Multi-Dimensional Kernel Generation for Loop Nest Software Pipelining

Alban Douillet†, Hongbo Rong‡, Guang R. Gao†

† University of Delaware
Newark, DE 19716, USA

‡ Microsoft Corporation
Redmond, WA 98052, USA

Abstract. Single-dimension Software Pipelining (SSP) has been proposed as an effective software pipelining technique for multi-dimensional loops [16]. This paper introduces for the first time the scheduling methods that actually produce the kernel code. Because of the multi-dimensional nature of the problem, the scheduling problem is more complex and challenging than with traditional modulo scheduling. The scheduler must handle multiple subkernels and initiation rates under specific scheduling constraints, while producing a solution that minimizes the execution time of the final schedule.
In this paper three approaches are proposed: the *level-by-level* method, which schedules operations in loop level order, starting from the innermost, and does not let other operations interfere with the already scheduled levels, the *flat* method, which schedules operations from different loop levels with the same priority, and the *hybrid* method, which uses the level-by-level mechanism for the innermost level and the flat solution for the other levels. The methods subsume Huff's modulo scheduling [8] for single loops as a special case. We also break a scheduling constraint introduced in earlier publications and allow for a more compact kernel. The proposed approaches were implemented in the Open64/ORC compiler, and evaluated on loop nests from the Livermore, SPEC200 and NAS benchmarks.

1 Introduction

Software pipelining (SWP) is an important loop scheduling technique that overlaps the execution of consecutive iterations of a loop to explore instruction-level parallelism [9, 8, 13, 1, 7, 10]. Traditionally, it is applied to the innermost loop of a loop nest. The schedule can be extended to outer loops by hierarchical reduction [9, 11, 17]. Loop transformations can be performed to the innermost loop before SWP [2, 18, 12].

Single-dimension Software Pipelining (SSP) [16] 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 that satisfies 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)* and sent as input to the scheduler [16]. The kernel is then computed. If the register pressure is reasonable [5], registers are allocated [14] and the final code is generated [15].

[1] This work was supported in part by the DOD, by DARPA contract No.NBCH30904, by NSF grants No.0103723 and No.0429781, and by DOE grant No.DE-FC02-OIER25503.

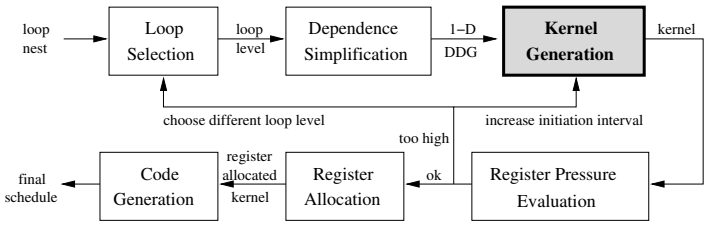


Fig. 1. SSP Compilation Framework

In this paper, we present for the first time algorithms for the *kernel generation* step. The computed kernel must minimize the execution time of the final multi-dimensional schedule. The problem is complex - as it involves the overlapping of operations from several loop levels (dimensions) of a loop nest, a challenge not encountered in traditional modulo scheduling. The kernel is partitioned into *subkernels*, one per loop level and each 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?

Three approaches are proposed and studied. The *level-by-level* approach generates the subkernels in order, starting from the innermost. Once a subkernel has been computed, it cannot be altered. 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 flat method while exploring a large solution space and focusing resources on the innermost loop. The three approaches subsumes Huff's scheduler [8] as a special case when the loop nest is a single loop.

We also break an SSP limitation that forced operations 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 hybrid approach avoids the pitfalls of the two other approaches and produces schedules on average twice faster than modulo-scheduling schedules. Because of its large search space, the flat approach may not reach a good solution fast enough and showed poor results.

The rest of the paper is organized as follows. First, the SSP technique is reviewed. In section 2, the kernel generation problem for SSP is presented. Section 3 explains 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) [16, 15, 14, 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, 17, 11], 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. From the SSP point of view, the loop levels enclosing the selected loop are ignored. Therefore, the selected loop becomes the outermost loop. Within an iteration of the outermost loop, inner loops run sequentially.

Beside being able to software pipeline any loop level and overlap the execution of the prolog and epilog of the inner loops, 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 [16].

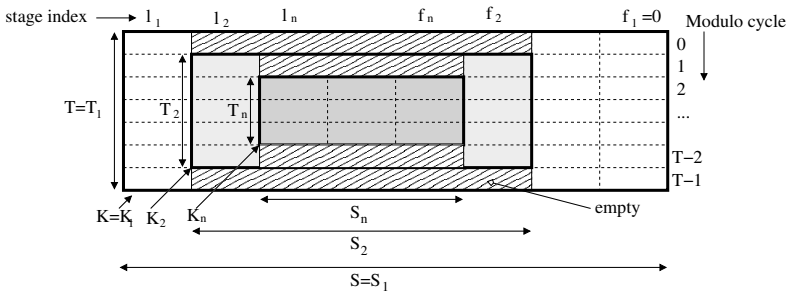


Fig. 2. Generic SSP Kernel

The final SSP schedule is derived from the kernel. Unlike the MS kernel, the SSP kernel has multiple initiation intervals and is 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 are empty because of the kernel nesting constraints presented next (Fig. 2).

2.2 Problem Statement

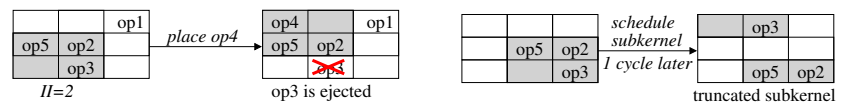
The operations in the kernel must obey the scheduling constraints. A possible conservative definition of those constraints is given below. The modulo property and the resource constraints are identical to those used in MS. However, the dependence constraints now include the number of unused cycles term (uc, defined in Sec. 4), corresponding to the empty stages. The other constraints only exist in SSP. Let $\sigma(op)$ be the schedule time of operation op in the kernel. The constraints are:

– **Modulo Property**: operations are issued every T cycles. T is the initiation interval of the kernel.
– **Resource Constraints**: at any given cycle of the kernel, a hardware resource is not allocated to more than one operation.
– **Dependence Constraints**: $\sigma(op_1) + \delta \leq \sigma(op_2) + k * T - uc(op_1, op_2, k)$ 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 Constraints**: $\sigma(op)+\delta \leq S_p * T_n$ 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 $< d_2, ..., d_n >$.
– **Kernel Nesting Constraints**: operations from different loop levels cannot be scheduled in the same stage and a stage cannot be enclosed between stages of deeper loop levels.

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 when the loop nest is a single loop, the problem is NP-hard [19].

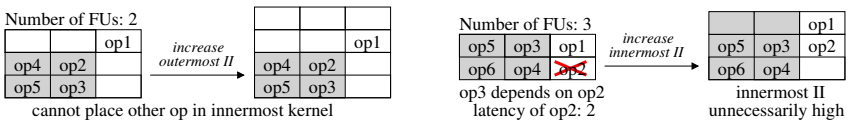
2.3 Issues

To satisfy the constraints mentioned above, several issues need to be solved. First, the kernel is composed of subkernels with different initiation intervals (II) which must be respected during the scheduling process. In Fig. 3(a), the II of the innermost kernel is 2 and the number of functional units (FUs) is also 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 the subkernel is not truncated as shown in Fig. 3(b)



(a) Strict Initiation Rate of Subkernels

(b) Truncation of Subkernels



(c) Useless II Increment Decision

(d) Non-Optimal II Increment Decision

Fig. 3. Kernel Generation Issues

Then, the multiple II feature raises issues of its own. When the scheduler cannot find a solution and the II of one subkernel must be incremented, which subkernel should be chosen? Fig. 3(c) and 3(d) shows examples of inefficient schedules because of poor II increment decisions.

3 Breaking the Kernel Nesting Constraints

The kernel nesting constraints were originally introduced for implementation reasons. We now show that those constraints are unnecessary and can be removed. 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. 4: $op1$, an operation from the outermost loop can now be scheduled in the same stage as operations from the innermost level. Second, because the number of stages may decrease, so may the register pressure.

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.

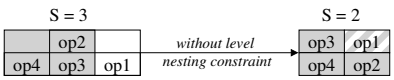


Fig. 4. Removal of the Kernel Nesting Constraints

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

4 Solution

The algorithm framework, shared by the three approaches, is derived from Huff's algorithm [8, 13, 1] and shown in Fig. 5. Starting with the minimum legal II [16] for each loop level, the scheduler proceeds as follows. The minimum legal scheduling distance ($mindist$) between any two dependent operations is computed. Using that 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 scheduling freedom of an operation. The operations are then scheduled in the kernel in a heuristic-based 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 ejected or not-scheduled operations are updated accordingly. The process is repeated until all the operations are scheduled.

```
SSP_SCHEDULER(approach, priority, II_increment, max_II_try, max_op_try):
 for each loop level i do
     set T_i to the minimum legal II for that level
 end for
 for max_II_try attempts do
     initialize mindist table and modulo resource table
     compute slack of operations
     for max_op_try attempts do
         choose next operation op according to approach and priority
         if no operation left then
                 enforce sequential constraints
                 return success
         end if
         schedule operation op
         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 for
     choose level i to increase II according to II_increment
     increment T_i by 1
 end for
 return failure
```

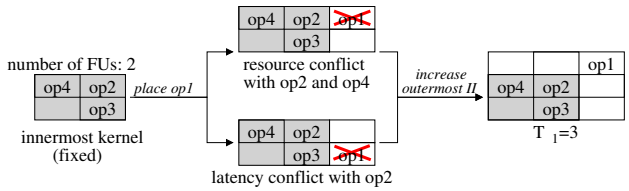
Fig. 5. Scheduling Framework

After too many iterations without success (max_op_try attempts), the II of one subkernel is incremented and the scheduler starts over. After max_II_try II increments, the scheduler gives up. If a solution is found, the scheduler enforces the sequential constraints and returns successfully. The different steps are detailed in the next subsections.

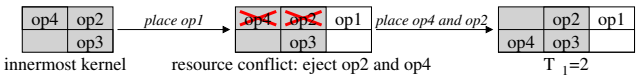
The proposed approaches are correct. As shown in the next subsections, the generated kernel respects all the scheduling constraints. Because the algorithm is based on modulo scheduling, the resource constraints are also honored. Moreover, when applied to a single loop, the method is Huff's algorithm and therefore subsumes modulo scheduling as a special case.

4.1 Scheduling Approaches

Three different scheduling approaches are proposed. 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. Operations within the virtual operation cannot be rescheduled. The method is simple and fast. However, the early scheduling decisions made in the inner loops might prevent the scheduler from reaching more beneficial solutions later in the scheduling process. Fig. 6 shows an example where we assume 2 functional units and a dependence between op_1 and op_2 with a latency



(a) Level-by-Level Scheduling Solution



(b) Flat Scheduling Solution

Fig. 6. Advantage of the Flat Approach over the Level-by-Level Approach

of 2 cycles. The level-by-level scheduler must increase T_1 to 3 in order to schedule $op1$ whereas the flat scheduler can reschedule inner operations to obtain a kernel with $T_1 = 2$.

Flat scheduling considers operations from all loop levels as potential candidates. 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 slow down the scheduler.

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

4.2 Enforcement of the SSP Scheduling Constraints

The dependence constraints, $\sigma(op_2) - \sigma(op_1) \geq \delta - k * T + uc(op_1, op_2, k)$, is enforced through the $mindist$ table. Because the table is generated before scheduling the operations, uc must be expressed independently of the yet unknown schedule time of op_1 and op_2. The following tight upper bound is proposed:

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

The right-hand side of the equation is the $mindist(op_1, op_2)$. By construction [4], the dependence constraints are always enforced.

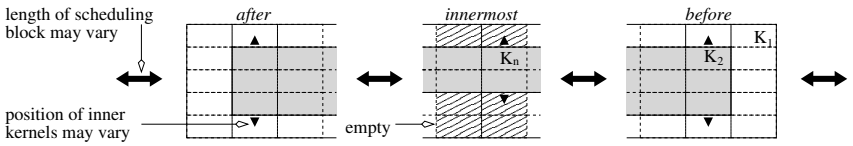


Fig. 7. Scheduling Blocks Example

The sequential constraints are not enforced during the scheduling process, but as a posteriori transformation once a schedule that satisfies the other constraints has been found. At that time, empty stages are inserted in the schedule until the sequential constraints are verified. The need for extra stages 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. 7). 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. Usually, such an update is incremental. However, a dummy START and a dummy STOP operations are used to mark the boundaries of each scheduling block. As the slack is computed relatively to the distance between the START and STOP operations, if a dummy operation of one block is ejected and rescheduled, the slack of every operation within this block has to be recomputed.

4.3 Kernel Integrity

In the level-by-level approach, the subkernels are computed separately and therefore the initiation intervals are always respected. Truncation is avoided by forbidding the subkernels from being scheduled in cycles that would cause it.

In the flat approach, the initiation intervals are enforced by scheduling an operation first within the current boundaries of its subkernel. If impossible, the operation is scheduled at some other cycle. The subkernel is then correspondingly moved. All the operations that are then not within the boundaries of the kernel anymore are ejected. Therefore, subkernels cannot be truncated.

4.4 Operation Selection

The operation selection order is determined by a two-level priority mechanism. The primary priority is based on the loop level of the operation. 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. In *block*

lexical order, the operations are scheduled in the order of scheduling blocks: before-innermost-after. In *unsorted* order, the primary priority is bypassed. Then, 3 secondary 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 resource used at 90% 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. 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

The legal range of schedule cycles for an operation selected for scheduling is defined by $[estart, lstart]$. If the operation is scheduled for the first time, $estart$ is chosen. Otherwise, the next value in the legal range since the last scheduling attempt is chosen. If there is none, the other scheduled operations, the availability of resources, and the II of the level of the operation are ignored and $estart$ is chosen. Conflicts created by the decision will be solved by later ejecting the scheduled operations involved in the conflicts.

4.6 Initiation Interval Increment

Several heuristics are proposed to decide which subkernel should have its II incremented when the scheduler times out. With *lowest slack first*, the average slack of the operations of each level is computed. The loop level with the lowest average slack is selected. With *innermost first*, the first level (from the innermost to the outermost) in which not all the operations have been already scheduled is selected. The heuristic is to be used only with the innermost first scheduling priority, With *lexical*, the first loop level in lexical order in which not all the operations have been already scheduled is selected. The heuristics is to be used only with the lexical scheduling priority.

5 Experiments

The proposed solution was implemented in the Open64/ORC2.1 compiler. 19 loop nests of depth 2 or 3, extracted from the NAS, SPEC2000, and Livermore benchmark suites, were software-pipelined at the outermost level and run on an Itanium2 workstation.

5.1 Comparison of the Scheduling Approaches

The best execution time for each approach was measured (Fig. 8). On average, hybrid and level-by-level schedules are twice faster than MS schedules. In several occasions, the flat solution is slower. Even when given as much as 10 times more attempts to find a solution, the flat scheduler fails and had to increment the initiation intervals, resulting in a slower final schedule. In one case (*liv-5*), the flat schedule was able to perform better than the level-by-level approach. As expected, the hybrid approach combined

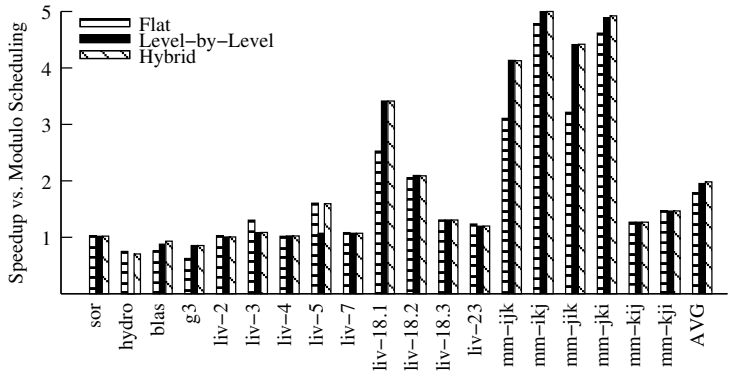


Fig. 8. Execution Time Speedup vs. Modulo Scheduling

the advantages of the two other methods and, for all benchmarks but *liv-3*, produces a kernel with best execution time. Therefore, the hybrid approach should be the method of choice to generate SSP kernels.

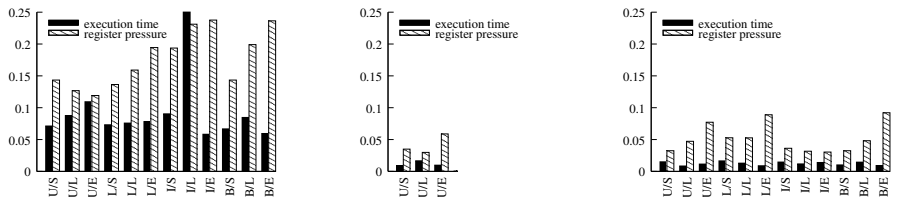
The register pressure was also measured. On average, the register pressure in SSP schedules is 3.5 times higher than with MS schedules, in line with results from previous publications. The hybrid and level-by-level approaches have comparable register pressures, whereas the pressure is lower for the flat approach as the initiation intervals are higher. For *hydro*, the register pressure was too high with the level-by-level approach. It was observed that the register pressure is directly related to the speedup results. The higher the initiation intervals, the lower the register pressure and the execution time of the schedules.

5.2 Comparison of the Heuristics

Fig. 9 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.

For the flat scheduler, the best heuristic is highly dependent on the benchmark being evaluated. On average, each heuristic produces schedules 7.5% slower than the best schedule. Those high variations are explained by the size of the search space. For the two other methods, the choice of the heuristics have little influence on the execution time of the final schedule.the quality of the computed solution.

The II increment heuristics were also compared for the two approaches that use them: flat and hybrid. For the flat scheduler, the slack and lexical order produce the



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

Fig. 9. Comparison of the Operation Selection Heuristics

fastest schedules and are on average below 8% of the best schedule. The innermost order can produce schedules 30% slower. Because the slack order is not dependent on the operation selection heuristic used, it is to be preferred. 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 that has performed a complete and systematic study on each of the subjects: scheduling, register allocation, and code generation [16, 14, 15].

Modulo-scheduling techniques were extended to handle loop nests through hierarchical reduction [9, 17, 11], 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 [2, 18, 12].

Finally, there exists other theoretical loop nest software pipelining techniques such as hyperplane scheduling [3]. Such method not consider fine-grain resources such as function units and registers.

7 Conclusion

This paper proposed for the first time kernel generation methods and heuristics for the Single-dimension Software Pipelining framework and break the kernel nesting constraints introduced in earlier publications [15]. We proved that each technique enforces all the SSP scheduling constraints. Experiments demonstrated that, although the level-by-level and hybrid approaches show comparable schedules in terms of execution and

register pressure, the hybrid method is to be preferred because it outperforms the level-by-level approach in some cases. The flat method was victim of its own large search space and could not find good solutions in a reasonable amount of time and had to settle for kernels with larger initiation intervals. The choice of the heuristics have little influence on the final schedules for the hybrid and level-by-level approach.

References

1. Allan, V.H., Jones, R.B., Lee, R.M., Allan, S.J.: Software pipelining. ACM Comput. Surv. **27**(3) (1995) 367–432
2. Carr, S., Ding, C., Sweany, P.: Improving software pipelining with unroll-and-jam. In: Proc. of HICSS'96, IEEE Computer Society (1996) 183–192
3. Darte, A., Schreiber, R., Rau, B.R., Vivien, F.: Constructing and exploiting linear schedules with prescribed parallelism. ACM Trans. Des. Autom. Electron. Syst. **7**(1) (2002) 159–172
4. Douillet, A.: A Compiler Framework for Loop Nest Software-Pipelining. PhD thesis, University of Delaware, Newark, Delaware, USA (2006)
5. Douillet, A., Gao, G.R.: Register pressure in software-pipelined loop nests: Fast computation and impact on architecture design. In: Proc. of LCPC'05, Springer-Verlag (2005)
6. Gao, G.R., Ning, Q., Dongen, V.: Extending software pipelining techniques for scheduling nested loops. In: Proc. of LCPC'94. (1994) 340–357
7. Govindarajan, R., Altman, E.R., Gao, G.R.: A framework for resource-constrained rate-optimal software pipelining. IEEE Trans. Parallel Distrib. Syst. **7**(11) (1996) 1133–1149
8. Huff, R.A.: Lifetime-sensitive modulo scheduling. In: Proc. of PLDI'93, ACM Press (1993) 258–267
9. Lam, M.: Software pipelining: an effective scheduling technique for vliw machines. In: Proc. of PLDI '88, ACM Press (1988) 318–328
10. Llosa, J.: Swing modulo scheduling: A lifetime-sensitive approach. In: Proc. of PACT'96, IEEE Computer Society (1996) 80
11. Muthukumar, K., Doshi, G.: Software pipelining of nested loops. In: Proc. of CC'01, Springer-Verlag (2001) 165–181
12. Petkov, D., Harr, R., Amarasinghe, S.: Efficient pipelining of nested loops: unroll-and-squash. In: Proc. of IPDPS'02, IEEE (2002)
13. Rau, B.R.: Iterative modulo scheduling: an algorithm for software pipelining loops. In: Proc. of MICRO 27, ACM Press (1994) 63–74
14. Rong, H., Douillet, A., Gao, G.R.: Register allocation for software pipelined multi-dimensional loops. In: Proc. of PLDI'05. (2005) 154–167
15. Rong, H., Douillet, A., Govindarajan, R., Gao, G.R.: Code generation for single-dimension software pipelining of multi-dimensional loops. In: Proc. of CGO'04. (2004) 175–186
16. Rong, H., Tang, Z., Govindarajan, R., Douillet, A., Gao, G.R.: Single-dimension software pipelining for multi-dimensional loops. In: Proc. of CGO'04. (2004) 163–174
17. Wang, J., Gao, G.R.: Pipelining-dovetailing: A transformation to enhance software pipelining for nested loops. In: Proc. of CC '96, Springer-Verlag (1996) 1–17
18. Wolf, M.E., Maydan, D.E., Chen, D.K.: Combining loop transformations considering caches and scheduling. Int. J. Parallel Program. **26**(4) (1998) 479–503
19. Wood, G.: Global optimization of microprograms through modular control constructs. In: Proc. of MICRO 12, IEEE (1979) 1–6