Parallel RISC Architecture. A Functional Approach Based on Backus's FP language

Mihaela Maliţa[1] , and **Gheorghe M. Ştefan**[2]
[1]Saint Anselm College, Manchester, NH, (mmalita@anselm.edu)
[2]PUB, Bucharest, Romania, (gstefan@arh.pub.ro)

Abstract − *The main consequence of building* ad hoc *structured hardware for parallel computation is the huge difficulty we have to program it. The paper discusses a new framework introducing the concept of* **parallel RISC** *engine, as a simple and efficient solution for executing FP like languages proposed by John Backus [2] as an alternative to the von Neumann style of performing computation. A first version for the hardware solution is already implemented in silicon [12].*

Key words: parallel architecture, parallel programming, functional programming, integral parallel architecture, Backus's FP System.

1 Introduction

No one describes better the deadlock of parallel computing than David Patterson which last June wrote in [6] the following about the stage of multi-core industry:

> *"... the semiconductor industry threw the equivalent of a Hail Mary pass when it switched from making microprocessors run faster to putting more of them on a chip-doing so without any clear notion of how such devices would in general be programmed. ... The trick will be to invent ways for programmers to write applications that exploit the increasing number of processors found on each chip without stretching the time needed to develop software or lowering its quality. Say your Hail Mary now, because this is not going to be easy."*

Indeed, parallel computing started wrong, with *ad hoc* constructs considering that more than one machine, more or less sophisticatedly interconnected, will have the brute force to solve the continuously increasing hunger for computing power. The approach was, and is, wrong for two obvious reasons:

- ***programmability*** is very low, because it was proved that is impossible to ignore the sophisticated physical details in order to write efficiently complex programs
- ***portability*** is also very low, because code already written for sequential algorithms is almost impossible to be efficiently translated automatically for various complex parallel engines

and one hidden, but essential reason:

- the lack of ***parallel architecture*** which is supposed: (1) to hide from the programmer the physical details of the actual engine, and (2) to facilitate the automatic translation of the huge sequential software legacy in as much as possible efficient parallel code.

Thus, the problem is to find the shortest path from an appropriate computational model to the simplest possible parallel architecture and to find a validation procedure. Our proposal is:

1. to start with Stephan Kleene's computation model of *partial recursive functions*, because it is a n-based model, i.e., it assumes in the initial statements the use of n variables and/or functions, with n of any size
2. to associate, as the simplest architectural interface, the *Functional Program System* (FP System) proposed by John Backus [2], because of its inherent parallel approach
3. to use *"The Berkeley's View of Parallel Landscape"* [1] as the validation environment for the simplest generic hardware implementation: a ***ConnexArray***^{TM} based engine [9].

We presented the first step in [5], the second step is initiated in this paper, while the last one will be only sketched in this paper and is left to be completed for future works. The second section describes the structure of the parallel RISC engine (pRISC), the structure

which emerged from Kleene's computation model. The third section proves that the FP system of Backus describes efficiently the architecture of the pRISC engine. Preliminary evaluations of the first embodiment of a pRISC architecture – a ***ConnexArray***^{TM} based system – are sketched in the last section.

2 The *parallel RISC* engine

The path from an appropriate computational model to an integral parallel architecture (IPA) is covered in [5] and the associated last silicon implementation – a 65 nm version of a SoC built by *BrightScale*, containing a ***ConnexArray***^{TM} with 1024 execution units – is described in [12]. ***ConnexArray***^{TM} is the hardware considered in our approach as the generic support for a pRISC engine. It is represented in Figure 1, where:

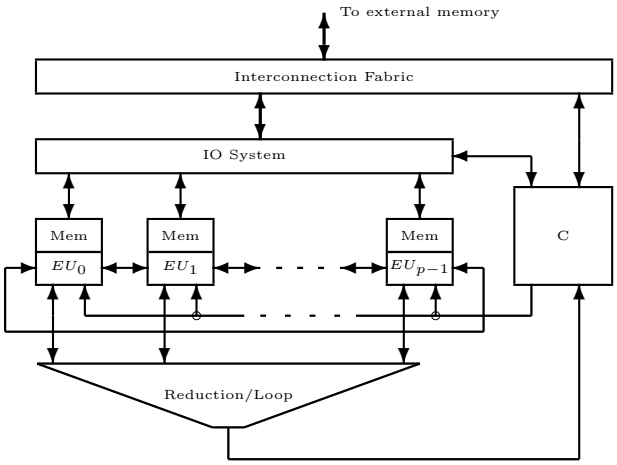

Figure 1: ***ConnexArray***^{TM}**.** The cellular array of p execution units – $EU_0, \ldots, EU_{p-1}$ – each with its own local memory (Mem) is controlled by a sequential engine (C).

C is a general purpose sequential processor used as the controller of the whole system. It issues the sequence of instructions executed in predicated mode in each one of the p execution units (EU).

EU$_i$ is a small & simple EU which executes, according to its internal state, the instruction issued by C.

Mem is the local memory in each processing cell. It is used to store data (the entire cell works as an execution unit) or data & programs (the entire cell works as a processing element, PE).

Reduction/Loop is a tree structured circuit which implements: (1) vector to scalar reduction functions, sending back to C the result, (2) closes a combinational loop over the array of EUs.

IO System transfers data vectors transparently to the processing performed in each one of the p cells

Interconnection Fabric controls the data & program transfers between C & array and the external memory.

The user view of ***ConnexArray***^{TM} is a two-dimension array of scalars (see Figure 2): the constant vector `index`, m p-component vectors stored in the cellular distributed memory (each Mem module stores m scalars) and t p-component vectors distributed in the EU's register files.

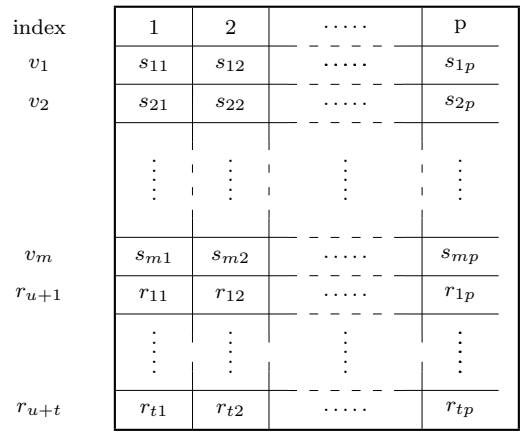

Figure 2: **The user view of *ConnexArray***^{TM}**.** The local memories Mem store m p-component vectors, while the local register files store t p-component vectors. The horizontal (spatial) dimension provides time consuming communication, while the vertical (temporal) dimension allow any efficient virtual interconnection network.

The user of ***ConnexArray***^{TM}sees a $p \times m$ array of scalars. The horizontal (spatial) dimension p is supported by a very simple linear interconnection network, i.e., the distance between two elements on this dimension is in $O(p)$. The simple & small interconnection hardware implies low speed on this dimension. The vertical (temporal) dimension m is supported by the most flexible "interconnection network": the random access mechanism of the Mem modules in each EU. On this dimension the distance between two elements is small & constant. The vertical flexibility can and must be used in order to deal with "big distance" connections in the two-dimension array of variables.

The controller C has a standard organization centered on a register file of u 32-bit scalars. The instruc-

tion set executed by the whole system – C & EUs – is defined on the concatenation of two register files:

- the scalar register file of C
- the vector register file distributed in the p EUs

thus, the instruction set is defined on $t+u$ registers, the first u registers are the scalar registers in C, while the next t registers are vector registers in the p EUs. For example, let be $t = u = 16$. Then, the instruction

```
add r24 r3 r27;
```

adds to each component stored in the vector register 27 the value from the **scalar** register 3 and sends the result in the vector register 24, while the instruction

```
add r24 r18 r27;
```

adds in `r24` the **vector** `r18` with vector `r27`.

The specific instruction for vector processing in ***ConnexArray***^{TM} is the predicated execution expressed as in the following example:

```
where (r25 == 0) add r20 r20 17  ;
    elsewere    xor r20 r20 r20 ;
```

where: in all the cells ***where*** the component of the vector stored in the register 25 is zero, the component of the vector stored in the register 20 is incremented with 17, while ***elsewere*** (where the component of the vector stored in the register 25 is different from zero) the component of the vector stored in the register 20 is cleared.

Another specific function is:

```
where (r20 = <1,5,6>) first r22;
```

which provides in `r22` a vector with 1 only on the first position pointed by the sub-vector `<1,5,6>` in `r20`, and 0 in rest. For example: if
`r20 = <7,4,1,5,7,1,5,6,8,1,5,6,4,2,4,5>`
then the result is
`r22 = <0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0>`.
Those kinds of operations are supported by the combinational loop performed by *Reduction/Loop* circuit.

The full power of the pRISC engine must be *proved* using a theoretical programming model and *evaluated* using the broadest possible functional spectrum.

3 FP system on pRISC

The seminal paper of John Backus *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs* [2] offers the best theoretical environment for proving that the pRISC engine is a good candidate for a generic parallel computer. The Functional Programming Systems (FP Systems) proposed by Backus was introduced as an alternative to the *von Neumann style of programming* – the main paradigm of sequential computation. ***FP Systems can be seen as the definition of parallel systems from the programming point of view***. Therefore, we consider that the pRISC engine, defined in the previous section, must be able to deal efficiently with the *objects*, *functions* and *functional forms* introduced by Backus in his FP Systems.

Objects. The objects defined in FP Systems, *atoms* and *sequences*, can be found in the pRISC architecture in the form of scalars ($x,\ y, \ldots$) managed by the controller C and the vectors ($< x_1, \ldots, x_p >$) processed in the linear array of EUs.

Functions. The following set of primitive functions, almost identical with the set proposed in [2], are examined from the point of view of how they are implemented in our pRISC engine.

Selector:
$i :< x_1, \ldots, x_p > \rightarrow x_i$
The value i is searched in the constant vector `index` (Figure 2), the resulting Boolean vector is used to select the component x_i and to send it to the controller C through the Reduction circuit.

Tail:
$tl :< x_1, \ldots, x_p > \rightarrow < x_2, \ldots, x_p >$
The first element is searched using the `index` vector and is deleted using the serial connections between EUs.

Reverse:
$reverse :< x_1, \ldots, x_p > \rightarrow < x_p, \ldots, x_1 >$
This is the most difficult function for the pRISC engine, because of the very simple interconnection network. It is performed with the help of the IO System, which is featured with hardware support for performing any permutation as an *out of core* function.

Distribute:
$distrl :< y, < x_1, \ldots >> \rightarrow << y, x_1 >, \ldots, < y, x_p >>$
$distrr :< y, < x_1, \ldots >> \rightarrow << x_1, y >, \ldots, < x_p, y >>$
are solved depending on the computational context in two ways: (1) by loading an uniform vector containing on each position the value y, or (2) by issuing a $\{scalar, vector\} \rightarrow vector$ operation with the scalar y broadcasted by C toward each EU.

Length:
$length :< x_1, \ldots, x_p > \rightarrow m$
Is performed in two steps: (1) generates a Boolean vector with 1 on each position corresponding to the p components of the vector, and (2) using the `add` reduction function, the 1s are added and sent to the C controller.

Transpose:
$trans :<< x_{11}, \ldots, x_{1m} >, < x_{21}, \ldots, x_{2m} >, \ldots,$
$< x_{n1}, \ldots, x_{nm} >>\rightarrow$
$<< x_{11}, \ldots, x_{n1} >, < x_{12}, \ldots, x_{n2} >, \ldots,$
$< x_{1m}, \ldots, x_{nm} >>$
is solved on the temporal dimension with no computation because expanding each of the m variables of the initial vectors on the spatial dimension (horizontally), we obtain the n variables of the final vectors vertically, on the temporal dimension:
$< x_{11}, \ldots, x_{1m} >$
$< x_{21}, \ldots, x_{2m} >$
...
$< x_{n1}, \ldots, x_{nm} >$
where each vector is a m-variable "column".

Append:
$apendl :< y < x_1, \ldots, x_p >>\rightarrow< y, x_1, \ldots, x_p >$
$apendr :< y < x_1, \ldots, x_p >>\rightarrow< x_1, \ldots, x_p, y >$
Is solved inserting in the first position the scalar y issued by the controller C.

Rotate:
$rotl :< x_1, \ldots, x_p >\rightarrow< x_2, \ldots, x_p, x_1 >$
$rotr :< x_1, \ldots, x_p >\rightarrow< x_p, x_1, \ldots, x_{p-1} >$
The two-direction linear connection between cells allows the rotate function in both directions. For vectors with less components than the number of EUs the reduction and insert function are used to perform the operations.

Search:
$src :< y, < x_1, \ldots, x_p >>\rightarrow< b_1, \ldots, b_p >$
with $b_i = (y = x_i)\ ?\ 1 : 0$.
The scalar y is issued by the controller and is searched in each EU generating a Boolean vector with 1 on each match position.

Conditioned search:
$csrc :< y, < x_1, \ldots, x_p >, < b_1, \ldots, b_p >>\rightarrow$
$< c_1, \ldots, c_p >$
with $c_i = ((y = x_i)\&b_{i-1})\ ?\ 1 : 0$.
The search is performed only in the cells preceded by a cell when the previous search (`src` or `csrc`) provided a match.

Example. The sequence of operations:
`src d`, `csrc o`, `csrc g`, `csrc _`
identify all the occurrences of the sequence `dog_` in the sequence $< x_1, \ldots, x_p >$. It allows to define the `stream search` operation.

Stream search:
$ssrc :<< y_1, \ldots, y_s >, < x_1, \ldots, x_p >>$
returns a Boolean vector with 1s pointing all the occurrences of $< y_1, \ldots, y_s >$ in $< x_1, \ldots, x_p >$.

Insert data:
$ins :< x, k, < x_1, \ldots, x_p >>\rightarrow$
$< x_1, \ldots, x_{k-1}, x, x_k, \ldots, x_p >$
The scalar k is searched in the `index` vector to identify the insert position for x.

Delete:
$del :< k, < x_1, \ldots, x_p >>\rightarrow$
$< x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_p >$
The scalar k is searched in the `index` vector to identify the delete position.

Functional forms. A functional form depends of functions or objects. The most common functional form is the **composition**, which denotes the application of a sequence of functions f_1,f_2, ..., f_q to x:

$$(f_1 \circ f_2 \circ \ldots \circ f_q) : x \equiv ((f_1 : f_2 : \ldots : (f_q : x) \ldots))$$

Construction:
$[f_1, \ldots, f_n] : x \rightarrow< f_1 : x, \ldots, f_n : x >$
Is parallel speculation. It can be performed on the variable x issued by C and processed in each EU according to the content (data and/or program) of the local memory Mem.

Insert:
$/f :< x_1, \ldots, x_p >\rightarrow f :< x_1, /f :< x_2, \ldots, x_p >>$
Is executed as a reduction function in $O(log\ p)$ time.

Apply to all:
$\alpha f :< x_1, \ldots, x_p >\rightarrow< f : x_1, \ldots, f : x_p >$
Is typical data parallelism. The function f is stored in the program memory used by C.

Condition:
$(p \rightarrow f; g) :< x_1, \ldots, x_p >\rightarrow$
$< (p \rightarrow f; g) : x_1, \ldots, (p \rightarrow f; g) : x_p >$
where:
$(p \rightarrow f; g) : x \rightarrow if\ ((p : x) = 1)\ f : x;\ else\ g : x;$
Is the data parallel predicated execution.

Example. Let be the following definition:

Def $ABS \equiv (/+) \circ (\alpha(lt \rightarrow (- \circ\ reverse); -)) \circ\ trans$

Applying it:

$$ABS :<< x_1, \ldots, x_p >, < y_1, \ldots, y_p >>$$

provides the sum of absolute difference of the two vectors: $< x_1, \ldots, x_p >$ and $< y_1, \ldots, y_p >$.

The functions used by Backus to define FP Systems are efficiently executed by the pRISC engine. The associated architecture is expressed as a FP System. This approach represents a theoretical backup for the specification of a functional language for pRISC like engines.

Thus, parallel computation can start based on a solid theoretical foundation, avoiding risky *ad hoc* constructs. The pRISC engine and the associated FP System based architecture, complemented with multithreaded hardware support (see [13]), is a promising start in saving us from saying "Hail Mary" when deciding what to do to improve our computing machines.

4 Programming pRISC in FP

The four forms of parallelism allowed by the pRISC architecture – data, time, speculative and reduction parallelism (see [13]) – cover theoretically all aspects of the intense computation paradigm [12]. But, the efficiency of pRISC in performing all the aspects of intense computation remains to be proved. In this section we sketch only the complex process of evaluating the proposed pRISC architecture. The best plan for this process is to consider all dwarfs (motifs) outlined in "A View from Berkeley" [1], where is provided a comprehensive presentation of the problems to be solved by the emerging actor on the computing market: the ubiquitous parallel paradigm. Many decades just an academic topic, "parallelism" becomes an important actor on the market after 2001 when the clock rate race stopped. This research report presents 13 computational motifs which cover the main aspects of parallel computing. In this section we will make a preliminary evaluation of them in the context of organization and architecture just introduced in the previous two sections. The cellular network of PEs or EUs has the simplest possible interconnection network. This is both an advantage and a limitation. On one hand, the area of the system is minimized, and it is easy to hide the associated organization from the user, with no loss in programmability or in the efficiency of compilation. On the other hand, some limitations are expected in certain application domains. Follows short comments about how the proposed pRISC architecture works for all of the 13 motifs.

Dense linear algebra. The computation in this domain operates mainly on $N \times M$ matrices. The main operations performed are: matrix addition, scalar multiplication, transposition of a matrix, dot product of vectors, matrix multiplication, determinants of a matrix, Gaussian elimination, solving systems of linear equations and the inverse of a $N \times M$ matrix. Depending on the size of the product $N \times M$ the internal representation of the matrix is decided. If the product is small enough (usually, no bigger than 128), each matrix can be expanded as a vertical vector and associated to one EU, resulting in p matrices represented by $N \times M$ p-component vectors. But, if the product $N \times M$ is big, then q EUs are associated with each matrix, resulting in parallel processing of p/q matrices represented in $N \times M/q$ p-component vectors. For all the operations above listed the computation is usually accelerated almost p times, but not under $(p/(log_2 q)$ times. The most used operation is the inner product of two vectors. It is expressed in FP System as follows:

$$\textbf{Def}\ IP \equiv (/+) \circ (\alpha\times) \circ trans$$

Sparse linear algebra. There are two types of sparse matrices: (1) randomly distributed sparse arrays (represented by few types of lists), (2) band arrays, represented by a stream of short vectors.

For small random sparse arrays, converting them internally into dense array is a good solution. For big random sparse arrays the associated list is operated using the efficient search operations provided by pRISC architecture. Thus, the multiplication of a sparse $N \times M$ matrix with a M-component sparse vector is done in $O(u + v)$, where u is the number of non-zero components in the initial vector and v is the number of non-zero components in the resulting vector.

The band arrays are first transposed using the function `trans` in a number of vectors equal with the width w of the band. Then the main operations are very easy performed using appropriate `rotl` and `rotr` operations. Thus, the multiplication of two band matrices is done on pRISC in $O(w)$.

Spectral methods. The typical examples are: FFT or wavelet computation. Because of the "butterfly" data movement, how the FFT computation is implemented depends on the length of the sample. The spatial and the temporal dimensions of the proposed architecture help the programmer to easily adapt the data representation to result in an almost linear acceleration. In order to reduce, almost eliminate, the slowdown caused by the rotate operations, the stream of samples is loaded using, as much as possible, the temporal dimension of the architecture. In [3] the FFT computation is evaluated on the pRISC architecture (for example: if FFT is considered for 1024 floating point samples the computation is done in 1 clock cycle per sample).

***N*-Body method.** This method fits perfectly on the proposed architecture, because for $j = 0$ to $j = n - 1$ the following equation must be computed:

$$U(x_j) = \sum_i F(x_j, X_i)$$

Each function $F(x_j, X_i)$ is computed on a single EU, and then the sum is a *reduction* operation linearly accelerated by the array. Depending on the value of n, the data is distributed in the processing array using the spatial dimension only, or for large n, both the spatial and the temporal dimension are used. For this motif results an almost linear acceleration.

Structured grids. The grid is distributed on the two dimensions of our array: the spatial dimension and the temporal dimension. Each processor is assigned a column of nodes (on the temporal dimension). It performs

each update step locally and independently of other lines of nodes. Each node has to communicate only with a small number of neighboring nodes on the grid, exchanging data at the end of each step. The system works as a cellular automaton. The computation is accelerated linearly on the proposed architecture.

Unstructured grids. Unstructured grid problems are updates on an irregular grid, where each grid element is updated from its neighbor grid elements. Parallel computation is disturbed by the non-uniformity of the data distribution. In order to solve the non-uniformity problem a preprocessing step is required to generate an easy manageable representation of the grid. We expect moderate performances on pRISC.

Map reduce. The typical example of a map reduce computation is the Monte Carlo method. This method consists in many completely independent computations working on randomly generated data. This type of computation is highly parallel. Sometimes it requires the add reduction function, for which the proposed architecture has special accelerating hardware. The computation is linearly accelerated.

Combinational logic. There are a lot of very different problems falling in this class. We list here only the most important and the most frequently used:

- block processing, exemplified by AES and DES encryption. For example, AES works in 4×4 arrays of bytes, each array is loaded in one EU, and the processing is completely SIMD-like with linear acceleration on the pRISC architecture.
- recursive & non-recursive convolution encoding are computed efficiently using (1) right pipeline propagation in the array, (2) predicated data parallel processing, (3) reduction add function.
- image rotation for black & white or color bit mapped images is performed (1) by loading the $m \times m$ array of pixels into the processing array on both dimensions (spatial and temporal), (2) executing a local transformation, and third restoring the transformed image in the appropriate place.
- route lookup, used in networking; it supposes three data-base like operations: longest match, insert, delete; for all we have functions in the pRISC architecture (similar with: `src`, `csrc`, `ins`, `del`).

Graph traversal. The array of 1024 machines can be used as a "speculative device". Each EU starts with a full graph stored in its data memory, and the computation provides the result when one EU, if any, finds the solution. Limitations are generated by the dimension of the data memory of each EU or by the IO System capabilities. More investigation is needed to evaluate the actual power of pRISC in solving this problem.

Some problems related with graphs are easily solved if matrix computation is involved (example: computing the distance between all the elements of a graph).

Dynamic programming. Viterbi decoding is a typical example presented in [1]. The parallel strategy is to distribute the states among the cells. Each state has its own distinct cell. The inter cell communication is done in a small neighborhood. Each cell receives the stream of data which is thus submitted to a speculative computation. The work done on each processor is similar. The last stage is performed using the functions of the Reduction circuit. The degree of parallelism is limited to the number of state considered by the algorithm.

Back-track and branch & bound. The basic backtracking SAT algorithm, for example, runs on a p-cell engine by choosing $log_2\ p$ literals, instead of one on a sequential machine, assigning for them all the values form $00\ldots0$ to $11\ldots1$, simplifying the formula and then recursively checking if the simplified formula is satisfiable.

For parallel branch & bound we use the case of the Quadratic Assignment Problem. The problem deals with two $N \times N$ matrices: $A = (a_{ij})$, $B = (b_{kl})$. The global cost function:

$$C(p) = \sum_{i}^{n} \sum_{j}^{n} a_{ij} \times b_{p(i)p(j)}$$

must be minimized finding the permutation p of the set $N = \{1, 2, \ldots, n\}$. Dense linear algebra methods already discussed are involved here.

Graphical models. Besides the Viterbi algorithm (already discussed) used for decode, this motif is well represented by parallel hidden Markov models. The architectural features reported in research papers refers to fine-grained SIMD processor arrays connected to each node of a coarse-grained PC-cluster. Thus, a pRISC engine can be used efficiently as an accelerator for general purpose sequential engines.

Finite state machine. The authors of [1] claim that for this motif "nothing helps". But, we consider that the array of cells with their local memory loaded with non-deterministic FSM descriptions work very efficient as a speculative engine for applications such as deep packet inspection, for example.

At the end of this short introductory analysis, *which must be detailed by future investigations*, we claim that for almost all the computational motifs the pRISC architecture performs very well. Maybe for some of the motifs additional multi-threaded computation may be helpful (see [13]).

5 Concluding remarks

The pRISC engine is area and energy efficient. The architecture described and partially evaluated in this paper is based on actual silicon implementations used to measure real performances for intense computations. The hardware results are spectacular (6.5 $GOPS/mm^2$ and 40 $GOPS/Watt$ [12]) and, complemented by the architectural support provided in this paper by a Backus's FP System approach, positions pRISC engine & architecture as a promising solution.

pRISC engine & its architecture are both *small & simple.* The simplicity of the engine allows its hiding behind an efficient architecture. No complex interconnection network between cells and small & simple EUs are the main premisses for a transparent architecture. *"Small is Beautiful"* claims [1]. (See also [8].)

Portability & programmability is high for a simple & generic architecture. High diversity and complexity dominate the main parallel architecture targeted today by the application engineers. What they need is ***only one simple architecture*** for porting existing applications or for developing new ones. The pRISC environment is a promising candidate.

Acknowledgments The authors got a lot of support from the main technical contributors to the development of the *ConnexArray*^{TM} technology, the *BA1024* chip, the associated language, and its first application: E. Altieri, F. Ho, B. Mîţu, M. Stoian, D. Thiebaut, T. Thomson, D. Tomescu.

References

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

[2] J. Backus: "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM*, August 1978, 613-641.

[3] I. Lorentz, M. Malita, R. Andonie: "Fitting FFT onto an Energy Efficient Massively Parallel Architecture", *The Second International Forum on Next Generation Multicore / Manycore Technologies*, June, 2010.

[4] M. Maliţa, G. Ştefan, D. Thiebaut: "Not Multi, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation" in *ACM SIGARCH Comp. Arch. News*, Vol. 35, 5, Dec. 2007.

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

[6] D. Patterson: "The Trouble with Multicore", *IEEE Spectrum*, July 2010.

[7] G. Ştefan, D. Thiebaut: "Memory Engine for the Inspection and Manipulation of Data", *United States Patent 6,760,821*, July 6, 2004; Filed: Aug. 10, 2001.

[8] G. Ştefan, M. Maliţa: "Granularity and Complexity in Parallel Systems", *Proceedings of the 15 IASTED International Conf*, 2004, Marina Del Rey, CA, pp.442-447.

[9] G. Ştefan: "The CA1024: A Massively Parallel Processor for Cost-Effective HDTV", in *Spring Processor Forum: Power-Efficient Design*, May 15-17, San Jose, CA 2006.

[10] G. Ştefan, et al.: "The CA1024: A Fully Programable System-On-Chip for Cost-Effective HDTV Media Processing", in *Hot Chips: A Symposium on High Performance Chips*, Stanford Univ., August, 2006.

[11] G. Ştefan: "The CA1024: SoC with Integral Parallel Architecture for HDTV Processing", in *4th International System-on-Chip (SoC) Conference and Exhibit*, November, Newport Beach, CA, 2006.

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

[13] G. Ştefan: "Integral Parallel Architecture in System-on-Chip Designs". *The 6th International Workshop on Unique Chips and Systems*, Atlanta, GA, USA, December 4, 2010, 23-26.