A Co-simulation Study of Adaptive EPIC Computing

Valentin Stefan Gheorghita
Politehnica University of Bucharest
Bucharest
Romania
septica@cs.pub.ro

Weng-Fai Wong, Tulika Mitra
Department of Computer Science
National University of Singapore
Singapore
{wongwf | tulika}@comp.nus.edu.sg

Surendranath Talla
Agere Systems
Altanta, GA
U.S.A
talla@agere.com

ABSTRACT

Reconfigurable computing offers the embedded systems designers the flexibility of application specific optimizations on a generic platform. In this paper, we are concerned with a fine-grain, tightly coupled, dynamically reconfigurable architecture we call Adaptive EPIC. A generic EPIC architecture is augmented with a dynamically reconfigurable structure. In this paper, we describe an experimental setup to evaluate the performance of such a processor. Our results show that such architecture can offer significant performance improvements for low frequency, and hence low power, core processors.

1. INTRODUCTION

In the context of embedded processing, there are two main ways to improve performance. In the first approach, one first selects a processor that meets macro constraints such as cost, footprint etc. and then optimizes the intended application for such a processor. In the current state-of-art, this often entails assembly programming the core sections of the code. Alternatively, given the application, one can optimize the hardware to be used to execute the application. This allows the designers better control over the design and in better meeting the optimization criteria. However, engineering a new processor is often a very expensive proposition.

The introduction of tightly coupled reconfigurable processors offers a new degree of freedom in the design space. They give the designs the cost-effectiveness of using off-the-shelf silicon with the flexibility of optimizing parts of the hardware for specific applications.

In an earlier work, we introduced a dynamically reconfigurable processor architecture we called *Adaptive EPIC* (*Explicitly Parallel Instruction Computing*) [20]. The basic design consists of a segmented reconfigurable array that is tightly coupled with an EPIC processor. The AEPIC architecture combines the advantages of the EPIC with its simpler architecture backed by well-known compiler technology, and that of programmable logic that exploits fine-grain parallelism through explicit control over micro-architectural features. In the paper, we described the brief design of the processor and the compiler considerations that are needed to work with such a processor. The contributions of the present work are as follows:

- We describe a simulation infrastructure that realistically simulates both the main EPIC core and reconfigurable component. We use the state-of-the-art FPGA technology to obtain realistic measurements for the latter.

- We present simulation results on embedded benchmarks that show that the concept significantly benefits embedded computing especially when the processor needs to operate at lower power and hence lower frequencies.

2. PREVIOUS WORK

The earliest known computing system based on reconfigurable devices was proposed and implemented by Gerald Estrin at UCLA [8]. It is a hybrid machine consisting of a general-purpose processor interconnected with high-speed logic devices, which were reconfigured manually. The introduction of Field Programmable Gate Array (FPGA) devices by Xilinx in the mid 80's [22, 25] spurred research in FPGA based reconfigurable computing engines. PRISM [1], PAM [24], and SPLASH-2 [9] are some pioneering efforts in this direction. More recently, researchers have explored variations of FPGA architectures and also some radical architectures, which combine programmable processor with reconfigurable logic. Some examples of the former are DPGA and MATRIX [5]; while RAW [23], PipeRench [10], Garp [13], PRISC [18], RaPiD [6], Cameron [11] and Chimaera [12] exemplify the latter. A survey of some of the past work can be found in one of the author's thesis [20].

In the last two-three years, even some FPGA manufacturers have shown interest in combining FPGA with standard processor cores. These devices are being targeted for the embedded market, and this is reflected in the choice of the processor cores chosen to go with the programmable logic (from 8 bits microprocessor at 40Mhz up to 32bits RISC processor at 166Mhz). Some released or announced devices are: Triscend's E5 and A7, Atmel's FPSLIC and Chameleon Systems' CS2000. All these devices, except Chameleon ones, use fine-grained reconfigurable array; the FPGA logic blocks operates on one or two bit wide data. The exception, CS2000, contains up to eighty-four 32-bit datapath units (DPUs) each of which includes a 32-bit arithmetic-logic unit (ALU). Enzler and Platzner [7] presented the current products in this domain and the future trends.
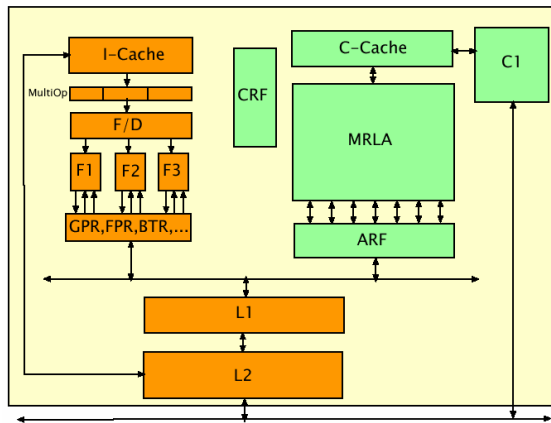

Figure 1. AEPIC architecture

3. ADAPTIVE EXPLICITY PARALLEL INSTRUCTION COMPUTING

The AEPIC architecture provides a **dynamically varying** EPIC style architectural interface to the executing process. This means the interface observed by the executing program in any machine cycle is that of an explicitly scheduled EPIC architecture [16, 19]. The variation can be in terms of the number and types of instructions that can be executed on any given machine cycle. A machine that implements AEPIC architecture may be composed of hardwired functional units and some programmable logic that can be reconfigured to implement application specific instructions. On an AEPIC machine, the running program also controls the adaptation. However, the decisions of when and how to reconfigure are pre-determined by the compiler and embedded into the code generated for the given application.

3.1. Key features of the architecture

The AEPIC architecture is motivated by a desire to (1) enable efficient reconfiguration of the processor data-path at runtime, (2) allow compiler to determine the reconfiguration decisions in a flexible and efficient manner and, (3) allow AEPIC researchers to study a wide variety of AEPIC machine configurations. In order to achieve these goals, the AEPIC architecture proposes the following novel features.

Compiler specified resource allocation. Here we are referring to resources that are intended for hosting Configured Functional Units (CFU). AEPIC delegates to the compiler the task of specifying which regions of the program code will execute on the programmable logic and when they are allocated (de-allocated) to (from) the programmable resources on processor.

Architecturally transparent resource assignment. Although the AEPIC compiler decides which particular piece of the computation should be performed on the programmable logic each cycle, the processor determines which region of the programmable logic resource is utilized for hosting that computation.

Support for efficient context switching and modular software development. AEPIC architecture allows multiple CFUs to be instantiated simultaneously and groups them into distinct sets so that on any cycle, a particular set of CFUs is considered active. These active CFUs are the ones on which operations can be executed. The architecture also provides special instructions to alter these CFU sets or to switch between sets to make a different one active.

Explicitly controlled configuration cache hierarchy. AEPIC provides architectural mechanisms to explicitly control the data placement in the configuration cache hierarchy. This feature is a natural extension of the explicitly controlled data cache hierarchy mechanisms provided in some EPIC architectures [16]. It is expected to play an even more significant role in AEPIC processing where the costs of configuration cache misses can be more expensive than the costs of conventional data cache misses. Since applications are expected to have a much smaller number of configurations than the number of program values (which go through the traditional cache hierarchy), explicit control of configuration data placement is expected to be feasible and advantageous.

Implicitly specified operands for configured functional units. Unlike typical RISC operations, some of the operations performed by CFUs may take a large number of input/output operands. In order to simplify the instruction decode logic and to keep the instruction format simple, operands for CFU operations are not specified as part of the instruction itself. Instead, AEPIC architecture specifies operand assignment operations that associate specified registers as sources (destinations) for input (output) operands for CFU operations.
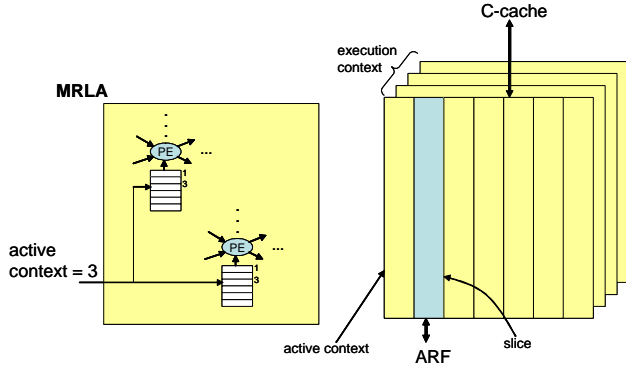
Figure 2. Structure of MRLA & multiple-context MRLA

3.1.1 AEPIC Architecture

Figure 1 shows the abstract architecture of an AEPIC machine. The core component 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) and *Array Register File* (ARF) connected together via bus interconnect. *Configured Functional Units* (CFU) are implemented using the programmable MRLA. Configurations are cached in the *C-Cache*. The C-Cache merges into the standard memory hierarchy thus providing a rapid means of instantiating CFU whose configurations are held as data in the standard memory hierarchy of the system. The *Configuration Register File* (CRF) consists of a set of *configuration registers* (CR). Each CR names a CFU, which may be instantiated or held in the C-Cache. Most of the AEPIC instructions use a configuration register as an operand to refer to a (virtual) CFU. The full details of the proposed instruction set can be found in Talla's thesis [20].

We shall now describe a little more detail about the *Multi-context Reconfigurable Logic Array* (MRLA). The structure of the MRLA is shown in Figure 2. Like a typical Field Programmable Gate Array (FPGA), the MRLA is a two dimensional structure that is composed of programmable logic and interconnect blocks. We shall use the term *Programmable Element* (PE) to refer to both the programmable logic block as well as the programmable interconnect block. The behavior of each PE is determined by a *configuration instruction*. Just as is the case for FPGA, any given logic design can be emulated on the MRLA by supplying suitable *configuration instructions* for all the programmable elements of the array. CFUs are none other than sets of logic designs implemented by means of configuration instructions.

To allow for rapid and dynamic reconfiguration, the MRLA permits multiple contexts to be present in the logic area. In a standard FPGA each programmable element can only take a single configuration instruction. This implies that only one logic design can be resident on the array until it is reconfigured by re-associating a new set of configuration instructions. In the MRLA, each programmable element can be associated with *multiple* configuration instructions. This allows multiple logic designs (CFUs) to be simultaneously resident on the MRLA. Selecting the appropriate configuration instruction for each of the programmable element can activate the desired logic design.

Configuration instruction slots for each PE (called the *configuration memory*) are stored in an ordered sequence and all PEs have the same number (*D*) of configuration instruction slots. MRLA takes an input called *context_id* that can take values from 1 to *D*. A value of k to the *context_id* input selects the kth configuration instruction from the configuration memory as the instruction for each PE. The kth configuration instruction is referred to as the *active configuration instruction* for that PE.

The set of configuration instructions with identical index in the configuration memory of a PE is referred to as an *execution context*. The execution context that is associated with currently active configuration is called the *active context*. MRLA can be effectively viewed as an array of FPGA, one array per execution context; and the *context_id* serves as the index into this array. Selection of an execution context makes all the CFUs of that context available for instruction processing by subsequent instructions.

In addition, AEPIC inherits several of the innovative features of EPIC architectures such as MultiOp, speculative and predicated execution, decoupled branches, efficient boolean reductions, compiler controlled cache behavior [16].

3.2 Programming AEPIC

In this subsection, we will describe how the CFUs can be used by means of pseudo-code. Programming the CFUs consists of two parts: the configuration and the usage.

The CFU configuration code 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

The above code will allocate space and registers for a configuration as well as load it in the C-cache and MRLA. The calloc in Line 1 allocates adequate number of blocks in the C-cache for the configuration located at memory address pointed to by reg. It also associates configuration register cr with the configuration. The malloc instruction in line 2 allocates the required number of slices on MRLA on context specified by literal cid for the configuration associated with cr. The number of slices required by the CFU is obtained from the information stored in

configuration cr. At line 3 the configuration data associated with cr is transferred from C-cache to MRLA. The instructions inp and outp (lines 4 and 5) associate array registers files as input and output registers for the cr. There can be several input and output register, with the base given by ar and the count given by the literal lit. Also several configurations can be associated with the same register. This is because there can only be one configuration call at a time and it is only during the input and output of the computation implemented with the configuration that this association takes meaning.

Each time that a CFU is activated, 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 call the necessary input data must be transfered to the CFU input registers using standard instructions such as register moves or memory loads. At line 2, the context with the configuration to be executed is set as the current context. This is done by the setctx instruction. This instruction must be present only if the required configuration is not in the current context. The instruction in line 3 call the operation opid on the CFU associated with cr. This effectively triggers the execution of the CFU on the given input. The next instruction (wtc) waits for the execution on the CFU to complete. It effectively stalls the processor. Finally, the output from the CFU's computation is removed.

4. AEPIC SIMULATOR

The AEPIC simulator used for this study is based on the cycle level simulator of the HPL-PD EPIC architecture [16] that is distributed with the Trimaran ILP Compiler infrastructure [21]. Though a paper design, HPL-PD has significant influence on the definition of the Intel IA-64 architecture [14].

The infrastructure was re-engineered to accept the AEPIC instruction set. The simulator generates run-time information such as clock cycles taken for execution, average number of operations executed per cycle, static instruction counts, configuration register allocated overheads, as well as detailed information about execution profile on the adaptive component of AEPIC such as time spent for data-path reconfiguration, computation time (cycles) on MRLA, as well as the effectiveness of configuration cache (C-Cache and C1, the second level configuration cache).

The AEPIC simulator is composed of five key modules (Figure 3). Each of these modules processes a different subset of the AEPIC ISA. The AEPIC Interpreter (which

is essentially the original Trimaran simulator shown in Figure 4) reads the AEPIC code and processes the instructions in sequence as dictated by the compiler generated Plan of Execution (POE). The interpreter processes the non-memory related AEPIC instructions from EPIC subset of AEPIC ISA. The Data Cache Manager processes the non-configuration data related memory operations. The Configuration Manager processes the adaptive extension instructions. Among these instructions, Configuration Cache Manager processes those dealing with the configuration cache hierarchy while Array Manager processes those that deal with MRLA reconfiguration.
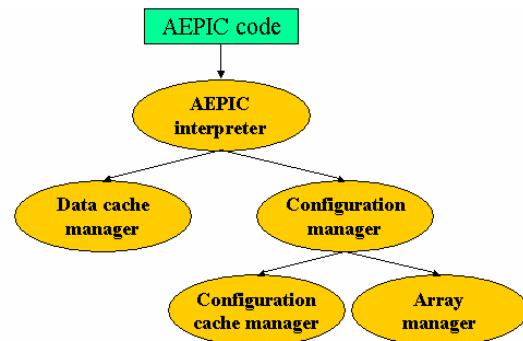

Figure 3. AEPIC Simulator Components

An important issue is how the application can be optimized to exploit the AEPIC features. For this purpose we need to identify the parts of an application that are most suited to run on the MRLA. Although ideally the compiler should do this automatically, we still do not have good compiler algorithms for this automatic partitioning. Therefore we did this partitioning manually. Using runtime information from the EPIC simulator, we identified the compute intensive parts of the application. By considering the speedup gained by performing the computation on the CFU, the estimated time to reconfigure the MRLA, and the estimated time needed to transfer input/output data to/from the CFU, we select sections of the code to be performed in the CFU.

To obtain realistic estimations of the cycle time and number of execution cycle, we used FPGA technology to approximate the MRLA. The chosen parts of the applications are implemented in Xilinx Vertex XCV1000 FPGA using a high-level hardware language Handel-C [2]. We decided to use Handel-C instead of more efficient hardware description language because of the ease of converting C code to Handel-C. What is generally involved is the insertion of parallel constructs to the C code. An example of this conversion is shown in the Appendix. Note that even though Handel-C has a very well defined statement-based timing model that makes it

Table 1. Performance of the benchmarks

Benchmark	EPIC Cycles	AEPIC Cycles	FPGA Cycles	FPGA Freq (Mhz)	AEPIC Speedup	FPGA Gates
ADPCM Decoder	5,708,383	2,266,857	411,280	15	2.52	15,645
G721 Decoder	323,269,902	122,676,715	8,528,970	20	2.64	18,531
G721 Encoder	85,343,654	34,996,803	2,173,124	20	2.44	18,531
Idea Encrypt	21,384,752	12,352,755	1,413,568	15	1.73	15,788
Pegwit Encrypt	66,071,148	36,967,445	186,591	30	1.79	77,483

easy to calculate cycle count of execution, we still need the FPGA to measure reconfiguration time.
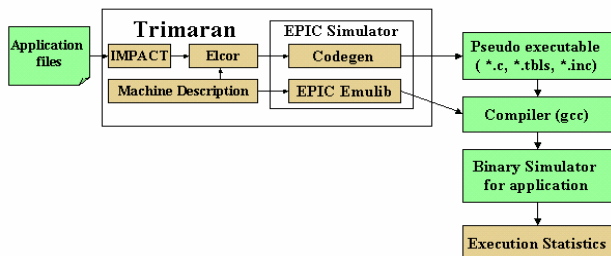

Figure 4. EPIC Simulation

The final step is to add the AEPIC instructions into the application to reconfigure data-paths and control the CFU. The resulting AEPIC application is compiled and ran with the same input. As the FPGA setup runs on Microsoft Windows while Trimaran runs on Linux, we use a remote FPGA server that offers RPC-like service to the AEPIC simulator. The FPGA server will load and execute the compiled Handel-C code and report back the execution cycles to the AEPIC simulator. The whole process is shown in Figure 5. This co-simulation framework gives us a more realistic picture of the AEPIC's performance.

5. RESULTS

We used four benchmarks to evaluate the AEPIC architecture. These four benchmarks consist of two encryption algorithms, IDEA [15] and Pegwit, and two audio decoders algorithms, G721 and ADPCM. The last three benchmarks are from the MediaBench suite [17].

We used RC1000 development board from Celoxica [3] with Xilinx Virtex XCV1000 FPGA [26] to simulate the MRLA. The AEPIC core with 4 integer units and 2 load-store units was simulated on Pentium III processor.

5.1. Basic Speedups

The speedup obtained for each application is presented in Table 1. This speedup was computed by assuming that the EPIC main processor and the reconfiguration unit run at the same frequency. Read another way, it tells us that an AEPIC processor running at a lower frequency can
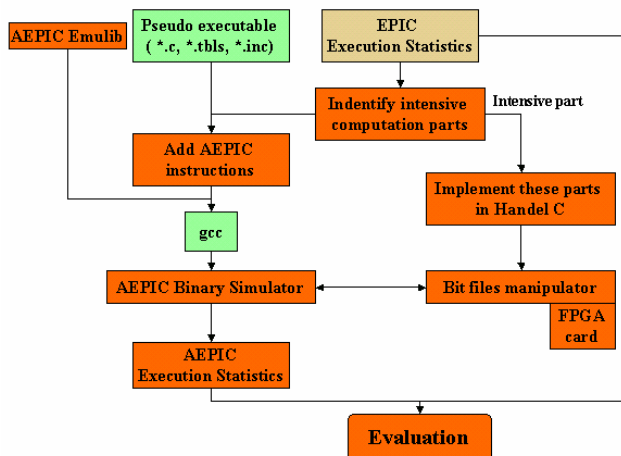

Figure 5. AEPIC Evaluation

achieve the same performance of an EPIC processor running at a higher frequency. In the Table, we also show the number of FPGA gates used to implement the application CFU. As can be seen, the CFUs are relatively small. We therefore assume that the configuration can be loaded in the cache and the CFU configured way ahead of its usage.

Table 2. Number of Input and Output Registers

Benchmark	Input Registers	Output Registers
ADPCM Decoder	4	3
G721 Decoder	2	1
G721 Encoder	2	1
Idea Encrypt	2	1
Pegwit Encrypt	4	4

Table 2 shows the total number of input and output registers needed for each benchmark. It should be noted that with our current implementation of stalling the main processor when the CFU executes ensures that the CFU is the sole bus master should it be necessary to obtain data from memory.

5.2. Performance trade-off between Core and CFU frequencies

Using the simulation data, we performed further study by assuming that the core processor runs at a higher frequency than the reconfigurable unit. This is not an unlikely situation if we project the speed difference between the current generation of embedded processors and FPGAs onto the AEPIC architecture. The key question is then where is the break-even point, in other words, at what kind of speed differentiates will it no longer be useful to have a reconfigurable unit because the main core processor is fast enough to handle the computation. The simulator will report on the total number of cycles the execution took, $Cycles_{AEPIC}$ that assumes that the core and the reconfigurable component are running at the same frequency. To adjust for the difference in core and reconfigurable component, we re-compute the executable cycles as follows. Let $Cycles_{FPGA}$ be the component of $Cycles_{AEPIC}$ that is the estimated number of cycles consumed by the reconfigurable component. We obtain this from the FPGA implementation of the computation core using Handel-C. The same implementation, after placement and routing, will also report the number of gates needed to implement the logic as well as the clock frequency (f_{FPGA}) with which the circuit can be executed. From the execution logs of the simulations, we used the following formula to compute the AEPIC execution cycle count for a specific main core processor frequency f.

$$
\begin{aligned}
Cycles_f &= Cycles_{AEPIC} - Cycles_{FPGA} + \\
&\quad (Cycles_{FPGA} \times f / f_{FPGA}) \\
&= Cycles_{AEPIC} + Cycles_{FPGA} \times (f / f_{FPGA} - 1)
\end{aligned}
$$

Table 3. AEPIC Speedup relative to Core Frequencies

CPU Freq	ADPCM Decoder	G721 Decoder	G721 Encoder	Idea Encrypt	Pegwit Encrypt
60	1.63	2.31	2.17	1.29	1.78
120	1.11	1.96	1.86	0.96	1.76
180	0.84	1.69	1.63	0.77	1.74
240	0.68	1.49	1.45	0.64	1.73
360	0.49	1.21	1.19	0.48	1.69

After the above recalibration, we compute the speedups for various core frequencies. The results are shown in Table 3. They show that AEPIC is particularly effective when the main core processor is running at a low frequency.

5.3. Multiple, Smaller CFU

The clock frequencies we obtained after placement and routing of the portion of the code identified for execution in the reconfigurable part of the processor is typically between 15 to 30 MHz. The realizable clock frequency is determined by the complexity of the circuit that affects the critical path of the circuit. Since the design of AEPIC allows for a number of CFU slices to be dynamically loaded, we experimented with splitting the code (and hence the circuit) to be realized in the CFU into smaller pieces. We need to recalibrate the counting of execution cycles reported by the simulator. Using AEPIC simulator, we counted the number of CFU calls and the number AEPIC cycles. In our applications, the main core processor will wait for the CFU to complete its operation before proceeding. Therefore we can compute the number of cycles spent by the main core processing in waiting for the CFU to complete its work as follows:

$$
TotalCycles_{CFU_i} = \sum_{j}^{Calls_{CFU_i}} \left\lceil f / f_{FPGA_i} \right\rceil \times Cycles_j
$$

$$
WaitCycles_f = \sum_{i}^{\text{\# of CFUs}} TotalCycles_{CFU_i}
$$

where f is the frequency of the main core, f_{FPGA_i} is the frequency of CFU slice i estimated using FPGA technology, and $TotalCycles_{CFU_i}$ is the total number of cycles executed by CFU i. This is the sum of all cycles executed by CFU i for each call to it.

We split the code for the reconfigurable unit in both audio decoder applications that was tested in first experiments. The total number of gates in the split CFUs is about the same as that for a single CFU. In fact, in some cases, because of further circuit simplifications, it is slightly lower than the single CFU case. The code for the CFUs of the other benchmarks is too simple to be split. For ADPCM the total number of CFU cycles is 121,228,705 for eight CFU. In Table 4 we show the number of calls for every CFU and the attained frequency. The speedups obtained are presented in Table 5.

For the G721 benchmark the total number of CFU cycles is 2,266,857 for three CFU. Table 6 presents the number of calls for every CFU and its frequency. The speedups for G721 are presented in Table 7.

Table 4. Splitting the CFU for ADPCM

Freq	56	60	120	35	27	37	120	60
Call				1,180,160				267,850

Table 5. ADPCM Speedup

Main CPU Freq	AEPIC 1 CFU Speedup	AEPIC 8 CFU Speedup	Improvement
60Mhz	2.31	2.43	5.17%
120Mhz	1.96	2.27	16.31%
180Mhz	1.69	2.12	25.12%
240Mhz	1.49	1.98	32.79%

Table 6. Splitting the CFU for G721

Freq	24	15	59
Call		147,520	116,240

Table 7. G721 Speedups

Main CPU Freq	AEPIC 1 CFU Speedup	AEPIC 8 CFU Speedup	Improvement
60Mhz	1.63	1.62	-1%
120Mhz	1.11	1.26	14%
180Mhz	0.84	1.00	20%
240Mhz	0.68	0.85	26%

For both benchmarks, it shows that splitting the CFUs resulted in smaller CFUs that can be realized with higher frequencies. This extended the speedups afforded by AEPIC by closing the gap between the core's and the CFU's frequencies.

6. CONCLUSION

In this paper, we described a simulation environment and provided evaluation for a fine-grain, dynamically reconfigurable processor that consists of an EPIC core tightly coupled with a reconfigurable unit. Evaluation using four embedded benchmarks using FPGA technology to stand in for the CFUs shows that AEPIC shows particular potential for low frequency, and hence low power, systems. Under such assumptions, we were able to achieve a speedup of up to 2.43 times in performance.

As an extension of the current work, we would like to investigate how we can automatically identify the part of the application that is of the correct granularity and that can be executed efficiently on AEPIC. We would also like to explore issues relating to structuring applications so as to operate the CFUs in parallel with the main core.

7. ACKNOWLEDGMENT

8. REFERENCES

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

[2] Celoxica. DK1.1. http://www.celoxica.com/home.htm

[3] Celoxica. RC1000 Product Information Brochure. http://www.celoxica.com/products/technical_papers/datasheets/DATRHD002_0.pdf

[4] K. Compton and Scott Hauck. Configurable Computing: A Survey of Systems and Software. *Technical report, Northwestern University, Dept. of ECE*, 1999

[5] A. DeHon. *Reconfigurable Architectures for General Purpose Computing*.PhD thesis, MIT AI Lab, September 1996.

[6] C. Ebeling, D.C. Cronquist, and P. Franklin, RaPiD-Reconfigurable Pipelined Datapath, *6th Annual Workshop on Field Programmable Logic and Applications*, 1996.

[7] R. Enzler and M. Platzner, Dynamically Reconfiguration Processors. *Telematik, Zeitschrift des Telematik-Ingenieur-Verbandes*, 7(1), 6-11, 2001.

[8] G. Estrin. Organization of computer system – the fixed plus variable structure computer. In *Proceedings of the Western Joint Computer Conference,* pages 33-40, 1960.

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

[10] S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. *Intemational Symposium on Computer Architecture*, pp. 38-49, 1999.

[11] J. Hammes et.al. Cameron: High-level Language Compilation for Reconfigurable Systems. *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1999.

[12] S. Hauck. The Chimaera reconfigurable functional unit. In *Proc. of IEEE Symp. on FPGAs for Custom Computing Machines*, Napa Valley, California, 1997, pp. 87–96., 1997.

[13] J. R. Hauser and J. Wawrzynek, Garp – A MIPS processor with reconfigurable coprocessor. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 24-33. April 16-18, 1997.

[14] Intel Inc. Itanium Architecture. http://developer.intel.com/design/itanium/manuals/index.htm

[15] International Data Encryption Algorithm. http://www.eskimo.com/~weidai/cryptlib.html.

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

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

[18] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmed functional units. In *Proceedings of 27th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-27)*, pp.172--180, Nov. 1994.

[19] M. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik and S. G. Abraham. Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity. HPL Technical Report HPL-96-120. Hewlett-Packard Laboratories, February 1997.

[20] S. Talla. *Adaptive Explicitly Parallel Instruction Computing*. PhD thesis, New York University, 2000.

[21] Trimaran ILP Research Infrastructure, 1998. http://www.trimaran.org.

[22] S. M. Trimberger. *Field Programmable Gate Array Technology*, Kluwer Academic Publishers, 1994.

[23] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it All to Software: Raw Machines. *IEEE Computer*, pages 86--93, September 1997.

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

[25] Xilinix Inc., San Jose, C.A. *The Programmable Logic Data Book*, 1994.

[26] Xilinx Inc., San Jose, C.A. *Virtex™ Data Sheet*. http://www.xilinx.com/partinfo/ds003-1.pdf

Appendix – Example of Conversion of a procedure from application g721

Original C code
```
static int fmult( int an, int  srn)
{    short       anmag, anexp, anmant;
     short       wanexp, wanmag, wanmant;
     short       retval;

     anmag = (an > 0) ? an : ((-an) & 0x1FFF);
     anexp = quan15(anmag) - 6;
     anmant = (anmag == 0) ? 32 :
        (anexp >= 0) ? anmag >> anexp : anmag << -anexp;
     wanexp = anexp + ((srn >> 6) & 0xF) - 13;
     wanmant = (anmant * (srn & 077) + 0x30) >> 4;
     retval = (wanexp >= 0) ? ((wanmant << wanexp) & 0x7FFF) :
        (wanmant >> -wanexp);
     return (((an ^ srn) < 0) ? -retval : retval);
}
```

Handel-C equivalent
```
/* Handel-C requires explicit specification of bit width */
signed int 32 fmult(signed int 32 an, signed int 32 srn) {
signed int 32 rettmp, temp2, anmag,retval,wanmant,anmant;
signed int 6 wanexp,temp1, anexp;
signed int 5 retq;
unsigned int 6 shift;

anmag = (an > 0) ? an : ((-an) & 0x1FFF);
/* No procedure call in Handel-C – this is a macro call */
retq = quan15(anmag);
/* Concantenates 0 and retq to the length of 6 bits */
anexp = (signed int 6)(0@retq) - 6;
/* Statements 1 and 2 are to be executed in parallel */
par {
   if (anmag == 0)  anmant = 32; /* 1 */
   else
     par { /* Statements 1.1 and 1.2 are executed in parallel */
        if (anexp >= 0) /* 1.1 */
           anmant = anmag >> ((unsigned int 6)(anexp))  ;
        if (anexp <  0) /* 1.2 */
           anmant = anmag << ((unsigned int 6)(-anexp));
   }
   wanexp = anexp +  /* 2 */
   (signed int 6)(((srn >> 6) & 0xF) <- 6) - 13; /* take LSB 6 bits */
}
wanmant = (anmant * (srn & 077) + 0x30) >> 4;
if (wanexp < 0)
   retval = wanmant >> ((unsigned int 6)(-wanexp));
else
   retval = (wanmant << ((unsigned int 6)(wanexp))) & 0x7fff;
return ((an^srn) < 0)?-retval:retval;
}
```