Registers On Demand: Integrated register allocation and instruction scheduling

Johan Janssen

Department of Electrical Engineering
Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands

Henk Corporaal

Department of Electrical Engineering
Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands

Abstract

Two of the most important phases of code generation for instruction level parallel processors are register allocation and instruction scheduling. Applying one phase before the other has in both cases its drawbacks. In this paper we present a new method which integrates register allocation and 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 results are presented. From the experiments we conclude that our method gives good results in particular for applications for which registers are a critical resource.

Keywords: register allocation, instruction scheduling, instruction level parallelism, TTA.

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 phases can have negative effects in 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 a 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, *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, this is called spill code. The spill code generated is however inserted in the already scheduled code and therefore another scheduling step is required to reschedule the code.

Most methods for register allocation are based on graph coloring [4, 5], a number of improvements are published on that method [6]. Lately some work is done on the communication between the instruction scheduling and register allocation. The papers [7, 8, 9] 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 incorporate late assignments is discussed in [10]; if late assignment fails early assignment is tried using the information obtained by the first step to avoid false dependencies. All mentioned methods are based on graph coloring. Graph coloring assumes that operations do not move relative to one another. In the presence of instruction scheduling, this is a wrong assumption.

In [11] an integrated trace scheduler and allocator is presented. Their method uses a list scheduler to schedule the trace. Our approach is to schedule for Transport Triggered Architectures which has as a consequence that list scheduling is not applicable therefore operation scheduling is used [12]. The basic idea in [11] resemble the ideas presented here, although we concentrate on basic block scheduling. Furthermore using operation scheduling results in more and more complex bookkeeping than described in [11]. It is also not clear how spilling and insertion of caller-saved and callee-saved code is done.

In this paper we will discuss a method that integrates instruction scheduling and register allocation, the method is called Registers on Demand or in short RoD. This paper is organized as follows: in section 2 a description of the framework used for the experiments is given in the next section RoD is presented which includes register selection, insertion of spill code and insertion of caller-saved and callee-saved code. In section 4 the results of the experiments are presented and the final section gives the conclusions and a description of future work.

2 Architecture and Compilation Trajectory

For the experiments we used a TTA processor template. With this template a processor, for an application specific task, can be designed quickly. The processors build, have a transport triggered architecture. This architecture and its code generation trajectory are explained below.

Transport Triggered Architectures
TTAs resemble VLIW (very large instruction word) architectures. TTAs and VLIWs 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 [13].

A TTA consists of FUs and register files (RFs) connected by a data transport network. An example TTA is given in figure 1. The shown architecture has one load-store unit, one integer units (ALUs), 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.

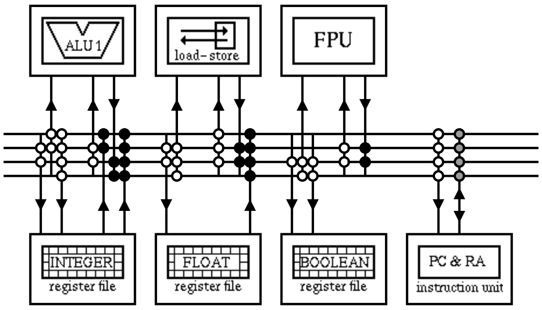


Figure 1: General structure of a TTA.

Each FU contains operand and result registers. Performing an operation with n (source) operands consists of moving $n-1$ operands and one trigger to the operand registers respectively trigger register of a FU that is capable to perform the operation. The result of the operation can be read from the result register. As an example we show how an add-instruction translates into transports:

```
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`.

Bypassing is one of the advantages of TTAs above more traditional architectures. Using bypassing the compiler can eliminate the need of some RF accesses, 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 is:

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

The resulting schedule saved one register. However using early assignment the 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 for other live-ranges and perhaps could prevent spilling or the introduction of a false dependency. For more details on TTAs and prototype realizations we refer to [14].

Compilation Trajectory
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 parallel TTA code using operation scheduling. The scheduler uses, when available, profiling information. This profiling information is obtained from the simulator and used for 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. The scope of the scheduler is a basic block. More information on the scheduler can be found in [12].

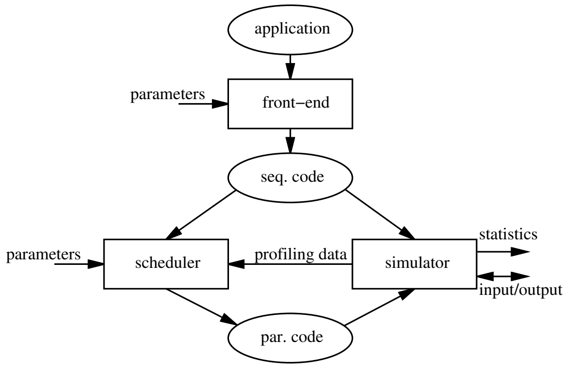


Figure 2: The compilation trajectory.

3 Registers On Demand

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

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

- A variable is *used* at a point in a program when its value is referenced in an expression.
- A variable is said to be *live* at a point if it has been defined earlier and will be used later.
- The *live range* of a variable is the execution range between definitions and uses of a variable.

In this section the following subjects are discussed: How to select a register for a variable (section 3.1), how to generate spill code (section 3.2) and how to handle caller-saved and callee-saved code (section 3.3).

3.1 Register Selection

The allocation/scheduling algorithm works in the following way. The first step is the analysis phase which consists of control flow analysis, data flow analysis and memory disambiguation. In the second step the data dependence graph is constructed. Then the basic blocks are scheduled in order of their execution frequency. Operations are selected for scheduling if all their upwards exposed dependencies are fulfilled. When multiple operations are available a heuristic is used to select one. Scheduling of an operation consists of scheduling of 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. As an example, in figure 3 the algorithm for scheduling an operation is given where b is the basic block which is scheduled; t, o and r are respectively the trigger move, the operand move and the result move of the operation; f is a FU; U_{set} is the set of all FUs and MAX_CYCLE is the maximum size of a basic block.

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

Figure 3: Algorithm for scheduling an operation

In our approach a register is assigned to a variable when a move is scheduled which performs a read or a write of this variable. To obtain an efficient schedule in terms of the number of dynamic executed instructions the most frequently executed basic blocks are selected for scheduling first. This gives a larger scheduling freedom in the more frequently executed parts of a program. The number of registers that are available is larger in the first scheduled basic blocks so spill code due to a lack of registers is driven to the less frequently executed basic blocks and false dependencies are less likely to occur.

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 Table (RRT) is added. This RRT indicates if a register is free in that cycle or not. Initially every basic block has one cycle and thus one RRT. If during scheduling additional cycles are required in a basic block then the contents of the RRT of the last cycle in the basic block is copied to the RRT of the new cycles.

The RRTs do the bookkeeping of the available free registers. Whenever a variable needs a register then the RRTs 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 RRTs. 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 RRTs of the basic block should be updated. This is possible since we have now more information about the cycles in which the variable is live.

We now show the concept of the bookkeeping of the RRTs in a small example. Consider figure 4, the unscheduled code consists of an addition and a multiply. The operand move of the multiply has a variable which is already allocated to register r0. The basic block is partly scheduled; *move* means that the slot is filled with a scheduled move; register r0 is set as not free since we do not know in which cycle the operand move will be scheduled, the worst case is in the last cycle. Further assume that the operand and trigger of the multiply and the trigger of the addition are the only uses of their definitions. In step 2 the trigger and operand of the addition are scheduled. Since the trigger is the first move that use or defines variable x, a register has to be selected; in this example register is r1 is chosen. This has as a consequence that in cycle 0 this register is not free anymore. In the next step we schedule the result move of this operation. Since r1 is free from cycle 2 to the end of the basic block we can select r1 for variable y. As a consequence also the trigger move of the multiplication is assigned to r1. Since we do not know in which cycle this trigger move will scheduled we must assume that any cycle is possible so we claim r1 in cycle 2 and higher. In the 4th step the trigger of the multiplier is scheduled. As can be seen easily the result of the addition can be bypassed to the trigger of the multiplier. This has as a consequence that register r1 is not needed anymore and the RRTs must be updated accordingly. In the last step the operand and result of the multiplication are scheduled and their variables are allocated to the first available registers.

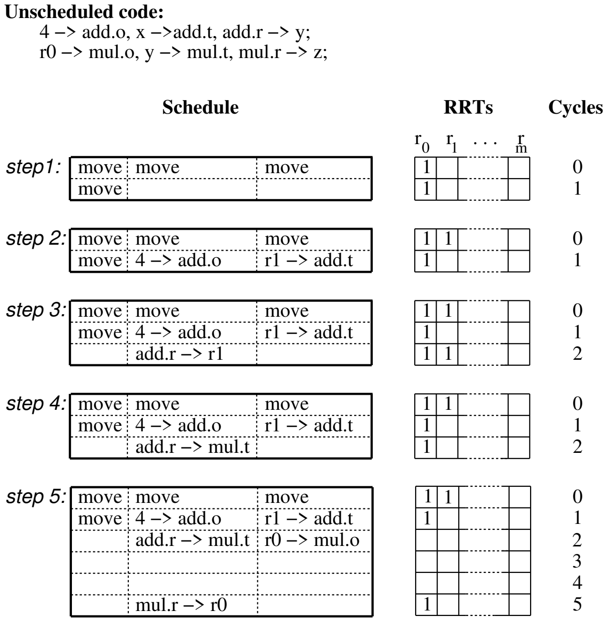


Figure 4: Example of RoD.

3.2 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. RoD checks all RRTs 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 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 5 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. Since basic block 2 has a higher execution frequency it will be picked first for scheduling/allocation. 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.

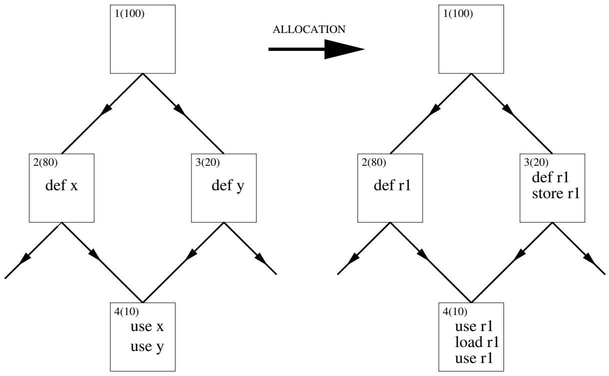


Figure 5: Spilling example.

The second goal is about the impact on the execution time of the produced code. Since we use a scheduling approach that works on the most frequently executed basic blocks first, all registers are available to these most frequently executed basic blocks and as a result the spill code is pushed to the less frequently executed basic blocks. This has as a result that the insertion of extra code for storing and loading will have less effect on the total execution time.

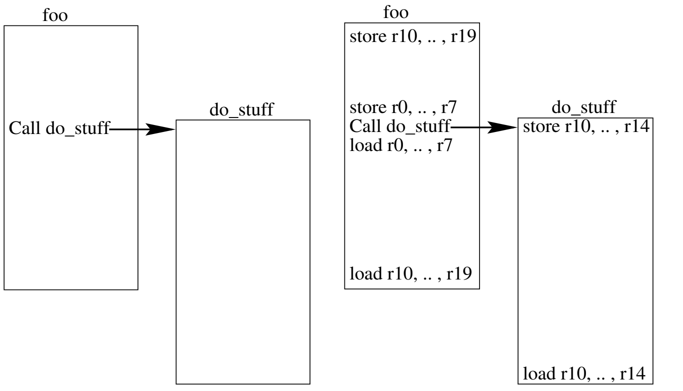
A spilled variable is written to a memory location on the stack. The memory address is an offset from the *frame pointer*. The address of the frame pointer is supplied by the calling function.

Computing the memory address of the store or load requires an addition. 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. The disadvantages of this method are that if no spilling was needed we have less registers at our disposal and some false dependencies 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 we try to schedule the addition and load in such a way that the results of the addition is bypassed to the load, as a result no register is needed and therefore we do not need reserved registers for address calculations. 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.

3.3 Caller-saved and Callee-saved Code

When a function call occurs, 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.



a: Example Caller/Callee functions b: Placement of caller/callee saved code

Figure 6: Placement of caller/callee-saved registers.

In the following we will call the function that calls another function the *caller* and the function that is called the *callee*. For example in figure 6a function *foo* is the caller and function *do_stuff* is the callee. In order to generate efficient code compilers divide the set of machine registers in caller-saved and callee-saved registers. The used calling convention equally divides the registers in caller-saved and callee-saved registers. The caller-saved registers are stored and reloaded by the caller and the callee registers are stored and reloaded by the callee. In figure 6b this is depicted for an architecture with 20 registers. Note that only those register 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 profiling. Consider the function in figure 7. The variable x is live in the basic blocks 1 and 2. Since the call in block 2 is executed more frequently than the header block and leaf block of the function it is more advantages 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 advantages to allocate it in a caller-saved register since the basic block 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 also called the return address. This value is stored in the beginning of the function and reloaded at its exits.

Caller-Saved Code When a call 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 variables which are already allocated the allocator has no choice; either they are allocated in caller-saved registers and thus need caller-saved code or they are allocated in callee-saved register and thus need no caller-saved code. 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 code has been generated, we set all caller-saved register as occupied in the RRT 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 callee-saved registers or they should be spilled.

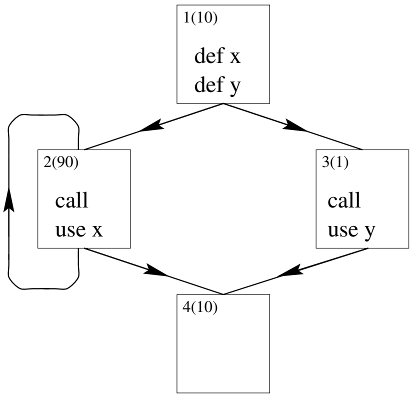


Figure 7: Heuristic for caller/callee-saved registers.

Callee-Saved Code The callee-saved code must be placed at the entry and exits of a function. This is the last step in the scheduler/allocator. The approach we used is to schedule/allocate all code except the entry and exit basic blocks. 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 created in the exit basic blocks. All callee-saved registers that are never used are set as occupied in all RRTs of all unscheduled basic block to avoid the situation that extra callee-saved code must be generated in already scheduled code. The disadvantage is of course that in the not yet scheduled basic blocks the set of free registers is limited to the caller-saved registers and the saved callee-saved registers.

4 Experiments

The goal of our experiments is to measure the performance of the new introduced scheduler with integrated register allocation. We restrict the experiments to mainly integer type of code. However, all methods can be applied to floating-point intensive code as well. The results are compared with the early assignment approach as described in [12], which uses the advanced techniques as described in [9].

The selected TTA has 8 move busses, three integer units with a single cycle latency, two load-store units with a latency of 2 and one floating point unit with latency of 3. The integer RF has 4 read and 4 write ports, the number of integer register is varied in the experiments. The floating-point RF has 48 registers and one read and one write port and the boolean RF has 4 registers and two write ports. The TTA has one delay

Table 1: Speedup of RoD compared with early assignment.

	# registers					
prog.	32	24	20	16	12	10
a68	-0.3%	-0.1%	0.1%	0.2%	3.0%	4.9%
bison	0.0%	0.0%	-0.1%	0.7%	2.0%	3.4%
compr.	0.0%	0.0%	0.0%	1.1%	26.3%	14.9%
dhryst.	0.0%	-1.5%	0.0%	0.3%	-1.1%	-2.5%
gzip	0.1%	0.1%	0.1%	0.3%	10.1%	11.8%
sieve	0.0%	0.0%	0.0%	0.0%	0.0%	15.4%
sort	-1.3%	3.2%	8.8%	9.1%	22.1%	31.4%
sum	0.0%	0.0%	0.0%	0.0%	0.0%	7.0%
uniq	-1.2%	-1.2%	9.8%	6.9%	1.1%	7.7%
wc	5.7%	5.7%	5.7%	5.7%	7.8%	10.8%
average	0.3%	0.6%	2.4%	2.4%	7.1%	10.5%

slot for branches.

The results of the experiments are listed in table 1. The first column lists the name of the benchmark. For each number of integer registers the relative speedup of RoD compared to the early assignment method is computed. As shown, the performance advantage of RoD may be more than 10% when registers are scarce.

Figure 8 compares the average cycle count increase using the same set of benchmarks, of early assignment and RoD, for different register numbers. This figure clearly shows that RoD is less hurt by having fewer registers. This is due to the better avoidance of false dependencies and pushing the spill code to the outer loops. However for some experiments there is a small performance decrease, the reasons are not yet well understood but it is to be expected that the heuristics used within the scheduler can still be improved and better tuned.

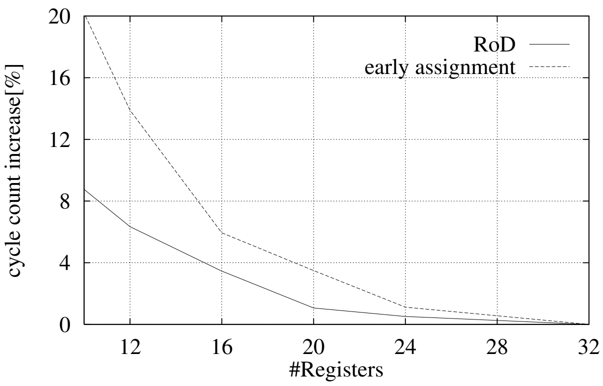


Figure 8: Performance comparison of early assignment and RoD.

5 Conclusions and Future Work

We have argued that instruction scheduling and register allocation in the context of ILP processors should be integrated. A new register allocation method is proposed and the selection of a free register, insertion of spill code and caller/callee-saved code is discussed. The method is compared with an early assignment approach. The results show an improvement for low register numbers. We believe that this method has great potential; in the future we will extend it to a larger scheduling scope, like a region. A larger scheduling scope means also more potential false dependencies which can be avoided with RoD. Another topic is register allocation over function calls, this so called inter-procedural register allocation can result in less caller and callee saved code. In the future we will also look in the area of irregular architectures since we feel that this method is far better suited for these type of architecture than early and late assignment.

References

[1] 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.

[2] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] Cindy Norris and Lori L. Pollock. A scheduler-sensitive global register allocator. In *Supercomputing '93 Proceedings*, Portland, November 1993.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] Henk Corporaal and Paul van der Arend. MOVE32INT, a Sea of Gates realization of a high performance Transport Triggered Architecture. *Microprocessing and Microprogramming*, 38:53–60, September 1993.