A Comparison of Modulo Scheduling Techniques for Software Pipelining

Peter Pfahler, Georg Piepenbrock¹

Universität-GH Paderborn, Fachbereich 17, D-33095 Paderborn/Germany

Email: peter@uni-paderborn.de

Abstract

Software pipelining is a well-known and effective technique for generating compact loop schedules for instruction level parallel computers. This paper presents the results of an experimental evaluation and comparison of different scheduling algorithms that generate software pipelines. We implemented these algorithms in an uniform retargetable compiler environment that can be instantiated by providing target machine descriptions. This environment and a carefully designed benchmark suite enable us to perform a fair comparison of the implemented techniques. We evaluate well-known non-hierarchical and hierarchical schedulers and a hybrid technique developed in our group. Our analysis indicates that scheduling algorithms based on variations of the “classical” non-hierarchical modulo scheduling technique will probably yield the most effective software pipelines.

Keywords

Software Pipelining, Instruction Level Parallelism, Superscalar Processors, VLIW

1 Introduction

The compilation for high performance computer architectures such as VLIW, superscalar and pipelined processors heavily depends on techniques that make use of the offered instruction level parallelism. Besides local and global scheduling techniques parallelizing code arrangement for loops is one of the major challenges in the design of an optimizing code generation for such processors. Software pipelining is a well-known and effective technique for generating compact loop schedules for instruction level parallel computers. Software pipelining aims at reducing the run time of program loops by overlapping the execution of adjacent iterations. The overlap is achieved by generating a schedule for the loop body in which instructions from different iterations are executed in parallel.

Various heuristic techniques for generating software pipelines have been published in the past decade (see [RaF93] for an overview). All these techniques generate pipelined loop code which shows impressive speed-ups compared to code produced by simple local instruction schedulers. One important and well-known family of techniques for generating software pipelines is *Modulo Scheduling*.

¹supported by the “Deutsche Forschungsgemeinschaft”, DFG project “Übersetzungsmethoden für VLIW-Parallelrechner-Architekturen”

ing introduced by Rau and Glaeser [RaG81] (see Sect. 3).

This paper presents the evaluation of two “classical” modulo scheduling algorithms [RaG81] [Lam88] and compares them with the *RSPS* algorithm developed in our group [Pie95]. We used an uniform evaluation environment where these three schedulers have been implemented. This evaluation environment is machine independent and can be instantiated by providing a target processor specification. It allows us to analyze and compare the pipeline scheduler’s performance for different target processors with more or less constrained resources. Even more interesting results are achieved by investigating how the pipeline algorithms cope with different source loop characteristics. Besides using realistic input programs (Livermore loops, loops from numerical applications) we designed a small suite of synthetic benchmarks. These benchmarks consist of loops of different size and varying dependence characteristics (see Sect. 2.3 for details).

The rest of this paper is organized as follows: In Sect. 2 we describe the evaluation environment. Sect. 3 gives a concise overview over software pipelining in general and presents the techniques that we investigated, including *RSPS*. Sect. 4 presents the comparative evaluation and discusses its results.

2 The Evaluation Environment

The quality of pipeline scheduling techniques can only be compared if these techniques generate code for the same target processor. Furthermore, the effectiveness of pipeline schedules clearly depends on specific target processor and source code characteristics (e.g. the number of parallel functional units, the number and size of recurrence cycles). Consequently, one of the main goals of our work was to make the different software pipelining approaches comparable by evaluating them in an uniform and machine independent environment.

2.1 The Compiler Environment

Our code generators start from an intermediate language representation produced by an ANSI C frontend. This intermediate representation is transformed into the *Operation Dependence Graph (ODG)* by the code selection phase. The ODG is the central data structure for the following code generation tasks (functional unit assignment, register assignment, and scheduling). It consists of nodes representing machine operations and edges that model the data and control flow between machine operations. It is the task of a dependence analysis phase to compute the data dependence relations in a way that leaves the greatest possible freedom for the following scheduling phase. The Omega dependence test [Pug92] is used in our environment.

The functional unit assignment phase maps the operations in the ODG to the target processor’s functional units (FUs). If the functional units are different in instruction set, speed, resource consumption, or connection to storage units (e.g.

register banks), the chosen assignment can have a great influence on the quality of the resulting schedules. If, for instance, the target processor is organized in clusters having individual register banks [CNO87], the FU assignment phase has to compute a mapping which both distributes the operations over the FUs for parallel execution and keeps data-dependent operations locally on the same cluster to avoid inter-cluster copy operations. The functional unit assignment technique from [Ell86] works in this way and yields very good results for acyclic dependence graphs. Our compilers use this technique in an enhanced form which makes it suitable for pipeline scheduling by taking modulo resource constraints (see Sect. 3) into account [Pie95].

The register assignment phase maps symbolic registers to physical register banks and registers of the target processor. Like the assignment of functional units, register assignment in pipeline schedules can have a great influence on the quality of the generated code [Pin93] [HGA92] [GoH88][DGS92]. In our environment we use postpass register assignment, i.e. the registers are assigned after instruction scheduling. This phase ordering does not unnecessarily restrict scheduling freedom by introducing additional dependencies due to register reuse. On the other hand, postpass register assignment can lead to problems caused by overlapping register lifetimes in software pipelines. These problems are addressed in Sect. 4.

2.2 Machine Modeling

Figure 1: The Machine Models

To be able to easily retarget our compiler environment we isolated all machine specific information in a module called *MachSpec*. All other compiler modules access machine-dependent information only via *MachSpec*. *MachSpec* (which is generated from a machine description language) provides the processor specific information required in the code generation phase, i.e. the resource requirement of machine operations, the instruction set of the functional units and structural information such as the reachability of register banks from the different functional units.

We used three different processor models for our evaluation. All models are built of integer units (*I*), floating point units (*F*), a control unit (*C*) for branching, load/store units (*LS*) for memory access, and copy units (*cp*) implementing the data transfer between register banks in cluster-structured machines. Fig. 1 shows the machine models. Processor M1 imitates the structure of the PowerPC 604 superscalar processor. Processors M2 and M3 are two and three cluster architectures. They use the M1 structure inside their clusters with an additional copy unit for data transfer between clusters. The design of these processor models was guided by the instruction frequency distribution in typical realistic input programs. Consequently, an evenly balanced load can be expected on the functional units of these processors.

2.3 Benchmarks

We used three classes of benchmarks for evaluating the pipeline schedulers:

The Livermore Loops as a classical benchmark set turned out to be very well suited for demonstrating impressive speedups due to software pipelining in general. Some of the loops, however, are only of limited significance for comparing the effectiveness of different pipelining algorithms (see Sect. 4).

The synthetic benchmarks set SCC contains 5 source programs. It evaluates the performance of the pipeline schedulers when the number of strongly connected components (*SCC*) grows². The SCCs are cyclic subgraphs of the ODG caused by loop carried data dependencies in the source programs. This benchmark set stresses the SCC characteristics because handling cyclicly dependent nodes turns out to be the major challenge for software pipeline schedulers (see Sect. 3).

The third class of benchmarks contains four numerical source programs (“sor”: sequential overrelaxation, “horner”: polynomial evaluation, “convol”: convolution, “matrix”: matrix multiplication). These programs have been transformed on the source level by SRP (*skewing, reversal, permutation*) transformations to increase parallelism in the innermost loop [WoL91]. Applying such transformations prior to software pipelining can lead to speed-up factors of 2 to 4 compared to pipelining the original loops [SPP94]. On the other hand these transformations lead to more complex loop exit conditions and index expressions, which makes them a challenging benchmark suite for comparing pipeline schedulers.

Since the evaluation described in this paper considers the software pipelining

²The SCC number ranges from 3 (SCC1) to 7 (SCC5).

Figure 2: Overlapping loop iterations (a) and steady state of a software pipelined loop (b)

algorithms in their basic versions, our benchmark programs do not contain loops with internal control flow.

3 Pipeline Scheduling Techniques

Software Pipelining exploits instruction level parallelism by overlapping the execution of consecutive loop iterations. This is achieved by starting the execution of a loop iteration before its predecessor iteration is finished. The constant number of cycles between the start of consecutive iterations is called *initiation interval II*. Fig. 2(a) shows this pipelined execution of a loop.

The task of a software pipelining scheduler is to construct a static loop schedule, which, when executed, leads to this pipelined execution. For this purpose, the scheduler makes use of the fact that after a few cycles the software pipeline executes a repeating pattern of instructions. This repeating pattern is called *steady state* (see Fig. 2(b)) and becomes the body of the generated loop code. The number of instructions in the steady state is equal to the initiation interval II. The instructions that are executed before the steady state is reached are coded in the so-called *prologue*. After leaving the steady state, the remaining iterations are ended in the *epilogue* code. Thus, software pipelining can also be seen as a loop transformation technique transforming the original loop schedule into the prologue, steady state (the new body) and epilogue parts.

Software pipelining is constrained by both machine and source code characteristics. Resource constraints of the target machine clearly limit the possible amount of iteration overlapping II, since the steady state code of length II must be a conflict-free arrangement of all operations contained in the original loop schedule. This fact is used to compute a lower bound for the initiation interval called the *minimum resource II* (ResMII) from the resource requirement of the original loop.

Data dependencies in the source code are the second source for pipelining constraints. Loop carried dependencies, like e.g. “operation a from iteration $i+k$

must not be executed before operation b from iteration i ” clearly limit the possible amount of iteration overlapping. Such dependencies lead to a second lower bound on II , the *minimum precedence II* (RecMII): The II can not be smaller than the sum of delay times on dependence cycles $d(c)$ in the ODG divided by the sum of the iteration distances $p(c)$ on these cycles. The lower bound on the initiation interval MII is defined to be the maximum of RecMII and ResMII.

All modulo scheduling algorithms try to construct loop schedules for a single iteration that can be software pipelined with an initiation interval of II cycles. They start their search for a pipeline schedule with an II value of MII. Compared to schedulers for straight-line code (*local schedulers*) they have to deal with two additional problems: An operation scheduled for cycle c of the loop code will be executed in parallel with all operations from cycles k where $c \bmod II = k \bmod II$. This fact is easily modeled by replacing the local scheduler’s resource reservation table with a table of length II which is indexed modulo II ³. As opposed to local schedulers, the modulo scheduler can fail because of these so-called *modulo resource constraints*.

Dependence cycles lead to the introduction of scheduling ranges for operations. Like local schedulers the modulo schedulers have to obey earliest possible scheduling positions of operations depending on the positions of their predecessors. Additionally, the initiation interval and the scheduling decisions for successors in subsequent iterations determine a latest possible scheduling position (*recurrence constraint*). As opposed to local scheduling, scheduling can now fail if an operation cannot be placed within its legal scheduling range.

In both cases of failure the partial pipeline schedule can not be extended to a valid pipeline schedule for the whole loop. The initiation interval has to increased and the scheduling process restarts. The schedule length of an unpipelined loop iteration serves as a natural upper bound for effective initiation intervals. Asymptotically, the length of the initiation interval, being equal to the number of cycles in the steady state, determines the execution time of the software pipelined loop. Therefore, the II values are generally used to judge and compare different pipeline scheduling techniques.

Flat Modulo Scheduling (FMS)

Flat modulo scheduling [RaG81] has initially been proposed for so-called *Generalized Vector Computations (GVC)*, i.e. loops without recurrences that have a known trip count. The resulting acyclic dependence graph is processed by a list scheduling algorithm which uses typical critical path aspects for prioritizing operations. [RaG81] also describe a straight-forward extension to loops with recurrences. By ignoring backward edges in the dependence graph the list scheduler for GVCs can be reused. Operation priority is again determined by the critical paths in the acyclic graphs. Additionally the scheduler has to obey the recurrence constraint that assures correctly timed inter-iteration dataflow.

³This is where the name “Modulo Scheduling” comes from.

Figure 3: FMS results for the SCC(1..5) benchmarks: II (grey bar) compared to MII (black bar) and length of the unpipelined loop (white bar)

In this basic version FMS does not give priority to time-critical operations on dependence cycles. This clearly leads to sub-optimal initiation intervals if the resources for such operations are occupied by previously scheduled non-critical operations. Fig. 3 demonstrates this effect by comparing the size of initiation interval produced by FMS to the minimum possible initiation interval MII and the maximal useful II (size of the unpipelined loop schedule). FMS is only able to construct optimal or near-optimal results if both the number of cyclicly dependent operations is small and the resource pressure is low, i.e. on machines which offer a sufficient number of parallel functional units.

The results of the original FMS can clearly be improved by extending the list scheduler to take time-critical dependence cycles into account when determining scheduling priorities. This has been proposed by a number of authors [DeT93], [WHS92], [LVA95].

Hierarchical Software Pipelining (HMS)

We consider hierarchical modulo scheduling as proposed by Lam [Lam88]. This algorithm first processes the dependence graph's strongly connected components (SCC) induced by source loop recurrences separately and independently. If this succeeds for one II value for all SCCs this first scheduling phase terminates.

In the second phase an acyclic dependence graph is constructed by reducing the SCCs to hypernodes which are connected to all predecessors and successors of the component (*acyclic condensation*). The hypernodes have complex resource patterns, which correspond to the resource usage in the precomputed SCC schedules. The reduced dependence graph has to be scheduled with the initiation interval used in scheduling the SCCs. Since the reduced graph is acyclic, a simple local scheduler for modulo-constrained resources can be used. The scheduling is successful, if all operations, including the meta operations representing the SCCs, can be integrated in the schedule. If scheduling fails because

Figure 4: HMS results for the SCC(1..5) benchmarks: II (grey bar) compared to MII (black bar) and length of the unpipelined loop (white bar)

of the modulo resource constraint, all scheduling decisions are discarded and the whole process repeats with an increased II.

HMS clearly gives priority to the operations in recurrence cycles by scheduling them independently in its first phase. Furthermore its hypernode reduction strategy can easily be extended to handle loops with internal control flow. The severe drawback of HMS lies in the fact that although compact pipeline schedules for the SCCs can be generated, these schedules lead to complex resource usage patterns for the hypernodes in the second scheduling phase. Thus, HMS frequently fails in its second phase due to resource conflicts leading to sub-optimal II results. Fig. 4 illustrates this.

Resource Sensitive Software Pipelining (RSPS)

Resource Sensitive Pipeline Scheduling has been developed to improve the results of the hierarchical algorithm HMS [Pie95]. RSPS is similar to HMS, the main difference being, that SCCs are scheduled with knowledge of the resource usage in the previously scheduled components. During the first scheduling phase the resource requirement of the components is kept stored in the modulo resource reservation table. The so computed SCC schedules always fit together in the second phase because they cannot cause resource conflicts with each other. The order in which the SCCs are processed is determined by their cycle length to prioritize time critical subgraphs. One essential difference between RSPS and HMS lies in the fact that the resource sensitive approach fixes the modulo positions of the SCC members in its first scheduling phase whereas HMS does so when finally arranging the hypernodes in the second phase.

The reservation table resulting from scheduling the SCCs is reused for arranging the acyclic condensation. Simple (non-SCC) operations are placed in positions where they do not conflict with the previously scheduled components. Arranging the meta operations always succeeds because their resource requirement is

Figure 5: RSPS results for the SCC(1..5) benchmarks: II (grey bar) compared to MII (black bar) and length of the unpipelined loop (white bar)

already reserved in the resource reservation table. The only thing to be done is finding a schedule cycle satisfying the precedence constraints and the modulo positions of the components members. Scheduling a simple operation can of course fail because of the modulo resource constraint.

RSPS can be considered as a hybrid approach between flat and hierarchical modulo scheduling. It is no flat algorithm since it uses precomputed schedules for the SCCs. It is not really an hierarchical technique, since it doesn't process the SCCs independently. The scheduling results in Fig. 5 show that the goal of generating near-optimal pipeline schedules has been reached by the RSPS algorithm.

4 Evaluation

The results for the SCC benchmarks shown in Sect. 3 give a first impression of the performance of the investigated scheduling algorithms. This section provides more benchmarking results and a detailed performance analysis. In addition to the comparison on the base of the II length we will investigate the register requirement of the pipelined loop code generated by the three schedulers.

4.1 Execution time

MATH Benchmark

The SRP transformed numerical programs (c.f. Sect. 2.3) are characterized by large loop bodies with many strongly connected components where some of the SCCs are quite large. Nevertheless the loop's resource usage dominates the lower bound on II ($MII = ResMII$). The results in Fig. 6 show that only the RSPS algorithm computes pipeline schedules with an initiation interval close to or at the lower bound.

Figure 6: Pipelining results for the MATH benchmark.

Livermore Benchmark

We use the first 14 Livermore loops to generate software pipelined loop code for processor M2⁴. The speed-up for the software pipelined loops on M2 ranges between 3.8 (loop 7) and 1.3 (loop 5) compared to the parallel schedule of one loop iteration.

In loops 8, 9, 10 and 13 the lower bound for II is determined by resource usage ($MII = ResMII$). Therefore the results of the scheduling algorithms for these programs are similar to the results obtained for the synthetic benchmarks. In loops 5, 6 and 14 the resource pressure is very small because of long dependence cycles ($MII = RecMII$). With the exception of FMS in loop 14 all scheduling algorithms generate pipeline schedules with an optimal initiation interval. The remaining loops are small with a low resource pressure. Here the FMS algorithm generates the poorest results which once again is clearly due to the fact that cyclicly dependent operations are not given scheduling preference. HMS and RSPS almost always achieve optimal initiation intervals for these loops.

⁴The scheduling results for M1 and M3 are similar and are omitted in this paper.

Figure 7: Pipelining results for the Livermore Loops on M2(Legend see Fig. 6)

4.2 Register requirement

In addition to the length of the initiation interval the register requirement of the generated loop code deserves attention. The register requirement grows with growing concurrency caused by modern processors with more instruction level parallelism and/or effective parallelization techniques, like software pipelining. If the generated schedule requires more registers than available spill code has to be inserted. This can lead to a serious performance degradation due to additional machine cycles and the necessary memory accesses.

Apart from the growing register requirement, one particular problem shows up in software pipelined code: Register lifetime may exceed the length of the initiation interval. Fig 8 illustrates this fact. Register *r1* with a lifetime of 5 cycles exceeds the length of the *II* (3 cycles). This leads to overwriting the *r1* value before reading it in the steady state. Fig. 8 also shows two ways to solve this problem. One can make use of *register queues* to store living register values. If such queues are not supported by the target hardware they can be implemented by register-to-register copies (Fig. 8b). The second approach is called *Modulo variable expansion* [Lam88]. The steady state code is duplicated an appropriate number of times. Each copy uses a different register to propagate the value in question (Fig. 8c). The best way, however, to avoid this problem is to have a scheduler that tries to minimize register lifetimes.

None of the investigated scheduling algorithms considers the minimization of register pressure. Being based on the level-oriented as-soon-as-possible list scheduling algorithm, they all tend to generate longer register lifetimes than e.g. the subtree-oriented strategies used in classical compiler code generation. Nevertheless, there are striking differences.

Figure 8: Register Lifetime exceeds the II (a), register queue (b) and Modulo variable expansion (c)

FMS generated software pipelines have the least register lifetimes that exceed the initiation interval (0.6 register values on the average for the Livermore benchmarks on all three machines). HMS produced 3.5 and RSPS 5.6 exceeding register values. The other benchmark sets led to similar results. The reason for this effect is obvious: HMS must move whole SCC schedules to find a conflict-free scheduling position for the hypernodes. This lengthens the lifetime of all values flowing into the SCC nodes from outside. With RSPS the situation is still worse: Since the modulo-positions of the prescheduled SCC nodes are fixed, the second scheduling phase can only place them at these modulo positions, i.e. every IIth cycle. As our tests show this can cause extremely long register lifetimes.

To summarize our discussion we would like to classify the investigated scheduling techniques by four criteria:

- preferential treatment of recurrence cycles
- independent scheduling or recurrence cycles
- time of decision binding for the relative positions of nodes on dependence cycles (i.e. the construction time of the SCC schedules).
- time of decision binding for modulo positions of nodes on dependence cycles (i.e. the time when a SCC member's modulo position is fixed).

The following table summarizes this classification:

	SCC		Binding	
	preference	independence	relative pos.	modulo pos.
HMS	yes	yes	early	late
RSPS	yes	resource dependence	early	early
FMS	no	no	late	late

For two-phase techniques (HMS, RSSP) *early* simply means the first phase, *late* means the final scheduling phase.

The hierarchical modulo scheduling technique HMS clearly has the advantage of giving preference to dependence cycles. The problem with HMS lies in the fact that scheduling decisions for a SCC are fixed early and independently from other SCCs. This frequently results in HMS not being able to integrate the SCC schedules into a final schedule in its second phase leading to increased initiation intervals.

Resource sensitive pipeline scheduling RSPS avoids these problems by considering resource dependences between the nodes of different SCCs. This approach considerably simplifies SCC integration leading to optimal or near-optimal initiation intervals. This advantage, however, must be paid for in terms of register requirement because the early binding of modulo positions for cyclicly dependent nodes may cause very long register lifetimes. This early binding, on the other hand, is necessary to model the resource interdependence of different SCCs.

Flat modulo scheduling FMS results in reasonable register pressure although register minimization is not regarded in this algorithm. Late binding of all scheduling decisions and the individual arrangement of operations clearly influences register lifetimes positively. The clear disadvantage of FMS in its original form lies in the suboptimal initiation intervals of the generated software pipelines. This is certainly due to not giving preference to nodes on recurrence cycles in the FMS version we used for this evaluation.

Summarizing, non-hierarchical modulo scheduling in an enhanced form that preferences recurrence cycles seems to be the most promising technique. As mentioned in Sect. 3 such enhancements have been proposed by some authors. A recent publication [LVA95] additionally integrates register minimization aspects. Unfortunately, this technique has not yet been integrated into our evaluation environment.

5 Summary and Conclusion

We compared different software pipelining techniques by implementing them in a uniform compiler environment and having them generate code for different instruction parallel machines. In this paper we investigated the non-hierarchical technique FMS [RaG81], the hierarchical technique HMS [Lam88], and the hybrid variant RSSP [Pie95].

The evaluation of our experiments shows that the hierarchical approach has problems scheduling the hypernodes in the acyclic condensation of the data dependence graphs because the schedules for the hypernodes have been produced independently. This problem is especially severe if resource pressure is high, i.e. we have large dependence graphs or small machines. Current instruction level parallel processors provide a quite limited amount of parallel resources. Therefore, pipeline schedulers face both resource and register pressure. Furthermore,

specialized optimization techniques applied before pipeline scheduling tend to further decrease cycle length or remove cycles by moving dependencies out of inner loops. Thus, resource consumption dominates the minimal initiation interval and pipeline schedulers must be able to cope with hard resource constraints. This excludes HMS from the list of promising pipeline scheduling approaches.

The RSSP variant avoids hypernode arrangement problems by taking the resource consumption of previously scheduled components into account while scheduling SCCs. In most cases the RSSP algorithm yields smaller initiation intervals. This advantage, however, must be paid for in terms of register requirement because the early binding of modulo positions for cyclicly dependent nodes may cause prohibitively long register lifetimes.

The basic non-hierarchical pipeline scheduling algorithm suffers from not giving cyclicly dependent operations (SCCs) preference over acyclic operations. Especially if the minimal initiation interval is dominated by the length of cyclic dependence paths, non-critical operations tend to occupy resources required by critical operations on cyclic paths. This prevents FSP from achieving optimal or near-optimal initiation intervals. Register pressure is low in FMS generated software pipelines. This is clearly due to its ability to schedule all operations individually. Furthermore, FMS turns out the most flexible approach which is open for integrating additional optimization criteria. Recent publications, e.g., propose enhanced versions of FMS integrating both preferential treatment of recurrence cycles and register minimization heuristics [LVA95].

Together with the fact that the FMS implementation is considerably simpler than the implementation of the hierarchical technique, non-hierarchical pipeline schedulers turn out to be the most promising candidate for future production quality instruction schedulers.

References

- [CNO87] Colwell, R. P., Nix, R. P., O'Donnel, J. J., Pappworth, D. B. and Rodman, P. K., *A VLIW Architecture for a Trace Scheduling Compiler*, in Proc. 2nd Int'l Conf. on Arch. Support for Progr. Languages and Operating Systems, Oct 87, 180 – 192.
- [DeT93] Dehnert, J. C. and Towle, R. A., *Compiling for the Cydra-5*, Journal of Supercomputing (7) (May 1993), 181–227.
- [DGS92] Düsterwald, E., Gupta, R. and Soffa, M. L., *Register Pipelining: An Integrated Approach to Register Allocation for Scalar and Subscripted Variables*, in Proc. 4th International Conference on Compiler Construction, CC'92, vol. 641, Springer-Verlag, 1992, 192–206.
- [Ell86] Ellis, J. R., *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, MA, 1986.

- [GoH88] Goodman, J. R. and Hsu, W. C., *Code Scheduling and Register Allocation in Large Basic Blocks*, in Proc. International Conference on Supercomputing, ACM, 1988, 442–452.
- [HGA92] Hendren, L. J., Gao, G. R., Altman, E. R. and Mukerji, C., *A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs*, in Proc. 4th International Conference on Compiler Construction, CC'92, vol. 641, Lecture Notes in Computer Science, Springer-Verlag, 1992, 176–191.
- [Lam88] Lam, M., *Software Pipelining: An Effective Scheduling Technique for VLIW Machines*, in Proc. SIGPLAN 88 Conf. on Programming Language Design and Implementation, June 1988, 318 – 328.
- [LVA95] Llosa, J., Valero, M. and Ayguade, E., *Bidirectional Scheduling to Minimize Register Requirements*, in 5th Workshop on Compilers for Parallel Computers, Malaga Spain, 1995.
- [Pie95] Piepenbrock, G., *Methoden des Software-Pipelining für Prozessoren mit Instruktionsparallelität*, PhD Thesis Universität-GH Paderborn (1995).
- [Pin93] Pinter, S. S., *Register Allocation with Instruction Scheduling: a New Approach*, in Proc SIGPLAN 93 Conference on Programming Language Design and Implementation, 1993, 248–257.
- [Pug92] Pugh, W., *A Practical Algorithm for Exact Array Dependence Analysis*, Communications of the ACM 35-8 (1992), 102–114.
- [RaG81] Rau, B. R. and Glaeser, C. D., *Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing*, in Proc. 14th Annual Microprogramming Workshop, 1981, 183–198.
- [RaF93] Rau, B. R. and Fisher, J. A., *Instruction-Level Parallel Processing: History, Overview and Perspective*, Journal of Supercomputing 7 (May 1993), 9–50.
- [SPP94] Slowik, A., Pfahler, P. and Piepenbrock, G., *Compiling Nested Loops for Limited Connectivity VLIWs*, in Proc. 5th International Conference on Compiler Construction, CC'94, vol. 786, Lecture Notes in Computer Science, Springer-Verlag, 1994, 143–157.
- [WHS92] Warter, N. J., Haab, G. E., Subramanian, K. and Bockhaus, J. W., *Enhanced Modulo Scheduling for Loops with Conditional Branches*, Proc. 25th Intern. Symposium on Microarchitecture, Portland (1992).
- [WoL91] Wolf, M. E. and Lam, M. S., *A Loop Transformation Theory and an Algorithm to Maximize Parallelism*, Transactions on Parallel and Distributed Systems 2/4 (Oct. 1991), 452–471.