Exposed Datapath for Efficient Computing

Magnus Björk Magnus Själander Lars Svensson Martin Thuresson
John Hughes Kjell Jeppson Jonas Karlsson Per Larsson-Edefors Mary Sheeran Per Stenstrom

Chalmers University of Technology

***Abstract*— We introduce FlexCore, which is the first exemplar of a processor based on the FlexSoC processor paradigm. The FlexCore utilizes an exposed datapath for increased performance. Manually scheduled micro-benchmarks yield a performance boost of up to a factor of two over a traditional five-stage pipeline with the same functional units as the FlexCore. The compiler is always capable of scheduling the instructions of a general-purpose application onto the FlexCore on par with a traditional GPP in terms of cycle count.**

The flexible interconnect allows the FlexCore datapath to be dynamically reconfigured as a consequence of code generation. Additionally, specialized functional units may be introduced and utilized within the same architecture and compilation framework.

The exposed 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 paradigm.

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 colocated with the microprocessor on a single System-on-Chip (SoC). The SoC then constitutes a highly heterogeneous computing system, tuned for a set of specialized tasks.

The present practice 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 the heterogeneous SoC style 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. Moreover, unlike in a conventional VLIW machine, most of the specialized datapath elements would be idle at any given moment, so instruction bandwidth and memory footprint would be wasted.

In our approach, called the FlexSoC scheme [1], we address these problems by moving away from the traditional GPP instruction set architecture (ISA) and use a very wide control word. This allows us to control the functional units in the datapath at a much more fine grain 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 using co-design has shown significant performance improvements [2].

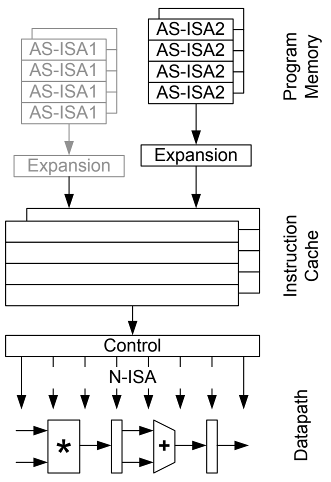

Fig. 1. AS-ISA instruction decoding into N-ISA instructions

Intuitively, both the instruction bandwidth as well as the static code size will be negatively affected by the wide control work. Therefore, in our FlexSoC scheme, we introduce an architecture with 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 which are expanded on the fly to N-ISA instructions when fetched from the program memory. Figure 1 illustrates this scheme.

In this paper, we introduce one exemplar of the FlexSoC paradigm, dubbed the FlexCore [3]. The FlexCore is used to evaluate datapath utilization and for investigating the demands on the reconfigurable instruction decoder. Our results show that the performance increase indeed comes at a cost of increased instruction bandwidth and larger static code size.

II. The Baseline FlexCore Architecture

To offer the full programmability of a GPP, we have decided to include all the functional units necessary to emulate a full-featured processor in our baseline architecture. Application studies in our field of interest, in particular comparisons [4] of two audio compression standards (MP3 and Ogg Vorbis), have convinced us that full GPP functionality is necessary for flexibility. We have opted to use a traditional, single-issue, five-stage processor similar to the Hennessy-Patterson 32-bit DLX and MIPS R2000 as a pattern [5]. A five-stage processor is not a high-performance design, but in FlexSoC, our ambition is to provide application performance mainly through the use of specialized accelerators rather than through conventional methods. Thus, our core is designed to be flexible and gradually extensible, with accelerators according to application requirements.

The datapath is fully exposed and is controlled through the 91 bit wide N-ISA control word (see Figure 2). The baseline FlexCore consists of four functional units: register file, arithmetic logic unit (ALU), load/store unit (LS Unit), and a program control unit (PC Unit), with all units connected to a flexible, fully connected interconnect. To allow for GPP functionality, each functional unit output port is connected to a data register. These act as pipeline registers in the various pipeline configurations that can be assembled using the interconnect. Thus, it is easy to create different pipelines by routing a result from one functional 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 extra data registers (RegA and RegB) have been included. The two data registers are used in the execute and load/store stage and allow data to bypass the ALU and LS unit.

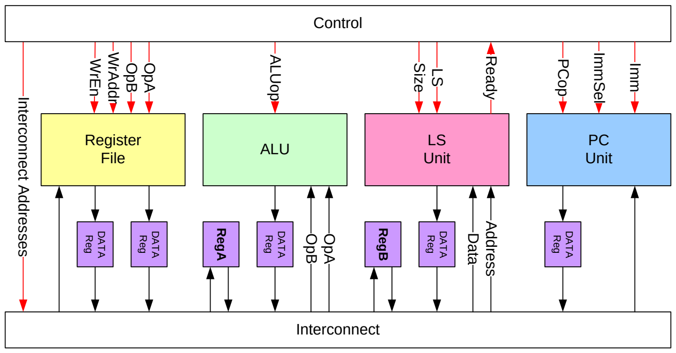

Fig. 2. Illustration of a baseline FlexCore. Note that each DATA Reg also has an enable signal not shown in the figure.

The baseline FlexCore can act as a GPP since it can emulate a conventional pipeline and run instructions in the same way[1]. The FlexCore architecture also allows for high resource utilization due to the flexibility in scheduling, thanks to the fine-grained N-ISA control word.

[1]Excluding support for floating point operations and multiplications.

A. *N-ISA: Exposed Datapath*

As described in Section I, the datapath of a FlexSoC processor can be precisely controlled through the N-ISA control word. The N-ISA depends heavily on the architecture and its functional units. The N-ISA for the baseline FlexCore architecture is shown in Figure 3. Starting from the least significant bit, the control word consists of bits that control the interconnect, the program counter unit (which also includes the 32-bit immediate value), the two data buffers, the load/store unit, the ALU, and finally the register bank. Most of the bits are very straightforward to determine from the expected functionality of the functional unit. For example, the bits controlling the register file contain the fields for which two registers to read, which register to write, a write enable signal, and two stall signals for each read port data register[2]. 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 $n = N \cdot \lceil \log_2(M) \rceil$.

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.

In each cycle, the controller outputs an N-ISA word, which controls all units in the datapath as well as the interconnect. The exposed-datapath approach differs from the traditional pipelined control-word found in general-purpose processors and digital-signal processors, where one control word (corresponding to one instruction) contains information about all the pipeline stages, for this *and* consecutive cycles. In a space/time diagram, the GPP creates a diagonal control word, while the N-ISA gives a horizontal control word.

As can be seen in Figure 3, the N-ISA word consists of 91 bits. Compared to a traditional 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 for storage of the program. Therefore, FlexSoC assumes a reconfigurable instruction decoder/expander in the hardware/software interface. Our results show that both the static code size and the instruction bandwidth must be addressed using the proposed scheme.

The goal to give 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 scheme [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 here.

[2]These are used to stall the datapath i.e. on a data cache miss.

In Section IV, we show performance gains from using an exposed datapath and a flexible interconnect with the same functional units as a traditional five-stage processor.

B. Extensions to the Baseline FlexCore

The FlexCore architecture can be extended with application specific accelerator units, simply by adding more ports to the 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 normal pipeline, the forwarding network with control logic has to be modified. A conventional fixed pipeline depth also makes it cumbersome to add functional 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 coprocessor, which causes communication overheads.

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

III. Compiling for FlexCore

The flexibility of FlexSoC architectures enables numerous compilation strategies. Given the ability of FlexCore to emulate a conventional processor, we chose as our initial approach to translate conventional GPP-like code into N-ISA code. This work has shown that the FlexCore can indeed emulate a conventional five-stage processor in real time: on the examples we have tried out, the number of cycles required for running the same program on a DLX and on the FlexCore has differed by less than 3%.

The translation of single GPP instructions to N-ISA code is straightforward. We use the same pipeline structure as in the DLX, 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 bank. The second cycle uses the ALU and one data register. The third cycle uses the load/store unit and the other data register. Finally, the fourth cycle uses the write port of the register bank. Sequences of such instructions are merged using the static optimization techniques described below, to achieve pipelining and forwarding.

The technique of executing GPP programs on the FlexCore is useful for showing that it is at least as powerful as the DLX processor, and can be configured to work as one. Obviously, this is not the best way to exploit the architecture. Even though the interconnect allows communication between any two units, DLX programs use only the paths corresponding to those found in the DLX processor. We therefore aim to compile high level code down to N-ISA along the lines of other compilation methods for general datapaths, such as [6]. This enables the pipeline length and structure to be changed as often as needed, and even allows programs to use the functional units in any order. Currently, profiling allows the programmer to manually schedule performance-critical regions.

A. Instruction-Level Static Code Optimization

Translating DLX 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 DLX code, thanks to its pipeline 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 functional unit that produces it, rather than waiting for it to be written to the register bank first. Another simple optimization is to change the register read port, if two registers would otherwise be read by the same port in the same cycle.

Pipelined processors usually do these optimizations at runtime. However, due to the exposed control word of the FlexCore, we can and must do them 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 functional unit. Normally, registers are marked as *available*, which means that their value can be read from the register bank. 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 bank, 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-pipeline 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 conflict occurs. However, we do not expect such aggressive optimizations to have a significant efficiency impact.

B. Basic-Block Level Static Code Optimization

Basic blocks should also be scheduled as tightly as possible. We model basic blocks as sequences of N-ISA instructions, ending with a branch, a static jump, or a dynamic jump. A branch has a condition (such as *the zero flag of the ALU is* *set*) and two addresses; a static jump has one address; and a dynamic jump has no information in the N-ISA word (the destination address is passed through the interconnect). At the end of the block, there may be a tail, which is another sequence of N-ISA instructions. These instructions are merged at run-time with the instructions of the next basic block by bitwise or operations. The tail is used to model configurations where each AS-ISA instruction may span over several cycles.

How soon the next block can start to execute after a jump can be calculated using the instruction level optimization methods for the tail of the first block and the code of the second block. To do this, all possible paths in the program must be calculated. This is easy to do for branches and static jumps. The possible destinations of dynamic jump operations can be found out by keeping track of all code addresses that are stored in data registers (i.e., all *jal* and *jalr* instructions). The delay of branches and static jumps can be stored in the block, while the delay of dynamic jumps is better represented as a number of *nop*s in the successor block (assuming that only one function can return to a specific address, while a specific function can return to several different addresses).

IV. Experiments and Results

The initial benchmarks for FlexCore are taken from the embedded domain. The first benchmark is the matrix operation *sum of absolute differences* (SAD), a common kernel in many media applications such as MPEG-2 video encoding. The benchmark takes two 3x3 matrices and returns the sum of the absolute difference between each of the corresponding matrix elements. In the other benchmark, *matrix convolution,* a 3x3 filter matrix is applied on each pixel of a 4x4 image. The image also has a border consisting of zeros around it in order to handle the pixels on the edges correctly. Equation 1 shows the operation on each pixel, where F is the filter image and X the original image. The computed result is rounded down to 255 or up to 0 if necessary.

$$Y[x,y] = \sum_{i=0}^{2}\sum_{j=0}^{2} F[i,j] \cdot X[x+i-1, y+j-1] \quad (1)$$

In order to compare the FlexCore against a 5-stage GPP, we have used WINDLX[3], a simulator for the DLX architecture. To distinguish between the performance gains achieved by an exposed datapath and the flexible interconnect, a FlexCore with only the interconnects present in the GPP pipeline has also been simulated; it is identified as “Exposed GPP” in the tables. Each benchmark has been manually optimized for the three architectures and both the static and dynamic instruction count have been analyzed.

For the convolution benchmark, we have added a multiplier to the FlexCore architecture. To make the comparison to the GPP-implementation, we have used the same 4-cycle delay as in WINDLX in our architecture.

[3]WINDLX: Developed by Herbert Grünbacher, University of Technology Vienna, Inst. für Technische Informatik

In the FlexCore datapath, all functional units, including a multiplier, can work directly with data in the register file. This is not true for the original DLX, which has special purpose multiplication registers and associated move instructions to move data to and from these registers. The WINDLX simulator does not model this special purpose register for integer multiplication and will have better performance compared to an exact model. This difference works in favor for the GPP and our result might be a bit more pessimistic because of this.

A. Performance Evaluation

The metrics used in the experiments are static code size and dynamic instruction count. The results are presented in Table I. For both benchmarks, the FlexCore architecture managed to perform the same task using only half the cycles of the GPP. The speedup of a factor of two is clearly achieved by the more efficient use of the available functional units. The exposed data path together with the flexible interconnect yields a substantial performance boost without resorting to codesign or adding dedicated accelerators.

TABLE I
Simulation results for the benchmarks SAD and convolution.

	SAD		Convolution	
	Code Size	Cycle Count	Code Size	Cycle Count
GPP	17 inst	152 (100%)	40 inst	2735 (100%)
Exposed GPP	20 inst	85 (56%)	68 inst	1774 (65%)
FlexCore	18 inst	76 (50%)	49 inst	1423 (52%)

The code size presented in Table I show that the FlexCore programs are of the same size or slightly larger than their GPP counterparts. This together with the fact that each FlexCore instruction is almost three times larger than a GPP instruction clearly shows that both the static code size and instruction bandwidth need to be addressed.

As can be seen in Table I, a FlexCore with a fully interconnected network achieves a speedup of 11% to 20% compared to only utilizing an exposed datapath, as in the Exposed GPP example. This is because several interconnect paths not available in the GPP were used. Table II shows a list of those paths. However, out of the 90 paths available in the interconnect, only 12 were used for SAD, and 17 for convolution. The interconnect is clearly underutilized for these benchmarks.

TABLE II
Non-GPP interconnect paths used in FlexCore benchmarks.

From	To
Register bank	Register bank
ALU	Register bank
Immediate	Register bank
Multiplier	ALU
Register bank	Load/Store
Load/Store	Multiplier

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 by-pass network; in cases where the value is never again read from the register file, the write is unnecessary. In the FlexCore architecture, a compiler could potentially skip the generation of such writes, as long as it can statically find such locations in the program. In this particular case, the number of register writes was reduced from 76 to 57 (by 25%) for the SAD benchmark and from 1438 to 921 (by 36%) for the convolve benchmark by manual scheduling of the instructions. This reduces the contention of the register file as well as saves power[4].

B. Implementation

To evaluate the performance in terms of delay, power, and area, VHDL implementations have been created for the different processors. Note that no control logic has been implemented for the evaluated processors in our comparison. The exposed GPP is a traditional GPP where the control logic has been removed. Therefore, when disregarding control logic and instruction fetch the exposed GPP and traditional GPP are equivalent. The impact on performance by control logic and instruction fetch for the different processors is addressed within the FlexSoC project, however, it is not the topic of this paper.

The VHDL has been synthesized (Synopsys Physical Compiler [7]) and placed and routed (Cadence NanoEncounter [8]) using a commercially available 0.13μm technology. Timing and power estimations (Synopsys PrimeTime [9] and PrimePower [10]) were done on the placed and routed netlists using back-annotated capacitances. The power estimations are for maximum clock frequency for each of the processors and with PrimePowers default values for activities on the inputs. Table III shows the result of timing and power estimations for the baseline FlexCore, a FlexCore extended with a multiplier, the exposed GPP and a traditional GPP. Both the GPP and exposed GPP are equipped with a multiplier.

TABLE III
TIMING, POWER, AND AREA ESTIMATIONS.

	Timing (ns)	Power (mW)	Area (mm^2)
GPP (MULT)	4.4 (100%)	35.70 (100%)	0.426 (100%)
Exposed (MULT)	4.4 (100%)	35.70 (100%)	0.426 (100%)
FlexCore (MULT)	5.5 (125%)	34.47 (97%)	0.444 (104%)
FlexCore	5.1 (116%)	34.39 (96%)	0.275 (65%)

The FlexCore implementations perform worse in terms of delay in comparison to a GPP and exposed GPP pipeline. This is most likely due to the more flexible interconnect that gives a performance penalty. On the other hand, power and area are competitive with that of a GPP and exposed GPP pipeline.

The FlexCore implementation shows more promising results when considering execution time and energy dissipation for a whole application. The Convolution benchmark shows comparable execution time and slightly lower energy dissipation compared to the exposed GPP pipeline, Table IV. For the much shorter SAD benchmark the full potential is not shown for the FlexCore, since the difference in cycle-count is much smaller. Compared to a traditional GPP the FlexCore has both an improved execution time as well as lower energy dissipation. This is due to the much fewer cycles needed to execute the two benchmarks.

[4]It also complicates exception handling, which is however not the topic of this paper.

TABLE IV
EXECUTION TIME AND ENERGY DISSIPATION FOR SAD AND CONVOLUTION.

	SAD		Convolution	
	Time (ns)	Energy (nJ)	Time (ns)	Energy (nJ)
GPP	669 (100%)	23.88 (100%)	12034 (100%)	430 (100%)
Exposed	374 (56%)	13.35 (56%)	7806 (65%)	279 (65%)
FlexCore	418 (62%)	14.41 (60%)	7827 (65%)	270 (63%)

The longer delay of the FlexCore implementation can be decreased by restricting the flexibillity of the interconnect. As shown in the previous section, only a limited number of all available paths are being utilized.

V. 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], [6] 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 have also 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 functional units (SIMD).

This technique is used both in modern general purpose computers and specific media processors. For a 5-stage DLX implementation, Nia and Fatemi report a speedup of more than a factor of 3 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.

VI. Conclusion

The exposed pipeline of the FlexCore offers distinct performance benefits when compared to a GPP with corresponding datapath hardware. The flexible interconnect network further improves performance, and also allows special-purpose datapath elements 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 the DLX at cycle-counts comparable to a standard DLX implementation.

We have analyzed two media kernels and shown that the FlexCore has a speedup of a factor of two in terms of cycle count, compared to a traditional 5-stage GPP with the same functional units. Using cycle times obtained from placed and routed layouts, we show that this cycle count translates to a 35 to 38% total execution-time improvement. 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 standard GPP instruction. Since the number of static instructions does not get improve for the FlexCore, the static code size is also larger. This clearly motivates the FlexSoC scheme of introducing a reconfigurable instruction decoder, to reduce both the static code size as well as instruction bandwidth.

Future work includes incorporating a reconfigurable instruction decoding framework in the 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 be sharing the same hardware.

Acknowledgment

The FlexSoC project is sponsored by the Swedish Foundation for Strategic Research.

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] M. Björk, J. Hughes, K. Jeppson, J. Karlsson, P. Larsson-Edefors, M. Sheeran, M. Själander, P. Stenstrom, L. Svensson, and M. Thuresson, "FlexSoC technical report Q1 2006," Computer Science and Engineering, Chalmers University of Technology, Tech. Rep. 2006-8, 2006.

[4] J. Mårts and T. Carlqvist, "A Hardware Audio Decoder Using Flexible Datapaths," MSc Thesis, Chalmers University of Technology, March 2006.

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

[6] 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.

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

[8] *Encounter User Guid Version 4.1*.

[9] *PrimeTime X-2005.06 Synopsys Online Documentation*.

[10] *PrimePower Manual Version W-2004.12*.

[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.