Impact of Software Bypassing on Instruction Level Parallelism and Register File Traffic

Vladimír Guzma, Pekka Jääskeläinen, Pertti Kellomäki, and Jarmo Takala

Tampere University of Technology, Department of Computer Systems
P.O. Box 527, FI-33720 Tampere, Finland
{vladimir.guzma,pekka.jaaskelainen,pertti.kellomaki,jarmo.takala}@tut.fi

Abstract. Software bypassing is a technique that allows programmer-controlled direct transfer of results of computations to the operands of data dependent operations, possibly removing the need to store some values in general purpose registers, while reducing the number of reads from the register file. Software bypassing also improves instruction level parallelism by reducing the number of false dependencies between operations caused by the reuse of registers. In this work we show how software bypassing affects cycle count and reduces register file reads and writes. We analyze previous register file bypassing methods and compare them with our improved software bypassing implementation. In addition, we propose heuristics when not to apply software bypassing to retain scheduling freedom when selecting function units for operations. The results show that we get at best 27% improvement to cycle count, as well as up to 48% less register reads and 45% less register writes with the use of bypassing.

1 Introduction

Instruction level parallelism (ILP) requires large numbers of function units (FU) and registers, which increases the size of the bypassing network used by the processor hardware to shortcut values from producer operations to consumer operations, producing architectures with high energy demands. While increase in explorable ILP allows to retain performance on lower clock speed, energy efficiency can also be improved by limiting the number of registers and register file (RF) reads and writes [1]. Therefore, approaches aiming to reduce register pressure and RF traffic by bypassing the RF and transporting results of computation from one operation to another directly provide cost savings in RF read. Some results may not need to be written to registers at all, resulting in additional savings. Allowing values to stay in FUs reduces further the need to access a general purpose RF, while keeping FUs occupied as a storage for values, thus introducing a tradeoff between the number of registers needed and number of FUs.

Programs often reuse GPRs for storing different variables. This leads to economical utilization of registers, but it also introduces artificial serialization constraints, so called "false dependencies". Some of these dependencies can be avoided in case all uses of a variable can be bypassed. Such a variable does not

need to be stored in a GPR at all, thus avoiding false dependencies with other variables sharing the same GPR. In this paper we present several improvements to the earlier RF bypassing implementations. The main improvements are listed below.

- In our work we attempt to bypass also variables with several uses in different cycles, even if not all the uses could be successfully bypassed.
- We allow variables to stay in FU result registers longer, and thus allow bypassing at later cycles, or early transports into operand register before other operands of same operation are ready. This increases the scheduling freedom of the compiler and allows for further decrease in RF traffic.
- We use a parameter we call "the look back distance" to control the aggressiveness of the software bypassing algorithm. The parameter defines the maximum distance between the producer of a value and the consumer in the scheduled code that is considered for bypassing.

2 Related Work

Effective use of RF bypassing is dependent on the architecture's division of work between the software and the hardware. In order to bypass the RF, the compiler or hardware logic must be able to determine what are the consumers of the bypassed value, effectively requiring data flow information, and how the direct operand transfer can be performed in hardware.

While hardware implementations of RF bypassing may be transparent to programmer, they also require additional logic and wiring in the processor and can only analyze a limited instruction window for the required data flow information. Hardware implementations of bypassing cannot get the benefit of reduced register pressure since the registers are already allocated to the variables when the program is executing. However, the benefits from reduced number of RF accesses are achieved. Register renaming [2] also produces the increase in available ILP from removal of false dependencies. Dynamic Strands presented in [3] are an example of an alternative hardware implementation of RF bypassing. Strands are dynamically detected atomic units of execution where registers can be replaced by direct data transports between operations. In EDGE architectures [4], operations are statically assigned to execution units, but they are scheduled dynamically in dataflow fashion. Instructions are organized in blocks, and each block specifies its register and memory inputs and outputs. Execution units are arranged in a matrix, and each unit in the matrix is assigned a sequence of operations from the block to be executed. Each operation is annotated with the address of the execution unit to which the result should be sent. Intermediate results are thus transported directly to their destinations.

Static Strands in [5] follows earlier work [3] to decrease hardware costs. Strands are found statically during compilation, and annotated to pass the information to hardware. As a result, the number of required registers is reduced already in compile time. This method was however applied only to transient operands with a single definition and single use, effectively up to 72% of dynamic

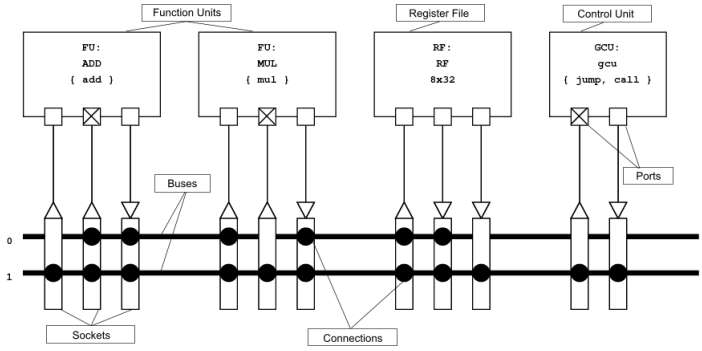


Fig. 1. Example of TTA concept

```
add r3,r1,r2
add r5,r4,r3
mul r1,r3,r5
```
(a)

```
r2 -> add.t; r1 -> add.o;
add.r -> r3; r4 -> add.o;
r3 -> mul.o; r3 -> add.t;
add.r -> r5; ...;
...; r5 -> mul.t;
mul.r -> r1; ...;
```
(b) Without bypassing (.o and .t denoting inputs and .r result)

```
r2 -> add.t; r1 -> add.o;
add.r -> mul.o; r4 -> add.o;
add.r -> add.t; ...;
...; add.r -> mul.t;
mul.r -> r1; ...;
```
(c) Register r3 bypassed twice and r5 once

Fig. 2. Example of schedule for two *add* and one *mul* operations for Risc like architecture (a) and TTA architecture (b)(c) from Fig. 1

integer operands, bypassing about half of them [5]. Dataflow Mini-Graphs [6] are treated as atomic units by a processor. They have the interface of a single instruction, with intermediate variables alive only in the bypass network.

Architecturally visible "virtual registers" are used to reduce register pressure through bypassing in [7]. In this method, a virtual register is only a tag marking a data dependence between operations without having physical storage location in the RF. Software implementations of bypassing analyze code during compile time and pass to the processor the exact information about the sources and the destinations of bypassed data transports, thus avoiding any additional bypassing and analyzing logic in the hardware. This requires an architecture with an exposed bypass network that allows such direct programming, like the *Transport Triggered Architectures (TTA)* [8], *Synchronous Transfer Architecture (STA)* [9] or *FlexCore* [10]. The assignment of destination addresses in an EDGE architecture corresponds to software bypassing in a transport triggered setting. Software only bypassing was previously implemented for TTA architecture using the experimental MOVE framework [11] [12]. TTAs are a special type of VLIW architectures as shown on Fig. 1. They allow programs to define explicitly the operations executed in each FU, as well as to define how (with position in instruction defining bus) and when data is transferred (*moved*) to each particular

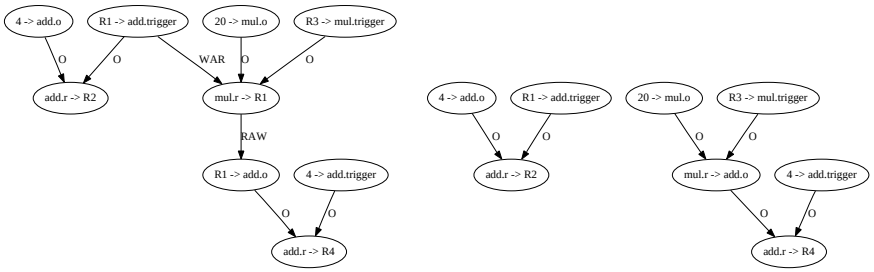


Fig. 3. DDG: a) without bypassing b) with bypassing and dead result move elimination

port of each unit, as shown on Fig. 2(b)(c). A commercial application of the paradigm is the Maxim MAXQ general purpose microcontroller family [13].

With the option of having registers in input and output ports of FUs, TTA allows the scheduler to move operands to FUs in different cycles and reading results several cycles after they are computed. Therefore the limiting factor for bypassing is the availability of connections between source FU and destination FUs. The MOVE compiler did not actively software bypass, but performed it only if the "opportunity arose".

3 Software Bypassing

Instruction level parallelism (ILP) is a measure of how many operations in a program can be performed simultaneously. Architectural factors that prevent achieving the maximum ILP available in a program include the number of buses, the number of FUs, as well as the size of and the number of read and write ports in RFs. Software bypassing helps to avoid some of these factors. Figure 3(a) shows a fragment of a Data Dependence Graph (DDG). In the example, *R1* is used as an operand of the first *add*, and also as a store for the result of the *mul*, subsequently read as an operand of the second *add* ("read after write" dependence, *RAW*). This reuse of *R1* creates a "write after read" dependence between read and write of *R1*, labeled *WAR*. When the result of the *mul* operation is bypassed directly into the *add* operation, as shown in Fig. 3(b), the *WAR* dependence induced by the shared register *R1* disappears. Since the DDG fragments are now independent of each other, the scheduler has more freedom in scheduling them. Careless use of software bypassing by the instruction scheduling algorithm can also decrease performance. One of the limiting factors of ILP is the number of available FUs to perform the operations in parallel. Using the input and result registers of an FU as temporary storage renders the unit unavailable for other operations. We have identified a parameter, *look back distance*, for controlling the tradeoff. The parameter defines the distance between a move that writes a value into the RF, and a subsequent move that reads an operand from the same register. The larger the distance, the larger number of register accesses can

```
1: function ScheduleOperation(inputs, outputs, lookBack)
2:     success := false
3:     cycle := 0
4:     while not success do
5:         ScheduleASAP(cycle, inputs)
6:         TryBypassOperands(lookBack, inputs)
7:         success := ScheduleASAP(cycle, outputs)
8:         if success then
9:             RemoveDeadResults(inputs)
10:        else
11:            Unschedule(inputs)
12:            Unschedule(outputs)
13:            cycle := cycle + 1
14:        end if
15:    end while
16: end function
```

Fig. 4. Schedule and bypass an operation

be omitted. However, the FUs will be occupied for longer, which may increase the cycle count. Conversely, smaller distance leads to smaller number of register reads and writes removed, but more efficient use of FUs.

Multiported RFs are expensive, so architects try to keep the number of register ports low. However, this can limit the achievable ILP, as register accesses may need to be spread over several cycles. Software bypassing reduces RF port requirements in two ways. A write into a RF can be completely omitted, if all the uses of the value can be bypassed to the consumer FUs (*dead result move elimination* [14]).

Reducing the number of times the result value of a FU is read from a register also reduces pressure on register ports. With less simultaneous RF reads there is need for less read ports. This reduction applies even when dead result move elimination cannot be applied because of uses of value still later in code. The additional scheduling freedom gained by eliminating false dependencies also contributes to reduction of required RF ports. The data transports which still require register reads or writes have less restrictions and could be scheduled earlier or later, thus reducing the bottleneck of limited RF ports available in single cycle.

Our instruction scheduler uses operation-based top-down list scheduling on a data dependence graph, where an operation becomes available for scheduling once all producers of its operands have been scheduled [15]. Figure 4 outlines the algorithm to schedule operands and results of a single operation. Once all the operands are ready, all input moves of operation are scheduled (Fig. 4, line 5). Afterwards, bypassing is attempted for each of the input operands that reads register, guided by the look back distance parameter (line 6). After all the input moves have been scheduled, the result moves of operation are scheduled (line 7). After an operation has been successfully scheduled, the algorithm removes writes

```
1: function TryBypassOperands(inputs, lookBack)
2:     for each candidate in inputs do
3:         if candidate reads constant then
4:             continue
5:         end if
6:         producer = producer of value read by candidate
7:         limitCycle = producer.cycle + lookBack
8:         if limitCycle < candidate.cycle then
9:             continue                          ▷ Producer move too far away
10:        end if
11:        Unschedule(candidate)
12:        mergedMove := Merge(producer, candidate)
13:        success := ScheduleASAP(mergedMove)
14:        if not success then
15:            Restore producer and consumer
16:        end if
17:    end for
18:    return true
19: end function
```

Fig. 5. Software bypassing algorithm

into register that will not be read (line 9). If scheduling of result moves fails, possibly due to writer after read or write after write dependency on other already scheduled moves, all of the scheduled moves of operation are unscheduled and scheduling restarts with higher starting cycle (lines 11 to 13).

Figure 5 shows the outline of our bypassing algorithm. When considering bypassing of register, algorithm computes the distance between the producers result write into the RF and the read of a register, scheduled previously (Fig. 4, line 5). If this distance is larger than specified (Fig. 5, line 8), bypassing is not performed. Otherwise, the candidate moves is unscheduled and a new move is created with the producer's FU result port as the source and consumer's FU operand port as the destination. Such a merged move is then scheduled to as early as possible cycle with respect to data dependencies and restrictions of the resources induced by the already scheduled code. If scheduling fails, the original producer and the costumer are restored and algorithm continues with the next input operand. Figure 2 shows scheduled code without bypassing(b) and with bypassing(c).

4 Experimental Setup

In order to implement a purely software solution to RF bypassing, we based our experimental setup on the TTA architecture template. For comparing the effect of software bypassing on the number of RF reads, writes, and on ILP, we varied the look back distance used by the algorithm. We explored the effectiveness of software bypassing with limited RF port resources by defining two TTA

Table 1. Resources of architectures (a) and benchmark applications (b) used in our experimental setup

(a)

Machine name	"big"	"small"	"wide"
Registers in RF	128	128	128
RF read ports	10	5	5
RF write ports	5	1	1
Count of FUs	7 (11)	7 (11)	14 (22)
$\sum$ of FUs inputs	14 (27)	14 (27)	28 (54)
$\sum$ of FUs results	7 (12)	7 (12)	14 (24)
Count of buses	10	10	10

(b)

adpcm	ADPCM routine test
fft	In-place radix-4 DIT FFT
jpeg	JPEG decoding
mpeg4	Mpeg4 decoding (192x192)
Tremor	Ogg Vorbis decoding (40KB)

Table 2. Number of dynamic register reads and writes and ratio reads/writes (r/w) for small machine: a) without bypassing, b) with best bypassing

(a)

	reads	writes	r/w
adpcm	203053	172103	1.17
fft	84493	37143	2.27
jpeg	11401300	7628810	1.49
mpeg4	311915000	190901000	1.63
Tremor	301137000	207185000	1.45

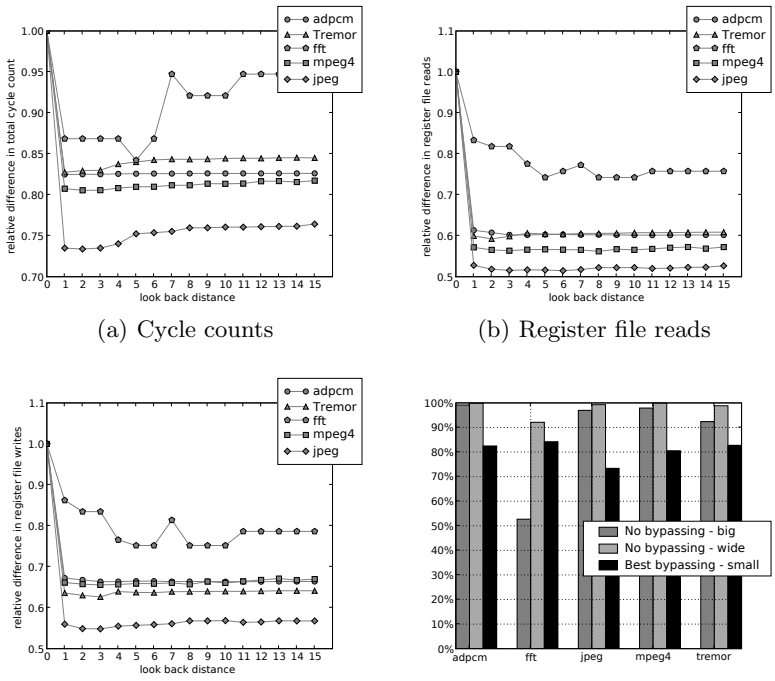
(b)

	reads	writes	r/w
adpcm	122266	114219	1.07
fft	62728	27927	2.24
jpeg	5877870	4182930	1.40
mpeg4	175806000	125165000	1.40
Tremor	180258000	129774000	1.38

processors with different RF resources, as described in Table 1(a). The machine we refer to as "big" allowed us to see to what extent a large enough number of ports in the RF defeats the benefits of software bypassing. The machine named "small" is identical to "big" except for the much reduced number of ports in the RF. This machine should show how much the proposed software bypassing algorithm is able to reduce the need for additional RF ports while maintaining the performance. We also explored the effect of software bypassing with different number of FUs. The machine referred to as "wide" has identical number of registers and RF read and write ports as "small", but double the number of FUs as in the "small" machine. This allowed us to investigate tradeoffs in storing results longer in FUs by varying look back distance values. The benchmarks are listed in Table 1(b).

5 Results

Figure 6(a) shows the comparison for a small machine with look back distances of one to fifteen, against a schedule without software bypassing. The results show that for most of the benchmarks, the performance for different look back distances varies, with a general tendency for better results with smaller look back distance. Improvements in the cycle count for the best look back distance ranges from 27% to 16%.



(a) Cycle counts (b) Register file reads

(c) Register file writes (d) Cycle count comparison

Fig. 6. Relative comparison: (a), (b), (c) small machine with different look back distances and (d) small, big and wide machines vs. small machine without bypassing (100%)

Figures 6(b) and 6(c) compare RF reads and writes for a small machine with look back distances 1–15, against a schedule without bypassing, with the *jpeg* benchmark having the highest decrease in RF reads of 48%, and RF writes of 45%, and *fft* having a smallest decrease of 25% for reads and writes. The best performing look back distance in terms of cycle count does not correspond with best results in decrease of the number of RF reads and writes. This supports our claim from Section 3, that too aggressive use of bypassing may lead to FUs being occupied for too long time, forcing the scheduler to delay other operations on the same FU, while on the other hand, saving more of the register reads and possibly writes.

Table 2(a) shows the ratio between register reads and writes without bypassing, ranging from 1.17 to 2.27. Table 2(b) shows the same ratio with best bypassing for reducing register accesses, ranging from 1.07 to 2.24. This decrease indicates that more register reads than writes were bypassed, thus not only transient variables were bypassed, but also variables with multiple reads.

Figure 6(d) takes the cycle counts of the small machine without software bypassing, and compares them with the big machine without software bypassing,

the wide machine without software bypassing, and with bypassing using the best performing look back distance for the small machine for each of the benchmarks (see Fig. 6(a)). The limited number of RF read and write ports in the small machine causes an increase in cycle counts for the small machine without software bypassing due to serialization of RF accesses. In addition, added FUs in the wide machine allowed the compiler to exploit ILP better, also providing a better cycle count than the small machine.

With software bypassing, however, the cycle counts on the small machine are in most cases smaller than on the big machine and the wide machine, with *jpeg* having the highest decrease of 24% compared to the big machine. The loop-oriented *fft* benchmark is an exception. This is due to the current bypassing algorithm being unable to handle variable uses crossing loop boundaries. Therefore, the RF bottleneck could be avoided poorly in this case due to the need to fill the FUs at the beginning of the loop and read the results to GPRs at the end of the loop. However, the presented software bypassing algorithm decreased cycle count for *fft* by 16%, narrowing the gap between the big and the small machines from 48%, without bypassing, to 32% with bypassing.

6 Conclusions

This work explored some of the benefits that software bypassing offers to improve performance and reduce the cost of embedded applications implemented using TTA processors. In particular, we explored the effect of the bypassing look back distance on performance, and showed that using software bypassing, an architecture with a limited number of RF ports can outperform an architecture with more RF ports or additional FUs without software bypassing. We also showed that while small look back distance leads to higher savings in cycle counts, larger distance leads to saving more register reads and writes.

In the future we plan to explore the possibilities of software bypassing in global instruction scheduling and cyclic scheduling, bypassing whole subgraphs of data dependence graph atomically. We predict that software bypassing is most beneficial when done before or during register allocation, which will be verified by experiments. In addition, we plan to evaluate the effect of reduced number of RF reads and writes on energy savings.

This work was supported by the Academy of Finland, project 205743.

References

1. Hoogerbrugge, J., Corporaal, H.: Register file port requirements of Transport Triggered Architectures. In: MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture, New York, NY, USA, ACM Press (1994) 191–195
2. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, San Francisco, US (1998)
3. Sassone, P.G., Wills, D.S.: Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In: Proc. IEEE/ACM Int. Symp. Microarchitecture, Washington, DC, USA, IEEE Computer Society (2004) 7–17

4. Burger, D., Keckler, S.W., McKinley, K.S., Dahlin, M., John, L.K., Lin, C., Moore, C.R., Burrill, J., McDonald, R.G., Yoder, W., the TRIPS Team: Scaling to the end of silicon with EDGE architectures. Computer **37**(7) (2004) 44–55
5. Sassone, P.G., Wills, D.S., Loh, G.H.: Static strands: Safely exposing dependence chains for increasing embedded power efficiency. Trans. on Embedded Computing Sys. **6**(4) (2007) 24
6. Bracy, A., Prahlad, P., Roth, A.: Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In: MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (2004) 18–29
7. Yan, J., Zhang, W.: Virtual registers: Reducing register pressure without enlarging the register file. In Bosschere, K.D., Kaeli, D.R., Stenström, P., Whalley, D.B., Ungerer, T., eds.: HiPEAC. Volume 4367 of Lecture Notes in Computer Science., Springer (2007) 57–70
8. Corporaal, H.: Microprocessor Architectures: from VLIW to TTA. John Wiley & Sons, Chichester, UK (1997)
9. Cichon, G., Robelly, P., Seidel, H., Bronzel, M., Fettweis, G.: Compiler scheduling for STA-processors. In: PARELEC '04: Proceedings of the international conference on Parallel Computing in Electrical Engineering, Washington, DC, USA, IEEE Computer Society (2004) 45–60
10. Thuresson, M., Sjalander, M., Bjork, M., Svensson, L., Larsson-Edefors, P., Stenstrom, P.: Flexcore: Utilizing exposed datapath control for efficient computing. In: Proc. Int. Conf. on Embedded Computer Systems: Architectures, Modeling and Simulation. Samos, Greece. (2007) 18–25
11. Corporaal, H., Mulder, H.J.: Move: a framework for high-performance processor design. In: Proc. ACM/IEEE Conf. Supercomputing, Albuquerque, NM (1991) 692–701
12. Janssen, J., Corporaal, H.: Partitioned register file for TTAs. In: Proc. 28th Annual Workshop on Microprogramming (MICRO-28). (1996) 303–312
13. Maxim Corporation: MAXQ microcontroller home page at `http://www.maxim-ic.com/products/microcontrollers/maxq.cfm` (2007)
14. Corporaal, H., Hoogerbrugge, J.: Code generation for Transport Triggered Architectures. In: Code Generation for Embedded Processors. Springer-Verlag, Heidelberg, Germany (1995) 240–259
15. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc. (1986)