University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Multidimensional Kernel Generation for Loop Nest Software Pipelining

Alban Douillet, Hongbo Rong, Guang R. Gao

CAPSL Technical Memo 64
February 14, 2006

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

Single-dimension Software Pipelining (SSP) has been proposed as an effective software pipelining technique for multidimensional loops [18]. However, the scheduling methods that actually produce the kernel code have not been published yet. Because of the multi-dimensional nature of SSP kernels, the scheduling problem is more complex and challenging than with modulo scheduling. The scheduler must handle multiple subkernels and initiation rates, scheduling constraints specific to SSP while producing a solution that minimizes the execution time of the final schedule.

In this paper three approaches are proposed: the flat method, which schedules operations from different loop levels with the same priority, the level-by-level method, which schedules innermost operations first and does not let other operations interfere with the already scheduled levels, and the hybrid method, which uses the level-by-level mechanism for the innermost level and the flat solution for the other levels. We also break a scheduling constraint introduced in earlier publications and allow for a more compact kernel. The proposed approaches, combined with different sets of heuristics, have been implemented in the Open64/ORC compiler, compared on loop nests from the Livermore and NAS benchmarks.

Contents

List of Figures

1 Introduction

Software pipelining (SWP) is an important loop scheduling technique that overlaps the execution of consecutive iterations of a loop to explores instruction-level parallelism [9, 14, 1, 8, 15, 19, 7, 2, 11, 10, 22]. Traditionally, it is applied to the innermost loop of a loop nest. The schedule can be extended to outer loops by hierarchical reduction [9, 12, 20]. Alternately, loop transformations can be performed to the innermost loop before SWP [3, 21, 13].

Single-dimension Software Pipelining (SSP) [18] is a unique resource-constrained framework for software pipelining a loop nest. The scheduling technique overlaps the iterations of any loop in a loop nest which satisfy the dependence constraints. The compilation framework is shown in Fig. 1. First the loop level deemed the most profitable is selected and the multi-dimensional data dependence graph (n-D DDG) is simplified into a one-dimensional DDG (1-D DDG) to be sent as input to the scheduler [18]. The kernel is then computed. If the register pressure is reasonable [5], registers are allocated [16] and the final code is generated [17].

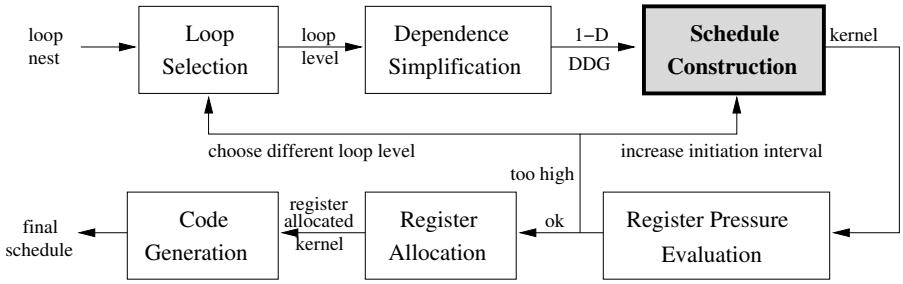


Figure 1: SSP Compilation Framework

However, the scheduling methods that actually produce the kernel code have not been published yet. In this paper, we present for the first time algorithms for the *kernel generation* step. The kernel generation itself is far from simple - as it involves the overlapping of operations from several iteration levels (dimensions) of a loop nest, a challenge not encountered in traditional software pipelining. There now is one subkernel per loop level in the loop nest, each one with its own initiation interval. Those subkernels interact with each other and optimizing one subkernel could have a negative impact on the others. Moreover, when the scheduler fails and the initiation interval must be increased, which subkernel should be chosen? After all, the main challenge is to generate a kernel that will, at the same time, minimize the execution time of the final multi-dimensional schedule.

Three approaches are proposed and studied. The *level-by-level* approach schedules the subkernel one by one starting from the innermost. Once a subkernel has been scheduled, it cannot be undone. The *flat* approach does not lock a subkernel once fully scheduled. Operations from any loop level may be considered and undo previous decisions made in a different subkernel. A larger solution space can therefore be explored. Finally, the *hybrid* approach schedules the innermost subkernel first and locks it. The other operations are then scheduled using the flat method. It allows for a shorter compilation time than the fast method while exploring a large solution space and focusing resources on the innermost loop.

We also break an SSP limitation that forced operation from different loop levels to be scheduled in distinct stages and that may artificially bloat the size of the kernel. We prove that, with minor modifications to the code generator and without code size increase, operations other than innermost can actually be scheduled in the same stage than operations from a different level.

The proposed approaches and heuristics associated with them have been implemented in the Open64/ORC compiler and analyzed on loop nests from the Livermore and NAS benchmarks. Experimental results show that the three approaches produces comparable schedules in terms of execution time and register pressure. However the level-by-level approach is more scalable and less dependent on the heuristics being used. The hybrid approach suffers from the drawbacks of the two other methods.

The rest of the paper is organized as follows. First, the modulo scheduling technique used for single loops is reviewed, followed by the SSP technique for multi-dimensional loops. In section 2, the kernel generation problem for SSP is presented, along with the associated issues. Section 3 explain how to schedule operations from different levels into the same stage. The next section presents the scheduling methods in details. The last three sections are devoted toward experiments, related work, and conclusion, respectively.

2 The SSP Kernel Generation Problem

2.1 Single-Dimension Software Pipelining

Single-dimension Software Pipelining (SSP) [18, 17, 16, 5] is a resource-constrained software pipelining method for both perfect and imperfect loop nests with a rectangular iteration space. Unlike other approaches [9, 6, 20, 12], SSP does not necessarily software pipeline the innermost loop of a loop nest, but directly software pipelines the loop level estimated to be the most profitable. The advantage of SSP over modulo-scheduling (MS) is that instruction-level parallelism or data cache reuse properties present in the outer loops are now accessible. Without prior iteration space transformations, a faster schedule with better cache performance can be found. If the innermost loop level is chosen, SSP is equivalent to classical modulo scheduling. SSP retains the simplicity of modulo scheduling, and yet may achieve significantly higher performance [18].

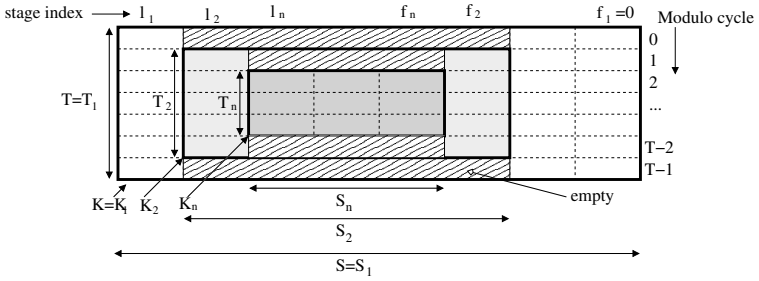


Figure 2: Generic SSP Kernel

The final SSP schedule is derived from the SSP kernel. Unlike MS kernels, the SSP kernels have

multiple initiation intervals and are composed of one subkernel K_i per loop level i in the loop nest. Each subkernel has its own number of stages S_i and initiation interval T_i. We note f_i and l_i the index of the first and last stages of K_i in the full kernel. Some slots must be empty because of the level separation constraint presented next (Fig. 2).

2.2 Problem Statement

The operations in the kernel must obey the constraints described below. The resource constraint is the same as with MS. However the dependence constraints now includes the number of unused cycles term (uc), corresponding to the empty stages. uc will be defined in Sec. 4. The sequential and level separation constraints only exist in SSP. Let $\sigma(op)$ be the schedule time of operation op in the kernel. Then we have:

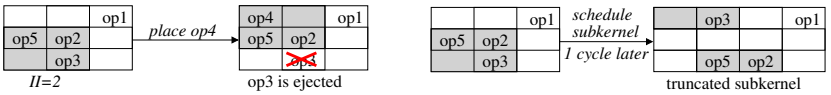
- **Resource Constraint**: at any given cycle of the kernel, a hardware resource is not allocated to more than one operation.
- **Dependence Constraint**: $\sigma(op_1) + \delta \leq \sigma(op_2) + k * T - uc(op_1, op_2, k)$ for all the dependences from the 1-D DDG from op_1 to op_2 where δ is the latency of the dependence and k the distance.
- **Sequential Constraint**: $\sigma(op) + \delta \leq S_p * T_n$ for every positive dependence $\overrightarrow{d} = (d_1, ..., d_n)$ originating from op in the original multi-dimensional DDG and where d_p is the first non-null element in the subvector $(d_2, ..., d_n)$.
- **Level Separation Constraint**: operations from different loop levels cannot be scheduled in the same stage.

The SSP kernel generation problem can then be formulated as follows: given a set of loop nest operations and the associated 1-D DDG, schedule the operations so that the scheduling constraints are honored and the initiation interval of each subkernel is minimized. Even in the simplest case where the loop nest is a single loop, the problem is NP-hard.

2.3 Issues

Besides satisfying all the constraints mentioned above, several issues need to be solved. First, the kernel is composed of subkernel with different initiation intervals which must be respected during the scheduling process. In Fig. 3(a), the II of the innermost kernel is 2. When inserting op_4, op_3 must be ejected to maintain the current II. Also, if a subkernel is rescheduled to a different cycle, one must make sure that subkernel is not truncated as shown in Fig. 3(b)

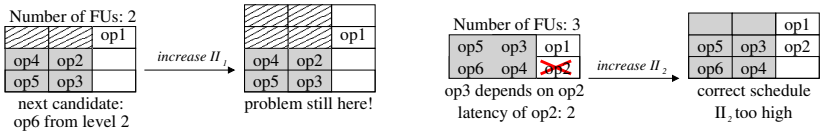
Then the multiple II feature raises issues of its own. Which loop level should have priority when minimizing the II of the subkernels. When the scheduler cannot find a solution and the II of one subkernel must be incremented, which subkernel to choose. Fig. 4 shows examples of poor schedule because of poor II increment decisions.



(a) Strict Initiation Rate of Subkernels

(b) Truncation of Subkernels

Figure 3: Multiple Subkernels Issues



(a) Decision does not solve the problem

(b) Decision is not optimal

Figure 4: Examples of Poor II Increment Decisions

3 Breaking the Level Separation Constraint

The level separation constraint was originally introduced for implementation reasons. Operations from different levels were required to be scheduled in distinct stages. We now show that this constraint is unnecessary and can be simplified. The advantage is two-fold. First, it gives more freedom to the scheduler which may be able to find a more compact kernel as shown in Fig. 5 where op_1 and op_2 are from different loop levels but scheduled in the same stage. Second, because the number of stages is decreased, so is the register pressure.

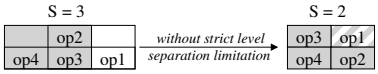


Figure 5: Removal of the Level Separation Constraint

To produce a correct final schedule from such a kernel, it is sufficient to conditionally emit the operations of the kernel. During the emission of stages for the execution of loop level i only operations from level i and deeper are emitted. If operations from other loop levels are present in the stage, they are simply ignored. Fig 6 shows the example of an outermost operation op_2, which conditionally appears in a stage of the middle loop.

Since the innermost loop is most frequently executed, it is not desirable to put operations from other levels into the innermost sub-kernel, in case they artificially increase its II. Also the conditional emission of operations in the innermost stages still require code duplication to be used. Therefore, we enforce instead a weak limitation, called the *innermost level separation limitation*, that forbids outer loop operations to be scheduled into the innermost loop stages.

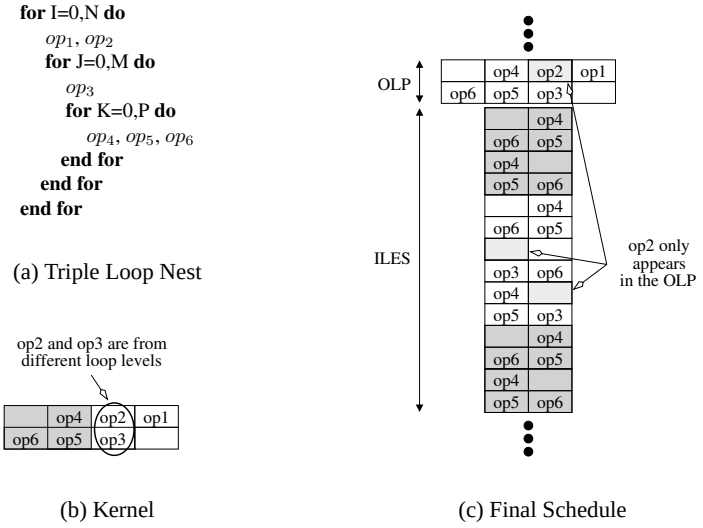


Figure 6: Example of Conditional Emission of Operations

4 Solution

The algorithm framework, shared by the three approaches, is derived from Huff's algorithm [1, 14, 8] and shown in Fig. 7. Starting with II=MII, the scheduler proceeds as follows. The minimum legal scheduling distance ($mindist$) between any two dependent operations is computed. Using the mindist information, the earliest and latest start time, $estart$ and $lstart$ respectively, of each operation is computed. The difference $lstart - estart$, called $slack$, is representative of the freedom of the operations to be scheduled in the kernel. The operations are then scheduled in the kernel one after the other in a heuristic-based operation selection order. If the scheduling of the current operation does not cause any resource conflict, the choice is validated. Otherwise the conflicting operations are ejected. In both cases the $estart$ and $lstart$ values of the other operations are updated accordingly. The process is repeated until all the operations are scheduled. After too many iterations without success, the II of one of the sub-kernels is incremented and the scheduler starts over. When a solution is found, the scheduler enforces sequential constraint and returns successfully. The different steps are detailed in the next subsections.

The proposed approaches and the algorithm framework is correct. As shown in the next subsections, all the scheduling constraints have been respected. Because the algorithm is based on the modulo scheduling, the resource constraints are also honored.

Moreover, when applied to a single loop or to the innermost loop of a loop nest, the method becomes the Huff's modulo scheduling algorithm. Therefore our method subsumes modulo-scheduling as a special case.

```
SSP_SCHEDULER(approach, mii[], priority, ii_incr_method):
 for each loop level i do
     set ii[i] to mii[i]
 end for
 attempts ← 0
 while (attempts < max_attempts) do
     initialize mindist table, MRT
     compute slack values
     placed_ops ← 0
     while (placed_ops < max_placed_ops) do
         choose next operation op according to approach and priority
         if no operation left then
                 enforce sequential constraints
                 return success
         schedule operation op
         placed_ops ← placed_ops + 1
         eject operations violating resource constraints with op
         eject operations violating dependence constraints with op
         eject operations violating innermost level separation limitation with op
         update slack and MRT
     end while
     choose level i to increase II according to ii_incr_method
     ii[i] ← ii[i] + 1
     attempts ← attempts + 1
 end while
 return failure
```

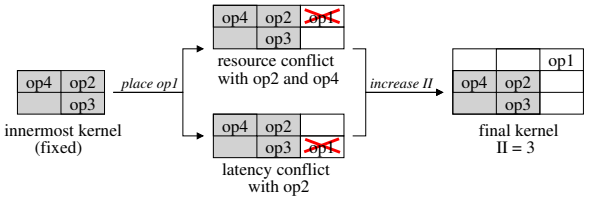
Figure 7: Scheduling Framework

4.1 Scheduling Approaches

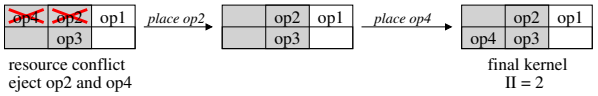
Three different scheduling approaches are proposed. *Flat scheduling* treats the loop nest as if it were "flattened" as a single loop. When backtracking, conflicting operations from all levels can be ejected from the schedule. The main advantage of this approach is its flexibility. Early decisions can always be undone. Such flexibility leads to a larger solution space, and potentially better schedules. On the down side, the search space might become too large and the method too slow to find a solution in time.

With *level-by-level scheduling*, the operations are scheduled in the order of their loop levels, starting from the innermost. Once all the operations of one level are scheduled, the entire schedule becomes a virtual operation from the point of view of the enclosing level. The virtual operation acts as a white box both for dependences and resource usage. A direct property is that a subkernel computed earlier cannot be undone through backtracking. The method has the advantage of being relatively simple and fast. However the early scheduling decisions made in the inner loops might prevent the scheduler from reaching optimal solutions in the outer levels. Figure 8 shows an example where the level-by-level scheduler is forced to increase the initiation interval of the outer kernel to 3 in order to schedule $op1$. On the other hand, a flat scheduler can reschedule inner operations in other scheduling cycles and produce a final kernel with $II = 2$.

The hybrid approach embeds the flat scheduling into a level-by-level framework. The innermost level is scheduled first. Its kernel becomes a virtual operation and the flat scheduling method is used for the other loop levels. The hybrid approach is intuitively a good compromise between level-by-level and



(a) Level-by-Level Scheduling Solution



(b) Flat Scheduling Solution

Figure 8: Advantage of the Flat Approach over the Level-by-Level Approach

flat scheduling, as confirmed by the experimental results. It can find better solutions than the level-by-level method without paying the cost in terms of compile time.

4.2 Enforcement of the SSP Scheduling Constraints

The dependences are enforced through the $mindist$ table. If there is a dependence between two operations op_1 and op_2 with a latency of δ and a distance of k, then we have the dependence constraint:

$$\sigma(op_2) - \sigma(op_1) \geq \delta - k * T + uc(op_1, op_2, k) \tag{1}$$

Because the $mindist$ value is computed once and for all before scheduling the operations, we need to express the right-hand side of the inequality independently of the schedule time of op_1 and op_2. Uc corresponds to the number of empty cycles in the kernel between op_1 and op_2 in the final schedule. A tight upper bound is the number of stages between the two operations, $\lfloor \sigma(op_2)/T \rfloor - \lfloor \sigma(op_1)/T \rfloor + 1 + k$, times the maximum number of empty cycles in a stage, $T - T_n$. and because $x - 1 < \lfloor x \rfloor \leq x$, we have:

$$uc(op_1, op_2, k) \leq \left\lceil \frac{\sigma(op_2) - \sigma(op_1)}{T} + k + 2 \right\rceil * (T - T_n) \tag{2}$$

If Equ. 1 is verified for that upper bound, then it is always verified. By injecting the upper bound of uc from Equ. 2, we obtain after simplification:

$$\sigma(op_2) - \sigma(op_1) \geq \frac{T}{T_n} * (\delta + 2 * T - (k + 2) * T_n) \tag{3}$$

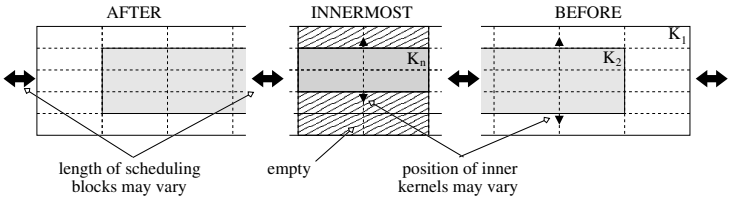


Figure 9: Scheduling Blocks Example

The right-hand side value of Equ. 3 is the value used to initialize $mindist(op_1, op_2)$. By construction, it guarantees that the dependence constraint is always enforced.

The sequential constraint is not enforced during the scheduling process, but as a posteriori transformation once a schedule that satisfies all other constraints has been found. The schedule is then scanned and the sequential constraint checked. If it is not honored between two operations, empty stages are inserted in the schedule until it is honored. The case occurs rarely enough to justify such a technique.

To enforce the innermost level separation limitation without any extra computation cost, the schedule is conceptually split into three *scheduling blocks*: $before$, $innermost$ and $after$. Operations that lexically appear before (after, respectively) the innermost loop are scheduled independently into the 'before' ('after') scheduling block (Fig. 9). Innermost operations are scheduled into the 'innermost' scheduling block. Within each scheduling block, the length of the schedule may vary without breaking the separation limitation and final length of the full schedule is only known at the very end. The modulo resource reservation table is shared between the three blocks.

When an operation is scheduled or when an operation is ejected, the slack of dependent operations must be recomputed. In modulo scheduling, such an update is level-by-level and only operations that need it are updated. Our case is slightly different to take care of the innermost level separation limitation. In implementation, a dummy START and a dummy STOP operations are inserted and pre-scheduled at the beginning and the end of each of the three scheduling blocks. If a dummy operation of one block is ejected and rescheduled, because the distance between operations are related with the length of each block, the slack of every operation within this block has to be recomputed.

4.3 Kernel Integrity

In the flat approach, the II is enforced by scheduling the operations first within the current boundaries of the subkernel. If impossible, the operation is scheduled at some other cycle. The subkernel is then correspondingly moved. All the operations not within the boundaries of the kernel are ejected. In the level-by-level approach, the problem never arises as the subkernels are scheduled separately.

In the level-by-level approach, the integrity of the subkernels is enforced by forbidding the cycles that would cause the subkernel to be truncated. With the flat approach, the problem does not exist because the subkernels are never locked until the very end.

4.4 Operation Selection

The order in which operations are selected for scheduling influences the final shape of the kernel. The following primary scheduling priorities based on the level of operations are used. In *innermost first* order, the operations are scheduled in depth order, starting from the innermost. In *lexical order*, the operations are scheduled in the order they appear in the original source code. The lexical order follows the order of most of the dependencies in the 1-D DDG. In *block lexical* order, the operations are scheduled in the order of scheduling blocks: before-innermost-after. The scheduling order follows the natural order of the execution of the loops and of most dependencies. Finally, in *unsorted* order, the loop level of the operations does not influence the scheduling priority.

After the primary scheduling priorities, three secondary scheduling priorities are used to break ties. With *slack* priority the operations with a smaller slack are scheduled first. Critical operations, i.e. operations that use any critical resource (a resource used 90% of the time or more in the schedule), have their slack divided by two to increase their priority. With *smaller lstart* priority the operations with a smaller latest start time are scheduled first. The priority can be seen as a *top-down* scheduling approach. And with *larger estart* priority the operations with a larger earliest start time are scheduled first. It is a *bottom-up* scheduling approach.

4.5 Operation Scheduling

Once an operation has been chosen as the next candidate for scheduling, a schedule cycle that satisfy the dependence, resource and level separation limitation must be allocated to it. There are several steps to choose such a cycle. First one must identify the range of possible cycles where to schedule the operation. An operation can be scheduled between its $estart$ and $lstart$ values. If the operation is to be scheduled in a top-down approach, find the smallest cycle within that range. Make sure that the operation is scheduled within the II of the loop level it belongs to, and that it only uses available resources. If the operation is to be scheduled in a bottom-up approach, find the latest cycle satisfying the same conditions. If the operation was scheduled but has been ejected since, start the search after the cycle it was scheduled to save time. If no cycle could be found, ignore the other scheduled operations, the availability of resources, and the II of the level of the operation. If the operation is the virtual operation, then schedule it only where it will not be truncated as explained earlier. Operations that may be in conflict with the candidate operation will be ejected and rescheduled later.

4.6 Initiation Interval Increment

When the scheduler fails to find a solution with the current set of II values for each level, one loop level must be chosen and its II incremented by one. Several approaches are proposed. With *lowest slack first*, the average slack of the operations of each level is computed. The II of the loop level with the lowest average slack value is incremented. With *innermost first*, used in conjunction with the innermost first scheduling priority, the levels are considered from the innermost to the outermost. The first level that did not manage to schedule all its operations is chosen to increase the II. Its inner loops do not need to

increase their IIs because they, at some point during the scheduling process, managed to have all their operations scheduled. Finally, with *lexical*, used in conjunction with the lexical scheduling priority, the first loop level in lexical order that did not manage to schedule all its operations is chosen to increase the II.

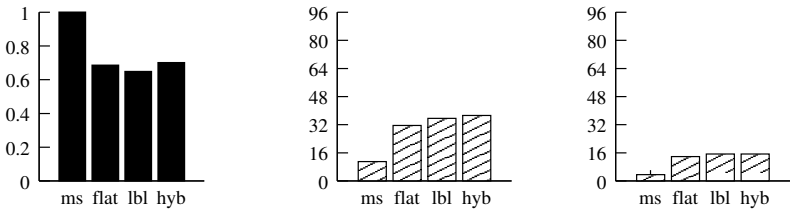
The last two approaches, innermost first and lexical, are dependent on the chosen scheduling priority scheme. They cannot be used for other scheduling priorities.

5 Experiments

We have implemented the proposed approaches and heuristics in the native Open64/ORC2.1 compiler and tested loop nests amenable to SSP from the NAS benchmark suite and the Livermore loop kernels. Every level but the innermost was tried leading on a total of 24 loop nests. Each schedule was run three times on an Itanium workstation.

Results showed that the three approaches produces comparable schedules in terms of execution time and register pressure. However the level-by-level approach is more scalable and less dependent on the heuristics being used. The hybrid approach suffers from the drawbacks of the two other methods.

5.1 Comparison of the Scheduling Approaches



(a) Relative Execution Time (b) INT Register Usage (c) FP Register Usage

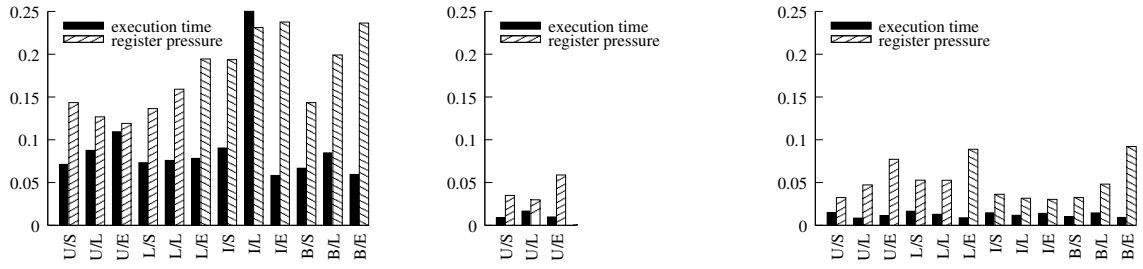
Figure 10: Comparison of the Quality of the Kernels

For each loop level of each loop nest, the best solution of each method was compared to the others. The results are shown in Fig. 10. Overall SSP schedules are 35% faster than MS schedules but uses on average 3.5 times more integer registers. The performance of the three proposed approaches is comparable with a small advantage to the level-by-level approach. Indeed the search space of the flat method is much larger than for the level-by-level approach and a solution cannot always be found in a reasonable amount of time. In that case, the II of one level is increased, leading to sub-optimal solution and a higher execution time. A direct consequence of this is a smaller number of stages and therefore a lower register pressure. When the number of operations is limited, the flat approach is comparable to the level-by-level approach.

The hybrid approach shows performance numbers below expectation. It appears that the method took the worst of the two others. It puts pressure on the scheduler by giving too much priority to the innermost level while being overwhelmed by the size of the solution space when using the flat scheduler.

5.2 Comparison of the Operation Selection Heuristics

Figure 11 compares the results of the different operation selection heuristics for each scheduling approach. The minimum execution time and register pressures were recorded and the relative difference of each heuristic to the minimum was computed for each test case. The average is shown in the figure. The first letter U, L, I, or B stand for the primary selection method: Unsorted, Lexical, Innermost first or Block lexical respectively. The second letter S, E, or L for the secondary method: Slack, largest Estart or smallest Lstart. Level-by-Level scheduling was only tested for the unsorted primary method because all methods are equivalent when a single loop level is scheduled at a time.



(a) Flat (b) Level-by-Level (c) Hybrid

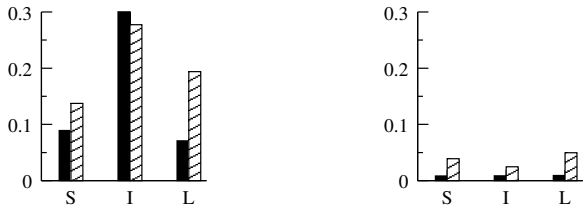
Figure 11: Comparison of the Operation Selection Heuristics

The flat scheduler appears to be highly dependent on the heuristics used. There is no clear best heuristic. On average, each heuristic gives a solution 7.5% slower than the best heuristic for a give loop nest while the integer register pressure is on average 18% higher. The best compromise can be found with B/S and L/S. This high variations is also explained by the size of the solution space. Under the looser constraints used by the flat scheduler, there exist several correct solutions that can be attained.

Under tighter constraints, used by the level-by-level scheduler and indirectly by the hybrid approach, those variations disappear. The choice of the heuristics seems to have no influence on the quality of the computed solution.

5.3 Comparison of the II Increment Heuristics

Figure 12 compares the II increment heuristics for the flat and hybrid schedulers where the heuristics matter. As before, the average relative difference to the minimum for each test case under the same scheduling approach is measured. The three heuristics, lowest Slack, Innermost first, and Lexical, are noted S, I, and L, respectively.



(a) Flat (b) Hybrid

Figure 12: Comparison of the II Increment Heuristics

For the flat scheduler, the lexical order produces the fastest schedules. One may add that the register pressure is higher though. Indeed, because the right level is chosen when the II is incremented, a tighter solution is found. With a lower II, the number of stages is greater and so is the register pressure. Lower execution time comes to the expense of registers.

For the hybrid scheduler, the impact of the II increment heuristics is limited. Indeed, the innermost level, which contains most of the operations, is treated as a special case. Therefore there is not much scheduling pressure left for the other levels (2 to 3 maximum).

6 Related Work

SSP is not the only method to software pipeline loop nests. But it is the first one that has performed a complete and systematic study on each of the subjects: scheduling, register allocation, and code generation [18, 16, 17].

MS techniques were extended to handle loop nests through hierarchical reduction [9, 20, 12], in order to overlap the prolog and the epilog of the inner loops of successive outer loop iterations. Although seemingly similar in idea to the level-by-level approach proposed here, hierarchical reduction software pipelines every loop level of the loop nest starting from the innermost, dependencies and resource usage permitting. The dependence graph needs to be reconstructed each time before scheduling each level, and cache effects are not considered. SSP only tries to software pipeline a single level and to execute its inner loops sequentially. MS has also been combined with prior loop transformations [3, 21, 13].

Finally, there exists other theoretical loop nest software pipelining techniques such as hyperplane scheduling [4]. They however do not consider fine-grain resources such as function units.

7 Conclusion

This paper proposed for the first time kernel generation methods (flat, level-by-level, hybrid) and heuristics for the Single-dimension Software Pipelining framework. The flat approach schedules all the operations of the loop nest simultaneously. The level-by-level approach schedules the operations one level at

a time. Decisions made in deeper levels cannot be undone. The hybrid approach uses the level-by-level approach for the innermost level and the flat approach for the others. We also break the strict level separation limitation introduced in earlier publications [17]. We proved that each technique enforces all the SSP scheduling constraints.

Experiments demonstrated that, although the three approaches shows comparable schedules in terms of execution and register pressure, the level-by-level method is to be preferred because it is more scalable and less dependent on the heuristics than the flat method. The hybrid approach unfortunately inherited the drawbacks of the two other methods and produces slightly slower solutions.

References

[1] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.

[2] Erik R. Altman and Guang R. Gao. Optimal modulo scheduling through enumeration. *Int. J. Parallel Program.*, 26(3):313–344, 1998.

[3] Steve Carr, Chen Ding, and Philip Sweany. Improving software pipelining with unroll-and-jam. In *HICSS '96: Proc. of 29th Hawaii Int. Conf. on System Sciences*, pages 183–192. IEEE Computer Society, 1996.

[4] Alain Darte, Robert Schreiber, B. Ramakrishna Rau, and Frédéric Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Trans. Des. Autom. Electron. Syst.*, 7(1):159–172, 2002.

[5] Alban Douillet and Guang R. Gao. Register pressure in software-pipelined loop nests: Fast computation and impact on architecture design. In *Proc. of the 18th International Workshop on Languages and Compilers for Parallel Computing*, Yorktown, New York, USA, October 2005. Springer-Verlag.

[6] Guang R. Gao, Qi Ning, and V. Dongen. Extending software pipelining techniques for scheduling nested loops. In *Proc. of 6th Int. Workshop on Lang. and Compilers for Parallel Computing*, pages 340–357, 1994.

[7] R. Govindarajan, Erik R. Altman, and Guang R. Gao. A framework for resource-constrained rate-optimal software pipelining. *IEEE Trans. Parallel Distrib. Syst.*, 7(11):1133–1149, 1996.

[8] Richard A. Huff. Lifetime-sensitive modulo scheduling. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 258–267. ACM Press, 1993.

[9] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *PLDI '88: Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 318–328. ACM Press, 1988.

[10] Josep Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT '96: Proc. of the 1996 Conference on Parallel Architectures and Compilation Techniques*, page 80. IEEE Computer Society, 1996.

[11] Soo-Mook Moon and Kemal Ebcioğlu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Trans. Program. Lang. Syst.*, 19(6):853–898, 1997.

[12] Kalyan Muthukumar and Gautam Doshi. Software pipelining of nested loops. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 165–181, London, UK, 2001. Springer-Verlag.

[13] D. Petkov, R. Harr, and S. Amarasinghe. Efficient pipelining of nested loops: unroll-and-squash. In *16th Intl. Parallel and Distributed Processing Symposium (IPDPS '02)*, Fort Lauderdale, FL, April 2002. IEEE.

[14] B. Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proc. of the 27th annual international symposium on Microarchitecture*, pages 63–74. ACM Press, 1994.

[15] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *J. Supercomput.*, 7(1-2):9–50, 1993.

[16] Hongbo Rong, Alban Douillet, and Guang R. Gao. Register allocation for software pipelined multi-dimensional loops. In *PLDI '05: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 154–167, 2005.

[17] Hongbo Rong, Alban Douillet, R. Govindarajan, and Guang R. Gao. Code generation for single-dimension software pipelining of multi-dimensional loops. In *CGO '04: Proc. of the Int. Symposium on Code generation and optimization*, pages 175–186, 2004.

[18] Hongbo Rong, Zhizhong Tang, R. Govindarajan, Alban Douillet, and Guang R. Gao. Single-dimension software pipelining for multi-dimensional loops. In *CGO '04: Proc. of the Int. Symposium on Code generation and optimization*, pages 163–174, 2004.

[19] John Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *PLDI '96*, pages 1–11, New York, NY, USA, 1996. ACM Press.

[20] Jian Wang and Guang R. Gao. Pipelining-dovetailing: A transformation to enhance software pipelining for nested loops. In *CC '96: Proc. of 6th Int. Conf. on Compiler Construction*, pages 1–17. Springer-Verlag, 1996.

[21] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. *Int. J. Parallel Program.*, 26(4):479–503, 1998.

[22] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Register constrained modulo scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 15(5):417–430, 2004.