Compilation for Instruction Parallel Processors

U. Kastens
University of Paderborn, FRG

1 Introduction

Performance of computers is drastically increased by hardware facilities operating in parallel. In order to take advantage of that power, parallel tasks have to be identified, scheduled and executed on different levels of software, depending on the kind of parallelism which is addressed. This paper concentrates on parallelism on instruction level which has to be planned in the code generation phase of compilers. An overview is given over those code generation problems and techniques of their solutions.

From the view of hardware we distinguish concepts of parallelism on different architecture levels: Distributed processors with local memories are coupled via asynchronous message passing. The processors of a multiprocessor machine cooperate asynchronously via a shared memory. Within a single processor several instructions may be executed in parallel by an instruction pipeline or by several functional units both synchronized by the processor's clock. This parallelism on instruction level is considered in this paper. Vector processors (not considered here) execute single operations simultaneously on an array of data elements. Finally on register transfer level single processor components are controlled in parallel by horizontal micro instructions or hardware logic.

From the view of program execution different kinds of parallelism are observed: The operating system schedules processes to several processors which may be distributed. A single program task may be composed of several asynchronous processes, which are explicitly synchronized. This level is usually called *coarse grained parallelism.* Any program allows operations to be executed in parallel, even if it is specified in a sequential style. Such *fine grained parallelism* can be analyzed by compilers, in order to take advantage of instruction level parallelism in the processor. (A functional program specification exhibits the parallelism more directly.) This paper is devoted to problems of this kind of translation. *Data parallelism* analyzed for the translation to vector machine code is not considered here. Classifications of parallel architectures like [Dun90] often ignore the instruction level parallelism, since it is completely transparent for high level programming. On the other hand a parallelizing compiler is absolutely necessary to achieve the potential performance of such a processor.

Parallelism on the level of micro programs is a rather old technique. Hence in their development and translation similar problems as considered here have to be solved. That is why many of the compiler techniques mentioned here originate from the area of micro programming. As soon as processor design lifts control of internal parallelism to the

instruction level, translation and scheduling tasks are shifted from hardware design and micro programming into compiler code generation tasks. As an example the MIPS processor [HJG82] requires its instruction sequences being properly scheduled for pipelining to be executed correctly. On the other hand processors like the RISC [PaS82] incorporate hardware facilities to control and possibly correct the scheduled instruction sequence. Here hardware effort is invested for the solutions of problems which could be better solved in the compiler. The situation is in general comparable with the tendency going from CISC class processors to those of RISC class: Implementation decisions are moved from the processor hardware to the compiler software where deeper knowledge on the particular program is available. This reduction of hardware complexity can be used to increase speed and/or the number of functional units operating in parallel.

Besides pipelining the second concept of instruction level parallelism occurs in processors which have several functional units synchronously executing operations. Such processors range from a few units in processors like the i860 [Mar89] or RS/6000 [IBM90] to so called VLIW (very long instruction word) processors like the TRACE [Fis81], ESL or Cydra [RGP82] with more than 20 units. Again techniques for instruction scheduling are required to generate correct and efficient code for processors of this class. The degree of useful fine grained parallelism very much depends on the program characteristics. Broader investigations like that in [CGL89] are still required.

In this paper both kinds of instruction level parallelism, pipelining and multiple functional units are considered together from the view of compiler synthesis. Section 2 introduces a simplified abstract model for that parallelism. The problems of instruction scheduling, allocation of functional units and of resources are discussed in Section 3, with specific emphasis on their interrelation. While Section 3 artificially simplifies the problem by application to basic blocks of the source program, in Section 4 an overview over some techniques applied to larger control flow structures (including loops) is given. The presentation here should be understood as an introduction to this class of compilation problems, being rather incomplete since the topics are subject of actual research at many places.

2 Model

In this section we introduce a roughly simplified model for the execution of instructions by processors which internally operate in parallel. We concentrate on the two concepts of pipelining and multiple functional units. The model does not exhibit specific processor components. It is restricted to the consecutive and parallel execution of operations and constraints on the composition of the code. This is the view of a compiler which ensures that the code is semantically correct and takes best advantage of the processors parallelism. Of course the design of compiler code generation for a particular processor needs much more specific information on the hardware than this model provides. Here it is sufficient for an overview on the structure of the problems and an idea of the algorithms for their solution.

In our model the machine code is a sequence of *instructions*. It is fed into the processor which initiates execution of the next instruction on each processor *cycle*. Execution of a branch may specify the next instruction explicitly. Execution of a straight line instruction sequence by a *sequential processor p* is depicted in Fig. 1. The processor is shown as a window being moved over the instruction sequence along the time axis. In the j-th cycle it contains the instruction I_j which is currently being executed. Since each instruction is a single elementary operation of a (possibly RISC-like) instruction set there is no parallelism observable on code level for this processor.

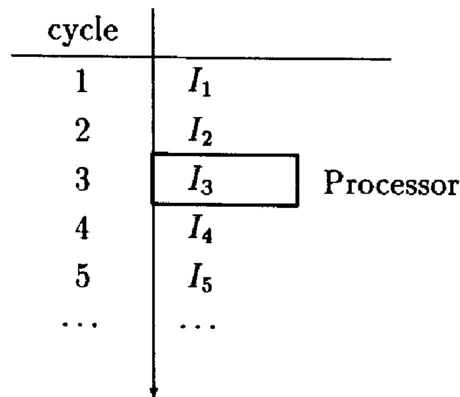

Fig. 1: Sequential Processor

This diagram is easily extended modeling a processor with an instruction *pipeline*. It consists of n stages operating in parallel on n consecutive instructions, e.g. decoding the instruction, fetching the operands, and computing and storing the results. In Fig. 2 the processor window is extended to contain 3 instructions, one in each stage. When moving the window an instruction enters the pipeline in stage 0 and leaves it completely executed after 3 cycles. Fig. 2 shows the situation in cycle 4 when the instructions I_2, I_3, I_4 are in the 3-stage pipeline. This model assumes that processing of any instruction takes one cycle in each stage.

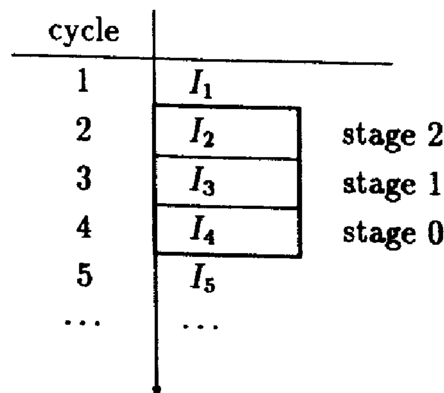

Fig. 2: Pipelining Processor

Another kind of parallelism is observed if the processor consists of *multiple functional units* (FUs). Each functional unit executes one operation in each cycle. The operations again are elements of an instruction set, which may differ for the specific FUs (e.g. some FUs for integer operations others for floating point operations). From the view of the code now each instruction is a tuple of m operations which are allocated to each of the m FUs and executed in parallel. Fig. 3 shows a processor with 3 FUs executing instruction I_3 which consists of the operations $I_{3.1}, I_{3.2}, I_{3.3}$. At this point it should be mentioned that we don't consider here how these long instructions are represented in memory, fetched from there, and how their operations are directed to the FUs.

cycle	FU_1	FU_2	FU_2
1	$I_{1.1}$	$I_{1.2}$	$I_{1.3}$
2	$I_{2.1}$	$I_{2.2}$	$I_{2.3}$
3	$I_{3.1}$	$I_{3.2}$	$I_{3.3}$
4	$I_{4.1}$	$I_{4.2}$	$I_{4.3}$
5	$I_{5.1}$	$I_{5.2}$	$I_{5.3}$
⋮		...	

Fig. 3: Multiple Functional Units

These two concepts of parallelism of course can be combined: If each of the m FUs is organized as a pipeline with n stages we get the situation of Fig. 4 where in each cycle the processor is executing in parallel n instructions in different stages each containing m operations.

cycle	FU_1	FU_2	FU_2
1	$I_{1.1}$	$I_{1.2}$	$I_{1.3}$
2	$I_{2.1}$	$I_{2.2}$	$I_{2.3}$
3	$I_{3.1}$	$I_{3.2}$	$I_{3.3}$
4	$I_{4.1}$	$I_{4.2}$	$I_{4.3}$
5	$I_{5.1}$	$I_{5.2}$	$I_{5.3}$
⋮		...	

Fig. 4: Pipelined Multiple Functional Units

In reality we have FUs which differ in the number of cycles needed to complete an operation (e.g. a floating point FU may take longer than an integer FU, or one FU dedicated

for multiplication may take longer than one for simpler operations only). Hence the operations on different FUs have a different *delay*, i.e. the number of cycles needed for completion. In Fig. 5 the FUs have a delay of 2, 4, and 3 cycles represented by the shape of the processor window.

cycle	FU_1	FU_2	FU_2
1	$I_{1.1}$	$I_{1.2}$	$I_{1.3}$
2	$I_{2.1}$	$I_{2.2}$	$I_{2.3}$
3	$I_{3.1}$	$I_{3.2}$	$I_{3.3}$
4	$I_{4.1}$	$I_{4.2}$	$I_{4.3}$
5	$I_{5.1}$	$I_{5.2}$	$I_{5.3}$
...		...	

Fig. 5: Pipelined Multiple Functional Units with different delays

Finally we may have the situation that certain FUs cannot initiate a new operation on each cycle. There has to be a *latency* of k cycles until the next operation can be fed into the FU. The processor window in Fig. 6 has the additional constraint that at most one operation may be within the shaded area which covers k cycles. If the FU is pipelined its stages usually take k cycles each. As a consequence the operation sequence for such a FU must have gaps at least of length $k-1$ filled with no operations.

cycle	FU_1	FU_2	FU_2
1	$I_{1.1}$	$I_{1.2}$	$I_{1.3}$
2	$I_{2.1}$		$I_{2.3}$
3	$I_{3.1}$	$I_{3.2}$	$I_{3.3}$
4	$I_{4.1}$		$I_{4.3}$
5	$I_{5.1}$	$I_{5.2}$	$I_{5.3}$
...	...		

Fig. 6: Functional Unit with latency 2

This model of a processor window moving over the instruction sequence will be used in the discussion of instruction scheduling in Sect. 2. We then say two operations O_1 and O_2 scheduled in the instruction sequence at positions $I_{m.h}$ and $I_{n.g}$, with $m \leq n$, *meet in*

the window, if $n - m < s_h$, where s_h is the number of pipeline stages of FU_h. In that case execution of O_1 and O_2 will overlap.

Branching. In context of parallelism execution of branch instructions causes specific effects. If a conditional branch is entered into a pipelined processor (single FU), the branch decision may be taken in stage $k > 0$ (e.g. the operand read stage or the execution stage). Hence at that point k more operations from the instruction sequence have already entered the pipeline. They will be executed independent of the branch decision. We have a *delayed branch* with a delay of k instructions. A code generator would try to move k instructions behind the branch which do not depend on the branch decision. In Fig. 7 I_3 may be a branch, decided in stage 2 with a delay of 2. In the next cycle either I_6 or I'_6 is fed into the pipeline. Usually it takes additional time for fetching the next instruction if the branch is taken. In that case the pipeline will be drained for some cycles, whereas no gaps occur if control follows the instruction sequence. The compiler may take advantage from this behavior by arranging branches such that the preferred direction is the faster one.

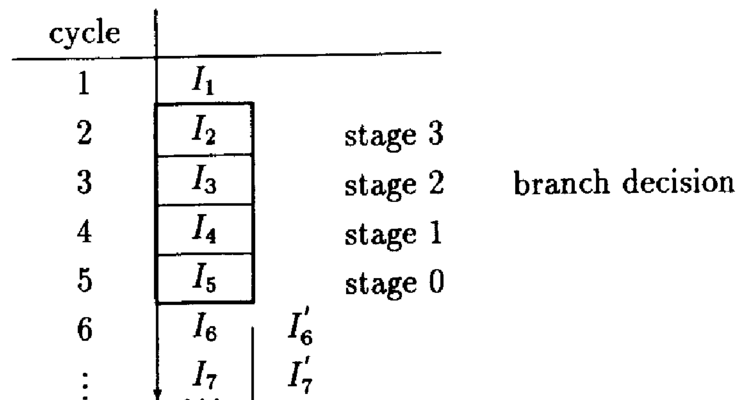

Fig. 7: Delayed Branch

A processor with several FUs may evaluate more than one branch in parallel. In this case a precedence (either defined over the FUs or over the associated condition code registers) determines which branch will be effective. The instruction then has the semantics of a branch cascade executed in parallel. If the precedence relation is predefined by the hardware, its use imposes additional constraints on allocation of operations to FUs and/or of values to condition code registers. The effect is combined with that of delayed branches if the FUs operate in pipelining mode.

The concept of multi-way branches is further extended in the experimental machine described in [Ebc88]. There a single instruction forms a binary tree. Its inner nodes are branch decisions, its leafs are labels. Each edge carries several operations. The branch conditions are checked on instruction fetch. Then only those operations on the thus determined path in the tree are directed to the FUs.

The Cydra machine [RGP82] incorporates an additional control feature: Any operation of an instruction may be disabled for execution depending on the contents of a condition code. That feature is used for shielding operations during the prologue and epilogue of compacted loops, or for composition of code from different basic blocks in a straight line instruction sequence. It of course leaves the disabled FUs inactive at execution time.

3 Instruction Scheduling and Allocation

The code generation phase of a compiler produces an instruction sequence from a representation of the analyzed source program. Certain optimizing transformations, which are either generally applicable or are specific for the kind of processor parallelism, may be applied to that representation. Then the code generator has to schedule the operations, allocate them to functional units, and allocate intermediate values to registers, and other resources to each operation.

The objectives of code generation are to preserve the semantics of the program and to take best advantage of the processor's parallelism. The latter goal is approached - like in sequential code generation - by producing an instruction sequence as short as possible, especially for frequently executed program parts, i.e. inner loops. For that purpose the program representation has to exhibit the *fine grained parallelism* of the source program. So each basic block is represented by a DAG, where the nodes stand for operations and the edges for values read or computed by the operations. More information on the context of a basic block is computed by data flow analysis and added to the representation in order to improve the code in a larger context, see Sect. 4.

The processor's parallelism imposes additional constraints onto code generation. Certain resources cannot be used by several operations which are executed in parallel. The model introduced in the previous section gives the base to exhibit those constraints: Any two operations, which meet within the processor window at the same time, may be in conflict.

In case of a pipelining processor it may be necessary to insert empty operations in order to separate operations which would be in conflict otherwise. Such a situation has to be avoided as far as possible. In case of multiple functional units conflicts have to be avoided by leaving empty some fields of instruction tuples. Again the objective is to produce the code as compact as possible.

In the following we give short specifications of the problems: scheduling, functional unit allocation, and resource allocation. They are separated here for ease of presentation, although they are highly interrelated.

Instruction Scheduling. The operations of the program representation are arranged into a sequence of sets of operations. Each set will be mapped to an instruction tuple by solving FU allocation. Hence the cardinality of the sets must not exceed the number of FUs. If there is only one FU that mapping is trivial. The following constraints have to

be obeyed by the scheduling decisions:

1. The resulting schedule considered as a partial order has to cover the partial order of the program DAG. Furthermore in case of pipelining an operation O_2 depending on O_1 must not be scheduled earlier than d instructions after O_1, where d is the distance of the read and write stages in the pipeline (possibly shortened by hardware bypasses).
2. Two Operations O_1 and O_2 scheduled at $I_{m.g}$ and $I_{n.h}$ must not use the same resources exclusively if they meet in the processor window. A predicate conflict (O_1, m, g, O_2, n, h) may check that during the process of scheduling. In general these constraints can be checked completely only if FUs and operation resources are already allocated.
3. The schedule must preserve the lifetime of values stored in resources like registers, memory, or condition codes. An operation must not write into a resource before the last operation has read the value stored there. Of course these constraints can be considered only if storing resources are allocated before or while scheduling. As long as intermediate values are represented symbolically (by edges in the DAG) no conflict arises.

The complexity of this instruction scheduling problem and the structure of solution algorithms depend on the constraints to be considered. In the scheduling theory problems are characterized by the following properties [Bru81]:

- n Jobs (operations)
- m processors (FUs, $m = 1$ for a single FU pipelined processor)
- a dependency relation (program DAG)
- different job durations d (operations may be dedicated to FUs with different numbers of pipeline stages)
- minimization of the latest job completion (length of instruction sequence).

This problem classification does not cover the constraints (2) and (3) above. The classical resource constraint scheduling problems cover (2) instead of (1), i.e. they neither consider a precedence relation nor dependencies from storing resources. Hence any of the classical scheduling algorithm has to be augmented by additional constraints heuristically, in order to yield tractable but suboptimal algorithms.

The well-known quadratic algorithm of Hu [Hu61] computes optimal solutions if the DAGs are restricted to trees and all operations require unit time. It is often applied in the general case, augmented by heuristic criteria and additional constraints yielding suboptimal

solutions. It basically labels the operation nodes by numbers l (i.e. their level above the bottom nodes). If x is the shortest completion time, assuming arbitrary many FUs, then for each operation $x - l$ is the latest start time of the operation such that x can be reached. The labeling can easily be computed by a walk backwards through the DAG. It exhibits the shortest reachable schedule length, as well as the maximal achievable degree of parallelism, and critical paths in the DAG. The second phase fills the instructions in forward order with a preference for higher labeled operations. This is the place where additional constraints are inserted, and heuristics may be applied. E.g. in [GiM86] operations are preferred if they may cause conflicts with a greater number of their successors. The algorithm of Coffman and Graham [Cof76] uses a refined labeling. It has been proven to be optimal for DAGs and 2 FUs.

Such scheduling algorithms may be applied before, after, or interleaved with allocation of FUs and storing resources. If it is done before, then allocation might cause conflicts of class (3) above. In such a case the initial schedule has to be modified (in general lengthened) on the base of only local information. If scheduling is applied after allocation, then the choices for parallel execution may be drastically reduced; e.g. two independent expressions may not be computed in parallel because the same registers are used. Furthermore scheduling algorithms like the above, have to be augmented by often complicated lookahead computation, in order to avoid deadlocks due to mutually exclusive use of resources, e.g. updating "safe paths" in the postpass pipeline scheduling of [HeG83].

Functional Unit Allocation. Each operation of the DAG is allocated to a FU which will execute it. The "column" within the instruction sequence is determined, while scheduling fixes the "row" for the operation. This problem is trivially solved, if each operation can be executed on exactly one FU (e.g. one integer one floating point and one load/store unit), or if there is only one (pipelined) FU at all. If there are several FUs of the same class choices have to be made, which are interrelated with scheduling: The operations of one instruction have to be executed in parallel by different FUs. If the FUs of each class are completely equivalent, e.g. with respect of access of registers, FU allocation can be done trivially after scheduling without any penalty on the schedule length. In that case all registers and FUs have to be completely connected by expensive hardware structures, like in the Cydra machine [RGP82].
More sophisticated solutions are required for FUs with local register sets. The TRACE machines [Fis81] have this property. Here explicit register transfer operations move values from a register of one FU to that of another FU. Furthermore the operation takes more time and needs a different bus, if the FUs are in different "clusters". Obviously these constraints cause another interrelationship between scheduling, FU and register allocation. Ellis suggests the following heuristical approach for his Bulldog compiler [Ell86]: The FUs are allocated during a first scheduling phase, where no other constraints are considered. The schedule itself is discarded. Then a second scheduling phase with integrated register allocation computes the final schedule. Here the costs for additional register transfer operations can be taken into account. Another approach for FU allocation before scheduling could be based on a "closeness" relation over the program DAG described for hardware synthesis in [McF83].

Resource Allocation. Two categories of resources have to be distinguished: *storing resources* e.g. registers, which store values for being read in subsequent operations and *execution resources* like busses or memory ports, which are occupied exclusively by an operation for the duration of its execution.

The execution resources are allocated to single operations. Scheduling constraints (no. 2 above) guarantee that conflicts are avoided. A limited number of equivalent instances within a resource class and the use of resources from more than one class raises the complexity of the scheduling algorithm. This category of resources is usually considered in scheduling theory; algorithms, strategies, and heuristics are investigated. Due to the short duration of its access (a single operation or some of its stages) there usually isn't an allocation problem.

Storing resources realize data flow between operations. They are allocated to symbolic values. In the DAG a symbolic value is represented by a set of edges starting from the same node. The symbolic values can be understood as instances of resource classes with unlimited cardinality. Resource allocation maps them to real instances of limited classes. Reuse of one resource r for different values v_1, v_2 requires that in the resulting instruction sequence the lifetimes of v_1 and v_2 (from the definition to its last use) are disjoint. Hence the decisions made for allocation are interrelated with those of scheduling.

In generation of sequential code this register allocation problem is well understood. There are methods and algorithms for trees [AhJ76,SeU70], DAGs and basic blocks [Bel66] and control flow graphs by graph colouring [Cha82]. They all reduce the amount of spill code needed if not enough registers are available. Here also the interrelationship between allocation and evaluation order causes high complexity of the problem. Integrated optimal solutions are tractable only for trees (if restrictions on the register classes hold). The solution for more general structures either fix the evaluation order first or apply suboptimal heuristics.

The situation for generation of parallel code is comparable: If the decisions of scheduling, allocation of storing resources and of functional units are serialized, early decisions should be triggered by heuristic preferences on later constraints. Hence a schedule can be computed (after FU allocation as suggested above) such that the maximal resource requirements are minimized. For a DAG it could be expressed by a graph layout with minimal cut width [Pfa88]; For allocation then methods like colouring may be applied. (Techniques like these are used in the solution of comparable problems for high level hardware synthesis [Pfa88].) There are several approaches which start with resource allocation [GiM86,HeG83]. In this case additional restrictions are imposed on the scheduling decisions (no. 3 above). They may cause serialization which could often be avoided if preferences of the data flow structure influence the allocation decision. The interrelationship of these decision structure still needs deeper investigation.

4 Global Techniques

In sequential code generation improving transformations are more effective if they are applied on the base of data flow information computed for the whole flow graph of a procedure or the program rather than considering each basic block separately. That global view is even more important for generation of parallel code. Usually the degree of fine grained parallelism found in single basic blocks does not provide enough operations for simultaneous execution. It may be sufficient for a single pipelined FU, but a larger number of FUs can't be kept busy by only local techniques. Furthermore special techniques have to be applied to loops where improvements are most effective in runtime savings.

In the following some methods are presented which increase the degree of parallelism either on the level of the abstract program structure, or between adjacent basic blocks, or within loops. Many of the ideas and techniques originate from the area of micro code optimization. Horizontal micro code instructions control several hardware components operating in parallel. Hence the optimization problems are similar to those considered here on instruction level.

In the following the DAG representation for basic blocks is assumed to be embedded in a global flow graph of block nodes linked according to the static control structure of the program. Furthermore data flow analysis for reaching definitions and live variables should have been applied. Its results (represented by use-def and def-use chains) describe the data flow via variables across block boundaries [ASU86,Kas90].

If definitions and uses of certain variables can not be determined at compile time due to aliasing, their store and load operation have to remain in the original order relative to each other. The program graph is augmented by specific sequencing dependencies, which further restricts scheduling. Hence an effective aliasing analysis for pointers, parameters, and array indices is required, especially since memory access is often a bottleneck for highly parallel processors. The methods discussed below are originally presented on a level where allocation of values to registers is already done. In that case that data flow information is required for registers instead of variables. Since the interrelations between allocation and scheduling decisions are treated in the previous section, this aspect is not discussed here again.

Generalized Data Dependency Graph (GDDG). In [IKI89] a technique for micro code optimization is presented which increases the freedom for scheduling decisions in the context of global program flow by transformations of the data flow graph described above. It is based on the following idea: The range where a specific operation O can be scheduled is bounded by those operations which compute the inputs for O and those which use its output. These points (given by the use-def and def-use information) lie not necessarily in the same block to which O initially belongs. Hence the operation can be moved out of the block if additional constraints on the use of variables in adjacent blocks hold. In the GDDG these dependencies are introduced for operations outside of blocks. A subsequent scheduling phase is applied to each basic block. It first computes a schedule comprising

those operations, which must be executed in that block. Then without increasing the length of the instruction sequence, free positions are filled with operations which may be executed there according to the GDDG relations. The authors suggest some heuristics based on the branch structure of the flow graphs to determine the order in which the blocks are scheduled and may-operations are chosen.

Trace Scheduling. The trace method was initially presented by Fischer in [Fis81], in the context of micro code compaction, and then transferred to VLIW compilation. It is based on the following idea: Instruction scheduling is applied to larger paths through the program graph rather than to single basic blocks. Such a path is chosen by certain heuristics. The operations of all consecutive blocks on a path are scheduled into a single instruction sequence, ignoring outgoing branches and labels where control flow joins within the path. The large number of operations from several blocks allows an effective scheduling which yields rather compact code for that path. Within that instruction sequence operations may be moved - even across branches and joins. Hence additional compensation code is required in the adjacent blocks: Operations which are moved down (up) over a branch (join) must be copied to the branch target (origin of the join). They increase the number of operations in paths considered later and thus reduce its code quality. Hence paths which are expected to be executed frequently should be scheduled early. Loop bodies are considered separately before their enclosing path. Their resulting instruction sequence is then treated as a meta instruction.

Under certain conditions large amounts of compensation code are copied, theoretically leading to exponentially increased code size in the worst case. The refined trace method of Tree Compaction [LaA83] avoids this effect: Paths are chosen such that they don't contain a join of control flow.

It is important for the trace method in general to make a good choice on execution frequency at compile time, since the code quality differs for the paths. Measurements from sample runs may be fed back into compilation. A complete compiler design including Trace Scheduling is described in [Ell86].

Loop Scheduling. Special methods are applied for optimization of loop bodies. Whereas conventional loop optimization tries to reduce the number of operations in the loop body as far as possible, here operations of subsequent iterations are merged in order to increase the degree of parallelism. This situation is similar to optimization for vector machines where parallelism of data rather than operations has to be analyzed. For both targets especially data dependencies between values of different iterations and array indices based on induction variables have to be analyzed (see e.g. [Ken81]). In fact some of the techniques discussed below were originally developed for vectorizing compilers.

A rather old technique is *loop unrolling*. The loop body is copied several times. Loop control can be eliminated between adjacent copies yielding larger basic blocks. Then operations from different iterations may be scheduled in parallel. This method is refined in several ways: In [Nic87] conditions for the degree of unrolling are developed, considering nested loops, too.

Several techniques introduce pipelining as an abstract concept to loop execution, see Fig. 8: The central part of the loop code (called *steady state*) is considered as a pipelined "processor" with n "stages". It contains all operations merged from n successive loop iterations. Repetition of this code advances computation of n loop iterations in "parallel". The code is embedded in a prologue, initiating the n first iterations, and an epilogue finalizing the n last ones. If parallel instructions are to be generated the technique has to be combined with scheduling in order to achieve both a short and compact steady state.

Aiken and Nicolau [AiN88a] suggest a technique to find the pattern of a steady state by combining a greedy scheduling with loop unrolling. In [AiN88b] the same authors introduce *Perfect Pipelining*, a combination of loop unrolling and *percolation scheduling*. The latter technique is based on local transformations of the program graph, which move operations backwards as far as possible and combine them with other operations to form compound instructions. Here the pattern of the steady state is developed in the program graph.

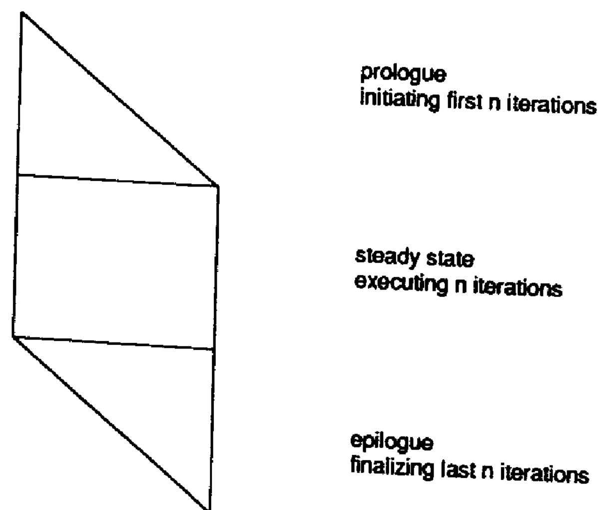

Fig. 8 Loop Pipelining

The so called *Software Pipelining* technique [Lam88] achieve the same goal without unrolling the loop explicitly. On the base of a schedule for a single instance of the loop body and the available resources a lower bound for the distance of the initiation of two adjacent loop iterations is computed. Starting from that lower bound schedules with the operations from several loop iterations are tried until the one for the steady state is found. Lam applied the technique in a compiler for the iWarp processor. Similar techniques are presented for the Cydra in [RGP82]. The techniques mentioned here further differ in the ability to handle branching and nested loops within the loop body.

Acknowledgment: The cooperation with P. Pfahler and G. Piepenbrock in a project for VLIW compilation (supported by the DFG) and with Chr. Ewering and G. Gerhardt in a project for high level hardware synthesis (supported by the BMFT) provided valuable contributions to this paper.

5 References

[AhJ76] Aho, A. V. and Johnson, S. C., *Optimal Code Generation for Expression Trees*, Journal of the ACM 23 (July 1976), 488–501.

[ASU86] Aho, A. V., Sethi, R. and Ullman, J. D., *Compilers Principles, Techniques and Tools*, Addison Wesley, Reading, MA, 1986.

[AiN88a] Aiken, A. and Nicolau, A., *Optimal Loop Parallelization*, Proc. SIGPLAN '88 Conf. Progr. Lang. Design and Impl. (June 1988).

[AiN88b] Aiken, A. and Nicolau, A., *Perfect Pipelinig : A New Loop Parallelization Technique*, Proc. 2nd Europ. Symp. on Programming, Nancy (March 1988).

[Bel66] Belady, L. A., *A Study of Replacement Algorithms for a Virtual Storage Computer*, IBM Systems Journal 5 (1966), 78–101.

[Bru81] Brucker, P., *Scheduling*, Akademische Verlagsgesellschaft, Wiesbaden, 1981.

[Cha82] Chaitin, G. J., *Register Allocation Spilling via Coloring*, SIGPLAN Notices 17 (June 1982), 98–105.

[Cof76] Coffman, E. G., ed., *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, New York, 1976.

[CGL89] Cohn, R., Gross, T., Lam, M. and Tseng, P. S., *Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor*, SIGPLAN Notices 24 (May 1989), 2–25.

[Dun90] Duncan, R., *A Survey of Parallel Computer Architectures*, IEEE COMPUTER (February 1990).

[Ebc88] Ebcioglu, K., *Some Design Ideas for a VLIW Architecture for Sequential-Natured Software*, Parallel Processing (1988).

[Ell86] Ellis, J. R., *Bulldog: A Compiler for VLIW Architectures MIT Press*, Cambridge, MA, 1986.

[Fis81] Fisher, J. A., *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Trans. on Computers C-30 (1981), 478–490.

[GiM86] Gibbons, P. B. and Muchnick, S. S., *Efficient Instruction Scheduling for a Pipelined Architecture*, SIGPLAN Notices 21 (July 1986), 11 - 18.

[HeG83] Hennessy, J. and Gross, T., *Postpass Code Optimization of Pipeline Constraints*, ACM Transactions on Programming Languages and Systems 5 (1983), 422–448.

[HJG82] Hennessy, J. L., Jouppi, N., Gill, J., Baskett, F., Strong, A., Gross, T. R., Rowen, C. and Leonard, J., *The MIPS Machine*, Proceedings IEEE Compcon, San Francisco (February 1982).

[Hu61] Hu, T. C., *Parallel Sequencing and Assembly Line Problems*, Operations Research 9 (1961).

[IBM90] IBM Corporation, *IBM Risc System/6000 Technology*, 1990.

[IKI89] Isoda, S., Kobayashi, Y. and Ishida, T., *Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph*, IEEE Transactions on Computers (October 1989).

[Kas90] Kastens, U., *Übersetzerbau*, Oldenbourg, München, 1990.

[Ken81] Kennedy, K., *A Survey of Data Flow Analysis Techniques*, in Program Flow Analysis: Theory and Applications, Steven S. Muchnick and Neil D. Jones, eds., Prentice Hall, Englewood Cliffs, NJ, 1981, 5–54.

[LaA83] Lah, J. and Atkins, D. E., *Tree Compaction of Microprograms*, Proc. 16th Annual Microprogramming Workshop (1983).

[Lam88] Lam, M., *Software Pipelining: An Effective Scheduling Technique for VLIW Machines*, Proc. SIGPLAN 88 Conf. on Progr. Lang. Design and Impl. (June 1988).

[Mar89] Margulis, N., *The Intel 80860*, Byte 14 (December 1989), 333–340.

[McF83] McFarland, M. C., *Computer-Aided Partitioning of Behavioral Hardware Descriptions*, 20th Design Automation Conference (1983).

[Nic87] Nicolau, A., *Loop Quantization Or Unwinding Done Right*, in Proc. 1st Supercomputing Conf., Lecture Notes in Computer Science, vol. 297, Springer Verlag, Heidelberg, 1987, 294–308.

[PaS82] Patterson, D. A. and Sequin, C. H., *A VLSI RISC*, Computer 15 (September 1982), 8–22.

[Pfa88] Pfahler, P., *Übersetzermethoden zur automatischen Hardware-Synthese*, Universität-GH Paderborn, FB 17, Dissertation, 1988.

[RGP82] Rau, B. R., Glaeser, C. D. and Picard, R. L., *Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support*, Proc. 15th Annual Microprogramming Workshop (1982).

[SeU70] Sethi, R. and Ullman, J. D., *The Generation of Optimal Code for Arithmetic Expressions*, J. ACM 17 (October 1970), 715 - 728.