SAFE PRE-PASS SOFTWARE BYPASSING FOR TRANSPORT TRIGGERED PROCESSORS

Pertti KELLOMÄKI   Vladimír GUZMA   Jarmo TAKALA
Tampere University of Technology, Finland
P.O. Box 553, FIN-33101 Tampere, Finland
Fax +358 3 3115 4561. Email {firstname.lastname}@tut.fi

Abstract: **The Transport Triggered Architecture is an architectural style where the internal transport buses of a processor are exposed in the instruction set. Since software has full control over data transports, data can be directly transfered between functional units, an optimization known as *software bypassing*.**
Software bypassing can potentially reduce the need for general purpose registers, because temporary values can be kept in functional units. Taking advantage of the reduced register pressure requires software bypassing to be done before register allocation and scheduling. In this paper we analyze under which conditions software bypassing can be done on a Data Dependence Graph without compromising the schedulability of the graph.

I. INTRODUCTION

The *Transport Triggered Architecture* (TTA) [4] is an architectural style for processor design that allows for easy customization of the computational resources and the topology of the internal interconnection network. The TTA concept is similar to VLIW [6] in that instruction level parallelism is exposed in the Instruction Set Architecture (ISA) of the processor.

Energy efficiency is becoming an increasingly important consideration. A popular solution is the use of Application Specific Instruction Set Processors (ASIP), which include features not found in general purpose processors such as an ability to extend the instruction set with application specific operations. An ASIP processor can be considerably more energy efficient for a particular computing task than a general purpose processor, while being more flexible than a full ASIC solution.

Because of the ease of customization, TTA processors are well suited to be used as ASIPs. The resources of a TTA processor can include functional units that speed up particular applications, and unnecessary connections can be omitted from the interconnection network.

Software bypassing is an optimization made possible by the architecture of TTA processors. Because the programmer (or the compiler) has full control over the transport buses of the processor, values can be moved directly between functional units. Many temporary values are only used once, and if such values can be bypassed, they need not be written in registers at all. Software bypassing can thus potentially reduce register pressure, i.e. the maximum number of registers needed simultaneously.

The decrease in register pressure can only be utilized if software bypassing is done before register allocation in a compiler. However, too aggressive bypassing can result in resource starvation in the later stages of the compilation. In this paper we analyze the conditions under which software bypassing can be safely performed before resource allocation in a compiler.

The rest of the paper is structured as follows. Section II explains Transport Triggered Architectures in more detail, and Section III describes the TCE toolkit, a hardware/software codesign environment based on TTAs. Section IV discusses software bypassing, and Section V analyzes how software bypassing affects the schedulability of a Data Dependence Graph on a given TTA processor. Related work is reviewed in Section VI, and Section VII concludes the paper.

II. TRANSPORT TRIGGERED ARCHITECTURE

In this section we describe the Transport Triggered Architecture.

Computer programs often contain very fine level parallelism called Instruction Level Parallelism (ILP), where some consequtive instructions do not have mutual data dependences, and so could be executed simultaneously.

There are essentially two ways to exploit ILP. *Superscalar* processors hide the parallelism from the programmer and let the hardware determine the exact order in which instructions are executed. This is done very successfully in current general purpose processors, with the expense of complicated control logic.

The other approach is to expose the parallelism in the instruction set, leading to *Explicitly Parallel Instruction Computing* (EPIC). In EPIC, the program binary explicitly indicates which instructions are independent of each other, and could be executed simultaneously.

VLIW [6] processors are a typical example of the EPIC approach. A VLIW processor is composed of several independent execution units, and each instruction provides an operation to be executed by each of the units at the same cycle. A VLIW processor can thus provide great computational power with relatively simple hardware, provided that the compiler can effectively utilize the execution units. In practice, the EPIC approach works best in application areas where control flow is relatively predictable, such as numerical computation and digital signal processing.

The Transport Triggered Architecture takes the VLIW approach one step further by exposing the interconnection network of the processor to the compiler. The instruction word of a TTA processor contains a slot for each internal transport bus of the processor, determining which transport takes place on the bus. Computation takes place as a side effect of the data transports.

Figure 1 illustrates a simple TTA processor consisting of five functional units, two register files, and a control unit. The units are connected by three buses. Some of the ports are marked with an “x”, indicating that they are *triggering* ports. Writing a value into a triggering port causes the functional unit to start computation, hence the term “Transport Triggered”. After the latency of the operation on the given unit, the result of the computation appears in the result register of the unit.

There are few constraints on how data transports can be arranged, so the designer has plenty of freedom in designing the application specific functional units. When application specific instructions are retrofitted into an existing processor design, the new instructions must adhere to the architectural constraints of the design. For example, instructions may only be able to read one or two registers, and to write the result into one register. A TTA processor on the other hand can easily accommodate multi-input multi-output operations, long latency functional units and so forth.

Two aspects of a TTA processor are of particular interest. The buses are a limiting factor for the speed of the processor. The more connections on a bus, the more load it has, which affects how fast the bus can be driven up or down. The number of read and write ports a register file has affects the area of the register file considerably. With software bypassing it is not necessary to route all values via registers, so a register file with a smaller number of ports can be used.

III. THE TCE TOOLKIT

The TTA-based Codesign Environment (TCE) [9] is a toolset that uses the TTA paradigm as a platform for developing application specific instruction set processors. The toolkit is a collection of tools that allows a designer to create a custom architecture, compile applications written in the C language for the architecture, simulate execution of applications on the target, and evaluate the cost in terms of execution cycles, area and energy. Both command line and graphical tools are provided to help designer.

TCE can be used for designing processors manually. The designer uses a graphical tool (shown in Figure 1) to instantiate an architecture template, and populates it with resources. TCE provides a library of predefined processor units (register files, functional units, long immediate units), and the designer can also create new application specific units. The units are connected to each other with transport buses using the graphical tool.

A processor design can then be evaluated by compiling applications for it and simulating them. Estimates of area and energy for the design can also be obtained at this point. Once the designer is satisfied with the architecture, the final phase of TCE design flow takes place. Program image and processor generation includes generating the HDL files for the selected architecture, and generating the bit image of the application. The processor architecture can be synthesized from the HDL files using third party tools.

In order to overcome the traditional disadvantage of long instructions in VLIW designs, emphasized in the case of TTA, instruction compression is often used in this step. The binary image of the application can be compressed automatically and a corresponding decompressing block can be added to the control unit of the target processor.

In addition to the fully manual workflow, TCE can also be used in a semi-automated fashion. Using the design space explorer, the designer can automatically create an architecture based on the requirements of the application.

Starting from an initial architecture provided by the designer, the design space exploration continues with automated addition and removal of the resources. Each of the architectures generated is evaluated during this process, and the designer is given a set of architectures, and the associated information about the cycle counts required for executing the application, as well as area and energy estimates.

IV. SOFTWARE BYPASSING

In most processors, data flow between instructions takes place via registers. This can cause a pipelined processor to stall if an instruction needs to wait for a previous instruction to write its result into a register. To prevent stalls, *bypassing* is used to short-circuit values to their consumers before register writes have completed. An analogous technique can be used in TTAs, as the compiler is able to move data directly from one functional unit to another, an optimization known as *software bypassing*.

By eliminating redundant data transports, software bypassing can improve the efficiency of a program and reduce energy consumption. It can also decrease register pressure as it may not be necessary to store some values in registers at all.

IV.1 A TTA Oriented Data Dependence Graph

In the context of conventional (operation triggered) processors, a Data Dependence Graph (DDG) is a directed acyclic graph, where nodes correspond to operations, and weighted edges correspond to dependences between operations (e.g. [1], p. 722). Dependences are induced by shared storage locations (registers or memory), and the weights model latencies of operations.

This representation cannot express bypassing in a convenient way. We define a TTA oriented DDG as follows. The set of nodes $\mathscr{N}$ consists of two disjoint sets, set $\mathscr{M}$ of *move nodes* and set $\mathscr{O}$ of *operation nodes*. Move nodes are labeled with data transports, and operation nodes are labeled with operations.

The set of edges $\mathscr{E}$ likewise consists of two disjoint sets, set $\mathscr{I}$ of *internal edges* and set $\mathscr{X}$ of *external edges*. Edges are labeled with architecturally visible storage locations. Internal edges model data dependences between the operands and results of an operation (i.e. data flow within a functional unit), and external edges model dependences between operations, induced by registers and memory locations.

Figure 2 depicts a simple TTA DDG. Internal edges are shown as dashed arrows and operation nodes as dashed ovals. Move nodes are labeled with data transports. For example, the label “$r2 \rightarrow add.o$” denotes that a value is read from register $r2$ and written to the operand port $add.o$ of the *add* operation.

Each node in a TTA DDG corresponds to a write into an architecturally visible register. The move nodes are explicit transports on the buses of the processor, and the operation nodes correspond to writes into the result registers of functional units.

Before resource allocation has taken place, registers in a DDG are virtual registers, and operations can be thought of as virtual functional units. Resource allocation then binds the virtual registers and functional units to the physical resources of a particular processor.

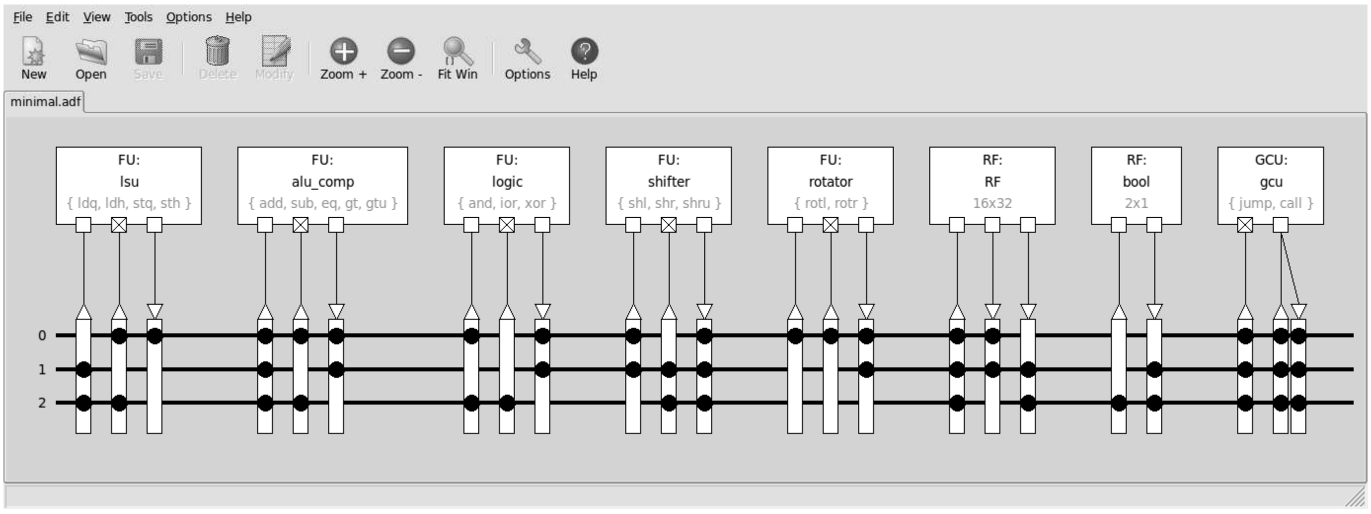
File Edit View Tools Options Help
New
Open
Save
Delete
Modify
Zoom +
Zoom -
Fit Win
Options
Help
minimal.adf
FU: lsu { ldq, ldh, stq, sth }
FU: alu_comp { add, sub, eq, gt, gtu }
FU: logic { and, ior, xor }
FU: shifter { shl, shr, shru }
FU: rotator { rotl, rotr }
RF: RF 16x32
RF: bool 2x1
GCU: gcu { jump, call }
0
1
2

Figure 1. A simple TTA processor.

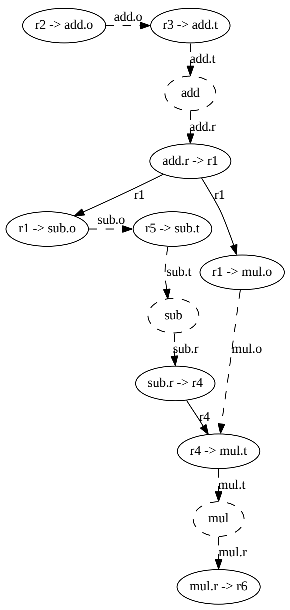

Figure 2. A simple TTA DDG .

IV.2 Bypassing on the Data Dependence Graph

Bypassing is easy to express on the TTA DDG. Figure 3 illustrates how the graph is transformed. Originally (Fig. 3, left), the operation op produces a result in the result register $op.r$, and the value in $op.r$ is transported to register R. A subsequent move reads the value in R and transports it to destination D.

When $op.r$ is bypassed directly to D, the graph is transformed according to Figure 3 (right). The label of $R \rightarrow D$ is changed to $op.r \rightarrow D$, and the node is made to depend directly on the operation node op.

If there are any dependences emanating from the nodes, they remain unaffected by the transformation. Move nodes can become redundant as the result of bypassing. If the value written into a register is never read from there, the move to register can be safely omitted. Figure 4 illustrates how software bypassing modifies the DDG of Figure 2. By-

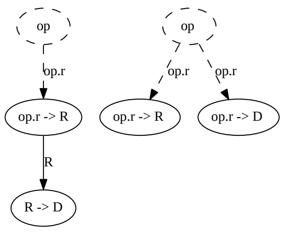

Figure 3. Bypass transformation.

passing register $r4$ makes the node $sub.r \rightarrow r4$ redundant, so it can be safely removed from the graph.

V. SOFTWARE BYPASSING AND SCHEDULABILITY

When software bypassing makes some moves to registers redundant, it can reduce the need for registers. However, overzealous bypassing can result in a DDG that cannot be scheduled on a given TTA processor. In this section, we derive conditions under which software bypassing can be performed without compromising schedulability.

The reason why schedulability can be compromised is that bypassing uses the result registers of functional units to store intermediate values. If a DDG requires more values to be kept in result registers than there are functional units, the DDG cannot be scheduled.

If intermediate results are always stored in registers, functional unit starvation cannot occur. The scheduler can trivially schedule operations nodes and the associated move nodes one after another using topological sorting, so that only one functional unit is in use at any given cycle. Since all values come from registers, all the moves to operand and trigger ports are ready to be scheduled. Once the results are written back to registers, the next operation is ready to be scheduled.

Similar reasoning can be applied to groups of operations. If a region in a DDG reads all of its inputs from registers and writes all of its outputs to registers, all its predecessors in the DDG can be scheduled before the region, and all its successors be scheduled after the region.

Definition 1. *A* bypass region *is a maximal connected sub-*

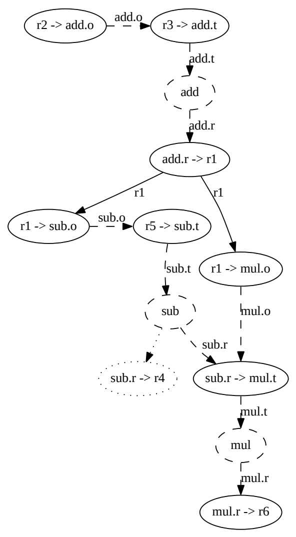

Figure 4. Effect of bypassing on a DDG.

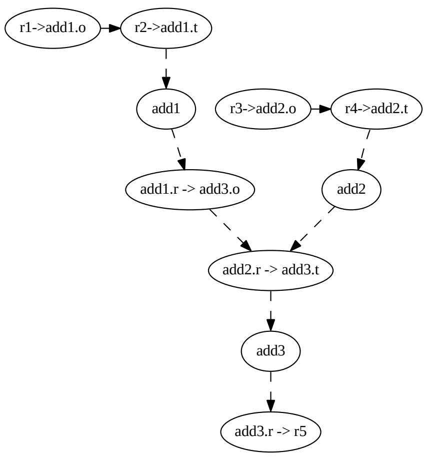

Figure 5. A bypass region.

graph of a DDG, where all the edges are internal edges.

All the data flow to and from a bypass region takes place via registers. This means that if all the bypass regions in a DDG can be scheduled, the entire DDG can be scheduled as well.

Figure 5 depicts a bypass region with three addition operations. The schedulability of the region depends on how the operations are assigned to functional units. Table 1 shows two different assigments of the operations to functional units.

add1 := fu1	add1 := fu1
add2 := fu1	add2 := fu2
add3 := fu2	add3 := fu3

Table 1. Assignments of operations to functional units.

If the functional units are assigned according to the leftmost column, there is no schedule for the bypass region. Since additions *add*1 and *add*2 are bound to the same functional unit, they cannot be scheduled on the same cycle. If they are scheduled on different cycles, then one of the operations will overwrite the result of the other operation. If the additions are bound to different FUs as in the rightmost column, they can be scheduled in any order that respects the partial order imposed by the dependences.

Clearly if all the operations in a bypass region are bound to different FUs, a valid schedule can be found. This is unnecessarily strict, however, and we will derive looser conditions in the following. For simplicity, we only consider function units with no pipelining. We also assume that all operations implemented by a functional unit have the same latency.

In order for bypassing to be possible, the architecture of the processor must have a corresponding connection. Operations in a DDG must therefore be at least partially bound to functional units when bypassing is performed.

If a bypass region contains two operation nodes that are bound to the same functional unit, incorrect scheduling might overwrite the result of the first one before the value is consumed. In a *sufficiently ordered* DDG, dependences ensure that this is not possible.

Definition 2. *A bypass region is* sufficiently ordered *if the following condition holds. Set $\mathcal{N}_O$ denotes the operation nodes of the bypass region, $fu(n)$ denotes the functional unit assigned to operation node n, and $s \geq p$ is a dependence relation that allows p to be scheduled on the same cycle as s but no earlier.*

$$\begin{aligned}
&\forall n_1, n_2 \in \mathcal{N}_O : \\
&\quad fu(n_1) = fu(n_2) \implies \\
&\qquad (\forall s \in succ(n_1), p \in pred(n_2) : s \geq p) \\
&\qquad \vee (\forall s \in succ(n_2), p \in pred(n_1) : s \geq p)
\end{aligned}$$

Lemma 1. *In any legal schedule of a sufficiently ordered bypass region, operation results are not overwritten before all their uses have been satisfied.*

Proof. The proof proceeds by contradition. Let n_1 and n_2 be two operation nodes that are assigned to the same functional unit, and s' be a use of the value of n_1 which receives a wrong value because n_2 overwrites it.

By Definition 2,

$$\begin{aligned}
&(\forall s \in succ(n_1) : \forall p \in pred(n_2) : s \geq p) \\
&\vee (\forall s \in succ(n_2) : \forall p \in pred(n_1) : s \geq p)
\end{aligned} \qquad (1)$$

We examine the disjuncts separately.

1. $\forall s \in succ(n_1) : \forall p \in pred(n_2) : s \geq p$
   Since s' is a use of the value of n_1, $s' \in succ(n_1)$. Hence we get
   $$\forall p \in pred(n_2) : s' \geq p$$
   Since the trigger move $\hat{t}$ of n_2 is a predecessor of n_2, we get $s' \geq \hat{t}$. Hence in any legal schedule $\hat{t}$ is scheduled later than or on the same cycle as s', so the value used by s' cannot be overwritten.
2. $\forall s \in succ(n_2) : \forall p \in pred(n_1) : s \geq p$
   We examine the trigger moves $\hat{t_1}$ and $\hat{t_2}$ of n_1 and n_2 respectively. For any successor $\hat{s}$ of n_2, $n_2 \geq \hat{s}$ and
   $$\forall p \in pred(n_1) : \hat{s} \geq p,$$
   and because $\hat{t_1}$ is a predecessor of n_1, we get
   $$\hat{s} \geq \hat{t_1}.$$

Since operations have a non-zero latency, we have $\hat{t_2} > n_2$, which yields

$$\hat{t_2} > n_2 \geq \hat{s} \geq \hat{t_1}.$$

Operation n_2 is hence triggered before n_1, and because of the assumption that operations have uniform latency, the result of n_2 cannot overwrite the result of n_1.

Thus s' cannot exist, which completes the proof. □

Definition 3. *A* bypass region based scheduler *schedules all the moves of a bypass region before starting to schedule moves in another bypass region. Since the operand, trigger and result moves of an operation are connected by internal edges, region based scheduling subsumes operation based scheduling.*

Theorem 1. *A sufficiently ordered bypass region is schedulable using region based top down list scheduling.*

Proof. Since operation nodes in a bypass region are already bound to functional units, the scheduler only needs to assign move nodes to cycles. Scheduling fails if at some point the scheduler cannot find a cycle in which there is room for the node being considered and in which the source of the move is live.

If the source of the move is a register, there is trivially room in the schedule. Registers are not overwritten by moves in the bypass region, so the move can be placed arbitrarily far ahead in the schedule.

If the source of the move is the result register of a functional unit, its value is live as long as another operation overwrites it. By Lemma 1, no operation in the bypass region can overwrite the result before it has been read, so only moves outside of the bypass region can overwrite the result.

Because of the nature of the bypass region based scheduler, already scheduled code outside the current region does not leave any functional units occupied. If the scheduler places all the moves relating to a functional unit after the cycle where the already scheduled code frees the unit, then already scheduled code cannot overwrite results within the current region. □

VI. RELATED WORK

Software bypassing as an optimization is only possible in an architecture where the internal transport buses are architecturally visible. There has thus been little work directly on software bypassing.

The compiler in the original implementation of transport triggered processors, the *Move* toolset [5, 10], implemented software bypassing, but as a kind of peephole optimization during scheduling. The scheduler performs bypassing if it notices that a value is available at a function unit, and redundant moves to registers are detected and removed. In contrast to our work, bypassing in Move takes place after register allocation, so the freed registers cannot be utilized for anything else. Even so, bypassing still reduces the need for read and write ports in register files [8].

The impact of software bypassing on instruction level parallelism and register file traffic in the context of the TCE toolkit was studied in [7]. The bypasser studied in this work was similar to that of the Move toolset.

The Synchronous Transfer Architecture (STA) [3] is similar to the Transport Triggered Architecture except that all the operands of an operation are read at the same cycle. The interconnection network of an STA processor is similar to that of a TTA processor in that it allows direct transfers from functional unit outputs to inputs. A prototype compiler is described in [3], but it is difficult to determine from the paper how well it handles bypassing in reality. More recently, fuzzy control systems have been applied to scheduling STAs [11].

Another architecture that enables software bypassing is *FlexCore* [12], where the datapath of a general purpose processor is exposed to the programmer. The compiler described in [12] compiles MIPS assembly code into the native instructions of the processor. The native instructions are interleaved as tightly as possible, and bypassing is done if an instruction needs to read a value that has not been written to a register yet.

In EDGE architectures [2], 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.

VII. CONCLUSIONS AND FUTURE WORK

Performing software bypassing on a Data Dependence Graph before registers and functional units have been allocated can reduce register pressure, and thus help generating better code for a Transport Triggered Processor. Too aggressive bypassing can render a Data Dependence Graph unschedulable on the resources of a processor.

In this paper we have presented conditions that guarantee that bypassing does not oversubscribe the resources of the processor. This allows bypassing to be performed before resource allocation.

We plan to implement prepass bypassing in the TCE toolkit [9]. In particular, we are interested in finding out how much bypassing we could do "for free". Data dependences naturally sequentialize some data transports, which therefore satisfy the conditions for safe bypassing.

Acknowledgments

This research was partially supported by Academy of Finland project number 205743.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, August 2006.

[2] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *Computer*, 37(7):44–55, 2004.

[3] Gordon Cichon, P. Robelly, H. Seidel, M. Bronzel, and Gerhard Fettweis. Compiler scheduling for STA-processors. In *PARELEC '04: Proceedings of the international conference on Parallel Computing in Electrical Engineering*, pages 45–60, Washington, DC, USA, 2004. IEEE Computer Society.

[4] H. Corporaal. *Microprocessor Architectures: from VLIW to TTA.* John Wiley & Sons, Chichester, UK, 1997.

[5] Henk Corporaal and Hans (J.M.) Mulder. Move: a framework for high-performance processor design. In *Proc. ACM/IEEE Conf. Supercomputing*, pages 692–701, Albuquerque, NM, 1991.

[6] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA '83: Proc. 10th int. symp. on*

Computer architecture, pages 140–150, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.

[7] Vladimír Guzma, Pekka Jääskeläinen, Pertti Kellomäki, and Jarmo Takala. Impact of software bypassing on instruction level parallelism and register file traffic. In Mladen Bereković, Nikitas Dimopoulos, and Stephan Wong, editors, *SAMOS*, volume 5114 of *Lecture Notes in Computer Science*, pages 23–32. Springer, 2008.

[8] Jan Hoogerbrugge and Henk Corporaal. Register file port requirements of Transport Triggered Architectures. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 191–195, New York, NY, USA, 1994. ACM Press.

[9] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala. Codesign toolset for application-specific instruction-set processors. In *Proc. Multimedia on Mobile Devices 2007*, pages 65070X–1 – 65070X–11, 2007. `http://tce.cs.tut.fi/`.

[10] Johan Janssen and Henk Corporaal. Partitioned register file for TTAs. In *Proc. 28th Annual Workshop on Microprogramming (MICRO-28)*, pages 303–312, 1996.

[11] Xiaoyan Jia, Jie Guo, and Gerhard Fettweis. Integrated code generation by using fuzzy control system. In *SCOPES '08: Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pages 1–10, 2008.

[12] M. Thuresson, M. Sjalander, M. Bjork, L. Svensson, P. Larsson-Edefors, and P. Stenstrom. FlexCore: Utilizing exposed datapath control for efficient computing. In *Proc. Int. Conf. on Embedded Computer Systems: Architectures, Modeling and Simulation. Samos, Greece*, pages 18–25, 2007.