Registers On Demand, an integrated region scheduler and register allocator

Johan Janssen and Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
Section Computer Architecture and Digital Systems
P.O. Box 5031, 2600 GA Delft, The Netherlands
e-mail: {johan, heco}@duteca.et.tudelft.nl

Abstract

Two of the most important phases of code generation for instruction level parallel processors are register allocation and instruction scheduling. Applying these two phase separately has major drawbacks like the introduction of false dependences, or a higher register pressure and thus more spill code. In this paper we present a new method which integrates register allocation and region scheduling, called Registers on Demand. We discuss register selection, spilling and the insertion of state preserving code around function calls. We have implemented the new method and compared its results with two early assignments methods. From the experiments we conclude that our method gives good results, in particular for applications for which registers are a critical resource. Therefore we propose that future ILP compilers should incorporate integrated instruction scheduling and register allocation.

Registers On Demand, an integrated region scheduler and register allocator

Abstract

Two of the most important phases of code generation for instruction level parallel processors are register allocation and instruction scheduling. Applying these two phase separately has major drawbacks like the introduction of false dependences, or a higher register pressure and thus more spill code. In this paper we present a new method which integrates register allocation and region scheduling, called Registers on Demand. We discuss register selection, spilling and the insertion of state preserving code around function calls. We have implemented the new method and compared its results with two early assignments methods. From the experiments we conclude that our method gives good results, in particular for applications for which registers are a critical resource. Therefore we propose that future ILP compilers should incorporate integrated instruction scheduling and register allocation.

1 Introduction

Modern optimizing compilers contain several optimization phases, including register allocation and instruction scheduling which have received widespread attention in past academic and industrial research [1, 2]. The separation between register allocation and instruction scheduling phases has its impact on the produced code, decisions made by one phase can have negative effects on the other. Also the order in which these two phases are applied is a point of dispute [3].

We speak of *early register assignment* when register assignment precedes instruction scheduling. Historically, the merit of early assignment was that processors had little instruction-level parallelism and few registers. So, whereas there was much to be lost by poor register allocation, there was little to be gained by good scheduling. The task of the register allocator is to assign the program's variables to a minimal number of registers. The selection of registers however may limit the possibilities to reorder instructions due to false dependences that are introduced with the reuse of registers.

When register assignment is done after instruction scheduling, i.e. *late register assignment*, the scheduler, uninhibited by anti-dependences, can generate an efficient schedule, but often increases the span of live ranges. This results in longer register lifetimes and more registers are needed. If there are not enough registers then the data must be written to memory, introducing spill code which itself also requires registers. Register allocation has to be repeated until no more spill code is introduced. The generated spill code is however inserted in the already scheduled code and therefore another scheduling step is required to reschedule the code. Rescheduling is usually applied on the basic block level and therefore the inserted code cannot be scheduled as efficient as normal code [4].

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 [5]. Both discussed methods have problems. In effect, the lack of communication and cooperation between the instruction scheduler and the register allocator can result in generating code that contains excess register spills and/or lower degree of instruction level parallelism than possible. Improved performance in one phase

can deteriorate the performance of the other phase, possibly resulting in poorer overall performance. Therefore instruction scheduling and register assignment should be integrated or the communication between the two phases must be improved.

In this paper we research a method that integrates instruction scheduling and register allocation, called Registers on Demand or in short RoD, and compare it with an advanced early register assignment method; both methods are implemented within the same compiler, making the measurements comparable. This paper is organized as follows. Section 2 discusses related work. Section 3 gives a description of the framework used for the experiments. Section 4 presents RoD which includes register selection, insertion of spill code and insertion of caller-saved and callee-saved code. Section 5 describes how to tune RoD using several heuristics. Section 6 evaluates the results of the performed experiments. The final section gives the conclusions.

2 Related work

Most methods for global register allocation are based the work of Chaitin [6, 7]; a number of improvements are published later on [8]. All these methods are based on graph coloring. Consequently, they assume that operations do not move relative to one another. However, in the presence of instruction scheduling, this could be a wrong assumption.

Lately some work is done on the interaction between the instruction scheduling and register allocation. The papers [9, 10, 11] describe a framework where register allocation is done before scheduling. In their approach the allocator is made aware of the code motions the scheduler wants to perform. With this information the allocator tries to avoid register assignments that prevent these code motions.

An approach that incorporates late assignments is discussed in [4]; if late assignment results in to high cost due to the insertion of extra code then early assignment is tried using the information obtained by the first step to avoid false dependences.

Bradlee [5] presented a method where the two phases, instruction scheduling and register allocation, are mixed. The initial passes of the instruction scheduler gives estimates of the scheduling cost given a certain number of registers. The scheduler is run locally with a limited number of registers and also with the maximum number of registers. These values are used to allocate a certain number of registers for each basic block.

Integration of the instruction scheduler and the register allocator has been considered as too complicated [5]. Another reason not to integrate the two phase is the lack of a global view for the register allocator. In spite of these disadvantages integration of the two phases is important for ILP processors, because it avoids the problems mentioned in the introduction. In [12] an integrated trace scheduler and allocator is presented. Their method uses a list scheduler to schedule the trace. It is not clear how spill code, and the insertion of caller-saved and callee-saved code is handled.

A different approach to integrating scheduling and allocation is taken in [13]; extra dependences and spill code are added to the program dependence graph such that excessive resource (both FU and register) requirements are reduced. This seems an interesting approach; however, it is not yet clear how this method performs in comparison to other methods.

Our approach is to schedule for Transport Triggered Architectures which has as a consequence that

cycle based list scheduling is not applicable therefore operation based list scheduling is used [14]. The basic idea in [12] resembles the approach presented here; however, we concentrate on region scheduling which exploits the most general scheduling scope [15]. Furthermore, using the same compiler framework as in [14] enables a true comparison with advanced early register assignment methods. Finally, we outline how to handle and tune the insertion of spill code, and of caller-saved and callee-saved code.

3 Architecture and compilation trajectory

For the experiments a processor template is used. With this template a processor, for an application specific task, can be designed quickly. The processors build, have a transport triggered architecture (TTA) which offers ILP. The available ILP is exploited by the software framework.

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 [15].

The heart of TTAs is the data transport network, containing one or more buses, to which FUs and register files (RFs) are connected. An example TTA is given in figure 1. The shown architecture has one load-store unit, two integer units (ALU), one floating-point unit (FPU), one instruction fetch unit, one 4-ported integer RF, a 2-ported floating-point RF and a 2-ported boolean RF. The data transport network has 4 buses; it can handle up to 4 data transports per cycle. This network does not have to be fully connected, although this relaxes the code generation process. 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 conditional execution by guarding data transports. For some applications it can be handy to have FUs dedicated to a specific task, for instance a Multiply-Add, AD-converter or an RS232 - interface [16]. These Special Functional Units (SFU) can also easily be integrated within the processor and exploited by the compiler. For more details on TTAs and prototype realizations we refer to [17, 18].

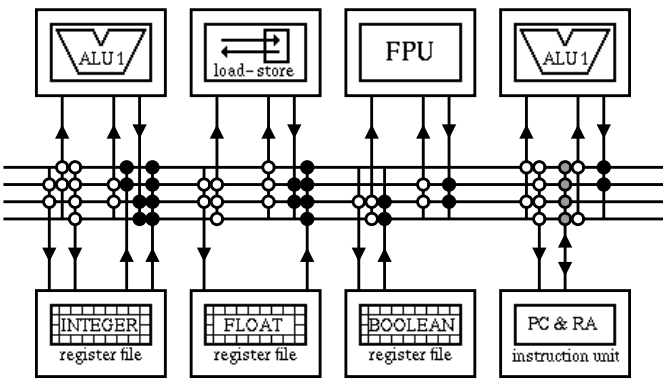

Figure 1: General structure of a TTA.

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 a FU. The result of the operation can be read from its result register. As an example we show how an `add` instruction translates into transports (moves):

```
add r3,r2,r1  ⇒ r1 → add.o; r2 → add.t
                add.r → r3
```

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`.

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 [15], see for example the following code fragment:

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

The scheduled version may turn out not to use register `r3`:

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

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.

The code generation trajectory is shown in figure 2. The GNU C/C++ compiler is used as compiler front-end; it is ported to produce sequential TTA code. The scheduler reorders this code and produces instruction level parallel TTA code. The scheduler uses, when available, profiling information. This profiling information is obtained from actual program runs and is used for speculative execution, controlling function inlining, loop unrolling and register allocation. The scheduler is parameterized: all information about the target TTA is read from a machine description file. More information on the scheduler can be found in [14].

4 ROD and region scheduling

To exploit ILP we use a region as a scheduling scope. Regions correspond to the body of a natural loop. Note that this body may contain arbitrary nested conditional statements. The region scheduler we use

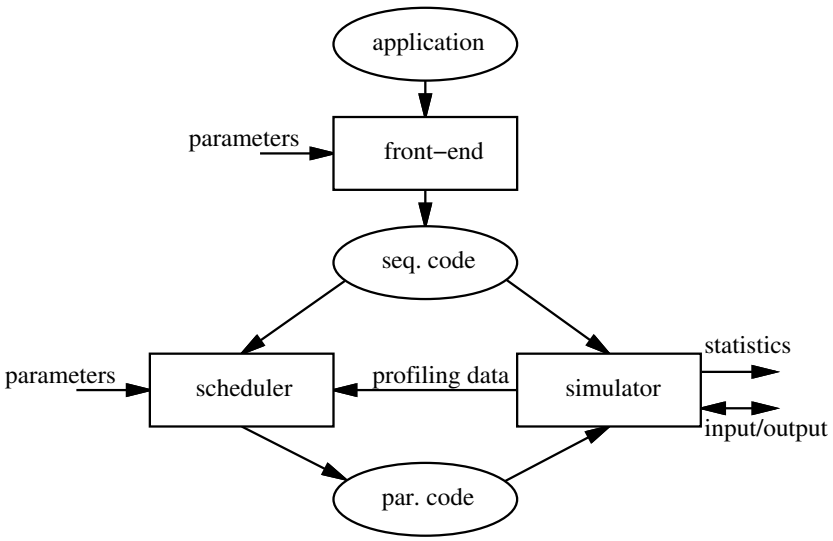

Figure 2: The compilation trajectory.

is inspired on the scheduler of Bernstein [19] and has extensions for multi-way branching, predicated execution and usage of profiling information. Regions are ordered for scheduling from innermost to outermost. Scheduling a basic block b consists of two steps, (1) basic block scheduling of b, and (2) importing operations from basic blocks that are reachable from b. The first step in combination with RoD is presented in section 4.1. The second step, importing operations, is discussed in section 4.2. Every register allocator must be able to generate spill code, this is the subject of section 4.3. How state preserving code is generated is discussed in the section 4.4.

4.1 Step 1: RoD and basic block scheduling

In this section we will give a short description of RoD. Before explaining the algorithm a few definitions are introduced:

Definition 4.1 *A variable is* defined *at a point in a program when a value is assigned to it.*

Definition 4.2 *A variable is* used *at a point in a program when its value is referenced in an expression.*

Definition 4.3 *A variable is said to be* live *at a point if it has been defined earlier and will be used later.*

Definition 4.4 *The* live range *of a variable is the execution range between definitions and uses of a variable.*

Definition 4.5 *The* def-use chain *of a variable is a directed graph that connects the operations that defines a variable to the operations that uses this variable. Note that a variable can have multiple definitions and uses.*

An example of such a def-use chain is given in figure 3; the def-use chain of variable x consists of 2 definitions and 2 uses.

The allocation/scheduling algorithm consists of the following steps. The first step is the analysis phase which consists of control flow analysis, data flow analysis (including detection of def-use chains)

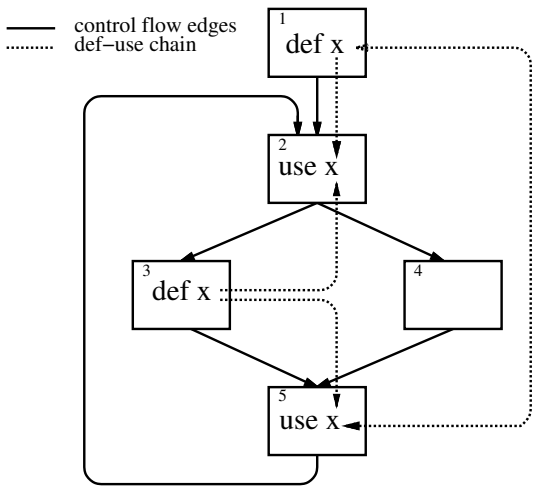

Figure 3: Example of a def-use chain.

and memory disambiguation. In the second step the data dependence graph is constructed. Then the regions are scheduled, starting with the innermost region. The basic blocks of the selected region are scheduled in a topological order: a basic block is scheduled after all its predecessors, in the same region, are scheduled. Operations are selected for scheduling if all their upwards exposed dependences are fulfilled. When multiple operations are available a heuristic is used to select one. Scheduling an operation consists of scheduling the moves of this operation. For each operation we must find a function unit 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.

```
proc ScheduleOperation(b, o, r, t)
beginproc
  for cycle = FirstCycle(t) to Max_Cycle do
    for each f ∈ U_set do
      if • Operation(t) ∈ OperationSet(f) ∧
         • IsUnitFree(f, cycle) ∧
         • TransportResourcesAvailable(t, cycle) ∧
         • SelectRegister(b, variable(t), t) ∧
         • ScheduleOperandMove(o, t, f) ∧
         • ScheduleResultMove(r, t, f)
      then
           return
      endif
    endfor
  endfor
endproc
```

Figure 4: Algorithm for scheduling an operation

In figure 4 the algorithm for scheduling an operation is given; b is the basic block which is scheduled; t, o and r are the trigger move, the operand move and the result move of the operation, respectively; f is a FU; U_{set} is the set of all FUs, and Max_Cycle is the maximum size of a basic block.

To ensure that variables with overlapping live ranges are not mapped onto the same registers some bookkeeping is necessary. Therefore to each cycle a Register Resource Vector (RRV) is added. This RRV indicates whether registers are free or not. Initially every basic block has one cycle and thus one RRV. If during scheduling additional cycles are required in a basic block then the contents of the RRV of the last cycle of the basic block is copied to the RRV of the added cycles.

Whenever a variable needs a register then the RRVs of all cycles that are in the live-range of the variable are checked for a common free register. The easiest way to do this is to intersect all involved RRVs. The resulting set then contains all registers that are free. The allocator can now 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 basic block should be updated. This is possible since we now have more information about the cycles in which the variable is live. The precise bookkeeping of RRVs has been described in [20].

4.2 Step 2: RoD and Region Scheduling

In this section the extensions needed to import operations are discussed. An operation is selected for importing to basic block *b* if it does not depend on operations located in a basic block which is reachable form *b* and all its upwards exposed dependences are full filed. To import an operation the scheduler has to check whether code duplication is necessary and where duplicates must be placed. The scheduler also has to check whether a code motion is speculative, and if so, whether it is possible to do it correctly. If needed and possible the speculative executed operation is guarded. More information about the instruction scheduler can be found in [14]. Figure 5 gives an example of a region code motion. In this figure we can distinguish four types of basic blocks.

- The destination basic block (b).
- The source basic block (b').
- Basic blocks where duplications have to be placed (D).
- Basic blocks between the source and destination basic blocks (I).

In our scheduling approach, we first schedule all instructions of basic block *b* and then the rest of the unscheduled code is examined to find instructions from other basic blocks that can be scheduled in this basic block. In the example it is possible to move an operation for basic block *b'* to basic block *b*. To obtain a correct program duplications have to be placed in the duplication basic blocks D.

Moving an operation over basic block boundaries can shorten or lengthen the live range of the variables. For variables whose live ranges shortens fewer basic blocks have to be checked and the chance of finding a register increases. The opposite holds for variables whose live ranges are stretched.

The live ranges of the operands of the operation can be shortened due to moving operations over basic block boundaries. To check whether a live range shortens we have to check whether this use is

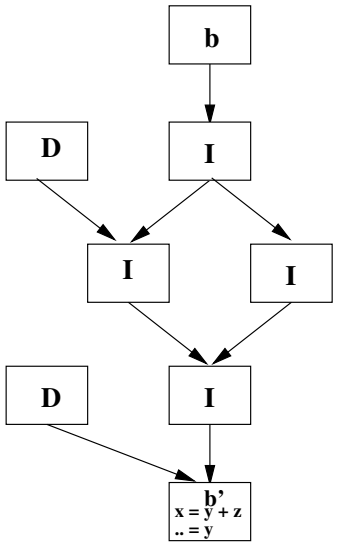

Figure 5: Code motion in a region scheduler

the last use of its live range. After this check the data flow information is updated accordingly. If the operand was already scheduled/allocated then also the information in the RRVs is updated. When in the example the scheduler decides to import the addition $x = y + z$ then the live range of z may shorten.

The live range of a definition will be stretched by the scheduler. Since we know in which basic blocks duplications are needed we know exactly over which basic blocks and cycles the live range of a definition will be stretched. This means that not only we have to check the RRVs of the basic blocks of the original live range but also the RRVs of basic blocks over which the live range is stretched. If in the example the scheduler decides to import the addition $x = y + z$ then the RRVs of the duplication basic blocks and the basic blocks between the source and destination basic block have to be checked for finding a free register for the variable x. Since the live range is stretched and thus more RRVs are checked the chance of finding a free register becomes smaller.

When the definition is already allocated the situation can arise that the allocated register is not free in one of the duplication blocks and thus the code motion can not be carried out. To prevent this we reset the allocation of the definition and we try to find a new register which is free in the complete new live range.

If scheduling fails in one of the duplication basic blocks all information must be restored to its old value, this includes the data-flow information, the contents of the RRVs and other resource information.

Scheduling the innermost regions first in combination with RoD has as the advantage that the most frequently executed basic blocks, which are often members of the innermost regions, are scheduled first. Consequently, possibly required spill code is pushed to outermost regions which are often less frequently executed.

4.3 Spilling

A processor has a limited number of registers therefore it is not always possible to find a register for each variable. In this case the scheduler/allocator has to insert spill/reload code. The goals of spilling in

combination with RoD are:

- The insertion of spill/reload code is not allowed in already scheduled code.
- The impact on the performance of the resulting code must be as small as possible.

The first goal is necessary because if spill/reload code is inserted in scheduled basic blocks then these blocks must be rescheduled which requires an extra scheduling step which might necessitate even more spill code, etc. This is the approach chosen for late assignment. Late assignment has additional problems in the context of TTA processors. The inserted load and store operations needed for the spill code can lead to invalid schedules as shown in the following example:

```
r1 → ld.t; ...          r1 → ld.t; ...
x → add.o; ...  ⟹      spill_address → ld.t;
ld.r → r3; ...          ld.r → add.o; ...
                        ld.r → r3; ...
```

If no register can be found for x then we have to insert a load. However simply inserting a load will result in an invalid schedule since the trigger register of the load is rewritten and thus the original load will load from a wrong memory location.

To prevent the insertion of spill/reload code in already scheduled code, RoD checks all RRVs that are spanned by the live-range of the variable for a free register. If no register is found it inserts for each definition and use of the variable spill or reload code. This will guarantee that spill/reload code is never inserted in scheduled code and no extra scheduling step is required. The process of spilling/reloading is illustrated in figure 6 where a fragment of a control flow graph of a program is given. The numbers in the upper left corner of the blocks indicate the basic block number. The number in parenthesis is the execution frequency of the block; this frequency is obtained by profiling. The left figure gives the situation before allocation. Assume that basic block 1 is already scheduled then basic block 2 will be picked first for scheduling/allocation since it has the highest execution frequency. For variable x register r1 is selected. Now suppose that there are no more free registers in block 4. This means that we can not find a register for the live range of variable y. The scheduler/allocator detects this when it tries to schedule the definition of y and it will insert spill code in the sequential unscheduled code in basic block 3 and reload code in basic block 4.

A spilled variable is written to a memory location. The memory address is an offset from the *frame pointer*. The address of the frame pointer is supplied by the calling procedure. Computing the memory address of the store or load may require an addition when the architecture does not support loads and stores with offsets (or when the supported offsets are somehow too small). The problem is however that the result of this addition is written to a register and it is this resource that was the bottleneck. A solution is to reserve some registers for address calculations [21]. The disadvantages of this method are that if no spilling was needed we have fewer registers at our disposal and some false dependences can be introduced which could have been avoided if the complete set of registers would have been available. Another disadvantage is that in case the scheduler could have used these reserved registers then spilling was not needed at all.

A better solution is to exploit the bypassing property of TTAs. This means that when we schedule reload-code, the addition and load must be scheduled in such a way that the result of the addition is

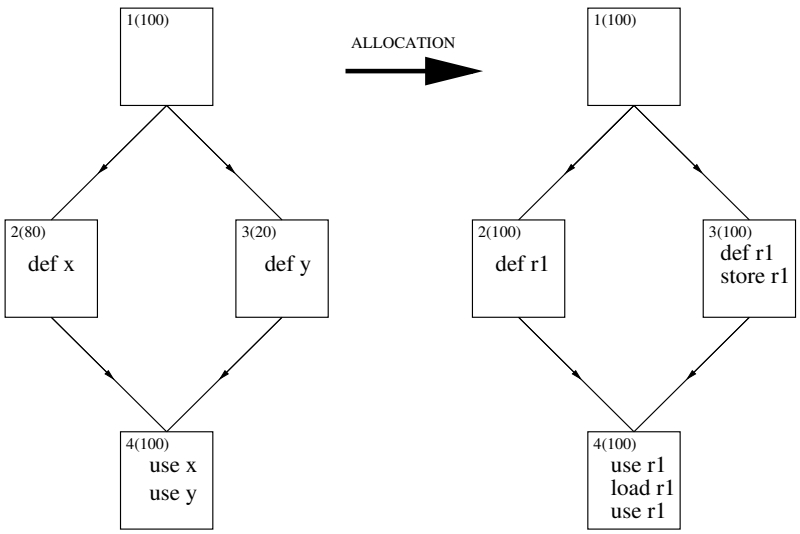

Figure 6: Spilling example

bypassed to the load, as a result no register is needed and therefore we do not need reserved registers for address calculations. Also the result of the load must be bypassed to the operand that needed spilling. Scheduling of an operation that has spill or reload code now consists not only of scheduling the operation but also of the inserted spill/reload code. So we make sure that the address calculation, the load or store, and the operation are scheduled in such a way that all variables are bypassed. To achieve this the allocator recognizes spill code. In figure 7 the most general spill code graph is shown. It shows the worst case situation for an operation with n operands and m results. In practice most operations have only one or two operands and one result; further it is not likely that all operands and results need spill/reload code. When loads and/or stores with address offsets are supported the graph complexity reduces substantially.

4.4 Caller-saved and Callee-saved Code

When a function is called, the state of the calling function must be saved so it can resume execution after the call. The state consists of the contents of the registers and the value of the program counter. The state is saved in memory on a known location; after the call the state is reloaded from memory and execution can resume.

In order to generate efficient code compilers divide the set of machine registers in caller-saved and callee-saved registers. Caller-saved registers are stored and reloaded by the caller and callee-saved registers are stored and reloaded by the callee. In figure 8 this is depicted for an architecture with 20 registers. Note, that only those registers have to be saved that are used in the function. Around the call to function *do_stuff* caller-saved code is inserted in function *foo* and at the start and end of function *do_stuff* callee-saved code is generated. Of course the function *foo* contains also callee-saved code since it is called by another function.

The task of the register allocator is to decide in which set, the caller or callee saved registers, the variable must be allocated in order to obtain an efficient schedule. A simple heuristic is to put the variable in a register where it influences as less as possible the dynamic execution frequency as determined by

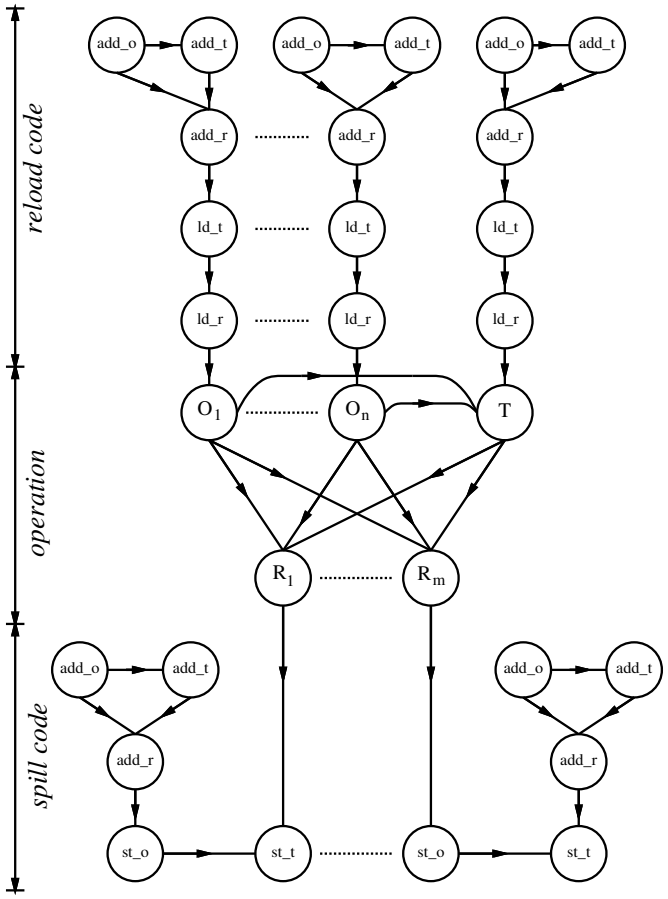

Figure 7: General spill/reload code dependence graph.

profiling. Consider the example in figure 9. The variable x is live in basic blocks 1 and 2. Since the call in block 2 is executed more frequently than the entry and exit basic blocks of the procedure it is more advantageous to allocate this variable in a callee saved register. For the other variable y which is live in basic blocks 1 and 3 it is more advantageous to allocate it in a caller saved register since basic blocks 1 and 4 are more frequently executed than basic block 3.

Every store or load needs an address calculation. The address is calculated in the same way as for spill/reload code: the address is the sum of the frame pointer and an offset. The last step is to save the program counter (the return address). This value is stored in the beginning of the function and reloaded at its exits.

Generating Caller Saved Code in combination with RoD

When a call operation is selected for scheduling, caller-saved code is generated. In order to do this the allocator must decide for which variables that are live across the call, caller-saved code should be generated. For already allocated variables there is no choice. For the variables that are not yet allocated the allocator uses profiling information to make the decision. When for all variables for which it is advantages to place them in caller-saved registers save/restore code has been generated, we set all caller-saved registers as occupied in the RRV of the cycle of the call. This is needed to avoid that other variables that are live over the call are assigned to a caller-saved register. These variables should be allocated in

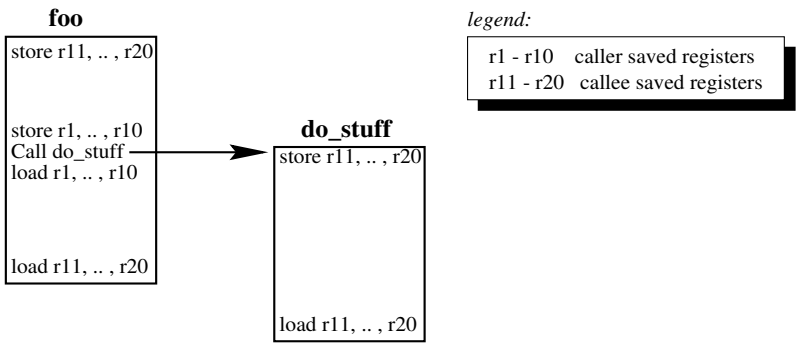

Figure 8: Placement of Caller/Callee saved registers.

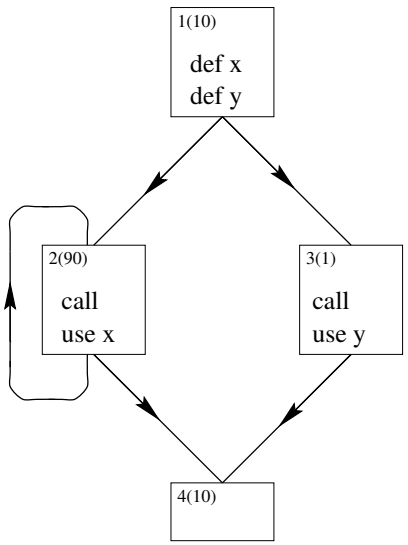

Figure 9: Heuristic for Caller/Callee saved registers.

callee-saved registers or they should be spilled.

Generating Callee Saved Code in combination with RoD

The callee saved code must be placed at the entry and exits of a procedure. The approach we used is schedule/allocate all code except the outermost region, so the entry and exit are not scheduled. Then the scheduled code is analyzed and for each used callee saved register a store is inserted in the entry basic block and loads are inserted in the exit basic blocks. All callee saved registers that are never used are set as occupied in all RRVs of all unscheduled basic blocks to avoid the situation that extra callee saved code must be generated in already scheduled code. The disadvantage is of course that in the basic blocks of the outer region the set of free registers is limited to the caller saved registers and the saved callee saved registers. This is usually not a severe problem; when there are many registers the probability of finding a register is high, since at least all caller saved register will be available. When there are only a few registers, all callee-saved code registers will be used in the innermost regions and thus all registers will be available.

5 Tuning RoD

In order to generate high quality ILP code heuristics are needed to guide the code generation process. These heuristics are needed for:

- Operation selection.
- Register selection.
- Selection of other resources, eg. move bus allocation, FU allocation.

In this section the heuristics for operation and register selection are presented. For the methods used to select the other resources we refer to [14].

Heuristics for operation selection

The performance of a list scheduler depends on the order in which instructions are selected, this holds also for an operation based list scheduler. For each operation we compute its as-soon-as-possible (ASAP) and its as-late-as-possible (ALAP) limits. ASAP gives the first cycle where the operation can be scheduled and ALAP the latest cycle where the operation can be placed without increasing the critical path length of a basic block. The difference between the two limits is called the *slack*. Our basic block instruction scheduler selects operations with a small slack because these operations have fewer cycles where they can be placed without increasing the critical path and are thus more critical.

For importing operations in a basic block b another heuristic is used. The order in which operations are tried from reachable basic blocks(b') should not only depend on the slack of the operation but also on the probability that control flow will reach b' after executing b. Since basic blocks have variable critical paths the slack is normalized with respect to the critical path length of the basic block. The heuristic for importing operations is the product of the probability that b' will be executed after b and the normalized slack of the operation.

Heuristics for register selection

Not only the order in which operations are selected is important to obtain efficient code but also the way registers are selected. We have already seen that allocating a variable in a caller or callee saved register can have a major impact on the dynamical code size. Therefore decisions have to be made where to allocate a variable. There a three categories: (1) allocate the variable in a caller saved register, (2) allocated the variable in a callee saved register, or (3) spill the variable. To make a good decision we compute for each variable three cost functions, one for each category. These cost functions indicate the potential performance loss caused by inserting operations:

Caller-saved costs: For each call in the live range of a variable we compute the costs of inserting caller-saved code. The cost function per call consists of the latencies of saving a register; the latency of an addition and the latency of a store, and the latencies of restoring a register which are the latencies of an addition and a load. The total latency is multiplied with the frequency of the call. We sum the costs of all calls in the live range of the variable. The caller saved cost for a variable v becomes:

$$C_{caller_saved}(v) = \sum_{call\ in\ live\ range(v)} (2 * L(\text{add}) + L(\text{ld}) + L(\text{st})) * f(call)$$

Callee-saved cost: The callee saved costs reflects the cost for adding callee saved code. The latency of the store code and the restore code are added and multiplied with the frequency of the first basic block of the function, in formula:

$$C_{callee_saved} = (2 * L\,(\text{add}) + L\,(\text{ld}) + L\,(\text{st})) * f\,(entry\ basic\ block)$$

Spill costs: The spill costs for a variable v are computed using the following formula which reflects the extra latency that is introduced by spilling:

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

A variable is allocated in a caller saved register when its caller-saved costs are lower then its callee-saved costs. If however no caller-saved register can be found we have to decide whether to spill the variable or the allocate it in a callee-saved register. This decision is based on a comparison between the callee-saved costs and the spill-cost of the variable. The category which minimizes the cost is selected. The decision rules are given with the following relations:

$$\begin{aligned}
r_{caller_saved}\,(x) &= ((C_{caller_saved}\,(x) < C_{callee_saved}\,(x)) \vee R_{callee_saved} = \phi) \wedge R_{caller_saved} \neq \phi \\
r_{callee_saved}\,(x) &= ((C_{callee_saved}\,(x) \leq C_{caller_saved}\,(x)) \vee \\
&\quad (R_{caller_saved} = \phi \wedge (C_{callee_saved}\,(x) \leq C_{spill}\,(x))) \wedge R_{callee_saved} \neq \phi) \\
r_{spill}(x) &= \overline{r_{caller_saved}(x)} \wedge \overline{r_{callee_saved}(x)}
\end{aligned}$$

where R_{caller_saved} is the set of all free caller-saved registers and R_{callee_saved} is the set of all free callee-saved registers. If $r_{caller_saved}\,(x)$ is true, then a caller saved register is selected for the variable x, if $r_{callee_saved}\,(x)$ is true, a callee saved register is selected, otherwise the variable is spilled.

Tuning the amount of generated spill code

As already discussed in this section we start scheduling the innermost regions first which results in a larger set of free registers and thus fewer spill code or false dependences. However in practical code the inner regions are not always the most frequently executed regions. In this case we would waste registers to pieces of code that are almost never executed. To account for this effect we use an extra heuristic for r_{spill}. For a to be allocated variable x we determine the maximum number of unallocated live ranges with higher spill costs than x, that are simultaneously live with x; this quantity is called $N_{live_high_cost}$. If this quantity is higher than the total number of available free registers for x then the variable x is spilled, in formula:

$$r_{spill}(x) = N_{free_regs} < N_{live_high_cost}\,(x)$$

where

$$N_{live_high_cost}(x) = \max_{\forall\, c \in\, live_range(x)} |\{v | v \text{ not allocated} \wedge v \text{ live in cycle } c \wedge (C_{spill}(v) > C_{spill}(x)\}|$$

If r_{spill} is true the variable is spilled. Spilling x instead of spilling another variable with higher spill cost leads to a better total performance. If the variable is not spilled the above discussed procedure is used to select a register.

During the experiments we noticed that importing operations works too greedy, which has as a result that to many live-ranges were stretched and no more register where available in the basic blocks between the source and destination basic blocks. This leads to excessive spill code. To limit the greediness we added the following heuristic to decide whether to import an operation x or not:

$$r_{import}(x) = N_{Iblocks} \leq a + \exp\left(b * N_{available_regs}(x)\right)$$

where a and b are empirically to be determined constants, $N_{Iblocks}$ represents the number of basic blocks between the source and destination basic blocks (the intermediate basic blocks), and $N_{available_regs}$ is a measure for the number of available registers:

$$N_{available_regs}(x) = N_{free_regs} - N_{live_high_cost}(x)$$

This heuristic means that importing operations from far ahead is allowed only if sufficient registers are available. Figure 10 plots above heuristic, using the empirically found values $a = -1$ and $b = 0.6$.

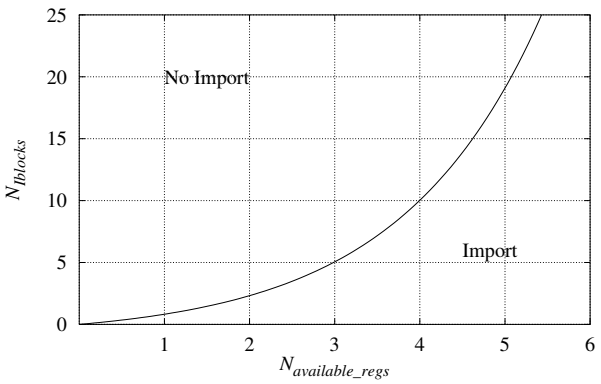

Figure 10: Greediness heuristic: r_{import}.

6 Experimental evaluation

In this section the result of our method are presented and evaluated. The goal of our experiments is to measure the performance of the new introduced scheduler with integrated register allocation.

For the experiments we selected an architecture that has moderate cost and high performance for all benchmarks. To prevent being biased towards a certain TTA configuration, this selection has been

Table 1: The target TTA used for the experiments.

Resource	Quantity
# Buses	8
FUs	
• # Integer FUs	3
• # Floating-point FUs	1
• # Load-Store FUs	2
Integer register file	
• # register	> 10
• # write ports	4
• # read ports	4
Floating-point register file	
• # registers	48
• # write ports	1
• # read ports	1
Boolean register file	
• # registers	4
• # write ports	2
Immediate units	2

made by using the automatic synthesis tool, see [17]. This tool is applied to a set of benchmarks. The resources of the selected architecture are listed in table 1. The load-store FUs have a latency of two cycles, the floating-point unit has a latency of three and jumps have one delay slot. The integer FUs have a single cycle latency. The number of read ports of the boolean register file is not listed because booleans are read implicitly by *guards*. These guards are used to support conditional data transports.

We restrict the experiments to mainly integer type of code. However, all methods can be applied to floating-point intensive code as well. The following 15 benchmarks are used: *a68, bison, compress, dhrystone, diff, gzip, sed, sieve, sort, sum, tr, unexpand, uniq, virtex,* and *wc*.

To measure the performance of the proposed scheduling/allocation method the benchmarks are first compiled to sequential move code and simulated with representative data-sets. Then the sequential code is scheduled for the target architecture of table 1. The last step combines the information from the parallel code with the profiling data in order to obtain dynamic information such as the cycle count.

The results are compared with two early assignment approaches. The first early assignment approach is a standard optimistic register allocator as described in [8], which we will call *standard early assignment* or in short SEA. The second approach uses also the optimistic register allocator but it uses also data dependence information. Doing this will reduce the chance of introducing false dependences. This method is described in [11, 10, 9], we will call this method *dependence conscious early assignment* or in short DEA.

In the first experiment we compared RoD with SEA. For each benchmark we measured the performance of 8 models. These models only differ in the number of integer registers. The largest model supports an unlimited number of registers. The smallest model has 10 registers. This has two reasons.:

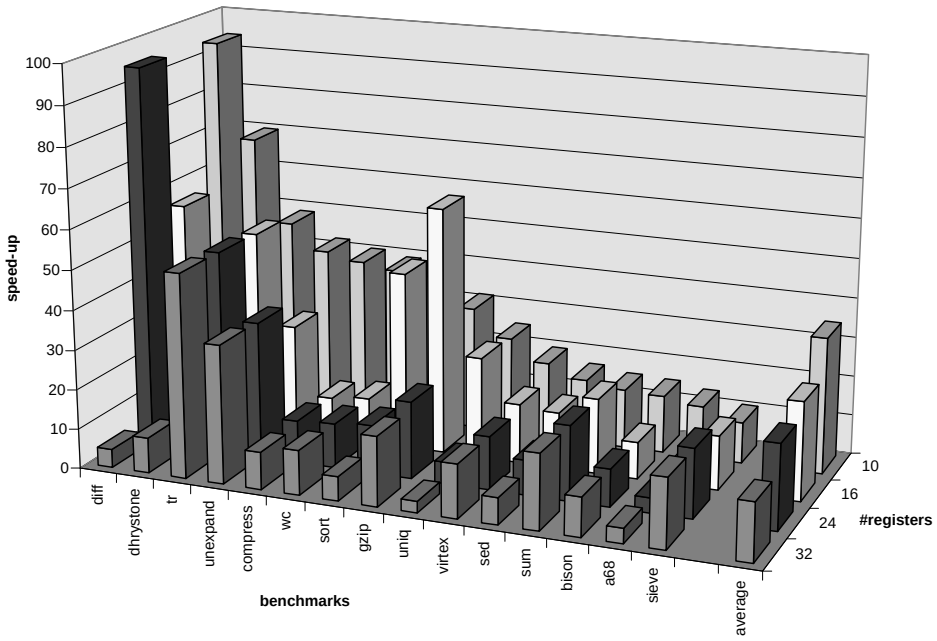

Figure 11: Speedup of RoD compared with SEA.

from the set of registers, 7 registers are reserved as special registers (for the stack pointer, frame pointer and parameter passing); secondly for fewer than 10 registers the early assignment approaches can not always find a solution; RoD has however no problems with lower register numbers.

Measurements of 4 models are shown in figure 11 (results of the other models can be found in the appendix). Along the x-axis the names of the benchmarks used are listed. The number of registers is placed along the y-axis. The speedup of RoD compared with SEA is listed along the z-axis. For each number of integer registers the relative speedup of RoD compared to the SEA is computed. As can be observed from figure 11 the speedup is dramatically. The main reason for this large speedup is that SEA does not undertake any attempt to avoid false dependences.

In the second experiment we compared RoD with DEA, the results are shown in figure 12. As we can see from figure 12 the speedup is smaller than in figure 11, but still substantially for lower register numbers. DEA tries to avoid false dependences and apparently it succeeds very well; however, when registers become a critical resource RoD outperforms DEA. In this situation RoD does a better job on avoiding false dependences; furthermore it can reuse freed registers due to software bypassing. And finally, the spill-code needed is placed in less critical parts of the code. When register are not so critical sometimes RoD has a negative speedup, this is most likely due to decisions made by the heuristics. For some odd reason the number of cycles sometimes decreases when the number of register decreases. Lately we achieved better speedup by tuning the heuristics; we expect that further tuning will result in even better figures in the near future.

We were also curious about how severe reducing registers hurts performance. This data is shown in figure 13; it shows the cycle count increase relative RoD in a configuration with an unlimited number of registers; all benchmarks are averaged. SEA gives, as expected, the largest cycle count increase for all models. For models with enough registers RoD performs slightly better than DEA. However when register become a critical resources RoD also outperforms DEA. It is also interesting to note that only

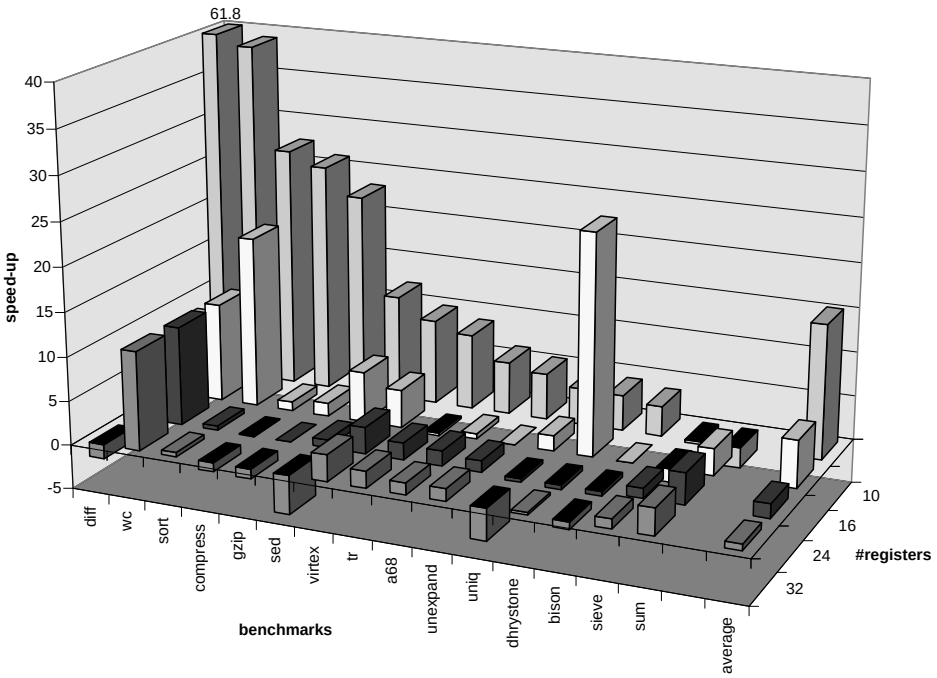

Figure 12: Speedup of RoD compared with DEA.

for 20 or fewer registers the number of register really seems to care. If we had taken more benchmarks with greater register pressure then the results for RoD would be even better.

TTAs are originally proposed for application specific purposes; therefore it is also interesting to know how many registers can be saved while the performance stays constant. Fewer registers results in a smaller chip area and in faster access to the register-file, both are important for application specific processors. From figure 13 we can conclude that RoD always outperform SEA, even when SEA has 32 registers and RoD only 10. In table 2 a comparison is made between RoD and DEA to find out what the savings are. The results are listed in table 2. We see that RoD needs substantially fewer registers than DEA and therefore is the best solution for embedded systems.

Table 2: Register savings of RoD compared to DEA with equal performance.

# registers for RoD	# registers for DEA	# increase[%]
10	16	60 %
12	18	50 %
16	21	31 %
20	> 32	- %

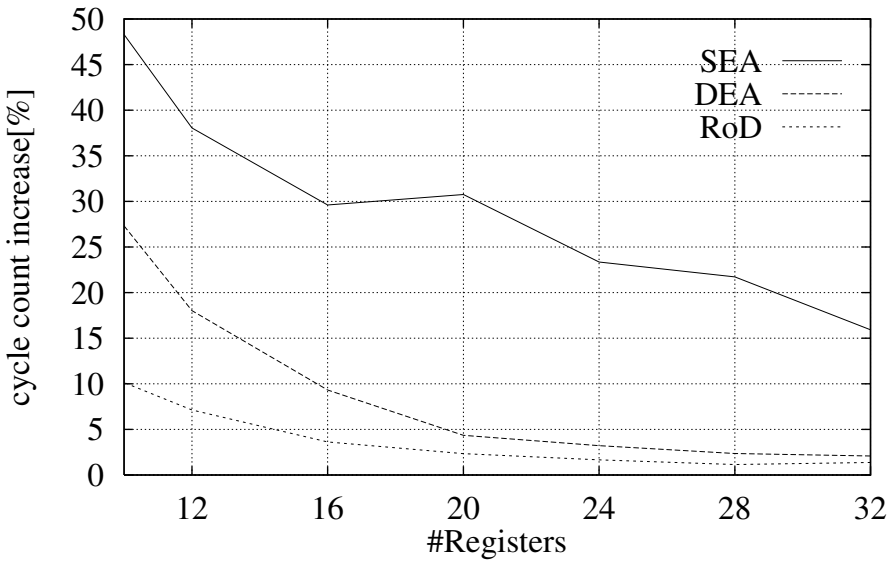

Figure 13: Cycle count increase of RoD, SEA and DEA.

7 Conclusions

We have argued that instruction scheduling and register allocation in the context of ILP processors should be integrated. A new method, which integrate instruction scheduling and register allocation is presented. This method, called RoD, can be applied to as well a basic block scheduling scope and a region scheduling scope. We discussed the general principle of RoD, the insertion of spill code, the insertion of caller/callee-saved code and several heuristics. The method is compared with two early assignment approaches, a standard early register allocator followed by a region scheduler (SEA) and a more advanced early register allocator followed by a region scheduler (DEA). All methods are implemented within the same compiler making reliable comparisons possible. The experiments showed a great performance increase when we compare RoD with SEA; speedups of 30% or more are achieved. Compared to DEA the results are less pronounced but when registers become scarce the improvements can be substantial, up to 15% on average for 10 registers.

Although we discussed the new method in the context of TTAs we believe that it is also applicable for superscalars and VLIW. A problem is however that these architectures do not suppress unnecessary register write-backs. This can be overcome by reserving a small set of registers for spilling. Alternatively, an extension for superscalar processors, as proposed in [22], could alleviate this problem. Lozanao 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 instructions do not have to be scheduled in the same cycle which relaxes the scheduling process and may result in even larger speedups.

We believe that our method has great potential. It is to be expected that in the future more aggressive scheduling techniques will be applied which exploit more ILP. As a consequence the register pressure increases. The strength of RoD is that it performs very good when registers are scarce.

Lately we also achieved improvements by tuning the heuristics, it is to be expected that further research is this area will result in even better results. It is also worth to mention that some benchmarks

profit more from other combinations of heuristics. Using a branch and bound technique which uses different heuristics will certainly give better results.

For embedded systems the number of registers is a critical resources, less registers means smaller chip area and faster access times. When the register access is in the critical path this will lead to shorter cycle times and better performance. In embedded systems not every register is equivalent accessible, which means in the context of TTA that different RFs are connected to different buses. With early assignment it is not possible to tackle this problem; we feel that RoD is capable to allocated registers also for these processors.

As a side effect of this research a method for inserting on the fly operations is developed while taking into account data dependences. This is not only needed for spilling and the insertion of caller- and callee saved code but can for instance also be used to make on the fly decisions like strength reduction and covering of the data dependence graph by more complex operations [23].

In the future we will extend the method so that it can also be used with software pipeling. Another topic is register allocation over function calls; this so-called inter-procedural register allocation can result in less caller and callee saved code. We will also perform research in the area of irregular architectures since we feel that this method is far better suited for these type of architectures than early or late assignment.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesly Series in Computer Science. Addison-Wesly Publishing Company, Reading, Massachusetts, 1985.

[2] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesly Series in Computer Science. Addison-Wesly Publishing Company, Wokingham, 1995.

[3] Philip Sweany and Steven Beaty. Post-compaction register assignment in a retargetable compiler. In *MICRO-23*, Orlando, Florida, December 1990.

[4] Thomas S. Brasier, Philip H. Sweany, Steven J. Beaty, and Steve Carr. Craig: A practical framework for combining instruction scheduling and register assignment. In *Parallel Architectures and Compilation Techniques (PACT '95)*, 1995.

[5] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating Register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International ASPLOS Conference.*, pages 122–131, Santa Clara, California, April 8–11 1991. ACM. Issue 26(4),1991 of SIGPLAN Notices.

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[7] 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.

[8] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[9] Wolfgang Ambrosch, M. Anton Ertl, Felix Beer, and Andreas Krall. Dependence-conscious global register allocation. In Jürg Gutknecht, editor, *Programming Languages and System Architectures*, pages 125–136, Zürich, 1994. Springer LNCS 782.

[10] Cindy Norris and Lori L. Pollock. Register allocation over the program dependence graph. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[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] Stefan M. Freudenberger and John C. Ruttenberg. Phase ordering of register allocation and instruction scheduling. In *Code Generation — Concepts, Tools, Techniques, Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, 20-24 May 1991*, Workshops in Computing, pages 146–172. Springer-Verlag, 1991.

[13] D. Berson, R. Gupta, and M.L. Soffa. Hare: A hierarchical allocation of registers. Technical Report 95-06, University of Pittsburgh, Computer Science Department, February 1995.

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

[15] Jan Hoogerbrugge and Henk Corporaal. Register file port requirements of transport triggered architectures. In *MICRO-27*, Santa Clara, December 1994.

[16] Herman Adriani, Françoise Harmsze, and Henk Corporaal. The utilization of a fully configurable microprocessor development environment for rapid VHDL prototyping and implementation of 'C'-based algorithms. In *Proceedings of ProRisc workshop*, Mierlo, The Netherlands, November 1996.

[17] Henk Corporaal and Reinoud Lamberts. TTA processor synthesis. In *First Annual Conf. of ASCI*, May 1995.

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

[19] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of ACM Sigplan'91 Conference on Programming Language Design and Implementation*, pages 241–255, Toronto, Canada, June 1991.

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

[21] Daniel M. Lavery et al Pohua P. Chang. The importance of prepass code scheduling for superscalar and superpipelined processors. *IEEE Transactions on Computers*, 44(3):353–370, March 1995.

[22] 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. MICRO28 proceedings.

[23] Marnix Arnold and Henk Corporaal. Data Transport Reduction in Move Processors. In *Proceedings of the third annual conference of the Avanced School for Computing and Imaging*, pages 68–75, June 1997.

A Appendix

This appendix presents the results of all benchmarks in combination with all models. Table 3 gives the speedup of RoD compared with SEA; table 4 compares RoD with DEA.

Table 3: Speedup of RoD compared with SEA.

	# registers							
program	unlimited	32	28	24	20	16	12	10
a68	1.8%	3.8 %	3.5 %	3.8 %	3.5 %	2.9 %	8.5 %	13.1%
bison	7.2%	9.8 %	9.5 %	9.5 %	10.3 %	9.3 %	12.9 %	14.5%
compress	10.4%	9.4 %	12.4 %	10.6 %	9.5 %	10.1 %	17.5 %	40.2%
dhrystone	10.9%	8.9 %	8.9 %	8.9 %	5.8 %	10.3 %	61.0 %	69.5%
diff	7.2%	4.6 %	85.5 %	95.1 %	136.3 %	55.8 %	57.0 %	93.8%
gzip	13.3%	17.5 %	19.4 %	19.4 %	20.3 %	62.0 %	30.4 %	31.2%
sed	-0.2%	6.7 %	3.9 %	8.8 %	10.1 %	14.1 %	13.0 %	16.1%
sieve	20.0%	17.4 %	17.4 %	17.4 %	13.6 %	13.6 %	13.7 %	10.3%
sort	1.4%	6.1 %	9.8 %	12.1 %	43.6 %	44.6 %	40.5 %	38.4%
sum	18.9%	18.9 %	18.9 %	18.9 %	18.9 %	18.9 %	18.9 %	14.9%
tr	51.4%	51.3 %	51.4 %	51.1 %	51.0 %	50.6 %	48.0 %	48.3%
unexpand	35.0%	34.8 %	34.0 %	34.4 %	34.4 %	27.5 %	53.7 %	41.9%
uniq	7.4%	2.9 %	5.7 %	5.4 %	30.3 %	25.5 %	26.6 %	24.5%
virtex	13.6%	13.5 %	13.4 %	13.4 %	13.3 %	14.9 %	13.7 %	19.3%
wc	11.2%	11.2 %	11.2 %	11.2 %	11.2 %	11.2 %	11.2 %	38.9%
average	14.0 %	14.5%	20.3 %	21.3 %	27.5%	24.8 %	28.4 %	34.3 %

Table 4: Speedup of RoD compared with DEA.

	# registers							
program	unlimited	32	28	24	20	16	12	10
a68	-0.9%	1.3 %	1.1 %	1.7 %	0.8 %	0.6 %	2.8 %	5.9%
bison	-0.4%	-0.8 %	-0.6 %	-0.5 %	-0.7 %	-0.0 %	0.7 %	3.4%
compress	-0.8%	-1.0 %	1.3 %	-0.1 %	-1.6 %	1.0 %	0.5 %	25.9%
dhrystone	2.0%	0.3 %	-1.4 %	-0.5 %	1.0 %	24.9 %	1.5 %	4.0%
diff	0.5%	-1.6 %	-1.8 %	-0.3 %	2.2 %	8.3 %	64.1 %	61.8%
gzip	6.7%	-1.1 %	-0.5 %	-0.0 %	2.0 %	1.5 %	14.5 %	22.9%
sed	-5.5%	-4.4 %	-1.5 %	0.9 %	5.4 %	5.7 %	5.9 %	11.8%
sieve	1.1%	1.1 %	1.1 %	1.1 %	1.1 %	-1.5 %	1.9 %	-0.3%
sort	-3.1%	0.5 %	2.2 %	0.5 %	1.8 %	19.4 %	24.9 %	27.3%
sum	3.0%	3.0 %	3.0 %	3.5 %	2.5 %	3.0 %	3.0 %	-2.3%
tr	2.0%	1.9 %	2.0 %	1.9 %	-0.2 %	-0.3 %	-2.2 %	8.5%
unexpand	0.7%	1.4 %	1.0 %	1.3 %	1.3 %	-0.0 %	5.8 %	5.2%
uniq	-0.7%	-3.6 %	-1.7 %	-0.4 %	-0.1 %	1.7 %	4.4 %	4.1%
virtex	2.7%	3.0 %	2.9 %	3.0 %	2.9 %	4.3 %	4.5 %	9.6%
wc	11.2%	11.2 %	11.2 %	11.2 %	11.2 %	11.2 %	11.2 %	38.9%
average	1.2 %	0.7%	1.2 %	1.6 %	2.0%	5.3 %	9.6 %	15.1 %