Integrated Register Allocation and Software Pipelining

Johan Janssen Henk Corporaal Ireneusz Karkowski

Department of Information Technology and Systems,
Delft University of Technology,
P.O. Box 5031, 2600 GA Delft, The Netherlands

johan@cs.et.tudelft.nl heco@cs.et.tudelft.nl irek@cs.et.tudelft.nl

Keywords: register allocation, instruction scheduling, instruction level parallelism, TTA, software pipelining

Abstract

Software pipelining is a powerful and efficient scheduling technique for exploiting instruction level parallelism in loops, it results in high performance but it increases the register requirements. Two methods are available to reduce the register requirements: increase the schedule length or insert spill code. Traditionally instruction scheduling and register allocation are applied in separate phases. However decisions made by one phase may negatively impact the other. In this paper we describe a method that integrates register allocation and software pipelining in a single phase. Also the insertion of spill code is discussed. The results of the experiments show that this integration may result in large performance improvements.

1 Introduction

Performance improvements of current computer architectures are mostly obtained by the exploitation of instruction level parallelism (ILP). ILP has been made possible due to higher transistor densities which allows the duplication of function units and data paths. Exploitation of ILP consists of mapping the ILP of the application onto the ILP of the target architecture as efficient as possible. This mapping is used for Very Long Instruction Word (VLIW) and superscalar architectures. These architectures may execute multiple operations per cycle. To take advantage of this potential the compiler uses instruction scheduling techniques to fill the cycles with operations as efficient as possible. Software pipelining is such a technique. It has received widespread attention in academic and industrial research [1, 2, 3]. This scheduling technique exploits ILP of loops by overlapping the execution of successive iterations. During the execution of a non-pipelined loop, the first iteration is started and executed to its completion. The second iteration is then initiated and executed until completion, etc. In contrast, for a pipelined loop, the second iteration of the original loop is allowed to begin before the first iteration is completed. The interval at which iterations of the software pipelined loop are started is called the *initiation interval* (II), which is expressed in the number of cycles. The goal of software pipelining is to find a schedule that overlaps iterations and uses the shortest possible initiation interval. A number of methods on construction of software pipelined loops were published, the most widely used is Iterative Modulo scheduling [2].

The disadvantage of aggressive scheduling techniques, such as software pipelining, is their high register pressure. The register requirements increase when machines have deeper pipelines and/or wider issue. To obtain a valid schedule the compiler must be able to match the register requirements with the available registers of the target processor. When more registers are needed, some additional actions are required, such as the insertion of spill code.

Traditionally register allocation and instruction scheduling are applied in separate phases. This separation has its impact on the produced code. Decisions made by one phase may have negative effects in the other. Also the order in which the two phases are applied is a point of dispute [4].

Recently we developed a method that integrates register allocation and instruction scheduling in a single phase, called Registers on Demand (RoD). We proved its effectiveness for basic block scheduling [5] and for region scheduling [6]. In this article we extend this method to software pipelining. This paper is organized as follows: first related work is discussed in section 2. In section 3 we describe the principle of software pipelining. Then we describe shortly the class of processors for which we will generate code. These processors are used for application specific designs for which it is important to reduce the cost by using a small chip area. Using as few registers as possible can help to achieve this goal. In section 5 the new method is described including the introduction of spill code. In Section 6 the method is evaluated. Finally section 7 states our conclusions.

2 Related Work

Most published work on software pipelining assume that register allocation is done after scheduling which is called *late assignment*. Applying software pipelining first can lead to loops with a high register pressure. Applying software pipelining with the intention to achieve high throughput without considering register requirements may not lead to practical solutions. Some modulo scheduling approaches use heuristics to produce near-optimal schedules with reduced register requirements [3], while others try to reduce the register requirements by applying a post-pass rescheduling process [7]. All these methods have in common that it is not guaranteed that the number of required registers is less or equal to the number of available registers. When this situation arises variables must be spilled to memory. However, software pipelining has some constraints that hinder the use of traditional techniques for register allocation and spilling: life ranges can cross the iteration boundary and can even be longer than the II. Furthermore, inserting spill code in an already scheduled software pipelined loop without affecting the whole schedule is very difficult and in many situations impossible. The option used in the Cydra 5 compiler [8] is to reschedule the loop with an increased II. If after several attempts this also fails, more conventional techniques are used to schedule the loop.

Another option is to allocate registers first and then schedule the loop. This has as an advantage that spilling is already done and no code has to be inserted in already scheduled code. However the standard register allocation approach as presented in [9, 10] introduces false dependencies which will result in a larger II and thus lower throughput. In [11] enhancements to the standard register allocator are presented which try to avoid the introduction of false dependencies.

3 Software Pipelining

Software pipelining is a technique to schedule loops such that each iteration in the produced schedule consists of instructions chosen from different iterations in the unscheduled code. When generating the schedule, the data dependences between operations within the same iteration and between operations of different iterations have to be respected. Many approaches to construct software pipelined loops have been researched [1, 2]. The most popular ones are based on iterative modulo scheduling as proposed by Rau [2]. The algorithm chosen for our research is also based on this method.

3.1 Overview of Modulo Scheduling

Iterative modulo scheduling tries to construct a correct schedule in a minimum number of cycles, called the initiation interval (II). First a lower bound on II called the minimum initiation interval (MII) is computed. Then, iterative modulo scheduling tries to schedule the loop in the cycles of the II. When scheduling fails the II is increased until a valid schedule is constructed that fits in the II cycles.

In order to reduce the number of iterations required to generate a valid schedule, the MII should be estimated as precise as possible. Resource constraints and precedence constraints are used for this estimation. For more information on computing the MII the reader is referred to the literature [2].

The schedule produced by a software pipelining algorithm consists of three different phases: the prologue, the kernel and the epilogue. The pipeline is started by the prologue, then the kernel is executed (the loop-body of the schedule), and finally the epilogue drains the pipeline. The execution time is mainly determined by the kernel.

```
L1:  add   r1,  r1,  #4
     ld    r2,  (r1)
     mul   r2,  r2,  #17
     sub   r3,  r3,  #4
     st    r2,  (r3)
     beqz  r3,  L1
```

Figure 1: Loop-body

In figure 1 the loop that executes the code snippet `b[n..1] = 17 * a[1..n]` is shown. The software pipelined version of the loop is given in figure 2. The loop in figure 1 needs 5 cycles for each loop iteration if we assume a latency of two for the multiplier, a single cycle latency for all other operations and that three operations may execute in parallel. Its software pipelined counterpart in figure 2 starts each 2 cycles a new iteration.

```
------------------Prologue------------------
    add r1, r1, #4
    ld  r2, (r1)
    add r1, r1, #4   mul r4, r2, #17
    ld  r2, (r1)     sub r3, r3, #4
-------------------Kernel-------------------
L1: add r1, r1, #4   mul r4, r2, #17  st    r4, (r3)
    ld  r2, (r1)     sub r3, r3, #4   beqz  r3, L1
------------------Epilogue------------------
                     mul r4, r2, #17  st    r4, (r3)
                     sub r3, r3, #4
                                      st    r4, (r3)
--------------------------------------------
```

Figure 2: Software pipeline with II = 2, R_{press} = 4

A drawback of modulo scheduling is that it can only handle single basic block loops. To overcome this problem if-conversion [12] can be used when predicated execution is supported by the processor.

3.2 Register Pressure

As already discussed in the introduction, software pipelining increases the register pressure of a loop: the life ranges of variables defined in an iteration can overlap with life ranges defined in subsequent iterations. In figure 2 the result of the load and the multiply interfere in the software bypassed version while this was not the case in the original loop. The number of registers needed in the original loop is 3, in the software pipelined version with II = 2 four registers are needed. If however the II is increased with one, only 3 registers are needed. The generated schedule with an II = 3 is given in figure 3.

```
                   Prologue
   add r1, r1, #4
   ld  r2, (r1)
   mul r2, r2, #17
                   Kernel
L1:add r1, r1, #4  sub r3, r3, #4
   ld  r2, (r1)                     st,  r2(r3)
   mul r2, r2, #17                  beqz r3, L1
                   Epilogue
                   sub r3, r3, #4
                                    st   r2(r3)
```

Figure 3: Software pipeline with II = 3, R_{press} = 3

The principle of trading performance against a lower register pressure is generally applicable for software pipelining, however this process cannot be extended infinitely. The loop in the previous example can never be scheduled when less than 3 registers are available. To overcome this problem spill code can be inserted.

```
                      Prologue
   add r1, r1, #4
   ld  r2, (r1)
   add r1, r1, #4   mul r4, r2, #17
   ld  r2, (r1)     sub r3, r3, #4
                      Kernel
L1:add r1, r1, #4   mul r4, r2, #17  mv r5, r4
   ld  r2, (r1)     sub r3, r3, #4   st r5, (r3)  beqz r3, L1

                      Epilogue
                    mul r4, r2, #17  mv r5, r4
                    sub r3, r3, #4   st r5, (r3)
                                     mv r5, r4
                                     st r5, (r3)
```

Figure 4: Software pipeline with II = 2, R_{press} = 5

Another aspect of software pipelining are long living variables (with live ranges larger than the II). Unless special measures are taken, this results in illegal schedules since values are generated before previous ones are read. An example of such a variable is the result of the multiply in figure 4. In the shown schedule we assume that the store operation is due some scheduling constraint placed in the second cycle of the loop. Furthermore, four operations can be executed in parallel. Without extra measurements the result of the previous (wrong) iteration is stored. The method used in the example to make the schedule correct is called a delay line [13]. A delay line is a register copy that splits the live range of a long live variable in two parts. As a result the register pressure increases.

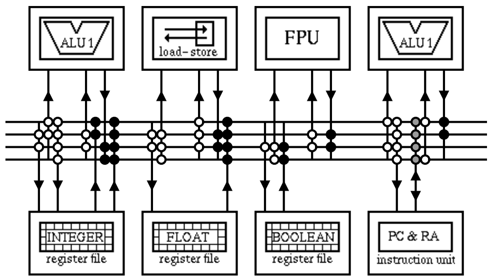


Figure 5: General structure of a TTA

4 Target Architecture

For the experiments we use a processor template. With this template, a processor for a given application can be quickly designed. The processors built, have a transport triggered architecture (TTA), which supports Instruction Level Parallelism (ILP). The available ILP is exploited by the back-end of the compiler.

TTAs resemble VLIW (very large instruction word) architectures; both contain multiple independent Functional Units (FUs). Rather than programming FU operations directly, TTAs are programmed by specifying the required data transports. As a side effect of these transports, FU operations are triggered. Instead of packing the operations in a single instruction, like VLIWs, TTAs pack multiple transports in a single instruction. An extra level of compiler control is introduced due the programming of these transports. More compiler control enables more code optimization opportunities [14].

The heart of TTAs is the data transport network, containing one or more buses, to which FUs and register files (RFs) are connected. This network does not have to be fully connected, although this relaxes the code generation process. An example TTA is given in figure 5. Using a template, the number and type of FUs and RFs and the capacity of the transport network can be changed easily. The template also supports predicated execution by guarding data transports. For more details on TTAs and prototype realizations, we refer to [15].

Each FU contains trigger, operand and/or result registers; a trigger register is a special kind of operand register in the sense that moving an operand to a trigger register starts an FU operation. Performing an operation with n (source) operands consists of moving $n-1$ operands to operand registers and one operand to a trigger register of an FU. The result of the operation can be read from its result register. As an example we show how an `add` instruction translates into three transports (moves):

```
r1 → add.o; r2 → add.t; // Addition
add.r → r3;
r3 → sub.o; r4 → sub.t; // Subtraction
sub.r → r5
```

First, the values of `r1` and `r2` are moved in the operand and trigger registers, `add.o` and `add.t`. After a delay (depending on the latency of the adder), the result can be moved from the result register `add.r` to `r3`. The subtraction is executed in the same manner.

Software bypassing is one of the advantages of TTAs above more traditional architectures. Using software bypassing and dead-code elimination the compiler can eliminate the need of some RF accesses [14], see for example the software bypassed version of the previous code:

```
r1 → add.o; r2 → add.t;
add.r → sub.o; r4 → sub.t;
sub.r → r5
```

The software bypassed version may turn out not to use register `r3` because the result of the addition is bypassed to the subtraction and not needed later on. An early assignment register allocator has no idea which moves are going to be bypassed by the scheduler. If the allocator would have known this in advance then it could have used the freed register `r3` for other live ranges and perhaps could prevent spilling or the introduction of a false dependence. Our integrated register allocation and instruction scheduling method avoids this problem.

5 RoD and Modulo scheduling

The division of instruction scheduling and register allocation into separate phases can adversely affect the performance of these phases and thus the quality of the generated code. In figure 6 the example of figure 1 is translated into TTA code. Its software pipelined counterpart is given in figure 7. When applying an early assignment method, the minimum number of registers needed to hold all variables is three. However this will result in an anti-dependency between the results of the load and multiplier and hence will result in a larger MII. Therefore to obtain a high throughput four registers are needed. After scheduling the number of required registers has

```
L1: #4    → add.o   r1;  → add.t
    add.r → r1
    r1    → ld.t
    ld.r  → r2
    r2    → mul.o   #17;→ mul.t
    mul.r → r2
    #4    → sub.o   r3;  → sub.t
    sub.r → r3
    r3    → st.o    r2;  → st.t
    #0    → eq.o    r3;  → eq.t
    eq.r  → b0
    b0: jump L1
```

Figure 6: TTA code of example

dropped to two, due the ability of TTAs to bypass values. This would support the conclusion that late assignment is more appropriate; however, introducing spill code in the presence of bypassing and software pipelining is a complicated task which can even result in deadlocks.

Since the register requirements of a software pipelined loop are highly dependent on the way the loop is generated it seems beneficial to apply the register allocation and the instruction scheduling of software pipelined loops in a single phase. In previous research we investigated integrated register allocation and instruction scheduling for basic block scheduling [5] and region scheduling [6]. This article describes how this method can be used within the context of software pipelining. In section 5.1 the principles of RoD in combination with software pipelining are discussed for a more in depth discussion of RoD we refer to [5] and [6]. When the number of registers is too small to hold all variables and increasing the II does not elevate this problem, spill code must be inserted. This is the subject of section 5.2.

5.1 Registers On Demand

The used allocation/scheduling algorithm consists of the following steps.

- Control flow analysis, data flow analysis and memory disambiguation are performed.
- Data dependencies are found and the data dependence graph is constructed.
- Regions are scheduled, starting with the innermost region. Some of these innermost regions are selected for software pipelining, the selection procedure is discussed in [13]. These regions are single basic blocks; when necessary if-conversion is used. The other regions are scheduled/allocated as described in [6].

Operations are selected for scheduling according to a priority as described in [13]. An operation-based list scheduler is used to schedule the operations. Scheduling an operation consists of scheduling the moves of this operation. For each operation, we must find an FU on which the operation can be performed and for each move, we must find transport resources which are a move bus and connections to the FUs or RFs. At that point we try to assign registers. In our approach, a register is assigned to a variable as soon as the first move is scheduled which performs a read or a write of this variable.

To ensure that variables with overlapping live ranges are not mapped onto the same register some bookkeeping is necessary. Therefore to each cycle a Register Resource Vector (RRV) is added. This RRV

```
                                   Prologue
     #4 → add.o;  r1 → add.t
     add.r → r1;  add.r → ld.t
     #4 → add.o;  r1 → add.t;   ld.r → mul.o;  #17 → mul.t
     add.r → r1;  add.r → ld.t; #4 → sub.o;    r3 → sub.t
                                   Kernel
L1:  #4 → add.o;  r1 → add.t;   ld.r → mul.o;  #17 → mul.t;mul.r → st.t; sub.r → r3;   sub.r → st.o; sub.r → eq.t;   #0 → eq.o;  b0:jump L1
     add.r → r1;  add.r → ld.t; #4 → sub.o;    r3 → sub.t;   eq.r → b0
                                   Epilogue
                                ld.r → mul.o;  #17 → mul.t;mul.r → st.t; sub.r → st.o
                                #4 → sub.o;    r3 → sub.t
                                                             mul.r → st.t; sub.r → st.o
```

Figure 7: Software pipelined TTA code

indicates whether registers are free or not in this cycle. Initially a software pipelined loop has II = MII cycles and thus MII RRVs, when the II is increased also the number of RRVs increases.

Whenever a variable needs a register, the RRVs of all cycles that are in the live range of the variable are checked for a common free register. For variables that have a definition or a use in the loop, all cycles ,thus II cycles, are checked since in a software pipelined loop there is no way to tell where the other definitions or uses will be scheduled. Normally the definition will be scheduled before the use, but with software pipelining this is not guaranteed. The easiest way to find the set of free registers is to intersect all RRVs of the II. The resulting set contains all free registers. The allocator can pick one of these free registers for allocating the variable. If an operation is scheduled which has already allocated variables, then to keep the bookkeeping precise, the RRVs of the loop should be updated. In the context of software pipelining this is only possible when all uses and definitions of the variable in the loop are scheduled, since only then it is precisely known in which cycles the variable is live.

When a variable is also live outside the software pipelined loop the RRVs of the basic blocks over which this variable is live are also updated when a register is assigned. This has the advantage that no extra analysis or repair code is needed to integrate the software pipelined loop in the total schedule.

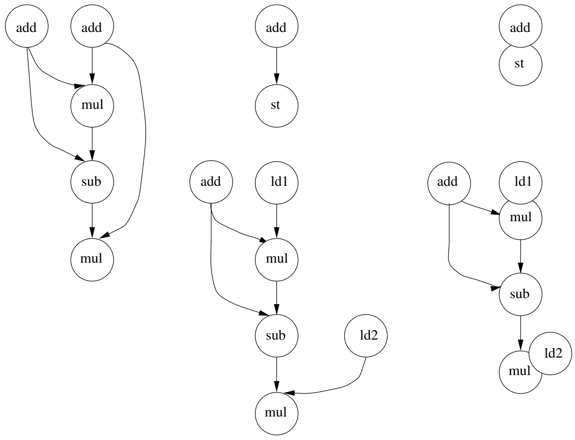
5.2 Spill Code

In practical processors the number of registers in the target machine is not unbounded, therefore the allocator must try to fit all variables in the available registers. When the allocator fails it will normally insert spill code to decrease the register pressure. However when using software pipelining we have another possibility: increasing the II. As was shown in section 3.2 this can result in a decreased register pressure. However increasing the II will not always be enough, since for some loops the minimum register requirements are larger than the available registers. In these situations spill code must be inserted. However straightforwardly introducing spill code will not always lead to a good solution. See for instance figure 8. In figure 8a the original graph is shown, this program requires three registers. When only two registers are available, spill code is introduced as shown in figure 8b, this schedule can be scheduled with two registers. Since instruction scheduling techniques tend to schedule instructions as early as possible it can happen that `ld2` is scheduled in the same cycle as `ld1`. When this happens, the register requirement is still 3 and spilling did not help. The situation gets even worse when additions are needed to calculate the memory address where the variable must be spilled. In [16] this problem is attacked by scheduling the nodes, which variables are spilled, and the spill code itself close together in a single scheduling step. However this does not guarantee that no deadlock situation can arise.

In [5, 6] we described a method for generating and scheduling spill code for basic block and region scheduling that always decreases the register pressure by using the software bypassing ability of TTAs. This method can be used in this situation as well. This method does not only schedule instructions in a single step, but also guarantees that software bypassing is used. Figure 8c shows the graph using this principle. As can be seen the new created live ranges disappear completely which guarantees that this method always converges.

Introducing spill code itself decreases the register pressure, but it has as a side effect that the II can increase which may result in an even larger decrease in the register requirements. In [16] it is even claimed that the introduction of spill code is preferable above increasing the II.

The next question is where in the scheduling process do we add spill code. In [5, 6] we insert spill code at the moment that we can not find a free register. In the context of software pipelining this approach leads to the following problems: 1) when spill code is inserted, the MII can change, so trying to obtain a valid schedule with the original MII, which is probably lower than the new one, is useless. 2) Increasing the II can result in a lower register pressure without spilling, so first we should try to increase the



(a) (b) (c)

Figure 8: Example of spilling.

II. In the approach presented here we first try to schedule/allocate the loop without spill code. When the scheduling process fails the II is increased. If the II is increased with more cycles than a predefined threshold value and still no valid schedule is found the algorithm inserts spill code for a variable. This process is illustrated in the figure 9.

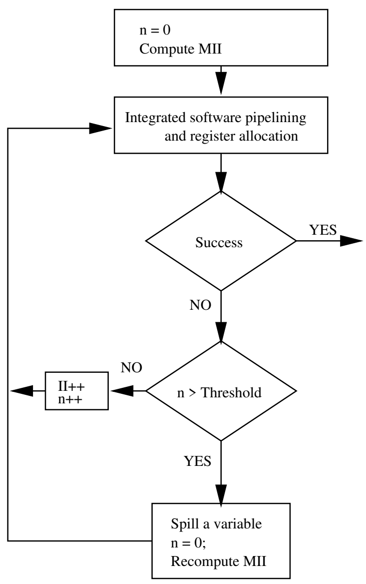


Figure 9: Flow diagram scheduling/allocation.

The live range that is selected for spilling, is the one with the lowest spill costs. The spill cost of a variable v is computed with:

$$\sum_{uses\ of\ v} (L\,(\text{add}) + L\,(\text{ld})) * f\,(\text{use}) +$$
$$\sum_{defs\ of\ v} (L\,(\text{add}) + L\,(\text{st})) * f\,(\text{def})$$

where $L\,(\text{add})$, $L\,(\text{ld})$, $L\,(\text{st})$ are respectively the latencies for an add, a load and a store operation. The expressions $f\,(\text{use})$ and $f\,(\text{def})$ are the frequencies of respectively the uses and definitions of v.

6 Evaluation

The goal of our experiments is to measure the performance of the introduced method. The results are compared with the standard early assignment (SEA) as described in [10] and to a dependence conscious early assignment (DEA) as described in [11] which tries to avoid the introduction of false dependencies.

The TTA processor selected for the experiments has 8 move busses, two integer units with a single cycle latency, two load-store units with a latency of 2 and two floating point units with latency of 3. The integer RF has two read and two write ports, the number of integer registers are varied during the experiments. The floating-point RF has also two read and two write ports, the number of floating point registers are also varied during the experiments. The boolean RF has 4 registers and two write ports. The TTA has one delay slot for branches.

The benchmarks used are kernels from the Livermore loops. We chose these benchmarks since the loops have little overhead code. In this way scheduling or allocation decisions for code from outside the loops have little effect on the results.

The results of the experiments are listed in table 1. The first column lists the number of registers used for the experiments. The second column gives the scheduling/allocation strategy, for RoD we used a threshold of 3. For each benchmark the II is given. Since the execution count of the benchmarks is highly dependent on the software pipelined loops, the II is a good indication of the scheduling/allocation efficiency.

Table 1: IIs for various software pipelining and register allocation strategies.

		Benchmarks				
# Regs	**Method**	LL1	LL2	LL6	LL7	LL10
32 int 32 fp	SEA	14	12	7	45	30
	DEA	5	11	5	14	26
	RoD	5	11	5	12	26
20 int 20 fp	SEA	14	12	7	45	30
	DEA	5	11	5	17	24
	RoD	5	11	5	12	26
16 int 16 fp	SEA	14	12	7	44	30
	DEA	5	11	5	23	29
	RoD	5	11	5	12	24
12 int 12 fp	SEA	14	13	10	44	48
	DEA	5	13	8	37	37
	RoD	5	11	5	14	33
10 int 10 fp	SEA	14	13	10	58	48
	DEA	7	19	8	42	46
	RoD	5	11	5	25	40

From the results of SEA we can conclude that ignoring false dependencies can have a tremendous negative impact on the produced code. When DEA is used, which tries to avoid false dependencies the results are much better, however it can not exploit the software bypassing ability of TTAs. Using RoD gives in general the best results, since it knows exactly step by step how the schedule is created and avoids those false dependencies that really influence the scheduling efficiency. Furthermore, it can exploit the software bypassing ability of TTAs. As a result spilling is not needed for many configurations for which DEA generates spill code.

As can be seen from the results, RoD achieves the same or better performance with fewer registers than the two other methods. Fewer registers result in a smaller chip area and in faster access to the registerfile which both are important for application specific processors such as TTAs.

7 Conclusions

In this paper we demonstrated that software pipelining increases the register pressure of loops. We discussed two methods to reduce the register requirements namely increasing the II and inserting spill code. Furthermore we have argued that instruction scheduling and register allocation should be integrated to obtain high performance.

Based on previous work we described a method that integrates register allocation and software pipelining. We also addressed the problem when the number of available registers is lower than the register requirements. We have shown that the new developed method can handle this by either increasing the II or by inserting spill code.

Although we discussed the new method in the context of TTAs, we believe that it is also applicable for superscalars and VLIW processors. A problem is however that these architectures do not suppress unnecessary register write-backs like software bypassing. This can be overcome by reserving a small set of registers for spilling. Alternatively, an extension for superscalar processors, as proposed in [17], could alleviate this problem. Lozano and Gao describe a hardware scheme that avoids the commits of variables that are only live in the reorder buffer. This is similar to software bypassing with the advantage that bypassed instructions do not have to be scheduled in the same cycle which relaxes the scheduling process and may result in even larger speedups.

The experiments showed that ignoring the introduction of false dependencies during register allocation results in lower performance then when taking them into account. The best results were achieved when register allocation and instruction scheduling are completely integrated, especially in cases where registers become a critical resource.

References

[1] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of ACM Sigplan'88 Conference on Programming Language Design and Implementation, ACM Sigplan Notices*, pages 318–328, Atlanta, June 1988.

[2] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63 – 74, San Jose, California, December 1994.

[3] Josep Llosa, Mateo Valero, Eduard Ayguade, and Antonio Gonzales. Hypernode reduction modulo scheduling. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 350 – 360, Ann Arbor, Michigan, November 1995.

[4] Philip Sweany and Steven Beaty. Post-compaction register assignment in a retargetable compiler. In *Proceedings of the 23th Annual International Symposium on Microarchitecture*, Orlando, Florida, December 1990.

[5] Johan Janssen and Henk Corporaal. Registers on demand: Integrated register allocation and instruction scheduling. In *Third Annual Conference of ASCI*, pages 61 – 67, The Netherlands, June 1997.

[6] Johan Janssen and Henk Corporaal. Registers on demand, an integrated region scheduler and register allocator. In *Poster Proceedings of the 7th International Conference on Compiler Construction*, pages 44 – 51, Lisbon, Portugal, April 1998.

[7] Alexandre E. Eichenberger and Edward S. Davidson. Stage scheduling: A technique to reduce register requirements of a modulo schedule. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 338 – 349, Ann Arbor, Michigan, November 1995.

[8] J. Dehnert and R. Towle. Compiling for the cydra 5. *The Journal of Supercomputing*, 7(1/2):181–228, May 1993.

[9] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the ACM SIGPLAN'82 Symposium on Compiler Construction.*, pages 98–105, Boston, June 1982. ACM, New York.

[10] K. D. Cooper P. Briggs, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. *SIGPLAN Notices*, 24(7):275–284, July 1989.

[11] Schlomit S. Pinter. Register allocation with instruction scheduling: A new approach. *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, 28(6):248–257, June 1993.

[12] N.J. Warter, G.E. Haab, and J.W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 170 – 179, Portland, Oregon, December 1992.

[13] Jan Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, February 1996.

[14] Jan Hoogerbrugge and Henk Corporaal. Register file port requirements of transport triggered architectures. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, Santa Clara, December 1994.

[15] Henk Corporaal. *Microprocessor Architectures; From VLIW to TTA*. John Wiley, 1997.

[16] Josep Llosa, Mateo Valero, and Eduard Ayguade. Heuristics for register-constrained software pipelining. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 250 – 261, Paris, France, December 1996.

[17] Luis A.Lozano C. and Guang R. Gao. Exploiting short-lived variables in superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 292 – 302, Ann Arbor, Mitchigan, December 1995.