Aggressively Bypassing List Scheduler for Transport Triggered Architectures

Aggressively Bypassing List Scheduler for Transport Triggered Architectures

Heikki O. Kultala, Timo T. Viitanen, Pekka O. Jääskeläinen, and Jarmo H. Takala
Department of Pervasive Computing, Tampere University of Technology, Finland
Email: {heikki.kultala, timo.2.viitanen, pekka.jaaskelainen, jarmo.takala}@tut.fi

***Abstract*—A new instruction scheduling algorithm for Transport Triggered Architecture (TTA) is introduced. The proposed scheduling algorithm is based on operation-based two-level list scheduling and tries to aggressively bypass data moves before scheduling them and resolves deadlocks by backtracking and bypassing less aggressively those moves that cause deadlocks. Compared to two earlier list schedulers for TTA processors, the proposed scheduler creates code that is on average 2.0 % and 2.2 % and best case of 15.2 % and 16.3 % faster while reducing the amount of register file reads by on average of 9.7 % and 6.9 % and best cases of 31.0 % and 19.0 %, and register file writes on average 18.0 % and 18.9 % and best cases of 48.1 % and 36.8 %. The scheduling time with the proposed scheduler is short enough for the algorithm to be used when performing design space exploration unlike in some instruction schedulers based on mathematical models. The scheduler also introduces a framework, which makes it very easy to be extended to support new optimizations.**

I. Introduction

Transport Triggered Architectures (TTA) are a sub-class of VLIW processor architectures. Like on VLIW processors, single instruction word is fetched from instruction memory on every clock cycle, but it can contain multiple independent parallel operations. In a TTA processor, instructions specify data transports and the actual operation is executed as a side-effect to the data transport [3]. Effectively, TTAs contain only a single instruction, move, which defines operand and result moves on the interconnection network and operations are executed as a result to operand move to a trigger port of a Function Unit (FU). Each FU has operand ports for transferring operands for the operations to be executed and result ports for reading results from the FUs. The input ports contain registers, thus operands transferred to a port in an earlier cycle can be used as an operand for a later operation. The source values of the moves can be immediate values, registers in Register Files (RFs), or FU result ports while destinations for moves can be registers in RFs or FU operand ports. Fig. 1 shows an example of a TTA processor datapath.

In TTAs, a single instruction can define several moves, one for each bus in the architecture. These fields in the instruction are called move slots. In Fig. 2, an example schedule of ADD R0,R1,R2 is illustrated. Operands from registers R1 and R2 are moved over buses *bus0* and *bus1*, respectively, to the input ports of function unit *ALU*. Move to trigger port *in1t* triggers the execution of ADD. The result is available in the next cycle and it is moved from the results port to register R2 over bus *bus0*. In this paper, the entity of operation with sources and destinations, e.g., ADD R0,R1,R2, is referred to as program operation. This is used as source entity for scheduling.

Bypassing means transferring operation results directly from the result port of a FU to the operand port of the FU performing the next operation, i.e., we avoid storing the result to register file and accessing the same register to read the operand value for the next operation. Many CISC, RISC and VLIW processors perform bypassing but it is not visible for the programmer; In the instruction set, all the bypassed values seem to come from the registers and the hardware bypass control logic then controls the datapath to omit register read and the value is bypassed instead. The TTA programming model allows the bypasses to be visible in the instruction set, so that bypassing is controlled by instructions, not by the control logic. This instruction-set visible bypassing is called software bypassing, which allows us to avoid complex bypass control logic. In addition, TTAs do not need that many read ports in register files as the compiler knows which values are bypassed thus register read port for values that are actually bypassed are not needed. In VLIW processors with hardware bypassing, there has to be a register read port available for all the operands of all the operations that can be executed in a clock cycle. Software bypass allows also dead result elimination, which means removing register writes for values that are used only once. This can decrease the amount of register writes by at least 45 % [4].

Instruction scheduling for TTAs is challenging due to high degree of freedom on scheduling the moves over the buses and clock cycles. The scheduling is even more challenging if software bypassing is to be exploited, because in TTAs scheduling with aggressive optimizations may easily lead to deadlocks.

Traditionally, operation-based list schedulers [1] have been used for compiling code to TTA type processors and with the moves in an operation are scheduled at once, then moves from the next operation are scheduled [4], [9]. Scheduling all the moves in an operation at once loses opportunities for efficient use of optimizations which affect multiple operations such as software bypassing.

In this work, we propose a novel algorithm for cycle-based two-level list scheduler, which allows aggressive optimization with the aid of efficient bottom-up scheduling direction. Aggressive optimizations are possible without deadlocks causing scheduling failures as backtracking is used to recover from the deadlocks. The more aggressive use of software bypass

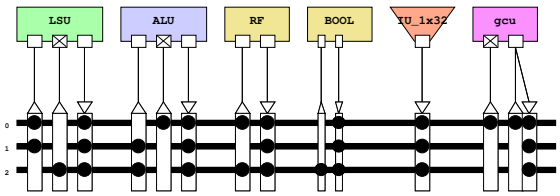

Fig. 1: Example datapath of TTA processor, which has an Load-Store Unit (LSU), Arithmetic-Logical Unit (ALU) and Global Control Unit (GCU), 32- and 1-bit register files, and special immediate register file. These units are connected via three buses, so the processor can execute three moves per clock cycle. Trigger ports of FUs are marked with X.

reduces register file writes and reads, which results in cycle count decrease.

II. Related Work

The software bypass algorithm for TTAs used in [4] schedules first all the moves from a program operation without software bypassing and then tries to improve the schedule by performing software bypass and rescheduling the bypassed moves to the cycle, which they can be scheduled with software bypass. This may result in a pessimistic schedule and lower performance.

A bypassing algorithm for TTAs with bottom-up scheduling is proposed in [5], where all the moves of a program operation are still scheduled together, but first the operand moves are scheduled to read their value from the registers. When scheduling the result move, the algorithm performs a check if all the consumers of the value can be bypassed and if possible, performs a direct bypass without scheduling the result write to a register. This algorithm schedules only result moves optimally but may still result in a pessimistic schedule for operand moves.

A method to analyze when bypass can be performed before scheduling is proposed in [10]. The method has limitations and can be applied when all the operations have the same latency. This constraint is impractical, as on almost any reasonable processor many simple operations such as integer addition can execute in one clock cycle but complex operations such as floating point operations or loads may take multiple cycles.

The instruction scheduler based on integer linear programming(ILP) proposed in [2] does not have these limitations as it schedules each move completely independently and can bypass very aggressively. It is immune to all scheduling order related inefficiencies. However, scheduling process is complex and may require a long scheduling time. In some cases the scheduling time is so long that it makes the algorithm impractical, thus this cannot be considered a robust scheduling method.

In [7], the instruction scheduler works in top-down direction, but result moves are scheduled separately, which allows operand moves to be scheduled to an early cycle, then bypassing values from the results and killing the dead results before

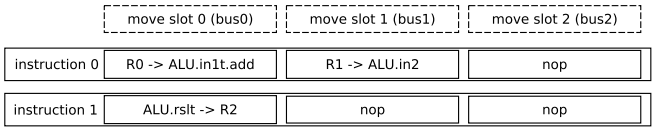

Fig. 2: TTA instruction format for the processor in Fig. 1. Unused move slots are indicated by nop.

they are scheduled. This means that no unnecessary result writes are scheduled. The processor template uses multiple result registers, which allows the late scheduling of the results; With only single result register the scheduling of the results cannot be delayed. They also propose a live-producer flow graph solution to the deadlock problem [12].

In this paper, we propose an instruction scheduling algorithm with bypassing for TTA processors. The proposed algorithm scheduling works well with architectures with only single output register per function unit as opposed to [7]. The proposed instruction scheduler introduces a more efficient bypass method not available in [4] and [5]. It is robust and the schedules can be obtained in reasonable time.

III. Scheduling Direction

There are two trivial scheduling orders for list schedulers: Top-Down (TD) or Bottom-Up (BU). In top-down approach, the producers of the Data Dependence Graph (DDG) are scheduled to the earliest possible cycle and then their consumers are scheduled to the earliest cycle allowed by the data dependencies and the execution resources of the processor. Bottom-up in turn means that the consumers are scheduled to the last possible cycle and their producers are scheduled to the last cycle allowed by the data dependencies and processor resources. If a basic block contains a branch, this is interpreted as the last operation of the basic block, so in a top-down scheduler, it is the last operation to be scheduled and the first one in a bottom-up scheduler.

In general, DDGs of basic blocks have characteristics of narrowing towards the end; there are many more source nodes than sink nodes. In the DDGs of the test cases used in section VI there are total of 18839 source nodes and 4565 sink nodes. In this kind of graph, the bottom-up scheduling leads to a schedule which has shorter live ranges for variables. In a top-down schedule, many values are produced too early and used much later since the dependencies from other operations prevent an dependent operation to be scheduled earlier. A bottom-up scheduler can schedule all these operations as late cycle as possible, and long live ranges only occur when the same variable value is used multiple times. This allows more effective use of bypassing without the bypassed values reserving the FU ports for too long time.

Another advantage of bottom-up scheduling is related to jumps on architectures with jump delay slots: When scheduling code on bottom-up fashion, a jump is first scheduled at correct position near the end of the basic block. It can always be scheduled to that optimal cycle because there are no other operations competing for the same processor resources.

a)

```
RF.1 -> ALU.in1t.shl, 2 -> ALU.in2 ;
ALU.out1 -> ALU.in2, RF.2 -> ALU.in1t ;
ALU.out1 -> LSU.in1t.st32, RF.3 -> LSU.in2
?bool.0 next_bb -> jump ; ...
full nop ;
full nop ;
```

b)

```
..., 2 -> ALU.in2 ;
?bool.0 next_bb -> jump; RF.1 -> ALU.in1t.shl,
ALU.out1 -> ALU.in2, RF.2 -> ALU.in1t ;
ALU.out1 -> LSU.in1t.st32, RF.3 -> LSU.in2
```

Fig. 3: a) Example case of the calculation code filling up the available slots, forcing jump to a too late instruction, leading to empty delay slot instructions and increase in code length and execution time. In this example the processor has two buses and two jump delay slots. b) Same code scheduled with a bottom-up scheduler. The jump has been first scheduled into the optimal instruction, and the calculation code is scheduled around it.

Moves from other program operations are then scheduled to other move slots in the instruction containing the jump or instructions preceding and succeeding the instruction. In a TTA, jump is typically executed in a separate control unit. Jump needs only one move slot, so a jump has negligible effect on scheduling other program operations. In the worst case, the bus occupied by the jump operand move may cause a move in the critical path to be scheduled at one cycle too early, resulting in a one cycle increase in the length of the basic block schedule.

Jumps sometimes present problems as we do not know the length of the schedule before scheduling the jump. This is not a problem with bottom-up scheduler because the schedule grows upwards; Only the position of the first instruction varies with the schedule, not the position of the jump. However, with a top-down scheduler, the length of the schedule is not known until all the operations have been scheduled, thus the jump should be the last operation to be scheduled. In TTAs, the scheduled moves from other program operations might have occupied all the move slots in the instruction, which is optimal for the jump. The moves may have taken also the slots in all the succeeding instructions until the end of the basic block. This can increase the basic block size by $DS + 1$ instructions, where DS is the number of delay slots in the architecture. However, instructions can often be moved from the succeeding basic blocks to new delay slot instructions, avoiding the code length increase. Fig. 3 shows examples, where this inefficiency appears with a top-down scheduler.

There are also other more complicated scheduling orders that schedule some operations in top-down fashion and some operations with a bottom-up fashion, but those are typically only used for loop scheduling, e.g., swing modulo scheduling [11].

IV. Deadlocks with Aggressive Optimizations

While software bypassing can improve the performance, the naive approach on trying to bypass everything before

(a)

```
char *a;
int i;
a[i]++;
```

(b)

```
i -> add.1            ; 1
a -> add.2            ; 2
add.out -> address    ; 3
address -> ld32.addr  ; 4
ld32.out -> oldvalue  ; 5
oldvalue -> add.1     ; 6
1 -> add.2            ; 7
add.out -> newvalue   ; 8
newvalue -> st.data   ; 9
address -> st.addr    ; 10
```

(c)

```
i -> add1.1           ; 1
a -> add1.2           ; 2
add1.out -> ld32.addr ; 3-4
ld32.out -> oldvalue  ; 5
oldvalue -> add2.1    ; 6
1 -> add2.2           ; 7
add2.out -> newvalue  ; 8
newvalue -> st.data   ; 9
add1.out -> st.addr   ; 10
```

Fig. 4: Simple code that causes deadlock with aggressive software bypassing when scheduled to TTA processor with only one ALU; (a) is the original C code. (b) is the sequential TTA code generated from the C code without bypassing; (c) is the sequential TTA code after performing bypasses from the address calculation. the two separate add operations are separated here by calling them add1 and add2.

scheduling will lead to deadlocks in scheduling. In order for a DDG to be schedulable with aggressive bypassing, it must not have more values to be held in the function unit result registers than there are result registers available in the processor architecture [10].

Consider the code in Fig. 4 on a processor with only one ALU, a single output register per FU, and a store operation, where the address port is the triggering port. When the address value is aggressively bypassed to both load and store, the write to the address variable is omitted and the reads from the address variable becomes reads from the ALU. This, however, causes a problem between scheduling the store (9-10) and the second add (6-7). The add that updates the value has to be scheduled between the load and the store for the store to read the correct value. However, it cannot be scheduled before the store, because the add would overwrite the address value in the result port of the ALU, which the store needs.

Another source of deadlocks in a TTA scheduler are too tight gaps between operations allowing either operands (in case on a top-down scheduler) or results (in case of a bottom-up scheduler) to be scheduled in the gap, but not both so that all the moves of the program operation cannot fit in the gap simultaneously. In this kind of case the already scheduled results or operands have to be unscheduled in order to allow scheduling to continue.

Fig. 5 shows a simple situation where an attempt to schedule an additional operation will cause a deadlock on a TTA processor with a single bus and a single ALU. Assume that a new *neg* operation ($R4 \rightarrow neg.trigger; neg.result \rightarrow R4$) is being scheduled. A bottom-up scheduler schedules first the result in place of the nop. Then it attempts to schedule the operand to the instruction above, but it cannot as there are no buses available. The operand cannot be scheduled to any higher instruction either as the add operation which is triggered in cycle 101 would overwrite the result of the neg operation.

In a similar fashion, a top-down scheduler schedules the operand in the place of the nop. It cannot schedule the result to the instruction below it as there are no free buses. It cannot schedule it to any later instruction because the add operation triggered in instruction 106 would overwrite the result of the

```
R0 -> ALU.in2,              ;inst 100
8 -> ALU.add.trigger        ;inst 101
ALU.out -> LDW.load.trigger ;inst 102
nop                         ;inst 103
R2 -> ALU.in                ;inst 104
LDW.load.out -> ALU.in2     ;inst 105
R3 -> ALU.add.trigger       ;inst 106
ALU.out -> R5               ;inst 107
```

Fig. 5: Example code on missing resources causing a deadlock in scheduling on processor with a single bus and a single ALU. Either one operand or one result of a new ALU operation can be scheduled to instruction 103, but not the whole operation. However, the operands cannot be scheduled before the ALU operation of instruction 101 and the results cannot be scheduled after the ALU operation of instruction 106 because the result would be overwritten by those operations before the result of the newly scheduled operation is read due to the programmer visible function unit latencies.

neg operation.

The same situation can occur also with more complex TTA processors when the most of the resources are in use. Adding more resources to the processor does not solve the problem, but it makes it occur less frequently.

This situation can already occur with the algorithms from [4], [5], thus they contain simple retry mechanism which can unschedule all the moves of the currently scheduled program operation and then try to schedule the operation to an earlier or later cycle.

V. Proposed Algorithm

The proposed algorithm is using the bottom-up direction to minimize the live ranges of variables, to maximize the opportunities for bypass, and to support more optimal scheduling of jumps. The algorithm works one basic block at a time. Before the actual scheduling a DDG of the scheduled basic block is generated with additional info of incoming and outgoing variables. Like [4] and [5], register allocation is done before the algorithm, but in the proposed scheduler some trivially short live ranges that cannot cause deadlocks such as stack offset addresses can be left without actual physical register to be explicitely marked to be bypassed by the algorithm.

A. Aggressive Bypass

In the proposed approach, aggressive bypass no longer means scheduling of all the moves from one program operation strictly together. The scheduler keeps a second level of ready list of moves of all "open program operations", we call the scheduling front list. When a program operation is selected from the main list of program operations, all the moves in a program operation are entered to the scheduling front list. When scheduling an operand move, an aggressive early bypass is attempted immediately. If the value can be bypassed, the bypassed move, which is now both operand and result move, is scheduled. All the other moves from the program operation, which produced the value, are added to the scheduling front list. Then another move from the scheduling front is selected for scheduling. This method may lead to deadlocks and, therefore, a recovery method is needed. When all the moves in the scheduling front list are scheduled, they are removed from the list and the backtracking will not touch them. However, this aggressive bypass cannot be succesfully applied to all possible bypasses, therefore we also use additional post bypassing mechanism as in [5].

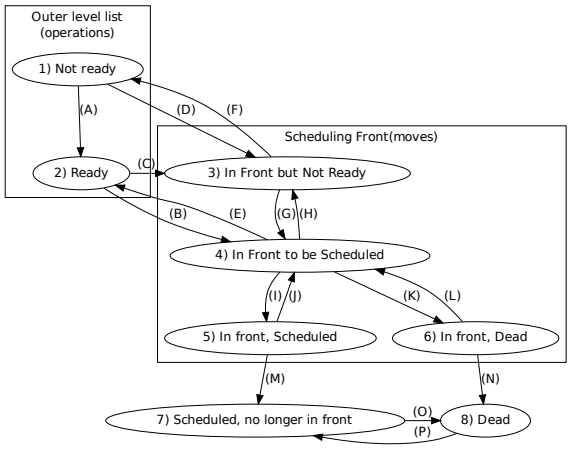

Fig. 6: The main states and state changes of moves in the proposed scheduler.

B. States of Program Operations and Moves

As there are two separate levels of ready lists in the scheduler, the program operations and moves of the currently scheduled part of the program may have one of several possible states. These states and their changes are an important part of the proposed scheduling algorithm, as the state of a move dictates which transforms can be performed and when. The possible states and the state changes are described in Fig. 6. The letters in the figure are the state changes:

- (A) All successors of an operation are scheduled, operation is ready to be scheduled.
- (B) Previous scheduling front fully scheduled, select new operation to initialize new front, or bypass puts moves of another operation into the scheduling front.
- (C) Previous scheduling front fully scheduled, select new operation to initialize new front, or bypass puts moves of another operation into the scheduling front. Operand moves are not ready until result moves are scheduled.
- (D) Bypass puts moves of another operation into the scheduling front.
- (E) Undo a bypass, drop ready moves from the front.
- (F) Undo a bypass, drop non-ready moves from the front.
- (G) All successors of a move that is in the front are scheduled.
- (H) Successors of a move that is in the front unscheduled.
- (I) Schedule a move.
- (J) Unschedule a move.
- (K) Kill a dead result.
- (L) Resurrect a dead result when undoing bypass.

(M) Whole scheduling front scheduled.
(N) Whole scheduling front scheduled, move is dead.
(O) Kill an already scheduled value.
(P) Resurrect an already scheduled value.

Moves that are fully scheduled and removed from the scheduling front can still be re-scheduled to different instructions or killed, via optimizations performed by scheduling later scheduling fronts. In case of backtracking, these moves are returned to the scheduled state to the instructions where they were originally scheduled. This rescheduling of already scheduled moves is performed for example in case of late bypassing, where the already scheduled uses of a value are changed to read their value from a newly scheduled operation.

C. Backtracking for Deadlock Recovery

The algorithm utilizes backtracking for recovering from scheduling deadlock situations. The scheduler keeps a stack of all the optimizations and other transforms, which it has performed and can, therefore, always revert the latest one. This allows it to revert to any previous state.

In the case a move cannot be scheduled, the scheduler first retries that move without bypassing or other optimizations. If the scheduling still fails, the scheduling front is searched for the bypass that caused the failed move to be in the front. This bypass is marked forbidden, and all the moves in the scheduling front are unscheduled and rescheduling is performed from the same program operation that initiated the scheduling of the scheduling front. Forbidden bypass means that bypass is not allowed for aggressive bypass but it is still possible with late bypass step, as it cannot cause deadlocks to other moves. If the move still cannot be scheduled, all the moves in the scheduling front are unscheduled again and we reschedule from the same program operation, which initiated the scheduling of the scheduling front with a constraint that it must be scheduled to an earlier cycle.

The stack containing the information of all the performed transforms is implemented as a tree-like data structure. Each node in this tree relates to one transform such as optimization of scheduling a move. If a transform consists of multiple parts, these can be separate nodes that are child nodes of the main transform. For example a transform, which reschedules an already scheduled move may contain two child nodes, one for unscheduling a move from the original cycle and another scheduling a move to a different cycle.

For the tree nodes a base class called BFOptimization is used. All the scheduler code which performs any modifications to the code being scheduled is in classes derived from this class.

Each object of this class contains two lists for children: *preChildren* and *postChildren*. *preChildren* contains the list of other transforms that this transform called before performing the actual tasks of this transform. *postChildren* contains the list of other transforms that this transform called after all the other tasks of this transform were performed.

This class has a public undo method and a protected pure virtual undoOnlyMe method. When the undo method of an BFOptimization is called, the object first calls the undo for all of it's postChildren, in reverse order, then calls it's own undoOnlyMe() method and then calls undo for all of it's preChildren in reverse order.

This mechanism allows the different optimizations and other transforms available for the scheduler to call each others while still allowing easy reverting of everything.

The class contains also pointers to some data structures shared with all the transforms such as the DDG containing the scheduled code and the resource manager containing details of the processor architecture and resources used by already scheduled operations. When a new instance of any subclass of BFOptimization is created, these data structures are passed to it.

New optimizations can be implemented by creating a new subclass of BFOptimizaton, implementing the operator()() and undoOnlyMe() methods and calling the optimizations from the code of existing classes. The undoOnlyMe() just has to revert all work done directly by the operator()() method.

D. Ready List Prioritization Heuristics

The ready list prioritization heuristics used here follows the approach proposed in [5]. In particular, a heuristic based on distance from the most further source node of the DDG is used for the furthermost operation ready list. The longer the distance, the higher the priority. Operation latencies are used as the distance metric.

Slightly different prioritization heuristics are in the inner scheduling: a higher priority is given to moves, which are the last unscheduled moves in an program operation. Also higher priority is given to the trigger move; If the heuristics gives a non-triggering operand as the next candidate to be scheduled, but the trigger move of the operation has not been scheduled yet, an operand swap is attempted to make the high priority move the trigger move. If the operation is not commutative, the trigger is scheduled first.

E. Other Optimizations

When some but not all uses of result are bypassed, the result write to a register remains. The bypassed results in the critical path are scheduled first and, in order to release the result port for the FU to be used by other program operations, the result write to register is then scheduled in a top-down fashion as close as possible to the other results. This is to minimize the time the FU is reserved for the program operation. This can, however, increase critical path for some earlier program operations due antidependencies. These are later rescheduled to later cycles in case antidependencies to these moves limit the schedule of other moves.

F. Algorithm Pseudo Code

The most important parts of the scheduling algorithm are shown in pseudo code in figures 7-11. On those algorithm descriptions runPreChild and runPostChildren routines are wrapper routines which add undo objects into the undo lists described in section V-C allowing the reversal of the routines.

```
for all m ∈ programOperation do
    schedulingFront ← schedulingFront ∪ {m}
end for
while schedulingFront has unscheduled moves do
    m ← schedulingFront.get()
    lc ← maxCycleLimit
    while m not scheduled do
        if m destinationIsVariable and m destinationValueNeverUsed then
            runPreChild(killMove(m))
        end if
        if m alive and m not scheduled then
            if m isJump or m isCall then
                runPreChild(scheduleMoveToCycle(m, maxCycleLimit-delaySlots))
            else
                if m is result to variable and has scheduled bypassed sister result then
                    sisterCycle ← cycle of scheduled sister result(m)
                    runPreChild(scheduleMoveTD(m, sistercycle, allowLateBypass))
                    if m alive and m not scheduled then
                        runPreChild(scheduleMoveTD(m, sistercycle, noLateBypass))
                    end if
                else
                    runPreChild(scheduleMoveBU(m, lc, allowEarlyBypass, allowLateBypass))
                end if
                if m alive and m not scheduled then
                    runPreChild(scheduleMoveBU(m, lc, noEarlyBypass, allowLateBypass))
                end if
                if m alive and m not scheduled then
                    runPreChild(scheduleMoveBU(m, lc, allowEarlyBypass, noLateBypass))
                end if
                if m alive and m not scheduled then
                    runPreChild(scheduleMoveBU(m, lc, noEarlyBypass, noLateBypass))
                end if
                if m alive and m not scheduled then
                    undoFront()
                    inducingBPSrc ← find the bypass which caused m to be in scheduling front
                    if inducingBPSrc ≠ null then
                        illegalBPSrcs ← illegalBPSrcs ∪ {inducingBPSrc}
                        forbidbypass ← true
                    end if
                end if
            end if
        end if
        if not forbidbypass then
            lcFront ← latest Scheduled Cycle Of Front
            if lcFront ≠ −1 and lcFront ≤ lc then
                lc ← lcFront − 1
            else
                lc ← lc − 1
            end if
            if lc < minCycleLimit then
                return false {Scheduling failed}
            end if
        end if
    end while
end while
schedulingFront.clear();
```

Fig. 7: The ScheduleFront routine which takes one program operation, creates scheduling front from that and schedules it.

```
if m sourceIsVariable and allowEarlyBypass then
    runPreChild(TryEarlyBypassMove(m))
end if
if m destinationIsVariable and allowLateBypass then
    runPreChildTryLateBypassFromMove(m))
    if m dead then
        return true
    end if
    runPreChild(tryPushAntidepDestinationsDown)
    prefResultCycle = cycle of bypassed sister
    runPreChild(scheduleMoveTD(m, prefResultCycle))
end if
lc = min(ddg.latestCycle(m, limits.latest))
lc = resourceManager.latestCycle(m, lc)
if lc ≠ −1 then
    resourceManager.assign(m, lc)
    if m isTrigger then
        runPostChild(rescheduleResultsClose(m.destinationOperation))
    end if
    return true
end if
return false
```

Fig. 8: The ScheduleMoveBU routine

```
if m destinationIsVariable and allowLateBypass then
    runPreChild(tryLateBypassFromMove(m))
    if m dead then
        return true
    end if
end if
ec ← max(ddg.earliestCycle(m, earliest))
ec ← resourceManager.earliestCycle(m, ec)
if lc ≤ maxCycleLimit then
    resourceManager.assign(m, ec)
    return true
end if
return false
```

Fig. 9: The ScheduleMoveTD routine

```
bypassSrc ← ddg.findBypassSrc(m)
if bypassSrc ∈ illegalBPSrcs∨ no connection for bypass ∨ any move of source operation of bypassSrc has data deps to unscheduled moves then
    return false
end if
bypassed ← BypassMoveFromSource(src, m))
if bypassed then
    if bypassSrc destinationValueNeverUsed then
        runPostChild(killMove(m))
    end if
    for all n ∈ bypassSrc.sourceOperation do
        schedulingFront ← schedulingFront ∪ {n}
    end for
    return true
end if
return false
```

Fig. 10: The TryEarlyBypassMove routine

```
bypassCandidates ← ddg.findBypassDestinations(m)
for all dst ∈ bypassCandidates do
    if processor has connection for bypass and lc < dst.cycle − bypassDistance then
        bypassDestinations ← bypassDestinations ∪ {dst}
        earliestDst ← min(earliestDst, dst.cycle)
    end if
end for
for all dst ∈ bypassDestinations do
    if dst.cycle = earliestDst then
        runPreChild(lateBypass(m, dst) {First do late bypass for moves in critical path}
    end if
end for
for all dst ∈ bypassDestinations do
    if dst.cycle ≠ earliestDst then
        runPreChild(lateBypass(m, dst) {Then do the ones not on critical path}
    end if
end for
if m destinationValueNeverUsed then
    runPostChild(killMove(m))
end if
return bypassDestinations ≠ ∅
```

Fig. 11: The TryLateBypassFromMove routine

VI. Evaluation

The proposed instruction scheduling algorithm was implemented to the TCE [8] toolkit. The toolkit allows customization of TTA processors from very small designs to large multi-core and vector processors. The toolkit contains code generation tools, which adapt to changes in the architecture and allows code generation to any TTA processor, which fulfils certain basic requirements. The benchmarks were run on the instruction-cycle accurate simulator of the TCE toolkit which captures statistics on clock cycles and register accesses.

The proposed scheduling algorithm was compared against the top-down scheduler from [4] and the bottom-up scheduler from [5]. Scheduling time with the proposed scheduler was very close to the scheduling time with these schedulers, but the scheduling time with the ILP scheduler [2] was over 100 times slower in some cases and it was not included in the comparison as it is meant only for scheduling highly optimized small critical basic blocks.

A post-pass global code motion-based delay slot filling algorithm was also performed for all the instruction schedulers. Clang 3.5.2 was used as the frontend of the compiler and LLVM 3.5.2 for mid-level optimizations. Loop unroll was enabled in LLVM with an unroll threshold value of 200.

A subset of the *CHStone* [6] benchmark was used. The *CHStone* benchmark is selected since it contains a range of real-world routines with varying amounts of control code and instruction-level parallelism. In particular, we have used the tests *adpcm*, *gsm*, *mips*, *jpeg*, *aes*, *blowfish*, *sha* and *motion*. These cases were selected as they are not micro-benchmarks, and the tests *dfadd*, *dfdiv*, *dfmul* and *dfsin* use 64-bit data types

TABLE I: Clock cycle count decrease on the benchmarks.

Compared to	[4]		[5]	
Test	TTA3	TTAsmall	TTA3	TTAsmall
blowfish	0.0 %	-1.9 %	-0.7 %	2.8 %
gsm	6.1 %	2.7 %	4.6 %	6.4 %
adpcm	0.4 %	0.9 %	-2.8 %	4.3 %
aes	-15.6 %	-5.6 %	-2.0 %	[5] fail
mips	10.0 %	4.2 %	0.8	1.4 %
jpeg	-9.1 %	3.4 %	2.4 %	-3.2 %
sha	8.8 %	8.3 %	-2.0 %	3.2 %
motion	3.2 %	15.2 %	16.3 %	-0.9 %
geometric mean	0.4 %	3.6 %	2.4 %	2.0 %
g. mean of both procs	2.0 %		2.2 %	

which are not supported by the TCE toolset.

Two processor architectures were used for the evaluation: TTA3 represents a medium-sized TTA processor with a good amount of instruction-level parallelism. It has two ALUs that can execute most operations, one adder which can only execution addition and subtraction, one LSU and a multiplier. It has six buses, which makes it capable of sustaining execution of three operations in parallel, and one RF with 64 registers, three read ports and one write port. TTAsmall represents a very small TTA processor which still allows some instruction level parallelism. There are separate adder, logic unit, shifter, multiplier and LSU FUs. Thanks to the three buses it can sustain average execution of 1.5 operations per cycle. It has one RF with 16 registers, one read port and one write port.

Table I shows the cycle count results. The proposed scheduler performs better than the reference schedulers on most benchmarks. However, on *aes*, the reference top-down scheduler outperforms both bottom-up schedulers. It should be noted that the reference bottom-up scheduler was unable schedule the *aes* benchmark for TTAsmall processor.

The assumption that the bottom-up schedulers easily outperform the top-down scheduler turned out to be false. The top-down scheduler was the fastest scheduler in some of the test cases and often outperformed the old BU scheduler, even though the BU scheduler had a more aggressive bypass algorithm. The reason for this is the following. In these processors, only small non-negative values can be used as short immediate, which is encoded in a single move slot in an instruction. In other cases, the value is transferred to the datapath as a long immediate, i.e., multiple move slots in the instruction word are used to encode the value, which is then moved into a dedicated long immediate register. This register is then used as a source in the actual move using the value. In many of the test cases, the code contains long immediate values of on average one of five instructions. The bottom-up schedulers first schedule the use of the long immediate value, and then often immediately schedules the long immediate write to the instruction preceding the use. As the long immediate values consume multiple move slots from the instruction encoding, this can prevent scheduling moves, which are in the critical path to these instructions. The top-down scheduler has scheduled these critical path moves before

A)

```
RF.1 -> LSU.in1t.ld32 ; # @2327
full nop ; # @2328
full nop ; # @2329
..., LSU.out1 -> ALU2.in2 ; # @2330
2 -> ALU2.in1t.shl, [IMM.0=-4] ; # @2331
IMM.0 -> ALU2.in1t.and, ALU2.out1 -> ALU2.in2 ; # @2332
```

B)

```
RF.1 -> LSU.in1t.ld32 ; # @2327
full nop ; # @2328
[IMM.0=-4] ; # @2329
2 -> ALU2.in1t.shl, LSU.out1 -> ALU2.in2; # @2330
IMM.0 -> ALU2.in1t.and, ALU2.out1 -> ALU2.in2; # @2331
```

Fig. 12: A) Example case of bottom-up scheduler placing long immediate write to a too late cycle. The LSU.out1 to ALU_LSU.in2 move cannot be scheduled to instruction 2331 because the long immediate consumes the second move slot from the instruction encoding. This forces the triggering of the load operation to cycle 2327 instead of instruction 2328. B) Same code scheduled with top-down scheduler. Top down-scheduler first schedules the "shl" operation to clock cycle 2330, and only after that the 'and" operation to cycle 2331. The immediate write is scheduled to instruction 2329, which does not contain any moves.

it schedules the long immediate write and ends up scheduling the long immediate write to some earlier instruction with empty move slots without disturbing the moves that are in the critical path. Fig. 12 illustrates this problem and the optimal schedule created by [4]. In this processor, all the negative immediate values have to be handled as long immediates.

Another reason why [4] performed relatively well was the effect of the scheduling direction to the inter-basic block delay slot filler: Sometimes a basic block contained a move which only copied value of one register to another register, and on the next basic block the value was accessed from the register. A top-down scheduler places these copies to the beginning of the basic block, allowing delay slot filling from the successive basic block. Schedulers working with bottom-up direction put these copies to the last instruction of the basic block, preventing delay slot filling from the successive basic block that uses the value when the value is in the critical path in the next basic block.

The number of register reads and writes from the benchmarks was measured. Table II contains the register read reduction numbers and Table III contains the register write reduction numbers.

On the TTAsmall processor the proposed scheduler gives the least amount of register reads on all but one benchmark and the amount of register writes is smallest in all but two of the benchmarks. On both tested processors, the worst register usage improvement was achieved on *blowfish* and *jpeg* benchmarks. About half of the execution time in *jpeg* benchmark was spent in single *memcpy* loop which copied data from one data buffer to another byte by byte. In this loop, many of the values were indices or updated pointers that went over a loop edge so that they could not by bypassed by the software bypassing algorithms which only work inside one basic block. Implementing loop scheduling support with

TABLE II: Register read count reduction results.

Compared to	[4]		[5]	
Test / Processor	TTA3	TTAsmall	TTA3	TTAsmall
blowfish	-9.5 %	12.0 %	-4.1 %	5.1 %
gsm	8.7 %	17.8 %	10.5 %	13.6 %
adpcm	31.0 %	16.2 %	19.0 %	6.2 %
aes	14.4 %	8.1 %	10.3 %	[5] fail
mips	0.3 %	1.2 %	1.1 %	10.3 %
jpeg	2.0 %	1.5 %	2.0 %	4.1 %
sha	20.8 %	3.8 %	11.3 %	6.9 %
motion	11.3 %	3.3 %	1.8 %	6.7 %
geometric mean	11.2 %	8.2 %	6.3 %	7.6 %
g. mean of both procs	9.7 %		6.9 %	

TABLE III: Register write count reduction.

Compared to	[4]		[5]	
Test / Processor	TTA3	TTAsmall	TTA3	TTAsmall
blowfish	-11.0 %	25.8 %	0.5 %	26.2 %
gsm	11.0 %	27.2 %	19.3 %	24.3 %
adpcm	43.3 %	28.1 %	26.1 %	11.6 %
aes	22.2 %	4.3 %	28.0 %	[5] fail
mips	0.6 %	2.6 %	6.1 %	18.9 %
jpeg	2.2 %	-0.9 %	5.4 %	6.7 %
sha	38.6 %	14.1 %	30.8 %	23.2 %
motion	48.1 %	15.4 %	17.9 %	26.5 %
geometric mean	20.6 %	15.3 %	18.1 %	20.0 %
g. mean of both procs	18.0 %		18.9 %	

bypassing over loop edges to the proposed scheduling algorithm should decrease significantly the register usage in this benchmark.

VII. Conclusions

A novel aggressively bypassing cycle-based bottom-up two-level list scheduler for TTA processors was proposed. In the *CHStone* test, the proposed scheduler results in on average 2.0 % less clock cycles than [4] and 2.2 % less clock cycles than [5] when tested on two different processor architectures. The best case speedup was 16.3 %. Savings on register file accesses were, however, greater: On average 9.7 % over [4] and 6.9 % over [5] on register reads, and 17.9 % over [4] and 18.9 % over [5] on register writes. The smaller number of register accesses decrease the processor power consumption and suggests processors with fewer register file ports, thus the proposed instruction scheduler can help to develop smaller and more power-efficient TTA processor designs without performance penalty.

While the proposed instruction scheduler cannot reach to performance of solver-based optimal scheduler such as the one proposed in [2], it can schedule cases where those optimal schedulers fail to produce code in reasonable time, and the proposed scheduler can be used as backup option in these cases; The most performance critical very small basic blocks could be scheduled with the solver-based scheduler and the less performance-critical and larger basic blocks are scheduled with the proposed scheduler.

VIII. Future Work

The long immediate scheduling problem and register-to-register moves causing delay slot filling inefficiences described in section VI are open research problems. Furthermore, the effects of top-down scheduling with the same optimizations need to be investigated and compared to results from the proposed bottom-up approach. Yet another interesting topic is to add loop scheduling and superblock scheduling support.

As the proposed scheduler gives a good framework for testing many small peephole optimizations, the effect of these need to be investigated. Also the cooperation of the proposed scheduler and an integer linear programming based scheduled should be investigated. Integrating register allocation to the proposed scheduling algorithm could also allow considerable improvement in register usage so it is to be examined.

Acknowledgment

This work was funded by the Finnish Funding Agency for Technology and Innovation (funding decision 1134/31/2015, ParallaX3) and ARTEMIS JU (grant agreement 621439, ALMARVI).

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] T. Äijö, P. Jääskeläinen, T. Elomaa, H. Kultala, and J. Takala. Integer linear programming-based scheduling for transport triggered architectures. *ACM Trans. Archit. Code Optim.*, 12(4):59:1–59:22, Dec. 2015.

[3] H. Corporaal and M. Arnold. Using transport triggered architectures for embedded processor design. *Integrated Computer-Aided Eng.*, 5(1):19–38, 1998.

[4] V. Guzma, P. Jääskeläinen, P. Kellomäki, and J. Takala. Impact of software bypassing on instruction level parallelism and register file traffic. In M. Bereković, N. Dimopoulos, and S. Wong, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5114 of *Lecture Notes in Computer Science*, pages 23–32. Springer, Heidelberg, Germany, 2008.

[5] V. Guzma, T. Pitkänen, and J. Takala. Use of compiler optimization of software bypassing as a method to improve energy efficiency of exposed data path architectures. *EURASIP J. Embedded Syst.*, 2013(1):1–9, 2013.

[6] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *J. Inf. Process.*, 17:242–254, 2009.

[7] Y. He, D. She, B. Mesman, and H. Corporaal. MOVE-Pro: A low power and high code density TTA architecture. In *Proc. Int. Conf. Embedded Comput. Syst.: Arch. Modeling Simulation*, pages 294–301, Samos, Greece, 2011.

[8] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala. Codesign toolset for application-specific instruction-set processors. In *Proc. SPIE Multimedia Mobile Devices*, pages 65070X–1 – 65070X–11, San Jose, CA, USA, 2007.

[9] J. Janssen. *Compiler Strategies for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, The Netherlands, 2001.

[10] P. Kellomäki, V. Guzma, and J. Takala. Safe pre-pass software bypassing for transport triggered processors. *Acta Technica Napocensis*, 49(3):5–10, 2008.

[11] J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, Boston,MA,USA, 1996.

[12] D. She, Y. He, B. Mesman, and H. Corporaal. Scheduling for register file energy minimization in explicit datapath architectures. In *Proc. Design, Automation Test in Europe Conference Exhibition*, pages 388–393, Dresden, Germany, 2012.