Design Space Exploration for an Embedded Processor with Flexible Datapath Interconnect

Tung Thanh Hoang, Ulf Jälmbrant, Erik der Hagopian, Kasyab P. Subramaniyan,
Magnus Själander, and Per Larsson-Edefors
VLSI Research Group, Department of Computer Science and Engineering
Chalmers University of Technology, SE-412 96 Gothenburg, Sweden
Email: {hoangt, kasyab, hms, perla}@chalmers.se

***Abstract*—The design of an embedded processor is dependent on the application domain. Traditionally, design solutions specific to an application domain have been available in three forms: VLIW-based DSP processors, ASICs and FPGAs; each respectively offering generality of application domain, energy efficiency and flexibility. However, while matching the application domain to the resources needed, the design space becomes huge. We present FlexTools, a tool framework built around the FlexCore architecture to evaluate performance and energy efficiency for different applications. Here we demonstrate FlexTools for design space exploration with a focus on the data-routing flexibility of the FlexCore processor, in search of energy-efficient interconnect configurations that are both cycle-count and hardware efficient. Evaluation results suggest that a well-optimized instance of a 65-nm multiplier-extended FlexCore processor datapath, obtained using FlexTools, executes nine integer EEMBC benchmarks with a 15% cycle count reduction and dissipates 17% less energy than a reference MIPS datapath.**

I. Introduction

It is a challenging task to perform design space exploration for embedded systems with the goal to reduce execution time and increase energy efficiency for an application domain. Since exploration needs to be performed across many levels of the design hierarchy, from application code to circuits, development tools for HW/SW co-design are essential to assist the designers during different exploration phases.

The FlexCore processor [1] is an attempt to combine the efficiency of an ASIC platform and the flexibility of a reconfigurable platform. The FlexCore processor has a datapath whose circuits are under fine-grain instruction control; the control signals—the Native-ISA (N-ISA)—are exposed to the compiler, via an instruction decompressor. The FlexSoC scheme approaches the implementation of embedded systems from a general-purpose processor (GPP) point of view: In its minimal configuration, the FlexCore processor represents a simple five-stage pipeline, but it can be further extended with datapath units. The FlexCore can always fall back on GPP behavior and execute general code with the same cycle efficiency as its template architecture, that is, a MIPS R2000 [2].

The wide N-ISA that results from the instruction decompression brings several benefits for a domain-specific architecture. First, by extending the N-ISA word, the FlexSoC scheme can accommodate datapath extensions such as different types of accelerators in a straightforward way. Second, despite the FlexSoC scheme mainly targets ASIC implementations, the FlexCore datapath can be made relatively flexible, in that the function of selected units can be altered ("morphed") at run time to suit the executed code. For example, performance can be boosted by way of double throughput (2×16 bit) computation and energy can be reduced by gating the supply voltage of idle units.

The FlexCore processor does not have a fixed datapath interconnect, so pipelining is not hard-coded into the datapath. Instead, the scheduler creates a static instruction schedule targeting a particular FlexCore datapath configuration. This makes it possible to explore interesting HW/SW co-design tradeoffs in a vast design space, to match computing and communication resources of a datapath with the computational need of its applications.

This paper has two main contributions:

- The FlexTools processor design environment that starts from application C code and ends up with a physical implementation of a FlexCore processor. Architecture-wise, FlexTools supports scheduling and simulation for an extensible FlexCore datapath. Hardware-wise, FlexTools performs mapping of the architecture to RTL and place-and-route implementations.
- A design space exploration methodology that uses FlexTools to optimize the FlexCore architecture, with respect to datapath interconnect, to a domain of applications.

In Sec. II, we review the basic aspects of the FlexCore processor. Sec. III describes the new FlexTools environment. The design space exploration methodology, together with exploration results, is presented in Sec. IV. Sec. V presents related work. Finally, Sec. VII concludes the paper.

II. Background - The FlexCore Processor

The common practice when designing processors is to use an instruction-set architecture (ISA) that defines the interface between software and hardware. But the ISA represents an abstract view of the implemented hardware and makes it impossible for the compiler to have direct control of the implemented hardware.

The multiplier-extended FlexCore datapath in Fig. 1 consists of a multiplier and the datapath units (including buffers) found in a simple MIPS R2000 processor; that is, Load/Store (LS), Register File (RF), Arithmetic and Logic Unit (ALU) and

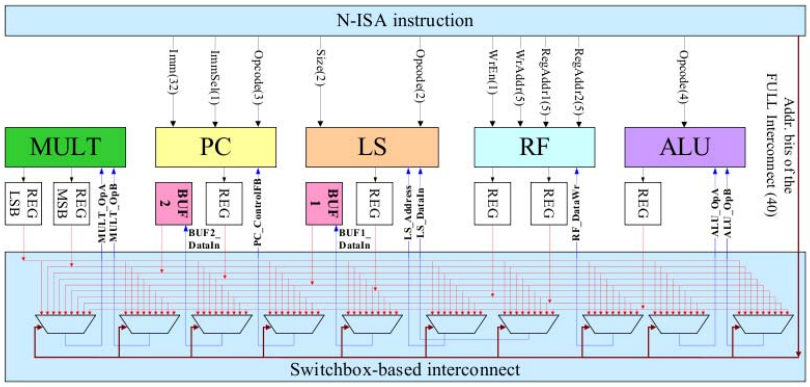


Fig. 1. Multiplier-extended FlexCore processor with a full 90-link interconnect. Here, only 100 bits of 109 total bits of the N-ISA instruction are depicted, since the enable signals of the datapath units are not shown.

	ALU_OpB	ALU_OpA	RF_DataWr	LS_Address	LS_DataIn	BUF1_DataIn	BUF2_DataIn	PC_ControlFB	MULT_OpB	MULT_OpA
MULT_LSB	GPP/Flex	GPP/Flex	GPP/Flex	Flex	Flex	Flex	Flex	Flex	Flex	Flex
MULT_MSB	GPP/Flex	GPP/Flex	GPP/Flex	Flex	Flex	Flex	Flex	Flex	Flex	Flex
PC_ImmPC	GPP/Flex	Flex	Flex	Flex	Flex	Flex	Flex	Flex	Flex	Flex
BUF2_DataOut	GPP/Flex	GPP/Flex	GPP/Flex	Flex	Flex	GPP/Flex	Flex	GPP/Flex	Flex	Flex
BUF1_DataOut	Flex	Flex	Flex	Flex	GPP/Flex	Flex	Flex	Flex	Flex	Flex
LS_Read	GPP/Flex	GPP/Flex	GPP/Flex	Flex	Flex	GPP/Flex	Flex	GPP/Flex	GPP/Flex	Flex
RF_ReadOut1	Flex	GPP/Flex	Flex	Flex	Flex	Flex	Flex	GPP/Flex	Flex	GPP/Flex
RF_ReadOut2	GPP/Flex	Flex	Flex	Flex	Flex	GPP/Flex	Flex	GPP/Flex	GPP/Flex	Flex
ALU_Rslt	GPP/Flex	GPP/Flex	Flex	GPP/Flex	Flex	GPP/Flex	GPP/Flex	GPP/Flex	Flex	GPP/Flex

Fig. 2. Overall format of the configuration file for the interconnect that maps datapath unit outputs to inputs. We use a binary representation to tell whether a link is implemented or not. The 33 links labeled ”GPP/Flex” are used in the databus of a MIPS R2000. The remaining 57 links, marked ”Flex”, are available for design space exploration in the FlexCore processor.

Program Counter (PC). Via an instruction decompressor, a compiler can directly control the datapath [1]. Thanks to the expressiveness of the N-ISA word, the compiler can efficiently schedule applications, at the expense that all applications need to be statically scheduled, since the processor offers no hardware support for dynamic scheduling.

The processor that we consider here is assumed to initially have a very rich interconnect with 90 links, representing interconnect links between the output of any datapath unit to the input of all other units. This *full* interconnect offers maximal freedom for a compiler to route data between the various datapath units, possibly enhancing the execution time over the GPP/MIPS datapath that in the context of interconnect has 33 dedicated links. Given the freedom to route data, a design space exploration methodology could take any set of applications, schedule these onto one datapath configuration and evaluate parameters such as execution time and utilization of the different datapath units and links. Since each application is statically scheduled, it is possible to *exactly* determine what datapath resources will be used. This information can then be back-annotated to the processor design phase to achieve a more optimal implementation, for example, by removing unused or infrequently used links. Sec. IV will provide a more thorough discussion on the topic of design space exploration.

Applications encoded in the wide N-ISA instruction format would require an excessive amount of instruction bandwidth and a large memory footprint. An instruction (de)compression technique based on partitioned look-up tables was developed in earlier work [3]. For a selected set of applications, the technique suggests an appropriate look-up table configuration and content, including how content is updated at run time. The technique was shown to reduce the 109-bit wide instructions to a standard format of 64 bits (that is, by 40%) for the EEMBC [4] *autcor*, *fft*, and *viterb* benchmarks. The 32 immediate bits are not touched by the (de)compression algorithm.

III. FlexTools - The FlexSoC Toolchain

The FlexTools toolchain accepts configuration files that define datapath units and their interconnectivity, and as final output it produces implementation data on GDSII format for chip fabrication as well as the code of the scheduled applications.

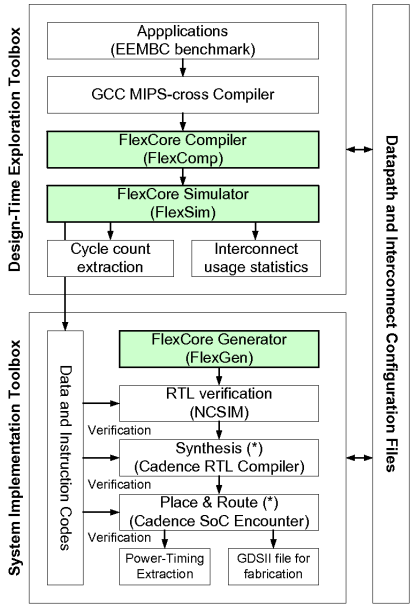
This paper focuses on design space exploration using different interconnect configurations, and an example of such a configuration is shown in Fig. 2. To also give an impression of how a datapath configuration file looks like, the datapath in Fig. 1 is defined by: `ALU=1`, `AGU=0`, `buffers=2`, `read-ports=2`, `write-ports=1`, `mult_delay=2`. Here, PC and LS units are default units, outside the scope of datapath unit reconfigurations.

Fig. 3 shows the overall structure of the toolchain as well as two distinct toolboxes:

A. The Design-Time Exploration Toolbox

The design-time exploration toolbox comprises a compiler, *FlexComp*, and a simulator, *FlexSim*, and provides an environment for exploring performance aspects of different FlexCore processor configurations. The toolbox also relies on external

tools such as GCC MIPS cross compiler, to translate C-code applications into MIPS assembly. The compilation of such applications under different hardware configurations results in different, statically scheduled FlexCore instructions that can be evaluated in FlexSim and profiled with respect to cycle count and utilization statistics of interconnect links and datapath units. Also, instruction and data codes are generated for logic verification of RTL code. The toolbox itself can be viewed as comprising the following tools:



(*) These steps require standard-cell libraries and backend scripts.

Fig. 3. Illustration of the complete FlexTools environment.

1) FlexComp - The Compiler: FlexComp is at the heart of FlexTools and represents vital functionality: Starting from MIPS assembly[1] input files and based on the given FlexCore configuration, FlexComp generates static schedules of FlexCore assembly instructions, called Register Transfer Notation (RTN) instructions and writes these instructions to output files. The scheduling engine plays a pivotal role in the compiler and will therefore be treated in detail under its own heading.

FlexComp relies on the following concepts, to achieve the scheduling operation:

a) Basic Blocks: The FlexCore processor lacks dynamic forwarding, so write-backs become an obstacle when scheduling MIPS instructions. In contrast, a pipelined MIPS architecture buffers the instructions and successively passes them along to the different pipeline stages. This means that when a branch is performed, parts of the instructions preceding the branch may have write-back stages to be performed after the branch has been taken. If there is an instruction in the delay branch slot, that instruction will execute after the branch.

In order to correctly handle write-backs for FlexCore, some write-backs that would have been performed after a branch instruction on a MIPS may have to be performed earlier, possibly before the branch, in the FlexCore. The compiler separates the MIPS code into sequential parts, that is, basic blocks, that have one entry and exit point, and schedules them separately. Scheduling per basic block allows write-backs to be handled independently of instructions outside the basic block.

b) Liveness Analysis: Since scheduling is performed at MIPS assembly level, temporary register manipulations become difficult to track. When scheduling a basic block of MIPS instructions it is, for the purpose of high performance, essential to know if values that are written in that basic block are used outside of the basic block. If the values are not reused, the register write operation may be omitted entirely in favor of other register write operations. The compiler employs liveness analysis that is used for each basic block to find out which registers are read in any successor to the basic block, without first being overwritten. A successor of basic block A is any basic block that can be reached starting from A, and recursively its successors, excluding A.

The liveness analysis was primarily used to prevent write-back to temporary registers. But even that was limited by the low-level nature of the MIPS code. For example, in the case of indirect jumps it was assumed that the successor of the basic block is unknown and thereby assuming that every liveout will be read in the unknown block, which leads to unnecessary register write-backs [5].

2) Scheduler: The scheduler performs the operations detailed below in order to arrive at a schedule of FlexCore assembly instructions that can be executed on the FlexCore configuration presented to it.

a) Extracting Micro Operations from MIPS: The FlexSoC scheme is capable of performing all MIPS operations, but since static scheduling is used, explicit forwarding is required. In the process of converting MIPS instructions, each basic block is handled separately. Only the final value of the registers that are used outside the current basic block needs to be written back to the register file. Other values can be forwarded directly without touching the register file to reduce congestion.

The MIPS instructions are processed in sequence and for each instruction's input register, the latest value of the register is used, that is, either the original value or a link to the operation that generated the updated value. By doing this, a tree of all the direct dependencies is being built. In addition to the direct dependencies, any register write should occur only after the previous read instructions, which require the old register value, are completed. Memory writes will occur in the same order after the transformation, but memory reads are free to reorder in between writes. This is a conservative rule to make sure memory accesses behave the same after transformation. This constraint will later be relaxed when possible, by using locally available information to differentiate unique memory addresses.

Because of the interconnect flexibility of the FlexSoC scheme, values can move directly from registers or buffers to datapath units without passing through the ALU. Another improvement is the elimination of trivial address calculations,

[1]The GCC MIPS cross compiler is used at the -O2 optimization level to evaluate FlexComp features.

like when the offset is zero or the address can be loaded directly as a 32-bit immediate. If the same immediate value or address calculation is needed twice, it only needs to be generated once. The baseline FlexCore processor uses the datapath units of a simple MIPS processor, but by removing unneeded micro operations and by using direct forwarding the performance can be improved. The performance gain comes from both exposing the datapath to software control, to improve utilization, and using a richer interconnect in the datapath to allow for more efficient routing.

b) Ordering: When scheduling a basic block, the goal is to create a compact schedule. The scheduling is done by greedily selecting as many instructions as possible according to a priority function, while using each limited resource, such as ALU, memory and multiplier, only once [6]. An instruction can only be scheduled if all the instructions it depends on are already scheduled. Move instructions take no time to perform: So long as the move can be fulfilled in a cycle, all additional instructions that only needed that move can be performed in the same cycle. Similarly, the old value of a register can be read in the same cycle as the new value is written. Because instructions are scheduled linearly, the order in which they are chosen greatly affects the quality of the resulting schedule. Currently the heuristic used is to prioritize the instruction with the least mobility, which is the instruction that has the most predecessor and successor levels combined.

c) Checking Constraints Using a SAT-solver: Scheduling instructions on the FlexCore processor is far from trivial. Because of the flexible architecture, the number of possible solutions is very large. The question of when and where to buffer and read data to make sure that all operations are satisfied quickly becomes complex. The scheduler expresses dependencies between instructions as *routing* constraints and employs a SAT-solver [7], [8], [9] to find a solution, if one exists.

The SAT-solver is used by incrementally adding more constraints and in the circumstance that a specific schedule solution fails, all associated constraints are removed leaving all previous constraints intact. Many values are forwarded directly between operations but when buffering is needed, we formulate constraints that force a particular value to be stored in one of the buffers or in the register file. If a value needs to be written permanently to the register file, it is written right away after it has been produced. Register reads and immediate values can be buffered if needed, to avoid resource conflicts that would arise otherwise. After scheduling all operations and making sure (via the SAT-solver) all variables can be routed within a minimum number of cycles, there are still a huge number of solutions within the constraints. The default SAT-solver solution may be far from a good solution, in terms of, for example, power dissipation, since it may cause an unnecessarily high amount of data traffic. We solve this by iteratively constraining all input values as much as possible to reduce unnecessary buffering.

The approach with SAT-solver based scheduling combined with the current heuristic suffers when the basic blocks become large. Unfortunately, this is also where the scheduling can do really well (in terms of execution-time performance) since unnecessary write-backs required before jumps are less of an issue and there are many more possibilities for parallelism than in smaller blocks. If we disregard caches and memory, then loop unrolling and inlining could really improve the execution time but could likely increase the compilation time considerably. Loop unrolling and inlining can always be limited if need be, but still, the performance benefits after compilation limits this. In addition, when resources (interconnect links and datapath units) are restricted to approach the minimum for allowing applications to be scheduled at all, then really bad scheduling run times were observed [5].

3) FlexSim - Simulator and Assembler: The FlexSim simulator has two modes of operation: MIPS simulation and simulation of the FlexCore RTN format, which is produced by the FlexComp compiler:

- FlexCore configuration-dependent simulator: The simulator has to be general enough to support a range of datapath configurations. The simulator approaches generality by modeling a fully interconnected FlexCore processor with a superset of datapath units. The compiler is responsible for the handling of interconnects and units at disposal, so that the generated code will be simulated correctly, as if run on the intended FlexCore processor. Interconnect links and datapath units that are not included in the intended FlexCore processor should therefore not be exercised, when the compiled FlexCore RTN code is run in the simulator.
- MIPS reference simulator: When directly simulating MIPS assembly, the simulator employs a 4-stage N-ISA buffer to obtain a MIPS-like pipelining of instructions[2]. When using the N-ISA buffer, it is straightforward to convert from MIPS assembly to FlexCore RTN code, since each MIPS instruction is converted to four FlexCore RTN instructions; one for each stage of the N-ISA buffer. The N-ISA buffer stores operations that will be carried out in the future similar to the MIPS pipeline registers. Thanks to the capability to simulate MIPS code on the FlexCore processor, we obtain a reference to the statically scheduled code of FlexComp.

The FlexSim simulator allows for tracing of the execution in a cycle-by-cycle manner and exposes the internal control signals of the FlexCore datapath. Furthermore, FlexSim has some functionalities for profiling applications, such as *i)* interconnects usage statistics that express how frequently interconnect links between datapath units are used, *ii)* mixed binary/RTN instruction format for debug, *iii)* operation traces for datapath units and memory, and *iv)* cycle-count and code-size extraction for individual functions used within applications.

The FlexSim tool also contains an assembler that is responsible for linking and producing binary outputs (Code/Data) that can

[2]FlexCore assembly is simulated without the N-ISA buffers.

be run on a FlexCore hardware implementation. The FlexSim assembler can produce two different code formats: *i)* Statically scheduled code words organized in 4×N-ISA length for use with the N-ISA buffer, or *ii)* N-ISA code words, also statically scheduled, for the FlexCore processor.

B. The System Implementation Toolbox

The system implementation toolbox (Fig. 3) has a frontend generator that can create behavioral (testbench) and structural (datapath) VHDL code. FlexGen generates a top-level VHDL module and an associated testbench for a particular FlexCore processor based on the specified datapath and interconnect configurations. All datapath units have been implemented and verified in advance, and their RTL code blocks reside in a code repository. The generator eliminates the hassle of error-prone VHDL coding to create FlexCore processors, and it can easily be extended to accommodate new units in a modular fashion. Not only RTL for ASIC implementation is available, but also an RTL version adjusted to FPGAs exists. The backend of the toolbox is based on a Makefile that chains several commercial EDA tools, such as NCSIM, RTL compiler, and SoC Encounter of Cadence to enable timing-driven physical implementation and verification, and to automatically extract power dissipation based on actual application traces.

1) FlexGen - The RTL Generator: Due to its fine-grained flexibility, the FlexSoC scheme mainly targets ASIC platforms. Thus, the FlexCore datapath units are implemented for an ASIC with high performance and low power dissipation in mind. We have implemented a number of elementary datapath units, including different adder architectures and multipliers with various number of pipeline stages. The FlexSoC scheme also support circuits with run-time adjustable computational precision, such as the Twin-Precision multiplier [10] and the Double-Throughput MAC accelerator [11]. By offering a rich set of datapath units, we improve the potential for high performance and energy efficiency, but only if we can perform complex processor design space exploration in an efficient manner.

In order to integrate the units into the FlexCore processor in a rational way, the pre-optimized RTL code of the datapath units is defined with pipeline registers on the output. Based on the configuration files for datapath units and interconnect links, the RTL generator, *FlexGen*, instantiates datapath units together with interconnect definitions to generate RTL code for the top, architectural level of a FlexCore processor.

IV. Design Space Exploration and Results

A. Exploration Flow

The FlexTools environment is an integral part of the FlexCore design space exploration methodology (Fig. 4) that will be described in the following. In this first paper on FlexTools, we will limit the exploration in the dimension of datapath units; only the datapath-unit configuration in Fig. 1 will be evaluated. Instead we will focus on design space exploration assuming a *variety of benchmarks* running on a FlexCore processor in which we *vary what interconnect links are available*.

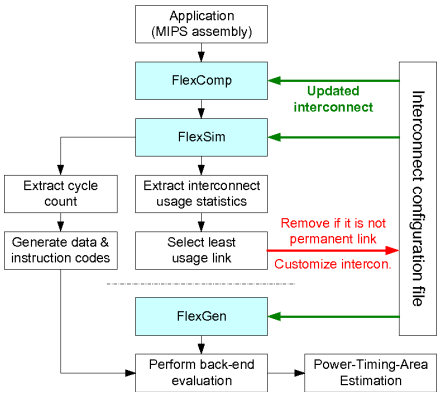


Fig. 4. Methodology flow for design space exploration of different FlexCore interconnect configurations.

Our overriding implementation intent is to minimize energy, by reducing the hardware complexity while maintaining the short execution time offered by the flexible interconnect. In a FlexCore datapath, energy savings are mainly a result of reduced cycle count. However, the other parameter of energy, that is power dissipation, can be reduced as we remove links in the interconnect. This is because less links can translate into both less leakage and less switched capacitance, since switchbox multiplexors and wires are removed and timing constraints are relaxed for individual gates.

There exist three levels of accuracy in our exploration methodology:

- Rapid exploration at the architectural level entailing evaluation of execution time (#cycles), execution profiles for the applications, etc.
- A more accurate, slower exploration based on logic simulation of synthesized RTL code. Gate-level timing information is now available and power dissipation information is accurate in the sense that application traces are available for switching power estimation.
- Exploration using the full backend implementation down to physical level, in which all circuit details (including wiring) are available.

The design space exploration shown here initially assumes the full interconnect, which is the 90-link configuration for the datapath-unit configuration in Fig. 1. To showcase the span of our exploration methodology, we use the most accurate level, which is most time consuming. To keep the exploration time down, we iteratively remove five links at a time.

To expose the datapath as well as the methodology to a diverse range of benchmarks, we use nine integer EEMBC benchmarks: *autcor*, *conven*, *fft*, *viterb*, from the Telecom suite, *rgbcmy*, *rgbhpg*, *rgbyiq* from the Consumer suite, and *aifirf*, *bitmnp* from the Automotive suite. Other EEMBC benchmarks are not considered, since they require floating-point computations, which are currently not supported by the FlexCore datapath. As a reference architecture, we run the benchmarks using our simulator for a MIPS datapath configuration.

	ALU_OpB	ALU_OpA	RF_DataWr	LS_Address	LS_DataIn	BUF1_DataIn	BUF2_DataIn	PC_ControlFB	MULT_OpB	MULT_OpA
MULT_LSB	1	2359	3082	0	0	1024	0	0	0	2056
MULT_MSB	0	0	0	0	0	0	0	0	0	0
PC_ImmPC	713936	4	34898	7795	99	4676	1889	0	0	257
BUF2_DataOut	9884	33801	22916	16418	4792	28232	0	3220	3	8
BUF1_DataOut	5105	30277	23893	9871	2059	0	26012	4128	1	2051
LS_Read	16840	40149	27947	18	3821	10144	1213	7780	8	2347
RF_ReadOut1	69986	266502	16653	40514	18040	6008	5168	14526	4410	5
RF_ReadOut2	212627	354698	14343	29997	12548	5713	6377	557	2305	3
ALU_Rslt	16442	317031	429161	56688	24669	42688	70972	192062	0	0

Fig. 5. Utilization statistics of the full, 90-link interconnect configuration. There are 20 links (shown in green/dark gray frames) that are never exercised and thus can be removed.

	ALU_OpB	ALU_OpA	RF_DataWr	LS_Address	LS_DataIn	BUF1_DataIn	BUF2_DataIn	PC_ControlFB	MULT_OpB	MULT_OpA
MULT_LSB	1	2359	2059	0	0	2048	0	0	0	2056
MULT_MSB	0	0	0	0	0	0	0	0	0	0
PC_ImmPC	713936	4	34898	7795	99	5198	1367	0	0	257
BUF2_DataOut	3057	57902	24953	11799	3005	16242	0	287	200	1026
BUF1_DataOut	10079	62306	33275	7049	1177	0	18637	7060	1	1035
LS_Read	16840	40149	26405	18	3821	6008	5768	7780	8	2347
RF_ReadOut1	30236	335550	19386	38904	15074	6490	8534	5108	1033	5
RF_ReadOut2	254230	229520	11586	39048	18183	3513	5212	9976	5485	1
ALU_Rslt	16442	317031	408053	56688	24669	87997	74831	192062	0	0

Fig. 6. Utilization statistics of the 70-link interconnect configuration. Green/dark gray frames are links that can be removed, to obtain a 65-link configuration.

	ALU_OpB	ALU_OpA	RF_DataWr	LS_Address	LS_DataIn	BUF1_DataIn	BUF2_DataIn	PC_ControlFB	MULT_OpB	MULT_OpA
MULT_LSB	1	2359	3083	0	0	1024	0	0	0	2056
MULT_MSB	0	0	0	0	0	0	0	0	0	0
PC_ImmPC	713944	4	34898	7798	99	4682	1872	0	0	257
BUF2_DataOut	2398	77704	30225	10673	4162	20840	0	2936	3	1029
BUF1_DataOut	3987	48346	20173	9696	2835	0	27832	3091	0	1035
LS_Read	16838	40147	27636	0	3820	2337	8299	7780	0	2350
RF_ReadOut1	33400	325352	17579	44983	19885	5456	8158	1575	4106	0
RF_ReadOut2	257811	233880	13464	31463	10558	8484	5943	14829	2618	0
ALU_Rslt	16442	317029	404500	56688	24669	71287	88839	192062	0	0

Fig. 7. Utilization statistics of the 65-link interconnect configuration. Green/dark gray frames are links that can be removed, to obtain a 60-link configuration.

We now run FlexTools and obtain the interconnect utilization statistics for the specified interconnect configuration; see Fig. 5. We notice that there are 21 links that are never used in any benchmark; these are represented as framed zero values. It is tempting to remove these from the datapath. However, these link candidates must be checked before removal, to guarantee that they are not belonging to the permanent links. Permanent links are those that constitute robust architectures. For example, at least one of the links between the LS_Address input of the LS unit and one of the following four, RF_ReadOut1, RF_ReadOut2, BUF1_DataOut, and BUF2_DataOut, must exist to ensure access to the LS unit. The most significant 32-bit field of the multiplier has not been accessed by any benchmark. Thus, from a utilization statistics point of view, the MULT_MSB port is not connected to any other datapath unit. However, we need to keep at least one link[3] from MULT_MSB to guarantee that the FlexCore datapath is compatible with the MIPS architecture.

The result of this first stage of design space exploration is that we safely can take away 20 links (those in frames in Fig. 5) and obtain a 70-link FlexCore configuration (Fig. 6) that is 7% more area efficient. Considering power dissipation when going from 90 links to 70 links, the hardware reduction yields a 4.4% power reduction over the nine benchmarks, as shown in Table I.

Returning to Fig. 6, the second stage of the exploration is to use the 70-link configuration as the starting point for further link removals, down to 65 links. Five links that are rarely used are framed in Fig. 6. We eliminate these links by editing the interconnect configuration file, to achieve a 65-link configuration.

Similarly, the third stage is to extract the interconnect utilization statistics of the 65-link configuration. From this we can identify a promising 60-link configuration, in which the framed links of Fig. 7 will be eliminated. To illustrate the FlexTools backend, a die graph of a 65-nm layout of this configuration is shown in Fig. 8.

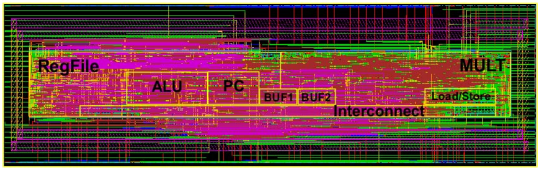


Fig. 8. Die graph of a 60-link 65-nm FlexCore datapath.

B. Exploration Results

So far we consider all nine benchmarks when performing exploration. The cycle count, power and energy dissipation of the respective benchmark for some of the examined FlexCore interconnect configurations are reported in Table I. We also normalize the average values of cycle count and energy dissipation of all configurations to the GPP/MIPS datapath and plot these in Fig. 9.

[3]The yellow/light gray link MULT_MSB to RF_DataWr will always be preserved during exploration.

TABLE I
CYCLE COUNT, POWER AND ENERGY DISSIPATION OF THE VARIOUS FLEXCORE INTERCONNECT CONFIGURATIONS FOR A 65-NM PROCESS TECHNOLOGY.

Parameter	Configuration	EEMBC benchmark									
		autcor	fft	conven	viterb	rgbcmy*	rgbhpg*	rgbyiq*	aifirf	bitmnp	Average
Cycle count	Flex 90-link	16110	136596	262093	265291	62310	19084	34069	24620	92658	101426
	Flex 70-link	16110	136596	262093	265291	62310	19084	34069	24620	92658	101426
	Flex 65-link	16111	136597	262094	265293	62312	19085	34071	24626	92659	101428
	Flex 60-link	16118	136860	262100	265299	62318	19091	34077	24632	92665	101462
	Flex 55-link	16218	137884	262100	265299	62318	19091	34077	24833	92665	101609
	Flex 40-link	17170	139995	263415	266543	63107	19898	34864	26315	93520	102759
	MIPS 33-link	21053	172381	278790	290744	68626	23479	38955	33683	102338	114450
Power (mW) at 383 MHz	Flex 90-link	8.94	10.64	8.87	9.29	8.97	8.93	9.14	9.90	9.03	9.30
	Flex 70-link	8.33	10.19	8.52	9.28	8.63	8.46	8.87	9.49	8.26	8.89
	Flex 65-link	8.26	10.24	8.41	9.22	8.65	8.39	8.85	9.30	8.16	8.83
	Flex 60-link	8.22	10.23	8.30	9.05	8.51	8.31	8.81	9.54	8.02	8.78
	Flex 55-link	8.73	9.81	8.88	9.52	8.51	8.82	9.06	9.80	8.15	9.03
	Flex 40-link	9.24	10.17	9.26	9.82	9.10	9.31	9.65	10.10	9.06	9.52
	MIPS 33-link	8.48	9.48	9.09	9.07	8.71	8.87	9.09	9.19	8.45	8.93
Energy (μJ)	Flex 90-link	0.38	3.79	6.07	6.43	1.46	0.44	0.81	0.64	2.18	2.47
	Flex 70-link	0.35	3.63	5.83	6.42	1.40	0.42	0.79	0.61	2.00	2.38
	Flex 65-link	0.35	3.65	5.76	6.38	1.41	0.42	0.79	0.60	1.97	2.37
	Flex 60-link	0.35	3.65	5.68	6.27	1.38	0.41	0.78	0.61	1.94	2.34
	Flex 55-link	0.37	3.53	6.08	6.59	1.38	0.44	0.81	0.64	1.97	2.42
	Flex 40-link	0.41	3.72	6.37	6.83	1.50	0.48	0.88	0.69	2.21	2.57
	MIPS 33-link	0.47	4.26	6.62	6.88	1.56	0.54	0.92	0.81	2.26	2.70

* The benchmark's input data size was reduced to relax memory requirements.

Averaged over all benchmarks, all FlexCore configurations use fewer cycles than the MIPS configuration. In the best case, cycle wise, the 60-link FlexCore configuration offers a 15.1% reduction in cycle count. In terms of energy dissipation, the 60-link configuration dissipates 16.8% less energy than the MIPS configuration. The 40-link FlexCore configuration represents the weakest compromise with a mere 7.1% reduction in cycle count and 12.8% energy reduction.

When moving from 60 links to 55 links, the energy in Fig. 9 starts to increase while cycle count remains relatively constant. The five links that were removed relate to routing outputs of the register file, buffer and the LS unit to the multiplier, and forwarding outputs of the multiplier to its inputs. On one hand, removing forwarding paths increase cycle count by a small amount but, at the same time, the remaining links that are connected to the multiplier are more intensely used than before, increasing multiplier power dissipation.

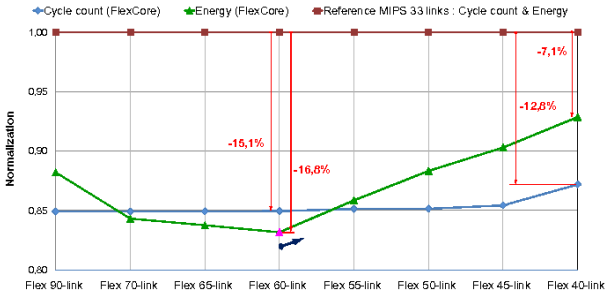


Fig. 9. Relative cycle count and energy dissipation for the various 65-nm datapath configurations (neglecting instruction decoding/decompression).

Finally, we notice that the values provided are the averages over several benchmarks, where several larger ones (viterb and conven) are dominant. Were we to optimize the FlexCore datapath for a narrow application domain, such as *autcor*, the energy reduction is at least 25% as seen in Table I. Optimization of the interconnect configuration to this single application is likely to improve the energy efficiency further.

V. RELATED WORK

There are some related projects that more or less share the same point of view as the FlexSoC scheme; the No-Instruction-Set-Computer (NISC) project [12] comes closest in concept. In this work, only control signals of the *local* interconnect paths between processing elements in the execution stage or the register file are investigated, and utilized according to the application profile. As a result, datapath control signals are not fully exposed (for example, those of the PC unit) to the compiler, leading to limited design space exploration. The NISC generates a datapath (allocates both units and their interconnects) adapted to the Control-Data Flow Graph (CDFG) [13], which is extracted from application execution, instead of reconfiguring interconnect between datapath units. Both NISC and FlexSoC are inspired by VLIW architectures, allowing the compiler more opportunities to schedule instructions with a high degree of parallelism. But unlike FlexSoC however, the information pertaining to interconnects between the datapath units is completely rejected in NISC, since VLIW traditionally uses dedicated interconnects.

In [14], a compiler-based framework has been introduced for energy-aware design space exploration. In this scheme, the power dissipation of datapath units is characterized after place-and-route and provided to the compiler to allow fast estimation of energy and performance. However, this scheme assumes dedicated interconnects. Another attempt to combine the flexibility of FPGAs and high-performance of ASICs is FlexASIC [15], in which many ASIC components communicate using an FPGA fabric. Under compiler guidance, the number of required ASIC components and their interconnect can be optimized to reduce number of masks required to configure the fabric. In contrast to FlexSoC, additional hardware is needed here to enable flexibility.

The work in [16] investigates compiler-assisted energy optimization by exploiting instruction slack as a more efficient means of functional unit mapping, for a VLIW cluster. Mei *et al.* [17] propose a C-based methodology to design VLIW/reconfigurable matrix to achieve flexibility. These contributions independently span two parts of the problem our present work addresses. In contrast to the works cited, FlexTools addresses application efficiency, while at the same time being capable of leveraging energy benefits offered by ASICs. Zhang *et al.* [18] independently combine the techniques similar to the ones in [16] with traditional circuit-based techniques, like power gating, to achieve energy efficiency. FlexTools provides an integrated toolchain that helps achieve optimal results at design time.

VI. Future Directions

The exploration presented here represents the first time the complete FlexTools environment has been available. This in turn opens up a number of avenues for further improvements:

- The instruction decompressor is available for the fully interconnected FlexCore. For FlexCore datapaths using other interconnect configurations, the decompressor configuration changes too [3]. Efforts carried out in parallel to this work target the automated generation of an instruction decompressor for different configurations. Integration of this into the toolchain will further enhance the ability to perform exploration of a complete FlexSoC system.
- In the explorations presented here no accelerators were used. Work is in progress to extend the suite of accelerators available, including power-gated and multi-precision circuits, to improve the computational efficiency of the FlexCore datapath further.
- It is also possible to vary parameters of the datapath units as suggested by the datapath-unit configuration in Sec. III. No such variation was used here in order to keep the comparison with the MIPS reference consistent. This represents another avenue to leverage flexibility.
- FlexTools was developed over a period of time, addressing issues as they arose. Presently an effort has been initiated to address the possibility of using a more advanced frontend like that of LLVM, to obtain scheduled code in a more standardized fashion.

VII. Conclusion

One major benefit of the FlexSoC scheme is the increased flexibility made possible by the interconnect. The drawback of flexibility in view of wide instructions is that instructions become harder to compress. However, as multiple standard instructions are mixed together into a micro-operation flow, fine-grained control of all aspects of the hardware is made possible. Exposing this flexible datapath to the compiler then further leverages any benefit a target application domain has to offer.

We have presented FlexTools; a toolchain that, together with the flexible FlexCore processor architecture, enables design space exploration of embedded processors. The framework has been shown to produce native instructions from MIPS assembly that compares well, cycle-wise, against the MIPS reference platform for different FlexCore configurations. The opportunity to perform design space exploration with the aim of energy efficiency, within the framework of a single toolchain, also leads to improved productivity without compromising quality.

References

[1] M. Thuresson, M. Själander, M. Björk, L. Svensson, P. Larsson-Edefors, and P. Stenstrom, "FlexCore: Utilizing Exposed Datapath Control for Efficient Computing," *Springer Journal of Signal Processing Systems*, vol. 57, no. 1, pp. 5–19, 2009.

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

[3] M. Thuresson, M. Själander, L. Svensson, and P. Stenstrom, "A Flexible Code Compression Scheme using Partitioned Look-Up Tables," in *Int. Conf. on High Performance Embedded Architectures and Compilers*, Jan. 2009, pp. 95–109.

[4] Embedded Microprocessor Benchmark Consortium. [Online]. Available: http://www.eembc.org

[5] U. Jälmbrant and E. der Hagopian, "Improved Configurability with FlexSoC," MSc Thesis, Chalmers University of Technology, March 2009.

[6] T. Schilling, M. Själander, and P. Larsson-Edefors, "Scheduling for an Embedded Architecture with a Flexible Datapath," in *IEEE Computer Society Annual Symp. on VLSI*, May 2009, pp. 151–156.

[7] Mini SATisfiability (MiniSAT). [Online]. Available: http://minisat.se

[8] G. Cabodi, L. Lavagno, M. Murciano, A. Kondratyev, and Y. Watanabe, "Speeding-Up Heuristic Allocation, Scheduling and Binding with SAT-Based Abstraction/Refinement Techniques," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 15, no. 2, pp. 1–34, 2010.

[9] S. Memik and F. Fallah, "Accelerated SAT-Based Scheduling of Control/Data Flow Graphs," in *IEEE Int. Conf. on Computer Design*, 2002, pp. 395–400.

[10] M. Själander and P. Larsson-Edefors, "Multiplication Acceleration Through Twin Precision," *IEEE Trans. on Very Large Scale Integrated Systems*, vol. 17, pp. 1233–1246, Sept. 2009.

[11] T. T. Hoang, M. Själander, and P. Larsson-Edefors, "Double Throughput Multiply-Accumulate Unit for FlexCore Processor Enhancements," in *IEEE Int. Symp. on Parallel and Distributed Processing*, May 2009, pp. 1–7.

[12] B. Gorjiara and D. Gajski, "Automatic Architecture Refinement Techniques for Customizing Processing Elements," in *Design Automation Conf.*, 2008, pp. 379–384.

[13] M. Reshadi, B. Gorjiara, and D. Gajski, "C-Based Design Flow: A Case Study on G.729A for Voice over Internet Protocol (VoIP)," in *Design Automation Conf.*, 2008, pp. 72–75.

[14] P. Raghavan, A. Lambrechts, J. Absar, M. Jayapala, F. Catthoor, and D. Verkest, "COFFEE: COmpiler Framework for Energy-Aware Exploration," in *Int. Conf. on High Performance Embedded Architectures and Compilers*, 2008, pp. 193–208.

[15] J. L. Wong, F. Kourshanfar, and M. Potkonjak, "Flexible ASIC: Shared Masking for Multiple Media for Multimedia Processors," in *Design Automation Conf.*, 2005, pp. 909–914.

[16] R. Nagpal and Y. N. Srikant, "Compiler-Assisted Leakage Energy Optimization for Clustered VLIW Architectures," in *ACM & IEEE Int. Conf on Embedded Software*, 2006, pp. 233–241.

[17] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study," in *Design, Automation and Test in Europe*, 2004, pp. 1224–1229.

[18] W. Zhang, Y.-F. Tsai, D. Duarte, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Reducing Dynamic and Leakage Energy in VLIW Architectures," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 1, pp. 1–28, 2006.