

## EPIC-ADAPTIVE EPIC BRIDGE

**Valentin Stefan Gheorghita<sup>1</sup>, Weng-Fai Wong<sup>2</sup>, Oana Florescu<sup>1</sup>**

<sup>1</sup> *Dept. of Computer Science, "Politehnica" University of Bucharest*

<sup>2</sup> *Dept. of Computer Science, National University of Singapore*

**Abstract:** Extending a generic architecture with a dynamically re-configurable structure offers to embedded systems designers the flexibility of application specific optimizations. This paper is concerned with a fine-grain, tightly coupled, dynamically reconfigurable architecture called Adaptive EPIC, previously introduced in (Talla, 2000). The paper investigates the granularity of the application part implemented on the reconfigurable structure of the AEPIC. The results show that significant performance improvements are obtained for a fine-grained partitioning of the application.

**Keywords:** Re-configurable computing, adaptive EPIC, processor simulation, patterns recognition.

### 1. INTRODUCTION

As far as embedded processing is concerned, there are two ways for improving performance: one consists of selecting the processor that meets the characteristics of the application and the constraints of cost and then optimizes the intended application for the selected processor; the other one, given the application, optimizes the hardware it would be executed on. The first approach usually means assembly programming of the application core sections. The second one allows hardware designers a better control over the design of the architecture and a better meeting of the optimization criteria. However, engineering a new processor is often a very expensive way.

The tightly coupled re-configurable processors introduce and offer a new degree of freedom in the design space. They have cheaper cost designs, combining existing off-the-shelf silicon with the flexibility of optimizing parts of the hardware for specific applications.

Adaptive EPIC (Explicitly Parallel Instruction Computing) is a dynamically re-configurable processor architecture, introduced by (Talla, 2000), whose basic design consists of a segmented re-configurable array tightly coupled with an EPIC processor. This architecture combines the advantages of the EPIC, which has a simple architecture backed by well-known compiler technology, and that of programmable logic that exploits fine-grain parallelism through explicit control over micro-architectural features.

The paper describes briefly the design of the architecture and considerations regarding the granularity of an application parts that are to be run on the re-configurable structure. The contributions of the present work are the following: presents small patterns that can be found in an application and that are suitable for being executed on the re-configurable part of the AEPIC architecture; simulation results on embedded benchmarks that emphasize the performance improvements obtained for a fine-grained versus a course-grained partitioning of an application.

## 2. PREVIOUS WORK

Estrin (1960) proposed and implemented the earliest known computing system based on re-configurable devices. It was a hybrid machine consisting of a general-purpose processor interconnected with high-speed logic devices, which were reconfigured manually. In the mid 80's Xilinx introduced Field Programmable Gate Array (FPGA) devices, which spurred the research in the area of FPGA based re-configurable computing engines. PRISM (Athanas and Silverman, 1993), PAM (Vuillemin, *et al.*, 1996) and SPLASH-2 (Gokhale, *et al.*, 1991) are pioneering efforts in this direction. Recently, researchers have explored variations of FPGA architectures (DPGA and MATRIX (DeHon, 1996)) and also some new architecture combining standard processors with re-configurable logic. These devices' target is the embedded market, as the processor cores chosen to go with the programmable logic show (low frequency, hence low power consumption). Some released or announced devices of this kind are: Triscend's E5 and A7, Atmel's FPSLIC and Chameleon Systems' CS2000. These new devices use fine-grained re-configurable array and the FPGA logic usually operates on one or two bit wide data.

## 3. ADAPTIVE EXPLICITLY PARALLEL INSTRUCTION COMPUTING

Explicitly Parallel Instruction Computing (EPIC) architectures take advantage of statically scheduled instruction level parallelism. Adaptive Explicitly Parallel Instruction Computing (AEPIC) architectures are EPIC processors with additional re-configurable components amenable to compiler optimizations. The interface observed by a program executing on the AEPIC architecture in any cycle of the machine is that of an explicitly scheduled EPIC architecture. Only the number and types of instructions that can be executed in a machine cycle can vary. The code generated for a given application contains the decisions of when and how to reconfigure the programmable logic to implement application specific instructions.

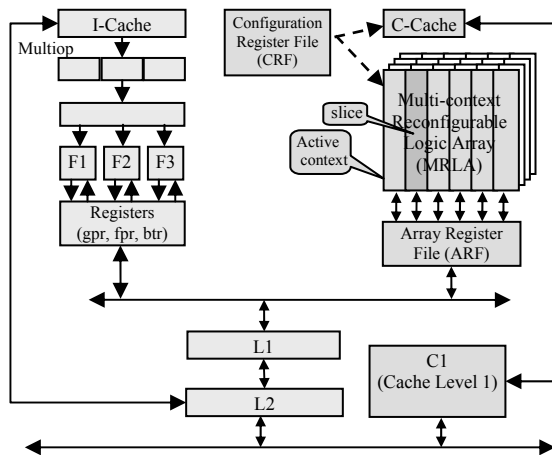


Fig. 1. AEPIC Architecture.

### 3.1 AEPIC Architecture

The abstract architecture of an AEPIC machine is shown in figure 1. The core component of the AEPIC architecture consists of a standard EPIC machine. The adaptive component of the AEPIC processor consists of the *Configuration Cache Hierarchy*, *Multi context Reconfigurable Logic Array (MRLA)*, *Configuration Register File (CRF)* and *Array Register File (ARF)* connected together via bus interconnect. On the programmable MRLA there are implemented the new added functional units, named *Configured Functional Units (CFUs)*. A standard memory hierarchy (two levels of cache) is used in order to provide a rapid means of instantiating CFUs whose configurations are held as data in the standard memory hierarchy of the system. Every configuration loaded in cache or MRLA has a configuration register assigned to it. The data is passed from the EPIC processor to the new added CFUs using registers from Array Register File. They are a set of general-purpose registers, floating point registers, predicate registers and branch target registers.

Like a typical Field Programmable Gate Array (FPGA) the MRLA is a two dimensional structure that is composed of programmable logic and interconnect blocks. The term *programmable element (PE)* is used to refer both the *programmable logic block* as well as the *programmable interconnect block*. A *configuration instruction* determines the behavior of each PE. CFUs are sets of logic designs implemented by means of configuration instructions.

Unlike FPGA, MRLA permits multiple contexts to be present in the logic area. For this, each PE has associated multiple configuration instructions. All PE have the same number of configuration instructions slots stored in an ordered sequence. At any moment, all PE have the same index of configuration instruction activated, and these instructions represent the MRLA active context.

### 3.2 Key Features of the Architecture

AEPIC architecture provides some interesting features in the context of re-configurable computing.

AEPIC delegates to the compiler the task of specifying which parts of the application will be executed on the re-configurable structure and when to allocate and de-allocate the needed resources. On the other hand, the main processor decides where to host that computation on the programmable logic resource. On any clock cycle a particular set of CFUs (context) is considered active and these are the only CFUs that can execute operations. The architecture also permits a quick changing (one cycle) of the active context.

Unlike typical RISC operations, most of the instructions performed by CFUs take a large number of input/output operands. In order to keep the instruction format as simple as possible, the operands

are not specified as part of the instruction, but assigned to specific registers, which are associated as sources/destinations for the instruction.

Another feature of the AEPIC is the explicit control over the configuration cache hierarchy. This feature plays a significant role in AEPIC processing where the costs of configuration cache misses can be more expensive than the conventional data cache misses. This explicit control is supposed to be feasible as the applications usually have a number of configurations smaller than the number of program values.

In addition to the above-mentioned features, AEPIC inherits from EPIC speculative execution, predication, decoupled branch architecture, efficient boolean reductions, compiler controlled cache behavior (Kathail, *et al.*, 1994).

### 3.3 Programming AEPIC

Programming a CFU consists of two steps. At first the CFU must be configured and then it can be used. The code for the CFU configuration is as follows:

Line	Code
1	calloc cr, reg
2	malloc cr, cid
3	incr cr
4	inp cr, ar, lit
5	outp cr, ar, lit

This code will allocate space and registers for a configuration and load it in the C-Cache and MRLA. The adequate number of blocks in the C-Cache is allocated with *calloc* (line 1). The address of the memory location is pointed to by *reg* and the configuration register *cr* is associated to the new configuration. The *malloc* instruction in line 2 allocates the required number of slices on MRLA (a slice is the allocation unit for MRLA) on the context specified by *cid* for the configuration associated with *cr*. The information stored in *cr* provides the number of slices needed by the CFU. The configuration data associated with *cr* is transferred from C-Cache to MRLA (line 3). Instructions in lines 4 and 5 associate to the configuration, input and output registers from the ARF. There can be several input/output registers for each configuration. *Lit* specifies the index of the input/output register of the configuration and *ar* specifies the associated register from ARF. Also, the same register can be associated to several configurations.

For the usage of a CFU the following code sequence is performed:

Line	Code
1	/* transfer data to input registers */
2	setctx cid
3	exec cr, opid
4	wtc cr
5	/* transfer data out of output registers */

Before the use of a CFU, the necessary input data must be placed in its input registers. If the CFU is not in the current context, the context that contains the configuration to be executed becomes active (line 2). The code in line 3 triggers the execution of the CFU associated with *cr* on the given input. The main processor can execute instructions in parallel with the CFU, but in order to collect the results from CFU, the *wtc* instruction must be performed. This instruction stalls the main processor until the CFU execution is completed. Finally, the output of the CFU can be used.

## 4. LARGE vs. SMALL CFUs

From the programming point of view a CFU represents a piece of sequential code, which runs in one or more cycles. As the input, it takes the data from the input registers, processes it, and puts the results into the output registers. All these registers are from the ARF.

### 4.1 CFU Designs comparison

The AEPIC speed is affected by the core speed (which is fixed), the CFUs speeds and the overhead added by the communications between the main processor and CFUs. Because the core and CFUs do not have the same speed (always core speed is higher than CFUs'), a CFU clock cycle may mean one or more main processor clock cycles. It is assumed that the CFUs frequencies are divisors of the core frequency.

The CFU clock frequency is obtained after placement and routing of the portion of the code identified for execution in the re-configurable part of the processor. It is determined by the critical path of the design that is affected by the complexity of the circuit. Our experiments showed that the clock frequencies for designs that take 3 to 10 cycles for execution are between 15 to 30 MHz. For simple designs, which take one or two cycles, the clock frequency obtained was up to 120 MHz.

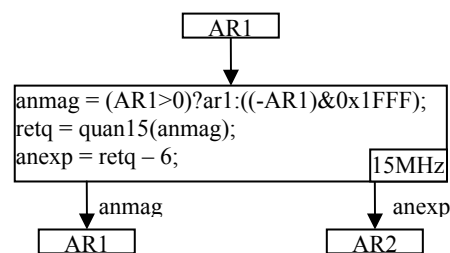


Fig. 2. CFU for the core of G721 decoder.

The figure 2 represents a part of the code of G721 audio decoder algorithm from the MediaBench suite (Lee, *et al.*, 1997). For this piece of code the design takes 3 cycles at a clock frequency of 15MHz. Also, the overhead added by the communication for every CFU execution is three main processor cycles (one for moving data into the input registers, one cycle for CFU starting the execution with *exec* instruction, and

one for retrieving the results from the output registers).

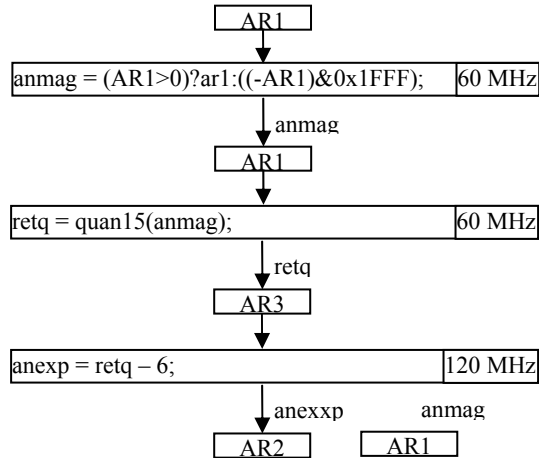


Fig. 3. A sequence of 3 CFUs for the core of G721.

Another implementation for the previous presented code may be done using a sequence of 3 circuits. The figure 3 presents the code for every circuit and the communications between them. The clock frequencies obtained were 60, 60, respectively 120MHz. Also, the communication overhead is 5 cycles (one for moving data into the input registers, three cycles for CFUs execution with *exec* instruction, and one for retrieving the results from the output registers). The communication between CFUs is realized using the ARF, and this does not add any overhead.

#### 4.2 CFU Code Patterns

Because large patterns are difficult to define, and even if they are defined, their occurrences in the applications are rare, the patterns presented bellow are small ones. On the other hand, these simple patterns are preferred because the CFUs that implement them have a high clock frequency.

```
if (index < MIN) index = MIN;
if (index > MAX) index = MAX;
```

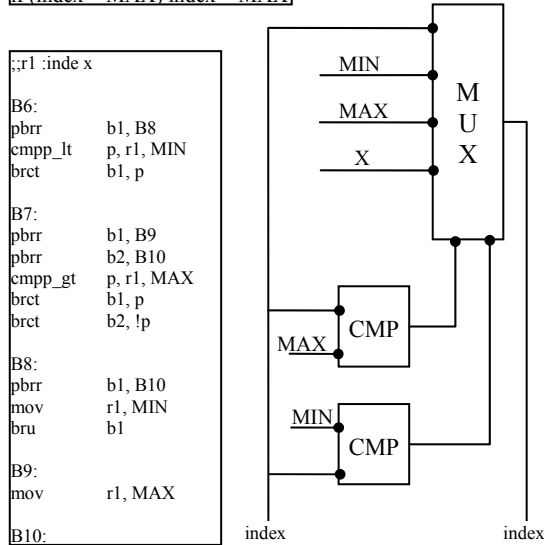


Fig. 4. Pattern 1: Saturation Circuit.

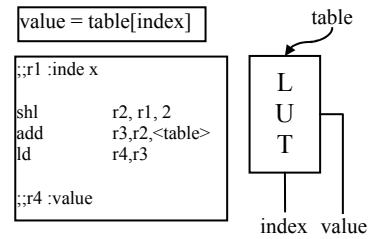


Fig. 5. Pattern 2: Table Value Selection Circuit.

Pattern 1. The two *if* statements from figure 4 saturate the value of *index* variable. The simple circuit that implements this code is made up of a multiplexer and two comparators. While the assembly code for EPIC takes 4 cycles in the best case and 6 in the worst case, this simple circuit performs the operation in one cycle.

Pattern 2. The selection of a value from a static *table* with constant values can be done by a simple circuit as shown in figure 5. Let us assume that *table* contains values of type *int*. The code written in assembly takes 3 cycles to run. The look-up table (LUT) stores the values of the *table*. Given the input *index*, the circuit computes *value* through a simple look-up operation.

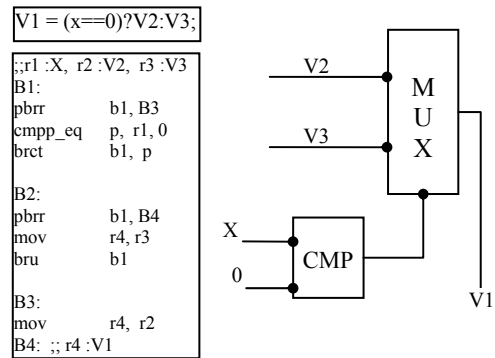


Fig. 6. Pattern 3: Value Selection.

Pattern 3. Another simple pattern is the selection of the correct value for a parameter *v1* upon a condition that generally can be expressed as a comparison with 0. The circuit shown in figure 6 performs the assignment in one cycle, instead of 3 cycles in the best case on EPIC.

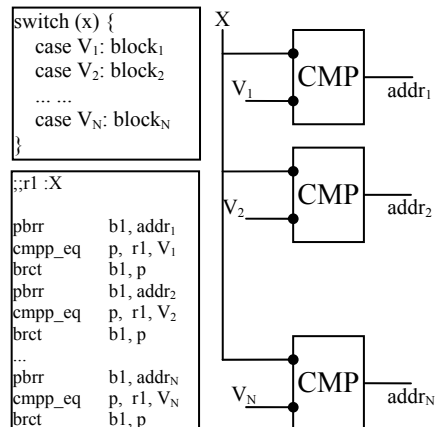


Fig. 7. Pattern 4: Address Selection.

Table 1

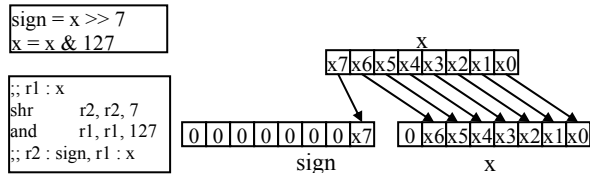


Fig. 8. Pattern 5: Sign Extraction Circuit.

Pattern 4. Extending the pattern above, figure 7 presents the circuit for the selection of the code to execute depending on the value of  $x$ . This circuit performs the selection in just one cycle, while the assembly code running on EPIC would take two cycles in the best case and  $2*n$  cycles in the worst case, if all  $n$  comparisons are made.

Pattern 5. Another very simple example is the extraction of the sign and the absolute value of a parameter. The assembly code in figure 8 is written for a variable  $x$  of 8 bits wide. Unless an extra register is used to keep a duplicate copy of  $x$  the two lines of code in the figure require a minimum of two cycles on any processor. The simple “bit-exact” circuit performs the task in one cycle.

Pattern 6. The evaluation of an arithmetic expression may be a good candidate for implementing in a circuit. It will take one CFU cycle to run, but the frequency of the circuit depends on the complexity of the expression. In order to select if the expression evaluation is hosted by the reconfigurable part or EPIC the following parameters must be considered: the core clock frequency, the number of cycles on EPIC, the CFU clock frequency and the communication overhead.

These simple patterns can be automatically detected in an application because they are easy to define and worth being implemented in the compiler as they have frequent occurrences in the applications for embedded systems.

## 5. RESULTS

Two benchmarks, G721 and ADPCM audio decoders algorithms from MediaBench suite (Lee, *et al.*, 1997) were used to evaluate the AEPIC architecture. A RC1000 development board from Celoxica with Xilinx Virtex XCV1000 FPGA was used to simulate the MRLA. The AEPIC core with 4 integer units and 2 load-store units was simulated using the AEPIC simulator introduced in a previous paper (Gheorghita, *et al.*, 2002).

Table 1 presents the comparison of running the same algorithms both on EPIC and AEPIC architectures. The first line of the table presents the number of cycles taken by running the algorithms on the EPIC processor. By moving the intensive computational part of the algorithms on the FPGA, the running cycles are divided in two types: core cycles (line 2) and CFU cycles (line 3). The clock frequencies and the number of cycles for designs are shown in lines 4 and 5.

Benchmarks on AEPIC

Benchmark	ADPCM	G721
EPIC cycles	5,708,383	323,269,902
Core cycles	1,855,577	114,147,745
CFU cycles	411,280	8,528,970
CFU freq (MHz)	15	20
CFU design cycles	8	7

To compute the number of cycles taken by an algorithm to run on the AEPIC architecture with a specified core frequency ( $f_{core}$ ) the following formula was used:

$$Cycles = Cycles_{core} + Cycles_{CFU} (f_{core} / f_{CFU})$$

The speedup obtained for running the benchmarks on AEPIC architecture with different core clock frequency are shown in table 2.

Table 2

AEPIC Speedup Relative to Core Frequencies

Core Freq.	ADPCM	G721
30 MHz	2.13	2.46
45 MHz	1.85	2.31
60 MHz	1.63	2.18
90 MHz	1.32	1.96
120 MHz	1.11	1.77

The table shows that AEPIC is particularly effective when the main processor is running at a low frequency.

Table 3

AEPIC Improvement for G721 Decoder

Core Freq	AEPIC 1 CFU speedup	AEPIC 8 CFUs speedup	Improvement
30MHz	2.46	2.63	1.07
60MHz	2.18	2.57	1.18
120MHz	1.77	2.36	1.34

Using the patterns identified above, the intensive computational parts of the algorithms were implemented as a sequence of circuits. All these circuits represent patterns presented above. To compute the improvement, for those CFUs faster than the core, the clock frequency was reduced to core clock frequency.

Table 4

## AEPIC Improvement for ADPCM Decoder

Core Freq	AEPIC 1 CFU speedup	AEPIC 3 CFUs speedup	Improvement
30MHz	2.13	2.37	1.11
60MHz	1.63	2.00	1.23
120MHz	1.11	1.48	1.33

For G721 decoder, the design was split in a sequence of 8 small circuits, with the following frequencies 60, 60, 120, 30, 30, 30, 120, respectively 60 MHz. The improvements are presented in table 3.

For ADPCM decoder, the design was split in a sequence of 3 small circuits, with the following frequencies 30, 15, respectively 60 MHz. The improvements are presented in table 4.

## 6. CONCLUSIONS

This paper investigates the granularity of the application part implemented on the re-configurable structure of the Adaptive EPIC architecture. The results show that the fine-grained partitioning is preferred instead of a course-grained one, especially when the core is running at high frequencies (120MHz).

A possible extension of the current work is to explore the possibilities of improving the compiler in order to schedule instructions in parallel on CFUs and EPIC, so as to avoid the stall of the core while CFUs are running.

## REFERENCES

**Athanas, P.M.** and **H.F. Silverman** (1993). Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, **26(3)**, pp 11-18.

**DeHon, A.** (1996). *Reconfigurable Architectures for General Purpose Computing*. PhD thesis, MIT AI Labs.

**Estrin, G.** (1960). Organization of computer system – the fixed plus variable structure computer. In *Proceedings of the Western Joint Computer Conference*, pp 33-40.

**Gheorghita, V.S., W. Wong, T. Mitra and S. Talla** (2002). A Co-Simulation Study of Adaptive EPIC Computing. In *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT 2002)*, pp 268-275.

**Gokhale, M., W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely and D. Lopresti** (1991). Building and using a highly parallel programmable logic array. *IEEE Computer*, **24(1)**, pp 81-89.

**Kathail, V., M. Schlansker and B. Rau** (1994). HPL PlayDoh Architecture Specification Version. Technical Report HPL-93-80, Hewlett Packard Laboratories, Technical Publication Department, 1501 Page Mill Road, Palo Alto, CA 94304.

**Lee, C., M. Potkonjak and W.H. Mangione-Smith** (1997). MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of 30<sup>th</sup> Annual IEEE/ACM Symposium on Microarchitecture (MICRO-30)*, pp 330-335.

**Talla, S.** (2000). *Adaptive Explicitly Parallel Instruction Computing*. PhD Thesis, New York University.

**Vuillemin, J., P. Bertin, D. Roncin, M. Shand, H. Touati and P. Boucard** (1996). Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, **4(1)**, pp 56-59.

**Xilinx Inc.** (1994), San Jose, C.A. *The Programmable Logic Data Book*.