Exposed Datapath Optimizations for Loop Scheduling

Heikki O. Kultala, Pekka O. Jääskeläinen, Johannes IJzerman,
Lasse K. Lehtonen, Timo T. Viitanen, Markku J. Mäkitalo, and Jarmo H. Takala
Tampere University of Technology, Finland
Email: {heikki.kultala, pekka.jaaskelainen, johannes.ijzerman, lasse.lehtonen, timo.2.viitanen,
markku.makitalo, jarmo.takala}@tut.fi

***Abstract*—Transport Triggered Architecture (TTA) processors allow unique low level compiler optimizations such as software bypassing and operand sharing. Previously, these optimizations have mostly been performed inside single basic blocks, leaving much of their potential unused. In this work, software bypassing and operand sharing are integrated with loop scheduling, allowing optimizations over loop iteration boundaries. This considerably further reduces register file accesses and immediate value transfers on tight loops – in some cases even eliminating all register file accesses from the loop body. In the benchmarked 12 small loops, compared to traditional VLIW-style processors, on average 63% of register file reads and 77% of register file writes could be eliminated. Compared to a compiler which performs these optimizations only inside a basic block, on average 58% of register file reads, 28% of register file writes could be eliminated. The additional register access reductions allow both direct energy savings from fewer register accesses and indirect energy savings by allowing the use of simpler register files with less read and write ports and a simpler interconnect network with less transport buses.**

I. Introduction

Transport Triggered Architectures (TTA) can be seen as a sub-class of Very Long Instruction Word (VLIW) processor architectures. As with ordinary VLIW processors, a single instruction word is fetched from instruction memory on every clock cycle, and the instruction can contain multiple parallel operations. In a TTA processor, instructions specify data transports and the actual operation is executed as a side-effect to the data transport [1]. This allows more fine-grained control of the processor datapath to the software and allows optimizations such as software bypassing [2] and operand sharing [3], which can be used to save power.

Like VLIWs, TTA processors are typically used in Digital Signal Processing (DSP) style workloads where the control flow is relatively simple, but the code contains large amounts of arithmetics organized as loop kernels. Therefore, performance on loop-oriented code is of utmost importance when compiling code for these processors. Compilers for TTA processors have typically unrolled loops heavily to expose static instruction-level parallelism needed for high performance. However, aggressively unrolled code consumes more instruction memory, and unrolling is not always practical if the iteration count of the loop is not known at compile time.

Loop scheduling is a special mode of operation in an instruction scheduler which is used for scheduling code in inner loops. A loop scheduler typically interleaves multiple iterations of loop, converting it to a "software pipeline" [4]. This allows the performance of the loop to be considerably increased without unrolling the loop. Loop scheduling and unrolling can both be applied simultaneously; a loop can be first partially unrolled and the partially unrolled loop can then be scheduled with a loop scheduler.

In this work, the TTA-specific optimizations *software bypassing* and *operand sharing*, are extended to work in combination with loop scheduling to allow these optimizations to be performed over loop iteration boundaries. This reduces register file accesses and amount of immediate values transferred to the processor datapath inside tight loops, while also removing a one-cycle overhead, which some of the previous software-bypass implementations sometimes cause for tight loops.

This paper is structured as follows: Section I discusses the background for this research. Section II reflects this paper against related work. Section III discusses the TTA programming model. Section IV discusses loop scheduling in the context of TTAs. Section V and Section VI describe the ideas of loop-carried software bypass and the loop-invariant operand sharing. Section VII contains evaluation of the proposed optimizations with register access, immediate transfer and cycle count benchmarks. Section VIII highlights further research topics related to this research, and Section IX concludes the work.

II. Related Work

A software bypassing algorithm similar to the one in the proposed work was evaluated in [2] and [5], but that work did not perform loop scheduling, and thus could not perform software bypassing of loop-carried data. Not performing any bypassing over loop iterations means not achieving optimal register usage savings on loops, but it also may cause an extra overhead of one cycle of latency in the loop cycle count in case there are data dependencies between iterations that are in the critical path of execution.

Operand sharing optimization implementation was introduced in [6] and further studied in [3], but neither of the references performed operand sharing over loop boundaries for loop-invariant values.

In [6], loop scheduling for TTA processors was described and a version of loop-carried software bypass was used to

bypass loop-carried values. However, this work only bypassed values produced and used at same clock cycle, thus not minimizing the number of register file accesses. In addition, it did not remove register writes inside the loop body for values which are only used after the loop, but not inside the loop.

In [7], register allocation was integrated with loop scheduling for a TTA processor. This thesis does not describe what was done for values that are only used after the loop, but not inside the loop, and it does not seem to perform operand sharing over loop boundaries. Their integrated register allocation and scheduling reduces the number of required registers, and also gets rid of the antidependence problem between different loop iterations, but does not reduce the number of accesses made to the registers.

In [8], an integer linear programming based loop scheduler was evaluated for TTA processors. Their instruction scheduler is not performing any of the TTA-specific optimizations, it is only rescheduling code generated by previous schedulers such as [6] or [7] to different instructions. That is, it is not performing any additional bypassing or operand sharing, thus does not decrease the register file or immediate transfer usage.

Software bypassing of loop-carried values and loop-invariant operand sharing were both used in the hand-optimized assembly analysis in [9], but it was not performed automatically by a compiler, and the benchmark set used in the evaluation was quite limited. In the proposed work, these optimizations are performed automatically by the compiler starting from C code.

III. Transport Triggered Architecture

Figure 1 shows an example of a TTA processor datapath. In TTA processors the basic element of execution is a move, which means a single data transport inside the processor datapath. A single instruction word can define several moves, one for each bus in the architecture. These fields in the instruction are called move slots. Each instruction maps to exactly one clock cycle; every move in one instruction word is executed at same clock cycle.

Figure 2 illustrates an example schedule of a TTA equivalent of a RISC-style instruction *ADD R0, R1, R2*. 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 is moved from the results port to register R3 over bus bus0.

The destination of a move can be either a register or a function unit operand port. If it is a function unit port which is specified as a trigger port of the function unit, the move also triggers an operation in the function unit as a side-effect. If the destination port is not triggering, the value can be used either by an operation triggered by another move at the same or a later cycle. In some TTA processors none of the ports are triggering, but there are special opcode fields in the instruction word for every function unit to trigger an operation [10].

The source of a move can be either a register, an immediate value, or a function unit result port. In case a move is from a

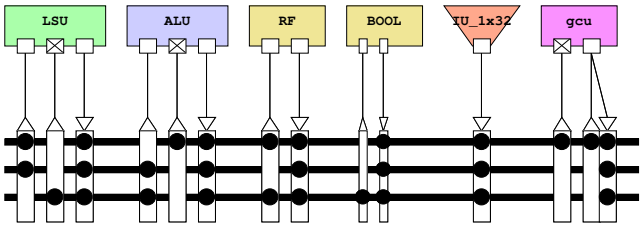

Fig. 1. Example datapath of TTA processor, which has a Load-Store Unit (LSU), Arithmetic-Logical Unit (ALU), Global Control Unit (GCU), 32- and 1-bit register files, and a immediate (constant) unit. These units are connected via three buses, so the processor can perform three data moves in an instruction cycle. The trigger ports of FUs are marked with X.

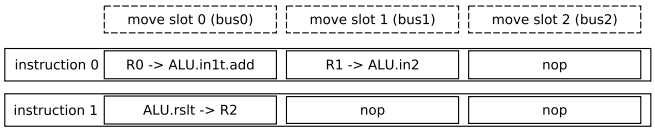

Fig. 2. TTA instruction format for the processor in Figure 1. Unused move slots are indicated by nop.

function unit result port to a function unit input port, this is called a *software bypass*. This can be used to reduce register file reads. Software bypassing also makes the control logic of the processor simpler as the bypass detection logic can be omitted from the hardware. In case all the uses of a value are bypassed, the result do not have to be written to a register. Omitting these register writes is called *Dead result elimination*. Software bypassing with dead result elimination can reduce the amount of register writes considerably [2] [11].

As TTA processors typically have function unit input registers, it allows operands to be written to function units earlier than the clock cycle where the operation begins execution. Furthermore, an operand used in consecutive operations may be written only once, saving register file accesses and internal bus traffic. This optimization is called *operand sharing*. The effect of operand sharing when applied inside single basic blocks was evaluated in [3].

Reducing the number of register accesses directly saves energy as the register file access can consume considerable amount of energy, but also indirectly, as reducing both register file reads and writes allow creating processors with data paths that have less register file read and write ports, which are more power-efficient. [5]

In some cases however TTA code can be longer, than an equivalent VLIW code. This is because the timing of the result writes is defined by the program *explicitly*, unlike in VLIWs

```
LSU.load RF.3, RF.1
LSU.load RF.4, RF.2
(full nop)
FPU.addf RF.5, RF.4, RF.3
```

Fig. 3. "Traditional VLIW" code for summation of two values loaded from memory with a 2-cycle LSU and an 4-cycle FPU. The schedule is only 4 cycles long, but as the result value will be written to RF.3 register 4 cycles later, total execution time of this code is 8 cycles.

```
RF.1 -> LSU.ld8.a
RF.2 -> LSU.ld8.a
LSU.out -> FPU.1
LSU.out -> FPU.addf.2
(full nop)
(full nop)
(full nop)
FPU.out -> RF.3
```

Fig. 4. Equivalent TTA code for the example in Fig.3. This schedule is 4 instructions longer. However, the result is written to the register RF.3 at same cycle and is available to be bypassed in the same cycle as in the VLIW case, that is, the actual execution is not longer than the VLIW code, but the TTA programming model makes the result wait time explicit.

where the result is always *implicitly* written to the destination after the static operation latency. As the result writes of operations are in different, later instruction cycles than the operand writes that trigger the operations, the same code may require more instructions to describe. Figures 3 and 4 show the example of TTA code that is longer than an equivalent VLIW code. Typically, however, later code can be overlapped with the result writes and the resulting execution time difference disappears.

IV. Loop Scheduling on TTAs

Software pipelining is a technique where the instruction schedule of multiple loop iterations is interleaved in such way that different phases of different iterations of a loop are executing concurrently. Figure 5 illustrates a software pipeline of a high level code where the calculation is shown in three high-level phases; load, calculate and store. In this example, three iterations of the loop are overlapped. First, an initialization code called *prologue* is executed. It initiates the execution of the first iteration(s) of the loop. The *loop body* (also known as the "kernel" or the "steady state") contains parts of code for multiple interleaved iterations of the original loop, so that the each original instruction of the loop is there exactly once, but in a different order and for a different iteration than the original non-pipelined loop. After the body has finished executing, most of the original iterations have fully finished, but the very last ones are not. In order to finish the last iterations, a code block called *epilogue* is executed.

Loop schedulers of compilers typically implement software pipelining via some variation of the "modulo scheduling" algorithm [12]. In modulo scheduling, a new (original) loop iteration starts between exact number of cycles, called *initiation interval* (II). The prologue and epilogue are then generated in such way that they together with the modulo scheduled loop body retain the data flow of the original unpipelined loop body.

In the proposed loop scheduler optimizations, we focus on the following style data-oriented inner loops, which we believe represent a majority of DSP-style inner loops:

a) Storing loops that produce multiple values and store them into an array in memory, and
b) Reduction loops that produce a single value into a register or a few registers out of the input data.

Naturally, there are also other classes of loops such as loops which only produce side effects such as write commands to an IO device, loops that simultaneously store multiple values into

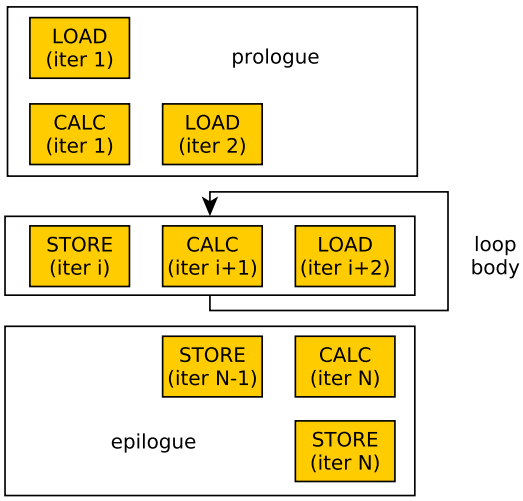

Fig. 5. High-level Example of software pipelining with loop with 3 phases: load, calc and store. In this example three iterations of the loop are overlapped, so the prologue contains the beginning of two iterations and epilogue contains the end of two iterations. Code which is in the same line can be executed in parallel. Here the loop iterations go from 1 to N, and the loop body is executed N-2 times as 2 iterations are handled by the prologue and epilogue.

memory and perform a reduction storing the result in a register, or loops which perform multiple iterations of calculations with same scalar data, but such loops are a not typically in parts of the program where performance matters.

A common example of a type *a* loop is the *memcpy* function which copies a range of values from one memory location to another, or an element-wise array addition. A simple example of a type *b* loop is the vector dot product, which accumulates a single value from the products of elements of two vectors. Practically all loops input at least one of their values in a register. Even when the values used for calculation originate from memory, at least a single pointer used for accessing the data is transferred into the loop kernel in a register.

These two classes of loops present different challenges and opportunities during loop scheduling. In *a*-type loops, the alias analysis is typically a challenge as the memory is usually both read and written inside the loop. Reads and writes may not be reordered unless the compiler can prove that the accesses do not target the same memory address. The alias analysis problem is, however, orthogonal to the proposed techniques of this paper.

Another challenge in some *a*-type loops is that the same index or pointer variable may be used for both a load and a store, which means that the value of the pointer of index must be alive when the store needs it, but the load of next iteration already needs the updated value. This causes an antidependence problem for overlapping the loads and stores. Antidependence problems can, however, be solved by register renaming. That is, by copying the value into another register. Software bypassing can be used to ease the problem by transferring the value to a function unit input port.

In *b*-type loops, where values in registers are produced, with

```
    for (int i = 0; i < n; i++) {
        *dstPtr++ = c;
    }
```

Fig. 6. A simple *memset* implementation used as an example case. Real memset implementations are typically more complex loops that write multiple bytes with one iteration, but this simple loop is also used with them for unaligned first or last iterations.

```
prologue:
    RF.1 -> gcu.lbufs.iters, 2 -> gcu.lbufs.len
    (full nop) ;
    (full nop) ;
    (full nop) ;
loop:
    RF.2 -> ALU.add.2, 1 -> ALU.1 ;
    RF.3 -> LSU.d, RF.2 -> LSU.st8.a, ALU.out -> RF.2 ;
```

Fig. 7. Schedule for the *memset* loop of Figure 6 without either of the optimizations. Without loop-carried bypass, the loop takes two cycles to execute as the first instruction reads the pointer from the RF.2 register and starts the pointer update add operation, and the second instruction writes the updated pointer value in the RF.2 register. LSU.d is the data input port of the LSU, LSU.st8.a is the address port of the LSU with opcode *st8* (store 8 bits). The example uses a loop buffer: Operand 1 of the *lbufs* operation controls the loop buffer's iteration count which in this example is dynamic (variable n in the original code) and comes from register RF.1, operand 2 sets the loop length in instructions (1).

ordinary processors without software bypassing, the updated result values are written to the register file on every loop iteration. TTAs allow omitting the write to the register file in the loop body and storing the final value to the result register only in the epilogue of the loop, if the intermediate values are bypassed across loop iterations.

When compared to traditional VLIWs, the *explicit* result writes (explained in Section III) of TTAs add an extra challenge to the loop scheduling. In *b*-type loops there must be an instruction which performs the result write, while with a VLIW or other traditional "operation-triggered processor", this instruction is not needed, as the result write is implied by the earlier instruction. This means that in order to achieve the same performance (initiation interval), the schedule for a TTA may need to be overlapped over more iterations than a schedule for an equivalent operation-triggered VLIW processor.

V. Loop-Carried Software Bypass

```
prologue:
    RF.1 -> ALU.1, -1 -> ALU.add.2
    ALU.out -> gcu.lbufs.iters, 1 -> gcu.lbufs.len
    (full nop) ;
    (full nop) ;
    RF.2->ALU.add.2, 1->ALU.1, RF.3->LSU.d, RF.2->LSU.st8.a
loop:
    RF.3->LSU.d, ALU.out->LSU.st8.a, ALU.out->ALU.add.2, 1->ALU.1
```

Fig. 8. Schedule for the *memset* loop of without loop-invariant operand sharing, but with loop-carried bypass enabled. The new pointer address, which is a result of the add operation, is bypassed to both the add operation itself over the loop edge, for calculating the pointer for the next iteration, and also to the store operation (LSU.st8.a).

When software bypassing is not performed for loop-carried values, all the loop-carried values have to be temporarily stored into registers or memory to pass them across iterations. As a result, the energy saving benefits of software bypass cannot be achieved and in addition, an extra clock cycle of latency is needed for the execution of the operation as the value is not bypassed. Integrating software bypassing into loop-scheduling and performing software bypassing also for loop-carried dependencies solves these issues, allowing both greater reduction in register file accesses and, in case the loop-carried values were in critical path of execution, one cycle faster loop kernel execution time.

Figures 6, 7 and 8 show an example of a loop where the lack of bypassing over a loop-carried dependence increases the initiation interval of the loop from one cycle to two cycles. The source code of this example is shown in Figure 6. Figure 7 contains the generated two-cycle code without loop-carried software bypassing, and finally, Figure 8 shows the one-cycle schedule of the same loop with loop-carried bypass enabled.

When a value is software-bypassed from one loop iteration to the next iteration, it cannot be bypassed to the first iteration with the same mechanism, as there is no previous iteration and the operation producing the result has not been executed. Therefore, the first iteration reads the value of the variable before the loop execution, which comes from a register. This is implemented by not performing the loop-carried bypasses for the moves in prologue, and only performing the bypass in moves in the loop body. The moves in the prologue read their values from the original source registers.

When performing bypassing for loop-carried data, there are two separate dead result elimination modes: In case all the uses of the value inside the loop are bypassed and the value is not used after the loop, the result write is removed from both the loop body and the prologue or the epilogue. In case all uses of a value inside a loop are bypassed, and the value is only used from a register after the loop, only the value in the last iteration needs to be stored to the result register. This is performed by scheduling these result write moves only to the epilogue of the loop and not to the loop body.

VI. Loop-Invariant Operand Sharing

Loop scheduling can benefit from the TTA's operand sharing optimization. The idea is that loop-invariant operands may be written to the operand port of the function unit only once before entering the loop, and the operand write can be totally omitted from the loop body. This saves power and also frees up processor resources such as buses, register file read ports and instruction word bits to be used for other operations within the loop. This can increase performance if these resources are limiting it.

Loop-invariant operand sharing means that the function unit input port is allocated for the value for the duration of the total execution time of all the iterations of the loop. This means that such an FU input port allocation algorithm must be used which allocates the input ports of the function units to different loop-invariant values. Often all the loop invariant values cannot be shared if the processor does not have enough function units to store all the shared values and also have at least one function unit of each type free for variant values.

In this work, a greedy usage-count prioritizing allocator was used. Before the loop is scheduled, all the different operand

```
prologue:
    RF.1 -> ALU.1, -1 -> ALU.add.2
    ALU.out -> gcu.lbufs.iters, 1 -> gcu.lbufs.len
    (full nop) ;
    RF.3 -> LSU.d, 1 -> ALU.1,
    RF.2 -> ALU.add.2, RF.2 -> LSU.st8.a
loop:
    ALU.out -> ALU.add.2, ALU.out -> LSU.st8.a
```

Fig. 9. Schedule for the *memset* loop of figure 6 with both optimizations enabled. The RF.3 $\rightarrow$ LSU.d and 1 $\rightarrow$ ALU.add.2 moves are moved into the prologue, and are being executed only once before the loop. This code can execute with just two transport buses, as there are no more than two moves in any instruction.

sources going into operations in the loop are counted. When performing this counting, registers which are modified in the loop are also counted, even though they are not loop-invariant. These operand sources are then sorted by the number of uses they have. For example, if register RF.0 is used three times and immediate value 4 is used twice and register RF.1 is used once, they are sorted into the order RF.0, 4, RF.1.

Function unit ports are then allocated for each value, beginning with the sources which are used the most often. In case a value in a register was invariant, i.e. it was modified in the loop, the allocation is marked as the port shared between multiple values, and no operand sharing is performed for this port. In case there is no free function unit port for some value, one previously made allocation is reversed and both values are then allocated to the same port, marked as shared between multiple values and no operand sharing is performed for this port. The allocation with the lowest usage count is reversed. When the value allocation to the ports is finished, all the moves in the loop body are scheduled. The writes of the shared loop-invariant operands to the function unit ports are only scheduled to the prologue of the loop, not to the loop body.

Figure 9 shows the memset example case of figure 6 with loop-invariant operand sharing enabled. The same code without loop-invariant operand sharing was shown in Figure 8. In this example, loop-invariant operand sharing has eliminated two moves from the loop body: One transferring the immediate value one to the ALU and another transferring the register RF.3 to the store data port.

VII. Evaluation

The proposed optimizations were evaluated using the TCE toolset [13]. The optimizations were implemented on top of the instruction scheduler algorithm described in [11], which was extended to perform modulo scheduling.

In this evaluation, a simple hardware loop buffer was used. This loop buffer takes the iteration count of the loop and instruction word count of the loop as parameters and keeps looping the given number of instructions that occur a specified delay slot cycles after the loop buffer setup instruction. This eliminates loop index calculations and checks from the loop body, and allows loops that are shorter than the number of delay slots in the processor architecture.

Benchmarks were performed on a TTA processor with three integer ALUs with a 1-cycle latency, an FPU with an FMA operation and a 4-cycle latency, a LSU with a 2-cycle load latency, six register read ports, three register write ports, and sufficient bus connectivity that the interconnection network is not a bottleneck with any of the test cases. The LSU did not have any addressing modes which require arithmetics, thus all address calculations were done by the ALUs.

We evaluated loop-carried bypass and loop-invariant operand sharing both separately and in combination. These cases are later referred to as *bypass*, *opshare* and *both*, respectively. The comparison was made against a baseline where operand sharing and software bypass are performed only inside basic blocks (*tta baseline*) and against a more limited loop-carried bypass implementation with the same limits as the MOVE instruction scheduler described in [6], without loop-invariant operand sharing (*tta move*). As the actual scheduling algorithm implementation was not available, we used our proposed scheduler with artificially added constraints present in their implementation.

In addition, as TTAs have been proposed as an alternative for VLIW processors, we made a comparison against a hypothetical VLIW design with similar execution resources (*vliw*). The VLIW comparison was done by calculating results based on the schedule for the TTA with the constraints that the VLIW programming model has in comparison to TTA. The constraints we added were the following: Every instruction has to write its result to a register file, bypassing has to be done at the cycle of the result arrival (models the hardware bypassing), and operands have to be read on the same cycle (models the lack of operand transport freedom).

A. Benchmarks

A variety of loops were selected as benchmarks. All are single-basic block loops as the current loop scheduler implementation used in the evaluation did not yet support loops with multiple basic blocks. The most common C library routines memset and memcpy were selected as their performance may have considerable impact on performance of a wide variety of workloads. Various other simple loops commonly used in linear algebra and other numerically intensive routines from signal processing workloads were selected. These versions of the loops used single-precision floating point arithmetics, as usage of floating point arithmetics in signal processing is increasing. The loops are the following:

- *4-way memcpy*. 4-way unrolled memcpy, from newlib. This copies 16 bytes at one iteration with four 32-bit memory reads and writes per loop. iteration. This is the main memcpy loop used by the TCE toolset used in the evaluation. A “storing loop”.
- *1b memcpy*. Bytewise memcpy, from newlib, used for unaligned copies and final iterations. A “storing loop”.
- *4-way memset*. 4-way unrolled memset, from newlib. This sets 16 bytes per loop iteration by storing four 32-bit integers. This is the main memset loop used by the TCE toolset used in the evaluation. A “storing loop”.
- *1b memset*. Bytewise memset, from newlib, used for unaligned/last iterations. A “storing loop”.

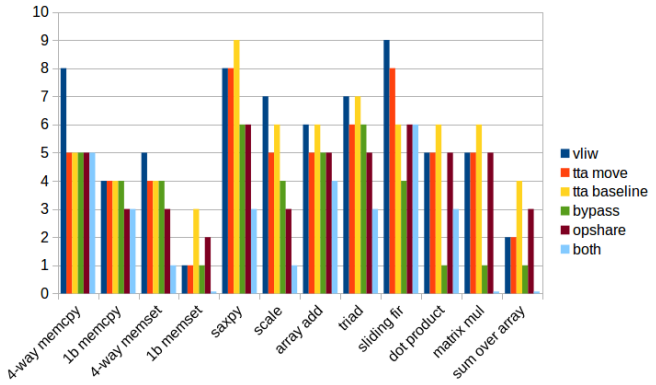

Fig. 10. Number of register file reads per iteration.

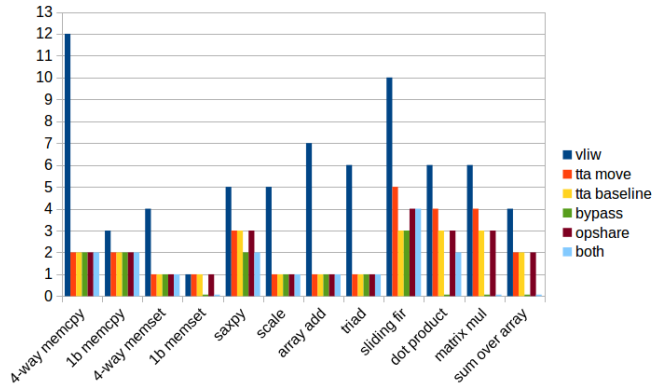

Fig. 11. Number of register file writes per iteration.

- *SAXPY*, from BLAS [14]. Elementwise accumulate every n:th value of one vector by product of scalar value and every m:th elements of another vector. A "storing loop".
- *Scale*. Vector by scalar multiplication, from STREAM [15]. A "storing loop".
- *Add*. Elementwise array addition, from STREAM [15]. A "storing loop".
- *Triad*, from STREAM [15]. Elementwise accumulate all elements of one vector by product of a scalar value and elements of another vector. A "storing loop".
- *Sliding FIR*, FIR filter with sliding window data copying, from DSPStone [16]. In addition to calculating the FIR, this loop copies values of one source array into new places in the source array and drops the last one so that the next sample could arrive into same place into memory. This is both a "storing loop" and a "reduction loop".
- *Dot product*. Dot product/FIR/convolution. A "reduction loop".
- *Matrix mul*. Matrix multiplication. Organized with dot-product-like inner loop, but with different index updates as going over rows of one matrix and columns of another matrix. A "reduction loop".
- *sum over array*. Summation of all elements of single array. A "reduction loop".

Only the steady state of the loop bodies was evaluated in these benchmarks as it is assumed the prologue and epilogue are not significantly contributing to the total execution profile. The "restrict" keyword of C99 and other similar methods were used to make sure alias analysis does not limit the scheduling of the benchmark codes.

B. Results

Figure 10 shows the number of register reads per loop iteration, and Figure 11 shows the number of register writes per loop iteration. In most of the benchmarks, both optimizations have similar reduction of register read accesses, but in the reduction type loops, loop-carried bypass gives greater improvement, eliminating most of register reads and all register writes.

In the matrix multiplication case, sum over array, and 1-byte memset loops, the optimal performance of eliminating all register accesses from the loop body is achieved by enabling both optimizations. In the dot product case without loop-invariant operand sharing, all loop-carried data could be bypassed as the two pointer updates used different function units. With loop-invariant operand sharing enabled, both pointer updates used a common function unit as they shared the constant value 4, and the loop-invariant operand sharing allocated the function units before the bypasser was run. This prevented bypassing the pointer updates. Bypassing them would have required an architecture with multiple result registers, which was not used in the evaluation.

The sliding FIR case shows an increase in register writes with loop-invariant operand sharing enabled. The reason for this is that the loop-invariant operand sharing is forcing some operations into certain function units which may disturb both local operand sharing and bypassing.

For all the benchmark cases, compared to the baseline TTA compiler, on average 36% of both register reads and register writes were eliminated by loop-carried bypassing, and 23% of register reads by loop-invariant operand sharing. Performing only loop-invariant operand sharing caused on average 3% increase in register writes. The combination of both optimizations gave on average 58% savings in register reads and 28% savings in register writes. Compared to the more limited loop-carried bypass of [6], on average 56% of register read savings and 34% of register write savings were achieved by performing both optimizations together. Compared to VLIW, on average 63% savings in register reads and 77% in register writes were achieved by enabling both optimizations.

Figure 12 shows the number of immediate transfers per loop iteration. The loop-carried bypassing performed alone had no effect on the number of immediate value transfers, as expected, but loop-invariant operand sharing has a huge effect on them. In half of the benchmark cases, all immediate value transfers inside the loop could be totally removed. The average reduction between all the benchmark cases which contain immediate values is 69%. The *SAXPY* loop which does not contain any immediate values is not included in this

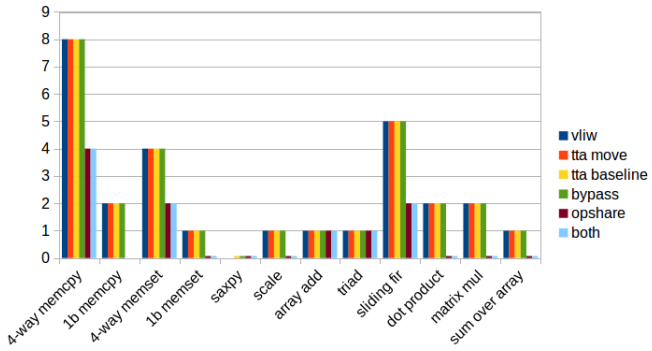

Fig. 12. Number of immediate value transfers per iteration.

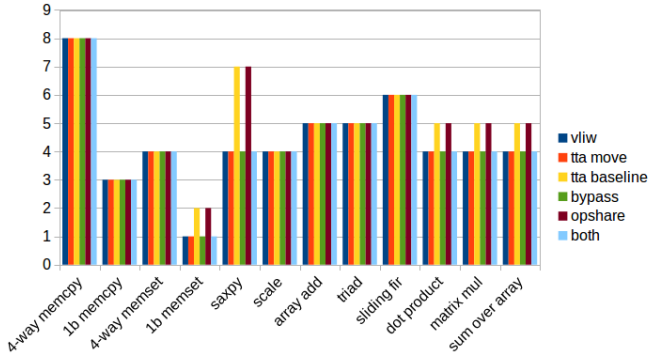

Fig. 13. Initiation interval in clock cycles.

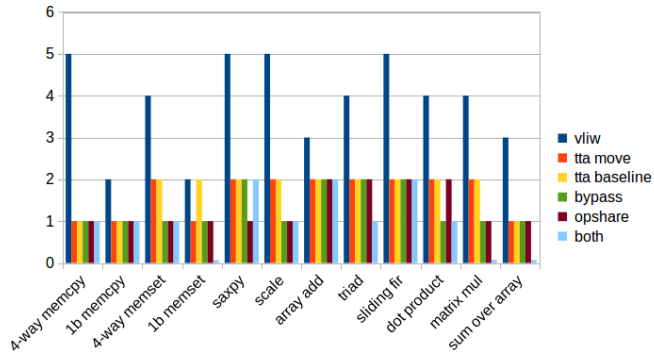

Fig. 14. Required number of register file read ports required to run the loop with optimal performance.

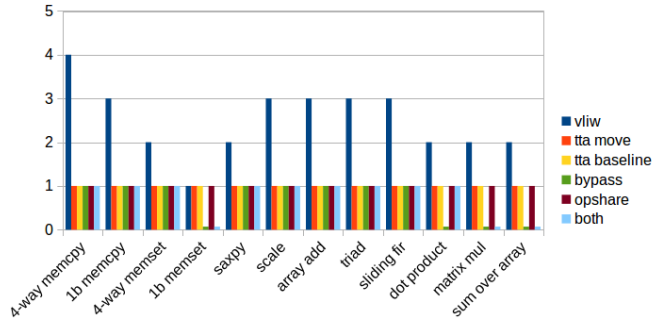

Fig. 15. Required number of register write ports required to run the loop with optimal performance.

number.

Figure 13 shows the initiation interval of the loops. This tells the length of a single loop iteration without considering the prologue or the epilogue. In the reduction loops and 1-byte memset, loop-carried bypass gives a single cycle improvement in the cycle count. The reason for this is that without over loop-edge bypass, there is one extra cycle of effective latency over the loop-carried dependencies, as the value is first stored to a register and only then read to the operation in next iteration, while with loop-carried bypass, it is bypassed and the operation of the next iteration can start immediately when the result has been calculated.

In the *SAXPY* loop, loop-carried bypassing gives a huge improvement compared to a TTA without it. The reason is that without loop-carried bypassing, the antidependence problem mentioned in Section IV prevented the loop from being interleaved with the instruction scheduler used in the evaluation, causing a very slow schedule, while loop-invariant bypassing removed some of the antidependencies causing this problem. A smarter instruction scheduler could have solved this problem better even without loop-carried bypassing. On average 13% of cycles were saved by loop-carried bypass. Loop-invariant operand sharing had no impact on the cycle count.

The number of required register file read and write ports to sustain the parallel function units in the optimal loop schedules was also evaluated. These results are shown in Figures 14 and 15. With the proposed optimizations, nine out of the twelve loops could be executed at full performance with just one register file read port. In *SAXPY* loop enabling only operand sharing seems to suffice with less register read ports (one instead of two) than when enabling both optimizations. This is because *SAXPY* with only loop-invariant operand sharing enabled is slower than *SAXPY* with both optimizations enabled. In case a better instruction scheduler would allow *SAXPY* to execute in five cycles without loop-carried bypassing, it would need two register file read ports.

Assuming at least one write and read port per register file are always needed, compared to the baseline TTA compiler without enabling either loop-carrier bypassing or loop-invariant operand sharing allows on average 21% less register file read ports. Enabling both of the optimizations allow on average 25% less. Compared to the more limited bypassing constrains of [6], the proposed optimizations allow on average 21% less register read ports. Compared to the VLIW baseline, a TTA with both of the proposed optimizations suffices with on average 65% less register file read ports.

The TTA with all the instruction schedulers could execute all the loops in optimal performance with just one register file write port. This translates to an average of 55% register file port reduction compared to the VLIW baseline, which needed one to four write ports.

C. Core-Level Power Consumption

In order to assess the overall benefits from the proposed optimizations, the power consumption of the whole core running the *Scale* loop was evaluated. We optimized the datapath of the processor for both the baseline TTA compiler and a compiler with the proposed optimizations such that both can still execute the loop on four clock cycle initiation intervals.

The processor for the proposed optimizations has five buses and one register read write port, while the baseline processor needs an additional bus and an additional register read port to reach the same cycle count without the optimizations. The processors were synthesized with Synopsys tools using 28 nm FD-SOI process technology with a 1 GHz clock rate and 1.1V supply voltage.

The baseline processor consumed 5.2 mW while executing the loop without the optimizations and 4.3 mW while executing the loop with the optimizations on. The smaller processor with both of the optimizations consumed 4.2 mW. Thus, in this loop, the optimizations directly saved 18% of power, and with the indirect power reduction of being able to utilize a smaller, more power-efficient processor datapath, total 20% of power was saved.

VIII. Future Work

Loop scheduling support is planned to be integrated to the integer linear programming-based scheduler proposed in [17] and the proposed optimizations are applicable to that. The integer linear programming-based scheduler may allow more efficient schedules for tight loops than the heuristical list scheduler used in evaluations of this work.

Support for performing loop-invariant operand sharing to only some loop-invariant values and leaving more than one free function unit for general use should be investigated, as in some cases the loop-invariant operand sharing was too aggressive, ending up disturbing later optimizations.

The proposed work only concentrates on reducing number of register accesses in the loop, but not the amount of allocated registers needed by the loop. Combining the proposed methods with the integrated allocation of [7] would be beneficial for tackling this.

IX. Conclusions

This paper proposed two new loop scheduler optimizations for Transport Triggered Architecture code generation. The optimizations improve the efficiency of software bypassing and operand sharing across loop iterations. The proposed loop optimizations can considerably decrease register file usage and number of immediate value transfers while executing tight inner loops. This reduces the number of required register file ports and transport buses to sustain the kernel execution performance.

With the proposed optimizations, nine out of the twelve tested loops could be executed at full performance with just one register file read port while sustaining the parallel function units.

Acknowledgment

The work has been financially supported by the Academy of Finland (funding decision 297548), Finnish Funding Agency for Technology and Innovation (funding decision 40142/14, FiDiPro-StreamPro) and ARTEMIS joint undertaking under grant agreement no 621439 (ALMARVI).

References

[1] H. Corporaal and M. Arnold, "Using transport triggered architectures for embedded processor design," *Integrated Computer-Aided Eng.*, vol. 5, no. 1, pp. 19–38, 1998.

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

[3] H. Kultala, J. Multanen, P. Jääskeläinen, T. Viitanen, and J. Takala, "Impact of operand sharing to the processor energy efficiency," in *2015 18th CSI Int. Symposium on Comp. Arch. and Digital Systems (CADS)*, Oct 2015, pp. 1–6.

[4] M. Lam, "Software pipelining: an effective scheduling technique for vliw machines," in *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. New York, NY, USA: ACM Press, 1988, pp. 318–328.

[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 Journal on Embedded Systems*, vol. 2013, no. 1, pp. 1–9, 2013.

[6] J. Hoogerbrugge, "Code generation for transport triggered architectures," Ph.D. dissertation, Delft University of Technology, The Netherlands, 1996.

[7] J. Janssen, H. Corporaal, and I. Karkowski, "Integrated register allocation and software pipelining," in *Proceedings of the fourth Annual Conference of the Advanced School for Computing and Imaging*. Citeseer, 1998, pp. 80–86.

[8] L. Jiang, Y. Zhu, and Y. Wei, "Software pipelining with minimal loop overhead on transport triggered architecture," in *Embedded Software and Systems, 2008. ICESS '08. Int. Conf. on*, July 2008, pp. 451–458.

[9] P. Jääskeläinen, H. Kultala, T. Viitanen, and J. Takala, "Code density and energy efficiency of exposed datapath architectures," *Journal of Signal Processing Systems*, vol. 80, no. 1, pp. 49–64, 2014.

[10] 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*, Samos, Greece, 2011, pp. 294–301.

[11] H. O. Kultala, T. T. Viitanen, P. O. Jääskeläinen, and J. H. Takala, "Aggressively bypassing list scheduler for transport triggered architectures," in *2016 Int. Conf. on Embedded Computer Syst.: Architectures, Modeling Simulation*, Samos, Greece, July 2016, pp. 253–260.

[12] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *MICRO 14: Proceedings of the 14th annual workshop on Microprogramming*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 183–198.

[13] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proc. SPIE Multimedia Mobile Devices*, San Jose, CA, USA, 2007, pp. 65 070X–1 – 65 070X–11.

[14] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, Sep. 1979.

[15] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers." 2001. [Online]. Available: http://www.cs.virginia.edu/stream/

[16] V. Zivojnović, J. M. Velarde, C. Schläger, and H. Meyr, "DSPSTONE : A DSP-oriented benchmarking methodology," in *Proc. Int. Conf. on Signal Processing and Technology*, Dallas, TX, 1994, pp. 715–720.

[17] 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.*, vol. 12, no. 4, pp. 59:1–59:22, Dec. 2015.