Noname manuscript No.
(will be inserted by the editor)

Code Density and Energy Efficiency of Exposed Datapath Architectures

Pekka Jääskeläinen · Heikki Kultala · Timo Viitanen · Jarmo Takala

Received: date / Accepted: date

Abstract Exposing details of the processor datapath to the programmer is motivated by improvements in the energy efficiency and the simplification of the microarchitecture. However, an instruction format that can control the data path in a more explicit manner requires more expressiveness when compared to an instruction format that implements more of the control logic in the processor hardware and presents conventional general purpose register based instructions to the programmer. That is, programs for exposed datapath processors might require additional instruction memory bits to be fetched, which consumes additional energy.

With the interest in energy and power efficiency rising in the past decade, exposed datapath architectures have received renewed attention. Several variations of the additional details to expose to the programmer have been proposed in the academy, and some exposed datapath features have also appeared in commercial architectures. The different variations of proposed exposed datapath architectures and their effects to the energy efficiency, however, have not so far been analyzed in a systematic manner in public.

This article provides a review of exposed datapath approaches and highlights their differences. In addition, a set of interesting exposed datapath design choices is evaluated in a closer study. Due to the fact that memories constitute a major component of power consumption in contemporary processors, we analyze instruction encodings for different exposed datapath variations and consider the energy required to fetch the additional instruction bits in comparison to the register file access savings achieved with the exposed datapath.

P. Jääskeläinen, H. Kultala, T.Viitanen and J. Takala
Tampere University of Technology, Finland
{pekka.jaaskelainen,heikki.kultala, timo.2.viitanen, jarmo.takala}@tut.fi

1 Introduction

The term "exposed datapath" is seen in the literature being attached to processor architectures where additional processor datapath details are given to the direct control of the programmer or the compiler [36, 18, 12]. Clearly, which details can be considered "additional" in this sense are relative.

In this article, we consider a processor to implement an *exposed datapath architecture* in case it presents a programming model that has one or both of the following features:

– Programmer access to function unit ports or small register files coupled with function unit ports to avoid overheads of a large centralized general purpose register file.
– More fine grained control of the datapath than only defining function unit operations.

Exposed datapath architectures are motivated by their simpler control hardware and the benefits from the additional programmer control [41, 20]. In addition to potentially enabling smaller and higher clock frequency processor designs, their impact to the energy efficiency is nowadays considered an even more interesting aspect [17]. The simplified decode and hardware instruction scheduling logic consumes less power, and the increased compiler control can decrease register pressure and register file utilization significantly.

The reduced register file pressure improves both the dynamic power consumption, due to the less register file accesses needed, and the static power consumption, as it is possible to serve the same amount of parallel operations with simpler register files [20]. There is an implication from reduced general purpose register pressure that is especially interesting in case of contemporary throughput-oriented *Graphics Processing Units (GPUs)*. For GPUs, which are optimized for massively data parallel workloads and the *Single Program Multiple Data (SPMD)* programming model, the reduced need for *General Purpose Registers (GPR)* translates to more "threads" (or instances of the kernel) to be potentially alive at the same time (with the same number of GPRs consumed). This helps increasing the utilization of the function units and to saturate the available memory bandwidth.

This paper reviews the work in exposed datapath architectures and evaluates their consequences to the code density and register file access savings. We identify the most useful features given their code density overheads in order to provide guidelines for processor designers that are considering adding exposed datapath features to their designs.

There are several common characteristics in the architectures reviewed in this article. They are usually very compiler-oriented; only limited dynamic scheduling of the instructions is performed, if any. Their assembly programming is tedious and error-prone due to the additional details exposed to the programmer – their feasible use requires a compiler that supports a higher-level programming language than the assembly. The low level programming model also implies the lack of legacy binary code support between processor families; due to exposing details of the datapath to programs, they need to be recompiled whenever the architecture is updated. In addition, especially the instruction-level parallel exposed datapath machines might have ultra wide instruction words due to the additional programmer control and due to no-operation bits needed because of the visible function unit pipelines and the lack of data hazard detection hardware.

The implications for the code density is given an emphasis in the article. In case the energy needed to read the possible additional instruction bits to control the datapath details from the instruction memory exceeds the energy benefit from the additional programmer control, the feature can be considered counter-productive in low power processor designs.

This article contributes the following: 1) An overview of exposed datapath architectures, both the academic and the commercialized offerings; 2) an analysis of the code density implications from different additional programmer controlled details; and 3) an evaluation of their implications for energy consumption by concentrating on the general purpose register access savings and the instruction fetch costs.

2 Exposed Datapath Architectures

The fundamental ideas in exposing most or all of the control of the processor internal components to the programmer can be traced back to the idea of *microcoding.* Microcoding was proposed in the early 1950s to be used to simplify and modularize the processor control unit implementation by allowing the processor designer to describe the processor control signals to be produced from programmer-visible instructions as small *microprograms* instead of hard-wired logic. In *horizontal microcoding*, the microprogram for each instruction describes precisely what happens at each cycle of the execution of the coded instruction in the processor datapath, including the register enable signals, data routing through multiplexors etc. In *vertical microcoding*, on the other hand, the microprogram contains instructions that need some level of further decoding by the processor hardware and expand to control signals for multiple execution cycles. One of the earliest commercially successful machines that employed microcoding was the System/360 series from IBM in the 1960s. [39, 32]

Microprograms are typically stored in read-only memories or programmable logic circuits (called *control stores*) inside the processor control unit. In some cases, the control store is writable, which enables on-the-field updates of the so called *firmware* of the processor. In the early days of microcoded machines, the writable control stores were used for instruction set emulation and to implement instruction sets customized for different higher-level programming languages within a single processor microarchitecture. [32]

In the light of the microprogramming and writable control stores, the concept of the exposed datapath architectures can be interpreted to reflect the fact that the details that have been typically visible only to the microcode programmer are exposed to the common programmers of the processor. That is, an “extremely exposed architecture” could fetch the datapath control signal values directly from the program memory instead of indirectly from a microcode control store or through a decoding stage. All the exposed datapath architectures in this review follow this principle to some degree. They mainly differ in the level of additional instruction decoding and hardware control logic needed, which places their programmer interface somewhere between horizontal and vertical microcoding.

2.1 From CISC to RISC and VLIW

Reduced Instruction Set Computer (RISC) was proposed by researchers in the University of California in the late 1970s. The motivation was that the earlier, *Complex Instruction Set Computers (CISC)* included overly complex instruction decoding and execution logic given the widespread use of higher-level programming language compilers that abstract the processor instruction set from the programmers. [34] The CISC machines can be implemented using microprograms which are expanded to RISC-like simple operations and control signals at runtime. Thus, in this point of view, the RISC style can be thought of as a step from CISC towards "exposed datapath architectures" by exposing the smaller granularity function unit operations visible in microprograms to the programmer.

FPS-164 was a scientific co-processor introduced in 1981 that included an instruction set similar to the ones in horizontal microprograms. With this architecture it became clear that such detailed instruction sets demand efficient higher-level language compilers to be feasible. Hand-programming of such machines was considered burdensome and error-prone which led to porting a Fortran-77 compiler to this architecture [37].

Research in horizontal microcode compaction (producing parallel microprograms automatically) led *Josh Fisher* to the ideas on the original *Very Long Instruction Word (VLIW)* architecture and the associated compilation techniques for global extraction of instruction level parallelism in the early 1980s. The global instruction scheduling technique called trace scheduling was used to efficiently utilize the statically scheduled VLIW architectures for programs with branches. In essence, the original VLIW was a statically scheduled multi-issue RISC, thus resembled the horizontally microcoded machines that expose fine grained parallel hardware resource schedule timings to the programmer. [14]

2.2 Bypassing Register File in Software

It is usual that the register file (RF) is bypassed (or "forwarded") in the processor pipeline to alleviate stalls resulting from data hazards. In case an instruction is reading a result produced by a previous instruction in the pipeline, without bypassing the instruction would need to wait until the result was written to the RF only after which it could read it to execute its operation. With bypassing, the result from a previous instruction can be forwarded to the input of the consuming function unit before it is written to the RF.

However, the bypass paths add complexity to the routing and the decoding logic of the processor datapath. In addition, if the bypassing is done in the hardware, the instructions still need to refer to general purpose registers (GPRs), thus need registers assigned to them by the compiler. This increases the register pressure and might lead to spilling temporary values to memory. The instructions also sometimes write the values to the register file unnecessarily in case the next instruction that got the data through the forwarding connections is the only consumer.

Exposing the bypass network, or the function unit result registers, to the programmer is a popular design choice to get benefits from a more visible datapath without dramatically changing the programming model. The idea of letting the

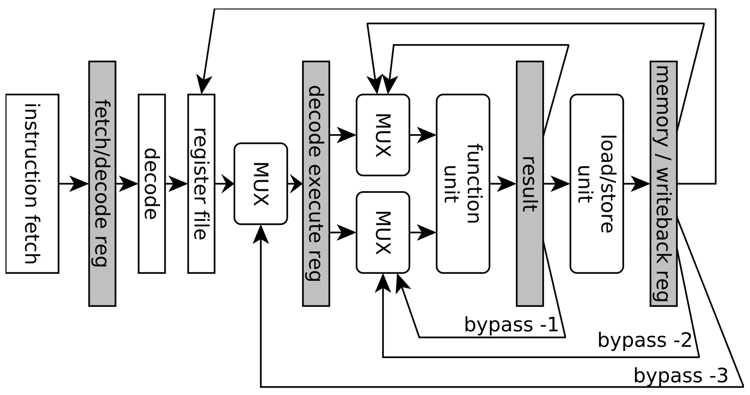

Fig. 1 Example of a classical 5-stage RISC pipeline with the bypass paths visible. Adapted from [41]. The darkened rectangles depict pipeline registers.

programmer to express register file bypassing in software is later in this article referred to as *software bypassing* [9].

Software bypassing reduces the pressure on the number of required GPRs and the number of RF accesses. For example, *virtual registers* are additional values introduced to the operand fields of the instruction that indicate that the operands should be routed from the bypasses instead of reading them from the register file [41]. The destination field is augmented with bits that enable omitting the result write to the register file. This capability is added on top of a traditional 5-stage pipeline architecture as depicted in Fig. 1. In this pipeline, the three bypass paths for both operands can be controlled explicitly using the virtual register operands; the values of the previous three instructions in the pipeline can be referred to directly without accessing the register file. Fig. 2 shows a pseudo assembly program controlling a pipeline with software bypassing support.

```
a)  add r1, r2, r3   b) add :, r2, r3
    mul r2, r1, r4      mul r2, bp1, r4
```

Fig. 2 Pseudo assembly code snippet implementing $r2 = r1 * (r2 + r3)$ with a) a GPR based instruction set (first parameter is the target register) and b) an architecture with the bypass paths visible using a virtual register markup to refer to the values coming straight from the bypass paths. In the latter, **:** marks a value that should not be written to the register file and **bp1** denotes a value in *bypass -1*.

The *Efficient Low-power Microprocessor (ELM)* project of the Stanford University studied techniques to reduce power consumption of embedded processors [12]. They identified that the main source of energy inefficiency in embedded RISC processors is the instruction and data supply, more specifically the caches. In order to reduce this bottleneck, they proposed the use of explicitly controlled instruction register files [4] and operand register files [2]. Both of these techniques increase the programmer-visibility by exposing more of the microarchitecture to the programmer. The operand registers enable explicit operand forwarding between function units, bypassing the larger general purpose register file.

Operand Register Files (ORF) are present also in NVIDIA's research architecture referred to as *Echelon* [24]. The compiler controlled latency optimized cores in their system are three way VLIWs with the additional level of data locality added to improve power efficiency.

Similarly to Stanford ELM and Echelon, MOVE-Pro considers adding a small set of registers local to the function units to improve the temporal freedom of the software bypassing optimization [18] and to reduce the general purpose register file accesses. In their work the small register file is located at the function unit output, in contrast to Stanford ELM and Echelon where the additional register files store function unit input operands. The MOVE-Pro paper does not evaluate whether the additional complexity of multiple programmer accessible output or input registers (instead of just one) is beneficial on average. In [18] the comparisons were made against a simple RISC architecture instead of the previous exposed datapath processors. For example, in [29] and [41] the authors observe that the vast majority of temporary values are short lived, which can be interpreted that they are used once or maybe twice in the program by instructions close to the producer, which logically would speak against the additional complexity of multiple operand or result registers.

Some of the AMD GPUs (at least the Evergreen [1] family), provide access to the previously computed ALU values. The access is done using special registers called *previous vector* and *previous scalar* that can be used as source operands. It allows limited software bypassing of the results produced by the previous instruction group. Similarly, versions of ARM Mali GPUs [27] have been found to utilize a programming model with direct access to function unit ports in their Vertex Shader cores. Some of the function units even have an additional delayed output register, which enables temporary variables to be accessed for longer time.

Out-of-the-box Computing proposes a statically scheduled architecture called *the Mill* that avoids the general purpose register file bottleneck by using a temporary result storage called *the Belt*. The Mill architecture also exposes the function unit pipeline latencies, like VLIW architectures. At the time of this writing, the architecture is not yet fully published, but some of its novel concepts are advertised in talks given by the CTO of the company [16].

Nevertheless, according to the recordings and slide sets of the talks, the Belt is an interesting concept analogous to a conveyor belt. It can be described as a FIFO that stores results from function units. New results pushed from one end push out old results from the other end. What differentiates this idea from stack or accumulator machines is that all the previous results, not only the last one, in the belt can be directly referred to by the next instructions.

The Belt idea is similar to accessing function unit result ports directly (especially the cases where there are multiple registers per output). The similar idea was proposed in the MOVE-Pro with its multiple result registers. However, in the Mill architecture, they take a step further and remove the general purpose register file completely. That is, it is possible to access temporary values only from the belt. The next level in the memory hierarchy is a scratch pad memory where long lived variables can be spilled. An interesting implication from this is that as the belt is always the implicit destination, the instructions do not need to encode destination specifiers at all, which potentially leads to instruction word width savings.

2.3 Data Transport Programming

Data transport programming provides additional programmer freedom on top of the software controlled bypasses: the possibility to control the timing of the function unit operand and result data transfers. This helps reducing the RF pressure further as there is often *slack* in the programs: it is not necessary to move the results to the next consumer or the register file immediately when they are ready. Thus, while the software bypassing capability alone reduces the number of accesses to the RF and the register pressure, the additional freedom of transport programming can simplify the number of RF ports required to serve multiple functions units even further. [9]

The concept of data transport programmed architectures was first proposed for control processors in the mid-1970s. The work published by *Lipovski* can be seen as one of the first research efforts in transport programmed processors [28, 35]. The proposed processor, named *Conditional MOVE (CMOVE)* included only one instruction: a data move between memory mapped control registers. The ALU was attached to the core as an I/O device.

Corporaal et al. have done extensive research on the benefits of data transport programming. In a project called MOVE, they studied a data transport programmed architecture for general purpose applications and high performance computing [6]. In addition to an example transport programmed processor implementation called MOVE32INT [8], the MOVE project also produced a processor design toolset with a retargetable C compiler, instruction set simulator, and automated design space exploration tools [10].

One of the key discoveries made in the MOVE project was the potential of transport programming in reducing the register file complexity bottleneck of wide VLIW architectures [20,23]. Significant reductions to the RF complexity were measured for transport programmed processors in the MOVE project. For example, in [20] the authors conclude that two parallel operations can be supported using only a two-ported register file, and 3.6 parallel operations using a six-ported register file. This is a quite dramatic improvement to the original VLIW which requires at least four read ports and two write ports to support two parallel operations, and at least eight read ports and four write ports to support the average of 3.6 parallel operations.

The MOVE group proposed a new classification for processor architectures according to how the instructions trigger the operation execution. In this classification, their data transport architecture belonged to the class of *Transport Triggered Architectures (TTA)* while the "traditional" architectures were classified as *Operation Triggered Architectures (OTA)* [7]. Instead of starting operations in function units by the instructions, TTAs start the function unit operations as side-effects of writing operand data to a "triggering port" of the function unit. Fig. 3 presents an example TTA processor along with an instruction controlling the data transports for a single cycle.

It should be noted that data transport programming does not imply that the function unit operations are triggered by transports; the operation execution can also be controlled separately from the data movements. This idea was proposed in later (2011) MOVE-Pro [18] work from the group of Corporaal. MOVE-Pro does not have designated trigger ports to start the operation like in the original MOVE, thus increasing instruction scheduling freedom by allowing arbitrary operand move

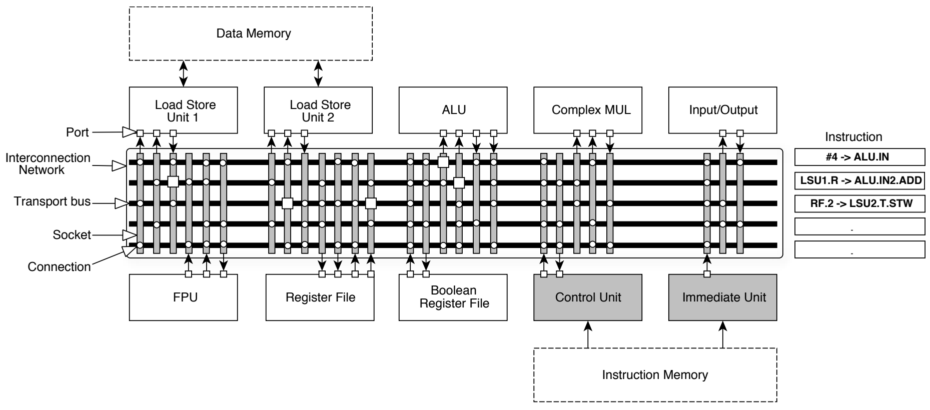

Fig. 3 Example of a TTA processor datapath. The datapath consists of function units, register files, boolean register files, and a customizable interconnection network. Data transports are explicitly programmed; a write to a special *trigger port* of a function unit starts the operation execution. The example instruction defines *moves* for three buses out of five, performing an integer summation of a value loaded from memory and a constant. In parallel, the third move stores a register to memory. The rest of the buses are idle. The connections enabled by the moves are highlighted with squares.

ordering for each operation. The other way to look at it is that, in fact, all of the ports in their architecture are potentially triggering, which means the instructions need to encode the triggering info (add an opcode field) for all types of operand move instructions, or a separate function unit issue field for each function unit that can be triggered in parallel, like in the traditional VLIW/RISC style. The programming simplification of allowing all move instructions to any operand to trigger the operation execution has been proposed previously as *Synchronous Transfer Architecture, (STA)* [15].

On top of the software bypassing, another interesting optimization proposed in the context of the TTA work in MOVE was *operand sharing* [9]. The idea is that as the function unit input ports are visible to the programmer and can have storage of their own, sometimes the previously used operand value can be reused in the successive computation. That is, the operand data that is the same for the next operation needs not to be moved again to the input port of the function unit. This optimization could be adopted also to non-transport programmed architectures in case the function unit input ports are exposed to the programmer. In that case, the optimization saves energy thanks to the reduced register file reads and avoiding routing the operand again to the function unit input as illustrated in the example in Fig. 4.

Research on TTAs in Tampere University of Technology was started in the early 2000s. This work was based on the TTA template produced in the MOVE project. An exposed datapath processor design and programming toolset named *TTA-Based Co-Design Environment (TCE)* [22] was released to public in 2009 with an open source license. It provides a retargetable architecture description language driven toolchain like the previous MOVE toolset. The architecture description language called *Architecture Description File (ADF)* is used to describe exposed datapath processors using a parametrized processor template. ADF en-

```
a)  add r1, r2, r3   b) add r1, r2, r3
    add r5, r2, r4      add r5,  ', r4
```

Fig. 4 Pseudo assembly code snippet implementing $r1 = r2 + r3; r5 = r2 + r4$; with a) a GPR based instruction set (first parameter is the target register) and b) an architecture with the function unit input port storage visible using a virtual register markup to refer to the previous values in the function unit ports. In the latter, $'$ marks a value that should not be read from the register file but from the function unit input port (shadow) register. This is called the operand sharing optimization.

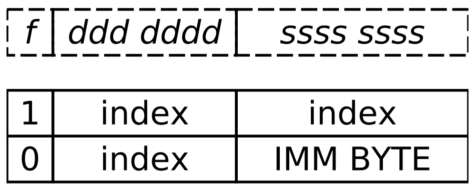

Fig. 5 The 16-bit instruction format of MAXQ consists of two modes: a transport between two locations, and a constant write to an index in the central transfer map [30].

ables describing original MOVE-style TTA processors, but also processors with more limited datapath control [5].

MAXQ [30] from Maxim Integrated (introduced in 2004) is a commercial microcontroller that implements the data transport programming paradigm. The 16-bit processor implements the instruction fetch, decode and execution done in a single pipeline stage. That is, the core is optimized for branch-heavy code, and the transport programming helps in simplifying the processor structure in order to provide the single-cycle instruction pipeline to reduce branch penalities. The architecture is similar to the early CMOVE architecture in its mapping of all components, including the function units, the register file, memories, and I/O devices to the same address space (referred to as *central transfer map*) [28,35]. MAXQ has a compact instruction encoding (see Fig. 5) used to refer to the locations in the address space to execute operations, access I/O devices or registers, either directly or indirectly through a prefix register.

2.4 Microprogramming

Some architectures provide a programmer interface that looks like the microprograms in microcode-based processor implementations. They allow controlling the datapath at the level of multiplexer select signals and expose the finest granularity function unit opcodes to the programmer. While data transport programmed architectures expose the movements of data between registers in the processor, the microprogrammed architectures expose even more details such as the pipeline register enable signals and, in general, have very little decoding, if at all, done to the instruction bits when they are converted to control signals.

In 2003, a processor concept later to be called *FlexCore* [36] was proposed within the *FlexSoC* [21] research project at Chalmers University of Technology. FlexCores are programmed using so called *Native-ISA (N-ISA)* instructions that are seemingly similar to horizontal microcode instructions. For instruction width reduction they proposed the use of a "reconfigurable instruction decoder" which is an instruction decompression unit that expands instructions encoded in so called *Application-Specific ISA (AS-ISA)* format to the more detailed N-ISA format [3]. This is a concept similar to vertical microcoding with a writable control store. The reconfigurability of the instruction decoder/decompressor allows the emulation of multiple traditional ISAs with a single FlexCore datapath, an idea similar to a use case for the original vertical microcoding. It also resembles the dynamic translation used in the Crusoe processors [25] where the translation is done using a more complex software layer. FlexCore is supported by a compiler with an instruction scheduler [33].

Similarly to FlexCore, the *No Instruction Set Computer (NISC)* [31] utilizes the idea of horizontally microcoded programming with no additional decoding logic. Their compiler can also generate a *Finite State Machine (FSM)* based control logic, making their design flow an interesting candidate as an implementation technique for high level synthesis.

Silicon Hive produced a processor template based on what they called the *Ultra Long Instruction Word (ULIW)* architecture, and offered several pre-designed cores as programmable IP blocks [11]. The designs they published exploited hierarchical distributed register files connected to a subset of function units to improve energy efficiency and to reduce the centralized register file bottleneck.

Silicon Hive applied an explicitly controlled datapath approach to support the scalable processor template (e.g., 41 parallel issue slots [26]). It is indicated that the datapath is exposed to the degree that the bypasses and the pipeline control are programmer controlled, but to which extent this programming model has been applied in the sold cores is not clear [38]. The ultra long instruction words would speak for it; their AVISPA design, for example, has 510-bit instructions, which leads a rather big part of the energy to be consumed in the "configuration memory" (instruction memory). The instruction memory energy efficiency can be improved by means of using the so-called "data-flow mode" which aims to compile the inner loops of programs to single instructions that are executed repeatedly on the input data, thus reducing the switching activity for the instruction stream. In this setting the processor looks like a *Coarse Grained Reconfigurable Array (CGRA).*

Static pipelining exposes a classical five-stage pipeline by removing the pipeline registers and letting the compiler to control the same aspects the hardware controls in the traditional pipelined version [13]. The concept resembles a microprogrammed machine with a template-based instruction encoding to limit the instruction word width.

2.5 Classification

In order to summarize the reviewed architecture variants, the architectures can be categorized according to their degree of explicit processor control given to the compiler as illustrated in Figure 6. In this picture, the innermost set includes the architectures that expose basic function unit operations to the programmer. These operations are of the complexity level of micro operations in microcoded CISC architectures. It can be thought that instead of expanding the complex instructions to simpler function unit operations at hardware, the programmer directly controls the basic operations. In case of VLIW, an additional detail visible to the

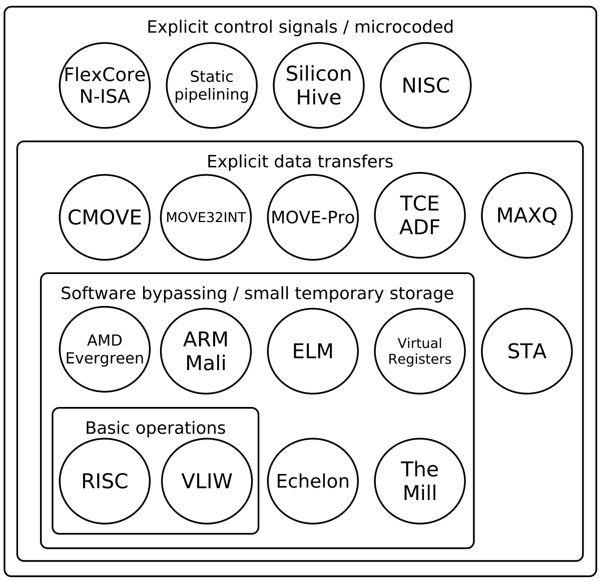

Fig. 6 The mentioned architectures and architecture concepts classified according to the level of processor details exposed to the programmer/compiler. Each outermost set is a superset of the innermost ones in the sense that the architectures in the outer sets provide more programmer control than the ones in the inner ones.

programmer is the function unit pipeline latencies, and they commonly lack data hazard detection in the hardware. While not generally considered *exposed datapath* architectures, they are included here as reference points.

The next degree includes architectures that provide capabilities to bypass a centralized register file with software. In practice, this is done using either explicit access to the bypass connections, function unit ports, result registers, or with smaller distributed storage placed closer to the function units.

Explicit data transfers mean the ability to temporally control the data movements between datapath registers. These registers can include register file ports and function unit operand or result ports.

The architectures that provide the greatest programmer freedom are in the outermost set. Their programming model resembles the microprograms of microcoded processors, presenting very low level details of the datapath implementation to the programmer.

3 Evaluation

In order to quantify the benefits of the different degrees of freedoms in various exposed datapath design alternatives, we identified a set of interesting design points for evaluation. In order to remove the compiler efficiency variance from the evaluation, we manually software pipelined typical loop kernels executed on this type of static architectures. From these benchmarks we evaluated the differences in the datapath resource usage, concentrating on the register file accesses and the number of required general purpose registers, the two main aspects that the exposed datapath features are assumed to improve. We also considered the effects of each variation on instruction encoding, and finally, we summed up the energy consumption of the register file accesses and the instruction fetch to see the overall energy impact from the different architecture design alternatives.

3.1 Benchmark Applications

Software pipelined 2-way unrolled versions of the *SAXPY (Single-Precision AX Plus Y)*, *convolution*, and the *DCT (Discrete Cosine Transform) butterfly* loops were used in the evaluations. These are common routines in linear algebra and signal processing. In all kernels the code being compared consists of only the steady state of a software pipelined loop, i.e., does not include the overhead code, which fills or drains the "software pipeline".

The sequential pseudo assembly programs of the kernels are listed in the following. We unrolled the kernels twice to hide latencies.

SAXPY:

```
loop: X0 = load(XPTR+0)
      X1 = load(XPTR+4)
      Y0 = load(YPTR+0)
      Y1 = load(YPTR+4)
      RES0 = fma(X0, A , Y0)
      RES1 = fma(X1, A , Y1)
      store(RES0, YPTR+0)
      store(RES1, YPTR+4)
      XPTR = XPTR+8
      YPTR = YPTR+8
      I = add(I,2)
      COND = cmple(I, LIMIT)
      cjump COND, loop
```

Convolution accumulates two different sums, which are then merged after the inner loop. The inner loop consists of the following operations:

```
loop: X0 = load(XPTR+0)
      X1 = load(XPTR+4)
      Y0 = load(YPTR+0)
      Y1 = load(YPTR+4)
      SUM0 = fma(X0, Y0, SUM0)
      SUM1 = fma(X1, Y1, SUM1)
      XPTR = XPTR+8
      YPTR = YPTR+8
      I = add(I,2)
      COND = cmple(I, LIMIT)
      cjump COND, loop
```

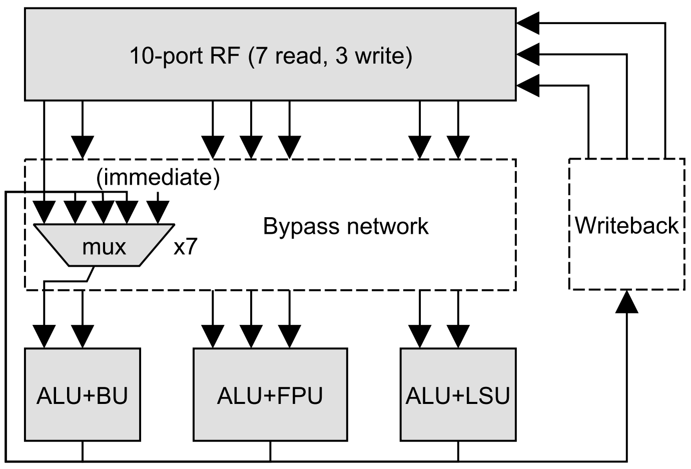

Fig. 7 Example VLIW datapath used as a basis for evaluation. Outputs of each function unit can be forwarded through the bypass network.

DCT butterfly:

```
loop: X0 = load(X0PTR+0)
      X1 = load(X1PTR+0)
      Y0 = fma(TW, X1, X0)
      Y1 = fms(TW, X1, X0)
      store(Y0, X0PTR+0)
      store(Y1, X1PTR+0)
      X0PTR = add(X0PTR, PINCR)
      X1PTR = add(X1PTR, PINCR)
      I = add(I,1)
      COND = cmple(I, limit)
      cjump COND, loop
```

3.2 Datapath Resources

A processor with the same execution resources and connectivity was designed for the considered exposed datapath models and the kernels were manually optimized to each of the variations. The resources of the processors were selected so that each variation could execute the SAXPY loop kernel optimally in 6 cycles and both the Convolution and the DCT Butterfly kernels optimally in 4 cycles, executing a single load or store operation at every instruction cycle.

The execution units are as follows:

1. A load-store unit with base+offset addressing
2. A floating point unit (FPU) and an integer arithmetic-logic unit (ALU) sharing the same execution port (or issue slot)
3. An integer ALU

4. A branch unit

   The latencies of the operations are as follows:

– Loads 3 cycles
– ALU operations 3 cycles
– FMA (Fused Multiply-Add) operations 4 cycles
– Integer operations 1 cycle
– Jumps 3 delay slots

3.3 Variations of Exposed Datapath

The following exposed datapath variations were evaluated:

Basic operations

– VLIW (the baseline)
  This models a RISC or a VLIW machine with fixed cycles for operand and result transports. Register bypasses are assumed to be available to reduce the latency, but the results are always written also to the register file.

Software bypassing

– SB: Software bypassing.
  Like "VLIW" except the programmer can read the result from the function unit directly (SB=Software Bypass). Writing the result to the register file can also be omitted.
– SBR: Software bypass with result register.
  Like "SB" except the result stays live in the result register until the next result computed in the same function unit overwrites it.
– SBR2: Visible function unit result ports with two result registers.
  Like "SBR" except that there is a temporary storage for the previous two results in each function unit. Only one of the result registers of the same function unit can be read at the same cycle.
– SBR2/2: Visible function unit result ports with two result registers and two read ports.
  Like "SBR2" except that both result registers can be read in parallel.
– SBR4: Visible function unit result ports with four result registers.
  Like "SBR2" except that the result storage can hold the four latest results.
– SBR4/2: Visible function unit result ports with four result registers and two parallel result reads.
  Like "SBR4" except that two of the result registers can be read simultaneously.
– ORF2: Like SB but each FU has a local input register file with 2 registers than can be written by any function unit, but read only by the corresponding function unit.
– ORF4: Like ORF2, but each input register file contains 4 registers.

Explicit data transfers

- OF: Temporal operand read freedom.
  Like VLIW/SB, but with the possibility to control the timing of the first operand transport. This has the potential to balance the load placed to the RF ports across more cycles. It allows operand sharing between operations and software bypassing some results that are produced earlier even though there is no result register. Writes to the register file can be omitted.
- TF: Full datapath transport freedom.
  The case where all operand and result transfers can be freely scheduled. The operations are triggered as a side-effect of a trigger port data transfer.
- TFR2: Full datapath transport freedom with two result registers.
  Like "TF" but with storage for two previous results in each function unit. Only one of the results in the same function unit can be read at the same clock cycle.
- TFR2/2: Full datapath transport freedom with two result registers and two parallel result reads.
  Otherwise like "TFR2" except that the two result registers can be read simultaneously
- TFR4: Full datapath transport freedom with four result registers.
  Like "TFR2" but with four result registers in each function unit. Only one of the results of the same function unit can be read at same clock cycle.
- TFTR4/2: Full datapath transport freedom with four result registers and two parallel result reads.
  Otherwise like "TFR4" except that two of the four result registers can be read simultaneously.

Table 1 General purpose register resource usage of SAXPY with the different datapath programming models. Reads and writes are the sum of general purpose register file reads and writes. The live GPRs value indicates the register pressure; it is the maximum number of GPRs in use at the same time. Rd port and wr port counts measure the pressure on the main register file resources; it is the number of read or write accesses the program has to perform at parallel to reach a cycle optimal schedule to the main GPR register file. Local RF access means accesses to a smaller register files closer to the function units. The local RFs can be either operand register files or output register files, depending on the architecture variation.

Architecture	reads	writes	live GPRs	rd ports	wr ports	local RF access
VLIW	21	10	11	4	2	0
SB	15	6	9	3	2	0
SBR	14	6	9	3	2	0
SBR2	12	5	8	3	2	9r
SBR2/2	9	3	7	2	1	12r
SBR4	7	3	6	3	1	14r
SBR4/2	4	0	4	1	0	17r
ORF2	5	3	3	2	1	11r + 4w
ORF4	4	2	2	2	1	12r + 5w
OF	8	5	8	2	2	0
TF	8	5	8	2	1	0
TFR2	7	4	7	2	1	12r
TFR2/2	6	3	6	2	1	11r
TFR4	4	1	4	2	1	13r
TFR4/2	3	0	3	1	0	13r

Table 2 General purpose register resource usage of convolution. See Table 1 caption for explanation.

Architecture	reads	writes	live GPRs	rd ports	wr ports	local RF access
VLIW	16	10	8	5	3	0
SB	8	5	6	3	2	0
SBR	8	5	6	3	2	0
SBR2	8	5	6	3	2	8r
SBR2/2	7	4	5	3	2	9r
SBR4	6	4	5	2	2	10r
SBR4/2	1	0	1	1	0	15r
ORF2	4	2	2	2	1	4r + 3w
ORF4	4	2	2	2	1	4r + 3w
OF	5	4	5	2	2	0
TF	5	4	5	2	2	0
TFR2	5	4	5	2	1	11
TFR2/2	5	4	5	2	1	12
TFR4	3	2	3	2	2	12
TFR4/2	1	0	1	1	0	13

Table 3 General purpose register resource usage of DCT butterfly. See Table 1 caption for explanation.

Architecture	reads	writes	live GPRs	rd ports	wr ports	local RF access
VLIW	20	8	10	7	2	0
SB	14	6	9	5	2	0
SBR	12	4	8	5	2	0
SBR2	12	4	8	5	2	8r
SBR2/2	8	3	7	3	2	12r
SBR4	11	4	8	4	2	9r
SBR4/2	5	0	4	2	0	15r
ORF2	8	2	4	3	1	6r + 4w
ORF4	6	2	2	2	1	8r + 4w
OF	8	5	7	3	2	0
TF	6	3	6	3	1	0r
TFR2	6	3	6	3	1	9r
TFR2/2	6	3	6	3	1	9r
TFR4	4	1	4	2	0	11r
TFR4/2	3	0	3	2	0	12r

3.4 Result Analysis

Software bypassing with dead result elimination seems to have considerable effect on the amount of register file usage. It enables 25-50% of all register writes to be eliminated on these benchmark kernels.

The number of register file reads and total data transfers can be reduced dramatically with transport programmed processors in comparison to just exposing the result ports of an otherwise VLIW-style architecture. The main reason for this was operand sharing; the values of the x and y pointers of SAXPY and convolution routines could be kept in the function unit port for multiple memory operations, and the scalar multiplier A could be kept in the FMA port for the duration of the whole inner loop.

Adding registers to the function unit outputs to allow result values to be bypassed later than the cycle they arrive seems to have more mixed results, and seem

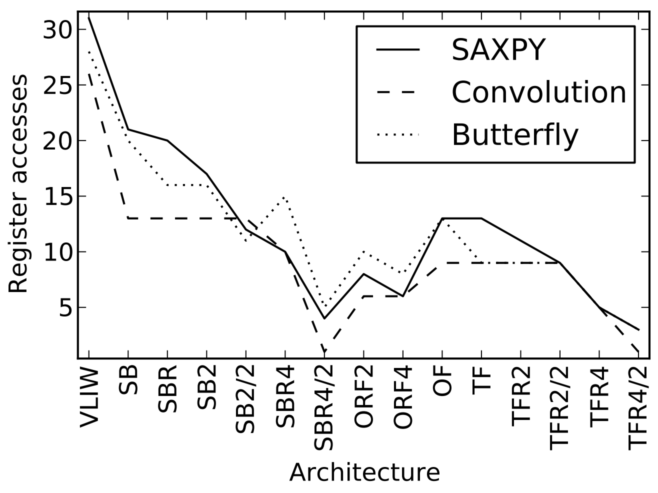

Fig. 8 General purpose register accesses on the benchmark kernels on the different architectures.

to depend more on the workload. For SAXPY, which has less dense code, adding more result registers seems to always help, but for Convolution benefits were seen only after adding a fourth result register.

The amount of result read ports seems to become important in all cases on processors without explicit data transfers. Multiple result registers seem to give a much larger benefit if two result registers can be read at same cycle. A single read port seems to become a bottleneck. Multiple output ports for the result registers may, however, lead to a more complex result register file and bypass network, and the cost of accessing the result registers starts to approach the cost of accessing ordinary registers in the main register file. Obviously, how much adding a smaller result storage gives benefit depends on the size of the larger register files in the hierarchy of which access is avoided using it.

On explicit data transfer processors with the temporal operand freedom, the ability to read multiple result registers of the same function unit at the same cycle is still beneficial, but the effect is considerably smaller than without transport freedom.

Increasing output registers of function units to four registers with two result read ports resulted in very few accesses to the main register file in all cases. However, in these benchmark cases all of the loop-carried variables could be kept in the result registers. With a larger kernel where all the loop-carried variables would not fit the output registers the benefit from four result registers would be smaller.

Even very small input register files that can be read by only single function units can cover most of the register data traffic in these cases and so reduce the amount of traffic to the main register file considerably. Even input register files of

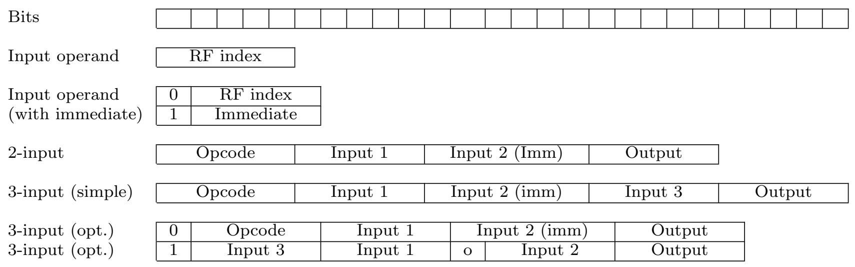

Fig. 9 Instruction encoding for VLIW issue slots in the example processor.

size two can lower the amount of accesses to the main register file considerably, but four input registers seem to give some further reduction in accesses to the main GPR file.

3.5 Code Density

VLIW processors use a horizontal instruction encoding where the instruction is divided into separate fields that control the different function units. Figure 9 shows a possible instruction encoding for the example processor, assuming 32 general purpose registers and 5-bit opcode fields and immediates. The ALU+BU and ALU+LSU units are encoded simply as an opcode field and three operand fields, for a total of 21 bits, each. A similar encoding for the ALU+FPU gives 26 bits. However, by only encoding the third operand for the two applicable operations *fmadd* and *fmsub* this can be optimized down to 23 bits. Removing the immediate, which the floating-point instructions are unlikely to use, saves one more bit.

In the SB variation, the instruction needs additional information to control the bypass network. As discussed in section 2, software bypass sources can be encoded as *virtual register* indices. Case SBR specifies that output registers hold their values on a *no-operation* instruction. Since a *nop* opcode is necessary in any case, SBR requires no additional instruction bits.

Cases SBR2 through SBR4/2 add more result registers to each FU, which can be encoded as additional bypass sources. The largest configurations have 12 sources, leaving values for only 20 general purpose registers. This is justified since these configurations can also run the test programs using fewer general purpose registers. Since most of the instruction word consists of register indices, increasing the number of logical registers has a large impact on the instruction width. It is possible to store the output port indices in separate fields as a middle ground.

Cases ORF2 and ORF4 add all the input registers as possible destinations for result, and registers of a single input register file as possible sources of operations. These can be encoded with virtual register indexes, where the result fields limit the amount of GPRs.

Explicit data transfer processors such as TTAs can be programmed by defining *moves* on a network of *transport buses*. The instruction word consists of a source and destination field for each bus. There is a large design space of possible networks, many of which are inefficient or nonfunctional. Due to this, earlier work often relies on automated design space exploration. In order to obtain a meaning-

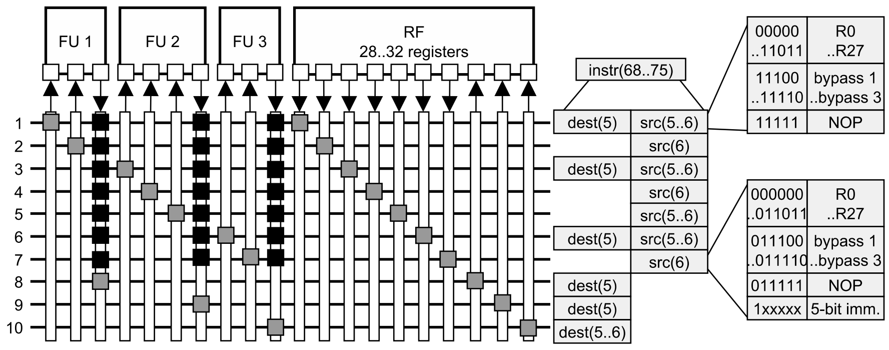

Fig. 10 Explicit data transfer version of the example VLIW datapath in Figure 7 with the same connections. Black boxes indicate *bypass connections*.

ful comparison, we designed a bus network with connectivity as close as possible to the example VLIW datapath, as shown in Figure 10.

The bus-encoded instruction word is almost identical to SB, except where SB has *nop* instructions for each FU, TF/OF must encode a lack of transfer on each bus. The *nop* condition can be encoded in either the source or destination field. In the example processor, it appears optimal to use an additional virtual register. Moreover, the optimization in Figure 9 is inapplicable, since the input operands of the ALU+FPU are spread onto three separately programmable buses. This costs 4 additional instruction bits. Like the corresponding software bypassing architectures, cases TFR2 through TFR4/2 can be encoded with either additional virtual registers or separate output index fields. Table 4 shows the tradeoff between programming model, GPRs and instruction bits.

In the analysis so far, exposed-datapath techniques have little effect on the instruction word and, in particular, bus encoding is very close to simple VLIW encoding. One feature that excaberates the differences in practical systems is *conditional execution*, where instructions can be predicated on guard bits; this is useful for programming conditional behavior without expensive branches. TF processors usually have predicates for each *move slot* instead of each *issue slot*, which would triple their amount in the example processor. Table 4 shows the effect of a 1-bit predicate on the example processor. While this table shows the maximum cost of predicating all move slots, it is possible to predicate only trigger or result moves with some added compiler complexity.

Moreover, we have not considered variable-length instruction encoding. Since practical programs have serial sections where the wide machines are underutilized, an encoding where multiple serial instructions fit in the space of one parallel instruction is desirable. One possible variable-length encoding scheme is to only encode active issue slots, indicated by a short bitmask at the start of the instruction. TF moves are less convenient units of encoding than VLIW slots since they are more numerous and have varying lengths. Alternate schemes have been proposed for TF, such as *template-based encoding*, which only allows some common *nop* patterns [19].

Table 4 Instruction width of the evaluated datapath with different programmer exposed features. Bits indicate instruction word size, GPRs is the number of general purpose registers available to the programmer. In *32 registers* and *64 registers*, the processor has the said number of RF indices, and all bypass sources are represented as virtual registers. *OR field* is like *32 registers*, except output register indices are encoded in a separate field; therefore, fewer bypass sources are needed. *1-bit predicates* is like *32 registers*, except with support for conditional execution, with 1-bit predicates for each issue slot (for VLIW and SB) or move slot (for OF and TF).

Processor	32 registers		OR field		64 registers		1-bit predicates	
	Bits	GPRs	Bits	GPRs	Bits	GPRs	Bits	GPRs
VLIW	64	32			74	64	67	32
SB(R)	64	29			74	61	67	29
SBR2	64	26	67	29	74	58	67	26
SBR2/2	64	26	70	29	74	58	67	26
SBR4	64	20	70	29	74	52	67	20
SBR4/2	64	20	73	29	74	52	67	20
ORF2	64	26			74	58	67	26
ORF4	64	20			74	52	67	20
OF/TF	68	28			78	60	78	28
TFR2	68	25	71	28	78	57	78	25
TFR2/2	68	25	74	28	78	57	78	25
TFR4	68	19	74	28	78	51	78	19
TFR4/2	68	19	77	28	78	51	78	19

3.6 Power Consumption

The results of the previous sections enable us to estimate the power effects of each programming model on the most affected parts of the processor; the register file (RF) and the instruction fetch. We characterize the leakage power and read energy of instruction memory and RFs using CACTI 6.5 [40]. Both instruction memories and RFs are modeled as low-power SRAMs on 32nm synthesis technology at 350K. We assume a 256-word L1 instruction cache. Non-power-of-two instruction memory widths were estimated by linear interpolation between 64-bit and 128-bit SRAM. We assumed the same local RF access energies as in [2], relative to a reference RF read energy estimated with CACTI. The output RF is assumed to replace an output register. Bypass network complexity nor any bypass energies were not modeled; this causes a minor bias in favor of SB2/2, SBR4/2, etc. which have more complex bypass network, and minor bias against OF/TF and ORF architectures which can read more operands locally from operand registers without bypass.

Figure 11 shows estimated power when running an average of the microbenchmark kernels at 500MHz. Exposed datapath techniques appear to give significant reductions in RF power. Instruction fetch takes up significant power, but not enough to make the code density overhead of TF significant. Cases SBR4/2 and TFR4/2 use very little RF power, but may be less attractive in practice due to the enlarged bypass network and additional FU output register hardware, which are not included in the power model. The ORF architectures with operand register files save a lot of power in the main RF accesses, but they may require slightly more complex interconnects for storing results as the results may go to any of the four register files. TFR4 seems to represent an interesting architecture, taking almost as little power as SBR4/2 without enlarging the bypass network but requires only one simultaneous read access to the small result register file.

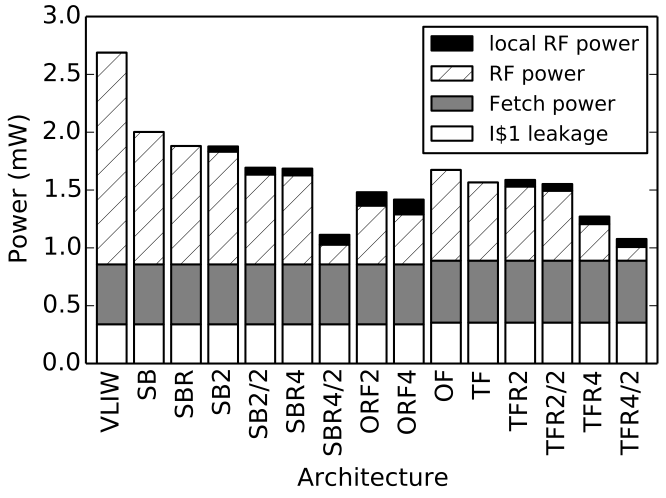

Fig. 11 Estimated RF and fetch power consumption on microbenchmark kernels (averaged) in the different programming models.

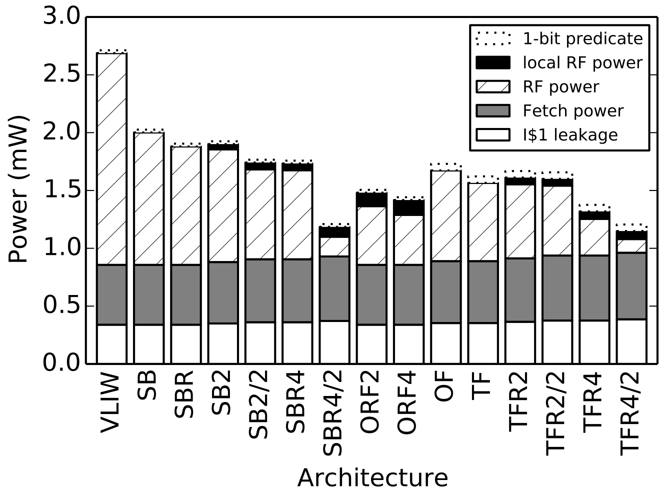

Fig. 12 Estimated RF and fetch power consumption when encoding output register indices in separate fields. Effect from predicated execution with 1-bit predicates is also shown.

There is a high probability of running out of general purpose registers on the larger architectures, especially for wider-issue processors. Therefore, we also obtained power figures with output register indices encoded in separate fields (middle column in Table 4). The encoding has some combinatorial inefficiency, but ensures that at least 27 or 28 GPRs are available. It can be regarded as the worst-case instruction bit cost of each architecture. The results are shown in Figure 12. Code density is now more visible than with virtual registers and has some practical effects, e.g., the RF power reduction from TFR2 and TFR4 is mostly offset by the increasing fetch power.

3.7 Effects from the Reduced Register File Complexity

A major benefit of the transport programming freedom in the explicit data transfer processors is the ability to reduce the number of register file ports, which is the main scaling bottleneck of VLIW architectures. Therefore, it is interesting to evaluate whether a shrunk machine with fewer register file ports and connections can be programmed with a shorter instruction. As shown in Figure 13, reducing the example TF architecture to three RF ports, and shrinking the connection network accordingly, reduces the instruction word from 68 bits to 62. With micro-optimizations to eliminate redundancies by, e.g., resizing the immediates, this can be shrunk further to 55 bits, 19% less than the baseline VLIW instruction.

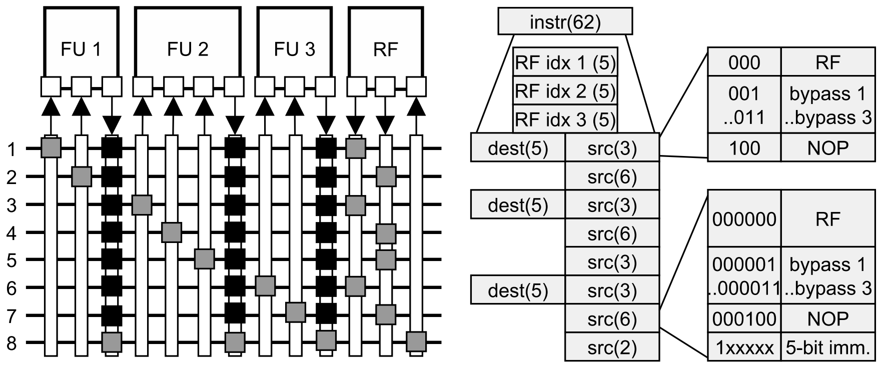

Fig. 13 Transport programmed datapath with reduced RF ports. RF indices are encoded separately from buses.

Reduced number of ports also affect the RF power consumption. According to CACTI, a memory access to the simplified 3-port RF uses 64% energy of the full 10-port RF. Figure 14 shows power figures assuming the minimum complexity RF required to run each benchmark in 6 cycles. TF instruction optimization similar to Figure 14 is applied whenever possible, but only has a small effect.

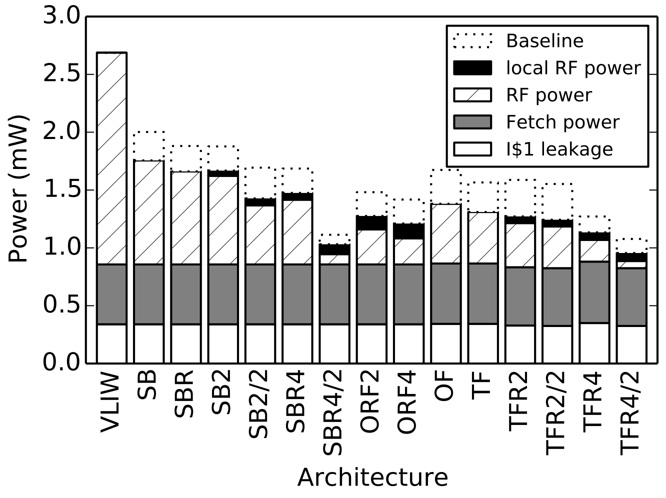

Fig. 14 Estimated register file and fetch power consumption on microbenchmark kernels (averaged) with reduced RF ports and connections. Compared against the machine with the baseline register file.

4 Conclusions

This paper reviewed different approaches to exposed datapath architectures. It provided an overview of the proposed alternatives and an evaluation of various interesting exposed datapath design points. The focus of the article was on the instruction density versus energy efficiency trade-off apparent in these kind of architectures. The goal was to find which of the additional programmer freedoms and control points are the most beneficial when taking in to account their impact on code density.

We confirmed earlier findings that exposed-datapath techniques give significant register file power reductions. Even the simplest benchmarked exposed datapath architecture, a VLIW with software bypassing, saves 25% of RF and fetch power. The processors with 4-word, 2-read port output RFs gave the largest savings of 59% and 60%, respectively. The effects on instruction fetch power were comparatively insignificant with a careful instruction encoding.

Processors with explicit data transfers required 13% less fetch and RF power on average than corresponding software-bypassing VLIWs, or 17% with reduced RF ports. However, due to their bus encoding, explicit data transfer machines have code density difficulties with serial code, which was not included in our model at this time. Since the instruction has many independent fields of varying size, conventional VLIW compression schemes for serial code may be impractical to implement, and for the same reason conditional execution may require more instruction bits. It depends on the case (e.g., how branchy the code is) whether the RF power advantage is large enough to offset these difficulties. For this paper we analyzed only data-oriented kernels typically mapped to this type of static data-oriented architecture.

In contrast to what was suggested in [29] and [41] of majority of variables being short-lived, we found that a particularly interesting design point is a processor with transport freedom with a 4-entry output register file fused to each function unit. With the considered kernels it consumes 20% less benchmarked power than the corresponding software-bypassing VLIW, 19% less than the baseline TF, and 52% less than the VLIW baseline. This style of design, which explores smaller operand or result buffers is discussed in the reviewed literature and deserves more attention. The standard operation programmed VLIW augmented with software bypassing capabilities and a small result register file with two parallel read accesses can get close to the efficiency of a transport programmed one. Further research is warranted to confirm whether these architectures remain efficient with larger test programs.

An interesting finding was that due to the ability to shrink the register file architecture without performance loss, the instruction encodings of the exposed datapath variations (especially transport programmed) also get smaller. This means that sometimes the additional programmer detail can, in fact, lead to improved code density, not always worsening like generally assumed earlier.

This article focused mostly on the energy benefits in exposed datapath variations. However, like it has been pointed out in earlier work [20], the benefits of exposing the datapath go beyond the energy savings. For example, exposing the datapath enables supporting wider VLIW designs with simpler register files, avoiding a common clock frequency scaling bottleneck. Whether this benefit is important enough to justify the additional compiler engineering complexity required depends on the instruction-level parallelism available in the applications of interest. Similarly, how major impact the power consumption of the datapath has to the total power consumption of the design is highly dependent on the targeted applications, especially on their data memory and I/O characteristics.

In the future, we plan to look more into the variations of exposed datapath architectures and their compiler support. One point that was left out from this evaluation was that of context switches: If fast interrupts or preemption are required from an exposed datapath processor design, what are the minimal additional requirements to the hardware and the software? This is not of high concern in architectures with multiple cores where separate cores can be dedicated for interrupts or I/O, but perhaps more so in ultra low power control-oriented applications, or if one wants to support an operating system with pre-emptive scheduling. In this evaluation we assumed that the cores using exposed datapath features are those that might interrupt, but usually are not interrupted, thus not used in *reactive* applications.

Acknowledgements

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 joint undertaking under grant agreement no 641439 (ALMARVI).

References

1. Advanced Micro Devices, Inc.: Evergreen Family Instruction Set Architecture Instructions and Microcode Reference Guide (2011). Rev. 1.1a
2. Balfour, J., Halting, R., Dally, W.: Operand registers and explicit operand forwarding. Computer Architecture Letters **8**(2), 60–63 (2009). DOI 10.1109/L-CA.2009.45
3. Bardizbanyan, A., Själander, M., Larsson-Edefors, P.: Reconfigurable instruction decoding for a wide-control-word processor. In: Proc. IEEE Int. Symp. Parallel Distr. Process., pp. 322–325 (2011). DOI 10.1109/IPDPS.2011.155
4. Black-Schaffer, D., Balfour, J., Dally, W., Parikh, V., Park, J.: Hierarchical instruction register organization. Computer Architecture Letters **7**(2), 41–44 (2008). DOI 10.1109/L-CA.2008.7
5. Cilio, A., Schot, H., Janssen, J.: Architecture Definition File: Processor Architecture Definition File Format for a New TTA Design Framework. Tampere University of Technology, Finland (2006). URL http://tce.cs.tut.fi/specs/ADF.pdf
6. Corporaal, H.: Transport triggered architectures examined for general purpose applications. In: Proc. Sixth Workshop Comput. Syst., pp. 55–71. Delft, the Netherlands (1993)
7. Corporaal, H.: Microprocessor Architectures: From VLIW to TTA. John Wiley & Sons, Chichester, UK (1997)
8. Corporaal, H., Arend, P.: MOVE32INT, a sea of gates realization of a high performance transport triggered architecture. Microprocess. Microprogramming **38**, 53–60 (1993). DOI 10.1016/0165-6074(93)90125-5
9. Corporaal, H., Hoogerbrugge, J.: Code generation for transport triggered architectures. In: Code Generation for Embedded Processors, pp. 240–259. Springer-Verlag, Heidelberg, Germany (1995). DOI 10.1007/978-1-4615-2323-9_14
10. Corporaal, H., Mulder, H.: MOVE: A framework for high-performance processor design. In: Proc. ACM/IEEE Conf. Supercomputing, pp. 692–701 (1991). DOI 10.1145/125826.126159
11. van Dalen, E., Pestana, S., van Wel, A.: An integrated, low-power processor for image signal processing. In: IEEE Int. Symp. Multimedia, pp. 501–508 (2006). DOI 10.1109/ISM.2006.27
12. Dally, W., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R., Parikh, V., J.Park, D.Sheffield: Efficient embedded computing. Computer **41**(7), 27–32 (2008). DOI 10.1109/MC.2008.224
13. Finlayson, I., Davis, B., Gavin, P., Uh, G.R., Whalley, D., Själander, M., Tyson, G.: Improving processor efficiency by statically pipelining instructions. In: Proc. ACM SIGPLAN/SIGBED Conf. Languages Compilers Tools Embedded Syst., pp. 33–44. ACM (2013). DOI 10.1145/2465554.2465559
14. Fisher, J.: Trace scheduling: A technique for global microcode compaction. IEEE Trans. Comput. **C-30**(7), 478–490 (1995). DOI 10.1109/TC.1981.1675827
15. G.Cichon, Robelly, P., Seidel, H., Matúš, E., Bronzel, M., Fettweis, G.: Synchronous transfer architecture (STA). In: Computer Systems: Architectures, Modeling, and Simulation, *Lecture Notes in Computer Science*, vol. 3133, pp. 193–207. Springer, Berlin, Germany (2004). DOI 10.1007/978-3-540-27776-7_36
16. Godard, I.: Drinking from the firehose: The Belt machine model in the Mill^{TM}CPU architectures (2013). URL http://ootbcomp.com/docs/belt/index.html
17. Guzma, V., Jääskeläinen, P., Kellomäki, P., Takala, J.: Impact of software bypassing on instruction level parallelism and register file traffic. In: Embedded Computer Systems: Architectures, Modeling, and Simulation, *Lecture Notes in Computer Science*, vol. 5114, pp. 23–32. Springer, Heidelberg, Germany (2008). DOI 10.1007/978-3-540-70550-5_4
18. He, Y., She, D., Mesman, B., Corporaal, H.: MOVE-Pro: A low power and high code density TTA architecture. In: Proc. Int. Conf. Embedded Comput. Syst.: Arch. Modeling Simulation, pp. 294–301 (2011). DOI 10.1109/SAMOS.2011.6045474
19. Heikkinen, J., Rantanen, T., Cilio, A., Takala, J., Corporaal, H.: Evaluating template-based instruction compression on transport triggered architectures. In: Proc. IEEE Int. Workshop. System-on-Chip for Real-Time Applications, pp. 192–195 (2003). DOI 10.1109/IWSOC.2003.1213033
20. Hoogerbrugge, J., Corporaal, H.: Register file port requirements of Transport Triggered Architectures. In: Proc. Annual Int. Symp. Microarchitecture, pp. 191–195 (1994). DOI 10.1109/MICRO.1994.717458

21. Hughes, K., Jeppson, P., Larsson-Edefors, M., Sheeran, M., Stenström, P., Svensson, L.: ”FlexSoC: Combining flexbility and efficiency in SoC designs”. In: Proc. IEEE NorChip Conf. (2003)
22. Jääskeläinen, P., Guzma, V., Cilio, A., Takala, J.: Codesign toolset for application-specific instruction-set processors. In: Proc. SPIE Multimedia Mobile Devices, pp. 65,070X–1 – 65,070X–11 (2007). DOI 10.1117/12.707233
23. Janssen, J., Corporaal, H.: Partitioned register file for TTAs. In: Proc. Ann. Workshop Microprogramming, pp. 303–312 (1996). DOI 10.1109/MICRO.1995.476840
24. Keckler, S., Dally, W., Khailany, B., Garland, M., Glasco, D.: GPUs and the future of parallel computing. IEEE Micro **31**(5), 7–17 (2011). DOI 10.1109/MM.2011.89
25. Klaiber, A.: The Technology Behind Crusoe Processors: Low-power x86-Compatible Processors Implemented with Code Morphing Software. Tech. rep., Transmeta Corp. (2000)
26. Leijten, J., Burns, G., Huisken, J., Waterlander, E., van Wel, A.: AVISPA: A massively parallel reconfigurable accelerator. In: Proc. Int. Symp.System-on-Chip, pp. 165–168 (2003). DOI 10.1109/ISSOC.2003.1267747
27. Lima: An open source graphics driver for ARM Mali GPUs (2013). URL http://limadriver.org
28. Lipovski, G.: The architecture of a simple, effective control processor. In: Euromicro Symp. Microprocess. Microprogramming, pp. 7–19 (1976)
29. Lozano, L., Gao, G.: Exploiting short-lived variables in superscalar processors. In: Proc. Ann. Int. Symp. Microarch., pp. 292–302 (1995). DOI 10.1109/MICRO.1995.476839
30. Maxim Integrated Products, Inc.: Introduction to the MAXQ Architecture (2004). Application Note 3222
31. Reshadi, M., Gorjiara, B., Gajski, D.: Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths. In: Proc. Int. Conf. Comput. Design, pp. 69–76 (2005). DOI 10.1109/ICCD.2005.112
32. Rosin, R.: Contemporary concepts of microprogramming and emulation. ACM Computing Surveys **1**(4), 197–212 (1969). DOI 10.1145/356556.356559
33. Schilling, T., Själander, M., Larsson-Edefors, P.: Scheduling for an embedded architecture with a flexible datapath. In: IEEE Comput. Soc. Ann. Symp. VLSI, pp. 151–156 (2009). DOI 10.1109/ISVLSI.2009.6
34. Smith, R.: A historical overview of computer architecture. IEEE Annals of the History of Computing **10**(4), 277–303 (1988). DOI 10.1109/MAHC.1988.10039
35. Tabak, D., Lipovski, G.: MOVE architecture in digital controllers. IEEE Journal of Solid-State Circuits **15**(1), 116–126 (1980). DOI 10.1109/JSSC.1980.1051344
36. Thuresson, M., Själander, M., Bjork, M., Svensson, L., Larsson-Edefors, P., Stenström, P.: FlexCore: Utilizing exposed datapath control for efficient computing. In: Proc. Int. Conf. Embedded Comput. Syst.: Arch. Modeling Simulation, pp. 18–25 (2007). DOI 10.1109/ICSAMOS.2007.4285729
37. Touzeau, R.: A Fortran compiler for the FPS-164 scientific computer. In: Proc. SIGPLAN Symp. Compiler Construction, pp. 48–57 (1984). DOI 10.1145/502874.502879
38. Vaughan-Nichols, S.: Vendors go to extreme lengths for new chips. Computer **37**(1), 18–20 (2004). DOI 10.1109/MC.2004.1260714
39. Wilkes, M.: The growth of interest in microprogramming: A literature survey. ACM Computing Surveys **1**(3), 139–145 (1969). DOI 10.1145/356551.356553
40. Wilton, S.J., Jouppi, N.P.: CACTI: An enhanced cache access and cycle time model. IEEE J. Solid-State Circ. **31**(5), 677–688 (1996). DOI 10.1109/4.509850
41. Yan, J., Zhang, W.: Virtual registers: Reducing register pressure without enlarging the register file. In: High Performance Embedded Architectures and Compilers, *Lecture Notes in Computer Science*, vol. 4367, pp. 57–70. Springer (2007). DOI 10.1007/978-3-540-69338-3_5