FlexCore: Utilizing Exposed Datapath Control for Efficient Computing

Martin Thuresson, Magnus Själander, Magnus Björk, Lars Svensson, Per Larsson-Edefors, and Per Stenstrom
Chalmers University of Technology

Abstract— We introduce FlexCore, the first exemplar of an architecture based on the FlexSoC framework. Comprising the same datapath units found in a conventional five-stage pipeline, the FlexCore has an exposed datapath control and a flexible interconnect to allow the datapath to be dynamically reconfigured as a consequence of code generation. Additionally, the FlexCore allows specialized datapath units to be inserted and utilized within the same architecture and compilation framework.

This study shows that, in comparison to a conventional five-stage general-purpose processor, the FlexCore is up to 40% more efficient in terms of cycle count on a set of benchmarks from the embedded application domain. We show that both the fine-grained control and the flexible interconnect contribute to the speedup. Furthermore, according to our VLSI implementation study, the FlexCore architecture offers both time and energy savings.

The exposed FlexCore datapath requires a wide control word. The conducted evaluation confirms that this increases the instruction bandwidth and memory footprint. This calls for efficient instruction decoding as proposed in the FlexSoC framework.

I. INTRODUCTION

Cost- and performance-sensitive application areas, such as cellular phones and other battery-powered multimedia devices, are not well served by present-day general-purpose computing platforms. To meet user expectations of features and battery capacity, designers instead resort to highly heterogeneous systems where a collection of specialized hardware accelerators (built for encryption, image and video coding, audio playback, etc) are controlled by an embedded microprocessor, such as an ARM core. For cost reasons, several accelerators will typically be collocated with the microprocessor on a single System-on-Chip (SoC). For typical embedded systems, memory is expensive, both in terms of money, as well as power. Because of the tight power and cost constraints together with the high volumes used, it is important to keep the memory usage and I/O activity down.

The present practice indeed has drawbacks. Tasks outside the set originally intended may not benefit from the computing capacity available: computing resources hidden inside an accelerator may be difficult or impossible to use in ways other than those considered by the accelerator designer. Even when possible, the software constructs necessary to access a "hidden" hardware block bear little resemblance to ordinary code.

For ease of software development and maintainability, a uniform hardware/software interface, similar to those offered by general-purpose processors (GPPs), would be highly desirable; but present-day GPPs cannot compete with heterogeneous SoCs in terms of performance at a given power level. Merging the accelerator datapath elements into the GPP infrastructure would be possible in principle, but a very wide instruction word would be required to make fine-grained control possible.

In our approach, called FlexSoC [1], we address these problems by moving away from the conventional GPP instruction set architecture (ISA) and use a wide control word. This allows us to control the datapath units in the datapath at a much more fine-grained level. Other important differences between our approach and the standard GPP-like ISA is that each control word, or Native-ISA (N-ISA), controls the datapath in the current cycle only and that forwarding needs to be done statically. Previous work on exposed control has shown significant performance improvements [2]. While they use hardware/software co-design, the FlexSoC approach envision a (extendable) pipeline together with a flexible instruction decoder.

To better understand the requirements of the instruction decoder, our study investigates how the instruction bandwidth as well as the static code size is affected by the exposed control. In particular, we note that even though the wide instructions allow for more efficient control, the number of instructions needed are still large, thus increasing the static code size considerably compared to a conventional GPP-version. In fact, all benchmarks are larger and require three times as much bandwidth as a GPP-version.

Consequently, in the FlexSoC framework we propose to use a programmable instruction decoder. This decoder allows the compiler to use a compressed Application-Specific ISA (AS-ISA) for each application, or set of applications, to be executed. Applications are stored as AS-ISA instructions that are expanded on-the-fly to N-ISA instructions when fetched from the program memory. Figure 1 illustrates this scheme. As will be shown, the findings in this article clearly motivate such a scheme.

In this paper, we make the following main contributions. 1) We introduce the FlexCore datapath, a datapath with exposed control based on the FlexSoC framework. 2) We evaluate the microarchitecture in detail and show that it enables better performance in cycle-count and execution time as well as energy efficiency. 3) In this framework, we also show that the exposed pipeline comes at a cost of both static code-size and instruction bandwidth.

This paper is organized as follows: The FlexCore architecture is introduced in Section II and III, followed by compilation techniques used, Section IV. Section V presents
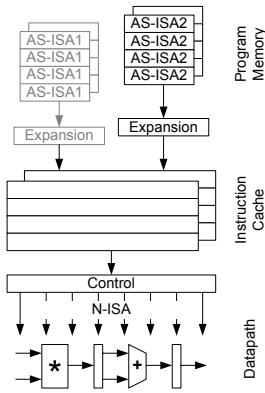
Fig. 1. AS-ISA instruction decoding into N-ISA instructions

Fig. 2. Illustration of a baseline FlexCore. Note that each DATA Reg also has a stall signal not shown in the figure.

how the FlexCore architecture was evaluated and the results are presented in Section VI. Related work is discussed in Section VII and the paper is concluded in Section VIII.

II. THE BASELINE FLEXCORE ARCHITECTURE

Application studies in our field of interest, in particular comparisons [3] of two audio compression standards (MP3 and Ogg Vorbis), have convinced us that full GPP functionality is necessary for flexibility. To offer the full programmability of a GPP, we have therefore decided to include all the datapath units necessary to emulate a full-featured processor in our baseline architecture. We have opted to use a conventional, single-issue, five-stage pipeline similar to the Hennessy-Patterson 32-bit DLX and MIPS R2000 as a template [4]. In FlexSoC, our ambition is to provide application performance mainly through the use of specialized accelerators and fine-grained control rather than through conventional methods. Thus, our core is designed to be flexible and gradually extensible, with accelerators according to application requirements. Recent research have shown that simple cores can be very energy efficient and successfully used in high-performance systems [5], [6].

The datapath is fully exposed and is controlled through a 91-bit wide N-ISA control word (see Figure 2). The baseline FlexCore consists of four datapath units: Register File, Arithmetic Logic Unit (ALU), Load/Store Unit (LS Unit), and a Program Counter Unit (PC Unit), with all units connected to a flexible, fully connected interconnect. To allow for GPP functionality, each datapath-unit's output port is connected to a data register. The data registers act as pipeline registers in the various pipeline configurations that can be assembled using the flexible interconnect. Thus, it is easy to create different pipelines by routing a result from one datapath unit to the next.

To allow instructions to be scheduled on the FlexCore in the same way as on a conventional five-stage pipeline, two data registers (RegA and RegB) have been included. The two data registers are used in the execute and load/store stage of a conventional GPP and allow data to bypass the ALU and LS unit.
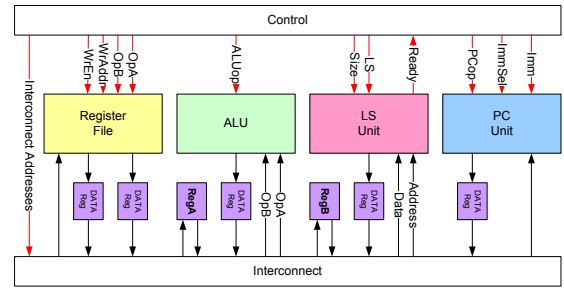
The baseline FlexCore can act as a GPP since it can emulate a conventional pipeline and run instructions in the same way. The FlexCore architecture also allows for high resource utilization due to the flexibility in scheduling, due to the fine-grained N-ISA control word.

A. N-ISA: Exposed Datapath

As described in Section I, the datapath of a FlexCore can be precisely controlled through the N-ISA control word. The N-ISA format depends heavily on the architecture and its datapath units. Figure 3 shows the N-ISA for the baseline FlexCore architecture. Starting from the least significant bit, the control word consists of bits that control the interconnect, the PC Unit (which also includes the 32-bit immediate value), the two data buffers, the Load/Store Unit, the ALU, and finally the Register File.

Interconnect	PC		D	LS	A	Register

Fig. 3. FlexCore N-ISA control word. The different fields are: Interconnect (24 bits), PC (37 bits, of which 32 bits are immediate), D (Data buffers, 2 bits), LS (Load/Store, 5 bits), A (ALU, 5 bits), and Register (18 bits). The total length is 91 bits.

The expected functionality of the datapath units allows us to identify the role of most of the bits in a straightforward manner. For example, the bits controlling the register file contain the fields denoting which two registers to read and which register to write. These bits also contain a write enable signal and two stall signals for each read port data register. The PC Unit handles the immediate value, and the ImmSel signal selects if the value emitted from the PC Unit should be the current Immediate value, or the address of the next instruction (which is used in jump-and-link-like instructions).

The N-ISA includes the bits controlling the interconnect. Since each output can be associated to any input in every cycle, the number of bits, n, needed to control an N-input, M-output interconnect is $n = N \cdot \lceil \log_2(M) \rceil$.

In each cycle, an N-ISA word controls all units in the datapath as well as the interconnect. The exposed-datapath approach differs from the conventional pipelined control word, which is found in general-purpose processors and digital-signal processors, and in which one control word (corresponding to one instruction) contains information about all pipeline stages, for this *and* consecutive cycles.

As can be seen in Figure 3, the N-ISA word consists of 91 bits. Compared to a conventional 32-bit GPP, the FlexCore requires an instruction bandwidth that is almost three times as large, in order to keep the datapath busy. The N-ISA is clearly not an efficient representation. Therefore, FlexSoC assumes a reconfigurable instruction decoder/expander in the hardware/software interface. It comes as no surprise that our results confirm that both the static code size and the instruction bandwidth must be addressed.

The goal of giving the compiler complete control of the hardware has the drawback that binary compatibility between processors with different datapath architectures is lost. With a reconfigurable instruction decoder, as proposed in the FlexSoC framework [1], it may however be possible to reuse the same AS-ISA for different hardware configurations, but that is a topic for future research and, thus, not addressed in this work.

In Section VI, we analyze performance gains from using an exposed datapath and a flexible interconnect with the same datapath units as a conventional five-stage pipeline.

III. EXTENSIONS TO THE BASELINE FLEXCORE

An advantage of the FlexCore architecture is that it can be extended with application-specific accelerator units, simply by adding more ports to the flexible interconnect and extending the N-ISA control word to include control signals for the new units. Since different units are treated equally, we hope to avoid complex ad-hoc solutions usually found in irregular interconnects. For instance, for each unit added to a conventional pipeline, the forwarding network with control logic has to be modified. A conventional fixed pipeline depth also makes it cumbersome to add datapath units and utilize them efficiently: either the new unit is put in the execute stage and can thereby only be used if the ALU is not used; or a new pipeline stage is added, which changes the architecture considerably; or the unit can be added as a co-processor, which causes communication overheads.

A fully connected crossbar guarantees that the interconnect will not restrict the scheduling of operations on the datapath units. This motivates its use in the explorative phase of the design. As seen in Section VI, the full connectivity may not be needed for a given application domain; this provides an opportunity to reduce the area and power requirements, once a suitable collection of datapath units has been determined.

A. Multiplier Extension

For this study, the baseline FlexCore has been extended with a multiplier in order to be able to efficiently execute embedded application benchmarks, such as the Fast Fourier Transform (FFT). The 32-bit multiplier, which is pipelined into two stages to balance its critical path to the other datapath units, is connected to the interconnect that has been extended with two input and two output ports. The two output ports deliver operands to the multiplier, while the result is divided into two 32-bit values. Each value is connected to an input port of the interconnect. One of the ports carries the 32 least
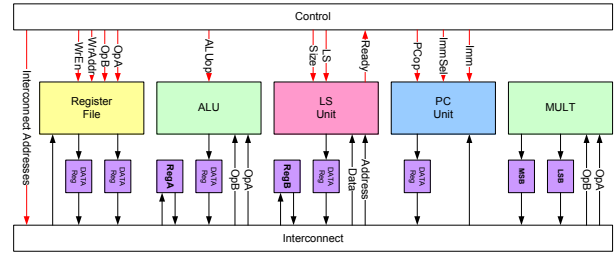

Fig. 4. Illustration of a baseline FlexCore extended with a multiplier. Note that each DATA Reg has a stall signal, while RegA, RegB, LSB, and MSB have an enable signal which is not shown.

significant bits of the result, while the other port carries the 32 most significant bits of the result, as shown in Figure 4.

The N-ISA for the extended FlexCore consists of 108-bits. The multiplier has no control signals, since it is only capable of executing one operation. The extension to the N-ISA is due to extra bits for addressing the added ports in the interconnect, as well as two enable signals for the LSB and MSB register that is holding the result after a multiplication.

IV. COMPILING FOR FLEXCORE

The flexibility of architectures based on the FlexCore concept enables numerous compilation strategies. Given the ability of a FlexCore to emulate a conventional GPP, we chose as our initial approach to translate GPP-like assembly instructions into N-ISA code. As this work will show, the FlexCore can indeed emulate a conventional five-stage pipeline in real time.

The translation of single GPP instructions to N-ISA code is straightforward. We use the same datapath structure as a five-stage pipeline, but the instruction fetch stage is implicitly handled by the FlexCore control unit. In other words, each instruction spans four cycles. The first cycle uses the immediate port and the read ports of the Register File. The second cycle uses the ALU and RegA. The third cycle uses the Load/Store Unit and RegB. Finally, the fourth cycle uses the write port of the Register File. Sequences of such instructions are merged using static optimization techniques (see Section IV-A below) to achieve pipelining and forwarding.

Obviously, making the FlexCore operate as a GPP is not the best way to exploit this architecture. However, we chose to execute GPP programs on the FlexCore to show that it is at least as powerful as a GPP. Even though the FlexCore interconnect allows for communication between any two units, GPP instructions use only the paths corresponding to those found in the GPP. Therefore, we aim to compile high-level code down to N-ISA along the lines of other compilation methods for general datapaths [7]. This enables the pipeline length and structure to be changed as often as needed, and allows for programs to use the datapath units in any order. Currently, profiling allows the programmer to manually schedule performance-critical regions. The manually scheduled parts are written in an RTN[1]-like format that can be interleaved with GPP instructions.

---

[1]Register Transfer Notation.

A. Instruction-Level Static Code Optimization

Translating GPP assembly code into N-ISA code yields a number of N-ISA instruction sequences that should be scheduled as tightly as possible, overlapping each other as allowed by resource conflicts and data dependencies. Resource conflicts are not an issue in the case of GPP assembly instructions, due to their pipelined structure. Data dependencies are more important, since consecutive operations often use the same register.

When one operation uses the contents of a register that is updated by the previous operation, several cycles can often be saved by forwarding: taking the value directly from the datapath unit that produces it, rather than waiting for it to be written to the register file first.

Pipelined processors usually do these optimizations at run-time. However, due to the exposed control word of the FlexCore, they must be done statically. The basic operation is to compose two sequences of N-ISA instructions sequentially, with as much overlap as possible. This is done by annotating each instruction with information about what resources that are used, and the status of all registers. Each register can have status *available*, *unavailable*, or *rerouted(p)*, where p is the name of an output port of a datapath unit. Normally, registers are marked as *available*, which means that their value can be read from the register file. A register is *unavailable* when a new value for the register is currently being computed and is not yet available. When the value is available but not yet written to the register file, the *rerouted(p)*-annotation tells the compiler where the value can be found. In such a case, the register read is omitted, and the value is fetched from the port p instead of the register port.

Techniques such as these are not restricted to rescheduled conventional pipelined programs, but can be used for any N-ISA code. They help determine whether any two annotated N-ISA sequences are composable with a fixed overlap. To find out the maximal possible overlap, we begin by composing them without overlap, then with one cycle overlap, thereafter with two cycles overlap, and so on until we fail. It may be possible to continue even further, but then we must perform a more careful analysis to make sure that no write order conflicts occur. However, we do not expect such aggressive optimizations to have a significant efficiency impact.

Even though the compiler at times needs to make pessimistic choices not to violate any data dependencies when scheduling, our results show that the generated code is only marginally slower on GPP-code compared to a fixed five-stage pipeline with dynamic forwarding and pipelined control.

V. Experimental Framework

To evaluate the performance of FlexCore, four different benchmarks from the Embedded Microprocessor Benchmark Consortium (EEMBC) were selected. The four benchmarks are the Fast Fourier Transform (FFT), Autocorrelation (Autocor) and Viterbi Decoder (Viterbi) from the Telecom benchmark suite and the High pass grey-scale (RGBHPG) filter from the Consumer benchmark suite. All selected benchmarks were executed to completion and the result presented excludes code belonging to the test-harness.

To distinguish between the performance gains achieved by an exposed datapath and the flexible interconnect, a FlexCore with only the interconnects present in a conventional GPP pipeline has also been simulated; it is identified as "Exposed GPP" in the tables.

We chose MIPS as the ISA for the GPP pipeline, since it is supported by mature free-ware tools and cross-compilers. Each of the benchmarks was compiled using a cross-compiler to MIPS assembly with the default optimization flags for EEMBC (-O2). The assembly code was profiled to identify computational kernels, which subsequently were manually scheduled, using RTN notations, for the Exposed GPP and the FlexCore. All the benchmarks were manually scheduled without doing any algorithmic changes or optimizations, such as loop unrolling. Each assembly instruction was translated into RTN notations and scheduled as early as possible onto the given datapath. An instruction can therefore be scheduled earlier than given by the original assembly code, if there are no data dependencies and no resource conflicts. Thus, three versions of each benchmark were generated:

- **GPP**- The output of the FlexCore compiler. As described in Section IV, the compiler schedules the instructions in the same way as if they would have been executed on a conventional GPP.
- **Exposed GPP** - The most frequent instructions (inner loops) have been manually scheduled using only the communication paths of a conventional GPP. For example, RTN-operations make it possible to remove unnecessary register updates.
- **FlexCore** - Manual scheduling of the same instructions as for the Exposed GPP but with the full interconnect of the FlexCore. For example, data can be moved directly from the ALU to the register file, without updating the buffer registers.

The code was executed on a cycle-accurate simulator that models a FlexCore connected to an ideal memory architecture, with single cycle latency. The simulator has been verified against a VHDL implementation (see Section V-A below). By using CRC-checks on the output of the benchmarks, we have verified that they execute correctly.

A. Hardware Implementation

To evaluate the performance in terms of delay, power/energy, and area, VHDL implementations were created for the different architectures. Note that no control logic has been implemented for the evaluated architectures in our comparison. The exposed GPP is a conventional GPP, where the control logic has been removed. Therefore, when disregarding control logic and instruction fetch, the exposed GPP and conventional GPP are equivalent. The impact of control logic and instruction fetch on the performance of the different architectures is not a topic of this paper, but this is an issue addressed within the FlexSoC project.
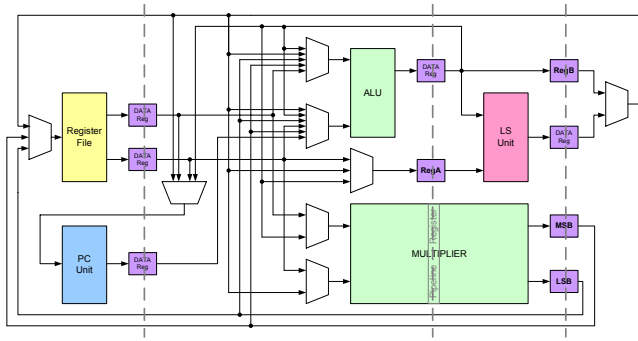
Fig. 5. Schematic of the datapath for the GPP and exposed datapath. Only the communication paths for data are shown, without any control signals.

Figure 5 shows a detailed schematic of the communication paths that are available to the GPP and the exposed datapath. The datapath is inspired by the DLX and MIPS R2000 datapaths [4]. 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 conventional 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.

The baseline FlexCore datapath using the flexible interconnect was designed such that an input port to a datapath unit can be connected to any output port of any datapath unit. This creates a fully connected crossbar switch as interconnect.
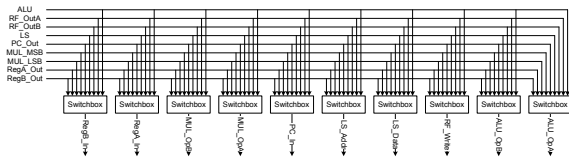

Fig. 6. Illustration of the crossbar switch of the fully connected interconnect.

Figure 6 illustrates how the crossbar switch is constructed out of a number of switchboxes. Each input to a datapath unit (in Figure 5) as well as to the two registers RegA and RegB is 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 7.
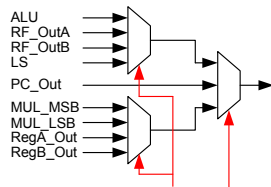

Fig. 7. Illustration of a switchbox.

The VHDL descriptions were synthesized (Synopsys Physical Compiler [8]) and placed and routed (Cadence NanoEncounter [9]) using a commercially available 0.13μm technol-
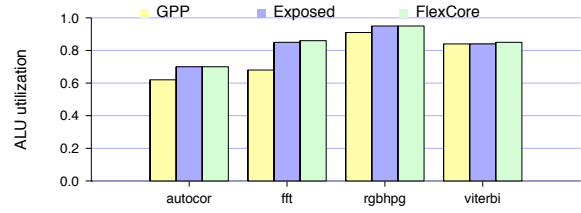

Fig. 8. Percentage of cycles the ALU is used.

ogy. Delay and power estimations were done on the placed and routed netlists with extracted resistance and capacitance. The power estimations are given for the maximum clock frequency for each of the datapaths.

VI. RESULTS

A. Cycle-Count Evaluation

In the first set of experiments we have executed the benchmarks and measured the dynamic cycle count. Table I shows the dynamic instruction count for the benchmarks. The columns labeled Loop show the number of cycles spent in the manually scheduled inner loops. Here we see that at least 75% of the executed cycles are spent in the inner loops. The percentage in the table allows us to compare the cycle count between the three architectures.

The FlexCore architecture is between 10% and 50% better in cycle count compared to the GPP for all the applications. The Exposed GPP improves only some of the applications, with FFT being the best with 20% improvement, while Viterbi could not be improved at all. These results show that the flexible interconnect improves the performance for the different applications.

In order to understand the difference in performance among the benchmarks, the utilization of the resources in the datapath was also monitored. As with many embedded media benchmarks we saw that the inner loops performed many arithmetic operations on register values.

Figure 8 shows the relative number of cycles the ALU was used, for the various benchmarks. For all applications, the ALU utilization in the FlexCore is higher than that in the Exposed GPP, whose ALU utilization in turn is higher than in the GPP. Comparing the results with the dynamic instruction count, we see that the benchmarks with the smallest ALU utilization gained the most by the FlexCore architecture. Autocor and FFT, which have only 60% and 70% ALU utilization, improved by 33% and 36% in cycle count with FlexCore, while RGBHPG and Viterbi with a 90% and 85% ALU utilization only gained 18% and 10%.

Furthermore, the number of ALU operations executed in total decreases in FlexCore. Figure 9 shows that between 5% and 22% of all ALU operations executed by the GPP are redundant and have been removed for FlexCore. An example of redundant ALU operations is the address calculation with a zero offset for load and store operations.

TABLE I
APPLICATION CYCLE COUNT

Datapath	Autocor	Autocor (Loop)	FFT	FFT (Loop)	RGBHPG	RGBHPG (Loop)	Viterbi	Viterbi (Loop)
GPP	1.5k (100%)	1.3k (100%)	58k (100%)	43k (100%)	3.4M (100%)	3.3M (100%)	268k (100%)	218k (100%)
Exposed GPP	1.3k (88%)	1.1k (86%)	47k (80%)	31k (71%)	3.2M (93%)	3.2M (97%)	268k (100%)	222k (102%)
FlexCore	1.0k (67%)	0.8k (62%)	37k (64%)	22k (50%)	2.8M (82%)	2.8M (86%)	243k (90%)	198k (90%)

In a GPP, all calculated values are written to the register file. Nevertheless, all written values that are used in the next instruction will be routed through the forwarding network. If the value is never again read from the register file, the write is unnecessary. In the FlexCore architecture, it is possible to skip the generation of such writes, as long as it is possible to statically find such locations in the program. Figure 9 shows the relative number of register writes for the different architectures compared to the GPP version. On average, the Exposed GPP allows us to remove 28% of all register writes and for FlexCore 36% are removed. This reduces the contention of the register file as well as saves power[2].

B. Performance Evaluation

TABLE II
DELAY, POWER, AND AREA ESTIMATES

Datapath	Timing (ns)	Power (mW)	Area (mm^2)
GPP	2.25 (100%)	6.57 (100%)	0.35 (100%)
Exposed GPP	2.25 (100%)	6.57 (100%)	0.35 (100%)
FlexCore	2.49 (111%)	8.07 (123%)	0.40 (115%)

Cycle count alone is not sufficient to make a comparison of the different architectures. One also needs to consider implementation aspects. Table II shows the result of timing, power, and area estimations for the FlexCore, the Exposed

[2]It also complicates exception handling, which is however not the topic of this paper.
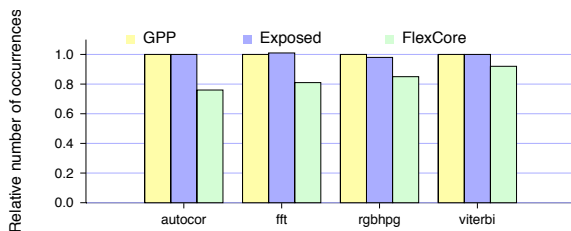
(a) ALU operations
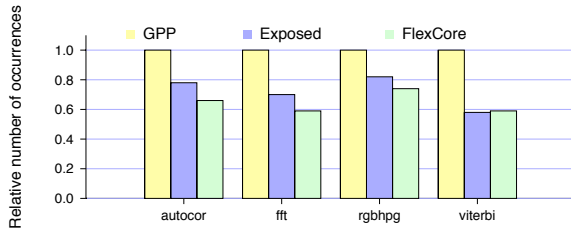
(b) Register Writes

Fig. 9. Number of operations for the benchmarks normalized to the GPP.

GPP and a conventional GPP. As previously explained, we have modeled the microarchitecture without control and thus the implementation result for the GPP and the Exposed GPP are the same. As expected, the increase in flexibility that comes with the fully connected interconnect is associated with a cost. Area-wise, the extra paths require 15% extra space. More importantly, the delay of the critical path is increased by 11% and the power dissipation is increased by 23%. By combining these results with the cycle count, the true performance of the different approaches is shown. Table III shows the total time as well as the amount of energy (*power · clock period · cycle count*) needed to execute each benchmark.

These figures show that three of the four applications are executed faster on the FlexCore than on the GPP, when the cycle time is also considered. Compared to the Exposed GPP, we notice that the performance boost is attributed both to the exposed control and the flexible interconnect. With GPP as reference, the Exposed GPP gives up to 19% performance improvement, while the FlexCore delivers improvements up to 30% on the same benchmark (FFT).

The energy result on the other hand is not as promising. Comparing the FlexCore architecture to the GPP shows that only two of the benchmarks are executed with less energy (8% for Autocor and 14% for FFT) while two expend more energy (12% for RGBHPG and 23% for Viterbi). This shows that if energy is a first-level constraint, the improvement in cycle count is not large enough to outweigh the penalty associated with the fully connected interconnect.

With this in mind we study how the interconnect is utilized in the applications. In the GPP architecture there are 33 possible communication paths, while the fully connected interconnect offers 90 paths. Table IV depicts all these paths as well as the ones that are used by our four benchmarks. Paths used by the benchmarks are marked with GPP and Flex. Here we see that out of the 90 paths available in the fully connected interconnect, 43 are *never* used. Notice that the compiler statically schedules data movements, so we can be completely certain that they will never occur in our benchmarks. It is clear that it should be possible to prune communication paths from the flexible interconnect, without losing any performance. An earlier study [10] showed that by pruning the network the delay can be reduced by 5% (2.4 ns), and at the same time reduce the power by 8% (6.8 mW).

C. Static Code Size

The static code size for the benchmarks is shown in Table V. The results are presented for both the full benchmark (excluding library code) and for the manually scheduled inner

TABLE III
EXECUTION TIME AND ENERGY DISSIPATION.

	Autocor		FFT		RGBHPG		Viterbi	
	Time (μs)	Energy (nJ)	Time (μs)	Energy (nJ)	Time (ms)	Energy (μJ)	Time (μs)	Energy (μJ)
GPP	3.4 (100%)	22 (100%)	131 (100%)	857 (100%)	7.7 (100%)	50 (100%)	603 (100%)	4.0 (100%)
Exposed GPP	2.9 (85%)	19 (88%)	106 (81%)	695 (81%)	7.2 (94%)	47 (94%)	603 (100%)	4.0 (100%)
FlexCore	2.5 (74%)	20 (92%)	92 (70%)	743 (86%)	7.0 (91%)	56 (112%)	605 (100%)	4.9 (123%)

TABLE IV
INTERCONNECT PATHS USED BY THE BENCHMARK ON THE GPP AND ON THE FLEXCORE

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

TABLE V
CODE SIZE (NUMBER OF INSTRUCTIONS) FOR BOTH APPLICATION AND THE INNER LOOP FOR EACH BENCHMARK.

Datapath	Autocor	Autocor (Loop)	FFT	FFT (Loop)	RGBHPG	RGBHPG (Loop)	Viterbi	Viterbi (Loop)
GPP	60 (100%)	16 (100%)	432 (100%)	42 (100%)	197 (100%)	43 (100%)	366 (100%)	66 (100%)
Exposed GPP	61 (102%)	17 (106%)	424 (98%)	30 (71%)	198 (101%)	42 (98%)	367 (100%)	67 (102%)
FlexCore	56 (93%)	12 (75%)	414 (96%)	21 (50%)	191 (97%)	37 (86%)	360 (98%)	61 (92%)

loop. First focusing on the full benchmarks, we see that the optimized versions are about the same or slightly smaller than the GPP versions.

Note that any change in code size is the result from the manually scheduled inner loops. Therefore we also present the code size for these. Here we see that the FlexCore architecture has made it possible to decrease the number of instructions significantly for some benchmarks. For example, the inner loop of the FFT needs only half the number of cycles compared to the conventional GPP. Since an N-ISA is about three times as large as a conventional GPP instruction, the total static code size for FlexCore is still larger than for a conventional GPP.

The code-size increase originates from some of the manual scheduling, which added postfix code to handle the first iteration of loops. This can be seen in the small increase in code size for the Exposed GPP.

VII. RELATED WORK

Reconfigurable architectures is an active area of research. Dedicated hardware is becoming less attractive because of huge initial costs, long time to market, and inability to adapt to new and changing standards. Reconfigurable hardware is a promising approach to address these problems, without forsaking the performance of dedicated hardware. Hartenstein has compiled a thorough survey of reconfigurable architectures [11]. Many modes of reconfigurability have been proposed: reconfigurable accelerators may be connected to a standard pipeline [12]; or reconfigurable tiles may be orchestrated to solve given problems [13], [14]. In contrast, the FlexSoC approach employs reconfigurability only in the instruction decoding hardware, leaving the actual data processing to highly efficient dedicated hardware.

The exposed datapath concept has recently been used in the No Instruction Set Computer (NISC) [15], [2], [7] project, where the control pipeline is removed and the controller emits a wide instruction word each cycle. Co-design refinement of hardware and software is used to reach the desired performance. The reported speedups are comparable to those we see for the FlexCore example. However, the static code size of a NISC program is claimed to be comparable to that of a GPP. While this might be true for a co-design approach where common complex operations can be implemented with few control bits, we have not seen the same results for the FlexCore architecture. In FlexSoC, we rely on compression and run-time expansion to solve the code size problem. Increased controllability of the datapath has been motivated by the reduction in hardware complexity, as in the Transport Triggered Architecture [16].

Liang *et al* [17] propose an architecture based on a reconfigurable interconnect and show good performance for some domain-specific computations. It is, however, not clear how the results translate to a wider domain of applications.

A common way to accelerate multimedia applications is to add sub-word parallelism within the datapath units (SIMD). This technique is used both in modern general-purpose computers and specific media processors. For a five-stage DLX implementation, Nia and Fatemi report a speedup of more than a factor of three with only minor growth in chip area [18]. The approach is orthogonal to those proposed here and would seem to make a fruitful addition to a FlexSoC core.

Similarly to FlexSoC, the FITS project [19], [20], [21] also envisions the use of flexible instruction decoders. Application profiling allows the selection of a 16-bit application-specific ISA that gives the same performance as the 32-bit baseline case. FlexSoC combines a similar application-specific ISA approach with the performance gains offered by the exposed datapath and the flexible interconnect.

The translation envisioned in the FlexSoC project is somewhat similar to microcode processing, where a complex ISA is broken down into micro-operations that are executed on the pipeline. The main purpose of microcode is to separate the architecture from the implementation and the microcode is usually derived from the already given ISA. In FlexSoC, this constraint is relaxed and the AS-ISA can be created by the compiler to fit the needs of the applications.

VIII. CONCLUSION

The exposed datapath of FlexCore offers distinct performance benefits when compared to a GPP with corresponding datapath units. The flexible interconnect network further improves performance, and also allows special-purpose datapath units to be integrated while maintaining a uniform programming interface. With knowledge of the datapath structure, a compiler can realize these performance benefits. Additionally, it is always possible to execute programs compiled for MIPS at cycle counts comparable to a standard MIPS implementation.

We have analyzed four applications and shown that for a set of embedded applications, FlexCore is between 10% and 40% faster in terms of cycle count compared to a conventional five-stage GPP with the same datapath units. This speedup is attributed to both the exposed datapath and the flexible interconnect. Using cycle times obtained from placed and routed layouts, we show that this cycle count translates to a total execution-time improvement of up to 30%. We also show that it is feasible to prune the fully connected interconnect without losing any performance in cycle count. This would allow for a more energy-efficient implementation as well as improve the cycle time. This performance boost comes at a cost of both instruction bandwidth and static code size: the FlexCore instructions are about three times as wide as a 32-bit GPP instruction. The number of static instructions is reduced for the FlexCore, but not by a factor of three. Therefore, the static code size will be larger for an application compiled for FlexCore. This clearly motivates a reconfigurable instruction decoder to reduce both the static code size as well as instruction bandwidth, as proposed in the FlexSoC framework.

Future work includes evaluating a reconfigurable instruction decoder together with FlexCore. Different compression schemes can be expected to be more or less suited to the ISA transformations needed, and to carry different implementation costs. Configuration of the instruction decoder could be a one-time event; but run-time, on-demand reconfiguration offers intriguing possibilities, where several tasks, each with a distinct AS-ISA, could share the same hardware.

REFERENCES

[1] J. Hughes, K. Jeppson, P. Larsson-Edefors, M. Sheeran, P. Stenstrom, and L. J. Svensson, "FlexSoC: Combining flexibility and efficiency in SoC designs," in *Proceedings of the IEEE NorChip Conference*, 2003.

[2] 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.

[3] J. Mårts and T. Carlqvist, "A hardware audio decoder using flexible datapaths," MSc Thesis, Chalmers University of Technology, March 2006.

[4] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, 2nd ed. Morgan Kaufman, 1998.

[5] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[6] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 506–517, 2005.

[7] M. Reshadi and D. Gajski, "A cycle-accurate compilation algorithm for custom pipelined datapaths," in *International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, September 2005.

[8] *Physical Compiler User Guide Version W-2004.12*.

[9] *Encounter User Guid Version 4.1*.

[10] M. Själander, P. Larsson-Edefors, and M. Björk, "A flexible datapath interconnect for embedded applications," in *IEEE Computer Society Annual Symposium on VLSI*, May 2007.

[11] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of Design, Automation and Test in Europe, 2001*, March 2001, pp. 642–649.

[12] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 2000, pp. 225–235.

[13] M. B. T. et al., "Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams," in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2004, p. 2.

[14] K. S. et al., "TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 1, pp. 62–93, 2004.

[15] B. Gorjiara, M. Reshadi, and D. Gajski, "Designing a custom architecture for DCT using NISC design flow," in *ASP-DAC'06 Design Contest*, 2006.

[16] H. Corporaal, "Ttas: missing the ilp complexity wall," *J. Syst. Archit.*, vol. 45, no. 12-13, pp. 949–973, 1999.

[17] X. Liang, A. Athalye, and S. Hong, "Dynamic coarse grain dataflow reconfiguration technique for real-time systems design," in *The 2005 IEEE International Symposium on Circuits and Systems*. IEEE Computer Society, May 2005, pp. 3511–3514.

[18] E. Nia and O. Fatemi, "Multimedia extensions for DLX processor," in *Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems*, Dec 2003, pp. 1010 – 1013.

[19] A. Cheng, G. Tyson, and T. Mudge, "FITS: framework-based instruction-set tuning synthesis for embedded application specific processors," in *DAC '04: Proceedings of the 41st annual conference on Design automation*. ACM Press, 2004, pp. 920–923.

[20] ——, "PowerFITS: Reduce dynamic and static i-cache power using application specific instruction set synthesis," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, 2005, pp. 32–41.

[21] A. C. Cheng and G. S. Tyson, "High-quality ISA synthesis for low-power cache designs in embedded microprocessors," *IBM J. Res. Dev.*, vol. 50, no. 2, pp. 299–309, 2006.