THE PUBLISHING HOUSE
OF THE ROMANIAN ACADEMY

*PROCEEDINGS OF THE ROMANIAN ACADEMY, Series A,*
Volume 7, Number 3/2006, pp. 000-000

# INTEGRAL PARALLEL COMPUTATION

Gheorghe ŞTEFAN

Universitatea Politehnica Bucuresti & Connex Technology, Inc.
gstafan@connextechnology.com

The natural development of an efficient computing machine starting from the theory of partial recursive functions leads us toward a multi-functional parallel computational structure that we call the Integral Parallel Machine (IPM). The resulting integral parallel architecture (IPA) seeks to optimize use of resources in both time and space; time through instruction-level and speculative parallelism, and space through data-level parallelism. A particular system level implementation of an IPM that is optimized for video processing is described. Test silicon has validated the IPA and a commercial version of an associated IPM delivering 200 16-bit integer GOPS is described.

*Key words:* parallel computing, recursive functions, time parallelism, data parallelism, speculating parallelism, integral parallel architecture.

## 2. INTRODUCTION

Computing workloads in the emerging world of "high definition" digital multimedia more closely resemble workloads associated with scientific computing – or so called *supercomputing* – than general purpose personal computing workloads that dominated markets in the 80's and 90's. Unlike traditional supercomputing applications, which are free to trade performance for "super" size or "super" cost structures, entertainment supercomputing in the consumer electronics imposes extreme constraints of both size and cost.

The traditional approach of implementing highly specialized integrated circuits (ASICs) is no longer cost effective as the R&D investment required for each new application specific IC is less likely to be amortized over the ever shortening product life cycle. At the same time, ASIC designers are able to optimize efficiency and cost through judicious use of parallel processing and parallel data paths. An ASIC designer is free to look for explicit and latent parallelism in every nook and cranny of a specific application or algorithm, and then exploit that in circuits. With the growing need for flexibility, however, an embedded parallel computer is needed that finds the optimum balance between ALL of the available forms of parallelism, yet remains programmable.

Flynn [2] developed perhaps the most widely cited taxonomy of parallel machines. Each parallel machine can be seen as a SIMD (single instruction - multiple data) machine, MIMD (multiple instruction - multiple data) machine or a MISD (multiple instruction - single data) machine. This taxonomy served its purpose well during the era of macro-parallelism - the age of system level parallelism implemented mostly as various configurations of von Neumann computers - where a given taxon is defined more by the interconnect and data coherency methods than by the architecture of each computing element. In the age of SoC implementations that have become the rule as a result of deep sub-micron fabrication technologies, a so-called *structural taxonomy* is less fitting and is even limiting. Real embedded applications tend to make Flynn's taxonomy or other related taxonomies [7] obsolete because they require more than one kind of functional parallelism.

Starting from the model of partial recursive functions, we analyze the relevant canonical circuits and propose a more appropriate taxonomy. The resulting ***functional taxonomy*** consists of data, instruction and speculative parallelism leading us to propose an Integral Parallel Machine (IPM) that features with equal

Recommended by Mihai DRĂGĂNESCU, member of the Romanian Academy

emphasis all the elements of the functional taxonomy, allowing a machine that approaches the efficiency of an ASIC.

We focus this analysis on embedded computation. The Integral Parallel Architecture (IPA) associated with a particular implementation of an IPM in the domain of embedded systems has specific aspects:

- Embedded computation requires more generality/flexibility than that offered by an ASIC, but less generality than that offered by a general purpose processor. Therefore, the instruction set architecture of an embedded computer can be optimized for an application domain, yet remain "general purpose" within that domain
- The flow of processing and data access are more predictable in embedded systems than, for example, PCs. Therefore, simpler "cache" mechanisms is used for both data and programs
- Input-output bottlenecks are more critical in achieving the targeted performance

It is noteworthy that these are also basic features of scientific computing.

The next section of this paper motivates and develops a functional taxonomy of parallel computers. This functional taxonomy is well suited to the real embedded high-performance applications. The approach starts from the computational model of recursive functions and proceeds to derive a conceptual framework for identifying and exploiting each of basic forms of parallelism. From this conceptual framework is extrapolated some basic characteristics of a particular IPM that is under development at Connex Technology, Inc. for high-definition television applications. Performance estimates for the IPM are presented as well.

## 2. CIRCUITS & PARTIAL RECURSIVE COMPUTATION

Before attempting to arrive at a functional taxonomy we must chose a model of computation that is appropriate to the task at hand. Whereas a structural taxonomy takes into account *data & programs*, a functional taxonomy will take into account ***variables & functions***. We will use as referential model for computation the model partial recursive functions [4].

### 2.1 Composition Rule

The circuit associated with the composition rule is the two-level construct associated to:

$$f(x_1,\ldots,x_n)=g(h_1(x_1,\ldots,x_n),\ldots,h_p(x_1,\ldots,x_n)).$$

The first level of circuits computes in parallel $p$ functions $h_i$. The second level computes the function $g$ only when all $p$ circuits have completed their computation on the first level. While *parallelism* is possible on the first level, computation between the two levels is necessarily *sequential*. There are two limit cases. First, the pure *serial composition* is represented by the case $p = 1$; computation is pure sequential. The other limit is a pure *parallel composition* (when $g(x) = x$).

The limiting parallel composition unconditionally grounds a parallel solution for computation. However, we will show that under certain conditions the serial composition can also be used to develop a parallel approach. Parallel composition suggests a "pool" of machines working in parallel to solve a computation consisting in $p$ small independent computations. Serial composition suggests a "pipe" of circuits (programmable machines) receiving a stream of values at one end and providing a stream of results on the other end.

### 2.2 Primitive Recursive Rule

The primitive recursive rule is a sort of serial composition. It is used to apply the same function, $h$, many times in order to compute the function $f(y)$. The primitive recursive rule, implemented in its iterative version, suggests the circuit, which consists of two descendent paths. The first path receives as input the value generated by computing the function $f(0)$. The second starts with 0 on its first stage and computes the increment of the input in each stage. The first path uses the result of the increment in each stage to provide a new computation of the function $h$. The input of the increment circuit is tested on each stage. The test result is used to stop the computation with the output represented by the input of the corresponding final stage.

The uniformity of the "infinite" iterative structure allows us to transform it into two (usually) *simple*, serially connected automata. The first automaton - the *function automaton* - corresponds with the first descendent path and containing circuits whose complexity is given by the complexity of the functions *h* and *f(0)*. The other automaton - the *counter automaton* - corresponds to the second descendent path of increment circuits. In effect, we have converted an unbounded space dimension into an unbounded time dimension; the number of iterations (i.e., time) for computing the function *f(y)* is now proportional to *y* instead of the number of circuit elements. The time remains $T_f(y) \in O(y)$, but the *size* of the circuit is now $O(\log y)$.

In order to provide a bounded solution, we introduce the *data loop* connection where a system uses data returned back from its outputs. Indeed, the function automaton and the counter automaton compute their outputs in the current clock cycle using the values generated on their own outputs in the previous cycle.

The advantage of using **data loops** is the ability to provide a realizable structure for primitive recursive computation. The price for this is the inherent serial nature of this mechanism. The loop must be fed with results provided by a previous computation, i.e., the "pipe" disappeared! Therefore, the stream of data can not, in general, flow throw the two serially connected automata. However, we will show that under certain conditions a solution to "restore" a constant-dimensioned pipe is possible.

### 2.3 Minimization Rule

The minimization rule introduces an additional restriction due to the fact that each intermediate result must be tested before continuing the computation. Now the loop is mandatory for performing the computation from the theoretical point of view, whereas the loop in the previous section is used only because a realizable structure is needed. Now a simple counter automaton proposes values for *f(x)* starting from 0, and a functional module validate/invalidate the proposal, sending back a one-bit signal. It is important to note that the loop associated with the minimization rule is a **control loop** whereas the loop associated the primitive recursive rule was a *data loop*. The former is completely and immutably serial in nature. The latter, however, offers some opportunity for parallelism, albeit at a cost of additional complexity.

## 3. THE FUNCTIONAL TAXONOMY OF PARALLEL COMPUTATION

In today's age of SoC implementations and compulsory functional integration, a functional taxonomy of parallel computing is both more relevant and less constraining. In developing such taxonomy, we will rely on the partial recursive model of computing as a sort of scaffolding.

### 3.1 Data Parallel Computation: the Natural Parallelism

Data parallel computation is performed in systems extended using *parallel composition*. This kind of parallel computing looks trivial, but in practice we are sometimes faced with hidden challenges.

For a programmable system, the functions $h_i$ are, in general, *complex*. Therefore, it is important to pay special attention to how the instructions associated with the *p* functions $h_i$ are stored. In practice, however, we observe that if *p* is big, then the functions $h_i$, for $i = 1, \ldots, p$, tend:

- to be identical: each programmable element receives the same sequence of instructions
- OR to be similar, i.e., each programmable element computing $h_i$ can receive the same sequence of instructions executed according to their unique internal state
- OR to belong to a narrow class, i.e., each programmable element computing $h_i$ receives few instructions executing them selectively based on their unique internal states.

In term of Flynn's taxonomy, the first case is a pure SIMD, the second case is a quasi-SIMD (with some MIMD aspects), and the third case is a pure MIMD, but one where "multi" can be replaced by "few".

The associated programmable structure contains many programmable machines – processing elements (PE) – that execute a stream of one (or "few") instructions issued by one (or "few") sequencers (S), and receiving data from a high-bandwidth input-output (I/O) system. The maximum degree of parallelism is *p*, whereas if few sequencers are used the actual degree of parallelism asymptotically approaches *p*.

### 3.2 Time Parallel Computation: the Induced Parallelism

The sequential computation can be converted into parallel computation under certain circumstances where the sequence of operations, organized as a pipeline, are applied to a long enough stream of data. Previously, two distinct cases of sequential computation were emphasized. The first occurs in the serial composition where the function *g* can not be computed before the computation $h_i$ is not completed. The second is related with the *data loop* used to apply the primitive recursive rule.

In the first case the string of *p* serially composed operations $g_i$ used in computing the function *f(x)* are performed in a pipe, *P*, of *p* machines, each having its own distinct function. If the function *f(x)* is computed for only one value of *x* there are no chances to perform efficient parallel computation because the definition of *f(x)* is pure sequential. However, if the function *f(x)* is computed for a stream, X, of *n* values $x_j$, then efficient parallel computation is possible by inserting the stream *X* into the pipe *P*. The resulting parallel process is characterized by latency proportional to *p* and a degree of parallelism, $\alpha$, given by the following relation:

$$\alpha = pn/(p+n)$$

which tends toward p for $n>>p$.

The sequential form of the *data loop* can be reduced to an efficient parallel form in the case where we need to compute the same function for many different values. There are two solutions for this case. One is to use a data parallel system with the *data loop* closed at the level of each processing element. The second way is to use a pipe of *p* machines each performing the same function, but with the *data loop* closed from the output of the pipe to its input. Each successive level contains data belonging to a distinct computation. The data is extracted from the pipe when the iteration is complete, i.e., the value of *y* is reached, and a new input value is inserted in the open slot. In each clock cycle the machine works to evaluate *p* functions. The performance of this solution is similar to the performance of the pipeline machine used for the serial composition.

Using a pipeline structure for "inducing" parallelism in both serial composition and *data loop* computation introduces **time parallelism**, the efficiency of which is limited by the amount of data involved in computation. For the applications of interest, this parameter is generally quite large.

Because the pipe is formed by programmable processing elements, the propagation from one PE to the next can be programmed to take place every *q* clock cycles. The highest speed, of course, occurs for $q=1$, but practical considerations may warrant $q>1$ in practice.

### 3.3 Speculative Computation: the Forced Parallelism

When computation requires use of the minimization rule, a pipeline structure with $q=1$ becomes impractical. The *control loop* needed in applying the minimization rule hinders the fluency of the pipeline.

We must introduce the third form of functional parallelism, *speculative parallelism*, allowing the pipe to degenerate in a "cross" in order to compute different functions on the same data. Only one result will be selected to continue the process of computation. Thus, speculative execution means to compute more than the functions ask in order to accelerate the overall process. Physically, this requires supplementary hardware which will compute in advance more than will be used in the future.

### 3.4 The Resulting Functional Taxonomy of Parallel Computation

The data structures implied by the previous discussion include *vectors* of scalars, processed in data parallel machines, and *streams* of scalars, processed in time parallel machines. The functional taxonomy follows:

- Data Level Parallelism (DLP): parallel computation working on vectors and having as result, vectors, scalars (by reduction operations) or streams (applied as inputs for time parallel computations). DLP computation is generated by the parallel composition and sometimes by *data loops* imposed by applying the primitive recursive rules.
- Time Level Parallelism (TLP): parallel computation working on streams and having as result streams, scalars (by accumulating operations), or vectors (applied as inputs for data parallel computations). TLP

computation is imposed by the serial composition and sometimes by the occurrence of the *data loops* used to apply the primitive recursive rules.

- Speculative Level Parallelism (SLP): an expanded form of TLP parallel computation working on scalars and having as result vectors reduced immediately by selection to a scalar. SLP computation optimizes the time parallel computation when *control loops*.

A closer analysis discloses the parallelism being possible anytime multiple data to be processed are available. More precisely, all kinds of parallel computations are data intensive. The main conclusion is that: **any computations can be performed efficiently in parallel if they are data intensive.**

## 4. CONNEX TECHNOLOGY IMPLEMENTS AN INTEGRAL PARALLEL ARCHITECTURE

We discuss in this Section a proposed **Integral Parallel Architecture** (IPA) that has been developed by Connex Technology, Inc (Connex) for the HDTV application.

### 4.1 The basic modules of Connex IPA

The basic modules of Connex IPA are described as:

- "Memory System" used to store data and programs, and to organize interface buffers;
- "Input-Output System" consisting of general purpose interfaces and, application specific interfaces
- "Host" is a general purpose controllers used to control the interaction with the external world or to run sequential operations that are neither data intensive or time intensive
- "Data Parallel System" is an array of *small* and *simple* PEs connected by the simplest network
- "Time Parallel System" with speculative capabilities is a dynamically reconfigurable pipe of PEs.

The IPM is organized around a Memory System in order to have maximum flexibility in partitioning the overall computation into tasks performed by different complementary resources.

### 4.2 The Data Parallel System

Based on extensive analysis, simulation and testing of high-performance media processing applications, three principles are emphasized in the Connex design of the Data Parallel System (DPS):

1. ***Spatial locality*** - each processor strongly interacts only in a small neighborhood
2. ***Amdahl's law*** - make the common case fast and avoid adding complex circuits in each PE
3. ***Fit the granularity with application domain*** - the number of PEs, and the instruction set complexity (defined by [5]), must be dimensioned according to the application domain.

With these principles in mind, several defining design elements have been incorporated:

- the strong data locality of the algorithms allows processing elements (PEs) to be connected in a compact linear array with nearest neighbor connections only
- the appropriate number of PEs is between 256 and 1024 (the first embodiment contains 1024 PEs)
- each PE contains a 16-bit ALU, an 8-word register file, a 256 word data memory, and a Boolean machine with an associated 8-bit state register
- the most complex single-cycle operations are add and subtract on 16-bit integers, with a small number of additional single-clock instructions supporting efficient (multi-cycle) multiplication
- the I/O Plan is a 2D network of shift registers with one register per PE
- two independent stack-based instruction sequencers; namely:
  - a 32-bit stack-based Instruction Sequencer (IS) that sequences arithmetic-logical instructions into the array of PEs
  - a 32/128-bit stack-based I/O controller (or "Smart-DMA") used to transfer data between the I/O Plan and the rest of the system
- the Smart-DMA and the IS communicate with each other using interrupts
- data exchange between the array of PEs and the I/O Plan is executed in one clock cycle, and is synchronized using a sequence of interrupts specific to each kind of transfer

- an IS instruction is conditionally executed in each PE depending on a Boolean test of the appropriate bit in the state register, i.e., up to eight toggled states, or "PE masks", may be stored and recalled to effect an instant "virtual reconnection" of the array

The Connex implementation of a DPS consisting of 1024 PEs running at 200 MHz delivers the following performance benchmarks:

- 200 GOPS (OPS: 16-bit integer simple operations per second)
- 5 GFLOPS (FLOPS: 32-bit floating point operations per second)
- 20 GMACS (MACS: multiply-accumulate operations per second with 32-bit integer output)
- 7 GDPS (DPS: 1024 element vector dot products per second with 16-bit signed integer output)

The architecture associated with the Array Controller is covered by a high level compiled language called the **Connex Programming Language** (CPL).

The logical organization of the DPS is best described as the two arrays, one of integers and another of Boolean variables. The integer array contains 256 1024-integer vectors, and the Boolean array contains 8 1024-bit vectors. The Boolean vectors are used to select the components of the integer vectors. The Boolean vector, therefore, may be thought of as a variable mask applied across the array of processing elements and vector operands. For example:

```
where (b0 & b3) v45 = v5 + v243;
```

means that the 46-th vector takes the value of the sum of the 6-th vector and the 244-th vector "row", but ONLY in the components selected by the vector resulting from bit-wise AND-ing the first Boolean vector with the fourth Boolean vector.

### 4.3 The Time Parallel System

Extensive analysis of embedded video, imaging and neural networking applications tells us:

1. the critical section of sequential computation can nearly always be performed efficiently using pipes with tens stages.
2. the speculative computations associated with the critical sequential sections rarely require testing of more than 3 conditions in order to select out the "correct" speculative results

We propose as the ideal TLP structure the dynamically reconfigurable pipeline of $n$ PEs ($n$ usually falls in the interval [8, 63]). The pipe must be able to *reshape dynamically* into a speculative "cross" network containing $m$ PEs (with $2 \leq m \leq 8$) working as a MISD parallel machine, and $n$-$m$ PEs continuing to work as two pipes: one providing the input for the $m$ PEs of the MISD machine, and another using only one result of the speculation made by the MISD machine. Consider the case where the data provided from the $i$-th stage of computation is processed in the $(i+1)$-th stage according to a condition computed in the $(i+2)$-th stage of the pipeline. If the number of speculations is $m$, then $m$ PE's must take the data provided by $PE_i$ and perform independent computations. Finally, $PE_{(i+m+1)}$ must be allowed to select only one result and to neglect the other $(m-1)$ results. The pipeline structure with dynamic speculative capabilities must provide for each PE the ability to dynamically apply as its input the selected output from one of the $m$ previous PEs in the pipeline.

The speed of pipe can be tuned to provide a result every $q$ clock cycles, i.e., each stage of the pipe consists of a computation performed in no more than $q$ clock cycles. If $q = 1$, then each PE executes the same instruction and the entire system works like a fixed function circuit. The speculation is a feature that allows the pipe to work at its maximum speed ($q = 1$). (In the first embodiment $n = 8$, $m = 4$.)

### 4.4 The relative efficiency of Connex IPA

The concept of *architectural efficiency* is related to a chosen application domain. We denote by $\alpha_{arch}(func)$ the architectural efficiency of the architecture *arch*, in the context *func*, and express it formally as:

$$\alpha_{arch}(func) = performance/(resources \times frequency)$$

where the following parameters are defined:

- *performance = func/sec* (e.g., FLOP/sec, OP/sec, MAC/sec, …)
- *resources* $\in$ *{transistors, area, price ...}*

- *frequency = clock_cycles/sec* (expressed in *GHz*)

For example:

$$\alpha_y(FLOP) = (GFLOP/sec)/(area(mm2) \times frequency(GHz))$$

The dimension is *FLOP/cycle/mm*$^2$ represents the architectural efficiency of the architecture *y* in performing FP operations. Similarly, $\alpha_y(OP)$ expresses, in *OP/cycle/mm*$^2$, the efficiency in performing simple operations. Based on RTL simulation benchmarks, the proposed IPM has an associated architecture with FLOP (a 32-bit floating point operation is performed in 40 clock cycles) efficiency approximated by:

$$\alpha_{connex}(FLOP) = (4GFLOP/sec)/(125mm^2 \ \ 0.2GHz) = 0.16\ FLOP/cycle/mm^2$$

Using a particular industry leading general purpose processor as a reference, publicly available benchmarks allow us to approximate the FLOP efficiency by:

$$\alpha_{xxx}(FLOP) = (3GFLOP/sec)/(250mm^2 \ \ 3GHz) = 0.004\ FLOP/cycle/mm^2$$

Therefore, the efficiency of the proposed IPA is **40X greater** in floating-point dominated applications.

Looking at fixed point operations, we approximate the 32-bit GOP efficiency of the proposed IPM as:

$$\alpha_{connex}(OP) = (48\ GOP/sec)/(125mm^2\ 0.2GHz) = 1.92\ OP/cycle/mm^2$$

whereas the reference processor's 32-bit GOP efficiency approximates to:

$$\alpha_{xxx}(OP) = (6\ GOP/sec)/(250mm^2\ 3GHz) = 0.008\ OP/cycle/mm^2$$

Therefore, the efficiency of the proposed IPM is **240X greater** in fixed-point dominated applications.

## 5. SUMARY AND CONCLUSIONS

A functional taxonomy of the available forms of parallel computing fits better with today's highly integrated embedded computation than the structural taxonomy of Flynn. We derived here such a functional taxonomy from a basic computation model, Kleene's partial recursive model, which uses variables and functions as characteristic elements instead of data and programs.

Data parallelism, time parallelism and speculative parallelism are ingredients of the ASIC designer's recipe for any specific application. This heterogeneous parallelism of a circuit must be emulated by integrating programmable machines that are each designed to exploit these ingredients, but in a more flexible manner. The result is the Integral Parallel Machine approach. The development of a particular Integral Parallel Machine under development at Connex for the HDTV application domain is described here.

Parallel processing is most appropriate where high data throughput and the high locality of data dependence are characteristics of the application. Each of the canonical mechanisms of computation - composition, primitive recursion and minimization - can be performed efficiently on parallel machines. However, we conclude that significant parallelism can be exploited besides the usual and obvious data parallelism of an algorithm. In particular, *time parallelism* is triggered by long streams of data received or generated as an intermediate stage. When a high rate of pipeline-generated results is required *speculative parallel* computation is triggered.

The degree of each type of parallelism must be carefully established according to the application domain. Optimizing the number of data parallel PEs, time parallel PEs, and speculative parallel resources must be done with a thorough understanding of the workloads involved. Characteristic structural parameters for an IPM include "width" of the Data Parallel System (i.e., number of PEs), the "depth" of the Time Parallel System (i.e., the number of pipeline stages) and the "height" of the Speculative Parallel System decision tree.

The instruction set architecture must reuse as much as possible the active area of Silicon in order to be able to compete with the ASIC solutions. This means that complex operations that do not occur most of the time should be "composed" of simple operations that do occur most of the time. Circuits that require a lot of logic should be avoided because the "price" paid for them are multiplied by the dimension of parallelism. In summary, we can usually embrace time parallelism and trade a small price in clock cycles per operation for large pay off in architectural efficiency.

## REFERENCES LIST

1. G. CHAITIN: "Algorithmic Information Theory", in *IBM J. Res. Develop*., Iulie, 1977.
2. M. J. FLYNN: "Some computer organization and their effectiveness", *IEEE Trans. Comp*. C21:9 (Sept. 972), pp. 948-960.
3. D. HILLIS: *The Connection Machine*, The MIT Press, Cambridge, Mass., 1985.
4. S. C. KLEENE: "General Recursive Functions of Natural Numbers", in *Math. Ann*., 112, 1936.
5. M. MALITA, G. STEFAN: "Granularity & Complexity in Parallel Systems" in *Proceedings of Modeling and Simulation – Marina del Rey*, CA, 2004.
6. G. STEFAN: *Loops & Complexity in Digital Systems. Lecture Notes on Digital Design* (in progress at: http://arh.pub.ro/people/george_stefan.html), 2002.
7. C. XAVIER, S.S. IYENGAR: *Introduction to Parallel Algorithms*, John Wiley & Sons, Inc, 1

*Received July 12, 2006*