2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS)

Impact of Operand Sharing to the Processor Energy Efficiency

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

***Abstract*—Transport triggered architecture processors may have function unit input registers, which allow operands to be written to function units earlier than the clock cycle where the operation begins execution. 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. In this paper, the effectiveness of operand sharing is analyzed from the perspective of improving the energy-efficiency of the processor. On average of 12.0 % and at the best case of 32.4 % of register file reads could be eliminated, resulting in the best case power savings of 5.3% and energy savings of 8.8 %. In one of the 14 measured cases, operand sharing allowed a register file read port to be removed without performance penalty.**

I. Introduction

Reading the operands from register file and transferring them through an interconnect to the function units can constitute a large share of power in a processor core. Long immediate values which are transferred in the instruction word may make code size larger increasing the energy for instruction fetching. Therefore, reducing the number of register file reads and immediate transfers is important for reducing the energy consumption of the processor core.

Transport Triggered Architectures (TTA) [1] are a class of exposed data path architectures which are programmed by explicitly programming the transfers (called moves) in the buses inside the processor core. The operations are executed as a side-effect when data is moved into trigger ports of *Function Units (FU)*. Multiple moves can be scheduled into the same instruction word when they have no data dependencies and each instruction word executes for exactly one cycle, allowing the execution of multiple operations per cycle. Apart from general-purpose *Register Files (RF)*, TTAs can include local registers in FU as input and/or output ports. This allows operands to be written to the function unit earlier than the actual operation execution begins and reading the result later than the result is produced.

Operand Sharing (OS) is one of the unique low-level software optimizations allowed by the TTA programming model [2]. In OS, transport of a register or immediate value to an operand port can be omitted when the desired value is already in the FU input register since the same value was used by a previous operation. This optimization can reduce the amount of register reads and immediate transfers, resulting in savings in power consumption.

978-1-4673-8023-2/15/$31.00 copyright 2015 IEEE

Figure 1(a) shows an unscheduled sequential TTA code and the graph representation of the same code scheduled for a TTA processor is illustrated in Fig. 1(b). The code before operand sharing in Fig. 1(b) does three moves and two register reads in one cycle, so it needs at least two register read ports and three buses. The graph representation of the same code after the operand sharing is applied is depicted in Fig. 1(c). This version of the program does a total of two register reads, both on different clock cycles, and it does a maximum of two moves in the same clock cycle, so it can run on a processor with only one register read port and only two internal buses.

This paper analyzes operand sharing as an optimization to reduce the energy consumption of computations. We benchmark the effect of operand sharing on register access rate and performance on several TTA processors, and perform synthesis and power estimation on selected cases.

II. Effects of Operand Sharing on Energy

Reading register values and transferring them through the interconnect to the function units can constitute a large share

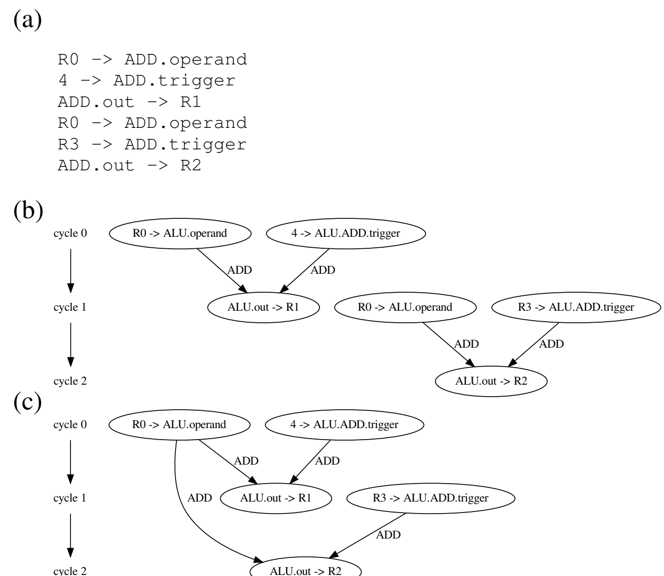

Fig. 1. (a) Original non-scheduled sequential program and (b) graph representations of the program (b) scheduled without applying operand sharing and (c) scheduled with operand sharing. In the graphs, the execution flow of the program is described from top to bottom, each ellipse represents one move, and each vertical level represents a single instruction word and single clock cycle when this instruction word is executed.

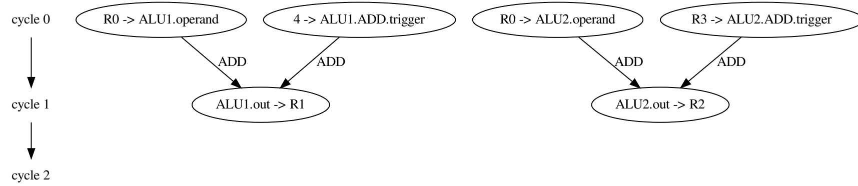

Fig. 2. Graph presentation of the program in Fig. 1 without operand sharing on a processor that has two *Arithmetic-Logical Units(ALUs)* capable of executing the ADD operation. Both additions can execute in parallel in the same cycle when operand sharing is not used. If operand sharing between the operations was used, both operations would have to execute sequentially in the same function unit, resulting in increase in cycle count.

of power consumption in a low-power processor core, and long immediate values which are transferred in the instruction word may make code size bigger which increases the energy used for instruction fetching.

The energy saved by operand sharing, however, is not directly proportional to the number of register reads, as the dynamic power consumption in CMOS devices depends on logic state transitions. An RF read eliminated by operand sharing often accesses the same address as on the previous cycle and proceeds through an identical path in the interconnection network. This kind of a read is inexpensive in terms of switching activity, and its elimination may even slightly increase the dynamic power. In this case, dropping the later operation transfers will decrease the number of register reads but does not decrease the actual energy consumption as the switching activity does not decrease. Operand sharing might still allow a narrower instruction word to be used in the case of the processor having variable-width instruction encoding, which might still save some energy.

Furthermore, operand sharing may allow the data path of the processor to be simplified by removing RF read ports or internal buses while retaining the original computation performance. Figures 1(b) and 1(c) show an example how the same code can be executed with fewer RF read ports when operand sharing is used.

III. Effects of Operand Sharing on Cycle Count

Although operand sharing cannot decrease the latency of operations, the bottleneck in scheduling a program may be the limited amount of register read ports and buses, in which case operand sharing may improve the performance. An example of such an improvement is depicted in Figures 3 and 1(c).

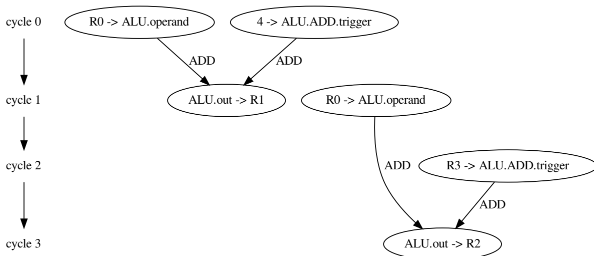

Fig. 3. Graph representation of of the program in Fig. 1 scheduled into a processor that has only one register read port. Due to the register read bottleneck, the program needs 4 cycles to execute instead of 3. Operand sharing reduces this to 3 cycles as the operand shared version has to perform only 1 register read for the second operation.

Figure 3 shows the original code on a processor with only one register read port. The code with operand sharing in Fig. 1(c) is one cycle faster.

Sharing operands between operations forces the operations to be executed in the same function unit. Sometimes this may also decrease performance as it prevents the operations to be executed in parallel. Figure 2 shows an example of the same program being executed in two cycles without operand sharing on processor with two ALUs and three register read ports.

Unfortunately operand sharing can also reduce performance by reserving a function unit input port, preventing other operations being scheduled between the operations with the shared operand. Figure 4 shows an unscheduled original code of a larger program. A graph representation of the program scheduled without operand sharing is illustrated in Fig. 5 and Fig. 6 shows the same program scheduled with operand sharing between the two ADD operations in the program. Without operand sharing the OR operation could be scheduled on cycle 1, between the ADD operations, but when the ADD operations share their first operand, the OR operation cannot be scheduled in between, as it would overwrite the operand in the input port of the function unit. Consequently, when operand sharing is used, the OR operation has to be scheduled to either after both ADD operations or before both the ADD operations.

IV. Related Work

In [2], code generation techniques for OS were proposed and it was reported that with OS 1.7 % of operand moves were eliminated, resulting in a 0-2 % cycle count decrease in their benchmarks. However, all of the eliminated operand moves were not register reads as some may have been constant values supplied through immediates. In this work, the focus was in reducing cycle count, while we focus on the potential

```
R0 -> ADD.operand
R3 -> ADD.trigger
ADD.out -> R1
R4 -> OR.operand
R5 -> OR.trigger
OR.out -> R6
4 -> SHL.operand
R1 -> SHL.trigger
SHL.out -> R7
R0 -> ADD.operand
R7 -> ADD.trigger
ADD.out -> R2
```

Fig. 4. Original non-scheduled sequential code of a larger program.

energy savings enabled by OS.

Small *Operand RFs (ORFs)* close to FUs were studied in the Stanford ELM architecture [3]. The idea is that moving operands from these smaller local RFs to the FU consumes less power than reading them from a larger central RF. Compared to operand sharing using only single FU port registers, ORF has the advantage of storing multiple operand values, which allows using OS not only for successive operations scheduled to the FU. However, reading an RF (even if small) is always more costly than always reading from a single register source.

In [4], Kim et al. propose a compiler-based solution to minimizing RF read power by FU selection and instruction scheduling in a manner that when a same operand is used multiple times, it usually goes into the same operand port of the same FU in order to minimize switching activity inside the FU. While this technique saves some switching activity, they still perform the RF read, which consumes power.

Optimizations similar to OS have been done in the hardware. For example, in [5], an optimization is proposed where hardware essentially performs OS by comparing the register indices of successive operations and reusing the old value when possible. Similarly to an FU port register OS, this solution works only between successive instructions. The obtained energy savings might be small due to the additional hardware required.

The energy savings available from other TTA-specific optimizations, software bypassing and dead result move elimination, are discussed in [6]. These optimizations are complementary to the one discussed in this paper and both can and should be used together.

MOVE-Pro is a TTA-derived architecture where there is no designated trigger port in the FUs [7]. In other words, any of the FU operand moves can be a trigger. This opens more opportunities for OS in comparison to the examined TTA template where OS cannot be applied to triggering operands. Using their proposed instruction scheduler, they report best case of 80 % RF energy reduction compared to a similar VLIW processor. However, most of the savings are assumed to come from software bypassing and dead result elimination [8]. The benefits of OS were not discussed and it is unclear how much the additional scheduling freedom's effect to the instruction width weakens the benefits from the additional optimization opportunities.

V. Evaluation

A. Benchmarks

We evaluate the performance of our methods with two benchmarks sets. On a more general-purpose DSP-type processors a subset of the *CHStone* [9] benchmark is used. This benchmark is selected since it contains a range of real-world routines, not microbenchmarks, with varying amounts of control code and instruction-level parallelism. We have used the tests *adpcm*, *gsm*, *mips*, *jpeg*, *aes*, *blowfish*, and *sha*. On a more customized processor highly-optimized OpenCL vector implementations of *Layered ORthogonal Lattice Detector (LORD)* [10] and *Linear Minimum Mean Square Error (MMSE) Detector* [11] algorithms were used. These algorithms contain cross-correlation type code where all the values in one array are multiplied with values in another array, which is considered an "operand sharing friendly" workload. The implementations were highly optimized for the best available power-to-performance ratio and the inner loops were unrolled by the programmer.

B. Processor Architectures

Five TTA type processors developed with the aid of *TTA-based Codesign Environment (TCE)* [12] were used for the benchmarks. The first processor, called *TtaSmall*, represents a small general-purpose TTA processor. It has an LSU and separate units for arithmetic, logic, shift, and multiply operations. There is a single register file with 16 32-bit registers with one read port and one write port. There is only one function unit of each type in order to minimize the processor size. The different operation classes are separated into their own function units instead of being in one monolithic ALU since in TTA, these separate units can still sometimes execute parallel code even with a very limited number of buses and register file ports. In such a case there are more opportunities for operand sharing due to the different types of operations not competing for operand ports. This processor has three buses and can start executing three operations per cycle, but in practice only 1.5 operations per cycle can be sustained, as operations require on average two moves to execute.

The second processor, *3alu_3read*, is a larger TTA with three ALUs, an LSU, a multiplier, six buses, and a register file with 16 32-bit registers, three read ports and one write port. This processor can practically sustain three operations per cycle. Three register read ports are considered reasonable for a TTA of this size, as usually one operand of an operation can be supplied either from an immediate or a bypass.

In order to investigate the relation of operand sharing with the amount of register read ports, we built modified versions of the previous processors: *TtaSmall_2read*, which adds a second register read port to *TtaSmall*, and *3alu_2read*, which reduces *3alu_3read* to only two register read ports. As a larger design case, we include a *Single Instruction Multiple Data (SIMD)* vector TTA processor which is customized for baseband algorithms, called *SDRCore*. The processor is divided into a separate vector data path for the main computation workload and a scalar data path for address and control operations. This processor is benchmarked with two baseband algorithms *LORD* and *MMSE*. We expect TTA-specific data path optimizations to be very effective in this case, as instruction fetch and decoding logic is amortized between many SIMD elements.

C. Operand Sharing Implementation

The compiler of the TCE toolset was modified to include the operand sharing optimizations. The instruction scheduler used in the benchmarks was a cycle-based list-scheduler [13] which worked in a bottom-up scheduling order. The compiler

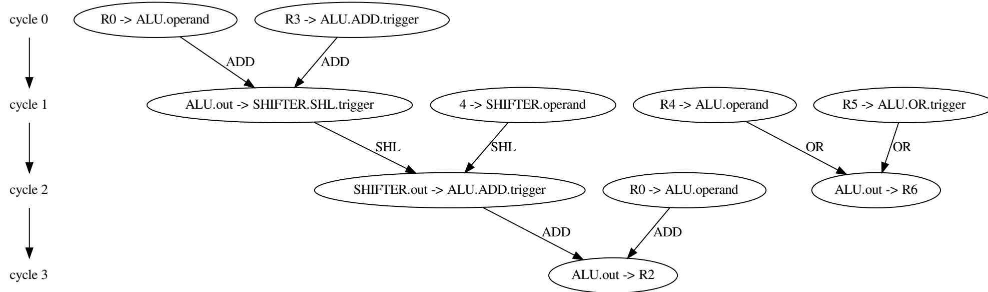

Fig. 5. Graph presentation of the program in Fig. 4 scheduled without operand sharing. The OR operation is scheduled between the two ADD operations.

also has support for software bypassing and dead result elimination. The algorithm to perform software bypassing is more aggressive than the methods used in [6] and eliminated a large fraction of register reads and writes, allowing the processor architectures to have relatively simple register files.

As only non-triggering operands may be shared on an ordinary TTA, the selection of operand order of commutative operations can have a large impact on the effectivity of operand sharing. Often a code contains many addresses to offsets of the same pointer, such as the stack pointer or the base pointer of a struct or an array. When the base pointer is given in the non-triggering operand, it can be shared, but the offsets given in the triggering operand cannot be shared. Therefore, operand swapping of commutative operations is implemented in combination with operand sharing such as to use non-triggering operands for values that are good candidates for sharing.

Before any operand move is scheduled, its data dependencies are analyzed. If a move has already been scheduled to a later cycle which uses the same value, the later move is removed from the schedule. Since equal constant values may not be detected from data dependencies, we check for them after scheduling the operand, and similarly remove the later operand move.

D. Register Read and Performance Results

Figure 7 shows clock cycle count and register read count on the *TtaSmall* processor. On this processor, operand sharing gave -0.6 % to 32.4 % decrease in register reads, average saving in register reads being 12.8 %, while performance in most tests stayed very close to performance without operand sharing. Two of the tests behaved differently from the others: in the *mips* test, operand sharing lead to an increased register read count, explained by operand sharing leading to scheduling decisions which reduced opportunities for software bypassing. In the *sha* test, operand sharing gave a noticeable 4.4 % performance increase.

Figure 8 shows clock cycle count and register read count on the *TtaSmall_2read* processor. When another register read port was added, the effectiveness of operand sharing did not change much. The processor with two read ports was slightly faster in all the tests, and the percentages of operands that could be shared was in the same range. Operand sharing gave

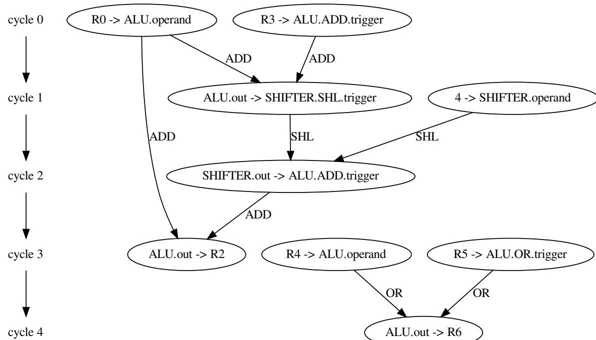

Fig. 6. Graph presentation of the program in Fig. 4 scheduled with operand sharing. Operand sharing keeps the input port of a function unit reserved preventing another operation to be scheduled between them. The OR operation has to be executed after both of the ADD operations, resulting in a slower schedule.

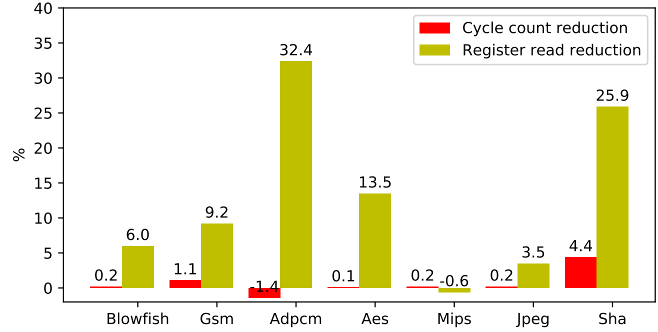

Fig. 7. Cycle and register read count reduction on *TTaSmall* processor.

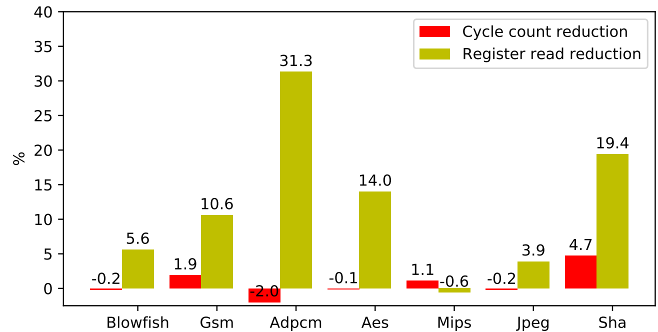

Fig. 8. Cycle count and register read count reduction on *TtaSmall_2read* processor.

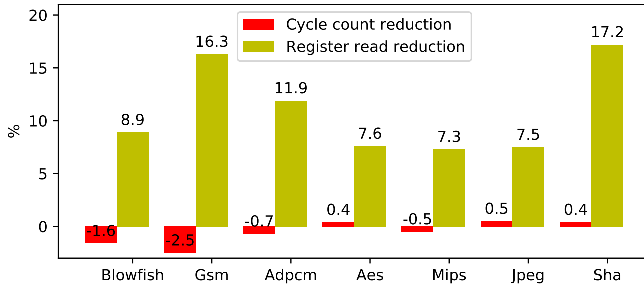

Fig. 9. Cycle and register read count reduction on *3alu_3read* processor.

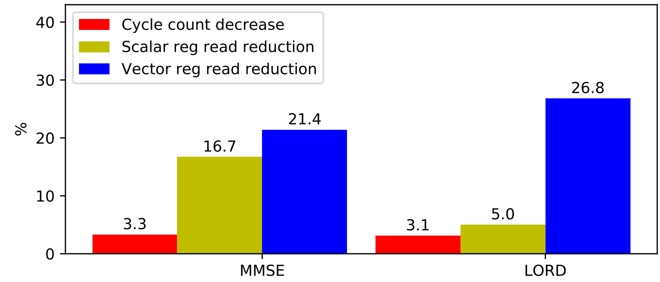

Fig. 11. Cycle and register read count reduction on *SDRCore* processor.

-0.6 % to 31.3 % savings in register reads, average saving in register reads being 12.0 %, and performance was very close to original in most tests. Two of the tests had similar interesting effect also on this architecture: in the *mips* test, operand sharing lead to an increased register read count and, in the *sha* test, operand sharing gave a noticeable 4.7 % performance increase.

By comparing the absolute results between these two architectures we can test the hypothesis of operand sharing allowing removal of register read ports. The *sha* test was the only test where the processor with only one register read port with operand sharing enabled could outperform the processor with more register file ports without operand sharing, 792205 cycles vs 793989 cycles. Thus only in this test we can for sure conclude that operand sharing can really save register file reads ports and lead to simpler processor data path. The other tests *TtaSmall_2read* performed considerably better than *TtaSmall* but operand sharing did not provide practical performance increase compared to *TtaSmall*.

Figures 9 and 10 show the cycle count and register read counts reduction when performing operand sharing on *3alu_3read* and *3alu_read2* processors. On the *3alu_3read* processor, operand sharing provides 4.7 % - 17.2 % reduction in register reads. Average reduction is 11.0 %. Operand sharing had a small effect on performance, ranging from a 1.6 % slowdown in the *blowfish* to a 0.5 % speedup in the *jpeg* test.

On *3alu_2read*, operand sharing gave a worst case of 4.4 % and a best case of 16.9 % savings on register reads, the average being 10.9 %. Operand sharing had only a small effect on performance: the best speedup, 1.3 % was achieved in *mips* test and worst slowdown in *blowfish* test, 1.6 %.

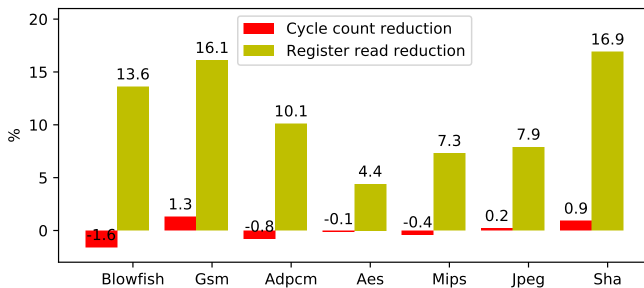

Fig. 10. Cycle and register read count reduction on *3alu_2read* processor.

On *3alu_3read*, operand sharing did not have any meaningful effect in any of the test cases to allow the removal of a register read port without reducing performance. However, in the *sha* test, performance with two register read ports and operand sharing was very close to performance with three read ports and no operand sharing, 412872 and 414414 cycles. In other tests, the *3alu_2read* processor performed relatively well even without operand sharing, and the performance improvements from operand sharing even with only two register read ports were minor. It appears that software bypassing and dead result elimination are more important optimizations than operand sharing in reducing register file port requirements.

Figure 11 shows the register file and cycle counts on *SDRCore* processor while executing *LORD* and *MMSE*. Both workloads contain cross-correlation-like computations where a single value is multiplied with many different values, and the inner loop where this happens is heavily unrolled. In this type of workload, operand sharing seems to give large savings on register file accesses, allowing 21.4 % of vector register accesses to be eliminated in the *MMSE* case and 26.8 % in the *LORD* case. Performance in both test cases was also considerably better with operand sharing enabled.

E. Power Simulation Benchmarks

The *3alu_3read*, *TtaSmall* and *SDRCore* processors were synthesized with Synopsys Design Compiler version I-2013.12, using 28nm *Fully Depleted Silicon On Insulator (FDSOI)* process technology. *Switching Activity Interchange Files (SAIF)* were produced from Modelsim simulations and used for power estimation in Design Compiler. Only the active processing time for each test case was included in the SAIFs. The designs were synthesized having leakage and dynamic power optimization and clock gating enabled. The *SDRCore* was synthesized for 1 GHz and the *3alu_3read* and *TtaSmall* for 500 MHz target clock frequency.

For *TtaSmall* processor, the average power to execute *sha* and *gsm* benchmarks, which represented the best and average cases based on the register access count, were measured to get the effect of the optimization to the overall power consumption of the processor. For *3alu_3read*, the *sha* and *adpcm* were selected for measurement as the best and average cases. For *SDRCore*, executions of both *MMSE* and *LORD*

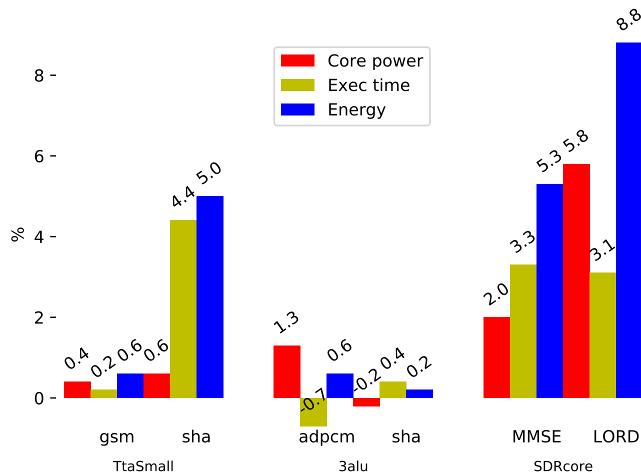

Fig. 12. Core power reduction, execution time reduction and the effect of these two on energy consumption with operand sharing. Numbers are reduction in percentage.

were simulated on the synthesized processor. All benchmarks were compared with a baseline where operand sharing was disabled, and the results are normalized to this baseline. Power estimation results are illustrated in Fig. 12.

On the *TtaSmall* processor, although the amount of register reads was reduced considerably, the power savings are only 0.4 % and 0.6 %. As the execution time of the benchmarks also decreased, the total energy spent executing the benchmarks decreased by 0.6 % and 5.0 %.

The *3alu_3read* processor showed a similar response to operand sharing. On the *adpcm* test case, there is 1.3 % decrease in power consumption, but the execution time is longer, leading to only 0.6 % lower energy consumption. On the *sha* test case, power consumption increased by 0.2 % but execution time decreased by 0.4 %, leading to only a 0.2 % total energy reduction.

On the *SDRCore* processor, operand sharing was more effective, reducing the total power consumption of the processor by 2.0 % when running *MMSE* and 5.8 % when running *LORD*. As the runtime also improved, the total energy used to execute the code decreased even more, and up to 5.3 % energy savings in *MMSE* and 8.8 % energy savings in *LORD* were observed.

VI. Conclusions

In this paper, we studied operand sharing, a low-level compiler software optimization enabled by TTA processors, focusing on its potential energy benefits.

In our measurements, we found out that the energy savings due to OS are very application-specific. Best case of 32.4 % reduction in RF reads and average of 12.0 % reduction in RF reads could be achieved. However, the power savings were at best 5.8 % and energy savings up to 8.8 %, the average savings being less than 1 %. The reason for the relatively small decrease in the average power consumption is assumed to be due operand sharing often eliminating operand transfers on consecutive clock cycles where there is a little operand switching activity to begin with.

The workload had great impact on the effectiveness. OS had a large impact on RF read count and performance in the cross-correlation-like codes *LORD* and *MMSE*, but on the more generic *CHStone* benchmark, the benefits were lower. In one test case, operand sharing allowed one of two register read ports to be removed from the architecture without performance penalty.

In our future research, we plan to continue investigating more complex compiler algorithms for utilizing operand sharing. For example, with multi-issue processors a pre-scheduling FU selection algorithm could consider operand sharing for improved sharing opportunities.

Acknowledgment

This work was funded by Academy of Finland (funding decision 253087), Finnish Funding Agency for Technology and Innovation (project ”Parallel Acceleration”, funding decision 40115/13), and ARTEMIS JU 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] J. Hoogerbrugge, “Code generation for transport triggered architectures,” Ph.D. dissertation, Delft University of Technology, The Netherlands, 1996.

[3] J. Balfour, R. Halting, and W. Dally, “Operand registers and explicit operand forwarding,” *Computer Architecture Letters*, vol. 8, no. 2, pp. 60–63, February 2009.

[4] D. Kim, D. Shin, and K. Choi, “Low power pipelining of linear systems: A common operand centric approach,” in *Proc. Int. Symp. on Low Power Electronics and Design*, Huntington Beach, CA, USA, 2001, pp. 225–230.

[5] H. Takamura, K. Inoue, and V. Moshnyaga, “Reducing access count to register-files through operand reuse,” in *Advances in Computer Systems Architecture*, ser. Lecture Notes in Computer Science, A. Omondi and S. Sedukhin, Eds. Berlin, Germany: Springer, 2003, vol. 2823, pp. 112–121.

[6] 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.

[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*, Samos, Greece, 2011, pp. 294–301.

[8] 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*, Dresden, Germany, 2012, pp. 388–393.

[9] 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.*, vol. 17, pp. 242–254, 2009.

[10] M. Siti and M. Fitz, “A novel soft-output layered orthogonal lattice detector for multiple antenna communications,” in *Proc. IEEE Int. Conf. Commun.*, vol. 4, Istanbul, Turkey, 2006, pp. 1686–1691.

[11] U. Madhow and M. Honig, “MMSE interference suppression for direct-sequence spread-spectrum CDMA,” *IEEE Trans. Commun.*, vol. 42, no. 12, pp. 3178–3188, Dec. 1994.

[12] 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.

[13] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann, 1997.