EURASIP Journal on Embedded Systems
a SpringerOpen Journal

RESEARCH Open Access

Use of compiler optimization of software bypassing as a method to improve energy efficiency of exposed data path architectures

Vladimír Guzma*, Teemu Pitkänen and Jarmo Takala

Abstract

In the design of embedded systems, hardware and software need to be co-explored together to meet targets of performance and energy. With the use of application-specific instruction-set processors, as a stand-alone solution or as a part of a system on chip, the customization of processors for a particular application is a known method to reduce energy requirements and provide performance. In particular, processor designs with exposed data paths trade compile time complexity for simplified control hardware and lower running costs. An exposed data path also allows the removal of unused components of interconnection network, once the application is compiled.

In this paper, we propose the use of a compiler technique for processors with exposed data paths, called *software bypassing*. Software bypassing allows the compiler to schedule data transfers between execution units directly, bypassing the use of a general-purpose register file, increasing scheduling freedom, with reduced dependencies induced by the reuse of registers, decreasing the number of read and write accesses to register files, and allowing the use of register files with less read and write ports while maintaining or improving performance and maintaining reprogrammability. We compare our proposal against an architecture exploration technique, *connectivity reduction*, which finds in compiled application all interconnection network components that are used and removes those which are not, leading to an energy-efficient application-specific instruction-set processor.

We observe that the use of software bypassing leads to improvements in application speed, with architectures having the smallest number of register file ports consistently outperforming architectures with larger number of ports, and reduction in energy consumption. In contrast, connectivity reduction maintains the same application speed, reduces energy consumption, and allows for increase in processor frequency; however, with the clock frequency increased to match the performance of software bypassing, energy consumption grows. We also observe that in case reprogrammability is not an issue, the most energy-efficient solution is a combination of software bypassing and connectivity reduction.

1 Introduction

In an embedded domain, unlike in more traditional high-performance computing, performance closely relates to energy. Efficient solutions utilize the knowledge of application or application domain to explore hardware and software techniques, eventually leading to application-specific instruction-set processors, and to provide enough performance for a particular task, or set of tasks, while minimizing energy requirements. The exploration of available instruction-level parallelism (ILP) and the use of instruction-set extensions are common ways to improve clock cycle performance, leading to lower operation frequencies and lower energy requirements.

When exploring ILP in computation, large number of general-purpose registers contributes to the increase in performance. Having program variables in independent registers allows data-independent computation paths to be scheduled in parallel, on different execution units. However, in terms of algorithm computation, the time and energy spent on transferring values between function units and register files are wasted, not contributing to actual computation directly, since only the energy spent

*Correspondence: vladimir.guzma@tut.fi
Department of Computer Systems, Tampere University of Technology, Tampere 33720, Finland

while computing in function units is actually useful to compute results.

In addition, when parallel execution requires access to several general-purpose registers in the same cycle, register files need to provide enough read and write ports to allow such access, leading to increased complexity and higher energy requirements of register files.

Another method on how to increase the performance of embedded processors is the customization of instruction set [1,2]. Complex computation patterns can be implemented as custom function units, providing better performance and freeing other processor resources for independent computation paths. This customization, however, often leads to implementation with higher number of input values and produces several results, further elevating the problem of the number of register file ports and necessary data transports between function units and register files.

Yet another method to reduce energy requirements of a custom processor is the optimization of the data path. To maintain the best programmability and allow maximum ILP exploitation, processor interconnection networks tend to be designed for the worst case scenario. However, once the application schedule is set, we can simply remove all the unused components of the interconnection network, eventually maintaining only the required connections (*connectivity reduction*). This has an effect on reduction in interconnection network complexity, instruction fetch and decode energy requirements, and can allow for increase in processor clock frequency. An unfortunate effect of connectivity reduction, however, is limitation to reprogrammability, or no reprogrammability at all, of such a processor. In case the application is modified and needs to be recompiled for the reduced architecture, the compiler may produce inefficient schedule working around missing connections or fail to schedule completely.

In this paper, we propose the use of a compiler optimization technique called *software bypassing*, suitable for architectures with exposed data paths, as a tool for improving energy efficiency. By allowing the compiler to schedule data transfers directly between function unit outputs and inputs, reading the value of previous computation from the register file becomes unnecessary. This helps reduce unnecessary energy costs of register files. In addition, in the case where all of the uses of produced values can be bypassed directly, the actual write of value to general-purpose register can be discarded during compilation (*dead result move elimination*), reducing the total number of data transfers on the interconnection network and further contributing to reduction in processor energy consumption.

Additional benefit of this optimization is the reduction of false data dependencies created by register allocation when several program variables reuse the same register to store the data, effectively allowing the scheduler more scheduling freedom and increases available ILP to explore during instruction scheduling.

We reason that the combined benefits of software bypassing (reduction in register file reads and writes, reduction in the number of data transfers on the interconnection network, and improved cycle count) match those of connectivity reduction when it comes to energy efficiency while maintaining full reprogrammability.

In situations where reprogrammability is not an issue, we propose the use of a combination of software bypassing and connectivity reduction. We reason that the benefits of these two complement each other, providing for large energy savings. In particular, in order to match the clock cycle improvements gained by the use of software bypassing, a processor with only reduced connectivity needs to run with higher clock frequency, eventually leading to an increase in energy requirements, possibly exceeding energy budget.

In our previous work [3], we considered a conservative approach to software bypassing and investigated its effect on clock cycle performance and reduction in register files reads and writes depending on the heuristic parameter guiding bypassing decisions. We also discussed the impact of heuristic when to bypass on register file and interconnection network energy consumption, however, using only a hardware estimator [4] and single architecture.

In this paper, we propose a novel bypassing algorithm and use an extensive set of register file architecture types to investigate the effect of bypassing and connectivity reduction on the energy of various processor core components directly influenced by one or both of the methods. In addition, we investigate the cost of matching the performance improvements brought by bypassing via synthesis for higher clock frequency when only connectivity reduction is available. We evaluate our approach using commercially available tools for processor synthesis, gate level simulation, and power analysis.

The rest of this paper is organized as follows: Section 2 discusses previous work. Section 3 gives a short introduction to architectures with exposed data paths and describes our choice, transport-triggered architectures. Section 4 gives an overview of our novel software bypassing algorithm with integrated dead result move elimination, as well as connectivity reduction. Section 5 outlines our experimental setup, and Section 6 provides a discussion of the results of our experiments. Finally, Section 7 concludes this paper.

2 Related work

Effects of bypassing register files are known and appreciated in processor design [5,6], with reported register file power reduction of 12% on average for Intel XScale

processor and performance loss of 2% in [5] and up to 80% register file energy reduction compared to Reduced Instruction Set Computer (RISC)/Very Long Instruction Word (VLIW) counterparts in [6]. More and more effort is spent in focusing on computation and distancing from the temporary data storage.

The traditional use of register files for storing data becomes a problem with monolithic register files (RF) in VLIW processors with a large number of function units. The requirement of a large number of RF ports in such case makes the use of monolithic RF prohibitively expensive. Common solutions involve the clustering of RF into a number of smaller ones. Intercluster communication can then be implemented using RF to RF copying and/or read/write between dedicated function units and RF across clusters. However, as shown in [7,8], using only register to register copies between clusters reduces achievable ILP when compared to monolithic RF. Results closer to a monolithic RF file can be achieved with the use of direct reads and writes from a dedicated function unit to RF in different clusters, suggesting that the RF to RF copies between clusters should be avoided when possible.

Another step towards better performance and more achievable ILP is bypassing data directly from function unit to function unit, avoiding the use of RF altogether. Such a solution improves performance and reduces the energy required by RF but can be also used to reduce the number of required RF ports while retaining performance.

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

While hardware implementations of RF bypassing may be transparent to a programmer, they also require additional logic and wiring in the processor and can only analyze a limited instruction window for the required data flow information. Hardware implementations of bypassing cannot get the benefit of reduced register pressure since the registers are already allocated to the variables when the program is executing. However, the benefits from the reduced number of RF accesses are achieved. Register renaming [9] also increases available ILP by the removal of false dependencies. Dynamic strands presented in [10] are an example of an alternative hardware implementation of RF bypassing. Strands are dynamically detected atomic units of execution where registers can be replaced by direct data transports between operations. In Explicit Data Graph Execution (EDGE) architectures [11], operations are statically assigned to execution units, but they are scheduled dynamically in a data-flow fashion. Instructions are organized in blocks, and each block specifies its register and memory inputs and outputs. Execution units are arranged in a matrix, and each unit in the matrix is assigned a sequence of operations from the block to be executed. Each operation is annotated with the address of the execution unit to which the result should be sent. Intermediate results are thus transported directly to their destinations.

Static strands in [12] follow an earlier work [10] to decrease hardware costs. Strands are found statically during compilation and annotated to pass the information to the hardware. As a result, the number of required registers is reduced already in the compile time. This method was, however, applied only to transient operands with a single definition and single use, effectively up to 72% of dynamic integer operands, bypassing about half of them [12]. The authors reported 16% to 24% savings in issue energy, 17% to 20% savings in bypass energy, 13% to 14% savings in register file energy, and 15% improvement in instruction per cycle, using a cycle-accurate simulator for two hardware models: Renesas (formerly Hitachi) SuperH SH4a and IBM PowerPC 750FX embedded microprocessor.

Dataflow mini-graphs [13] are treated as atomic units by a processor. They have the interface of a single instruction, with intermediate variables alive only in the bypass network. In [14], architecturally visible 'virtual registers' are used to reduce register pressure through bypassing. In this method, a virtual register is only a tag marking data dependence between operations without having physical storage location in the RF.

Software implementations of bypassing analyze codes during compile time and pass to the processor the exact information about the sources and the destinations of bypassed data transports, thus avoiding any additional bypassing and analyzing logic in the hardware. This requires an architecture with an exposed data path that allows such direct programming, like the *transport-triggered architectures* (TTA) [15,16], *synchronous transfer architecture* [17], *FlexCore* [18], *no-instruction-set-computer* [19], or *static pipelining* [20]. A commercial application of the TTA paradigm is the Maxim MAXQ general-purpose microcontroller family [21].

The assignment of destination addresses in an EDGE architecture corresponds to software bypassing in a transport-triggered setting. Software-only bypassing was previously implemented for a TTA architecture using the experimental MOVE framework [22,23] and MOVE-Pro [6]. TTAs are a special type of VLIW architectures. They allow programs to define explicitly the operations executed in each function unit (FU) as well as to define how (with position in instruction defining bus) and when

data are transferred (*moved*) to each particular port of each unit. With the option of having registers in input and output ports of FUs, TTA allows the scheduler to move operands to FUs in different cycles and read results several cycles after they are computed. Therefore, the limiting factor for bypassing is the availability of connections between the source FU and destination FUs. In our previous work [3], we introduced a simple, conservative software bypassing implementation. We, however, only focused on the improvements in cycle counts and register file read and write accesses when changing bypass aggressiveness heuristics. In [24], we discussed the mentioned simple bypassing algorithm in terms of energy only using a hardware cost estimation model [4] and single architecture.

3 Exposing data paths: transport triggering approach

TTA [15] is an exposed data path architecture which allows the number of architectural resources to be selected, e.g., selection of the number and size of register files, number of read and write ports for each register file individually. Similarly, the number of function units as well as the operation set of each function unit can be defined by an architecture designer. To connect them together, the interconnection network is designed, with choice of the number of buses and sockets to be used. Each socket provides connection between the function unit or register file port and one or more buses. This allows for fully connected architectures, with most compiler freedom to choose how to transport data between the source and destination, as well as heavily reduced connectivity, with buses connecting only a small number of components. It is necessary to point out that a complex fully connected interconnection network is expensive and limits the maximum clock frequency. Alternatively, an optimized connectivity could allow for higher frequency; however, it reduces scheduling freedom. With less alternatives for transport, potentially parallel data transports often need to be serialized.

Figure 1 shows an example of TTA. There are two function units in Figure 1, denoted as *FU:* one serving as an arithmetic and logic unit (ALU) and the other as LSU. *GCU:* denotes global control unit, which is responsible for branching operations, and *RF:* denotes a single register file with two read and two write ports. Figure 1 also shows five transport buses and 13 sockets connected to units and buses. It can be seen in Figure 1 that not all the sockets are connected to all the buses (as the black *dot* denotes connection).

Another interesting aspect of TTA comes from VLIW inheritance. An instruction defines what data transports are to be performed on each bus, which leads to wide instruction words. As a matter of fact, for each bus in the system, the instruction word encodes the source field of transport as well as destination field of transport. While increasing number of buses leads to more freedom for the compiler to schedule, it also increases the instruction width. Reducing the connectivity between sockets and buses typically leads to a lower number of bits required to encode data transport for individual bus and narrows the instruction width. However, in order to significantly reduce the negative impact of the instruction width, instruction compression can be applied. Using dictionary compression [25,26], for example, the code density can be improved significantly with a decrease in spent energy as well.

4 Software bypassing and connectivity reduction algorithms

In this section, we first discuss our two implementations of software bypassing: First is a conservative approach, previously published in [3], while the second one is an opportunistic approach, which has not been published previously.

4.1 Software bypassing and dead result move elimination

Software bypassing and dead result move elimination on TTA processors can be illustrated with an example. Let us consider the following code excerpt on RISC-type architecture:

```
add R3,r1,r2
add R5,r4,R3
mul r1,R3,R5
```

The same code in a typical TTA syntax on a machine with two buses could be the following:

```
r2 -> add.t; r1 -> add.o;
add.r -> R3; r4 -> add.o;
R3 -> mul.o; R3 -> add.t;
add.r -> R5; ...;
R5 -> mul.t; ...;
mul.r -> r1; ...;
```

We can see that individual data transport to operand (denoted with *.o* suffix) registers and trigger registers (*.t*) from the general-purpose registers (r*X* or R*X*) are explicitly defined. In the same manner, the data transport from the result register (*.r*) to general-purpose registers are explicit. It is worth noting that the timing of data transport is not fixed. For instance, register r4 is written to the operand register of *add* one cycle before register R3 is written to the trigger register of *add*, starting actual computation. Similarly, the operand write of *mul* is two cycles before the trigger write of the same operation starts execution.

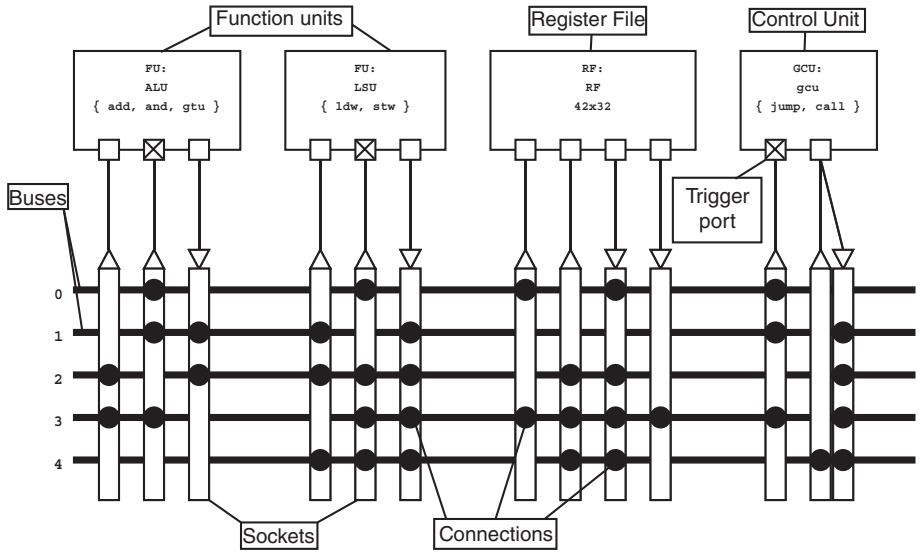

Figure 1 An example of TTA with five buses and reduced connectivity. A TTA architecture with reduced connections between sockets and buses. A fully connected architecture would have connection to all buses in each socket.

The same code with software bypassing and dead result move elimination applied could be the following:

```
r2 -> add.t; r1 -> add.o;
add.r -> mul.o; r4 -> add.o;
add.r -> add.t; ...;
add.r -> mul.t; ...;
mul.r -> r1; ...;
```

We can observe direct data transports from the result to operand or trigger registers. Data transports from the result register to the general-purpose register are completely scraped for both additions (dead result move elimination), as the results are used just once in multiplication, where they are transported directly from the result register of the adder. Compared to the first RISC-like example, we can observe that all the registers denoted with capital RX have disappeared, while the amount of work function units performed remains the same.

As the number of registers available in the architecture is limited, the compiler reuses the same registers to store different variables through the execution of the program. This leads to false dependencies when instruction scheduling as reordering of the data transport can be limited by not real data dependence, such as producer consumer, but false dependence such as write-after-read or write-after-write. The removal of the uses of registers reduces this problem induced by register allocation.

Our first bypassing algorithm is based on a data dependence graph as a part of list scheduling, previously published in [3]. In order to prevent possible deadlocks, the algorithm uses conservative implementation, which first schedules data transports of all operands before attempting to bypass operands directly from the result registers of other function units which produced required values. Redundant result writes to the register file can be removed only once all the uses of the value written to the register file get bypassed and the value is not used outside the current basic block. As a heuristic when not to bypass, simple distance in cycles between the original write to the register, where the producer produces a value, and the cycle where the value is scheduled to be read from the register to the operand of the consumer is used (*lookBackDistance*). It is notable that this implementation works with top-down scheduling algorithms [27].

Our second bypassing algorithm is also based on a data dependence graph. However, while in the first case we used a top-down scheduling algorithm, in this case, we reversed the direction and implemented bypassing during bottom-up scheduling. Due to the nature of bottom-up scheduling, we start the scheduling of operation by scheduling all result moves of operation. This has an advantage of immediate availability of information whether or not all of the uses of the result value become bypassed and an unnecessary write to the register file can be removed immediately.

In addition, our implementation starts with scheduling of result moves with an attempt to find all bypass destinations and create direct bypass moves - *early bypassing*. Only in case some of the destinations cannot be bypassed, or the result value needs to be used outside the scope of the current basic block, the result move to register in the register file is scheduled. Afterwards, the bypassing is attempted once again for the destinations that did not get

bypassed during early bypassing. While this *late bypassing* does not contribute to improvement in cycle counts of the current operation anymore, it still removes unnecessary read from the register file and frees register file read port.

Only once all of the result moves of the operation are scheduled, with or without bypasses, the algorithm attempts to schedule input operand moves as well. In case the schedule of operand moves fails, the result moves are unscheduled and a reschedule is attempted, with only early bypassing enabled. If the scheduling of operands still fails, result moves are unscheduled again and a reschedule is attempted, with only late bypassing enabled. Once again, if the schedule of operands still fails, scheduled moves are unscheduled and a reschedule is attempted, without any bypassing enabled. Only if all previous attempts fail, the starting cycle of the scheduling is decreased and a reschedule is attempted.

The outline of our scheduling algorithm is presented in Algorithm 1, with *inputs* denoting the set of the input operands of the operation being scheduled and with *outputs* denoting the possibly empty set of the results the operation produces. The *ScheduleOperandWrites* method simply tries to schedule input operands of the operation as late as possible once the results of the operation are successfully scheduled, taking into account the latency and pipeline characteristics of the operation on the selected function unit. The method *UnscheduleResultReads* simply unschedules all the previously scheduled results of the operation and undo possible bypasses.

Actual bypassing of result moves is presented in Algorithm 2, with *cycle* denoting the starting cycle from which the scheduling starts, *outputs* denoting the possibly empty set of results the operation being scheduled produces, and the two flags *bypassEarly* and *bypassLate* indicating if bypassing should be attempted early or late or both. The method *ScheduleCandidateALAP* tries to schedule the original result move to register as late as possible, starting from *cycle*, in case not all of the result reads were successfully bypassed or if the result is used in a different basic block.

Other bypassing strategies are possible, including preregister allocation bypassing [28], recursively bypassing chain of operations on critical path, bypassing after the block is fully scheduled without changing schedule to reduce only register file accesses, etc.

4.2 Simple connectivity reduction

With the freedom of design choices offered by transport-triggered architectures, the process of manually optimizing the connectivity of actual processor can become rather difficult. We start by manually selecting a register file configuration, as will be described in Section 5, and fully connected interconnection network.

We used simple connectivity reduction. The idea behind the algorithm is to schedule an application for fully connected TTA and then simply remove the connections of function units and register files to the buses that were never used in the existing schedule. In addition, whole function units and their respective sockets could be removed, if unused by the application. The reduction in the number of socket-to-bus connection should lead to less bits required to encode source and destination fields of data transports for buses and possibly allow for higher clock frequency to be achieved.

5 Experimental setup

We selected open-source *TTA-based Co-design Environment* [29,30] as the platform for our experiments.

To evaluate the effect of software bypassing and connectivity reduction, we selected an integer subset of CHStone v1.7 [31] benchmark, as described in Table 1. The selected designs were identical except for their register file configurations. The number of interconnection buses and function units remain the same for all the tested architectures as well as the total number of registers. We selected the number of interconnection buses in such a way that they do not limit the maximum achievable instruction-level parallelism available in the selected benchmarks. Selected register file configurations could be split into three categories:

1. Architectures with a single multi-ported (SM) register file
    - SM $1 \times 4 \times 4$ - 1 RF - 4 read 4 write ports (42 registers; Figure 2a)
    - SM $1 \times 3 \times 3$ - 1 RF - 3 read 3 write ports (42 registers; Figure 2b)
    - SM $1 \times 2 \times 2$ - 1 RF - 2 read 2 write ports (42 registers; Figure 2c)
    - SM $1 \times 4 \times 2$ - 1 RF - 4 read 2 write ports (42 registers; Figure 2d)

Table 1 Integer subset of CHStone benchmark used in our experiments

Benchmark	Origin
adpcm	CHStone/SNU
aes	CHStone/AILab
blowfish	CHStone/MiBench
gsm	CHStone/MediaBench
jpeg	CHStone/The Portable Video Research Group
mips	CHStone
motion	CHStone/MediaBench
sha	CHStone/MiBench

Algorithm 1 Scheduling moves of single operation in bottom-up fashion

```
1: ScheduleOperation(inputs, outputs)
2: resultsFailed := true
3: operandsFailed := true
4: bypassEarly := true
5: bypassLate := true
6: cycle := DDGLatestCycle(outputs)
7: while cycle > 0 or resultsFailed or operandsFailed do
8:    resultsFailed := ScheduleResultReads(cycle, outputs, bypassEarly, bypassLate)
9:    if not resultsFailed then
10:      {Result moves scheduled}
11:   else if bypassEarly and bypassLate then
12:      bypassLate := false
13:      continue {Retry with the same start cycle without late bypassing}
14:   else if bypassEarly and not bypassLate then
15:      bypassEarly := false
16:      bypassLate := true
17:      continue {Retry with the same start cycle without early bypassing}
18:   else if not bypassEarly and bypassLate then
19:      bypassLate := false
20:      continue {Retry with the same start cycle without any bypassing}
21:   else
22:      bypassEarly := true
23:      bypassLate := true
24:      cycle := cycle − 1
25:      continue {Retry with earlier start cycle and both bypassing methods}
26:   end if
27:   operandsFailed := ScheduleOperandWrites(cycle, inputs)
28:   if not operandsFailed then
29:      {Both, results and operands of operation, successfully scheduled}
30:      return true
31:   else
32:      {Operand moves cannot be scheduled}
33:      {with current position of result moves.}
34:      UnscheduleResultReads(outputs)
35:      if bypassEarly and bypassLate then
36:         bypassLate := false
37:      else if bypassEarly and not bypassLate then
38:         bypassEarly := false
39:         bypassLate := true
40:      else if not bypassEarly and bypassLate then
41:         bypassLate := false
42:      else
43:         bypassEarly := true
44:         bypassLate := true
45:         cycle := cycle − 1
46:      end if
47:   end if
48: end while
```

Algorithm 2 Schedule and bypass result reads

```
1: ScheduleResultReads(cycle, outputs, bypassEarly, bypassLate)
2: for all result in outputs do
3:    bypassSuccess := false
4:    if bypassEarly then
5:       Try to bypass all uses of the result move
6:       bypassSuccess := BypassMove(result, cycle)
7:       if bypassSuccess and result not used outside the current block then
8:          All moves that use result were bypassed
9:          continue
10:      end if
11:   end if
12:   {Not all uses of result were bypassed}
13:   {or the result is used in different block.}
14:   ScheduleCandidateALAP(result, cycle)
15:   if not result is scheduled then
16:      UndoBypass(result)
17:      return false
18:   end if
19:   if bypassLate then
20:      {Still try to bypass result uses to reduce antidependencies}
21:      BypassMove(result, cycle)
22:   end if
23: end for
24: return true
```

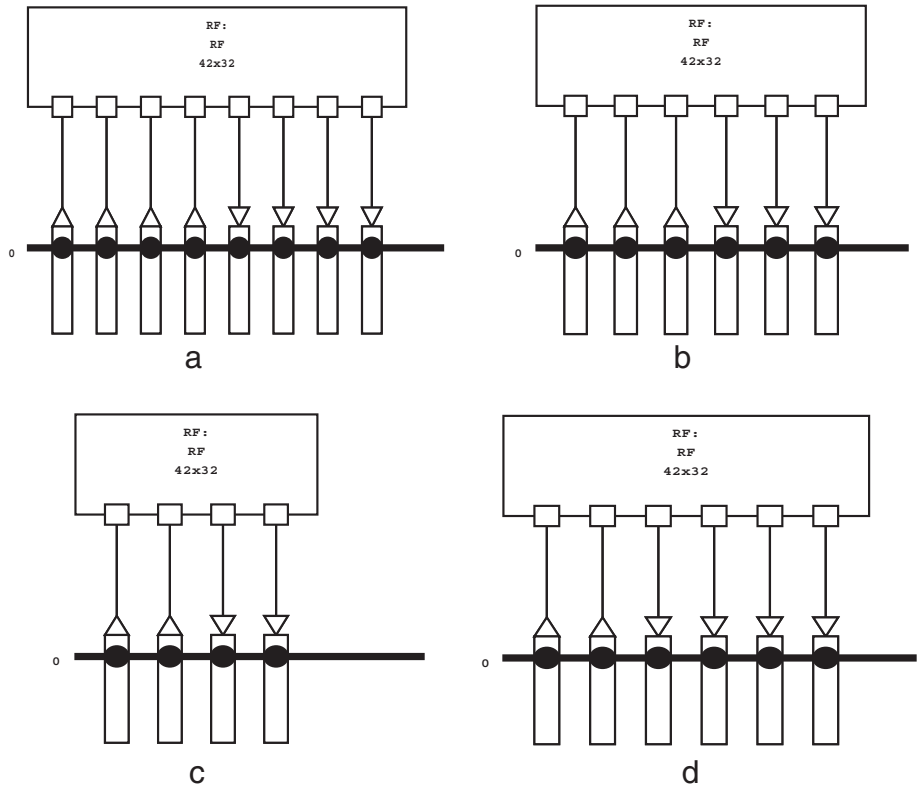

Figure 2 Register file with multiple read and write ports. An example of a single register file with multiple write ports, connected to the interconnection bus. **(a)** SM 1 × 4 × 4. **(b)** SM 1 × 3 × 3. **(c)** SM 1 × 2 × 2. **(d)** SM 1 × 4 × 2.

2. Architectures with a single register file with a single read and single write port (SS) or multiple register files with a single read and single write port (MS)
    - SS 1 × 1 × 1 - 1 RF - 1 read 1 write port (42 registers; Figure 3a)
    - MS 2 × 1 × 1 - 2 RFs - 1 read 1 write port in each (2 × 21 registers; Figure 3b)
    - MS 3 × 1 × 1 - 3 RFs - 1 read 1 write port in each (3 × 14 registers; Figure 3c)
3. Architectures with multiple register files with multiple read and write ports (MM)
    - MM 2 × 2 × 1 - 2 RFs - 2 read 1 write port in each (2 × 21 registers; Figure 4a)
    - MM 2 × 2 × 2 - 2 RFs - 2 read 2 write port in each (2 × 21 registers; Figure 4b)

Examples of register file implementations are shown in Figure 5a,b. The structure of the register file can be divided into three parts:

- input control
- register bank
- output control

For each data input port, the register file contains an input opcode, which specifies the register ought to be written in the register bank, and an input trigger, which describes when the data are ought to be written to the register described in the corresponding opcode.

For each data output port, the register file contains an output opcode, which describes which register is fed to the output port.

When using the register file with single input and output ports, the complexity of the write and read control

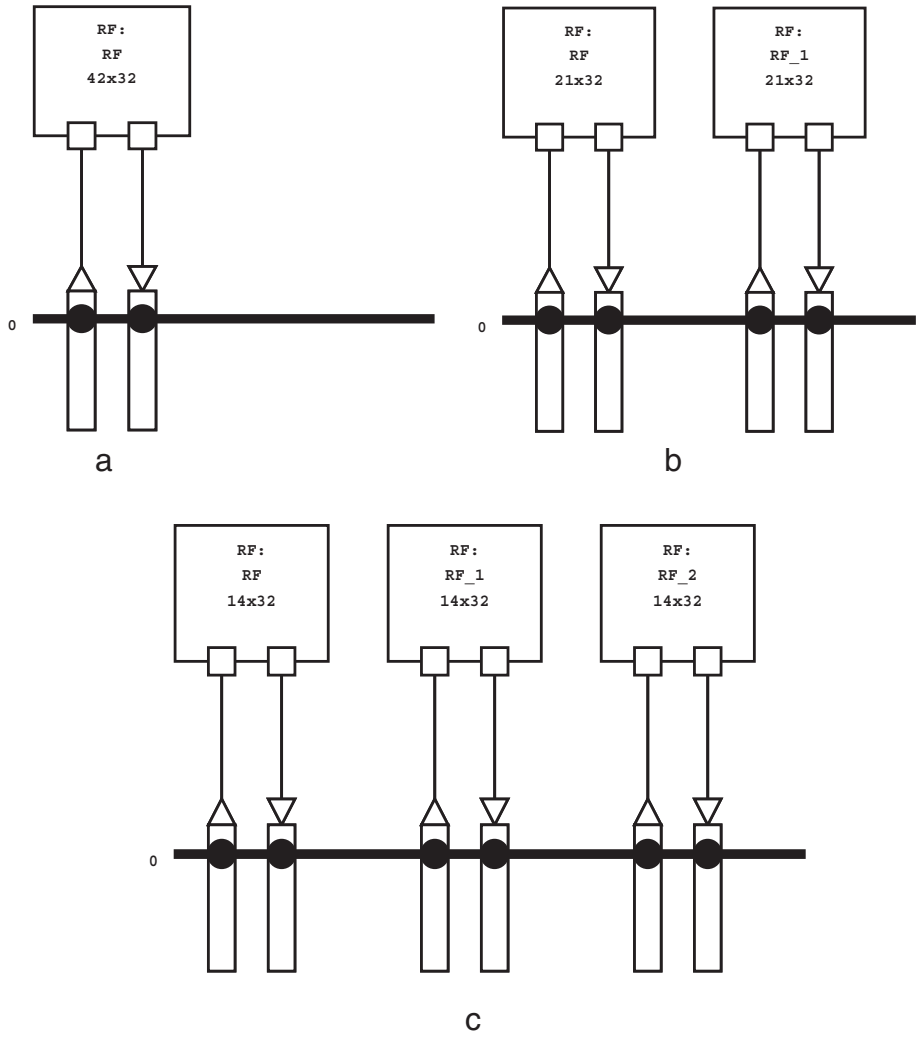

Figure 3 Register file with one read and one write port. An example of a register file with one read and one write port, connected to the interconnection bus. **(a)** SS 1 × 1 × 1. **(b)** MS 2 × 1 × 1. **(c)** MS 3 × 1 × 1.

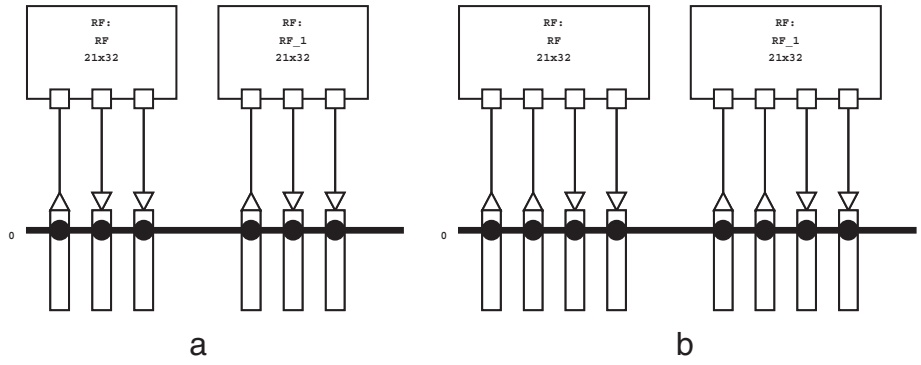

Figure 4 Two register files with two read and one or two write ports each. An example of two register files with two read and one or two write ports in each, connected to the interconnection bus. **(a)** MM 2 × 2 × 1. **(b)** MM 2 × 2 × 2.

is transferred to the interconnection network and, while being visible in the data path, can be optimized more effectively by compiler techniques and in the design space exploration. Usually, the register file tends to be the end point of the critical path of the processors. Increasing the input control capacitance and delay by adding a write port has an effect not only on the input control logic of the register file, but also in the interconnection network. The addition of read port to the register file has minor effects to the capacitance and the delay of the control logic.

Remaining parts of our processor stay the same, with three-integer ALUs, one multiplier, a load store unit, and eighteen buses to accommodate for ILP available across our set of benchmarks.

We schedule our selected benchmarks for the set of preselected TTA designs four times. First, we use our previously published top-down scheduling algorithm [3], with actual software bypassing disabled, and collect resulting data, such as the number of clock cycles the benchmarked application needs to end and the number of reads and writes of the registers in all available register files. We also collect information about instruction width and the number of socket-to-bus connections.

Afterwards, we enable software bypassing with a top-down scheduler and recompile our set of benchmark applications for all the selected architectures again, collecting the same data as above.

Collecting information from more conservative software bypassing implementation, we repeat the steps above using our new, early software bypassing during the bottom-up scheduling algorithm, presented in Section 4.1. We collect data without software bypassing enabled and again with software bypassing enabled. This will allow us to consider differences that scheduling and bypassing strategies have.

For second test, for each combination of benchmark and architecture, we remove unused connections. This has no effect on actual cycle counts, or the number of register file reads and writes, but reduces the number of socket-to-bus connections and in some cases narrows the instruction word width as the number of bits required to address all sockets connected to any bus can drop. The application of bypassing of course changes the schedule, so in some cases, the number of removed connections can be higher for the case without bypassing and vice versa.

Taking those eight sets of data, we synthesize each architecture to 130-nm CMOS standard cell ASIC technology with Synopsys Design Compiler and a run gate-level simulation with Mentor ModelSim. From the results of the gate-level simulation, we acquire gate activity for the Synopsys Power compiler. From the Synopsys Power compiler, we acquire power used by individual architectural components of the processor core, such as interconnection network, individual register files, function units, instruction fetch, and instruction decode.

The processors were synthesized to a 250-MHz clock frequency (4-ns clock period) since for this value, architectures with a larger number of read and write ports can still be synthesized and meet timing constraints.

In addition, after collecting the results from the experiments as described above, we select the architecture configuration that showed the best energy efficiency with software bypassing across our set of benchmarks and take the benchmark's cycle counts as a measure of real-time performance at a 250-MHz frequency. Connectivity reduction does not improve clock cycle performance; therefore, to achieve the same real-time performance, we compute the required frequency as follows: Required frequency $=$ 250 MHz $\times$ (Reduced connectivity cycles/Bypassed cycles). We then attempt to synthesize each of the benchmarks for it's required frequency, run gate-level simulation, and collect power data, as described above.

Results of our experiments are discussed in detail in Section 6.

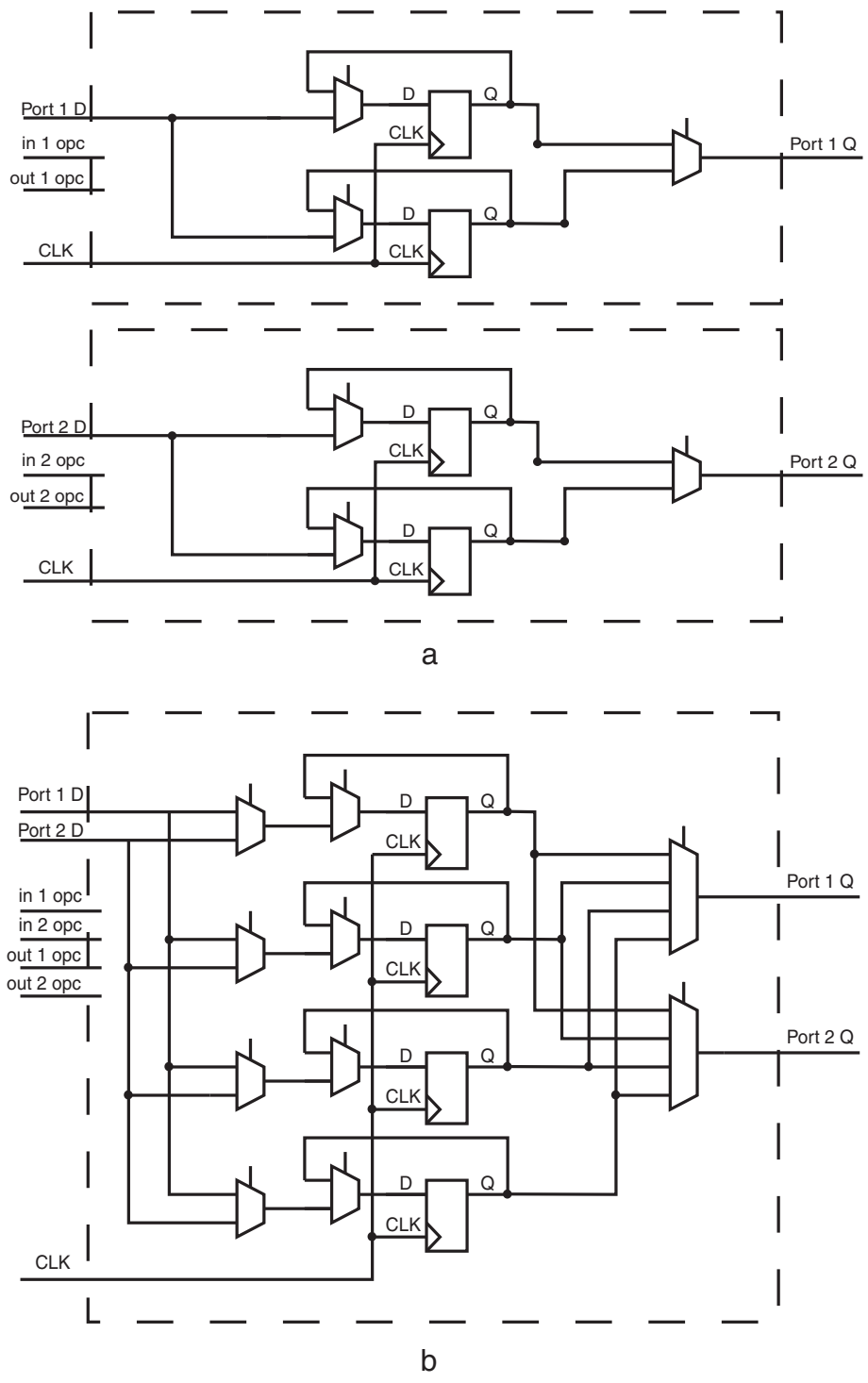

Figure 5 Examples of implementations of register files. In this figure, we display the implementation of **(a)** two register files, each with one read and one write port and **(b)** a single register file with two read and two write ports.

6 Results

In the following, we first present results collected during setting up our experiment in Subsection 6.1. Afterwards, we discuss energy results obtained by synthesis and simulation in Subsection 6.2. In addition, we discuss results collected when trying to match real-time performance obtained by the use of software bypassing via synthesizing for higher frequency in Subsection 6.3.

6.1 Data collected before synthesis and simulation

Figure 6a shows results we collected when scheduling our set of benchmarks for the selected architectures with and without software bypassing. The cycle counts do not change with connectivity reduction applied as reduction is done after the schedule is generated. All the results are normalized to the worst case of top-down scheduler (TD), which is the single register file with a single read and single write port. It is clear from the figure that the bypassing has a dramatic effect on the clock cycles. Additional scheduling freedom and direct read of result from the result register to the operand saves a significant number of clock cycles. This is a factor we expect to hugely contribute to total energy reduction when using software bypassing. The difference between our conservative top-down scheduling and bypassing compared to the more opportunistic bottom-up scheduling (BU) and bypassing is also visible on this figure. The use of the top-down scheduling algorithm translates to slightly better results in terms of clock cycle counts compared to the bottom-up scheduler, in particular for an architecture with a single read and single write port in the single register file. This is mainly caused by the use of another optimization, the *delay slot filling*. Our implementation of delay slot filling takes advantage of predicated execution, allowing to fill in more than delay slots after the branch operation. Practically, as soon as the predicate used by branch instruction is computed, the operations from the following basic blocks, including the top of the loop body, can be scheduled, guarded by the same predicate. When scheduling the loop body, the top-down scheduler schedules operations not on the critical path, such as loop counter increment and loop repeat condition evaluation, as early as possible, while the bottom-up scheduler schedules them as late as possible, virtually just before the branch takes place. In case

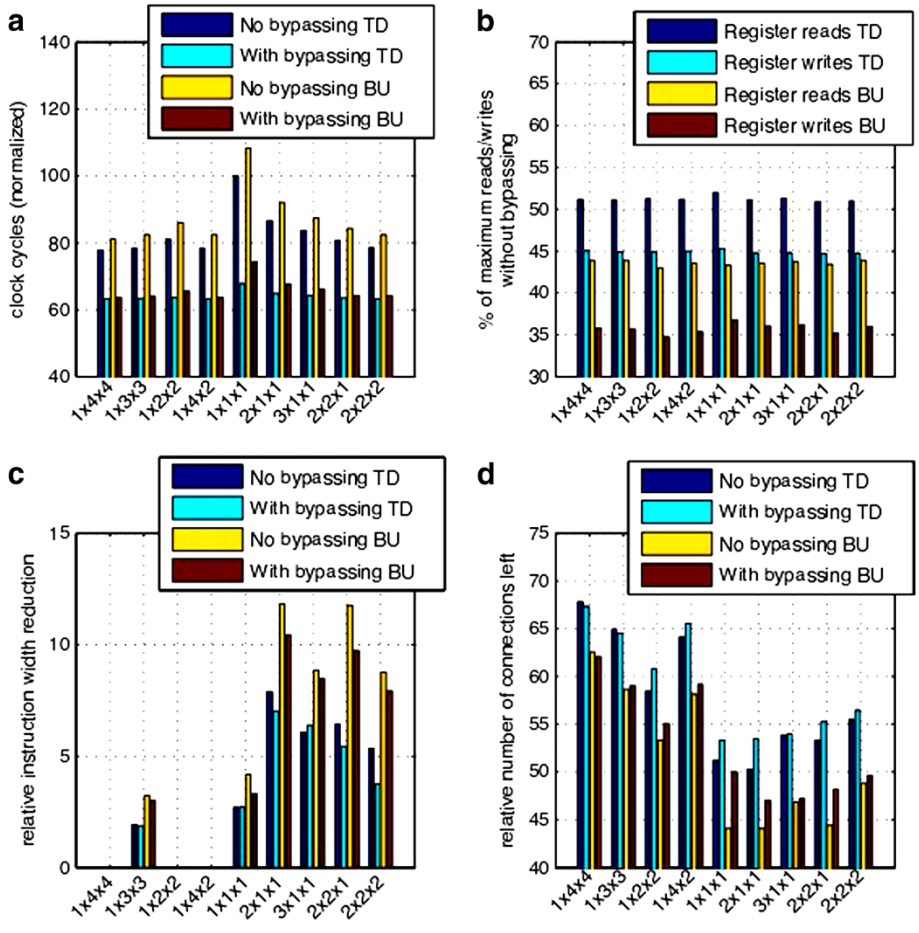

Figure 6 Statistics collected from software bypassing and reduced connectivity for each architecture. In this figure, we present **(a)** the number of clock cycles normalized to the worst case of no bypassing and top-down scheduler, **(b)** the number of register reads and writes normalized to the case with no bypassing and top-down scheduler, **(c)** reduction in instruction width normalized to the case without connectivity reduction with top-down scheduler, **(d)** the number of connections left after connectivity reduction normalized to the case without connectivity reduction with top-down scheduler.

loop counter increment is used immediately at the beginning of the basic block, in the case of the top-down scheduler, it is therefore possible to schedule the conditional execution of the operations from the top of the basic block even before the branch operation, and the branch operation then changes the control flow to the later part of the basic block after those filled operations, eventually creating a smaller loop body. In the case of the bottom-up scheduler, the late computation of the loop counter and loop predicate prevents this from happening. Further optimization of identifying such dependencies and taking a mixed approach of bottom-up and top-down scheduling would bring the best of both worlds.

However, once the bypassing is enabled, the cycle count differences become much smaller, removing the penalty of worse starting point of the bottom-up scheduler. More detailed results of the achieved clock cycles of the two typical cases of our benchmarks, *gsm* and *motion*, are presented in Figure 7a,b respectively. Here, we observe that the effects of using bypassing as well as different register file configurations varies between individual benchmarks. The addition of read and write ports improves the cycle count to a lesser extent than software bypassing, with exception of the *gsm* benchmark, where bypassing with a single register file with a single read and single write port does not improve performance as much as the addition of read and write ports or more register files. On the contrary, for the case of the *motion* benchmark, bypassing leads to rather uniform clock cycle counts through the range of architectures.

In Figure 6b, we can see reduction in the number of reads and writes of the register files for each of the selected architectures when software bypassing is applied. Results are normalized to the number of reads and writes without bypassing with TD. It is clear from this figure that

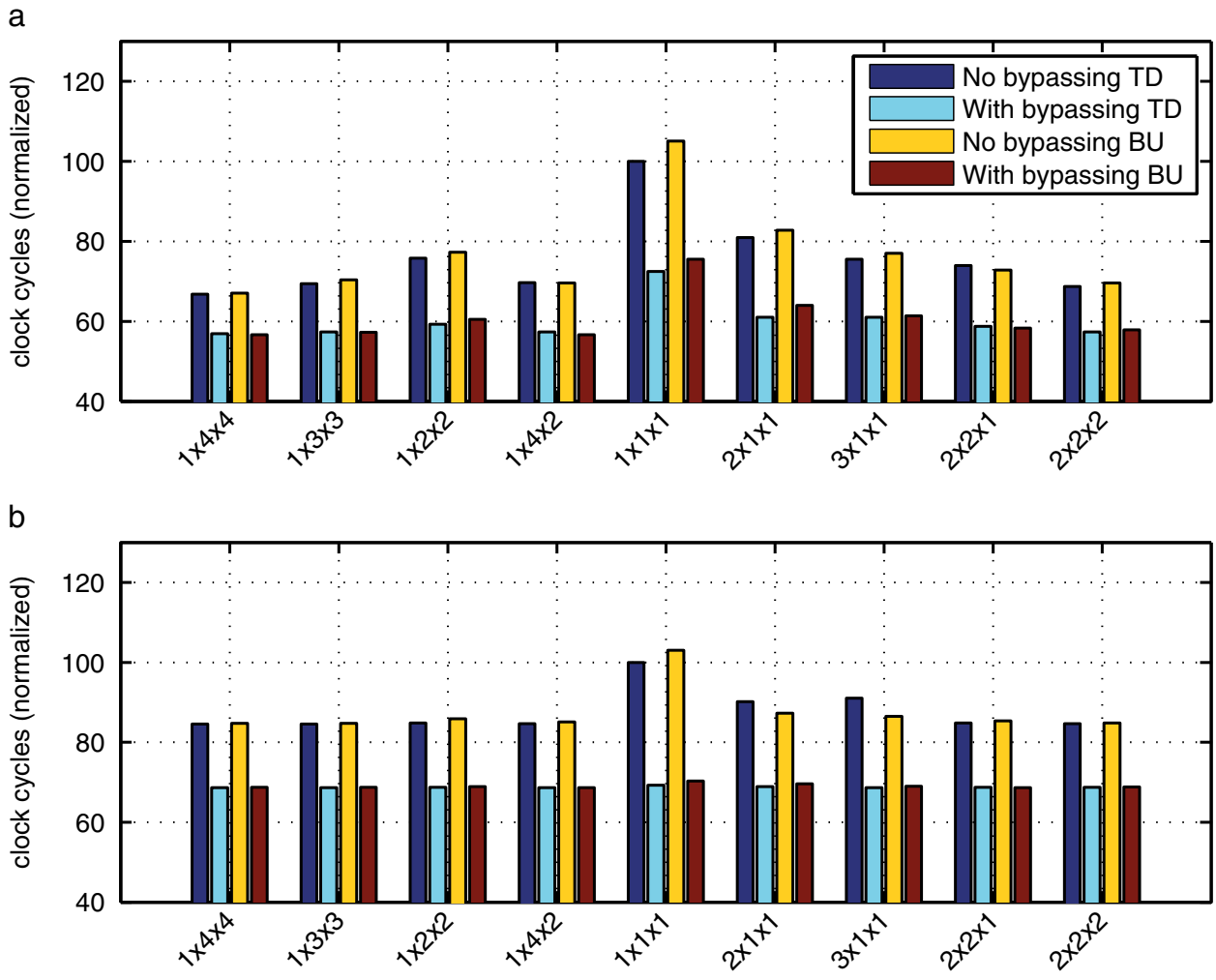

Figure 7 The number of clock cycles with and without software bypassing for each architecture. In this figure, we display the number of clock cycles normalized to the worst case without software bypassing with top-down scheduler for **(a)** *gsm* and **(b)** *motion*.

the use of bottom-up scheduling and bypassing leads to a significant decrease in the number of register reads and writes compared to more conservative bypassing using top-down scheduling. As previously described, a more detailed analysis of two individual benchmarks, *mips* and *motion*, is shown in Figure 8a,b, respectively. We observe that in the case of the *mips* benchmark, about 45% of register reads and writes are eliminated when using software bypassing, which represents the worst result from our set of benchmarks. On the other hand, the *motion* benchmark shows a dramatic difference in reduction of register file reads between the top-down and bottom-up schedulers, reducing over 80% of register writes and 60% of register reads.

In Figure 6c, we can see reduction in instruction width when connectivity reduction is applied, and Figure 6d shows the number of socket-to-bus connections left after connectivity reduction. It can be seen from Figure 6c,d

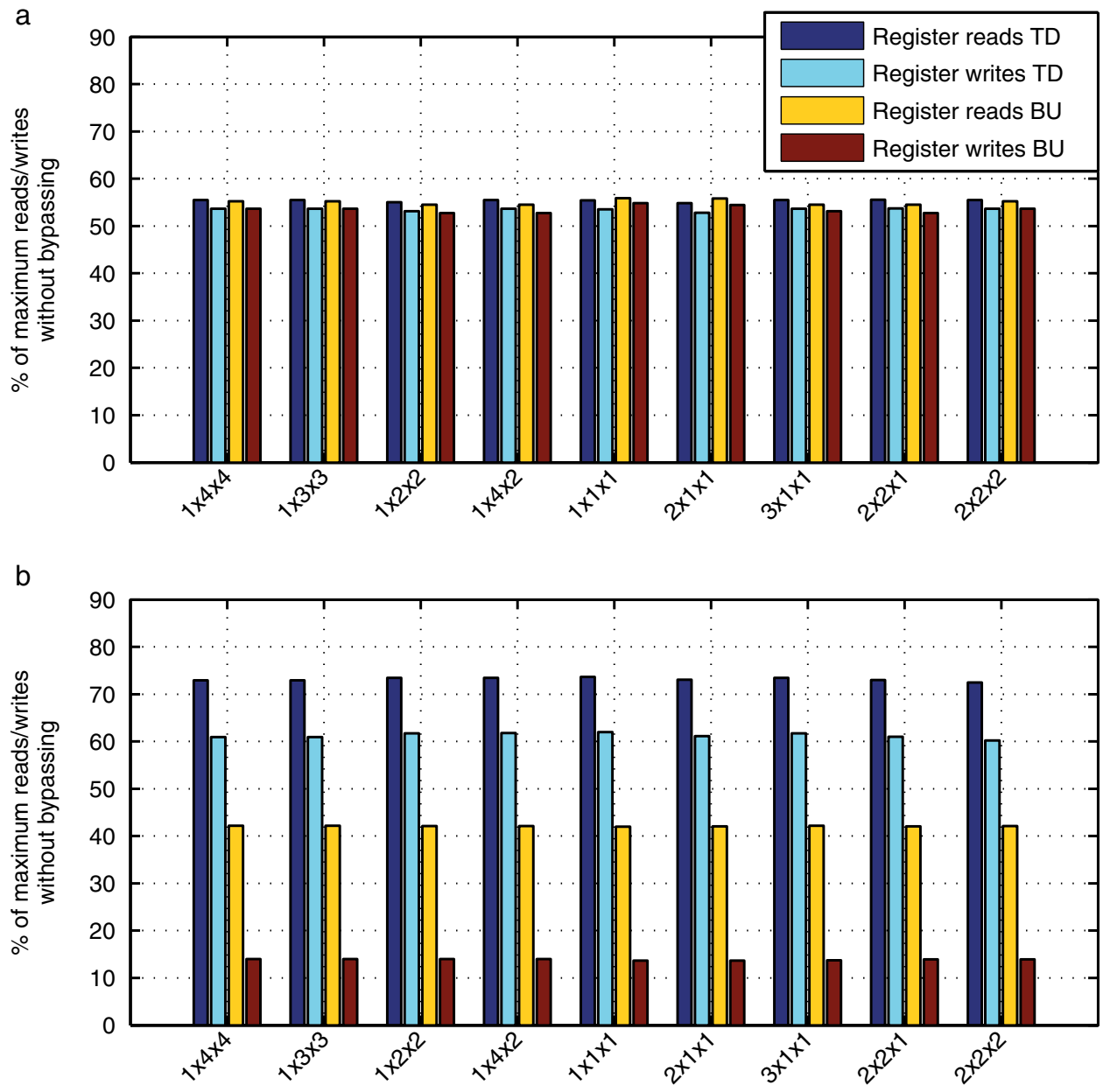

Figure 8 The number of register file reads and writes left after bypassing for each architecture. In this figure, we display the number of dynamic register file reads and writes after the application of software bypassing normalized to the worst case without software bypassing with top-down scheduler for **(a)** *mips* and **(b)** *motion*.

that there is variation when applying connectivity reduction for cases without bypassing and with bypassing. Since bypassing causes changes to the schedule, there are added direct data transports between function units and the schedule is more compact, leading to more activity per cycle. On the other hand, the number of data transports between function units and register file decreases with software bypassing; therefore, connectivity to the register file can be reduced. It can be seen from those two figures that once again bottom-up scheduling and early bypassing leads to more reduction of instruction width and a lower number of connections left; variation is however only about 5%.

A detailed view of the reduction of instruction width with software bypassing of benchmarks *adpcm* and *blowfish* is presented in Figure 9a,b, respectively. Here, we observe that for three of the architectures with a single register file, the removal of unused connection did not lead to a decrease in instruction width at all. For other architectures, we observe that reduction for architectures with several register files varies, following a very similar pattern as seen in Figure 6c, with variations of only few percentage points.

A detailed view of the number of socket-to-bus connections left after connectivity reduction for *adpcm* and *blowfish* is shown in Figure 10a,b, respectively. We observe a clear general pattern that actually spread through all the benchmarks. The least number of connections left is generally present with several simple register files, while a single multi-ported register file requires a larger number

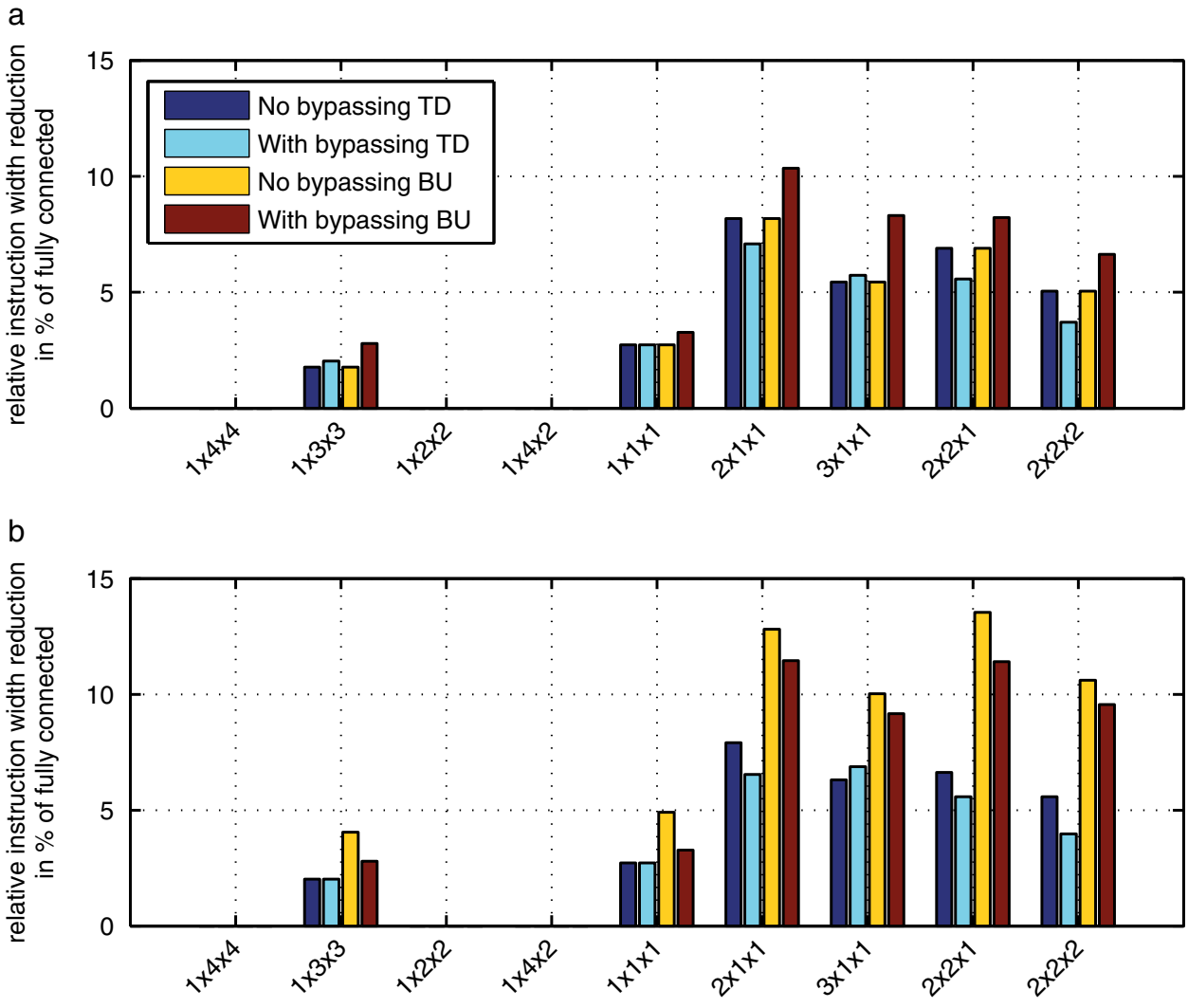

Figure 9 Reduction in instruction width after removing connections for each architecture. In this figure, we display reduction in instruction width after connectivity reduction compared to the case without reduced connections and software bypassing with top-down scheduler for **(a)** *adpcm* and **(b)** *blowfish*.

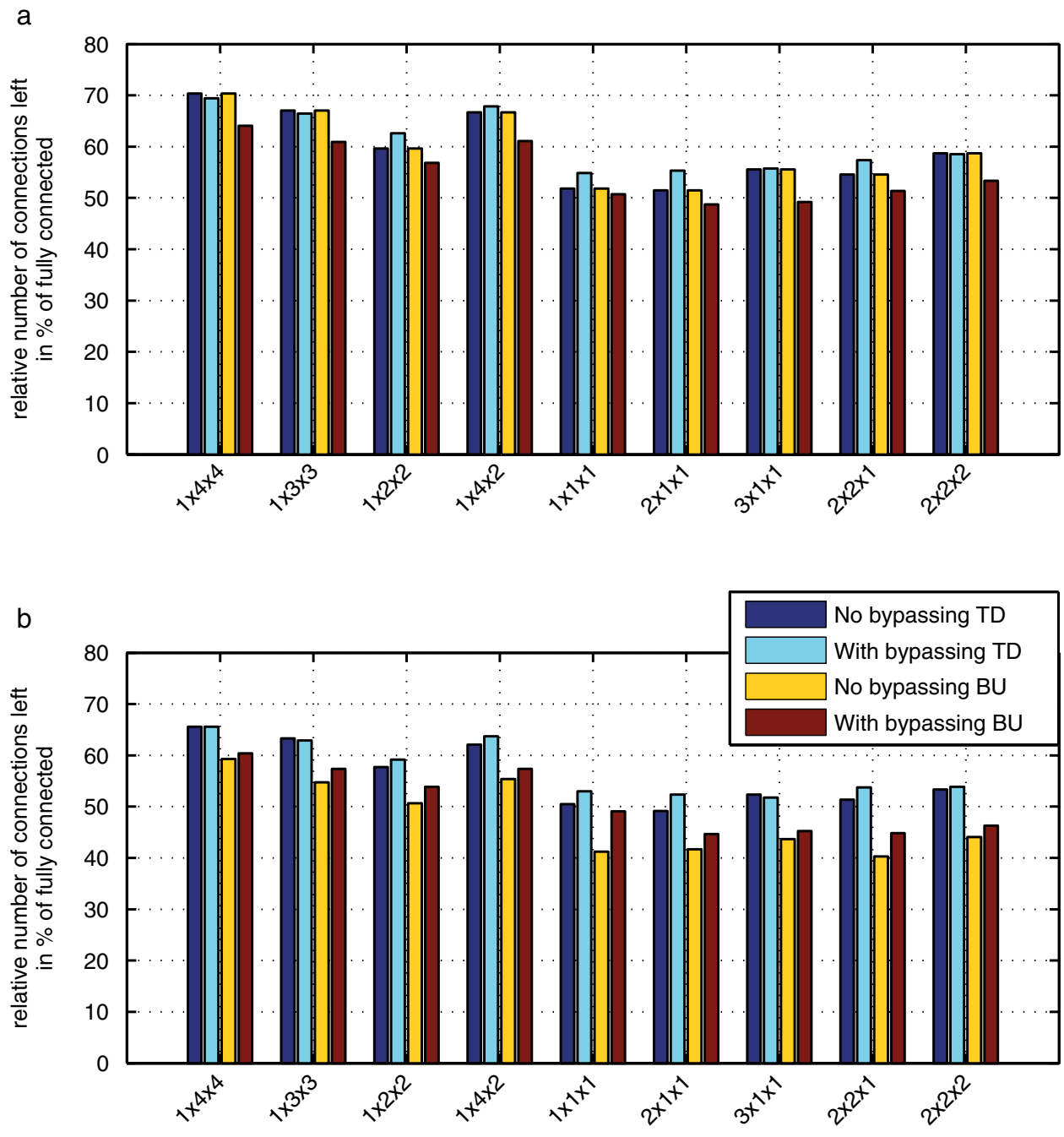

Figure 10 The normalized number of connections left after connectivity reduction for each architecture. In this figure, we display the number of connections left after connectivity reduction normalized to the case without connectivity reduction and software bypassing applied with top-down scheduler for **(a)** *adpcm* and **(b)** *blowfish*.

of connections to remain. It can also be seen from this figure that software bypassing has a little effect on the number of connections removed, with the largest difference of only about 5% and variation between the top-down and bottom-up schedules of only about 10%.

Overall, Figure 6 shows that the connectivity reduction indeed leads to reduction in instruction width and successfully removes a large number of socket-to-bus connections and that the software bypassing produces large reduction in dynamic register reads and writes as well as large drop in cycle counts, with eventually simpler architectures with a single read and single write port in the register file outperforming much larger architectures with multi-ported register files without bypassing.

The combination of these reductions has an effect on the power of individual processor components and results in energy reduction.

However, we can also observe that the effect of software bypassing on the successful removal of connections is clearly limited. There is typically at most 5% variation in the number of connections removed between using software bypassing or not and similarly a small variation in instruction width reduction. A larger variation is visible between top-down scheduling with conservative bypassing and bottom-up scheduling with early bypassing. We observe that while the older top-down scheduler provides a better starting point in terms of clock cycles than new bottom-up scheduler implementation, the difference narrows when bypassing is enabled, and overall, in terms of register file accesses, connectivity removal, and instruction width, the novel algorithm performs better.

6.2 Results of synthesis and simulation

After performing gate-level simulation on all our benchmarks, architectures, and optimization combinations as described in Section 5, we collected power data for individual processor components. We computed the energy of individual processor components using the common formula Energy = Power × Cycles/Frequency. We computed the averages for all benchmarked applications to focus on overall trends, in addition to individual benchmarks.

Figure 11 shows the energy of all processor core components as it changes for architectures and applied connectivity reduction and bypassing. Specifically, on Figure 11a, *FC TD* denotes a fully connected architecture configuration with top-down scheduling, *RC TD* denotes an architecture with reduced connectivity with top-down scheduling, *FC BU* denotes a fully connected architecture configuration with bottom-up scheduling, and *RC BU* denotes an architecture with reduced connectivity with bottom-up scheduling. On Figure 11b, *FC + bypass TD* denotes a fully connected architecture with software bypassing applied with a top-down scheduler, *RC + bypass TD* denotes software bypassing applied followed by the application of connectivity reduction with a top-down scheduler, *FC + bypass BU* denotes a fully connected architecture with software bypassing applied with a bottom-up scheduler, and *RC + bypass BU* denotes software bypassing applied followed by the application of connectivity reduction with a bottom-up scheduler.

It can be seen from the figure that the effect of connectivity reduction is larger for cases with single-ported register files. In each of the cases, the effect of software

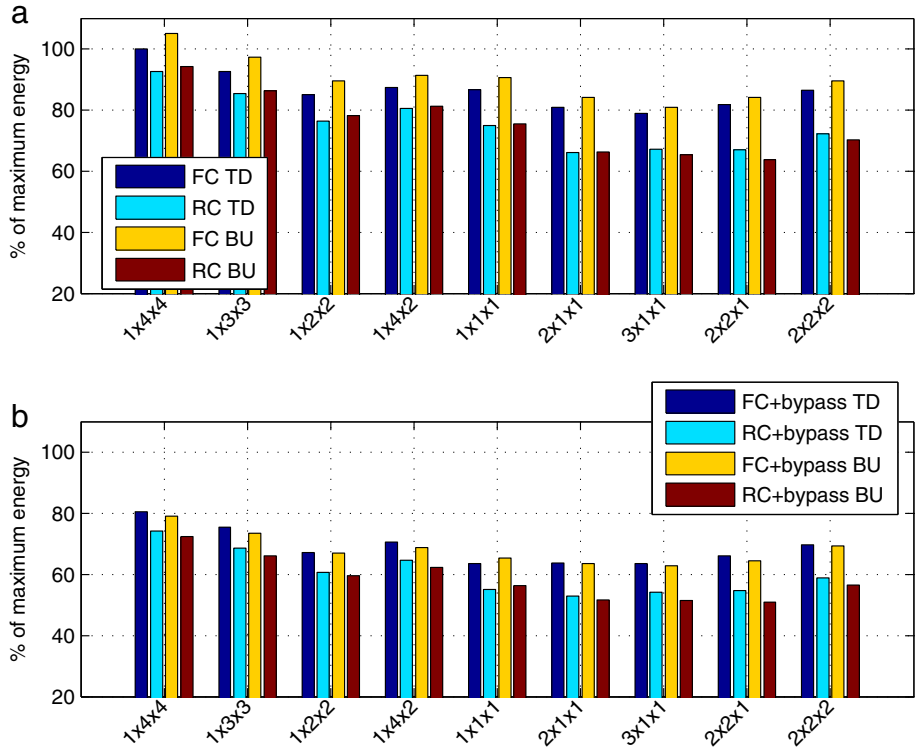

Figure 11 Average spent energy for each architecture. In this figure, we display the relative energy for **(a)** fully connected architectures and reduced connectivity without bypassing and **(b)** fully connected architectures and reduced connectivity with software bypassing.

bypassing on the overall energy is larger than that of connectivity reduction. Minimal energy can be found with the combination of both, software bypassing and connectivity reduction, as they complement each other, with about 50% reduction compared to the largest architecture without bypassing and connectivity reduction. It is notable that bottom-up scheduling without software bypassing produces worse energy results due to the penalty of slightly higher clock cycles. However, the application of connectivity reduction erases this penalty leading to results similar to top-down scheduling, and the application of bypassing and connectivity reduction produces slightly better energy results for bottom-up scheduling than top-down for most of the architectures. For individual benchmarks of *mips* and *motion*, Figure 12a,b shows the same data, with the left side of the figure displaying data without bypassing and the right side of the figure with bypassing enabled. The effect of connectivity reduction and software bypassing follows the patterns observed previously in Figure 6, with Figure 12a representing the worst observed result and Figure 12b the best observed result.

Overall, results in Figure 11 show that the combination of software bypassing and connectivity reduction leads to energy savings up to 50% using single-ported register files compared to the energy required by the largest architecture with four read and four write ports while maintaining or improving cycle counts.

Looking into more detail, Figure 13a shows the effect of bypassing and connectivity reduction on the energy of register files, with the left graph displaying data without bypassing and the right graph with bypassing. Due to it's nature, connectivity reduction does not contribute significantly to the reduction of energy of register files (with the exception of possible buffer distribution, as discussed previously in Section 5), and the main benefit is from the effect of bypassing, combining drop in cycle counts (Figure 6a) as well as actual reduction in the number of dynamic register reads and writes (Figure 6b). It can be seen from this figure that the addition of write port increases the energy dramatically, this trend is visible regardless of the use of software bypassing. As a matter of fact, from the single register file, we observed a linear progression when adding register file write ports. Starting from the architecture with a single register file with a single read and write port (denoted as $1 \times 1 \times 1$) through two read and two write ports ($1 \times 2 \times 2$), three read and write ports ($1 \times 3 \times 3$) until the most expensive four read and write ports ($1 \times 4 \times 4$). The addition of read ports (architecture with four read and two write ports, denoted as $1 \times 4 \times 2$), however, does not significantly differentiate from two read and two write ports. A similar observation can be made for the case of two register files. The architecture with two register files, each with a single read and single write port ($2 \times 1 \times 1$) does not differentiate significantly from the architecture with two read and single write ports in each register file ($2 \times 2 \times 1$). However, the addition of a second write port to both register files ($2 \times 2 \times 2$) leads to a significant jump in energy consumption.

While in Figure 6a,b we observed a fairly consistent clock cycle performance across the range of architectures after the application of software bypassing as well as reduction in register file reads and write, Figure 13a clearly shows how expensive more complex register file configurations are, even with software bypassing.

Figure 14a,b shows the breakdown of the register consumption of register files for *mips* and *motion* benchmarks, with a visible effect of dramatic reduction in register file reads and writes observed in Figure 8b causing a significant difference in energy between software bypassing with top-down scheduler and software bypassing with bottom-up scheduler in the case of *motion*.

Figure 13b shows the effect of bypassing and connectivity reduction on the interconnection network. Both reduced the connectivity and bypassing results in the drop of energy, with the combination of both providing the best results. However, while connectivity reduction causes a drop of energy by actually removing components that consume energy, the benefit from software bypassing is largely due to a decrease in cycle counts and interconnection network traffic. The use of new bottom-up scheduling and bypassing algorithm results in better interconnection energy results for all of the architectures.

Figure 15a,b shows the same information for *mips* and *motion*, respectively. As discussed in Section 5, the register files and interconnection network interact. Therefore, when synthesizing for reduced connectivity, the synthesis tool redistributed capacitance between the interconnection network and register file, which shows, for the architecture marked as $3 \times 1 \times 1$, as a slight increase in interconnection network energy.

In order to investigate the claim that connectivity reduction leads to a decrease in processor core energy, we present Figure 16a,b where the breakdown for *mips* and *motion* benchmarks shows combined energy of register file(s) and interconnection network, respectively. We observe that in cases where connectivity reduction caused a slight increase in register file energy or interconnection energy, the sum of those two still shows minimal difference compared to fully connected architectures. The same result has been observed for all the benchmarks.

Figure 17a shows the results of energy reduction in decode, following a similar suite. A notable exception

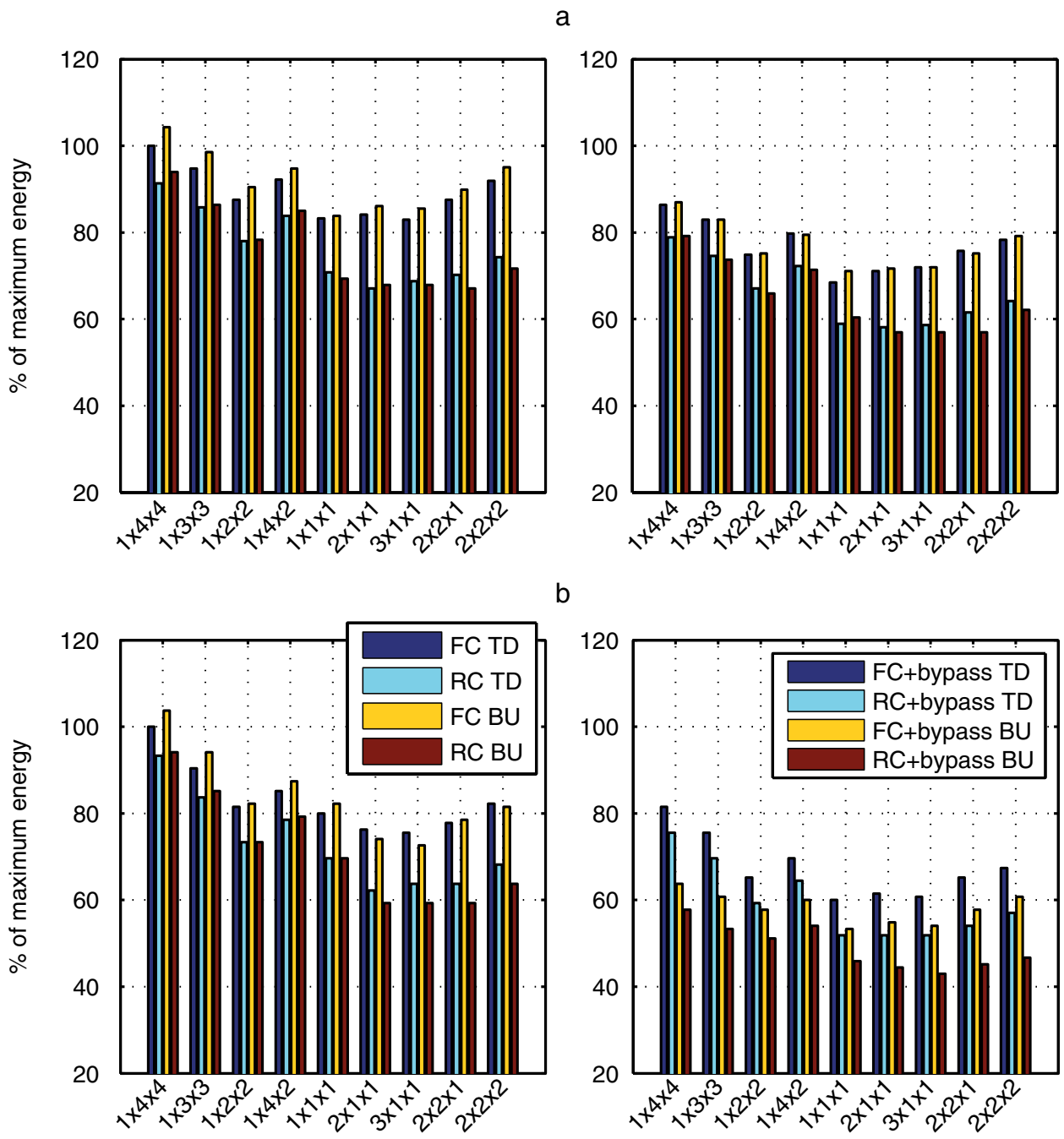

Figure 12 Relative energy for (a) *mips* and (b) *motion*. In this figure, we display the relative energy without bypassing in the left side and the relative energy with bypassing in the right side, normalized to the case with fully connected architectures without software bypassing with top-down scheduler.

is the visible effect of connectivity reduction, producing better energy savings than software bypassing for cases with multiple register files, while for single register files, bypassing seems to have similar benefits. Detailed results for *mips* and *motion* benchmarks are shown in Figure 18a,b, respectively.

Finally, Figure 17b shows the effects of bypassing and connectivity reduction on the instruction fetch energy. Here, the impact of connectivity reduction is caused by the decrease in instruction width, as outlined previously in Figure 6c, following the same trend. The effect of software bypassing in this case is caused only by the reduction

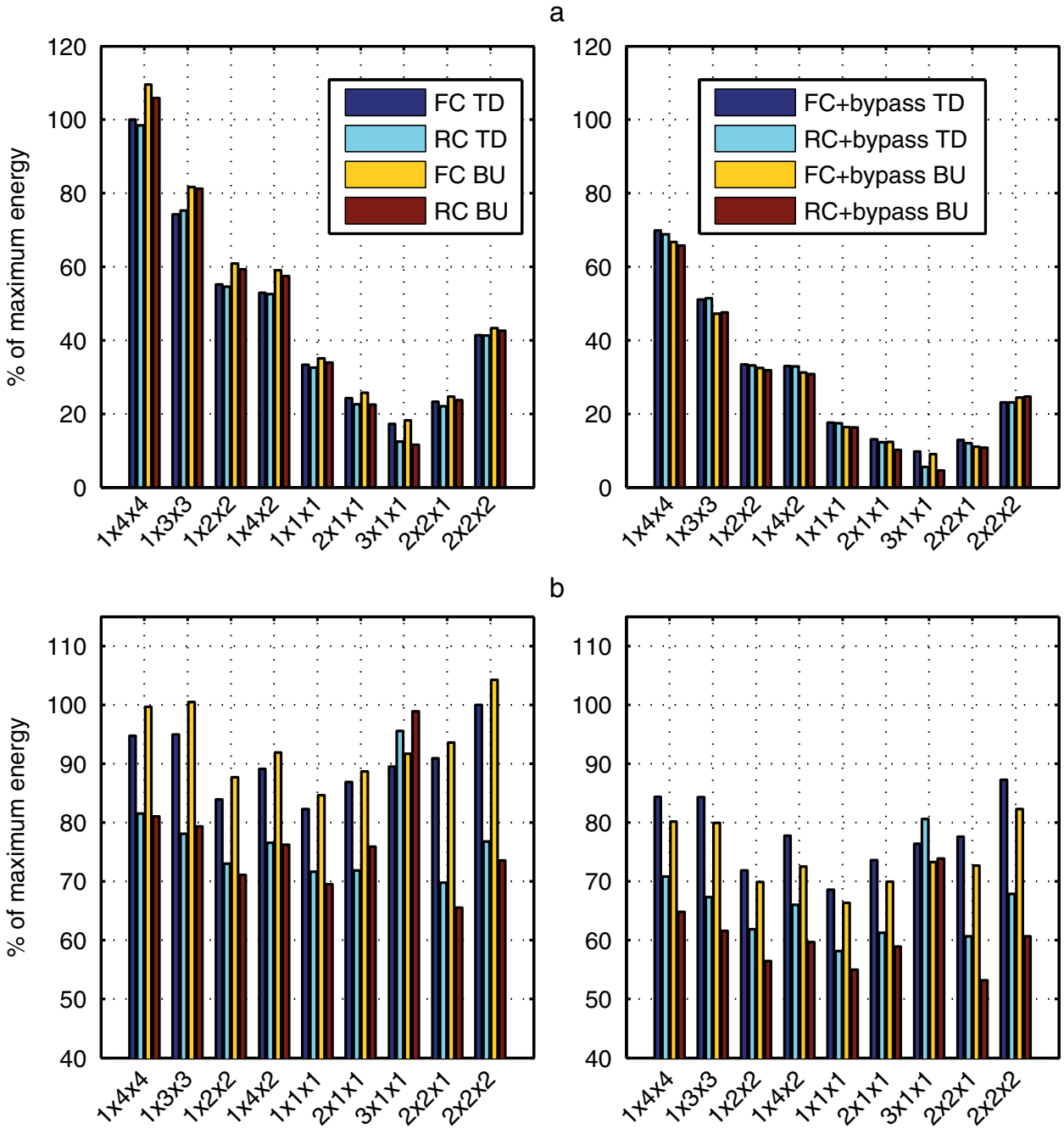

Figure 13 Relative energy consumption in (a) register files and (b) interconnection network. On the left side of this figure, we display the relative energy without software bypassing and on the right side, the relative energy with software bypassing.

in clock cycles. We can observe a larger impact of connectivity reduction with the bottom-up scheduler. The breakdown of results for *mips* and *motion* benchmarks is shown in Figure 19a,b, respectively.

While previously we considered various components in the processor core, the interesting question is how large the impact of those is on the total core energy. Figure 20 shows the breakdown of the processor components for different architectures. On left side of the figure, the results with the top-down scheduler are presented, and in the right side of the figure, the results with the bottom-up scheduler are presented. In particular, we selected

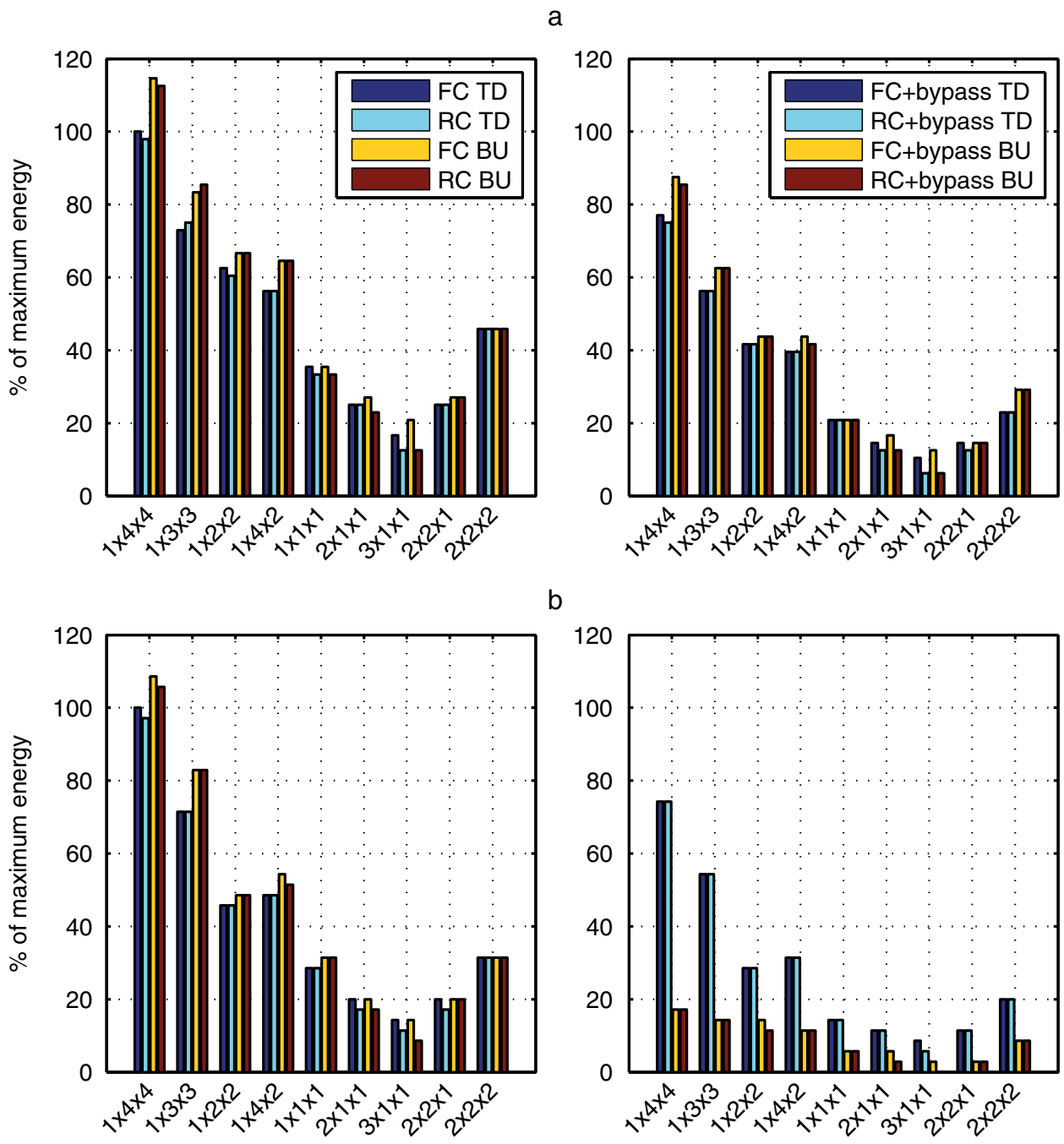

Figure 14 Relative energy consumption of register files for (a) *mips* and (b) *motion*. On the left side of this figure, we display the relative energy of register files without bypassing and on the right side, the relative energy of register files with bypassing.

components consuming significant amounts of energy, such as decode, fetch, interconnection network, and register files in previous results, making an observation that the actual amount of computation carried out by function units remains the same across the architectures and is not directly affected by connectivity reduction either. The application of software bypassing decreases the number of clock cycles but does not change the number of operation execution in function units; therefore, reduction in the energy of function units is due to the change in clock cycles. In a similar way, the reduced connectivity does not influence the energy of function units.

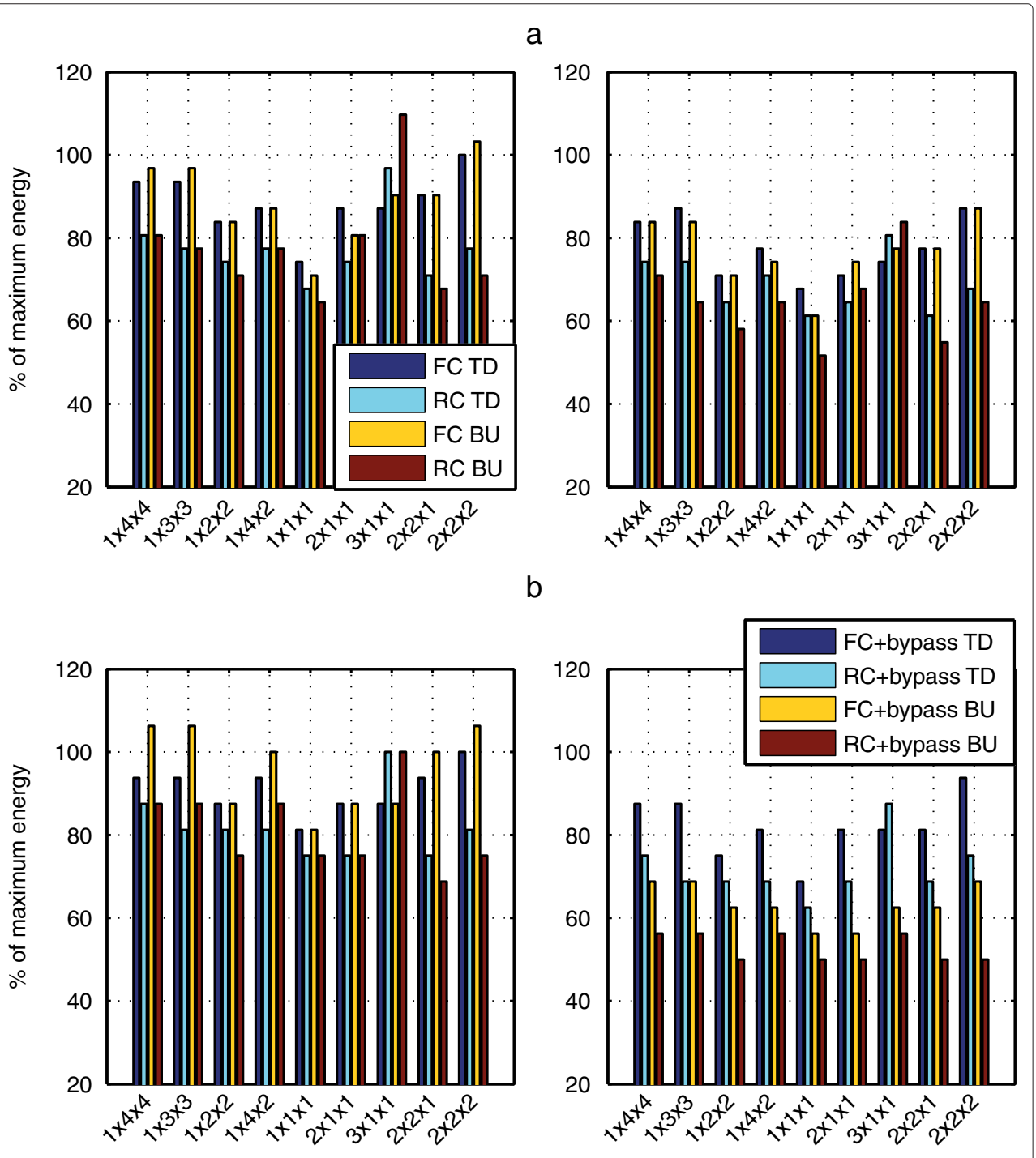

Figure 15 Relative energy consumption of interconnection network for (a) *mips* and (b) *motion*. On the left side of this figure, we display the relative energy of the interconnection network without bypassing and on the right side, the relative energy of the interconnection network with bypassing.

6.3 Matching real-time performance via synthesizing for higher clock frequency

We observed that several architectures resulted in similar low-energy requirements for cases with bypassing. We selected an architecture with two register files, each with a single read and single write port. We attempted to achieve the same real-time performance, without bypassing, while applying only connectivity reduction and synthesizing for higher clock frequency.

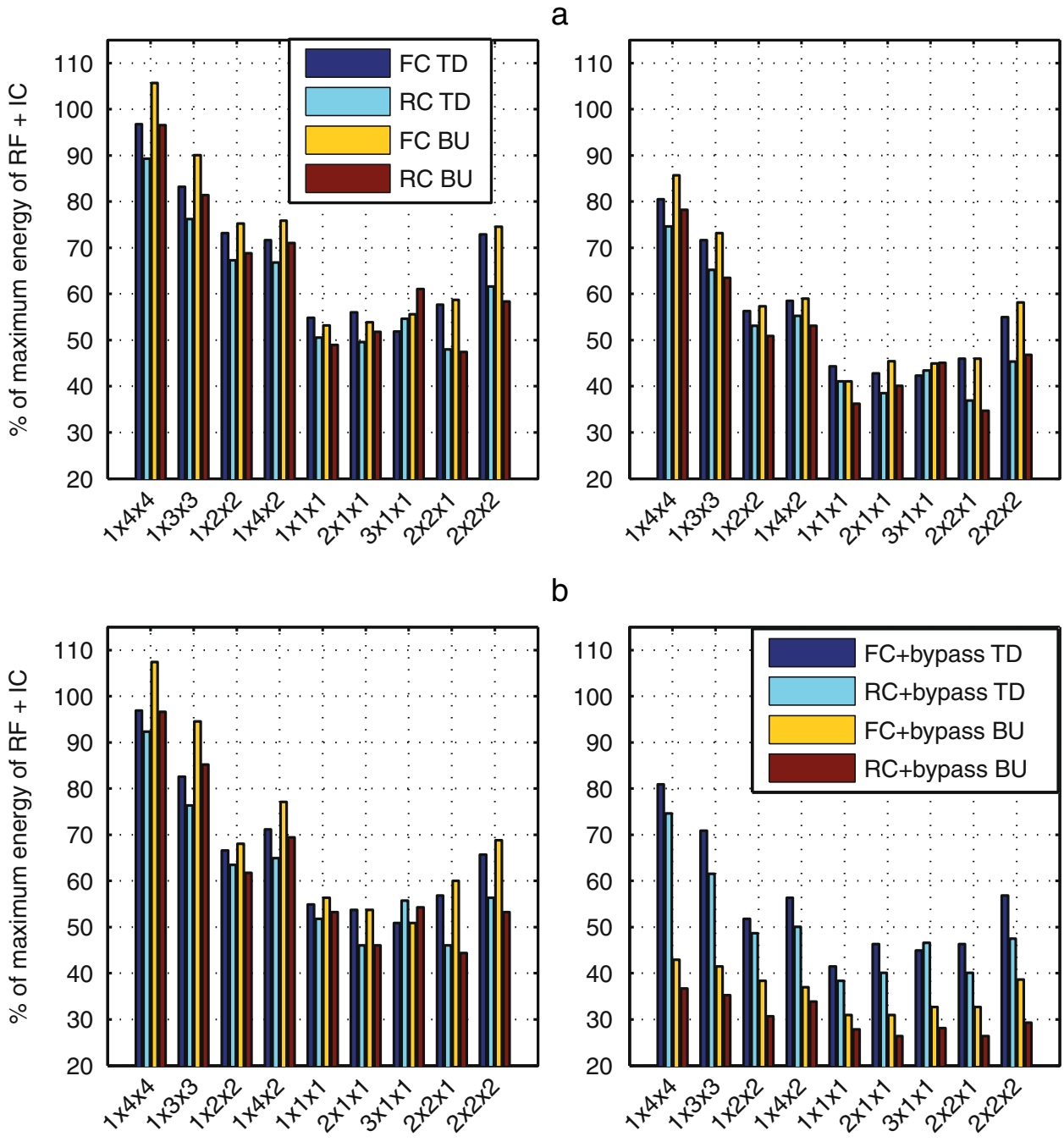

Figure 16 Relative energy consumption of interconnection network and register files for (a) *mips* and (b) *motion*. On the left side of this figure, we display the relative energy of the interconnection network and register files without bypassing and on the right side, the relative energy of the interconnection network and register files with bypassing.

Table 2 shows, for each benchmark, clock cycles with bypassing during bottom-up scheduling and without. It also shows the required time per cycle and equivalent clock frequency for each benchmark to achieve the same real-time results without bypassing as with bypassing. Setting timing constraints for each benchmark individually, we synthesized the architecture, simulated at gate level, and provided gate activity to the power compiler, as discussed in Section 5.

We observed that for all benchmarks, designs can be successfully synthesized and simulated at required frequencies. The results of this experiment are presented

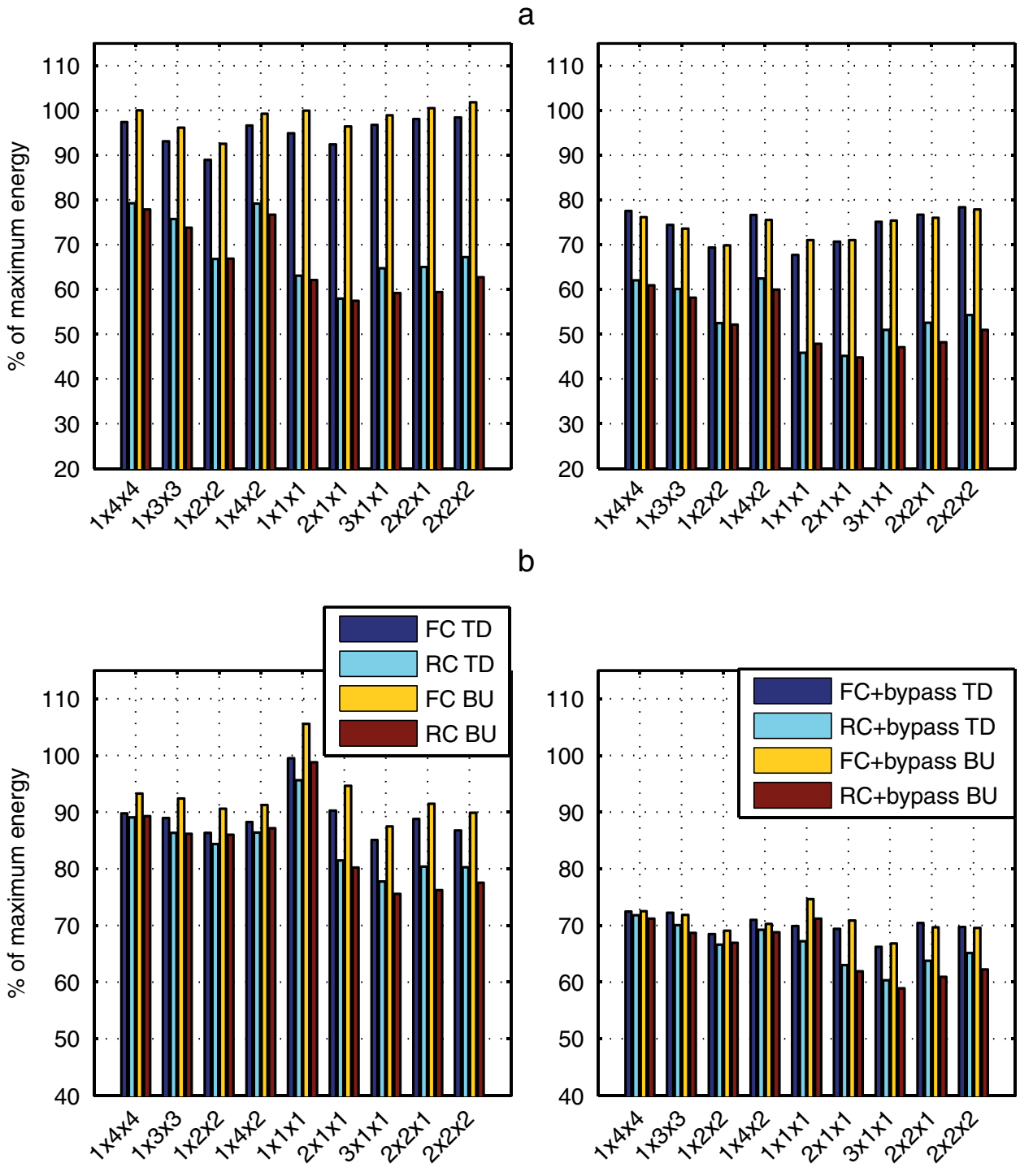

Figure 17 Relative energy consumption in (a) decode and (b) fetch. On the left side of this figure, we display the relative energy without software bypassing and on the right side, the relative energy with software bypassing.

in Figure 21, with a fully connected architecture with bypassing with a top-down scheduler (FC + bypass TD), software bypassing with reduced connectivity architecture with a top-down scheduler (RC + bypass TD), with fully connected architecture with bypassing with a bottom-up scheduler (FC + bypass BU), software bypassing with reduced connectivity architecture with bottom-up scheduling (RC + bypass BU), and the data acquired

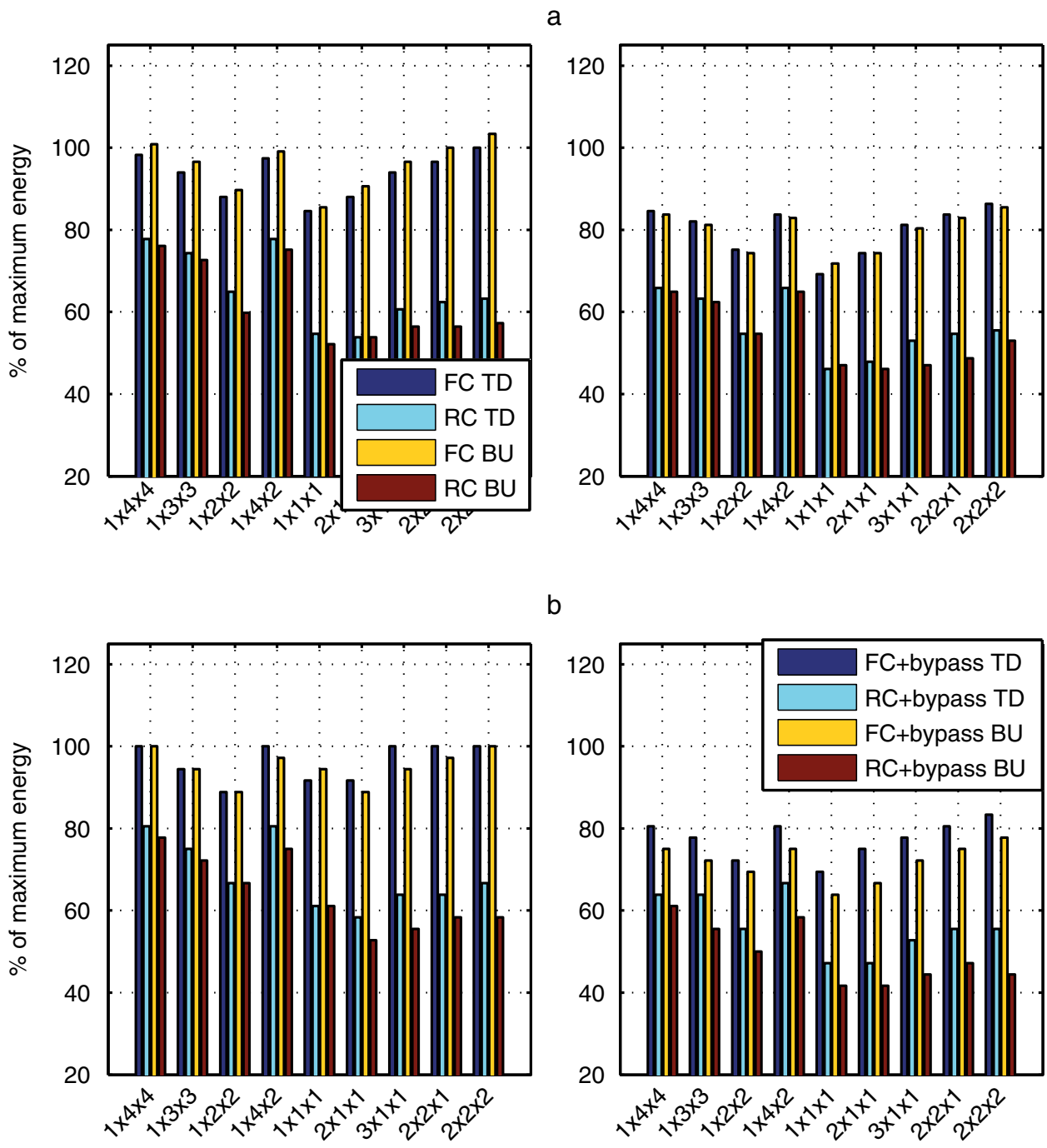

Figure 18 Relative energy consumption of decode for (a) *mips* and (b) *motion*. On the left side of this figure, we display the relative energy of decode without bypassing and on the right side, the relative energy of decode with bypassing.

when setting the clock frequency as required by Table 2 and using reduced connectivity with bottom-up scheduling (RC BU speed optimized).

Results in Figure 21 indicate that the increase in clock frequency required to achieve a shorter execution time to match performance with software bypassing leads to an increase in energy consumption for four of the eight benchmarks, when compared to the fully connected architecture with bypassing synthesized for 250 MHz.

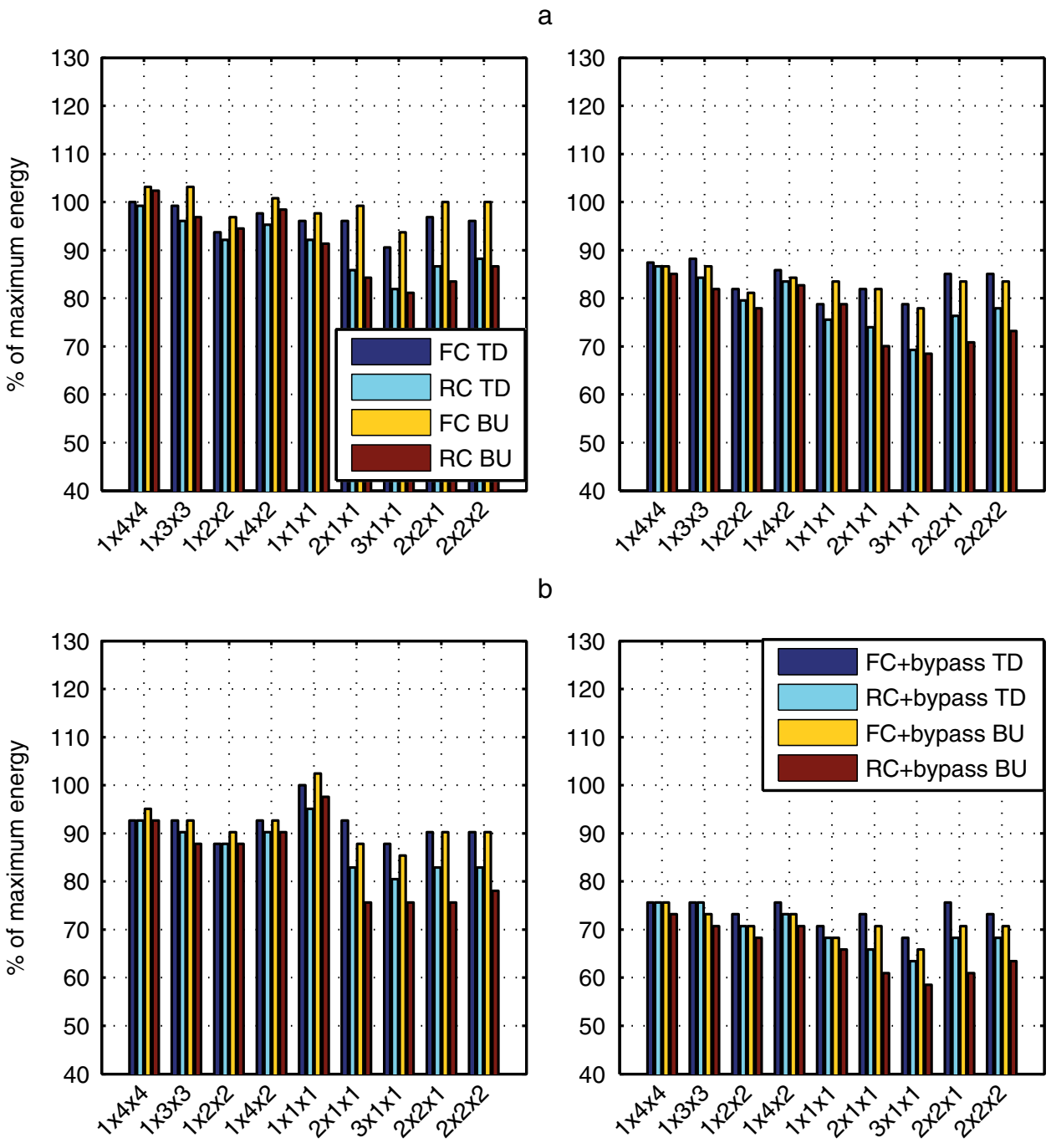

Figure 19 Relative energy consumption of fetch for (a) *mips* and (b) *motion*. On the left side of this figure, we display the relative energy of fetch without bypassing and on the right side, the relative energy of fetch with bypassing.

Exceptions from this trend are *aes*, *jpeg*, *mips*, and *motion* benchmarks. As can be seen from Table 2, the impact of software bypassing for those two benchmarks was relatively limited, and therefore, only a relatively small increase in clock frequency was required.

Compared to the combination of software bypassing and reduced connectivity at 250 MHz, however, synthesizing for higher frequency leads to an increase in energy requirements for all the cases.

Figure 22 shows the same results for register files, interconnection network, decode, and instruction fetch

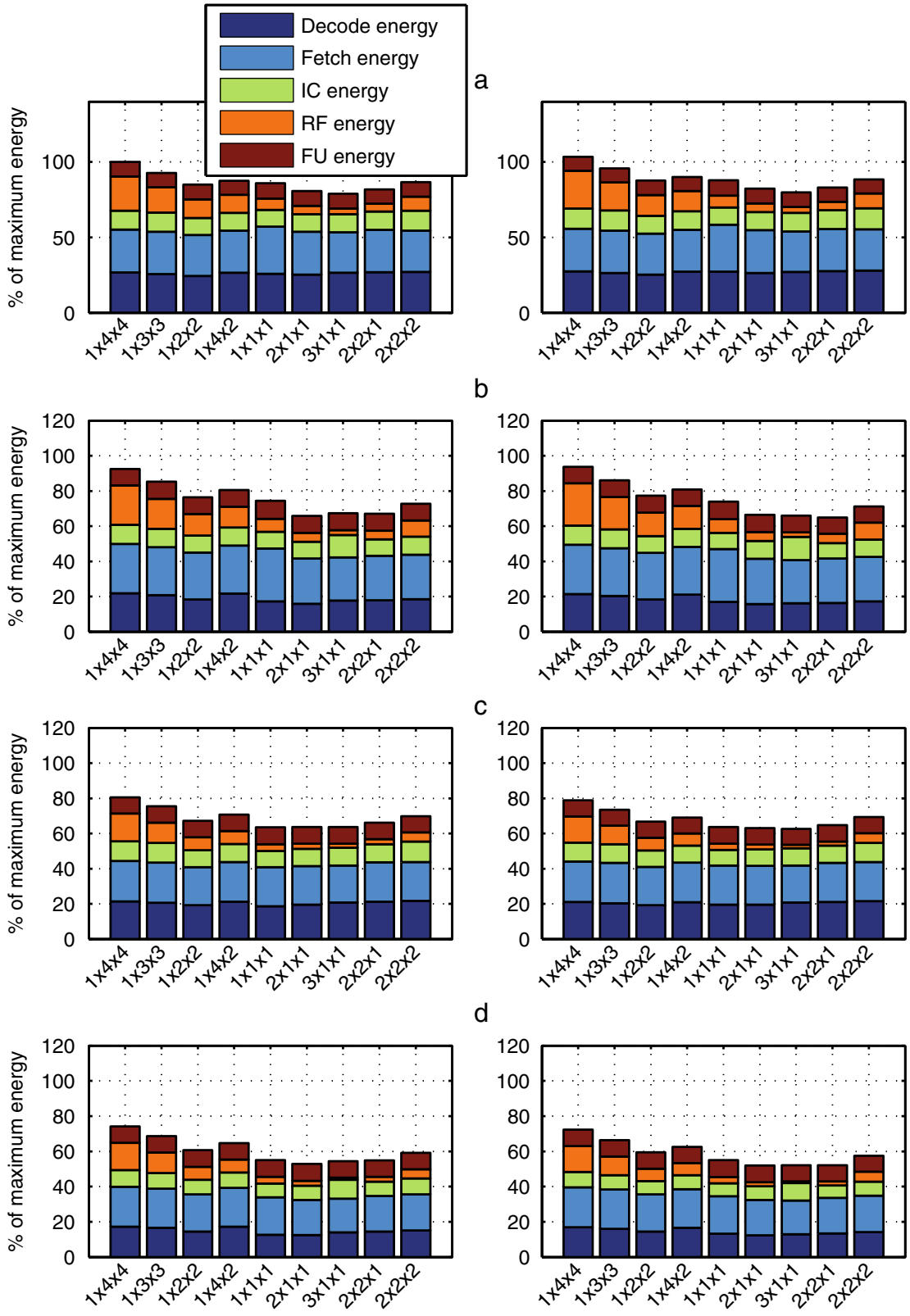

Figure 20 Relative energy consumption of various parts of the processor for each architecture. On the left side, we present results obtained using the top-down scheduler and on the right side, corresponding results using the bottom-up scheduler for **(a)** fully connected without bypassing, **(b)** reduced connectivity without bypassing, **(c)** fully connected with software bypassing, and **(d)** reduced connectivity and bypassing.

Table 2 Clock cycles and timing constraints to match real-time deadline for MS 2 × 1 × 1 architecture with bottom-up scheduling

Benchmark name	Cycles without bypass	Cycles with bypass	Required clock period (ns)	Required frequency (MHz)
adpcm	118,703	81,740	2.6	384
aes	94,471	74,462	3	333
blowfish	1,138,102	760,914	2.6	384
gsm	21,534	16,654	3	333
jpeg	4,343,468	3,355,832	3	333
mips	47,760	39,046	3.2	313
motion	14,217	11,331	3.2	313
sha	859,330	532,325	2.4	416

separately. Here, we can observe that the synthesis for higher clock frequency has the highest impact on the energy of the register files and instruction fetch.

7 Conclusions

In this paper, we evaluated our proposed method on how to improve energy efficiency of processor cores for exposed data path architectures - software bypassing, against a design space exploration technique - connectivity reduction. Our observation shows that both, compiler optimization of software bypassing and architecture optimization of connectivity reduction, lead to a decrease in energy requirements of the processor.

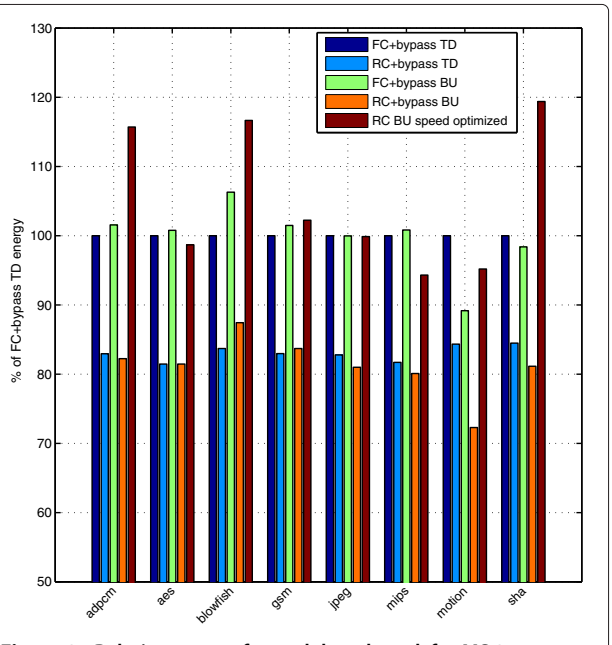

Figure 21 Relative energy for each benchmark for MS 2 × 1 × 1 architecture.

In particular, we observed that for the architecture with several register files, connectivity reduction brings benefits equaling that of software bypassing overall, mainly due to the large number of removed connections and consequent decrease in instruction width. It has, however, no significant effect on the register file energy and does not contribute to performance increase as such. In the case of a single register file, the overall effect of connectivity reduction on energy savings is much smaller than that of software bypassing. It is notable, however, that connectivity reduction allowed for higher clock frequency with architectures with a small number of register file write ports.

Software bypassing, on the other hand, showed a fairly consistent improvements to cycle counts, across all tested architectures. Eventually, even the most limited register file configuration achieved better clock cycle performance than the architectures with a large number of read and write ports, without bypassing.

While software bypassing does not contribute to reduction in instruction width, or reduction in the complexity of the interconnection network, the main benefits of software bypassing come from cycle count improvements and associated energy savings across all components and from register file savings.

We observed that software bypassing provides similar or better energy efficiency to processor customization by reducing connectivity while maintaining the full programmability of the processor.

We also showed that in order to match the performance achievable with software bypassing, architectures with reduced connectivity can be synthesized with higher frequency. However, this results in four of eight benchmarks increasing their energy requirements compared to software bypassing with a fully connected network as well as the loss of the reprogrammability.

In addition, the combination of software bypassing and reduced connectivity is more energy efficient than

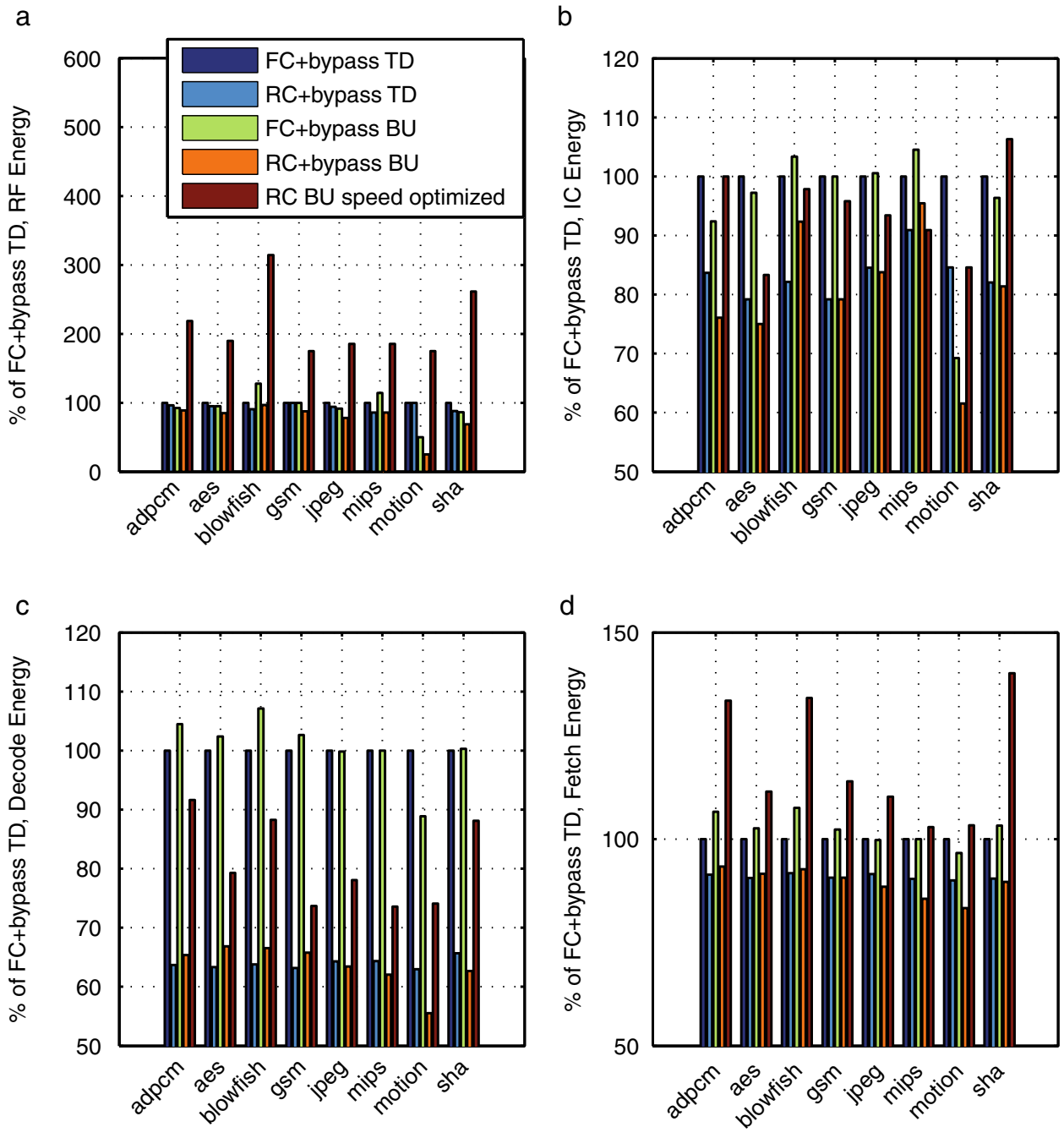

Figure 22 Relative energy of components for each benchmark for MS 2 × 1 × 1 architecture. We present energy comparison relative to fully connected architecture for (**a**) register files, **(b)** interconnection network, **(c)** decode, and **(d)** instruction fetch.

synthesizing for higher frequency for all the benchmarks. Therefore, if the reprogrammability is not an issue, it is still more effective to combine software bypassing with reduced connectivity, combining their respective benefits, than to use only reduced connectivity and synthesize for higher frequency.

Competing interests
The authors declare that they have no competing interests.

Acknowledgements
Part of the work presented in this paper has been financially supported by the Academy of Finland (funding decision 253087) and Radio Laboratory of Nokia Research Center.

Received: 10 September 2012 Accepted: 8 April 2013
Published: 10 May 2013

References

1. MAR Saghir, M El-Majzoub, P Akl, in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGA's*. Datapath and ISA customization for soft VLIW processors (Springer-Verlag Berlin, Heidelberg l' 2007 San Luis Potosi, 20–22 Sept 2006), pp. 1–10
2. N Clark, H Zhong, S Mahlke, in *MICRO-36*. Processor acceleration through automated instruction set customization (San Diego, 3–5 Dec 2003), pp. 129–140
3. V Guzma, P Jääskeläinen, P Kellomäki, J Takala, in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, ed. by M Bereković, N Dimopoulos, and Wong S. Impact of software bypassing on instruction level parallelism and register file traffic. Lecture Notes in Computer Science, vol. 5114 (Springer-Verlag Berlin, Heidelberg l'2008 Heidelberg, 2008), pp. 23–32
4. T Pitkänen, T Rantanen, AGM Cilio, J Takala, in *SAMOS*, ed. by TD Hämäläinen, AD Pimentel, J Takala, and S Vassiliadis. Hardware cost estimation for application-specific processor design. Lecture Notes in Computer Science, vol. 3553 (Springer Berlin Heidelberg, 2005), pp. 212–221
5. S Park, A Shrivastava, N Dutt, A Nicolau, Y Paek, E Earlie, in *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems, LCTES '06*. Bypass aware instruction scheduling for register file power reduction (ACM New York, 2006), pp. 173–181
6. D She, Y He, B Mesman, H Corporaal, in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*. Scheduling for register file energy minimization in explicit datapath architectures (IEEE Computer Society Washington, DC, USA l'2012 , isbn 978-1-4577-2145-8 Dresden, 12–16 Mar 2012), pp. 388–393
7. A Gangwar, M Balakrishnan, A Kumar, Impact of intercluster communication mechanisms on ILP in clustered VLIW architectures. ACM Trans. Des. Autom. Electron. Syst. **12**, 1 (2007)
8. A Terechko, E Le Thenaff, M Garg, J van Eijndhoven, H Corporaal, in *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. Inter-cluster communication models for clustered VLIW processors (IEEE Computer Society Washington, 2003), p. 354
9. DA Patterson, JL Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. (Morgan Kaufmann, San Francisco, 1998), pp. 177, 184–185
10. PG Sassone, DS Wills, in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Dynamic strands: collapsing speculative dependence chains for reducing pipeline communication (IEEE Computer Society Washington, 2004), pp. 7–17
11. D Burger, SW Keckler, KS McKinley, M Dahlin, LK John, C Lin, CR Moore, J Burrill, RG McDonald, W Yoder, the TRIPS Team, Scaling to the end of silicon with EDGE architectures. Computer. **37**(7), 44–55 (2004)
12. PG Sassone, DS Wills, GH Loh, Static strands: safely exposing dependence chains for increasing embedded power efficiency. Trans. on Embedded Computing Sys. **6**(4), 24 (2007)
13. A Bracy, P Prahlad, A Roth, in *MICRO 37: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. Dataflow mini-graphs: amplifying superscalar capacity and bandwidth (IEEE Computer Society Washington, 2004), pp. 18–29
14. J Yan, W Zhang, KD Bosschere, DR Kaeli, P Stenström, DB Whalley, T Ungerer, in *HiPEAC*. Virtual registers: reducing register pressure without enlarging the register file. Lecture Notes in Computer Science, vol 4367 (Springer Berlin, 2007), pp. 57–70
15. H Corporaal, *Microprocessor Architectures: From VLIW to TTA*. (Wiley, Chichester, 1997)
16. Y He, D She, B Mesman, H Corporaal, in *Proceedings of the 11th International Conference on Embedded Computer Systems (SAMOS-XI)*. MOVE-Pro: a low power and high code density TTA architecture (Springer-Verlag Berlin, Heidelberg l'2012 Samos, 18–21 July 2011)
17. G Cichon, P Robelly, H Seidel, M Bronzel, G Fettweis, in *PARELEC '04: Proceedings of the International Conference on Parallel Computing in Electrical Engineering*. Compiler scheduling for STA-processors (IEEE Computer Society Washington, 2004), pp. 45–60
18. M Thuresson, M Själander, M Björk, L Svensson, P Larsson-Edefors, P Stenström, in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. FlexCore: utilizing exposed datapath control for efficient computing (Springer-Verlag Berlin, Heidelberg l'2007 Samos, 16–19 July 2007), pp. 18–25
19. M Reshadi, B Gorjiara, D Gajski, in *Proceedings of the 23rd International Conference on Computer Design*. Utilizing horizontal and vertical parallelism with no-instruction-set compiler for custom datapaths (IEEE Computer Society Washington, DC, USA l'2005 San Jose, 2–5 Oct 2005), pp. 69–74
20. I Finlayson, GR Uh, D Whalley, G Tyson, An overview of static pipelining. Comput. Architecture Lett. **11**, 17–20 (2012)
21. Maxim Corporation: MAXQ Microcontroller home page (2007). [-24pt]http://www.maxim-ic.com/products/microcontrollers/maxq.cfm. Accessed 7 May 2013
22. H Corporaal, HJ Mulder, in *Proceedings of the ACM/IEEE Conference on Supercomputing*. MOVE: a framework for high-performance processor design (ACM New York, NY, USA l'1991, 18–22 Nov 1991), pp. 692–701
23. J Janssen, H Corporaal, in *Proceedings of the 28th Annual Symposium on Microarchitecture (MICRO-28)*. Partitioned register file for TTAs (IEEE Computer Society Press Los Alamitos, CA, USA l'1995 Ann Arbor, 29 Nov–1 Dec 1995), pp. 303–312
24. V Guzma, Pitkänen P, T Kellomäki, J Takala, in *Proceedings of the IEEE Workshop Signal Processing Systems*. Reducing processor energy consumption by compiler optimization (IEEE Computer Society Washington, DC, USA, 2009 Tampere, 7–9 Oct 2009), pp. 63–68
25. J Heikkinen, A Cilio, J Takala, H Corporaal, in *IEEE International Symposium on Circuits and Systems, 2005. ISCAS 2005*. Dictionary-based program compression on transport triggered architectures, vol. 2 (IEEE Computer Society Washington, DC, USA, 2005 Kobe, 23–26 May 2005), pp. 1122–1125
26. J Heikkinen, J Takala, H Corporaal, in *IEEE Workshop on Signal Processing Systems Design and Implementation, 2005*. Dictionary-based program compression on TTAs: effects on area and power consumption (IEEE Computer Society Washington, DC, USA, 2005 Athens, 2–4 Nov 2005), pp. 479–484
27. AV Aho, MS Lam, R Sethi, JD Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd edn. (Addison Wesley, Boston, 2006), pp. 721–726
28. P Kellomäki, V Guzma, J Takala, Safe pre-pass software bypassing for transport triggered processors. Acta Technica Napocensis. **49**(3), 5–10 (2008)
29. P Jääskeläinen, V Guzma, A Cilio, J Takala, in *Proceedings of the SPIE Multimedia on Mobile Devices*. Codesign toolset for application-specific instruction-set processors ('SPIE' society San Jose, 28 Jan 2007), pp. 65070X–1–65070X–11
30. TCE: TTA-based co-design environment (2002). http://tce.cs.tut.fi. Accessed 7 May 2013
31. Y Hara, H Tomiyama, S Honda, H Takada, Proposal and quantitative analysis of the CHStone Benchmark program suite for practical C-based high-level synthesis. J. Inf. Process. **17**, 242–254 (2009)

doi:10.1186/1687-3963-2013-9
Cite this article as: Guzma *et al.*: **Use of compiler optimization of software bypassing as a method to improve energy efficiency of exposed data path architectures.** *EURASIP Journal on Embedded Systems* 2013 **2013**:9.