# Reconfigurable Systems
# Past and Next 10 Years

Jean Vuillemin[1]

*Ecole Normale Supérieure*, 45 rue d'Ulm, 75230 Paris cedex 05, France.
This research was partly done at *Hewlett Packard Laboratories*, Bristol U.K.

**Abstract.** A driving factor in *Digital System DS* architecture is the *feature size* of the silicon implementation process. We present Moore's laws and focus on the shrink laws, which relate chip performance to feature size. The theory is backed with experimental measures from [14], relating performance to feature size, for various memory, processor and FPGA chips from the past decade. Conceptually shrinking back existing chips to a common feature size leads to common architectural measures, which we call *normalized*: area, clock frequency, memory and operations per cycle. We measure and compare the normalized *compute density* of various chips, architectures and silicon technologies.
A *Reconfigurable System RS* is a standard processor tightly coupled to a *Programmable Active Memory PAM*, through a high bandwidth digital link. The PAM is a FPGA and SRAM based coprocessor. Through software configuration, it may emulate any specific custom hardware, within size and speed limits. RS combine the flexibility of software programming to the performance level of application specific integrated circuits ASIC. We analyze the performance achieved by P1, a first generation RS [13]. It still holds some significant absolute speed records: RSA cryptography, applications from high-energy physics, and solving the Heat Equation. We observe how the software versions for these applications have gained performance, through better microprocessors. We compare with the performance gain which can be achieved, through implementation in P2, a second-generation RS [16].
Recent experimental systems, such as the *Dynamically Programmable Arithmetic Array* in [19] and others in [14], present advantages over current FPGA, both in storage and compute density. RS based on such chips are tailored for *video processing*, and similar compute, memory and IO bandwidth intensive. We characterize some of the architectural features that a RS must posses in order to be *fit to shrink*: automatically enjoy the optimal gain in performance through future shrinks. The key to scale, for any general purpose system, is to embed memory, computation and communication at a much deaper level than presently done.

## 1 Moore's Laws

Our modern world relies on an ever increasing number of Digital Systems DS: from home to office, through car, boat, plane and elsewhere. As a point in case, the shear economic magnitude of the Millenium Bug [21], shows how futile it would be to try and list *all* the functions which DS serve in our brave new digital world.

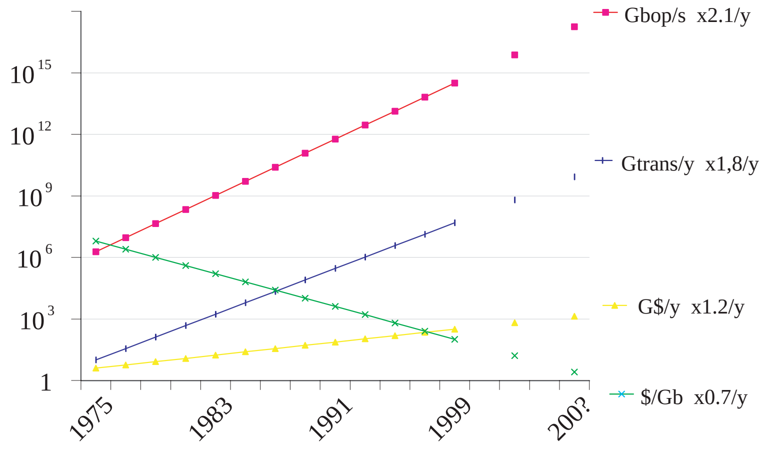

**Fig. 1.** Estimated number and world wide growth rate: $G = 10^9$ transistors fabricated per year; G bit operations computed each second; Billion $ revenues from silicon sold world wide; $ cost per $G = 2^{30}$ bits of storage.

Through recent decades, earth's combined raw compute power has more than doubled each year. Somehow, the market remains elastic enough to find applications, and people to pay, for having twice as many bits automatically switch state than twelve months ago. At least, many people did so, each year, for over thirty years - fig. 1.

An ever improving silicon manufacturing technology meets this ever increasing demand for computations: more transistors per unit area, bigger and faster chips. On the average over 30 years, the cost per bit stored in memory goes down by 30% each year. Despite this drop in price, selling 80% more transistors each year increases revenue for the semi-conductor industry by 20% - fig. 1.

The number of transistors per $mm^2$ grows about 40% each year, and chip size increases by 15%, so:

The number of transistors per chip doubles in about 18 months.

That is how *G. Moore*, one of the founders of *Intel*, famously stated the laws embodied in fig. 1. That was in the late sixties, known since as *Moore's Laws.*

More recently, *G. Moore* [18] points out that we will soon fabricate more transistors per year than there are living ants on earth: an estimated $10^{17}$.

Yet, people buy computations, not transistors. How much computation do they buy? Operating all of this year's transistors at 60 MHz amounts to an aggregate *compute power* worth $10^{24}$ *bop/s* - *bit operation per second*. That would be on the order of 10 million *bop/s* per ant!

This estimate of the world's compute power could well be off by some order of magnitude. What matters is that *computing power at large has more than doubled each year* for three decades, and it should do so for some years to come.

## 1.1 Shrink Laws

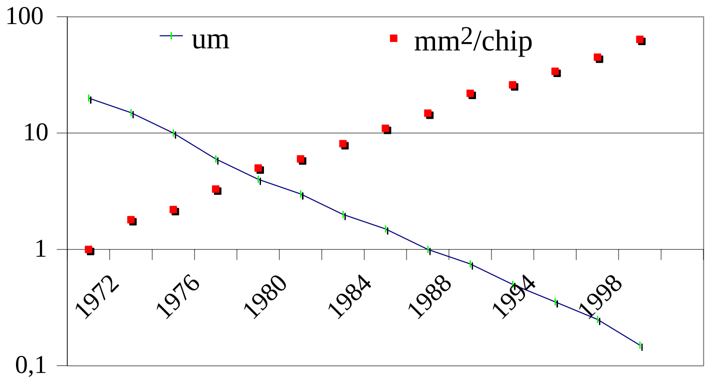

**Fig. 2.** Shrink of the feature size with time: minimum transistor width, in $\mu m = 10^{-6} m$. Growth of chip area - in $mm^2$.

The economic factors at work in fig. 1 are separated from their technological consequences in fig 2. The feature size of silicon chips *shrinks*: over the past two decades, the average shrink rate was near 85% per year. During the same time, chip size has increased: at a yearly rate near 10% for DRAM, and 20% for processors.

The effect on performance of scaling down all dimensions and the voltage of a silicon structure by 1/2: the area reduces by 1/4, the clock delay reduces to 1/2 and the power dissipated per operation by 1/8.

Equivalently, the clock frequency doubles, the transistor density per unit area quadruples, and the number of operations per unit energy is multiplied by 8, see fig. 2. This shrink model was presented by [2] in 1980, and intended to cover feature sizes down to 0.3 $\mu m$ - see fig. 3.

Fig. 4 compares the shrink model from fig. 3 with experimental data gathered in [14], for various DRAM chips, published between in the last decade. The last entry - from [15] - accounts for synchronous SDRAM, where access latency is traded for throughput. Overall, we find a rather nice fit to the model. In fig. 7, we also find agreement between the theoretical fig. 3 and experimental data for microprocessors and FPGA, although some architectural trends appear.

A recent update of the shrink model by Mead [9] covers features down to 0.03 $\mu m$. The optimists conclusion, from [9]:

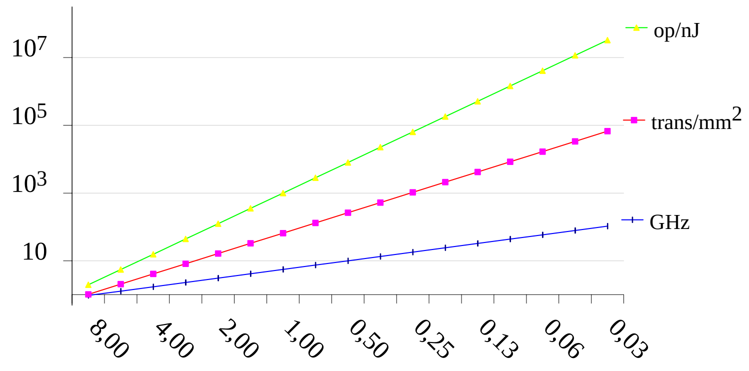

**Fig. 3.** Theoretical chip performance, as the minimum transistor width (feature size) shrinks from 8 to 0.03 micron $\mu m$: transistors per square millimeter; fastest possible chip wide synchronous clock frequency, in giga hertz; number of operations computed, per nano joule.

We can safely count on at least one more order of magnitude of scaling.

The pessimist will observe that it takes 2 pages in [2] to state and justify the *linear* shrink rules; it takes 15 pages in [9], and the rules are *no longer linear.* Indeed, thin oxide is already nearly 20 atoms thick, at current feature size 0.2 $\mu m$. A linear shrink would have it be less than one atom thick, around 0.01 $\mu m$. Other fundamental limits (*quantum mechanical effects, thermal noise, light's wavelength, ...*) become dominant as well, near the same limit. Although *C. Mead* [9] does not *explicitly* cover finer sizes, the *implicit* conclusion is:

We cannot count on two more orders of magnitude of scaling.

Moore's law will thus eventually either run out of fuel - demands for $bop/s$ will some year be under twice that of the previous - or it will be out of an engine - *shrink laws* no longer apply below 0.01 $\mu m$. One likely possibility is some combination of both: feature size will shrink ever more slowly, from some future time on.

On the other hand, there is no fundamental reason why the size of chips cannot keep on increasing, even if the shrink stops. Likewise, we can expect new architecture to improve the currently understood technology path. No matter what happens, how to *best use the available silicon* will long remain an important question. Another good bet: the amount of storage, computation and communication, available in each system will grow, ever larger.

## 2 Performance Measures for Digital Systems

Communication, processing and storage are the three building blocks of DS. They are intimately combined at all levels. At micron scale, wires, transistors

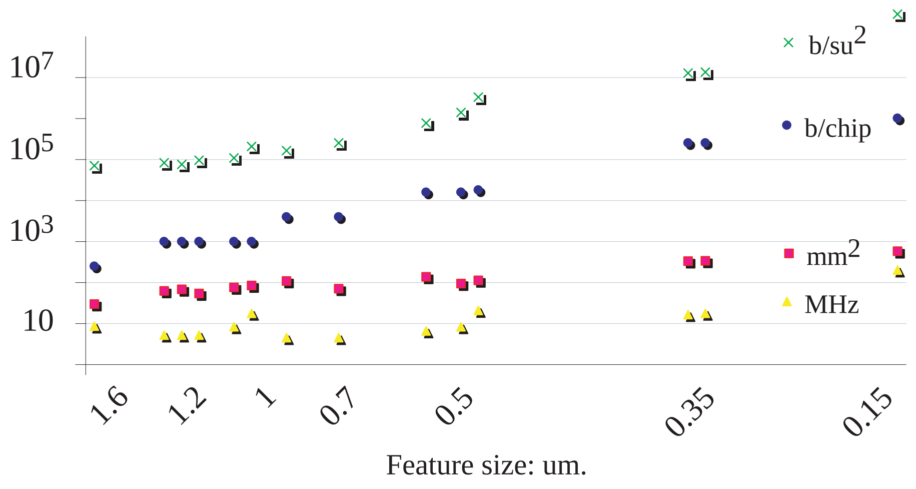

**Fig. 4.** : Actual DRAM performance as feature size shrinks from 0.8 to 0.075 $\mu m$: clock frequency in Mega hertz; square millimeters per chip; bits per chip; power is expressed in bit per second per square micron.

and capacitors implement the required functions. At human scale, the combination of a modem, microprocessor and memory in a PC box does the trick. At planet scale, communication happens through more exotic media - waves in the electromagnetic ether, or optic fiber - at either end of which one finds more memory, and more processing units.

## 2.1 Theoretical performance measures

*Shannon's Mathematical Theory of Communication* [1] shows that physical measures of information (bits $b$) and communication (bits per second $b/s$) are related to the abstract mathematical measure ot statistical entropy $H$, a positive real number $H > 0$. Shannon's theory does not account for the cost of any computation. Indeed, the global function of a communication or storage device is the identity $X = Y$.

On the other hand, source coding for MPEG video is among the most demanding computational tasks. Similarly, *random* channel coding (and decoding), which gets near the optimal for the communication purposes of Shannon as coding blocks become bigger, has a computational complexity which increases exponentially with block size.

The basic question in Complexity Theory is to determine how many operations $C(f)$, are necessary and sufficient for computing a digital function $f$. All operations in the computation of $f$ are accounted for, down to the bit level, regardless of when, where, or how the operation is performed. The unit of measure for $C(f)$ is one *Boolean operation bop*. It is applicable to all forms of computations - sequential, parallel, general and special purpose. Some relevant results (see [5] for proofs):

1. The complexity of $n$ bit binary addition is $5n - 3$ bop. The complexity of computing one bit of sum is 1 *add* = 5 *bop* (full adder: 3 in, 2 out).
2. The complexity of $n$ bit binary multiplication can be reduced, from $6n^2$ *bop* for the naive method (and $4n^2$ *bop* through *Booth Encoding*), down to $c(\epsilon)n^{1+\epsilon}$, for any real number $\epsilon > 0$. As $c(\epsilon) \mapsto \infty$ when $\epsilon \mapsto 0$, the practical complexity of binary multiplication is only improved for $n$ large.
3. Most Boolean functions $f$, with $n$ bits of input and one output, have a *bop* complexity $C(f)$ such that $2n/n < C(f) < 2n/n(2 + \epsilon)$, for all $\epsilon > 0$ and $n$ large enough. To build one, just choose at random! No explicitly described Boolean function has yet been proved to posses more than linear complexity (including multiplication). An efficient way to compute a *random* Boolean function is through a *Lookup Table LUT*, implemented with a RAM or a ROM.

Computation is free in Shannon's model, while communication and memory are free within Complexity Theory. The *Theory of VLSI Complexity* aims at measuring, for all physical realizations of digital function $f$, the combined complexity of *communication, memory*, and *computation.* The *VLSI complexity* of function $f$ is defined with respect to all possible chips for computing $f$. Implementations are all within the same silicon process, defined by some feature size, speed and design rules. Each design computes $f$ within some area $A$, clock frequency $F$ and $T$ clock periods per IO sample. The silicon area $A$ is used for storage, communication and computation, through transistors and wires. Optimal designs are selected, based on some performance measure. For our purposes: minimize the area $A$ for computing function $f$, subject to the real time requirement $F/T < F_{io}$. In theory, one has to optimize among all designs for computing $f$. In practice, the search is reduced to structural decompositions into well known *standard components*: adders, multipliers, shifters, memories, ...

## 2.2 Trading size for speed

VLSI design allows trading area for speed. Consider, for example, the family of adders: their function is to repeatedly compute the binary sum $S = A+B$ of two n bits numbers $A, B$. Fig. 5 shows four adders, each with a different structure, performance, and mapping of the operands through time and IO ports. Let us analyze the VLSI performance of these adders, under simplifying assumptions: $a_{fa} = 2a_r$ for the area (based on transistor counts), and $d_{fa} = d_r$ for the combinatorial delays of $fadd$ and $reg$ (setup and hold delay).

1. Bit serial (base 2) adder $sA2$. The bits of the binary sum appear through the unique output port as a time sequence $s_0$, $s_1$, ..., $s_n$, ... one bit per clock cycle, from least to most significant. It takes $T = n + 1$ cycles per sum $S$. The area is $A = 3a_r$: it is the *smallest of all* adders. The chip operates at clock frequencies up to $F = 1/2d_r$: the *highest possible.*

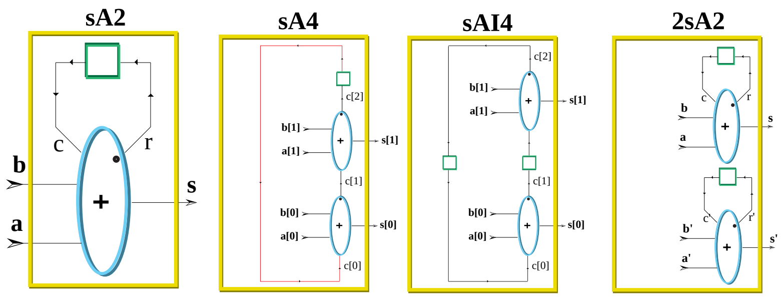

**Fig. 5.** Four serial adders: $sA2$ - base 2, $sA4$ - base 4, $sAI4$ - base 4 interleaved, and $2sA2$ - two independent $sA2$. An oval represents the full adder $fadd$; a square denotes the register $reg$ (one bit synchronous *flip-flop*; the clock is implicit in the schematics).

2. Serial two bits wide (base 4) adder $sA4$. The bits of the binary sum appear as two time sequences $s_0$, $s_2$, ..., $s_{2n}$, ... and $s_1$, $s_3$, ... two bits per cycle, through two output ports. Assuming $n$ to be odd, we have $T = (n+1)/2$ cycles per sum. The area is $A = 5a_r$ and the operating frequency $F = 1/3d_r$.
3. Serial interleaved base 4 adder $sAI4$. The bits of the binary sum $S$ appear as two time sequences $s_0$, *, $s_2$, *, ..., $s_{2n}$, *, ... and *, $s_1$, *, $s_3$, ... one bit per clock cycle, even cycles through one output port, odd through the other. The alternate cycles (the *) are used to compute an independent sum $S'$, whose IO bits (and carries) are interleaved with those for sum $S$. Although it still takes $n+1$ cycles in order to compute each sum $S$ and $S'$, we get *both* sums in so many cycles, at the rate of $T = (n+1)/2$ cycles per sum. The area is $A = 6a_r$ and the maximum operating frequency $F = 1/2d_r$.
4. Two independent bit serial adders $2sA2$. This circuit achieves the same performance as the previous: $T = (n+1)/2$ cycles per sum, area $A = 6a_r$ and frequency $F = 1/2d_r$.

The transformation that unfolds the base 2 adder $sA2$ into the base 4 adder $sA4$ is a special instance of a general procedure. Consider a circuit C which computes some function $f$ in $T$ cycles, within gate complexity $G$ *bop* and memory $M$ bits. The procedure from [11] unfolds $C$ into a circuit $C'$ for computing $f$: it trades cycles $T' = T/2$ for gates $G' = 2G$, at constant storage $M' = M$.

In the case of serial adders, the area relation is $A' = 5A/3 < 2A$, so that $A'T' < AT$. On the other hand, since $F' = 1/3d$ and $F = 1/2d$, we find that $A'T'/F' > AT/F$. An equivalent way to measure this, is to consider the density of full adders fadd per unit area $a_{fa} = 2a_r$, for both designs $C$ and $C'$: as $2/A = 0.66 < 4/A' = 0.8$, the unfolded design has a better $fadd$ density than the original. Yet, since $F' = 1.5F$, the *compute density* - in $fadd$ per unit area and time $d_{fa} = d_r$ - is lower for circuit $C'$: $F/A = 0.16 > 2/A'F' = 0.13$. When we unfold from base 2 all the way to base $2n$, the carry register may be simplified away: it is always 0. The fadd densities of this $n$-bit wide carry propagate adder is 1 per unit area, which is optimal; yet, as clock frequency is $F = 1/n$, the compute density is low: $1/n$.

Circuits $sAI4$ and $2sA2$ present two ways of *optimally* trading time for area, at *constant operator and compute density*. Both are instances of general methods, applicable to any function $f$, besides binary addition. From any circuit $C$ for computing $f$ within area $A$, time $T$ and frequency $F$, we can derive circuits $C'$ which optimally trades area $A' = 2A$ for time $T' = T/2$, at constant clock frequency $F' = F$. The trivial unfolding constructs $C' = 2C$ from two independent copies of $C$, which operate on separate IO. So does the interleaved adder $sAI4$, in a different manner. Generalizing the interleaved unfolding to arbitrary functions does not always lead to an optimal circuit: the extra wiring required may force the area to be more than $A' > 2A$. Also note that while these optimal unfolding double the throughput ($T = n/2$ cycles per add), the *latency* for each individual addition is not reduced from the original one ($T = n$ cycles per addition). We may constrain the unfolded circuit to produce the IO samples in the standard order, by adding reformatting circuitry on each side of the IO: a buffer of size $n$-bit, and a few gates for each input and output suffice. As we account for the extra area (for *corner turning*), we see that the unfolded circuit is no longer optimal: $A' > 2A$. For a complex function where a large area is required, the loss in *corner turning* area can be marginal. For simpler functions, it is not.

In the case of addition, area may be optimally traded for time, for all integer data bit width $D = n/T$, as long as $D < \sqrt{n}$. Fast wide $D = n$ parallel adders have area $A = nlog(n)$, and are structured as binary trees. The area is dominated by the wires connecting the tree nodes, their drivers (the longer the wire, the bigger the driver), and by pipelining registers, whose function is to reduce all combinatorial delays in the circuit below the clock period $1/F$ of the system.

*Transitive functions* permute their inputs in a rich manner (see [4]): any input bit may be mapped - through an appropriate choice of the external controls - into any output bit position, among $N$ possible per IO sample. It is shown in [4] that computing a transitive function at IO rate $D = NF/T$, requires an area $A$ such that:

$$A > a_m N + a_{io} D + a_w D^2, \tag{1}$$

where $a_m$, $a_{io}$ and $a_w$ are proportional to the area per bit respectively required for memory, IO and communication wires. Note that the gate complexity of a transitive function is zero: input bit values are simply permuted on the output. The above bound merely accounts for the area - IO ports, wires and registers - which is required to acquire, transport and buffer the data at the required rate. Bound (1) applies to shifters, and thus also to multipliers. Consider a multiplier that computes $2n$-bit products on each cycle, at frequency $F$. The wire area of any such multiplier is proportional to $n^2$, as $T = 1$ in (1). For high bandwidth multipliers, the area required for wires and pipelining registers is bigger than that for arithmetic operations.

The bit serial multiplier (see [11]) has a minimal area $A = n$, high operating frequency $F$, and it requires $T = 2n$ cycles per product. A parallel nave multiplier has area $A' = n^2$ and $T' = 1$ cycle per product. In order to maintain high frequency $F' = F$, one has to introduce on the order of $n^2$ *pipelining registers*, so (perhaps) $A' = 2n^2$ for the *fully pipelined* multiplier. These are two extreme points in a range of optimal multipliers: according to bound (1), and within a constant factor. Both are based on nave multiplication, and compute $n^2$ mul per product. High frequency is achieved through deep pipelining, and the latency per multiplication remains proportional to $n$. In theory, latency can be reduced to $T$, by using reduced complexity $n^{1+\epsilon}$ shallow multipliers (see [3]); yet, shallow multipliers have so far proved bigger than nave ones, for practical values such as $n < 256$.

### 2.3 Experimental performance measures

Consider a VLSI design with area $A$ and clock frequency $F$, which computes function $f$ in $T$ cycles per $N$-bit sample. In theory, there is another design for $f$ which optimally trades area $A' = 2A$ for cycles $T' = T/2$, at constant frequency $F' = F$. The frequency $F$ and the $AT$ product remain invariant in such an optimal tradeoff. Also invariant:

- The gate density (in $bop/mm^2$), given by $D_{op} = c(f)/A = C(f)/AT$. Here $c(f)$ is the *bop* complexity of $f$ per cycle, while $C(f)$ is the bop complexity per sample.
- The compute density (in $bop/smm^2$) is $c(f)F/A = FD_{op}$.

Note that trading area for time at constant gate and compute density is equivalent to keeping $F$ and $AT$ invariant.

Let us examine how various architectures trade size for performance, in practice. The data from [14] tabulates the area, frequency, and feature size, for a representative collection of chips from the previous decade: sRAM, DRAM, mPROC, FPGA, MUL.

The normalized area $A/\lambda^2$ provides a performance measure that is independent of the specific feature size $\lambda$. It leads [14] to a quantitative assessment of the gate density for the various chips, fig. 6 and 7.

Unlike [14], we also normalize clock frequency: the product by the operation density is the normalized compute power. To define the normalized the system clock frequency $\phi$, we follow [9] and use $\phi = 1/100\tau(\lambda)$, where $\tau(\lambda)$ is the minimal inverter delay corresponding to feature size $\lambda$.

- The non linear formula used for $\tau((l) = cl^e$ is taken from [9]: the exponent $e = 1 - \epsilon(l)$ decreases from 1 to 0.9 as $l$ shrinks from 0.3 to 0.03 $\mu m$. The non linear effect is not yet apparent in the reported data. It will become more significant with finer feature sizes, and clock frequency will cease to increase some time before the shrink itself stops.
- The factor 100 leads to normalized clock frequencies whose average value is 0.2 for DRAM, 0.9 for SRAM, 2 for processors and 2 for FPGA.

In the absence of architectural improvement, the normalized gate and compute density of the same function on two different feature size silicon implementations should be the same, and this indicates an optimal shrink.

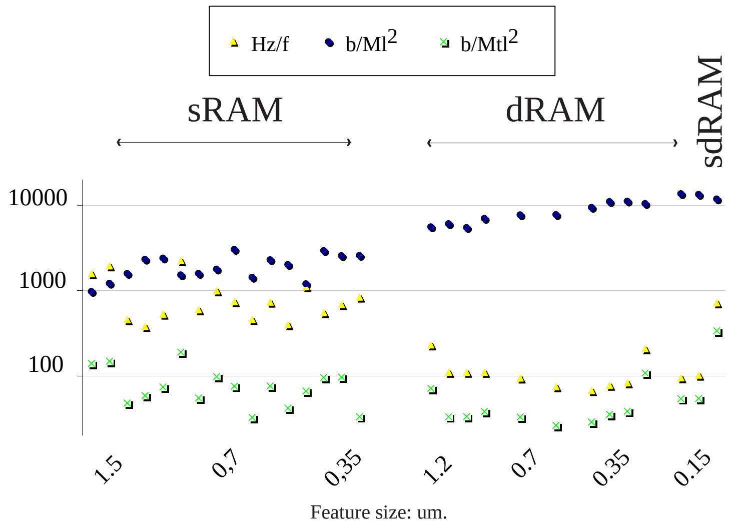

**Fig. 6.** Performance of various SRAM and DRAM chips, within to a common feature size technology: normalized clock frequency $Hz/\phi$; bit density per normalized area $10^6\lambda^2$; binary gate operations per normalized area per normalized clock period $1/\phi$.

- The normalized performance figures for SRAM chips in fig. 6 are all within range: from one half to twice the average value.
- The normalized bit density for DRAM chips in the data set is 4.5 times that of SRAM. Observe in fig. 6 that it has increased over the past decade, as the result of improvements in the architecture of the memory cell (*trench* capacitors). The average normalized speed of DRAM is 4.5 times slower than SRAM. As a consequence the average normalized compute density of SRAM equals that of DRAM. The situation is different with SDRAM (last entry in fig. 6): with the storage density of DRAM and nearly the speed of SRAM, the normalized compute density of SDRAM is 4 times that of either: a genuine improvement in memory architecture.

A *Field Programmable Gate Array FPGA* is a mesh made of programmable gates and interconnect [17]. The specific function - Boolean or register - of each gate in the mesh, and the interconnection between the gates, is coded in some binary bitstream, specific to function f, which must first be downloaded into the *configuration memory* of the device. At the end of configuration, the FPGA switches to user mode: it then computes function $f$, by operating just as any regular ASIC would.

The comparative normalized performance figures for various recent microprocessors and FPGA is found in fig. 7.

- Microprocessors in the survey appear to have maintained their normalized compute density, by trading *lower* normalized operation density, for a *higher*

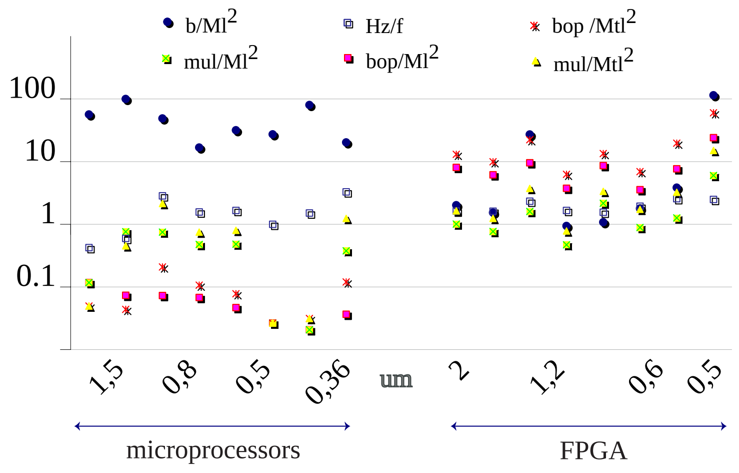

**Fig. 7.** Performance of various microprocessor and FPGA chips from [14], within a common feature size technology: normalized clock frequency $Hz/\phi$; normalized bit density; normalized gate and compute density: for Boolean operations, additions and multiplication's.

normalized clock frequency, as feature size has shrunk. Only the microprocessors with a built-in multiplier have kept the normalized compute density constant. If we exclude multipliers, the normalized compute density of microprocessors has actually decreased through the sample data.

– FPGA have stayed much closer to the model, and normalized performances do not appear to have changed significantly over the survey (rightmost entry excluded).

## 3 Reconfigurable Systems

A *Reconfigurable System RS* is a standard sequential processor (the host) tightly coupled to a *Programmable Active Memory PAM*, through a high bandwidth link. The PAM is a reconfigurable processor, based on FPGA and SRAM. Through software configuration, the PAM emulate any specific custom hardware, within size and speed limits. The host can write into, and read data from the PAM, as with any memory. Unlike conventional RAM, the PAM processes data between write and read cycles: it an active memory. The specific processing is determined by the contents of its configuration memory. The content of configuration memory can be updated by the host, in a matter of milliseconds: it is programmable.

RS combine the flexibility of software programming to the performance level of application specific integrated circuits ASIC. As a point in case, consider the system P1 described in [13]. From the abstract of that paper:

We exhibit a dozen applications where PAM technology proves superior, both in performance and cost, to every other existing technology, including supercomputers, massively parallel machines, and conventional custom hardware.

The fields covered include computer arithmetics, cryptography, error correction, image analysis, stereo vision, video compression, sound synthesis, neural networks, high-energy physics, thermodynamics, biology and astronomy.

At comparable cost, the computing power virtually available in a PAM exceeds that of conventional processors by a factor 10 to 1000, depending on the specific application, in 1992.

RS P1 is built from chips available in 92 - SRAM, FPGA and processor. Six long technology years later, it still holds at least 4 significant absolute speed records. In theory, it is a straightforward matter to port these applications on a state of the art RS, and enjoy the performance gain from the shrink. In the practical state of our CAD tools, porting the highly optimized P1 designs on oher systems would require time and skills. On the other hand, it is straightforward to estimate the performance without doing the actual hardware implementation. We use the Reconfigurable System P2 [16] - built in 97 - to conceptually implement the same applications as P1, and compare. The P2 system has 1/4 the physical size and chip count of P1. Both have roughly the same logical size (4k CLB), so the applications can be transferred without any redesign. The clock frequency is 66 MHz on P2, and 25MHz on P1 (and 33MHz for RSA). So, the applications will run at least twice faster on P2 than on P1. Of course, if we compare equal size and cost systems, we have to match P1 against 4P2, and the compute power has been multiplied by at least 8. This is expected by the theory, as the feature size of chips in P1 is twice that of chips in P2.

What has been done [20] is to port and run on recent fast processors, the software version for some of the original P1 applications. That provides us with a technology update on the respective compute power of RS and processors.

### 3.1 3D Heat Equation

The fastest reported software for to solving the Heat Equation on a supercomputer, is presented in [6]. It is based on the finite differences method. The Heat Equation can be solved more efficiently on specific hardware structures [7]:

- Start from an initial state - at time $t\Delta t$ - of the discrete temperatures in a discrete 3D domain, all stored in RAM.
- Move to the next state - at time $(t+1)\Delta t$ - by traversing the RAM three times, along the x, y and z axis.
- On each traversal, the data from the RAM feeds a pipeline of *averaging operators*, and the output of the pipeline is stored back in RAM.

Each *averaging operator* computes the average value $(a_t + a_{t+1})/2$ of two consecutive samples $a_t$ and $a_{t+1}$. In order to be able to reduce the precision of internal

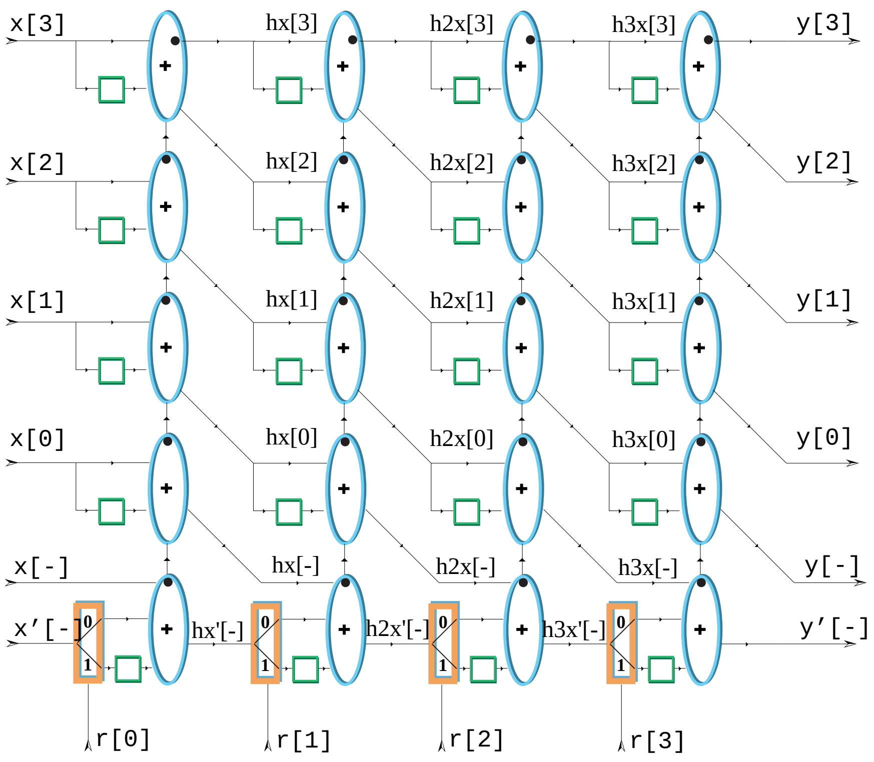

**Fig. 8.** Schematics of a hardware pipeline for solving the Heat equation. It is drawn with a pipeline depth of 4, and bit width of 4, plus 2 bits for randomized round off. The actual 1 pipeline is 256 deep, and 16+2 wide. Pipelining registers, which allow the network to operate at maximum clock frequency, are not indicated here. Neither is the random bit generator.

temperatures down to 16 bits, it is necessary, when division by two is odd, to distribute that low-order bit randomly between the sample and its neighbor. All deterministic round-off schemes lead to parasitic effects that can significantly perturb the result. The pseudo-randomness is generated by a 64-bit *linear feed-back shift-register LFSR*. The resulting pipeline is shown in fig. 8. Instead of being shifted, the least significant sum bit is either delayed or not, based on a random choice in the LFSR.

P1 standing design can accurately simulate the evolution of temperature over time in a 3D volume, mapped on $512^3$ discrete points, with arbitrary power source distributions on the boundaries. In order to reproduce that computation in real time, it takes a 40,000 MIPS equivalent processing power: 40 G instructions per second, on $32b$ data. This is out of the reach of microprocessors, at least until 2001.

### 3.2 High Energy Physics

The *Transition Radiation Tracker TRT* is part in a suite of benchmarks proposed by CERN [12]. The goal is to measure the performance of various computer architectures in order to build the electronics required for the Large Hadron Collider LHC, soon after the turn of the millennium. Both benchmarks are challenging, and well documented for a wide variety of processing technologies, including some of the fastest current computers, DSP-based multiprocessors, systolic arrays, massively parallel arrays, Reconfigurable Systems, and full custom ASIC based solutions.

The TRT problem is to find straight lines (particle trajectories) in a noisy digital black and white image. The rate of images is at 100 kHz; the implied IO rate close to 200 MB/s, and the low latency requirement (2 images) preclude any implementation solution other specialized hardware, as shown by [12].

The P1 implementation of the TRT is based on the *Fast Hough Transform* [10], an algorithm whose hardware implementation trades computation for wiring complexity. To reproduce the P1 performance reported in [12], a 64-bit sequential processor needs to run at over 1.2 GHz. That is about the amount of computation one gets, in 1998, with a dual processor, 64-bit machine, at 600 MHz. The required external bandwidth (up to 300 MB/s) is what still keeps such application out of current microprocessor reach.

### 3.3 RSA cryptography

The P1 design for RSA cryptography combines a number of algorithm techniques, presented in [8]. For 512-bit keys, it delivers a decryption rate in excess of 300 $kb/s$, although it uses only half the logical resources available in P1.

The implementation takes advantage of hardware reconfiguration in many ways: a rather different design is used for RSA encryption and decryption; a different hardware modular multiplier is generated for each different prime modulus: the coefficients of the binary representation of each modulus is hardwired into the logical equations of the design. None of these techniques is readily applicable to ASIC implementations, where the same chip must do both encryption and decryption, for all keys.

As of printing time, this design still holds the acknowledged shortest time per block of RSA, all digital species included. It is surprising that it has held five years against other RSA hardware. According to [20], the record will go to a (soon to be announced) Alpha processor (one 64b multiply per cycle, at 750MHz) running (a modified version of) the original software version in [8]. We expect the record to be claimed back in the future by a P2 RSA design; yet, the speedup between P1 was 10x reported in 92, and we estimate that it should be only be 6x on 2P2, in 97. The reason: the fully pipelined multiplier, found in recent processors, is fully utilized by RSA software. A normalized measure of the impact of multiplier on theoretical performance can be observed in fig. 7.

For the Heat Equation, the actual performance ratio between P1 and the fastest processor (64b, 250MHz) was 100x in 92; with 4P2 against the 64b, 750MHz processor, the ratio should be over 200x in 98. Indeed, the computation in fig. 8 combines 16 b add and shift, with Boolean operations on three low order bits: software is not efficient, and the multiplier is not used.

## 4 What will Digital Systems shrink to?

Consider a DS whose function and real time frequency remain fixed, once and for all. Examples: digital watch, $56kb/s$ modem and GPS.

How does such DS shrink with feature size?

To answer, start from the first chip (feature size l) which computes function $f$: area $A$, time $T$, and clock frequency $F$. Move in time, and shrink feature size to l/2. The design now has area $A' = A/4$, and the clock frequency doubles $F' = 2F$ ($F' = (2-\epsilon)F$ with non-linear shrink). The number of cycles per sample remains the same: $T' = T$. The new design has twice (or $2-\epsilon$) the required real time bandwidth: we can (in theory) further fold space in time: produce a design $C''$ for computing $f$ within area $A'' = A'/2 = A/8$ and $T'' = 2T$ cycles, still at frequency $F'' = F' = 2F$. The size of any fixed real time DS shrinks very fast with technology, indeed. At the end of that road, after so many hardware shrinks, the DS gets implemented in software.

On the other hand, microprocessors, memories and FPGA actually grow in area, as feature size shrinks. So far, such commodity products have each aimed at delivering ever more compute power, on one single chip. Indeed, if you look inside some recent digital device, chances are that you will see mostly three types of chips: RAM, processor and FPGA. While a specific DS shrinks with feature size, a general purpose DS gains performance through the shrink, ideally at constant normalized density.

### 4.1 System on a chip

There are compelling reasons for wanting a Digital System to fit on a single chip. Cost per system is one. Performance is another:

- Off-chip communication is expensive, in area, latency and power. The bandwidth available across some on-chip boundary is orders of magnitude that across the corresponding off-chip boundary.
- If one quadruples the area of a square, the perimeter just doubles. As a consequence, when feature size shrinks by $1/x$, the internal communication bandwidth grows faster than the external IO bandwidth: $x^{3-\epsilon}$ against $x^{2-\epsilon}$. This is true as long as silicon technology remains planar: transistors within a chip, and chips within a printed circuit board, must all layed out side by side (not on top of each other).

### 4.2 Ready to Shrink Architecture

So far, normalized performance density has been maintained, through the successive generations of chip architecture.

Can this be sustained in future shrinks?

A dominant consideration is to keep up the system clock frequency $F$. The formula for the normalized clock frequency $1/\phi = 100\tau(\lambda)$ implies that each combinatorial sub-circuit within the chip must have delay less than 100x that of a minimal size inverter. The depth of combinatorial gates that may be traversed along any path between two registers is limited. The length of combinatorial paths is limited by wire delays. It follows that only finitely many combinatorial structures can operate at normalized clock frequency $\phi$. There is a limit to the number $N$ of IO bits to any combinatorial structure which can operate at such a high frequency. In particular, this applies to combinatorial adders (say $N < 256$), multipliers (say $N < 64$) and memories.

### 4.3 Reconfigurable Memory

The use of fast SRAM with small block size is common in microprocessors: for registers, data and instruction caches. Large and fast current memories are made of many small monolithic blocks. A recent SDRAM is described in [15]: $1Gb$ stored as 32 combinatorial blocks of $32Mb$ each. A 1.6 $GB/s$ bandwidth is obtained: data is $64b$ wide at 200MHz.

By the argument from the preceding section, a large $N$ bit memory must be broken into $N/B$ combinatorial blocks of size $B$, in order to operate at normalized clock frequency $F = \phi$. A $N$ bit memory with minimum latency may be constructed, through recursive decomposition into 4 quad memories, each of size $N/4$ - layed out within one quarter of the chip. The decomposition stops for $N = B$, when a block of combinatorial RAM is used. The access latency is proportional to the depth $log(N/B)$ of the hierarchical decomposition.

A *Reconfigurable Memory RM* is an array of high speed dense combinatorial memory blocks. The blocks are connected through a reconfigurable pipelined wiring structure. As with FPGA, the RM has a configuration mode, during which the configuration part of the RM is loaded. In user mode, the RM is some group of memories, whose specific interconnect and block decomposition is coded by the configuration. One can trade data width for address depth, from $1 \times N$ to $N/B \times B$ in the extreme cases.

A natural way to design a RM is to imbed blocks of SRAM within a FPGA structure. In CHESS [19], the atomic SRAM block has size $8 \times 256$. The SRAM blocks form a regular pitch matrix within the logic, and it occupies about 30% of the area. As a consequence, the storage density of CHESS is over 1/3 that of a monolithic SRAM. This is comparable to the storage density of current microprocessors; it is much higher than the storage density of FPGA, which rely (so far) on off-chip memories.

After configuration, the FPGA is a large array of small SRAM: each is used as LUT - typically LUT4. Yet, most of the configuration memory itself is not accessible as a computational resource by the application. In most current FPGA, the process of downloading the configuration is serial, and it writes the entire configuration memory. In a 0.5x shrink, the download time doubles: 4x bits at (2-e)x the frequency. As a consequence, the download takes about 20 ms on P1, and 40 ms on P2.

A more efficient alternative is found in the X6k [17] and CHESS: in configuration mode, configuration memory is viewed as a single SRAM by the host system. This allows for faster complete download. An important feature is the ability to randomly access the elements of the configuration memory. For the RSA design, this allows for very fast partial reconfigurations: as we change the value of the 512*b* key which is hardwired into the logical equations, only few of the configuration bits have to updated. Configuration memory can also be used as a general-purpose communication channel between the host and the application.

## 4.4 Reconfigurable Arithmetic Array

The normalized gate density of current FPGA is over 10x that of processors, both for Boolean operations and additions - fig. 7. This is no longer true for the multiply density, where common FPGA barely meets the multiply density of processors which recently integrate one (or more) pipelined floating point multiplier.

The arithmetical density of RS can be raised: MATRIX [DeHon], which is an array of 8b ALU, with Reconfigurable Interconnect, does better than FPGA. CHESS is based on 4*b* ALU, which are packed as the *white* squares in a chessboard. It follows that CHESS has an arithmetic density which is near 1/3 that of custom multipliers. The synchronous registers in CHESS are 4*b* wide, and they are found both within ALU and routing network, to as to facilitate high speed systematic pipelining.

Another feature of CHESS [19], is that each *black* square in the chessboard may be used either as a switchbox, or as a memory, based on a local configuration bit. As a switchbox, it operates on 4*b* *nibbles*, which are all routed together. In memory mode, it may implement various specialized memories, such as a depth 8 shift register, in place of eight 4b wide synchronous registers. In memory mode, it can also be used as a 4b in, 4b out $4LUT4$. This feature provides CHESS with a LUT4 density which is as high as for any FPGA.

### 4.5 Hardware or Software?

In order to implement digital function $Y = f(X)$, start from a specification by a program in some high level language. Some work is usually required to have the code match the digital specification, bit per bit - high level languages provide little support for funny bit formats and operations beneath the word size.

Once done, compile and unwind this code so as to obtain the run-code $C_f$. It is the sequence of machine instructions, which a sequential processor executes, in order to compute output sample $Y_t$ from input sample $X_t$. This computation is to be repeated indefinitely, for consecutive samples: $t$=0, 1, For the sake of simplicity, assume the run-code to be straight-line: each instruction is executed once in sequence, regardless of individual data values; there is no conditional branch. In theory, the run-code should be one of minimal length, among all possible for function $f$, within some given instruction set. Operations are performed in sequence through the Arithmetic and Logic Unit ALU of the processor. Internal memory is used to feed the ALU, and provide (memory-mapped) external IO. For $W$ the data width of the processor, the complexity of so computing f is $W|C_f|$ bop per sample. It is greater than the gate complexity $G(f)$. Equality $|C_f| = G(f)/W$ only happens in ideal cases. In practice, the ratio between the two can be kept close to one, at least for straight-line code.

The execution of run-code Cf on a processor chip at frequency $F$ computes function $f$ at the rate of $F/C$ samples per second, with $C = |C_f|$. The feasibility of a software implementation of the DS on that processor depends on the real time requirement $F_{io}$ - in samples per second.

1. If $F/C > F_{io}$, the DS can be implemented on the sequential processor at hand, through straightforward software.
2. If $F/C < F_{io}$, one needs a more parallel implementation of the digital system.

In case 1, the full computing power - $WF$ in *bop/s* - of the processor is only used when $F/C = F_{io}$. When that is not the case, say $F/C > 2F_{io}$, one can attempt to trade time for area, by reducing the data width to $W/2$, while increasing the code length to $2C$: each operation on $W$ bits is replaced by two operations on $W/2$ bits, performed in sequence. The invariant is the product $CW$, which gives the complexity of $f$ in *bop* per sample. One can thus find the smallest processor on which some sequential code for $f$ can be executed within the real time specification. The end of that road is reached for $W = 1$: a *single bit wide sequential processor*, whose run-code has length proportionnal to $G(f)$.

In case 2, and when one is not far away from meeting the real time requirement - say $F/C < 8F_{io}$ - it is advised to check if code $C$ could be further reduced, or moved to a wider and faster processor (either existing or soon to come when the feature size shrinks again). Failing that software solution, one has to find a hardware one. A common case mandating a hardware implementation, is when $F \approx F_{io}$: the real time external IO frequency $F_{io}$ is near the internal clock frequency $F$ of the chip.

### 4.6 Dynamic Reconfiguration

We have seen how to fold time in space: from a small design into a larger one, with more performance. The inverse operation, which folds space in time, is not always possible: how to fold any bit serial circuit (such as the adder from fig 5) into a half-size and half-rate structure is not obvious. Known solutions involve dynamic reconfiguration.

Suppose that function $f$ may be computed on some RS of size $2A$, at twice the real-time frequency $F = 2F_{io}$. We need to compute $f$ on a RS of size $A$ at frequency $F_{io}$ per sample. One technique, which is commonly used in [13], works when $Y = f(X) = g(h(X))$, and both $g$ and $h$ fit within size $A$.

1. Change the RS configuration to design $h$.
2. Process $N$ input samples $X$; store each output sample $Z = h(X)$ in an external buffer.
3. Change the RS configuration to design $g$.
4. Process the $N$ samples $Z$ from the buffer, and produce the final output $Y = g(Z)$.
5. Go to 1, and process the next batch of $N$ samples.

Reconfiguration takes time $R/F$, and the time to process $N$ samples is $2(N + R)/F = (N + R)/F_{io}$. The frequency per sample $F_{io}/(1 + R/N)$ gets close to real-time $F_{io}$, as $N$ gets large. Buffer size and latency are also proportional to N, and this form of dynamic reconfiguration may only happen at a low frequency.

The opposite situation is found in the ALU of a sequential processor: the operation may change on every cycle. The same holds in *dynamically programmable* systems, such as arrays of processors and DPGA [14]. With such a system, one can reduce by half the number of processors for computing $f$, by having each execute twice more code. Note that this is a more efficient way to fold space in time than previously: no external memory is required, and the latency is not significantly affected.

The ALU in CHESS is also dynamically programmable. Although no specialized memory is provided for storing instructions (unlike DPGA), it is possible to build specialized *dynamically programmed* sequential processors, within the otherwise *statically configured* CHESS array. Through this feature, one can modulate the amount of parallelism in the implementation of a function $f$, in the range between serial hardware and sequential software, which is not accessible without dynamic reconfiguration.

## 5 Conclusion

We expect it to be possible to build *Reconfigurable Systems* of arbitrary size, which are fit to shrink: they can exploit all the available silicon, with a high normalized density for storage, arithmetic and Boolean operations, and operate at high normalized clock frequency.

For how long will the market demands for operations keep-up with the supply which such RS promise?

Can the productivity in mapping applications to massively parallel custom hardware be raised at the pace set by technology?

Let us take the conclusion from Carver Mead [9]:

> There is far more potential in a square centimeter of silicon than we have developed the paradigms to use.

## References

1. C. E. Shannon, W. Weaver, *The Mathematical Theory of Communication*, University of Illinois Press, Urbana, 1949.
2. C. Mead, L. Conway *Introduction to VLSI systems*, Addison Wesley, 1980.
3. F.P. Preparata and J. Vuillemin. *Area-time optimal VLSI networks for computing integer multiplication and Discrete Fourier Transform,* Proceedings of I.C.A.L.P (Springer-Verlag), Haifa, Israel, Jul. 1981.
4. J. Vuillemin, *A combinatorial limit to the computing power of VLSI circuits*, IEEE Transactions on Computers, C-32:3:294-300, 1983.
5. I. Wegener *The Complexity of Boolean Functions*, John Wiley & sons, 1987.
6. O. A. McBryan, P. O. Frederickson, J. Linden, A. Schüller, K. Solchenbach, K. Stüben, C-A. Thole and U. Trottenberg, "Multigrid methods on parallel computers—a survey of recent developments", *Impact of Computing in Science and Engineering*, vol. 3(1), pp. 1–75, Academic Press, 1991.
7. J. Vuillemin. *Contribution à la résolution numérique des équations de Laplace et de la chaleur*, Mathematical Modelling and Numerical Analysis, AFCET, Gauthier-Villars, RAIRO, 27:5:591–611, 1993.
8. M. Shand and J. Vuillemin. *Fast implementation of RSA cryptography*, 11-th IEEE Symposium on Computer Arithmetic, Windsor, Ontario, Canada, 1993.
9. C. Mead, *Scaling of MOS Technology to Submicrometre Feature Sizes*, Journal of VLSI Signal Processing, V 8, N 1, pp. 9-26, 1994.
10. J.Vuillemin. *Fast linear Hough transform,* The International Conference on Application-Specific Array Processors, IEEE press, 1–9, 1994.
11. J. Vuillemin. *On circuits and numbers*, IEEE Trans. on Computers, 43:8:868–79, 1994.
12. L. Moll, J. Vuillemin, P. Boucard and L. Lundheim, *Real-time High-Energy Physics Applications on DECPeRLe-1 Programmable Active Memory,* Journal of VLSI Signal Processing, Vol 12, pp. 21-33, 1996.
13. J. Vuillemin, P. Bertin , D. Roncin, M. Shand, H. Touati, P. Boucard *Programmable Active Memories: the Coming of Age*, IEEE Trans. on VLSI, Vol. 4, NO. 1, pp. 56-69, 1996.
14. A. DeHon *Reconfigurable Architectures for General-Purpose Computing*, MIT, Artificial Intelligence Laboratory, AI Technical Report No. 1586, 1996.
15. N. Sakashita & al., *A 1.6-GB/s Data-Rate 1-Gb Synchronous DRAM with Hierarchical Square-Shaped Memory Block and Distributed Bank Architecture*, IEEE Journal of Solid-state Circuits, vol. 31, No 11, pp 1645-54, 1996.

16. M. Shand, *Pamette*, a Reconfigurable System for the PCI Bus, 1998. http://www.research.digital.com/SRC/pamette/
17. Xilinx, Inc., *The Programmable Gate Array Data Book*, Xilinx, 2100 Logic Drive, San Jose, CA 95124 USA, 1998.
18. G. Moore. *An Update on Moore's Law*, 1998. http://www.intel.com/pressroom/archive/speeches/gem93097.htm
19. Alan Marshall, Tony Stansfield, Jean Vuillemin *CHESS: a Dynamically Programmable Arithmetic Array for Multimedia Processing*, Hewlett Packard Laboratories, Bristol, 1998.
20. M. Shand. *An Update on RSA software performance*, private communication, 1998.
21. *The millenium bug: how much did it really cost?*, your newspaper, 2001.