VLIW Compilation Techniques for Superscalar Architectures

Esther Stümpel, Michael Thies, and Uwe Kastens

Universität-GH Paderborn, Fachbereich 17, D–33102 Paderborn, Germany
{elfriede, mthies, uwe}@uni-paderborn.de

Abstract Efficient use of multiple functional units in superscalar processors requires instruction level parallelism to be detected and exploited. Thus special hardware in the form of dispatch units is used to uncover scheduling opportunities within an instruction window at run-time.
Using the superscalar *PowerPC 604* as an example we show that such processors still benefit from more broadly scoped scheduling at compile time. In our approach we reuse an existing retargetable *VLIW* compiler environment by instantiating it for a *VLIW* processor whose resources and instruction timings resemble those of the *PowerPC*.
This paper presents the transformation of the resulting horizontal *VLIW* code into vertical superscalar code and gives measurements showing the improvements gained from compile time scheduling.

1 Introduction

In modern superscalar processors, like *PowerPC 604*, a dispatcher unit distributes a stream of instructions to parallel functional units. Since that kind of instruction scheduling at run-time is based on very limited information, compile time scheduling can further improve utilization of the functional units. In this paper we show that *VLIW* techniques can be used for that purpose: We instantiate a retargetable *VLIW* compiler backend as if the *PowerPC* was a *VLIW* machine. The scheduled broad instructions are then linearized to be fed into the superscalar processor. There the dispatcher is expected to reproduce a schedule close to the one planned by the compiler.

Our development environment for *VLIW* compiler backends consists of several machine independent code scheduling modules. All machine specific information needed, is encoded in a single separate module, the machine specification module. This module itself is generated from a higher level declarative machine specification language called **Masl**[1], which allows to describe all relevant resources of the processor and information relevant for scheduling. Thus our *VLIW* environment is easily retargetable. Using the environment to generate a compiler for the *PowerPC 604* microprocessor requires two steps: First its relevant features have to be specified in the machine specification module. From this specification results a compiler, which generates parallelized code for the *604*'s

[1] Machine specification language

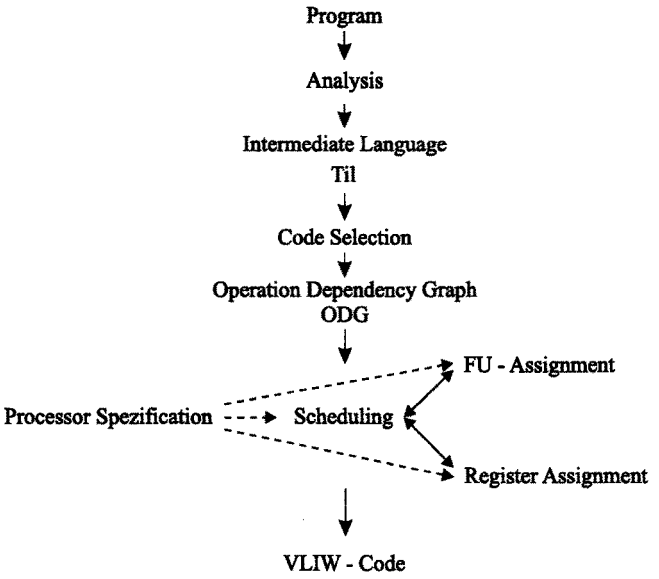

Fig. 1. Structure of the VLIW-Compiler

functional units, but arranged in a sequence of horizontal *VLIW*-instructions. Thus our second task is to transform the horizontal code adequately for the superscalar dispatching.

Figure 1 gives a concise overview of the scheduling environment. It was originally designed for comparative evaluation of different scheduling heuristics for *VLIW* processors. Because of its modular design it is easy to add or modify scheduling algorithms in the generated compiler backend. For our experiment the retargetability was the key aspect.

Code generation starts from an intermediate language representation (`Til`[2]) which is produced among others by an *ANSI C* frontend based on `lcc`[3]. During the code selection phase the intermediate representation is transformed into the operation dependency graph (*ODG*), which is the central data structure for functional unit assignment, register bank assignment, and scheduling. In this structure the nodes represent the machine operations and edges are used to model the data and control flow dependencies between the operations.

During the functional unit assignment phase operations from the *ODG* are mapped to specific functional units of the target processor. The chosen mapping remains fixed for the whole process. After scheduling the register assignment phase maps the symbolic registers referenced from the *ODG* to the physical register banks of the target processor. The last phase of our compiler backend outputs code in *VLIW* form.

The rest of the paper is organized as follows: In Sect. 2 we present the abstract machine model and discuss which aspects of the *PowerPC 604* are selected for

[2] The intermediate language

a model dedicated for instruction scheduling. Sect. 3 shows how the horizontal *VLIW* is linearized such that the dispatcher is guaranteed not to violate constraints when parallelizing the code again. Finally, in Sect. 4 we present the effects of scheduling.

2 Modelling the PowerPC 604

The instruction scheduling phase of our compilation environment is retargetable. It is instantiated for a particular processor by a description of processor resources, operations, and their resource usage patterns. Such a description is a model of the processor dedicated for instruction scheduling. Although that model is derived from the architecture description, it usually deviates significantly from it. Such deviations result from a process of decisions on which aspects of the processor are relevant for code optimization and scheduling, and thus for the quality of the generated code. Aspects that are not relevant can be neglected without loss of code quality. We first give a rough overview of the *PowerPC 604* architecture, and then point out some typical decisions made in modelling it for instruction scheduling.

2.1 Processor Architecture

The *PowerPC 604* [7,8] is a superscalar processor consisting of six execution units that operate in parallel. It is capable of issuing up to four instructions simultaneously and as many as six instructions can finish execution per cycle subject to certain restrictions. Fig. 2 gives an architectural overview.

There are three integer units: two single-cycle integer units and one multiple-cycle integer unit. Additionally there is a floating-point unit, a load/store unit, and a branch processing unit. These functional units access the register files to fetch their operands and write back their results. The integer units are connected with the general-purpose register file, the floating-point unit accesses the floating-point register file, and the load/store unit has access to all registers. There are separate registers for the branch unit, including the eight independent condition register fields.

Specific components ensure proper flow of instructions and operands and guarantee correct updating of the architectural machine state: The fetch unit supplies instructions to the eight-word instruction buffer; the decode/dispatch unit decodes instructions and dispatches them to the appropriate execution units; and the branch processing unit provides the fetch unit with branch target addresses, when a branch is predicted. Furthermore there is the completion unit, which finalizes instruction execution, i.e. writes back the results from the rename buffers to the corresponding register bank. This special unit is needed, because instructions are allowed to pass the execution stage *out-of-order*. To assure correct updating of the architectural machine state, an instruction isn't completed until all instructions ahead of it, corresponding to original program order, have been completed [7].

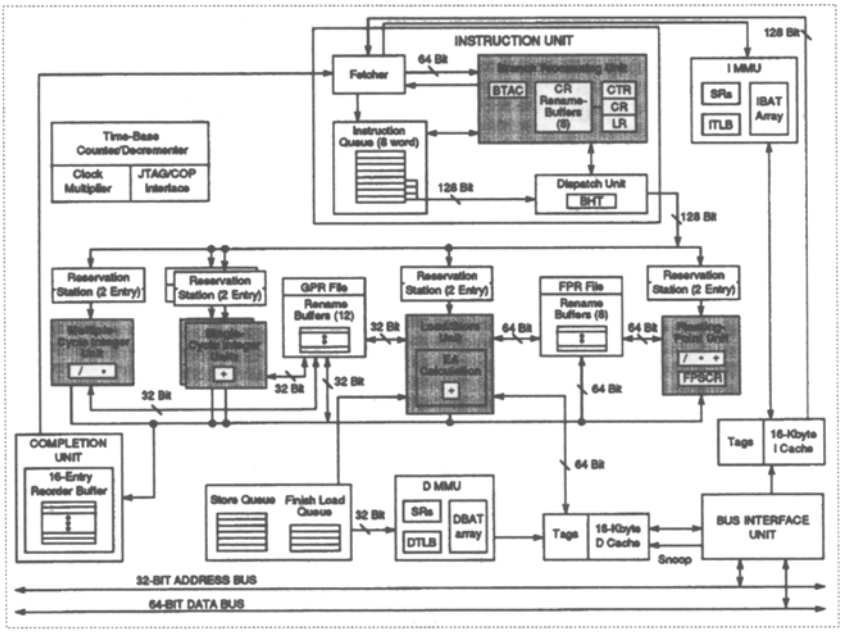

Fig. 2. Block diagram of the PowerPC 604 [7]

The general instruction flow follows the common pipeline shown in Fig. 3. Each instruction passes through six stages. They are common to all instructions except for the execution stage, which depends on the class of the instruction. In this special stage some of the individual execution units are even pipelined by themselves. The floating-point unit, for example, consists of three stages through which each floating-point instruction has to pass.

With the information given so far we can see, that the *PowerPC 604* with its multiple functional units and the pipelined instruction flow very much resembles a *VLIW* architecture, so that it is suitable for our experiment.

2.2 Instruction Scheduling Model

The key aspect in modelling a processor for instruction scheduling is to decide, which components of the processor are needed for a correct model and which can be neglected, because they have no influence on code optimization and scheduling. It is important to keep in mind, that the modelled processor is only a specific view of the real processor, kept as simple as possible, while still displaying realistic behavior with respect to the point of interest.

Our first simplification of the *PowerPC 604* concerns its instruction pipeline (shown in Fig. 3). The first three stages, fetching, decoding and dispatching, are common to all instructions. They are done in one step each and form a common prefix for the execution of each operation. Of course fetching of operations depends on the actual state of the instruction cache, but this is information which

in our approach is not available at scheduling time and therefore is neglected. There are no special constraints for decoding and dispatching. One only has to consider, that dispatching is restricted to a maximum of four operations per cycle. This is enforced indirectly by the number of available register ports which are needed for operand fetch. For these reasons we do not model the first three stages of the master pipeline. Assuming that the write-back capacity of the *604* processor is adequate, we even do not model the last two stages, because they too have no influence on instruction scheduling.

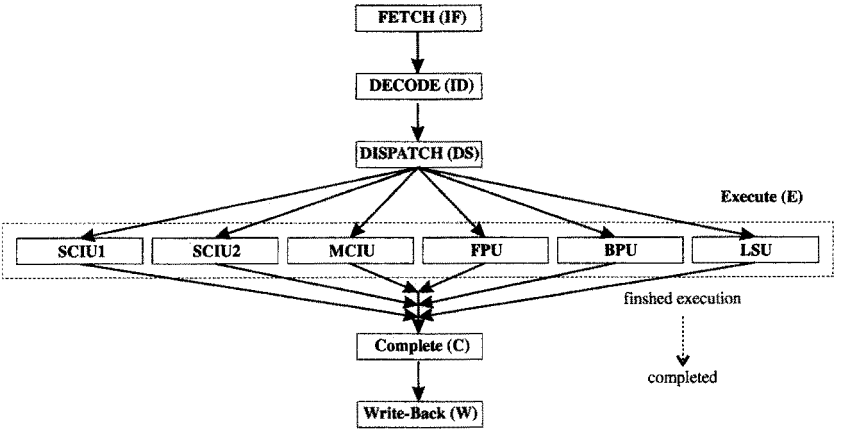

Fig. 3. Common pipeline of the PowerPC 604

From a scheduling perspective the execution stage of the master pipeline is the most interesting one, as it differs with the class an operation belongs to and thus the functional unit on which it is executed. As mentioned above some execution units themselves have internal pipelines which must be modelled in particular.

Our first decision concerning these pipelines is not to model the reservation stations, that belong to each execution unit. This is admissible, because they do not affect the execution of properly scheduled code. In the *604* they are needed to empty the instruction queue even if the execution units or operands are not yet available, so that instruction fetch can proceed without let. The compiler calculates an arrangement of the operations which does not rely on reservation stations. They can be seen as hidden reserves, on which one can fall back at execution time, if the schedule isn't obeyed to.

Conversely for the load/store unit the associated load/store queues are modelled. They are used for temporary storage of instructions for which the effective addresses have been translated and which are waiting to be completed. For every cache miss of an operation in these queues a cache block reload begins, and the next operation tries to access the cache. Thus the actual execution time for load/store instructions varies depending on the actual cache state. Only the

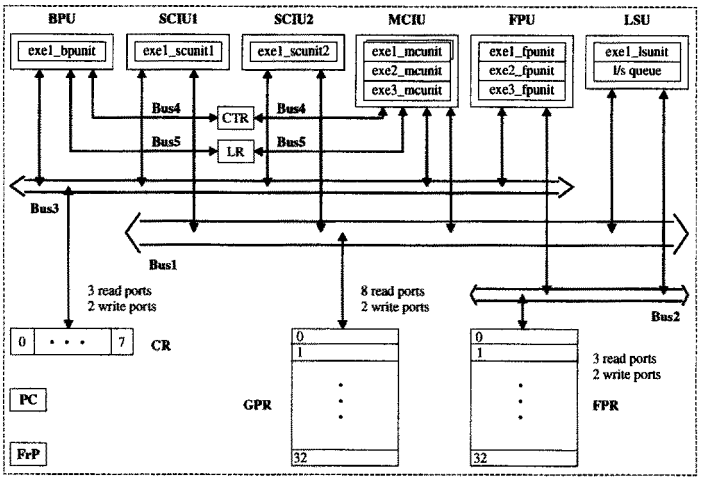

Fig. 4. The modelled processor

time for address calculation is predictable, so that we decided to model this stage and the load/store queues separately.

Fig. 4 shows the abstract model of the *604* resulting from the above decisions. One can see the internal pipelines for the execution stage and both register banks as well as the special purpose registers and the interconnecting buses.

Besides those hardware components other aspects have to be modelled, i.e. special relations between operations and special constraints for the execution of some operations. An example for such a special relation is the so called *dispatch serialization*.

This serialization constraint occurs when a `mtspr`[3] instruction that accesses either the count or link register (`CTR`, `LR`) or an `mtcrf`[4] instruction that accesses multiple condition register fields, which are all part of the *branch* unit, is dispatched to the responsible multiple-cycle *integer* unit. In these cases an interlock must be set, such that no further of these instructions or depending branch unit instructions may be dispatched until the original instruction executes and clears the interlock.

In order to model this serialization constraint we introduce a *virtual* resource `disp_serial` which is available only once. This resource is required for every execution cycle in all instructions subject to dispatch serialization and each branch unit instruction. In this way at any time only one of these operations is allowed to execute.

Similar constraints can even be modelled without such a virtual component: Some floating-point instructions for example aren't allowed to execute until any preceding instruction has left the three stage floating-point pipeline. Then this instruction starts execution and any subsequent floating-point instruction has to

[3] move to special purpose register
[4] move to condition register fields

wait until it has completed. This special rule is modelled by requiring all three stages of the internal execution pipeline simultaneously in each execution cycle of such an instruction.

These two examples in particular show that with rather simple techniques and little expense one can model the features relevant for scheduling instructions on the *PowerPC 604* processor.

3 Linearization of Horizontal Code

In this section we show how the horizontal code produced by the compiler backend is linearized to be processed by the dispatching unit of the *PowerPC*. The compiler schedules instructions as if the target was a *VLIW* processor. Each instruction consists of several operations to be executed in parallel by functional units of the processor. However, the *PowerPC* processes linear code and parallelizes the operations dynamically in the dispatching unit. The situation is shown in Fig. 5. There are two general requirements for linearization of the horizontal code: Dependencies between operations must be preserved, and the parallelization decided by the dispatcher should come close to the schedule planned by the compiler.

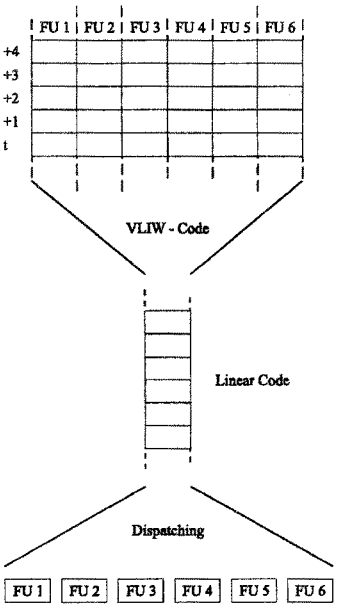

Fig. 5. Transforming horizontal VLIW code into vertical superscalar code

The key aspect in sequentialization is to preserve program semantics: Fig. 6 shows how careless sequentialization can change the semantics of a program.

Operation u and v both read operands in their first cycle of execution and write back their respective result in the last cycle of execution. The dependency between u and v stems from the fact that operation v overwrites an operand of operation u. In our example both operations are executed in the same *VLIW* instruction. This is possible because in *VLIW* architectures all reads happen synchronously before writes of the same instruction. Because of the dependency between u and v, v shouldn't be executed too early, i.e. no more than six cycles before u reads its source operand.

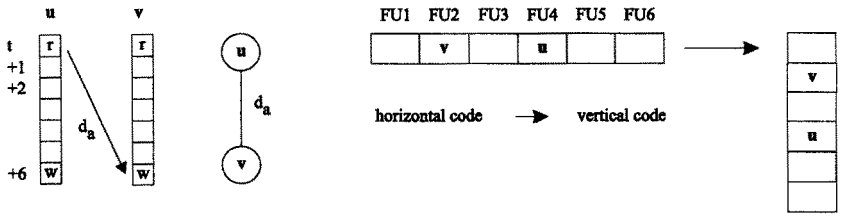

Fig. 6. Example for a naïve sequentialization changing semantics

If our sequentialization strategy would simply decompose the *VLIW* instruction from left to right, it would be impossible for the dispatcher to infer the correct dependency. On the contrary it would decide that u should never be executed earlier than v.

Thus it is of vital importance to arrange the operations of a single *VLIW* instruction in such a way that all dependencies among them are reflected correctly. This can be done by sorting them according to their dependency relations, using the structure of the data dependency graph *ODG*, which was built during the code selection phase. The nodes of the *ODG* represent operations of the program. Two nodes are connected by a directed edge from a to b, if there is a direct dependency from a to b (see Fig. 7). Thus a is on a higher *level*[5] of the directed acyclic *ODG* than b.

Taking this into account one can easily conclude that operations on the same level are independent from each other. They can be executed in arbitrary order. But operations which are dependent on each other are connected by a directed path in the *ODG*. Therefore they are on different levels. It follows:

> b is dependent on a if and only if there is a directed path from a to b in the *ODG*, with level(a) > level(b).

The correct preservation of all data dependencies between operations of a single *VLIW* instruction is guaranteed by sorting these operations in non-increasing order of their *levels* in the *ODG*. The correct dependencies between operations from different *VLIW* words are maintained by not changing their

[5] The level ist the set of nodes with equal distance from the root of the *ODG*.

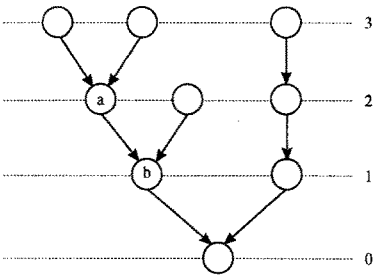

Fig. 7. Operations in the ODG structure

relative positions during sequentialization. Sequentialization is done during code output, which means it is a local and independent task for each single *VLIW* word. Because each *VLIW* word contains a constant number of operations, it is clear that sorting adds only small constant time to the processing of each instruction word.

If cache access allows, the dispatcher reconstructs the schedule from the sequential instruction stream. In case of deviation from the schedule there is no risk of making things worse, because we have preserved all data dependencies. Independent calculations have been interleaved by scheduling, so that the dispatchers lack of overview[6] is compensated, which has a positive effect even if there is deviation from the schedule plan.

Ad-hoc factors like cache misses or interrupts cannot be regarded at scheduling time. Their occurrence causes a temporary deviation of the *dynamic* schedule produced by the dispatcher from the *static* schedule planned by the compiler. This is the point where static planning and dynamic dispatching complement each other.

4 Scheduling Results

Instruction scheduling techniques for *VLIW* processors have been proven to be effective in decreasing execution time for that class of processors. Those techniques are made applicable for a superscalar processor by modelling it as if it was a *VLIW*-processor and subsequently serializing the scheduled horizontal code. We now present runtime measurements that show how *VLIW* scheduling performs on the *PowerPC 604*. In particular the results of different scheduling techniques are compared and the effects of static assumptions for load/store latencies are analyzed.

Measurements of generated code are performed on a *AIX* system with a single *PowerPC 604* processor, running at a clock rate of 110 MHz. The programs selected for testing are taken from the Stanford benchmark[7]: including three well

[6] In case of the *604* processor the dispatch unit has a local window of four operations on the code. But because of the reservation stations its decisions are based on a broader view of the code stream taking some additional operations into account.

[7] Programs have been made conformant to the *ANSI C* Standard.

known sorting algorithms, namely bubblesort, quicksort and treesort (**bubble**, **quick** and **trees**). Furthermore two matrix multiplication programs, one of them with integer coefficients (**intmm**) and the other with floating point coefficients (**mm**). Another benchmark from the mathematical domain is a program for *fast fourier transformation* (**fft**). and a program solving the *eight queens problem* (**queens**). All programs are embedded into external loops and repeated as often as needed to achieve significant timing results. All measurements are conducted under low load in multiuser mode. In order to filter random effects we take only runs without page faults and a *CPU* utilization greater or equal to 99 %. Then we calculate the mean value of five execution times, measured under these conditions.

Our goal is to measure the benefits of scheduling on its own. Unfortunately most standard compilers allow scheduling only to be activated in conjunction with an elaborate set of complementing optimizations. This makes it difficult to contribute the resulting savings in execution time correctly. Furthermore the aim of our work is not the absolute quality of the applied scheduling techniques but the feasibility of adapting them to superscalar processors—while preserving their positive effects. Thus a comparision of our results with those of a standard compiler would not contribute meaningful information in this respect.

4.1 Comparison of Scheduling Techniques

Examination of the run time behavior of code scheduled with different techniques shows, that the overall behavior is similar to that known from *VLIW* architectures: list scheduling techniques, which are known to be effective for *VLIW* show similar results on the superscalar *604* processor and loop restructuring techniques, namely software pipelining, show promising results in cases where they are expected to be successful. This is an important result, because it makes the transfer of *VLIW* optimizations to superscalar architectures worth while.

However, there exist some deviations from the *VLIW*-like behavior due to some characteristics of the *PowerPC 604* processor, respectively due to some particularities of our *VLIW* development environment. Especially the loop restructuring techniques which are present in our environment, are often disabled by the fact that they rely on special *VLIW* features, namely register queues, which usually do not exist in superscalar processors. But there are many techniques to solve this problem [2,4,6]. Another effect which turned out to be a handicap for loop restructuring is the fact, that the enormous code compactness of the restructured code overcharged the *604*'s write-back capacities. This causes an extraordinary occurrence of write-back stalls, destroying any positive loop scheduling effect.

This is a surprising effect, which shows that the write-back phase is not well balanced with the capacity of functional units. To consider this fact in future approaches a virtual resource, representing this bottleneck, has to be introduced in our modelling of the *604* processor.

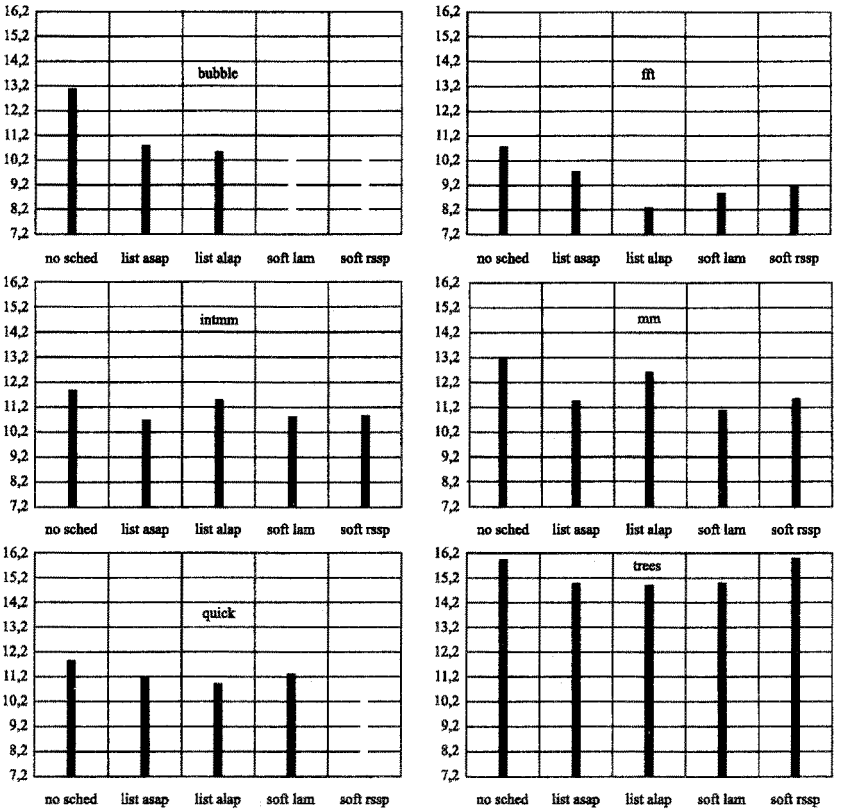

Fig. 8. Execution times in seconds for different VLIW scheduling techniques

In Fig. 8 the execution times in seconds for each of the scheduling algorithms are shown. We use the following abbreviations:

list asap : List Scheduling with As Soon As Possible Strategy;
list alap : List Scheduling with As Late As Possible Strategy;
soft lam : Software Pipelining according to Monica S. Lam;
soft rssp : Resource Sensitive Software Pipelining.

The first scheduling techniques are well known and can be found in literature [1,5]. The last one was developed in our research group and presented in [9]. The special feature of software pipelining lies in the fact, that different iterations of one loop can be interlaced and thus processed in parallel, to allow for better utilization of the parallel functional units inside a *VLIW* or *superscalar* processor. Operations outside any loops are scheduled with a standard technique, in our case with list scheduling.

Because it is not always possible to schedule loops successfully with software pipelining, in some cases the execution time of the scheduled code with software pipelining equals that of the scheduled code with list scheduling. These cases

are emphasized in the diagrams by using grey fill patterns for the time indicator instead of black.

Discussion of Results Concentrating first on the result of the simple list scheduling techniques, we find out that usually both have produced significant improvements in execution time. The maximum improvement rate is 23 % (`fft`) and the minimum rate is 3 % (`intmm`).

To discuss the results of the code scheduled with software pipelining techniques further information is needed. The execution time of a loop scheduled with software pipelining is dependent on two relevant factors. The first factor is the so called *Initiation Interval* (*II*), which is the length of the folded loop body. Its worst case equals the length of the original loop body. The second relevant factor is the number of times the loop is iterated. Software pipelining adds a fixed overhead to the new loop body that must be amortizised. Thus the number of iterations must be large enough to reach gains in execution time despite of the existing overhead.

It would go too much into detail here to present these values and discuss them exhaustively. Instead of this we give a short summary of the essential facts. There are two opposite effects:

1. In cases where the initiation interval is long, which means near or equal to the length of the unscheduled loop body, it appears that the run through values are small, so that there is an overall negative effect on execution time compared to pure use of list scheduling;
2. In other cases, where the initiation interval is short, which means about half the original length of the loop body, another problem arises: the lifetime of some register values inside the pipelined loop extends beyond the length of the initiation interval. So the pipelined loops would require special hardware support (register queues) not present in the *PowerPC* to be executed correctly.

For this reason we cannot take full advantage of the software pipelining techniques implemented in our development environment. But this is no real problem because there exist several solutions to the lifetime problem, although up to now none of them is implemented inside our environment.

To avoid the impression that software pipelining doesn't show any positive effects we refer to the programs `fft`, `intmm` and `mm`: For the `fft` program the initiation intervals found were relatively large too, but the pipelined loop was iterated so many times that the seemingly minor effect cumulated to a considerable gain in execution time.

Nevertheless this gain in execution time is smaller than that accomplished with any of the list scheduling strategies. But this goes along with another effect which is most prominent in loops allowing for a strong pipelining. Looking at the pipelined loop body of the central loop in the `fft` program, one could see that it consists mostly of operations which write back their results to integer registers. Because of this enormous compactness present in the loop body, there

is high pressure on the integer write back capacities of the *PowerPC 604*, which seem to be a possible bottleneck of the processor design. Indeed the processor is not able to process the compact loop body in the expected way, which leads to a diminished gain in execution time.

The programs `intmm` and `mm` are completely identical except for the fact that the first one requires integer arithmetic and the second one floating point arithmetic for the actual matrix multiplication. Indeed they show a similar behavior concerning the execution times, but not exactly the same: whereas for `mm` software pipelining reaches the largest gain in execution time, for `intmm` the list scheduling technique is more successful. This is astonishing, because the *604* has three integer units but only one floating point unit. So, one could expect, that there is a higher amount of parallelism in the integer version of matrix multiplication, than in the floating point version, and therefore a higher gain in execution time due to more compact loop bodies. But as for the `fft` program, greater compactness in code overcharges the write-back capacity of the *604*.

In order to verify this effect we have made minor modifications to the processor model which simulate a better approximation to the actual write-back behavior. For the `fft` benchmark these changes improve the performance of software pipelining significantly—even beyond the list scheduling techniques. However, some other test programs showed only marginal changes in execution times. This might be caused by the remaining small differences between the model and the real write-back stage or by additional effects not yet uncovered.

4.2 Latencies Modelled Imprecise

The code for a *VLIW* processor has to be scheduled exactly with respect to operation timings, and it is executed exactly according to that schedule. In our approach however, the scheduled code for the superscalar processor can be understood as a suggestion that may or may not be rearranged by the dispatcher. Hence, schedules are executed correctly even if they are too densely packed or unnecessarily scattered due to imprecise assumptions on operation latency. The effect should be observable by longer run-times in either case.

We changed the latencies for all load/store instructions, as a typical and important class of instructions. Changing load/store instructions was chosen because they allow for increase and decrease in modelled latency and there are no exact predictions of actual latencies available. The latencies were varied in both directions, at which decreasing stopped by a minimum value of one cycle latency for each instruction.

We examined the effects on execution time for the programs `bubble` and `fft`. The latter was selected, because of its complexity. Besides a sufficient number of load and store instructions it contains a manifold mixture of other instructions. This is important because, without any instructions executing on different functional units it would be impossible to observe any effect in changing load/store latencies. They would only result in multiple empty *VLIW* instructions, which would be deleted during sequentialization.

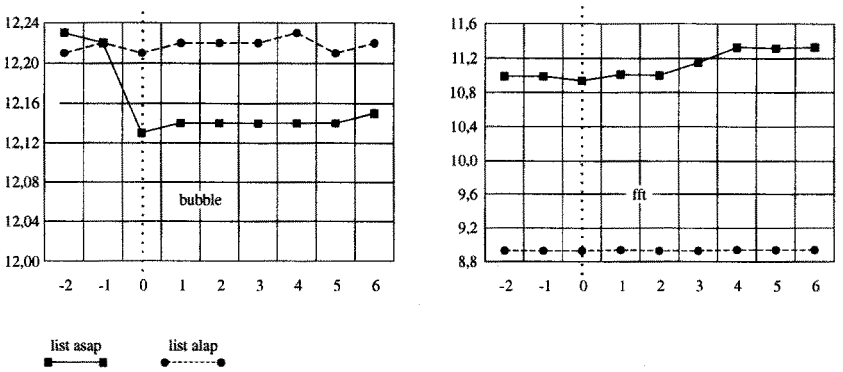

Fig. 9. Variation of latencies for load/store instructions

For this evaluation we only used the list scheduling techniques, because of their straight behavior. The results can be seen in Fig. 9. To illustrate the tendency of changes in execution time we have connected the discrete measure points by lines, although no intermediate values exist.

Two things can be seen: first the effect of changing latencies on execution time of the generated code is small. For `bubble` it is below 1 % and for the more complex `fft` a greater maximum of 3.5 %, which could be expected because it offers more opportunities for code rearrangements performed by the scheduling algorithm. The second aspect we see is that there exists a minimum for the originally chosen latencies.

It becomes clear by this example that the superscalar *604* processor with its dispatcher unit is quite tolerant against local or inexact scheduling assumptions. As mentioned above the code isn't guaranteed to be executed exactly as planned. The dispatcher unit decides the ultimate order of execution within a local scope, dependent on actual processor state. In this case it leads to the advantage, that code which was generated for a certain processor could run with good performance on a similar processor, even if there are differences in their models.

5 Conclusions

Our development environment for *VLIW* compiler backends has proven to be really retargetable, even when the target architecture was switched from abstract research-oriented *VLIW* processors to a real-world superscalar processor. Without retargetability it would not have been feasible to transfer our existing *VLIW* compiler techniques to the superscalar *PowerPC 604* with surprisingly little effort. Processor modelling and implementing code linearization have been the essential tasks with the linearization step adding only slightly to the run time of the generated compiler backend.

Measurements have shown that dynamic scheduling at run time is improved by static scheduling at compile time. Groups of operations beyond the dispatcher's instruction window can only be interlaced by statically rearranging the whole

code in (an approximation of) a perfect schedule. The static schedule becomes tolerant against disturbing dynamic effects occurring at run time (like unpredictable memory latencies) by means of the dynamic scheduling performed by the dispatcher. This guarantees correct execution and allows for temporary deviations from the planned static schedule. Tightly parallelized static loop schedules uncovered an unexpected problem: A bottleneck in the write-back capacity of the *PowerPC 604* causes the dispatcher to break an otherwise feasible schedule.

To summarize, we have shown that scheduling benefits for *VLIW* can be carried over to superscalar processors. In practice this can be done with minimal effort, if the *VLIW* techniques are implemented retargetable for different *VLIW* architectures. Static and dynamic scheduling complement each other, providing desirable robustness of static planning against dynamic effects.

Acknowledgments Thanks to the anonymous referees for their useful comments and suggestions.

References

1. E. G. Coffman. *Computer and Job-Shop Scheduling Theory.* John Wiley, New York, 1976.
2. Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Minimum register requirements for a modulo schedule. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 75–84, San Jose, California, November 30–December 2, 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.
3. Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.
4. Richard A. Huff. Lifetime-sensitive modulo scheduling. *SIGPLAN Notices*, 28(6):258–267, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*
5. M. Lam. Software pipelining: An effective scheduing technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.
6. Josep Llosa, Antonio González, Eduard Ayguadé, and Mateo Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 80–86, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.
7. Motorola. *PowerPC 604: RISC Microprocessor User's Manual.* USA: Motorola Literature Distribution, P.O. Box 20912, Phoenix, Arizona 85036, 1994.
8. Motorola. *PowerPC Microprocessor Family: The Programming Environments.* USA: Motorola Literature Distribution, P.O. Box 20912, Phoenix, Arizona 85036, 1994.
9. Peter Pfahler and Georg Piepenbrock. A Comparison of Modulo Scheduling Techniques for Software Pipelining. In *Proc. 6th International Conference on Compiler Construction, CC'96*, volume 1060, pages 18–32. Lecture Notes in Computer Science, Springer Verlag, 1996.