A Flexible Datapath Interconnect for Embedded Applications

Magnus Själander, Per Larsson-Edefors, and Magnus Björk
Department of Computer Science and Engineering
Chalmers University of Technology, SE-412 96 Göteborg, Sweden

Abstract— **We investigate the effects of introducing a flexible interconnect into an exposed datapath. We define an exposed datapath as a traditional GPP datapath that has its normal control removed, leading to the exposure of a wide control word. For an FFT benchmark, the introduction of a flexible interconnect reduces the total execution time by 16%. Compared to a traditional GPP, the execution time for an exposed datapath using a flexible interconnect is 32% shorter whereas the energy dissipation is 29% lower. Our investigation is based on a cycle-accurate architectural simulator and figures on delay, power, and area are obtained from placed-and-routed layouts in a commercial 0.13-μm technology. The results from our case studies indicate that by utilizing a flexible interconnect, significant performance gains can be achieved for generic applications.**

I. Introduction

Previous studies [1], [2] have shown that by exposing the datapath, through the use of a wide control word, significant performance gains can be obtained. With an exposed datapath the compiler/programmer gets total control of the resources of the datapath for each executed cycle. This may allow for a more efficient utilization of datapath resources when scheduling an application's operations onto the datapath, since the compiler is not limited to the fixed control given by an Instruction Set Architecture (ISA). The previous studies focused on hardware/software co-design in that a datapath was crafted to a specific application, or possibly to an application domain. The performance for an application can indeed be drastically improved by adapting the datapath to the specific needs of resources and communication paths. However, adapting the datapath to a particular application's needs will most likely hamper the datapath's performance when used for other applications.

A study on a Discrete Cosine Transform (DCT) application showed a 20% reduction in execution time, when the datapath of a MIPS processor became exposed [1], [2]. The DCT study indicates that the performance of a General Purpose Processor (GPP), whose datapath and control traditionally are co-optimized for general-purpose processing, can benefit from exposing the datapath. However, when introducing the concept of an exposed datapath, it may be wise to reconsider what datapath components and interconnects that we use. After all, the datapath of a traditional GPP has been carefully optimized together with its ISA. Thus, it may be possible to further improve the general-purpose processing efficiency by adapting the datapath to the use of an exposed, wide control word.

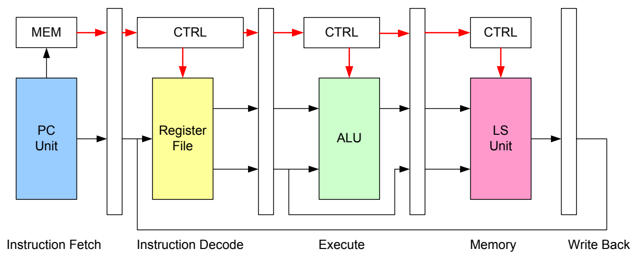
In this study, the overall purpose is to ascertain if a flexible interconnect, in combination with the exposed datapath, can offer performance gains for *both general-purpose and dedicated processing*. The insertion of a flexible interconnect allows data to be efficiently routed between datapath units, thus, potentially improve the utilization of these units, beyond what was possible by the optimized ISA. We will later show that the execution time indeed can be reduced by introducing a flexible interconnect into an exposed GPP datapath.

The interconnect evaluation is conducted on an experimental platform called FlexCore, which has been developed within the FlexSoC project [3]. The FlexCore platform makes it possible to model different architectures having the wide control word of an exposed datapath. Each modeled architecture is used to evaluate application performance and to generate a Hardware-Description Language (HDL) representation of the architecture. To obtain values on delay, power, and area, each generated HDL representation is taken through a conventional ASIC backend tool flow for a 0.13-μm technology.

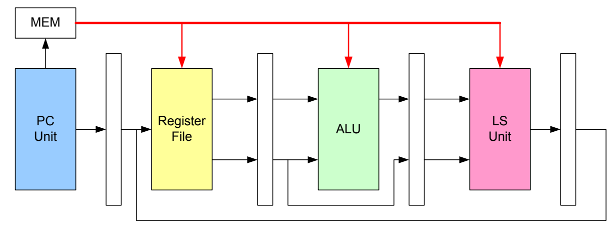
This paper is organized as follows: The architectural and experimental framework is presented in Section II and III. Section IV presents the FFT case study that has been conducted. Section V presents the results from the case study, followed by a discussion in Section VI. The paper is finally concluded in Section VII.

II. Architectural Framework

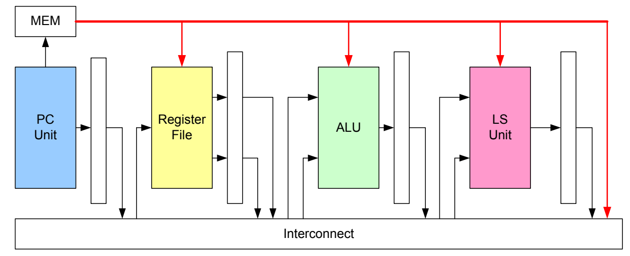
To evaluate what is the impact on performance when introducing a flexible interconnect into an exposed general-purpose datapath, three different architectures have been compared against each other. The compared architectures are illustrated in Figure 1.



(a) GPP Datapath



(b) Exposed Datapath



(c) Exposed Datapath with Flexible Interconnect

Fig. 1. Illustrations of the three architectures that have been compared against each other in this study.

The first architecture is that of a traditional five-stage GPP pipeline, as shown in Figure 1(a). In a traditional GPP, the movements of data are restricted to those defined by the Instruction Set Architecture (ISA) and the GPP's forwarding paths.

The second architecture is that of a GPP pipeline with an exposed datapath, Figure 1(b). Here the programmer/compiler has total control of the whole datapath for each executed cycle. This makes the control more fine grained and gives the compiler greater freedom to move data between the datapath units, whose communication options are no longer dictated by an ISA.

To take advantage of a flexible interconnect, the control word needs to be wider. A traditional GPP would not benefit from a flexible interconnect, since its rigid (ISA-defined) control limits the communication paths that could have been used in the datapath. Thus, the final architecture that we will evaluate is that of an exposed datapath that has a fully connected crossbar switch as interconnect. Here all inputs and outputs of the datapath units are connected to the interconnect switch, as shown in Figure 1(c).

A fully connected interconnect can appear to be an overkill in terms of flexibility. Still, we use this as a reference point for the maximum execution-cycle performance improvement that can be achieved, when there are no restrictions on how data can move between the datapath units. A more realistic interconnect is when some of the communications paths are removed; those with no or negligible impact on application execution-cycle performance. This pruning of communication paths reduces delay, power, and area for the interconnect.

We will in the following sections compare the conventional GPP in Figure 1(a) to *i)* the exposed datapath in Figure 1(b) and to *ii)* the exposed datapath with a flexible interconnect in Figure 1(c). This comparison will enable us to distinguish what performance gains are the result of exposing the datapath and what performance gains are the result of the flexible interconnect.

A. *Modeling of the Architectures*

We represent the three different architectures by using an experimental platform called FlexCore [4], [5], which has been developed within the FlexSoC project. The baseline FlexCore is inspired by a conventional single-issue, five-stage processor, but its datapath is fully exposed through a 91-bit wide control word (Figure 2). The baseline FlexCore consists of four datapath units: a register file, an Arithmetic Logic Unit (ALU), a Load/Store Unit (LS Unit), and a Program Control Unit (PC Unit). All these datapath units are attached to a fully connected interconnect of crossbar type. To allow for GPP functionality each datapath unit's output port is connected to a data register, which acts as pipeline register in the various pipeline configurations that can be assembled using the interconnect. To allow instructions to be scheduled on the FlexCore in the same way as on a conventional five-stage pipeline, two data registers are included (RegA and RegB in Figure 2). These registers are traditionally used in the execute and load/store stage of a five-stage GPP to allow data to bypass the ALU and LS Unit.

The three different architectures that are to be compared can be viewed as subsets of the FlexCore architecture. Actually, the third architecture, in which the datapath is exposed and has a flexible interconnect, is identical to FlexCore. We can represent the other two architectures simply by the way instructions are scheduled and by limiting the communication paths to those available for the architectures.

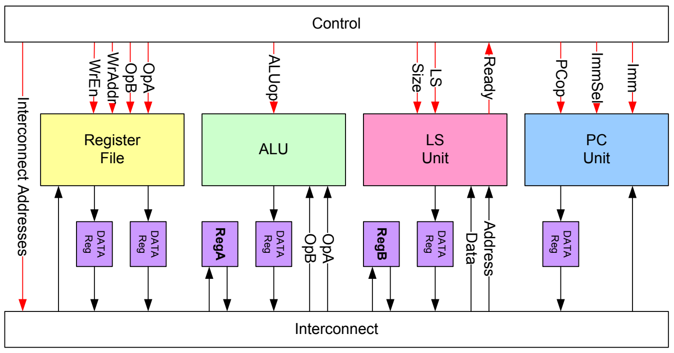


Fig. 2. Illustration of a baseline FlexCore. Note that each DATA Reg, RegA, and RegB also has an enable signal (not shown in the figure).

A compiler has been created for the FlexCore architecture. This takes GPP assembly code as input and schedules it as if the underlying architecture was a five-stage pipeline. The instruction scheduling of GPP assembly code onto the FlexCore architecture achieves an execution-cycle performance comparable to that of a traditional GPP.

By utilizing the datapath units more efficiently through the exposed control word, which can be viewed as a very expressive instruction, the instructions can be scheduled within fewer cycles. We model the exposed datapath by manually scheduling the instructions of an application for maximum datapath-unit utilization, while obeying the constraint of the rigid communication paths of the five-stage GPP datapath. Two examples of efficient scheduling are *i)* when we choose not to write temporary values to the register file and *ii)* when pipeline resources permit scheduling of instructions in parallel.

Similar to the exposed datapath, we model the exposed datapath with a flexible interconnect by using manual scheduling of the same part of the application. However, now we have no restrictions on the communication paths: Any input to a datapath unit can read its value from any output of all the datapath units. This corresponds to a fully connected crossbar switch as interconnect, with all inputs and outputs of the datapath units connected to it.

III. Experimental Framework

A set of tools, consisting of a compiler, a cycle-accurate simulator, and a Hardware-Description Language (HDL) generator, has been created for the FlexCore architecture. These three tools are used in our study on flexible interconnects and are consequently presented briefly in the following:

A. *Compiler*

The current version of the FlexCore compiler takes MIPS assembly code as input and generates a set of so-called Native ISA (N-ISA) instructions, which correspond to the instructions for the exposed datapath. The N-ISA instructions can be used for simulation, either by using the FlexCore simulator or by using standard HDL simulators with the HDL code obtained from the FlexCore HDL generator.

The compiler assumes a logical five-stage pipeline for scheduling the MIPS instructions onto the FlexCore architecture. For example, a fetched ADD instruction is translated into four partial N-ISA instructions. A partial N-ISA instruction consists only of the control signals needed for a particular instruction in a particular pipeline stage. The four partial N-ISA instructions for an ADD consist of: *i)* the control signals for reading the operands from the register file, *ii)* the control signals for the ALU to take the operands from the register file and perform an ADD operation, *iii)* the control signals to write the result from the ALU to one of the buffers, in order to bypass the load/store unit, and *iv)* the control signals for writing the result back to the register file.

The compiler performs static scheduling of the partial N-ISA instructions from several MIPS instructions, such that they overlap as much as possible, without causing conflicts between the control signals of different partial N-ISA instructions. When the partial N-ISAs are statically scheduled, the operands might not be available in the register file when they are to be read. The compiler therefore has to keep track of which values in the register file are valid. If a partial N-ISA instruction is requesting an invalid value from the register file, the compiler tries to either forward the value from the datapath or that particular instruction has to be delayed until the requested value is available.

The compiler does not have support for forwarding between N-ISA instructions from different basic blocks. This can cause a minor performance penalty compared to a conventional GPP, which uses dynamic forwarding.

It is currently only possible to take advantage of the fine-grained control of the exposed FlexCore datapath by doing manual scheduling of the operations to be performed. The FlexCore compiler therefore supports inline N-ISA instruction in the MIPS assembly code. The inline N-ISA instruction is written as Register Transfer Notations (RTNs) that are subsequently translated to N-ISA instructions by the compiler.

B. Cycle-Accurate Simulator

The simulator is a cycle-accurate model of a FlexCore implementation. The simulator takes the generated N-ISA code of the compiler as input and executes the application until it exits. The simulator has support for profiling of executed instructions as well as gathering statistics on resource utilization of the different FlexCore datapath units.

C. HDL Generator

The HDL generator allows a user to specify what datapath units a particular FlexCore implementation consists of and how they are connected to the interconnect. The generator has support for specifying which ports in the interconnect that should be connected with each other; specifically which input ports that are connected to a specific output port.

The generated HDL code can be taken through a conventional ASIC backend flow, in order to get accurate estimates on area, delay, and power. The generated HDL code can also be used in conventional HDL simulators, together with the instruction code from the compiler, to simulate the behavior of a FlexCore. Such simulations are routinely used to confirm the results from the cycle-accurate simulator.

IV. Interconnect Evaluation

Performance comparisons of the different architectures have been conducted by running the Fast Fourier Transform (FFT) benchmark from the Embedded Microprocessor Benchmark Consortium (EEMBC) [6] on each architecture using the FlexCore cycle-accurate simulator. The FFT benchmark is part of EEMBC's TeleBench suite and implements a decimation-in-time 256-point 16-bit FFT using a butterfly technique. To get estimates on delay, power, and area, each architecture has been generated using the FlexCore HDL generator and subsequently taken through synthesis and place-and-route.

A. FFT Application Scheduling

The FFT application was chosen for this study because of its reasonable size. This makes it possible to manually schedule a significant portion of the algorithm onto the investigated architectures, thus, allowing us to study and compare the performance gains for exactly the same software application.

The manual scheduling was done by compiling the C-code for the FFT application to MIPS assembly. The MIPS assembly was subsequently fed to the FlexCore compiler, which generated the final N-ISA instructions. The generated code was profiled with the help of the simulator and the FFT application's computational kernels were identified. The FFT application starts with a setup phase, in which data and twiddle factors are generated, before the actual FFT computation is carried out. The setup is done once, after which it is possible to execute multiple FFT computations.

The FFT computation has one clearly identifiable computational kernel, consisting of three nested loops. More than 73% of the total execution time is spent in the inner-most loop, when 100 FFT computations are carried out. This particular code segment has therefore been the target for manual scheduling. Not surprisingly, the code of the inner-most loop corresponds to the computations of a single FFT butterfly.

The manual scheduling was performed without any algorithmic changes to the FFT application. The MIPS assembly of the inner-most loop was simply exchanged with inline RTN code. In the scheduling procedure we took each MIPS instruction and scheduled it as early as possible onto the given datapath. If there exist no resource conflicts with already scheduled instructions, we might schedule the instruction that we currently consider *earlier* than what the MIPS assembly would suggest. The instruction order might therefore have changed. Unnecessary utilization of resources is avoided by, for example, not writing back temporary values to the register file.

To fully take advantage of the exposed datapath for general applications, there needs to be support by the compiler for efficient scheduling of computations onto the datapath. This is addressed not only in the FlexSoC project, but also in other research projects, such as that by Reshadi *et al* [7], even though their compiler targets co-design.

B. Hardware Implementation

The FlexCore HDL generator has been used to generate the architectures by specifying the communication paths that are available in the different datapaths. For this particular study, the baseline FlexCore has been extended with a 32-bit multiplier, which is pipelined in two stages. The two inputs to the multiplier are directly connected to the interconnect. Between the output of the multiplier and the interconnect we have inserted two 32-bit registers, where one holds the 32 most significant bits and the other holds the 32 least significant bits of the result.

In neither of the three different architectures, control logic and instruction fetch power were accounted for. By focusing on the datapath, the impact of the interconnect can be separated from the intrinsic gains of exposing the datapath, to avoid observing differences in delay and power that are due to different ways of handling the control or requirements on memory bandwidth. What implications various types of control have on the overall performance is an issue which is being addressed within the FlexSoC project.

When we discard the control logic and instruction fetch circuitry of the GPP datapath in Figure 1(a), this datapath is exactly the same as that of the exposed datapath. Figure 3 shows a detailed schematic of the communication paths that are available to the GPP and the

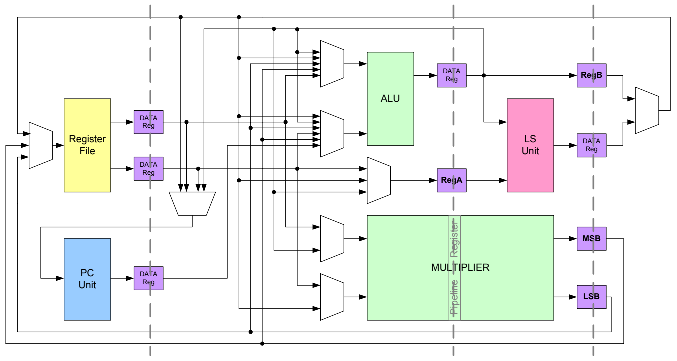


Fig. 3. Schematic of the datapath for the GPP and exposed datapath. Only the communication paths for data are shown, without any control signals.

exposed datapath. The datapath is inspired by the DLX and MIPS R2000 datapaths [8]. The multiplier is an intrinsic part of the datapath and takes its inputs from the register file. To improve performance, it is possible to forward results from the ALU and LS Unit directly. The output registers of the multiplier do not work as traditional pipeline registers, where a new value is clocked in for each new clock cycle. Instead these registers function such that they hold the result from the previous multiplication until a new multiplication is performed. This is to support the semantics of a GPP which uses move instructions to move data from the output registers to the register file.

The datapath which uses the flexible interconnect was generated by specifying the communication paths, such that an input port to a functional unit can be connected to any output port of any functional unit. This creates a fully connected crossbar switch as interconnect.

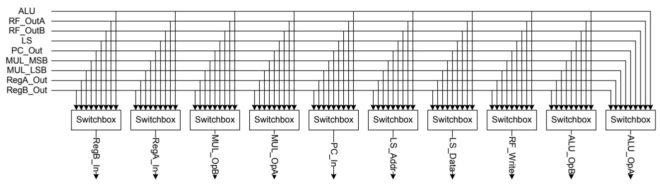


Fig. 4. Illustration of the crossbar switch of the fully connected interconnect.

Figure 4 illustrates how the crossbar switch is constructed out of a number of switchboxes. Each input to a datapath unit (in Figure 3) as well as to the two registers RegA and RegB are connected to its own switchbox. Each switchbox is in turn connected to all the outputs of the datapath units. The switchbox functions as a multiplexer, according to Figure 5.

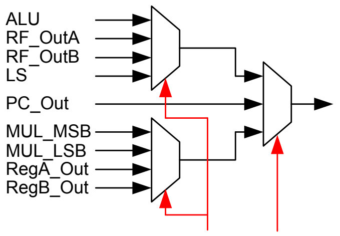


Fig. 5. Illustration of a switchbox.

The generated designs were taken through a synthesis and place-and-route flow [9], [10] for a commercial 0.13-μm technology. Delay and power estimates were obtained from resistance and capacitance extracted placed-and-routed designs.

V. Results

The three architectures were simulated with the FlexCore simulator running an application in the form of 100 consecutive FFT computations. Table I summarizes the number of executed cycles for the whole application and the manually scheduled inner-most loop, respectively, for each architecture. Table I shows that by deploying only the concept of exposed datapath, the inner-loop cycle count was reduced with as much as 30% compared to a traditional five-stage GPP. But by exploiting a fully connected interconnect, we show that the inner-loop cycle count can be improved with *a further 20%* compared to only utilizing the benefits of an exposed datapath.

We also observe that for the FFT application as a whole, the cycle count reductions are not as dramatic. This is because here almost 30% of the executed instructions have not been manually scheduled and therefore do not benefit from neither the exposed datapath nor the flexible interconnect.

TABLE I
Simulated Application Cycle Count

Datapath	Whole Application	Inner Loop
GPP	5 877 569 (100%)	4 300 800 (100%)
Exposed Datapath	4 750 769 (81%)	3 072 000 (71%)
Fully Connected	3 803 669 (65%)	2 150 400 (50%)

The delay, power, and area estimates of the different architectures are shown in Table II. Since the control has been excluded for all architectures, the estimates for the GPP and exposed datapath are the same. The fully connected interconnect increases the delay by 11% and the area by 15%. This is a reasonable overhead for the flexible interconnect, considering the application cycle count speedups. The power on the other hand is 23% higher than for the GPP and the exposed datapath, which indicates a high overhead for the interconnect.

TABLE II
Delay, Power, and Area Estimates

Datapath	Delay (ns)	Power (mW)	Area (mm^2)
GPP	2.25 (100%)	6.57 (100%)	0.35 (100%)
Exposed Datapath	2.25 (100%)	6.57 (100%)	0.35 (100%)
FlexCore	2.49 (111%)	8.07 (123%)	0.40 (115%)
Tailored	2.36 (105%)	6.83 (104%)	0.37 (106%)

Due to the high power dissipation for the fully connected interconnect, an interconnect was tailored for the FFT application. We designed this tailored interconnect by first simulating the architecture with a fully connected interconnect, using the FFT application and its manual scheduling. In a subsequent phase we extracted statistics of which communication paths that were utilized during the simulation. We used the statistics to restrict the communication paths by pruning the interconnect in the FlexCore generator. In order to not lose any general-purpose processing efficiency, we made sure that none of the communication paths that exist in the GPP datapath were removed from the interconnect. Like for the other three architectures, the generated HDL code was taken through an ASIC backend flow.

With the tailored interconnect the power overhead is reduced from 23% to only 4% and the area overhead is reduced from 15% to 6% compared to the GPP. The delay for the tailored interconnect is also reduced from 11% to 5%.

Flexible interconnects appear to be at a slight disadvantage, in terms of delay and power dissipation. However, total application performance in terms of execution time (cycle count × clock period)

TABLE III

INTERCONNECT PATHS USED BY THE FFT APPLICATION AND THE TAILORED INTERCONNECT

	PC In	RF Write	ALU OpA	ALU OpB	LS Address	LS Data	RegA In	RegB In	MULT OpA	MULT OpB
PC Out				FFT						
RF ReadA	FFT	FFT	FFT						FFT	
RF ReadB	FFT			FFT	FFT	FFT	FFT	FFT		FFT
ALU Out	FFT	FFT	FFT	FFT	FFT	FFT	FFT	FFT	FFT	
LS Out	FFT	FFT	FFT	FFT			FFT		FFT	FFT
RegA Out			FFT	FFT	FFT	FFT				FFT
RegB Out	FFT	FFT	FFT	FFT	FFT		FFT			
MULT LSB		FFT	FFT	GPP			FFT			
MULT MSB		GPP	GPP	GPP						

and energy dissipation (execution time × power dissipation) should be considered next to delay and power. The FFT application runs 28% faster on the datapath that uses a fully connected interconnect, as compared to that of a traditional GPP. Thanks to its shorter clock period, the tailored interconnect performs even better, with a 32% reduction of the execution time (Table IV).

TABLE IV

FFT APPLICATION EXECUTION TIME AND ENERGY DISSIPATION

Datapath	Execution Time (ms)	Energy Dissipation (μJ)
GPP	13.2 (100%)	86.7 (100%)
Exposed Datapath	10.7 (81%)	70.3 (81%)
Fully Connected	9.5 (72%)	76.7 (88%)
Tailored	9.0 (68%)	61.5 (71%)

When considering energy dissipation for executing the FFT application the architecture with the fully-connected, flexible interconnect performs 12% better than that of a traditional GPP, even though the flexible interconnect dissipates 23% more power. The tailored interconnect is not only the fastest solution but also the most energy efficient, with a 29% energy reduction compared to a conventional GPP.

VI. DISCUSSION

The results in the previous section clearly show that the rigid interconnect of a GPP datapath restricts the performance potential of an exposed datapath. On the other hand, a fully connected interconnect has high power dissipation. It is therefore interesting to further discuss if it is possible to create an interconnect that allows the full potential of exposed control to be exploited without leading to the somewhat high power dissipation of a fully connected interconnect.

The tailored interconnect for the FFT application shows that it is possible to reduce the number of communication paths without any degradation in execution-cycle performance. We will here give a short analysis of *i)* what paths helped to increase performance and *ii)* what paths in the fully connected interconnect that have not contributed to any performance improvement for the FFT application.

When we manually scheduled operations onto the exposed datapath with the rigid interconnect, some architectural shortcomings became evident. A major shortcoming is that when making a load or store to the memory the address has to come from the ALU, even though the address might already exist somewhere else in the pipeline. Similarly the data for the store instruction can only be read from RegA, Figure 3. This leads to a waste of resources that could have been used for other operations.

In a more flexible interconnect, data and address to the load/store unit can be read from several datapath units. Further, with a flexible interconnect RegA and RegB can be used for temporary storage. Such a temporary storage is useful in order to distribute write requests to the register file over time, since writes to the register file can be delayed until the write port becomes available. It can also be used to eliminate writes to the register file, if the data exists for short time and is to be consumed by one of the datapath units. This is not possible in the GPP datapath, because RegA and RegB get contaminated by ALU and store operations.

The improvements to the interconnect that are listed above are generic, and their effect on the execution of the FFT application can be observed also for the tailored, but still flexible interconnect. Table III shows the communication paths used by the FFT application, when it has been scheduled for a fully connected interconnect. The table notation is as follows: A FFT in the matrix indicates that the communication path between the output port on the left hand side and the input port stated at the top is utilized by the benchmark. The paths marked by FFT, together with those marked with GPP, are those of the tailored interconnect. Table V lists those communication paths that are not available in the traditional GPP datapath. Here we see that, in comparison to the GPP datapath, there exist more paths to the LS Unit, and to and from RegA and RegB. There are also more direct paths to and from the register file, which is a consequence of the fact that data for store instructions are not moved through RegA and that ALU results are not being moved through RegB.

TABLE V

NON-GPP INTERCONNECT PATHS IN THE TAILORED INTERCONNECT

From	To
Register File	Register File
Register File	LS Address
Register File	LS Data
Register File	RegB
ALU	Register File
ALU	LS Data
RegA	ALU
RegA	LS Address
RegA	MULT
RegB	LS Address
Multiplier	RegA

By looking at what paths that can be pruned away from a fully connected interconnect, we directly realize that some of the communication paths contribute very little to increase the performance. For example, there is little need for a communication path from the LS Unit's output to its *data* input. The only time this path would be used is if data is to be moved from one memory location to another[1]. If this operation would be necessary it is still possible to use, for example, RegA to temporarily store the data, and then forward it to the LS

[1]Movement of larger amounts of data would be best served by a DMA unit, if available.

data input port. There are other paths we can prune away; we can restrict the connections between datapath units with several output ports and datapath units with several input ports. An example of this would be to connect the register file with the ALU in the same way as for the GPP datapath in Figure 3. Here only one output port of the register file is connected to each input of the ALU. All the listed path restrictions to a fully connected interconnect have no or negligible impact on the execution-cycle performance of an application.

Considering the FFT application we see that the output of the LS Unit is never used by the LS Unit itself. The possibility of only connecting one of the output ports of a datapath unit to one of the input ports of another datapath unit that has several input and output ports has also been exploited. Further, we see that the PC output port, which is used for immediate values and for storing the program counter on a jump-and-link instruction, is only connected to the ALU. In a more general application, a communication path to the register file would be suitable for easy storage of the program counter.

The tailored interconnect for the FFT application addresses the major communication-path drawbacks of the rigid GPP interconnect. It should be noted that it does not limit the communication paths in a way that would hamper the cycle-count performance for other applications. Minor improvements could be made to the tailored interconnect, such as adding a communication path between the PC's output port and the register file.

To evaluate the generality of the tailored interconnect, we manually scheduled parts of the Autocorrelation benchmark (here called AutoBench) from EEMBC's TeleBench suit. The execution cycle-time improvements for the exposed datapath and the datapath with a fully connected interconnect are similar to those for the FFT application, see Table VI. The exposed datapath handles load and store instructions poorly, since the ALU is always utilized for these operations. This can be clearly observed by the low performance improvements for the exposed datapath. A flexible interconnect allows the ALU to be used for useful operations in parallel with load instructions in AutoBench. This can be noted by the significant performance improvements for the fully connected datapath.

TABLE VI
CYCLE COUNT, EXECUTION TIME AND ENERGY ESTIMATIONS FOR AUTOBENCH

Datapath	Cycle Count	Exec. Time (μs)	Energy (μJ)
GPP	168k (100%)	378 (100%)	2.5 (100%)
Exposed Datapath	150k (89%)	338 (89%)	2.2 (88%)
Fully Connected	118k (70%)	294 (78%)	2.4 (96%)
Tailored	118k (70%)	278 (74%)	1.9 (76%)

Table VII shows the interconnect paths used by AutoBench that are not part of a traditional GPP. The number of added paths are not as many as for the FFT benchmark, but what is important is that the added paths are *a subset of the interconnect that was tailored for the FFT application*. This allows AutoBench to be as efficiently scheduled on the tailored interconnect as on the fully connected interconnect.

The tailored interconnect exhibits significant execution time and energy improvements (Table IV and VI) and has shown that it can cater for the needs of the FFT and Autocorrelation benchmarks. We believe that these improvements are likely to apply also for other embedded applications, as argued for in this section.

TABLE VII
NON-GPP INTERCONNECT PATHS USED BY AUTOCORRELATION

From	To
Register File	Register File
Register File	LS Address
ALU	Register File
RegA	MULT

VII. CONCLUSION

The introduction of a tailored interconnect to an exposed GPP datapath has been shown to reduce the total execution time for an FFT and an autocorrelation application with 26-32%. Further, the tailored interconnect also reduce the total energy dissipation for the two applications with 24-29%. The tailored interconnect has been evaluated with only two applications, but the construction of the interconnect has been done such that it does not hamper the performance of general-purpose processing. In fact, the tailored interconnect should give an improved performance not only for dedicated applications, but also for most general-purpose applications.

VIII. ACKNOWLEDGMENTS

We thank our FlexSoC colleagues (Hughes, Jeppson, Sheeran, Svensson, Stenstrom, and Thuresson) whose work has made it possible to make this study. A special thanks goes to Martin Thuresson for his fruitful comments on the structure of this article and its illustrations. The FlexSoC project is sponsored by the Swedish Foundation for Strategic Research.

REFERENCES

[1] B. Gorjiara and D. Gajski, "Custom Processor Design Using NISC: a Case-Study on DCT Algorithm," in *Workshop on Embedded Systems for Real-Time Multimedia*, September 2005, pp. 55–60.
[2] B. Gorjiara, M. Reshadi, and D. Gajski, "Designing a Custom Architecture for DCT Using NISC Design Flow," in *Asia and South Pacific Conference on Design Automation*, 2006, pp. 116–117.
[3] J. Hughes, K. Jeppson, P. Larsson-Edefors, M. Sheeran, P. Stenström, and L. J. Svensson, "FlexSoC: Combining Flexibility and Efficiency in SoC Designs," in *Proceedings of the IEEE NorChip Conference*, 2003.
[4] M. Björk, M. Själander, L. Svensson, M. Thuresson, J. Hughes, K. Jeppson, J. Karlsson, P. Larsson-Edefors, M. Sheeran, and P. Stenstrom, "Exposed Datapath for Efficient Computing," Department of Computer Science and Engineering, Chalmers University of Technology, Tech. Rep. 2006-20, December 2006.
[5] ——, "Exposed Datapath for Efficient Computing," in *HiPEAC Workshop on Reconfigurable Computing*, January 2007.
[6] "Embedded Microprocessor Benchmark Consortium," www.eembc.org.
[7] M. Reshadi, B. Gorjiara, and D. Gajski, "Utilizing Horizontal and Vertical Parallelism with No-Instruction-Set Compiler for Custom Datapaths," in *International Conference on Computer Design (ICCD)*, October 2005.
[8] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, 2nd ed. Morgan Kaufman Publishers Inc., 1998.
[9] *Design Compiler User Guide Version W-2004.12*.
[10] *Encounter User Guide Version 4.1*.