Integral Parallel Architecture in System-on-Chip Designs

Gheorghe M. Ştefan

Faculty of Electronics, Tc. and IT, Politehnica University of Bucharest, Bd. Iuliu Maniu, 3-5, Bucharest, Romania

gstefan@arh.pub.ro

***Abstract* — The ubiquitousness of the parallel computational resources emerges in the rapid growing market of system-on-chip. Both, complex and intense computations are requested for solving the fast expanding functional spectrum of the mobile products. The current approach is unable to provide low area and low power solutions for the increased functional hungry. The proposed Integral Parallel Architecture (IPA) provides >100x increase for *GIPS/Watt* and *GIPS/mm*^{2} than the current structures. This new approach is based on *ConnexArray*™ technology, developed and tested on real chips, and on the Bubble-free Embedded Architecture for Multithreading (BEAM) execution. It is proposed an IP based model to manage tens of threads and a number of execution or processing units which starts from tens and goes up to thousands.**

I. Introduction

The SoC domain is driven by two forces:

- the functional spectrum is enlarging, requesting highly complex and high data-intense computation,
- the number of transistors per cm^2 of silicon increases,

while the possibility to follow this trend is limited by:

- our inability to fill up the size/complexity gap between *making* and *specifying* (the technological developments help us to have more transistors/die, but do not provide us with the techniques to write down more lines of code describing circuits with the corresponding complexity),
- our inability to provide architectural solutions for limiting the energy waste (only structural solutions are provided).

Our solution is based on the following main decisions:

1. To "move" the complexity from the circuit level to the informational level, increasing the weight of embedded computation, substituting as much as possible the ASIC approach with programmable solutions. The functional complexity will come mainly from programming.
2. To segregate the *complex* computation by the *intense* computation [13], in order to optimize independently these two too distinct forms of computation.
3. Because the resulting programmable solution will competes with circuits - "naturally" parallel structures - the engine must be a parallel one.

While for the initial stages of developing embedded computation using sequential architecture was a very good solution, in the actual stage of development parallel computation is a must, and the main problem is: *what kind of parallel architecture is the best fit for embedded computing?* Unfortunately, the answer is*: we need as many kinds as possible, because the diversity of circuits to be emulated* ***efficiently*** *requests a comparable architectural diversity*.

Our proposal takes into account the forms of parallelism which result from the most appropriate computation model to be used as starting point for defining what parallelism means. It is the model of ***partial recursive functions*** proposed by Stephan Kleene [6]. Based on Kleene's model, in [7] and [8] is proposed a new taxonomy for parallel computation. The taxonomy proposed by Michael Flynn [3], and the similar ones, are somehow "artificial", because are based on formal constructs derived from the sequential model of Allan Turing.

II. Integral Parallel Architecture

In [8] is proved that, according to Kleene's model, the building of a parallel model of computation can be exclusively based on the composition rule having the form:

$$f(x_1, \dots x_m) = g(h_1(x_1, \dots x_m), \dots h_n(x_1, \dots x_m))$$

which is a n-sized form (see in Fig. 1 its structural version) which describes two aspects of parallelism: the *synchronic* parallelism of computing n functions h_i, and the *diachronic* parallelism of pipelining h_i with the reduction function g.

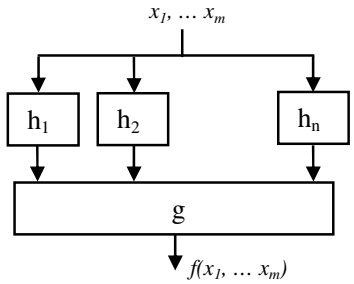

Fig.1. The structural representation for Kleene's composition rule.

Five types of parallel computation can be emphasized:

- **Data-parallel computation**, characterized by:
  $h_i(x_1,\dots x_m) = h(x_i), \quad g(h(x_1),\dots h(x_m)) = \{h(x_1),\dots h(x_m)\}$
- **Time-parallel computation**, characterized by:
  $m = 1$
- **Speculative-parallel computation**, characterized by:
  $h_i(x_1,\dots x_m) = h_i(x), \quad g(h_1(x),\dots h_m(x)) = \{h_1(x),\dots h_m(x)\}$
- **Reduction-parallel computation**, characterized by:
  $h_i(x_1,\dots x_m) = x_i, \quad f(x_1,\dots x_m) = g(x_1,\dots x_m)$
- **Thread-parallel computations**, characterized by:
  $h_i(x_1,\dots x_m) = h_i(x_i), \quad g(h_1(x_1),\dots h_m(x_m)) = \{h_1(x_1),\dots h_m(x_m)\}$

Any complex embedded application requests ***all*** these types of parallel computation.

A. Implementing IPA

An IPA is able to perform all types of parallel computation previously listed. The computation in a system with IPA is defined on the following data structures: *scalar*, *vector,* and *stream* of scalars, and uses for defining the computation: *functions* on scalars, vectors or streams (f(x,…), f(V,…), f(S,…)) and *function vectors* (F = <f_1 … f_m>).

We know that: (1) ***any computation can be expressed using a combination of the following particular forms***:

1. **data-parallel:** f(V_1 … V_n) = V
2. **reduction-parallel:** f(V) = x
3. **speculative-parallel:** F(x)=<f_1…f_m>(x)={f_1(x)…f_m(x)}=V
4. **time-parallel:** F(S_I)=<f_1…f_m>([x_1…x_n])=[y_1…y_n]=S ; a stream of scalars [x_1…x_n] is applied to the pipe of functions <f_1…f_m>; the result stream is [y_1…y_n]
5. **thread-parallel:** $f_1(x_1…x_m)=y_1$,…$f_n(x_1…x_p)=y_n$

We make the assumption that: (2) ***most of the frequent computations are performed efficiently if they are expressed using a combination of the previously defined functions***.

The assumption (2) is investigated in [9] based on [1]. The sentences (1) and (2) propose a *functional approach* mixed with a sort of *RISC approach* promoted starting with early 1980s. Let's call this approach: ***parallel RISC***.

B. Intense computing

The first four forms of parallel computation have a common characteristic: different kinds of patterns characterize them.

1. Data-parallel: each component of the vector results from the predicated execution of the same program.

2. *Reduction-parallel*: each vector component is equivalent related to the reduction function.

3. *Speculative-parallel*: applies, usually, the same variable to slightly different function.

4. *Time-parallel*: a pipe of functions <f_1…f_m> is applied to [x_1…x_n] providing an efficient computation for $n >> m$.

In all these cases the dominant characteristic of computation is its ***intensity***, i.e., a big amount of data is processed or is outputted. Therefore, both, data and program flow are ***highly predictable***, determining the features of the sub-architecture we propose for performing the ***intense computation***:

- the computation is done in a cellular structure of ***many*** *small & simple* processing/execution cells [11]
- array computing is the main type of processing executed in a linear network of cells
- the computation is a high-latency functional pipe
- buffer memory hierarchy with *out-of-core* executions.

C. Complex computing

The multi-threaded computation is a form of parallelism described by: $f_1(x_1…x_m)=y_1$, … $f_n(x_1…x_p)=y_n$, where each function represents a distinct program running on distinct data. Each of these computations is pattern-less. Therefore, we will refer to them as the ***complex computing***, characterized by:

- mono or ***multi*** *big & complex* processor organization
- multi-threaded programming model
- the computation is operating system based
- the memory hierarchy is cache-based.

Faced with intense computation, the current SoCs are designed with few standard complex cores and/or some specific accelerators (DSPs or specialized hardware).

D. Integral Parallel Organization

The first embodiment of a system with an IPA is the ***Connex System*** presented in Fig. 2, where we distinguish between the two kinds of computation, ***segregating*** them as:

- **ConnexArray*TM***: many-cell array of execution units (EU) or processing elements (PE) for intense computations [12]
- **Multi-Thread Processor** (MTP) is a *mono-* or *multi*-core **BEAM** processor for complex computations [2].

MTP uses one of its threads to control ***ConnexArray*TM*** in order to execute an ISA containing instructions for both, scalars and vectors. The entire system is programmed in C++ using the library ***VectorC*** [10]. A GNU C++ compiler is developed for the current IPA instruction set.

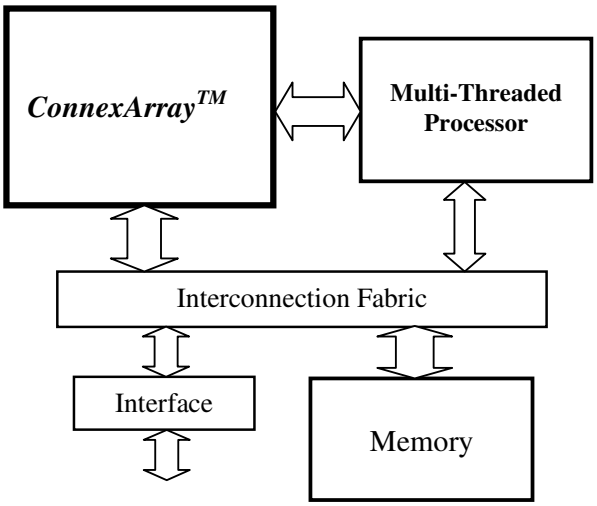

Fig. 2. Integral Parallel Organization: Connex System.

While the intense computation is executed on hundreds or thousands of cores, the complex computation accepts hardly more than 4 cores, because Interconnection Fabric limits less the intense computation. The data stream between Memory and **ConnexArray*TM*** is more predictable than the data and program streams flowing between Memory and MTP.

III. The Complex Computing Part of IPA

The complex part of the computation in Connex System is performed by MTP. Each MTP core is able to execute up to 8 cycle-level interleaved threads. Any active thread is in execution only if its current instruction flow can be executed bubble free. The main effect of BEAM is the increasing of the effective IPC, while saving the area used for the same purpose in the current processors by the branch predictor, superscalar execution units, and L2 cache. Preliminary evaluations show the increasing of performance by 2.5x – 4x, while the area of the engine is reduced with around 60% [2].

IV. The Intense Computing Part of IPA

ConnexArray™ is a cellular array which performs the intense part of the computation [12], [13]. It is already implemented on silicon in 3 versions. The last one, CA1024 (a SoC for the HDTV market, running at 400 MHz, having 1024 EUs, produced in 65 *nm* standard process in March 2008, see Fig. 4), has the following characteristics measured on actual chips:

- 400 GOPS (**G**iga 16-bit integer **OP**erations per **S**econd)
- **120 GOPS/Watt** and **6.25 GOPS/mm²**

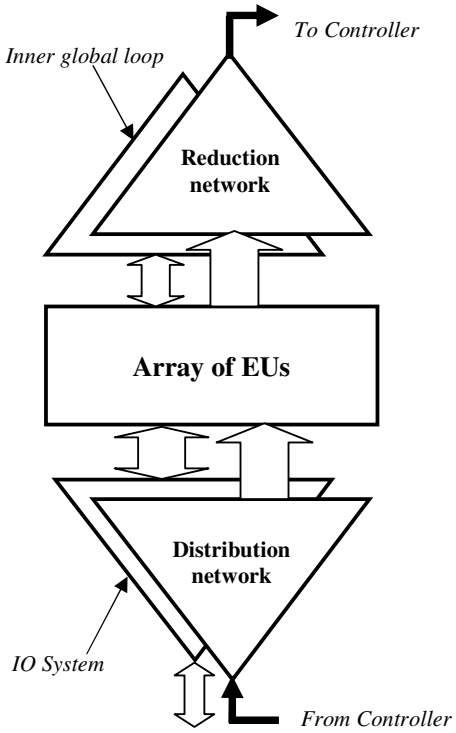

Fig. 3. *ConnexArray™*.

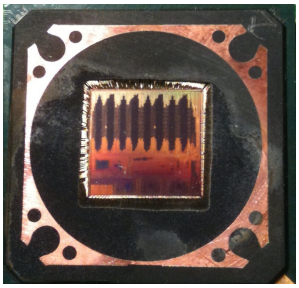

Fig. 4. *CA1024*.

The block diagram of ***ConnexArray™*** is presented in Fig. 3, where a linearly connected array of 1024 EUs receives the same instruction for each EU. The instruction is executed in each EU according with its own state. The reduction network, designed for the most frequently used reduction functions (`add`, `max`, …), sends back to the controller the requested data. An *inner global loop*, closed over the array, is used to classify the EUs according to the selected Boolean. The IO system works in parallel with and transparent to the main computation.

The SoC CA1024 contains besides the 1024 EUs (60% of the chip area) audio/video interfaces, a network of 4 MIPS and a ***time-parallel unit*** (8 16-bit processors).

A. *Basic Operations in ConnexArray™*

Operations on vectors are performed in constant number of cycles. Generic operations are exemplified in the following:

- **full vector ops**: `{carry, v5} = v4 + v3;` the corresponding integer components of the two operand vectors are added; `carry` is a Boolean vector
- **Boolean operation**: `b7 = b3 & b5;` the two Boolean vectors are ANDed component by component
- **predicated execution**: `v1 = b2 ? v3 – v2 : v1;` in any positions **where** `b2 = 1` the corresponding components are subtracted
- **vector rotate**: `v7 = v7 >> n;` the content of vector `v7` is rotated *n* positions right
- **strided load**: `load v5 addr burst stride;` the content of `v5` is loaded from the address `addr`, using bursts `burst`, on a stride of size `stride`
- **scattered load**: `sld v3 v9 addr stride;` `v3` is loaded indirectly using the address vector `v9`
- **strided store**: `store v7 address burst stride;`
- **gathered store**: `gst v4 v3 addr stride;` it is a sort of *indirect* store.

Each cell contains two sub-cells: the scalar unit and the Boolean unit. For input-output operations there is an IO Plane, distributed over the array, whose content is transferred from or to the array's vector memory in one cycle. On the other hand its content is loaded from or stored to the external memory in a number of cycles depending on the IO latency and bandwidth (around 164 clock cycles for a 400 MHz engine with 1024 16-bit EUs). The transfer process is transparent to the computation.

B. *VextorC: the programming language of ConnexArray™*

ConnexArray™ is programmed in ***VectorC***, a C++ language extension [10]. The extension is made by adding new primitive data types and by extending the existing operators to accept the new data types. In ***VectorC*** the conditional statements have become predication statements.

The new data primitives are, for example:

- **int vector**: vector of integers
- **short vector**: vector of shorts
- **selection**: vector of Booleans

Let be the following variable declarations:

```
int i1, i2, i3;
bool b1, b2, b3;
int vector v1, v2, v3;
selection s1, s2, s3;
```

Then a ***VectorC*** statement like: `v3 = v1 + v2;` replaces:

```
for (int i = 0; i < VECTOR_SIZE; i++)
    v3[i] = v1[i] + v2[i];
```

and `s3 = s1 && s2;` replaces this `for` statement:

```
for (int i = 0; i < VECTOR_SIZE; i++)
    s3[i] = s1[i] && s2[i];
```

The scalar statement: `if (b1) {i3 = i1 + i2};` has the correspondent in ***VectorC*** the vector predication statement:

```
WHERE (s1) {v3 = v1 + v2};
```

replacing this nested `for`:

```
for (int i = 0; i < VECTOR_SIZE; i++)
    if (s1[i]) v3[i] = v1[i] + v2[i];
```

The ***VectorC*** library is used as a programming tool for Connex System and also as a simulation environment.

C. *Computational performance*

Connex Architecture implements the infrequent, complex instructions, such as multiplication, division, floating point arithmetic instructions using integer resources sequentially. Thus the specific hardware requested for all infrequent operations uses less than 10% from the total area of the array.

This mode of implementing complex operations generates a specific mode of evaluating the performance of the Connex architecture. Claiming the peak performance is meaningless for our architecture, and deceitful for any kind of architecture.

Let's take the example of peak GFLOPS claimed for a typical general purpose processor: 2-4 GFLOPS. There are two factors limiting the peak performance to the effective performance: (1) *the weight of float instructions in current applications* (it is maximum 24% for the most intense float applications, while the medium weight is 18% [4], [5]), (2) *the stalls in the execution pipeline due to the various hazards* (Intel reports from 48% to 85% clock cycles as stall cycles (see http://www.anandtech.com/print/1909)). Results:

$$effectiveGFLOPS = 0.06 \times peakGFLOPS.$$

For Connex architecture the GFLOPS we claim are effective, because the engine uses for float operations exactly as much GOPS as the applications requests. For example, let be a 1024 32-bit cells array running at 1GHz an application which is not IO bounded. Results peak performance of 1 TOPS. The degree of parallelism is in the range of 30% - 90%. Let us take 60%. Then the effective performance is 0.6 TOPS. For a medium float application results the effective performance: 162 GIPS (Giga Instructions Per Second), out of which 29 GFLOPS, and 133 GIPS in integer operations (each floating point operation is executed in 16 clock cycles). Compared with a standard technology, the Connex approach provides more than two magnitude order more effective GFLOPs (from 121x to 243x).

V. Conclusions

1. The *distinction between complex and intense computation* triggers an efficient segregation which allow two magnitude orders increase for *GOPS/Watt* and *GOPS/mm*^{2} for the intense computation (in ***ConnexArray***^{TM}) and one magnitude order for the complex computation (in ***BEAM*** processor).

2. *IPA* expands efficiently the parallel computation at the level of embedded computing by following the golden rule of increasing the size of the design faster than its complexity.

3. Both, intense part and complex part of the system scales with very small performance penalties.

4. The architectural rule of keeping the logic *small & simple*, performing only frequent operations, avoids big, infrequently used active structures.

6. Programmability deserves an increased attention for architects also because the technological costs in nano-era make unmarketable the pure ASIC approach.

Acknowledgment

The author got a lot of support from main technical contributors to the development of the ***ConnexArray***^{TM} technology, the associated language, and the first applications: Emanuele Altieri, Petronela Bumbăcea, Valeriu Corduneanu, Frank Ho, Radu Hobincu, Mihaela Maliţa, Bogdan Mîţu, Lucian Petrică, Victor Radu Rădulescu, Marius Stoian, Dominique Thiébaut, Tom Thomson, Dan Tomescu.

References

[1] K. Asanovic, et al.: *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report No. UCB/EECS-2006-183.J.

[2] V. Codreanu, R. Hobincu: "Performance Gain from Data and Control Dependency Elimination in Embedded Processors" accepted at *ISETC 2010*. http://phd.arh.pub.ro/resources/beam/isetc2010.pdf

[3] M. Flynn: "Very High-Speed Computing Systems", in *Proceeding of the IEEE*, 54(12), December 1966, p. 1901-1909.

[4] J. Fritts: *Architecture and Compiler Design Issues in Programmable Media Processors*, PhD Thesis, Princeton University, Department of Electrical Engineering Advisor: Prof. Wayne Wolf, 2000.

[5] J. Hennessy, D. Patterson: *Computer Architecture. A Quantitative Approach*, Third edition, Morgan Kaufmann, 2003.

[6] S. Kleene: "General Recursive Functions of Natural Numbers", in *Math. Ann.*, 1936.

[7] M. Maliţa, G. Ştefan, D. Thiébaut: "Not Multi- but Many-Core: Designing Integral Parallel Architectures for Embedded Computation" in *International Workshop on Advanced Low Power Systems* held in conjunction with *21st International Conference on Supercomputing* June 17, 2007 Seattle, WA, USA.

[8] M. Maliţa, G. Ştefan: "On the Many-Processor Paradigm", in: H. R. Arabina (Ed.): *Proceedings of the 2008 World Congress in Computer Science, Computer Engineering and Applied Computing*, vol. PDPTA'08, 2008.

[9] M. Maliţa, G. Ştefan: "Integral Parallel Architecture & Berkeley's Motifs", in *ASAP09 - 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 7-9 July, 2009, Boston, MA, USA, pag. 191-194.

[10] B. Mîţu: "C Language Extension for Parallel Processing", BrightScale research report 2008. http://arh.pub.ro/gstefan/VectorC.ppt

[11] G. Ştefan, M. Maliţa: "Granularity and Complexity in Parallel Systems", in *Proceedings of the 15 IASTED International Conf*, 2004, Marina Del Rey, CA, ISBN 0-88986-391-1, p. 442- 447.

[12] G. Ştefan, A. Sheel, B. Mîţu, T. Thomson, D. Tomescu: "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing", in *Hot Chips: A Symposium on High Performance Chips*, Stanford University, August , 2006

[13] G. Ştefan: "One-Chip TeraArchitecture", in *Proceedings of the 8th Applications and Principles of Information Science Conference*, Okinawa, Japan on 11-12 January 2009.

[14] D. Thiébaut, M. Maliţa: "Pipelining the Connex Array," *BARC07*, Boston, Jan., 2007.