the schedule.

This is especially true for loops containing unpredictable branches. Secondly, the length of the register lifetimes will increase, thus increasing register pressure, and register spills. Values that are live across loop iterations may need renaming to maintain the correctness of the code. Renaming requires copy operations and/or loop unrolling, neither of which is desirable. Shifting operations increases code growth in the prolog and loop body, increasing cache misses. Finally, shifting operations can, under some circumstances, increase the length of the new loop body rather than decreasing it.

In order to avoid these problems, existing algorithms try to limit their greediness in a number of ways. Enhanced Pipeline Scheduling (EPS) uses a limited size scheduling window. At each cycle during scheduling, only the first K (usually $K = 16$) operations operations on each path are eligible to be scheduled. This limits greediness somewhat, but operations within the window can still be scheduled too early, and operations outside the window may be shifted too little. EPS also tries to schedule the loop acyclically as much as possible. Once all the operations on a given path have been scheduled once, no further shifting of the loop is allowed on that path. When all control flow paths have reached this state, scheduling is complete. Furthermore, priority is given to operations which are shifted a smaller number of times.

Resource Directed Loop Pipelining (RDLP) has weaker controls. It calculates the minimum II (MII) for each control flow path through the loop. It then alternates between acyclic scheduling and shifting all operations in the first cycle of the schedule. This process continues until the MII is reached on all path, or some limit on the number of shifts has been reached. Specifying a lower limit makes the algorithm less greedy, but reduces the opportunities for pipelining.

Both EPS and RDLP use a greedy strategy, but with some rather *ad hoc* heuristics to try to limit the greediness. The result is that many operations will be shifted too much and an occasional one shifted too little. What is really needed is a strategy that shifts operations just enough. In the next section, we present our approach to doing exactly this.

3 Iteration Preselection

Iteration Preselection is a new type of *Decomposed Software Pipelining* (DESP) algorithm [WE93]. Rather than trying to shift and schedule together, we break pipelining into two stages. First, we calculate the number of times each operation should be shifted using only an approximation of the resource constraints. We then shift each operation the appropriate number of times. In the second stage, we schedule the resulting loop body using an existing acyclic scheduling algorithm. It is only in this stage that we take full account of the resource constraints on the loop.

All DESP algorithms try to simplify software pipelining by breaking it into the two simpler problems of choosing to how many times to shift each operation in the loop body, and acyclic scheduling. Although the computational complexity of these two subproblems is not smaller than that of the software pipelining problem, good heuristics can be found for both subproblems. It can be more difficult to find such good heuristics which solve both subproblems together.

Existing DESP algorithms concentrate on loops containing a single basic block. In our opinion, however, the true strength of the DESP idea lies in global software pipelining. Global software pipelining with a variable II and reasonable code growth is an enormously difficult problem. Good heuristics can be found for the two subproblems, however. Acyclic global scheduling has been well studied and good heuristic algorithms such as Selective Scheduling [ME97] and DAGGER [CYS98] produce excellent results. The remaining problem is heuristics for shifting operations.

3.1 Single Path Loops

Our approach to shifting single path loops involves three steps. First, we calculate the MII for the loop. Secondly, we calculate the smallest number of times that each operation needs to be shifted for this II to be reached, assuming infinite resources. Finally, we shift the operations.

We calculate the MII using Lam's [Lam88] approach to calculating the initial MII for modulo scheduling[1]. This calculation finds the precise MII based on dependences, but uses only an approximation for the effects of resources. Thus, our algorithm takes some account of resources at this step, but the detailed resource allocation is not done until acyclic scheduling.

The goal of second step is to reduce the length of the longest acyclic dependence chain in the loop to the length of the MII. Such a loop can certainly be acyclically scheduled in MII cycles on a machine with infinite resources. The *height* of an operation measures the length of the longest acyclic dependence chain from that operation to the end of the loop. To reduce the acyclic dependence length, it is necessary to shift operations whose height is greater than the II. We call the number of times that an operation *op* must be shifted the *Times to Shift* or TTS(*op*). In the absence of cyclic dependences, we can calculate the TTS with the following formula.

$$\text{TTS}(op) = (\text{height}(op) - 1) \;/\; \text{MII}$$

In the presence of cyclic dependences, the situation is more complicated. Cyclic dependences may cause the height of an operation to increase when other ops are shifted. We introduce the notion of *circular height* (*CH*) to solve this problem. Intutively, Circular Height is a measure of the height of an operation, plus the amount that the operation's height will increase when dependent operations are shifted to below it. More formally, we calculate *CH* on the data dependence graph of the loop. The data dependence graph contains vertices representing operations, and directed edges for dependences. Each edge has a weight representing dependence distance. Edges with a zero weight are acyclic

[1] Our original formulation looked only at the dependence MII. The idea of using the resource MII as-well comes from the EPS++ algorithm.

dependences. Let $succ(op)$ denote the set of vertices v such that there exists an edge $(op,\, v)$. We initialise the CH of each operation op to latency(op). We calculate the final CH values using the following algorithm.

```
repeat
    change = FALSE
    for each v ϵ succ(op)
        ht = latency(op) + height(v) - weight(op, v) * MII
        if ( ht > CH(op) )
            CH(op) = ht
            change = TRUE
        endif
    endfor
until change == FALSE
```

These circular heights can be used as the height in the TTS formula above to calculate the number of times to shift each operation. The final stage of loop shifting is to actually shift the operations, using existing operation movement techniques.

3.2 Multiple Paths

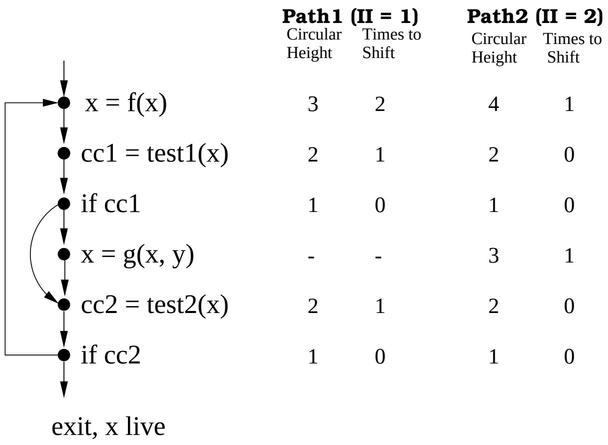
Calculating the number of times to shift each operation is considerably more complicated in the presence of multiple paths. First, the II for each path may be different. Secondly, several control flow paths may contain the same instruction. This single instruction may have different heights depending on which path one considers. A further complication is with *cross paths*. A cross path is the section of code dealing with control flow from one path to another. The idea of II is very difficult to apply to cross paths, since they involve moving from one path to another, perhaps every iteration.

To attempt to satisfy these conflicting requirements, we use a heuristic based on the single path case. We treat each path separately. For each path through the loop, we calculate the TTS for each operation. Branches are assigned a CH equal to the greater of their latency, or the CH of the operation op that computes the branch condition, minus the latency of op.

Based on these calculations we shift operations. If an operation falls on more than one path, we choose the maximum TTS from all the different paths. If an operation is duplicated while moving, due to the normal effects of compensation code in global scheduling, the two copies may be shifted independently.

3.3 An Example

Figure 1 shows the control flow graph for a simple loop containing two control flow paths. The MII for Path1 is one, and is two for path2. The figure also shows the circular height and TTS of each operation. All operations have a latency of one.



	Path1 (II = 1) Circular Height	Path1 (II = 1) Times to Shift	Path2 (II = 2) Circular Height	Path2 (II = 2) Times to Shift
x = f(x)	3	2	4	1
cc1 = test1(x)	2	1	2	0
if cc1	1	0	1	0
x = g(x, y)	-	-	3	1
cc2 = test2(x)	2	1	2	0
if cc2	1	0	1	0

Fig. 1. Example loop with variable II

First, we ensure that all loop back edges come from unconditional gotos rather than conditional branches. Where necessary, we add additional dummy basic blocks which branch back to the loop entry. This allows operations to move across the loop entry without necessarily becoming speculative. Part (a) of figure 2 shows the transformed CFG. Note that in our notation, a control flow edge that continues in a straight line represents the fall through path, while a curved control flow edge represents the taken path of a conditional branch.

Figure 1 shows that the operation $x = f(x)$ should be shifted. The loop entry is a join point in the CFG, with two incoming edges. Moving the op above this join will leave it just before the loop entry, and create a compensation copy just before the goto at the end of the loop, as in part (b). The operation $cc = test1(x)$ is moved in the same way going from part(b) to part (c). Moving from (c) to (d), the operation $x = g(x,\ y)$ moves speculatively above *if cc1* and out of the loop. The register x is live at this point, however, so the target register of the operation is renamed to *x'*. A copy operation is added at the original point in the CFG to copy *x'* to x, should that path be followed.

Part (e) considers the operation $cc2 = test2(x)$. This operation is on both paths but should only be shifted once on path1. On path2, it can stay in the current iteration. Moving the operation along path1, it first crosses the join point. This creates a compensation copy on path2, just above the join. A further compensation copy is created at the end of the loop, when the operation moves across the loop entry.

Finally, part (f) shows how the operation $x = f(x)$ is shifted a second time on path1. This creates a compensation copy for both joins that the operation moves across. A renaming copy operation is also needed. This operation is not shifted again on path2, since our original calculations (see figure 1) showed that it should be shifted only once on that path.

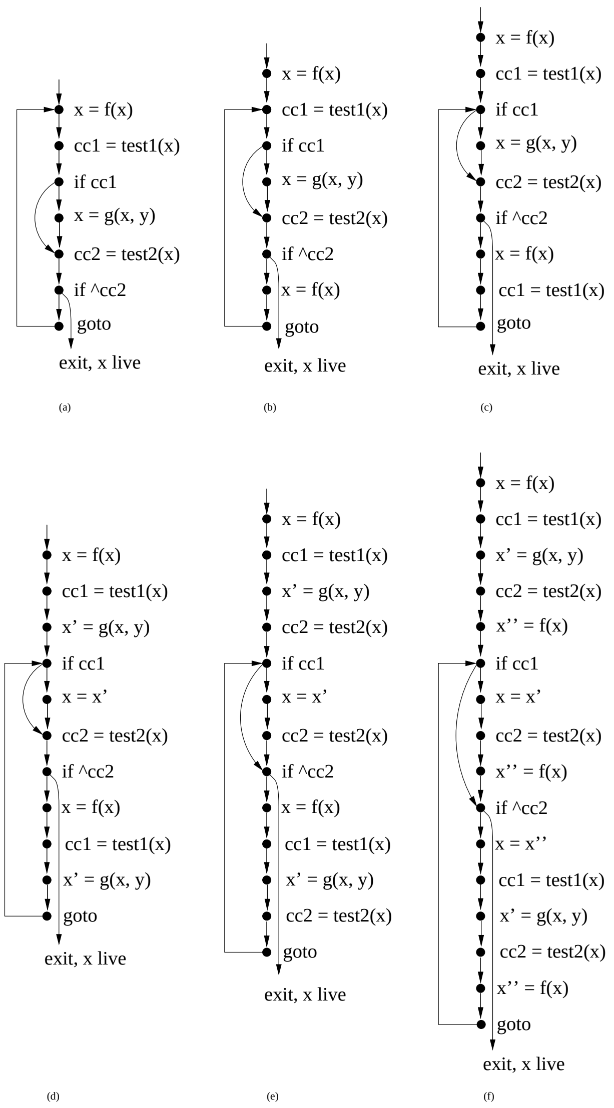


Fig. 2. Loop shifting steps

3.4 Acyclic Scheduling

Once shifting is complete, the resulting loop body is scheduled. We use a conventional DAG scheduling algorithm based on Selective Scheduling [ME97]. Any global scheduling algorithm such as trace scheduling can be used, but a general DAG approach allows operations from all paths to be scheduled together, making it more likely that the MII will be reached on all paths.

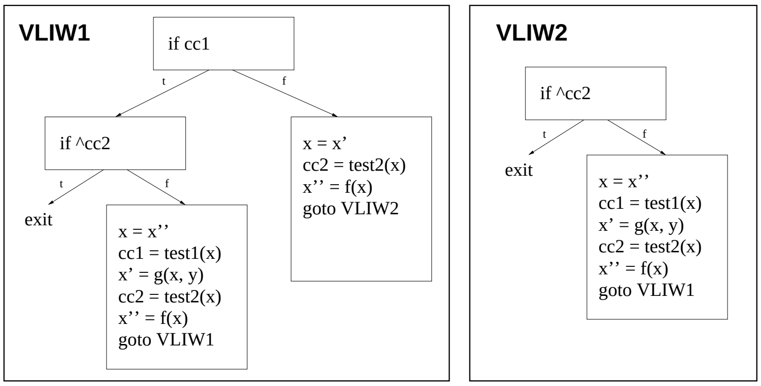


Fig. 3. Final Kernel Schedule

The final schedule for the example loop kernel appears in figure 3. The loop has been compacted into two VLIWs. The machine model is IBM's Tree VLIW, where a VLIW consists of a tree shaped control flow graph of operations. Only those operations on the path through the tree that is followed at run time are allowed to write their results back to the register file. No data dependences are allowed between operations in the same VLIW[2]. When path1 is followed, the machine repeatedly executes *VLIW1*, and achieves an II of one. When control moves to path2, the machine alternates between executing *VLIW1* and *VLIW2*, yielding an II of two.

Our approach is not limited to the Tree VLIW architecture, however. The two main features of the Tree VLIW are multi-way branching and conditional write-back. Architectures which allow only two way branching can be modeled by allowing only a single branch per VLIW. Conditional write-back can be modeled with predication, or by simply moving the operations in the VLIW above all branches in the VLIW.

---

[2] An exception to this general rule is with copy operations. An operation may read a register that is written by a copy operation in the same VLIW. These dependences on copy operations can easily be removed in a final pass over the VLIWs [ME97]

4 Experimental Results

We implemented Iteration Preselection in the Chameleon Compiler and ILP test-bed. Chameleon provides a highly optimizing ILP compiler for IBM's Tree VLIW architecture. The compiler performs many sophisticated traditional and ILP increasing optimizations. For comparison, we also implemented a version of Enhanced Pipeline Scheduling[3] with window size 16. Both algorithms used the same acyclic scheduling scheme. We compiled and scheduled benchmark programs whose inner loops contain branches and ran them on Chameleon's VLIW machine simulator.

The Chameleon compiler uses an existing C compiler (in our case *gcc*) as a front end to generate the original sequential machine code. The baseline for our speedup calculation is the number of cycles it takes to run this code on a simulated one ALU VLIW machine. This sequential code is then optimised to increase ILP, rescheduled by our software pipelining phase, and registers are re-allocated. We run this scheduled code on a machine simulator with the correct number of ALUs to calculate the number of cycles taken. The speedup is the baseline cycles, divided by the number needed for the relevant VLIW.

Benchmark	Description
wc	Unix word count utility
eight	Solve eight queens problem
bubble	Bubble sort array of random integers
bsearch	Binary search sorted array
eqn	Inner loop of eqntott

The purpose of the EPS limited scheduling window is not always understood. It exists to reduce the greediness of EPS scheduling and so *increase* the speedup while reducing code growth, register pressure and compilation time. The paper [NE93] examines the effect of varying the window size, and established 16 as a good size for balancing greediness against scheduling freedom. Our own experiments confirm this finding. In no case does increasing the window size to 32 give any but the most minor increase in speedup. Decreasing the window size below 16 usually reduces the speedup except in the case of bubble, where it sometimes produces faster code.

The speedup results for Iteration Preslection are very encouraging. Clearly, our non-greedy heuristics are shifting operations sufficiently, but no more. Our algorithm does better by preserving VLIW slots for operations which need to execute early. The only exceptions 8-ALU bubble and eqn, where our algorithm performs a lot of shifting to make a very low II achievable. The acyclic scheduler fails to pack the instructions into a schedule that length. It appears that in

[3] Our version does not use the original EPS register allocation scheme. Our scheduling window is also implemented slightly differently

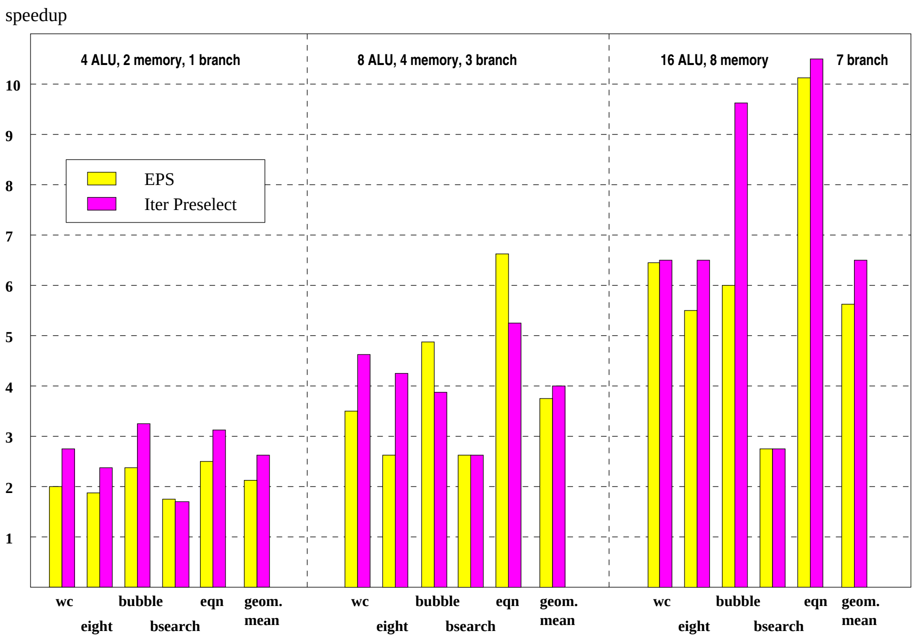


Fig. 4. Speedup

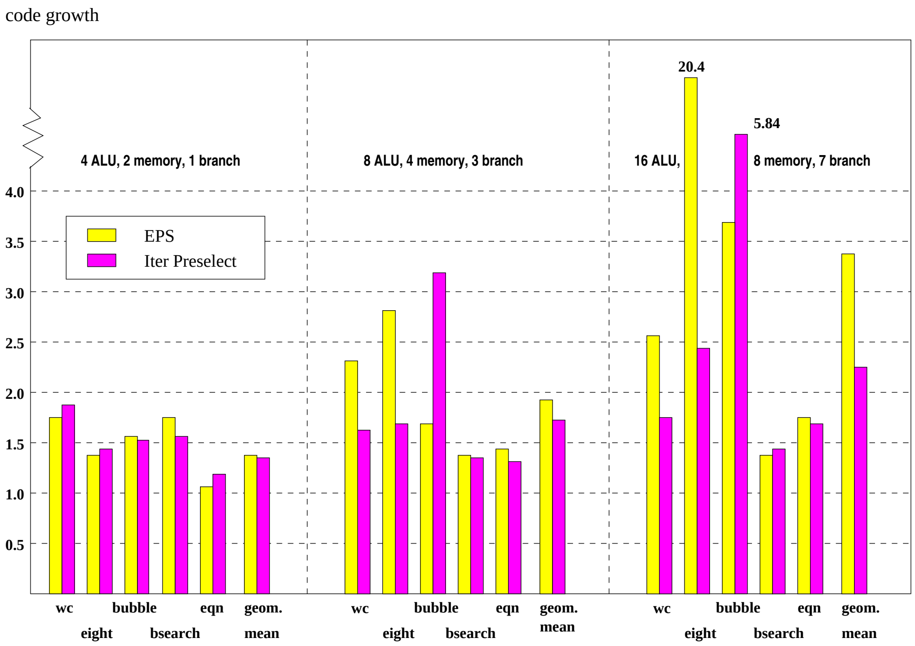


Fig. 5. Code Growth

these cases the resource MII that we compute is too optimistic in the presence of multiple paths. In general, however, iteration preselection performs better than EPS.

Iteration Preslection also controls code growth well. Again, our non-greedy strategy reduces both the size of the loop prolog and code growth from moving operations through the loop while shifting. The difference is most notable on the larger machines, where EPS's greedy scheduling combined with abundant resources place little limit on the code growth[4]. Where Iteration Preselection produces more code growth, it is normally the result of producing a pipeline with lower average II, and significant speedup. Smaller code size should also lead to faster execution time on machines with caches. We intend to demonstrate this with a cache simulator in future work.

5 Related Work

A large number of algorithms exist for global software pipelining with multiple IIs. Unrolling kernel recognition algorithms [AN90] are theoretically very powerful. In practice, however, they cause enormous code growth and their controlling heuristics restrict their power [NN97]. Several algorithms use code duplication to separate all paths in the loop and pipeline each separately. Again, the code growth from code duplication and cross paths can be huge [SM98]. A number of other strategies exist which trade code growth for restrictions on the attainable II [WPP95,SL96].

A number of DESP [WE93] algorithms exist. These algorithms use a similar strategy as ours for TTS numbers for the strongly connected components in the dependence graph. But they differ substantially in the placement of other operations. In addition to minimising the dependence length of the loop body, they also seek to remove restrictions on the acyclic scheduler. These DESP algorithms to shift operations very many times to try to convert acyclic dependences to loop carried ones. A global version of the original DESP algorithm has been proposed, but it assumes a fixed II when calculating the number of times to shift an operation, and treats control dependences as data dependences. The most recent work on GDESP separates all paths before pipelining.

EPS++ is an unpublished algorithm developed by IBM's VLIW group. This algorithm uses a non-greedy, as late as possible, acyclic scheduling algorithm for EPS. In a single path loop, this will place the operation in the same iteration as our approach. A modulo schedule is computed for the most commonly followed path, and the heights of operations are adjusted to ensure that the scheduling algorithm will place those operations in the cycle designated by the modulo

[4] The program *eight* is a particularly bad pathological case for EPS, combining a loop with a large initial acyclic height and a number of branches which can be scheduled almost arbitrarily early.

schedule. Software pipelining of other paths is only allowed in so far as it does not interfere with the most common path[5].

Recent work [DH99] on loop shifting by circuit re-timing has yielded fascinating theoretical results for single path loops. We do not believe this research is applicable to our work, however, since it seeks to shift operations much more than necessary. We also remain convinced that loop shifting is not the best strategy for local software pipelining. Unless a loop contains branches, modulo scheduling is likely to produce better results.

6 Conclusions and Future Work

We have presented a new decomposed algorithm for global software pipelining general purpose integer code. Our algorithm allows a variable II and speculative scheduling, while controlling code growth. We have demonstrated that the DESP framework is very suited to global software pipelining. In fact, the true potential of DESP seems to be in pipelining loops containing branches, rather than local software pipelining which already has many good solutions. The implementation of our algorithm shows that when compared with EPS it generally achieves better speedups or less code growth or both.

An open problem remains with operations which fall on several paths. Conflicting TTS numbers for different paths may cause poor scheduling decisions. Our future work will look at other strategies for choosing the TTS in this situation. We believe, however, that our experimental results show that the existing strategy is very successful for the programs tested.

7 Acknowledgments

We would like to thank the VLIW group at IBM's T. J. Watson Research Center for providing us with the Chameleon experimental test-bed. Special thanks to Mayan Moudgill and Michael Gschwind. Thanks also to Sylvain Lelait and Anton Ertl for their comments on this paper.

This research was supported by the Austrian Science Foundation (FWF).

References

[AN90] Alexander Aiken and Alexandru Nicolau. A realistic resource-constrained software pipelining algorithm. *Advances in Languages and Compilers for Parallel Computing*, pages 274–290, 1990.

[CYS98] Gang Chen, Cliff Young, and Michael Smith. Practical and profitable alternatives to greedy, single-path scheduling. Harvard University. Submitted to MICRO 31, November 1998.

[5] Initial experiments show that EPS++ produces similar or better results than ours, when execution of a loop is dominated by one path, and when EPS++ successfully identifies that path. Where this is not true, as in wc and bubble, EPS++ performs worse.

[DH99] Alain Darte and Guillaume Huard. Loop shifting for loop compaction. Technical report, Ecole Normale Sup/'erieure de Lyon, 1999.

[Jai91] S. Jain. Circular scheduling: A new technique to perform software pipelining. In *SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 219–228, June 1991.

[Lam88] Monica Lam. Software pipelining: An efficient scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN 88 Symposium on Programming Language Design and Implementation*, pages 318–328. ACM, June 1988.

[ME97] S. Moon and K. Ebcioğlu. Parallelisizing nonnumerical code with selective scheduling and software pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, November 1997.

[NE93] Toshio Nakatani and Kemal Ebcioğlu. Making compaction-based parallelization affordable. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):1014–1029, September 1993.

[NN97] Steve Novack and Alexandru Nicolau. Resource directed loop pipelining: Exposing just enough parallelism. *The Computer Journal*, 10(6), 1997.

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *27th Annual International Conference on Microarchitecture*. ACM, December 1994.

[SL96] M. Stoodley and C. Lee. Software pipelining of loops with conditional branches. In *29th International Symposium on Microarchitecture (MICRO-29)*, pages 262–273. IEEE/ACM, December 1996.

[SM98] SangMin Shim and Soo-Mook Moon. Split-path enhanced pipeline scheduling for loops with control flows. In *Micro 31*, pages 290–302. ACM/IEEE, November 1998.

[WE93] Jian Wang and Christine Eisenbeis. Decomposed software pipelinig. Rapports de Recherche 1838, INRIA Rocquencourt, F - 79153 Le Chesnay Cedex, January 1993.

[WPP95] Nancy J. Warter-Perez and Noubar Partamnian. Modulo scheduling with multiple initiation intervals. In *28th Annual International Conference on Microarchitecture*, pages 111–118. ACM, December 1995.