Journal of VLSI Signal Processing 2007

DOI: 10.1007/s11265-007-0127-9

Application-specific Processor Architecture: Then and Now

PETER CAPPELLO
Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93106, USA

Received: 30 January 2007; Revised: 21 June 2007; Accepted: 22 June 2007

Abstract. We first relate the architecture of systolic arrays to the technological and economic design forces acting on architects of special-purpose systems some 20 years ago. We then observe that those same design forces now are bearing down on the architects of contemporary general-purpose processors, who consequently are producing general-purpose processors whose architectural features are increasingly similar to those of systolic arrays. We then describe some economic and technological forces that are changing the landscape of architectural research. At base, they are the increasing complexity of technology and applications, the fragmenting of the general-purpose processor market, and the judicious use hardware configurability. We describe a 2D architectural taxonomy, identifying what, we believe, to be a "sweet spot" for architectural research.

Keywords: application-specific processor, computer architecture, field-programmable gate array, FPGA, general-purpose processor, processor array, systolic array, taxonomy

1. Introduction

This article is an outgrowth of a talk given at the 2006 ASAP conference titled "Multicore Processors as Array Processors: Research Opportunities" [5]. One thesis of the talk was that special-purpose processor (SPP) architectures must push the limits of technology before general-purpose processor (GPP) architectures do, and that we now are seeing GPPs embrace architectural features that systolic arrays embraced some 20 years ago. This article also develops this thesis. First, we observe the design forces whose net effect shaped systolic arrays. Then, we briefly provide evidence that these design forces now are shaping GPPs similarly. Finally, we consider some newer design forces that are acting on both SPPs and GPPs, and identify a sweet spot in a 2D taxonomy that ties them together. These new forces are both technological and economic. The technological force is reconfigurable hardware, as evidenced by FPGA systems. The economic force is a refinement of the GPP market: The GPP has given way to increasingly refined processors, such as server processors, hand-held processors, graphic processors, and sensors, to name a few. Generally, the design space of processors is becoming more complex. Increasing complexity leads to increased specialization of labor that affects design processes and development organizations.

2. 1986 VLSI Design Forces and Systolic Arrays

> Nature, to be commanded, must be obeyed. (*Sir Francis Bacon*)

The year 1986 in the title of this section reflects the 20-year anniversary of the workshop on systolic arrays, and is not intended as a birthday for systolic arrays. The seminal paper on systolic arrays is by C. Leiserson and H.T. Kung [16], and appeared in 1978. They developed this work into Section 3 of chapter 8 of Mead and Conway's book, *Introduction to VLSI Systems* [21], whose copyright is 1980.

This section briefly enumerates the design forces whose accumulated effect results in systolic arrays. These design forces primarily are technological and

economic. Special-purpose architectures are justified only when general-purpose processors (GPPs) cannot deliver the desired performance. For sufficiently large performance improvements, parallelism is required. Perhaps the simplest form of parallelism is pipelining, which is exploited to increase throughput. When pipelining is insufficient, one turns to multiprocessor parallelism, which, while increasing throughput, also reduces latency. In sum, application *performance requirements* that justify a special-purpose architecture often require parallelism:

(1) Sufficiently high performance implies parallelism. Parallelism, in its most general form, enables each processor to communicate with all other processors. This form is illustrated in Fig. 1a. However, physical and technological constraints render this clique-of-processors communication topology infeasible. Specifically, power and area are scarce resources. Power scarcity affects resistive delays:

(2) Power scarcity implies that resistive delay must be limited. This, in turn, has implications for how many long interconnects an architecture can accommodate:

(3) Limited resistive delay implies that long communication lines must be limited. Limiting long communication lines does not imply *eliminating* long communication lines, only that an architecture can accommodate at most a small quantity of long communication lines. In a 3D technology, completely connecting n processors requires interconnection lines that are $\Omega(\sqrt[3]{n})$. However, VLSI technology, was essentially a 2D technology: Completely connecting n processors requires interconnection lines that are $\Omega(\sqrt[3]{n})$. Moreover, the area of a chip dominates the chip's yield, and ultimately its fabrication cost. Area thus was a scarce quantity.

(4) Area scarcity implies that the number of wire crossings must be limited. Reducing area spurred research into interconnection topologies whose layouts were compact. Area considerations make completely connected processor architectures unscalable. However, a linear pipeline of processors satisfy all the performance and technological design constraints that have been enumerated thus far. These constraints are compatible with a pipeline of *heterogeneous* processors, as indicated in Fig. 1b. Special-purpose applications tend to have smaller markets than those for GPPs. Consequently, the design budget of a special-purpose processor tends to be smaller than that for a GPP. This economic aspect has consequences, which are summarized in the following two design forces.

(5) Smaller markets imply a scarcity of design dollars.

(6) Scarce design dollars implies component reuse.

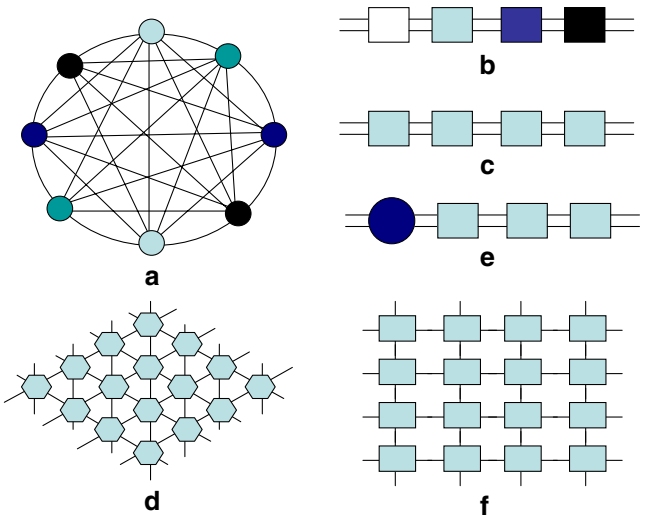

Figure 1. **a** A completely connected heterogeneous multiprocessor (8 processors shown). **b** A heterogeneous processor pipeline (4 processors shown). **c** A homogeneous processor pipeline (4 processors shown). **d** A hexagonal, homogeneous, hex-connected processor array (16 processors shown). **e** A linear processor array with two processor types. **f** A rectangular, homogeneous, 2D mesh processor array (16 processors shown).

A linear array of *homogeneous* processors reduces the design cost to that of a *single* processor with local interconnections. Figure 1c displays such an array, which, for example, could be used to compute a matrix–vector product [21]. The elegance of such a solution is immediately evident. Moreover, the 2D arrays combine pipeline parallelism and latency reduction multiprocessors with no increase in design complexity. Figure 1d displays such an array, which, for example, could be used to compute a [banded] matrix product [21]. Figure 1f displays a rectangular, homogeneous, 2D mesh-connected processor array which, for example, also can be used to compute matrix product. These processor arrays are startling in their elegance. Importantly, throughput can be maintained even as the application problem *size* increases simply by increasing the *size* of the array. In general, systolic arrays do not need to have a single, replicated processor type. To control system design costs, it suffices to *fix the number of processor types*, regardless of the *size* of the array. Figure 1e displays such an array that has two processor types, which, for example, could be used to solve a triangular linear system [21].

We conclude this section with two notes. First, systolic algorithms require *not* that their streams of data proceed in lock-step, but only that the *order* of the data within each stream must be maintained. Thus, globally clocked systolic arrays can be relaxed to wavefront arrays [17], their asynchronous analog. Indeed, relaxing global clocks has at least two intermediate forms which are duals: Islands of synchrony in an ocean of asynchrony, or islands of asynchrony in an ocean of synchrony. Hereafter, we refer to the union of systolic arrays and wavefront arrays as processor arrays. We refer to an element in this architectural class as a *processor array* (PA).

The second note concerns 3D PAs. First, attributes that we believe are intrinsic to PAs are enumerated:

1. Each processor has a set of neighbors, such that:
   - 1.1 The distance between neighbors is independent of the number of processors in the array.
   - 1.2 Only neighbors communicate directly.
2. The number of processor *types* is independent of the number of processor in the array.
3. They scale (i.e., it is sufficient to solve larger problem instances by using larger arrays).

Claim: No 3D PA has attributes 1 and 3. The argument for this claim is as follows. Let A be a 3D PA. It is convenient (but not necessary) to visualize A as an $n \times n \times n$ mesh. Since A scales, we can increase its extent in all three dimensions. The number of processors is $\Omega(n^3)$. Assuming a constant fraction of the processors are used in a constant fraction of the computational steps, power consumption is $\Omega(n^3)$. Let S be the smallest sphere enclosing A. Attribute 1 implies that the volume of S is $O(n^3)$. The surface area of S is $O(n^2)$. The amount of heat dissipated per unit of surface area is $\Omega(n)$. Thus, as n increases, S's surface temperature increases: Heat dissipation does not scale.

Problem sizes however may *not* be bounded in size (e.g., an $n \times n$ matrix). From the foregoing, we see that our PAs can use the third physical dimension only to an extent that is *independent* of the problem size. Bounded use of the third dimension nonetheless may suffice for bounded problems (e.g., achieving the performance of a human eye or ear).

3. 2006 VLSI Design Forces and General-purpose Processors

We now fast-forward to recent developments in GPPs. For some time, GPPs have increased performance by increasing clocking frequencies, which require increasing instruction pipelining, a strategy made problematic by the need to support branch instructions. The increasing power consumption to support these increases in clocking frequencies increases heat dissipation: The physical and economic limits of heat dissipation limited this path to performance increase. Power scarcity now also limits resistive delay, and hence the number and extent of long wires. Architects *again* have no choice but to embrace parallelism and greater locality of communication.

We now present some high-performance processor architectures, which, by example, indicate the extent to which these design forces have shaped contemporary processor architectures.

3.1. Chip Multiprocessors

Continuing to increase clock speed would have caused untenable increases in power consumption

and heat dissipation. Intel and AMD thus abandoned this strategy, adopting multicore architectures: Continue increasing transistors/chip, using the transistors to fabricate multiple processor cores per chip. That is, increase performance via parallelism rather than clock speed. If successful, the performance per watt will be much better than it would have been by sticking to the old strategy. The "if" has to do with the question of whether the software can make good use of the multiple processors. We return to that issue later. For now, suffice it to note that the multicore architecture is *starting* to look like a processor array. Figure 2 indicates the main architectural attributes of interest to us:

- Multiple processors.
- A crossbar switch between the processors and L2 caches.

The processors themselves may be increased in number without becoming a hardware bottleneck (software utilization, though, is another matter). However, the geometric relationship between the processor cores and the L2 cache via the crossbar switch does not scale. This relationship essentially is many-to-many, necessitating long wires, which in turn pose power/latency challenges that, for sufficiently many processors become problematic. One suspects, therefore, that this is an architecture in transition. Nonetheless, we already see significant movement in the direction of a processor array.

3.2. *Vector Intelligent RAM*

This processor architecture is driven to low power without loss of performance by specializing the processor to the kinds of media processing that hand-held devices are likely to require. We do indeed have an array of processors. Again, we have crossbars between the processor and DRAM, which, for sufficiently large processor pipelines, will not scale. One imagines that it will give way to a linear (topologically) pipeline of processor-DRAM pairs, which is more scalable (perhaps snaking around the chip). Even so, bringing DRAM so near the processors is a bold move in what may at least resemble the future of high performance per power processor architectures.

The Vector Intelligent RAM (VIRAM) architecture exemplifies, with respect to the context of this discussion, two points (Fig. 3):

1. There is no single GPP architecture anymore.
2. Communication locality is becoming a primary design goal.

The first point—the GPP market is increasingly fragmented or refined—will be taken up as a bona fide economic design force in the next section.

The second point is that sufficiently high performance requires memory and processing to be geometrically interleaved. As performance per power increases, the value of connecting an array of processors to a separate memory decreases. The degree to which caches mitigate this also decreases as performance per power increases. *Truly integrating memory and processor may be the most daunting challenge of this technological era*, since VLSI technology traditionally segregates processor manufacturing processes from memory manufacturing processes.

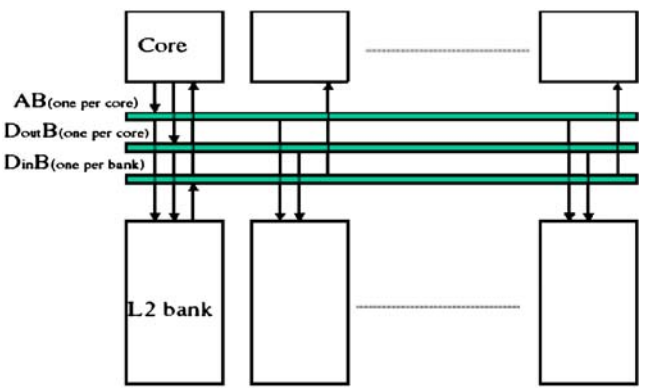

Figure 2. A typical crossbar in a chip multiprocessor architecture [18].

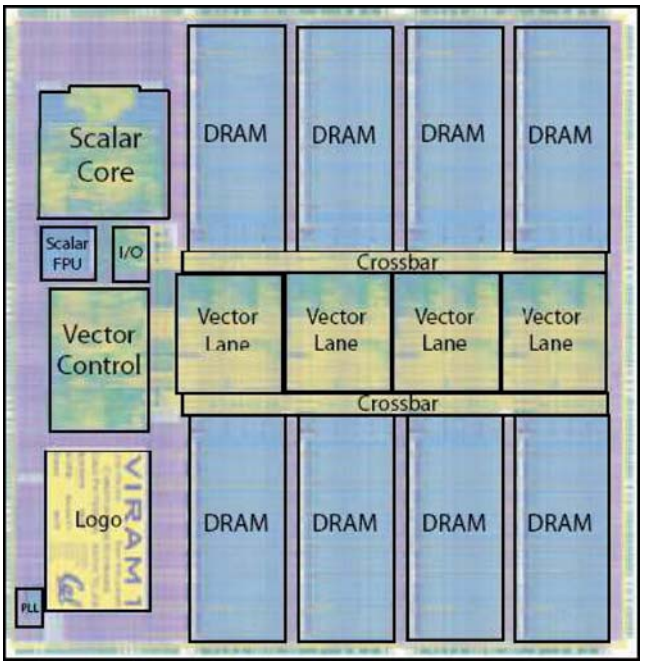

Figure 3. The floor plan of VIRAM1 [12].

3.3. *The Cell Processor*

We see from Fig. 4 that the cell processor is an array of processing elements of two types: The PowerPC Element (PPE) and the Synergistic Processor Element (SPE). Both aspects, the replication of processing elements and the bounded heterogeneity (only two types of processors) reflects well the design forces that acted to shape systolic arrays. Level 1 cache is interleaved with processors. This too reflects the desire for locality of communication, another force that acted to shape systolic arrays. However, the Element Interconnect Bus (EIB) would appear not to scale to a very large numbers of processing elements. The cell processor, in sum, moves the state of the art of processor architecture a great deal in the direction of processor arrays, but not all the way.

3.4. *Tera-ops Reliable, Intelligently adaptable Processing System*

The architects of Tera-ops reliable, intelligently adaptable processing system (TRIPS) (please see Fig. 5) explicitly acknowledge that power now is budgeted. As

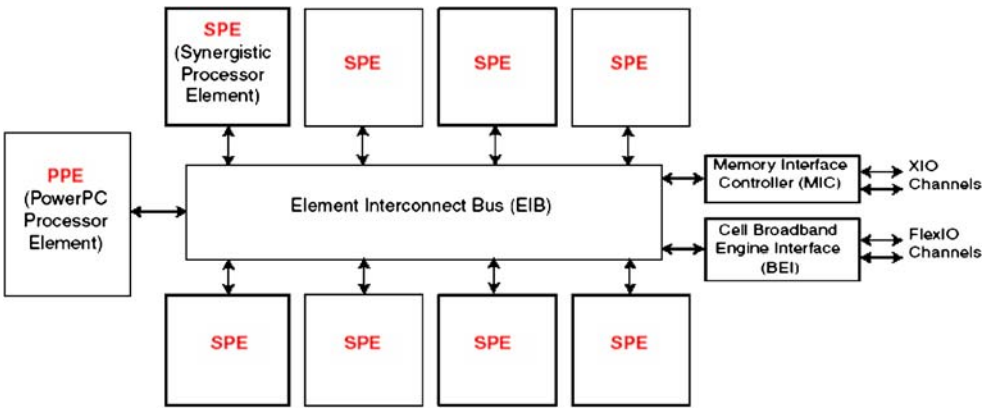

Figure 4. The floor plan of a cell processor [16].

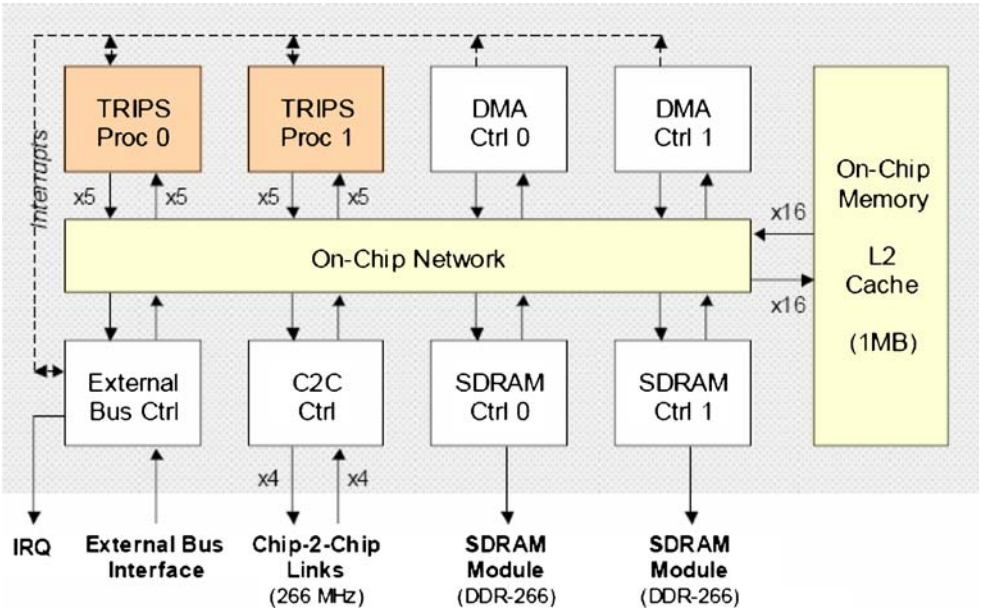

Figure 5. TRIPS chip block diagram [16].

a consequence, architects are hoping that more performance increases can be achieved via software (e.g., compiler analysis, just-in-time compilation/ runtime, and application programs), allowing certain simplifications of hardware requirements, and commensurate power reductions. They also acknowledge that wire delays now, and increasingly will, affect architecture. Of course, they acknowledge that concurrency must be exploited to improve performance per watt.

From the perspective of array processing, these architectures embrace *tiling*, a processing element array architecture: A processor comprises a set of tiles. The number of tile *types* is fixed, but the number of tiles is intended to scale. Figure 6 illustrates this architectural feature. This beautifully reflects both the parallelism of systolic arrays and component reuse. Also like systolic arrays, the same advantages accrue: increased performance from increased concurrency and increased designer productivity from component reuse.

To improve concurrency with, it is hoped, minimal application program modification, TRIPS uses an Embedded Data Graph Execution (EDGE) architecture. Under this scheme, at compile time, the program graph is partitioned into blocks, each of which exposes instruction-level parallelism (ILP) that is exploited by TRIPS processors. This is illustrated in Fig. 7. An important aspect of this architecture is the reduced use of registers: They are used at the boundary of blocks, but not within blocks, saving energy and increasing performance. This, in effect, used the 2D geometry of VLSI technology to create what might be termed a 2D very large instruction word (VLIW). Interestingly, this concept also has been explored by Santos et al. [23]. One difference between their respective approaches appears to be that, in TRIPS, each block asynchronously signals its completion to a global controller; in the 2D-VLIW, the block is scheduled (both placement and dispatch) statically and synchronously. One of the most difficult issues facing hardware designers concerns how much complexity can be shifted to application programmers. The EDGE/2D-VLIW architectures that exploit larger amounts of ILP may be a vital contribution to this most serious challenge. Also, since there are many blocks and many processors, some coarse-grained concurrency, in principle, also may be achieved.

The processors clearly are separated physically from the L2 cache and the off-chip memory, which limits scaling the number of processors per chip. Although not shown, the SDRAM is distributed among the TRIPS chips. So, there is an interleaving of memory and processors, albeit across chips.

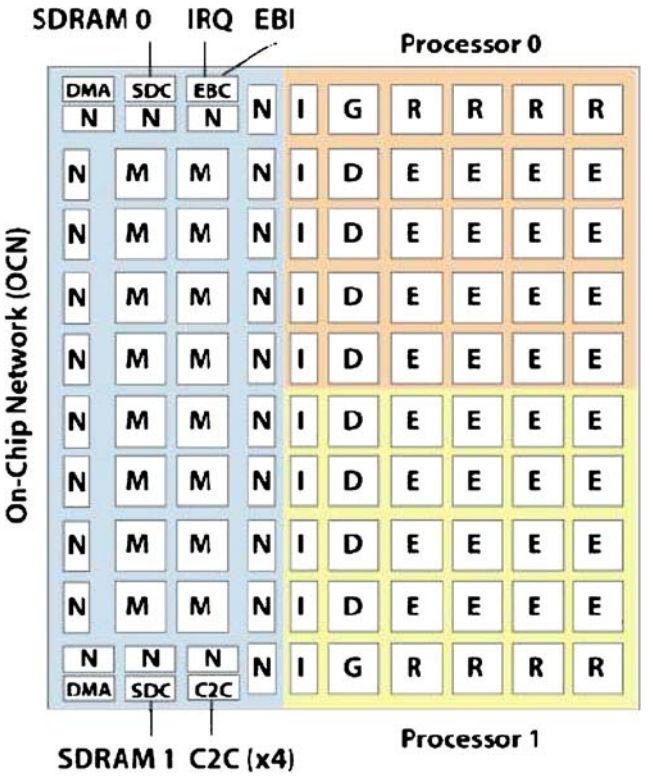

Figure 6. TRIPS tile-level microarchitecture: Processor[s] as an array of tiles: *D* data cache, *E* execution tile, *I* instruction cache, *G* global control, *M* memory (scratchpad or L2 cache), *N* network interface, *R* register bank [23].

3.5. *Reconfigurable Architecture Workstation*

The reconfigurable architecture workstation (RAW) architecture [1, 24, 25] explicitly treats the scalability problems of traditional wide issue GPP architectures. Figure 8 gives a schematic floor plan for such an architecture; Fig. 9 gives a schematic floor plan for the RAW architecture. The designers replace the crossbar

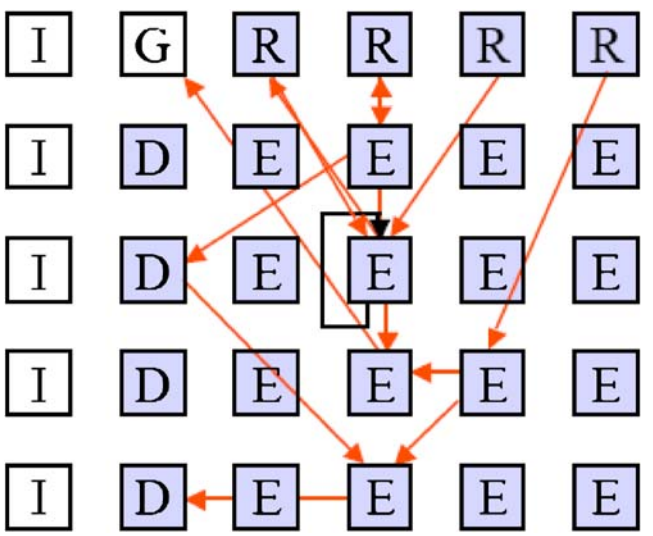

Figure 7. Block execution: Processor core tiles and interfaces: *R* tiles inject register values, *E* tiles execute block: load, compute, deliver outputs to R-tiles/D-tiles, and branch to G tile [23].

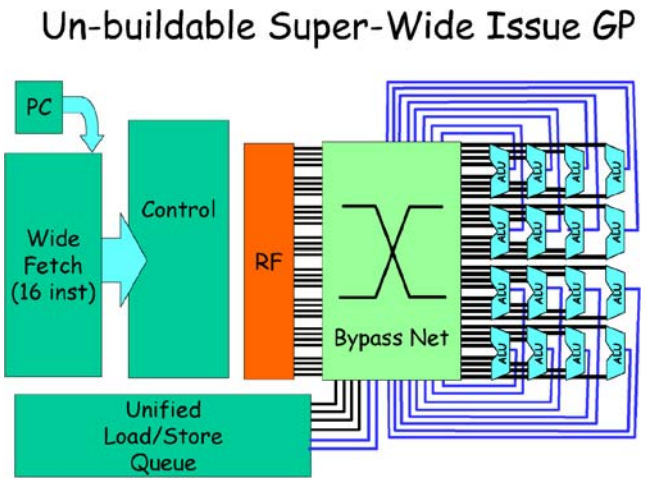

Figure 8. Scaling a wide issue GPP to an *unbuildable* extreme (from [26]).

bypass network with a point-to-point, pipelined, 2D mesh routing network. One important aspect, from the perspective of this article, is their respect for the need to *localize* communication. This is reflected in their explicit distribution of the register file, instruction cache, PC, and data cache. The resulting architecture is no less than a frontal assault on the scalability problems of the traditional architecture. Locality also is reflected in the routing network, which is pipelined-data's destination per cycle is never too far from its source.

A path breaking aspect of this approach, among many, is the replacement of hardwired connections with a *programmable* network, pushing the optimization of application communication from the hardware to the compiler. By pipelining, they reduce the maximum wire length and commensurately increase clock speeds without the customary attendant increases in power consumption. At the same time, they allow the network, via *software*, to be specialized according to the application. This significantly increases the complexity of efficient compilation. But, in principle, it reduces the design cost and the operation cost (power and/or time) of multi-phased applications. If this trade can indeed be made, it surely is a good use of the technology (transistors are cheap; people, power, and time are not).

The designers fully embrace the design forces that shaped systolic arrays for essentially the same reasons.

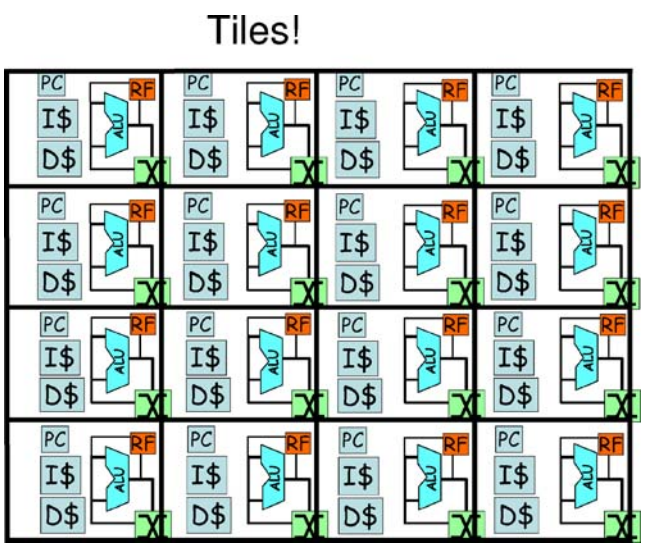

Figure 9. The distributed, scalable version of the GPP in Fig. 8 (from [26]).

Maximizing communication locality reduces long wires: Reducing the long wires yields the same performance per watt advantage sought by systolic array designers. Tiling reduces design costs: They leverage the design cost of a single tile, which is increasingly complex. The resulting architecture very much resembles a 2D systolic array (which could be emulated efficiently by the RAW).

The programmable routing network however, is an attempt to be more general than any particular systolic array. The idea is to make the RAW a programmable universal systolic array. Insofar as the routing network is a pipelined 2D mesh, this goal seems to be achieved. Indeed, high throughput stream processing is well suited to the RAW. Some low latency applications also can be supported by RAW. However, it is not clear that the most demanding low latency applications (e.g., high performance control) can be served by the pipelined 2D mesh network.

3.6. *Merrimac*

The Merrimac [9] architecture uses two 2 x 4 *cluster* arrays, separated by a microcontroller. Clearly, this is a kind of tiling: Component reuse reduces design cost and simplifies testing. Each cluster comprises four floating-point multiply-and-add units, 768 64-bit words of local registers, and 8 K words of stream register file: This exemplifies their perceived goal of *increasing locality to increase performance*.

Streaming itself is the primary vehicle in systolic architectures for achieving locality and increasing parallelism without increasing memory bandwidth. The *streaming* aspect of the Merrimac is done for the same reason with the same results.

The Clos network [7] aka a fat-tree [19] is a deviation from systolic architectures, but one that makes the Merrimac multiprocessor architecture more general purpose (Fig. 10). The Clos network has a shorter bisection width (diameter) than a torus. Clearly, some interconnect *lengths* grow as the number of processors grows. But, it is a slow-growing function that has some hope of accommodating rather large numbers of processors before a performance wall is hit. Again, limited resistive delay implies that long communication lines must be limited, *not eliminated*. A Clos network is a judicious use of long wires. It enables Merrimac to host computations that have low-latency requirements, making it an ambitiously general-purpose high-performance processor architecture.

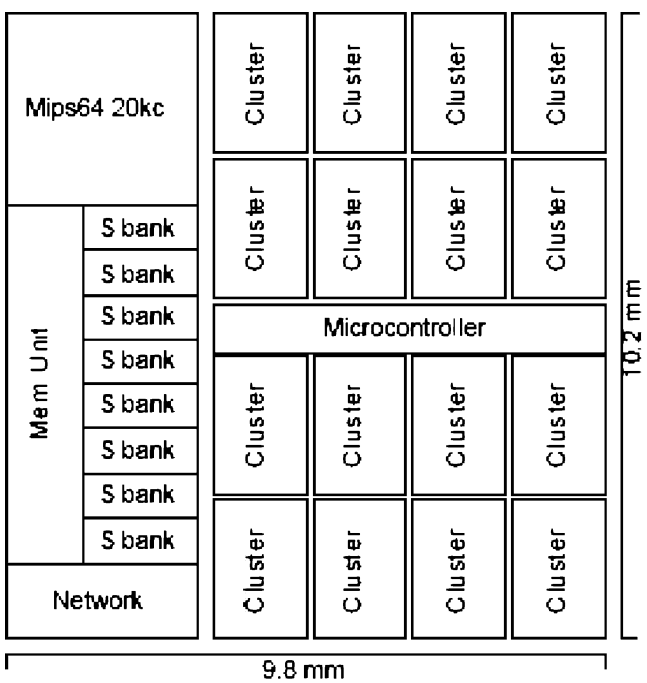

Figure 10. Floorplan of a Merrimac stream processor chip (from [9]).

4. Where are Architectures Headed?

In fairness to the architects of GPPs, their seemingly belated embrace of some fundamental technological design forces has to do with the set of applications they support. They needed to be concerned with supporting essentially any application; systolic arrays are designed to support highly regular algorithms, such as dense or banded matrix algorithms. The problem faced by GPP architects seemingly is to make processor array architectures efficient for general applications, which fundamentally is an algorithmic and programming challenge.

The existing design forces, both technologic and economic will continue into the future. A trend that already has started will likely increase in the future: the fragmentation of the processor market. For example, we used to have a GPP market. It increasingly is giving way to a market for desktop architectures, server architectures, hand-held architectures, graphics architecture, etc. Perhaps scientific architectures most resemble what once were GPP architectures. This fragmentation may have resulted from the size of the GPP market growing to the point that subdivisions were economically feasible. Since each niche architecture performs its niche of applications better than a GPP architecture, its operational cost advantages were justified even in the face of the smaller pot of design dollars destined for that architectural niche. These niches are *subclasses* of the GPP architectural class: Each architectural subclass benefits from the design dollars put into the general class, adding *some* design cost to create their architectural subclass, but *sharing* some design cost with all subclasses. Nonetheless, the smaller pot of design dollars for each niche puts it in a position that is reminiscent of application-specific architectures: smaller markets means fewer design dollars (to design their architectural subclass). In short, the implications of scarce design dollars is the same: increased impetus to reuse components. To these forces we add two more.

4.1. Field-programmable Gate Array

One technologic force is field-programmable gate arrays (FPGAs), which have advanced rapidly for some time. One can imagine two design points: (1) custom hardware that performs one application optimally and (2) general-purpose hardware whose software performs many applications (sub-optimally). The spectrum connecting these two extremes establishes the degree to which the hardware is programmable, which is fixed at the time the hardware is designed. FPGAs add another *dimension*: The degree to which the hardware can be reconfigured in the field. In this 2D *reconfigurability–programmability* design space:

(7) The degree of programmability increasingly can be field-configurable. For example, an FPGA can implement an ISA which can expand or contract in the field, depending on which instructions an application actually needs. There may be a *field-programmable* architectural tradeoff, for example, between the number of ALUs and the size of their instruction set. Similarly, the FPGA can implement a routing network that not only can change in the field, but whose *degree of programmability* can change in the field. Again, some applications need a wider set of communication topologies than others. Economizing hardware where one can allows us to expand hardware where the application can most benefit from expansion (e.g., decreasing the amount of hardware devoted to routing so that we can increase the number of processor pipelines in a non-recursive signal processing application where increasing throughput is more important than reducing latency). Co-location of diverse high-performance applications implies the need for reconfigurable hardware, so that each application can make good (albeit not fully optimal) use of the hardware substrate. The fraction of hand-held appliance hardware that is reconfigurable, for example, may increase. Sensors, on the other hand, have such tight power constraints, that we would not expect any fraction of their hardware to be reconfigurable for the foreseeable future.

4.2. Increasing Complexity and Adaptability

Focus now on the reconfigurability spectrum, which connects the two extremes of purely custom hardware with purely FPGA (of course, any FPGA is programmable to a limited extent). The space in between these extremes is populated by hybrid architectures: custom hardware dimpled with FPGA cores and FPGAs dimpled with ASIC cores. This

spectrum, which itself is changing rapidly due to rapid advance in FPGA technology, is emblematic of a new design force: Both applications and technology are increasing in complexity.

(8) *Increasing complexity implies increasing specialization of labor.* The *rate* of increase in complexity also appears to be increasing. In such turbulent times, the importance of *adaptability* increases. People, organizations, architectures, and designs all benefit from adaptability. This feature-adaptability, we predict, will be a *primary* force determining the success of people, organizations, architectures, and design processes.

(9) *Increasing rate of complexity increases implies increasing value of adaptable designs and design processes.*

4.3. An Architectural Sweet Spot

Based on the foregoing observations, we predict an architectural sweet spot: an architectural region that will receive an increasing amount of attention. Figure 11 gives a 2D taxonomy of processor architectures. The abscissa is the architecture's degree of field reconfigurability. At one extreme, we have an architecture that has no field reconfigurability. The other extreme is an architecture built entirely from an FPGA. The ordinate in this taxonomy is the architecture application specificity. At one extreme are architectures that are designed for a specific application. The other extreme represents general-purpose architectures. As indicated in Fig. 11, the taxonomy can be divided into quadrants. Application-specific integrated circuits (ASIC) belong in the lower left quadrant, since they are custom, non-configurable circuits. GPPs belong in the upper left: They are custom, non-configurable circuits that attempt, as far as possible, to be able to handle the maximum set of applications. A Custom Computing Machine (CCM) is very reconfigurable, hence its position in the right part of the taxonomy. If the CCM is reconfigured only at design time, to some particular custom computing machine, it might be best placed in the lower right quadrant. However, if the CCM can reconfigure operationally (at operation time), it, in effect, is a GPP, belonging in the upper right quadrant. The lower right quadrant often contains specific applications that are being prototyped before proceeding to ASIC design. A reconfigurable FIR filter "tissue" [6] also would belong in this quadrant.

We predict that attention given to architectures in the center of this 2D taxonomy will increase. We may conceptually define the center with four constraints: two lower bounds, which are primarily economic, and two upper bounds, which are primarily performance-oriented:

1. The specificity lower bound: If architectures are getting increasingly complex, their design is likely to get increasingly expensive. The more specific an architecture is the fewer applications it can serve, the fewer dollars can be devoted to its design. We thus expect that most architectures will target a set of applications that make their design economically viable.
2. The specificity upper bound: Demands for performance and the size of special markets, such as routers, have fragmented the GPP market. It is increasingly likely that no single GPP architecture

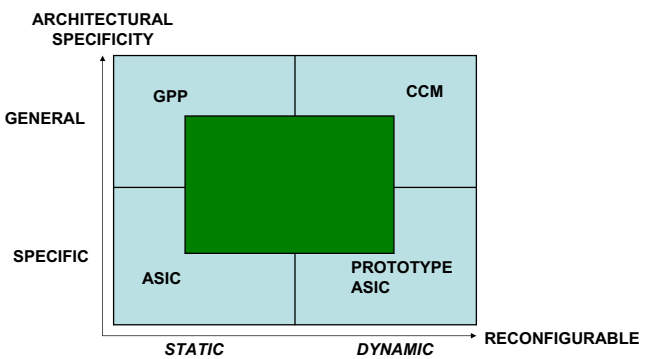

Figure 11. A 2D architectural taxonomy and its sweet spot.

will have sufficient performance to be used in any particular market fragment (e.g., the processor used in a router will be different from a processor used in a desktop computer). There essentially will be no GPP that is commercially viable. Truly GPPs will be actively researched and developed only in academic settings, if there.

3. The reconfigurability lower bound: We are not equating a circuit's complexity with its number of transistors. Rather, complexity concerns how difficult the architecture is to describe (i.e., its Kolmogorov complexity [27]). If one can describe the architecture by a small iterative or recursive program, it is not complex. In this descriptive sense, memory chips are not complex. (And indeed, we may expect that memory chips will continue to be ASICs.) For the applications of the future whose descriptive complexity is increasing, the cost of purely custom circuitry will be prohibitive.
4. The reconfigurability upper bound: We believe that most applications will have performance per watt requirements that preclude reliance on a purely FPGA architectural substrate.

The sweet region then is one that embraces architectures that are partially reconfigurable and partially custom, that serve a set of applications that is large enough to financially justify careful design, and which perform well for their applications, precluding application universality.

Several examples may serve to flesh out this taxonomy. The Cell processor, TRIPS, and the Merrimac processor all can be used as a general-purpose multiprocessor platform. Their main aspect is their multiprocessor architecture. Indeed, none advertise reconfigurability. So, it seems appropriate to place them in the upper left quadrant. The RAW's network programmability is not currently billed as field-configurable. Each tile however has a small amount of reconfigurable circuitry. It therefore can be placed to the right of the previously mentioned architectures.

To use the Tensilica Xtensa processor [26], one defines an application-specific processor, and the Tensilica Xtensa is configured appropriately, including software support. Since the Xtensa has asic cores, the resulting processor is efficient. Its *partial* reconfigurability reduces design time. However, once configured, Xtensa-based processors are not advertised to dynamically reconfigure (i.e., reconfigure during operation, for example, during the operation of a cell phone). It can be placed in the lower right quadrant.

LISATek [8] similarly allows one to quickly design an efficient application-specific architecture with associated software. It too is not advertised to operationally reconfigure. Within this taxonomy, it can be placed in the same neighborhood as the Xtensa processor.

MathStar's Field-Programmable Object Array (FPOA) is similar to an FPGA. However, the objects, which are 16 bit configurable machines, such as an Arithmetic Logic Units (ALU), Multiply-Accumulators (MAC), or Register Files (RF), are connected by a high-speed interconnect mesh. Its placement is similar to LISATek. Objects however have the ability to change communication patterns on a per-clock basis. That is, beyond design configurability, MathStar's FPOAs can reconfigure operationally. If we augmented the taxonomy with a third dimension representing *reconfiguration time*, the FPOA would be positioned differently from the LISATek in that dimension.

Finally, we refine the taxonomy with a third dimension concerned with the latency of the communication network that connects processing elements. This refined taxonomy is presented in Fig. 12. This essentially is a way of classifying applications, based on the performance requirement to produce a solution quickly (low latency network) versus high throughput. The former might arise, for example, in a high-performance recursive control application (e.g., robotic control of a fast-moving vehicle). The latter might arise, for example, in a router or a video server. (In a router or video server, not only is throughput more important that low latency, there is only a small requirement for persistent storage. Under these circumstances, off-chip memory appears to be an obstacle to locality that is tempting to directly address, as was done in the VIRAM.) High throughput applications often exhibit data parallelism; low latency applications *may* exhibit thread parallelism.

This brings us to an intriguing question: What part of the architectural sweet spot should be reconfigurable and what part should be custom circuitry? The space of architectures under consideration can be viewed as those comprising as set of partially reconfigurable processors connected by a partially

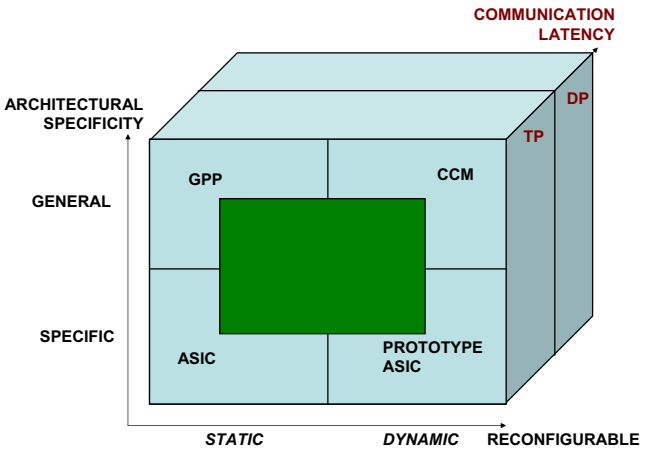

Figure 12. A 3D architectural taxonomy and its sweet spot, where the third dimension represents the latency of the communication network.

reconfigurable communication network. While this sweet spot is an architecturally rich design space, we quickly identify two simple, extreme categories of interest:

- An ASIC network connecting FPGA core processors.
- An FPGA network connecting ASIC core processors.

The Xilinx Virtex-II Pro X [28] might be considered an early entry in the latter category, sporting up to two PowerPC processor cores. Even these two extreme categories, which simplify the question of where to put the reconfigurability, in fact *oversimplify* important architectural opportunities and issues. For example, in the custom network of reconfigurable processor cores, what aspects should be reconfigurable? When should it be reconfigured? We could reconfigure the instruction set [architecture] between hyperblocks. However, the time to reconfigure may be too great to justify the time reduction that the customized instruction set [architecture] achieves on the hyperblock: This amount of reconfiguration may be too much or too frequent. How frequently/extensively can it profitably be reconfigured? Indeed, there is a tradeoff between frequency and extensiveness of reconfigurability: The more extensive the reconfiguration, the less frequent it must be to achieve a net time reduction. The tipping point, based on FPGA technology is improving with time: More extensive/frequent reconfigurations are possible with each new generation of FPGAs. One approach is to reconfigure only between applications. Another, is to have a two-level block scheme: An application is decomposed into a directed graph (digraph) of hyperblocks, each of which is executed as in an EDGE architecture. However, this digraph could be partitioned into subgraphs. Reconfiguration occurs between subgraphs, but not within a subgraph. The size of a subgraph would be a compiler parameter, which can be reduced as the time for an FPGA to reconfigure decreases.

The principal constraint, for time-sharing, is to avoid time-slicing applications on the same processor cores, if doing so requires reconfiguration. One could *partition* the cores among a set of concurrently running applications.

Similar considerations apply to when and how to [re]configure/[re]program a configurable/programmable communication network.

4.4. *Specialization of Labor*

We now return to the specialization of labor that results from increasing complexity of applications and technology. We need application domain experts, programming language designers, compiler writers, computer architects, computer engineers, FPGA experts, and VLSI circuit specialists. Using traditional terms, we need applications experts, computer scientists, computer architects/engineers, and electrical/materials engineers. It is increasingly difficult for any one person to be expert in the relevant disciplines, starting from application do-

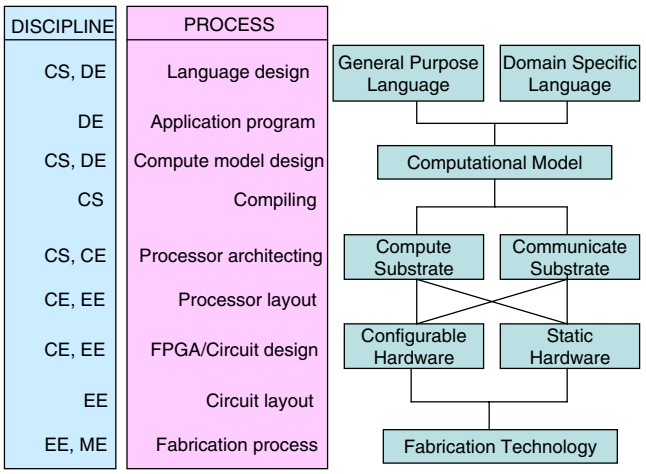

Figure 13. Specialization of labor. Disciplines appear on the left. *CE* Computer engineer, *CS* computer science, *DE* domain expert, *EE* electrical engineer, *ME* materials engineer. Processes appear in the *center*. The *right-hand part* of the figure depicts different layers in the design process.

main and going all the way to materials. While individuals with interdisciplinary knowledge are desirable, interdisciplinary collaboration is essential.

Figure 13 vertically depicts levels of representation (e.g., from programming language to VLSI circuits), activities (e.g., from application programming to implementing an FPGA in VLSI circuit technology), and actors (e.g., application experts to materials engineers). Let us briefly consider these representations, actions, and actors.

Programming language One reason that GPP multicore architectures did not emerge sooner is because the "problem" of parallel programming had not been "solved." It is a complex, time-consuming, error-prone business, to be avoided, if at all possible. Parallel programming now is central to the success of this new hardware era. Hardware manufacturers thus are gravely concerned. Since parallel programming remains an unsolved problem, the research into, and development of, effective, efficient parallel programming languages, tools, and systems is most urgent. There however is controversy about how programming languages and systems should change to facilitate the development of software for multiprocessor chips. The language should be *high level*, requiring the application programmer to say only *what* needs to be done, letting the compiler, as far as possible, decide *how* to do it. But, the language and computational model must not hide from the application programmer those "how" aspects that impact performance and which require *application knowledge* to exploit. Certain opportunities for concurrency are best identified at the application level. It may well be that parallel programming languages and systems will differentiate between broad application domains (i.e., computational models), such as between a single problem instance model and a stream processing model (e.g., signal processing). We are reminded of programming languages that are specific to "systolic computations" [10, 11].

The design of high-level general-purpose programming languages typically requires computer science expertise; domain-specific languages, of course, also require domain-specific expertise.

Like hardware, the high 800 cost of software development implies the desirability of reuse, which, in turn, implies the identification and abstraction of a computation's independent components.

Computational Model What should the computational model expose to the application programmer? What "how" details should it require of the application programmer? Different applications seem to yield different answers. Should the computational model be a sequential machine, where the compiler is entirely responsible for mapping the actual computation onto a multiprocessor? Most people dismiss this approach, arguing that the compiler cannot do a sufficiently good job.

A parallel random access machine (PRAM) is an abstract machine for designing the algorithms applicable to parallel computers. It is a multiple instruction

multiple data machine model that hides synchronization and communication issues from the application programmer, letting the programmer instead focus on maximizing concurrency. Should a PRAM be the application programmer's computational model? By hiding issues of synchronization and communication, it increases application programmer productivity and cleanly separates application code from synchronization and communication code. For this model to be successful, we need a compiler and multiprocessor architecture that together efficiently execute the application. The principle obstacle to this approach has been an efficient multiprocessor architecture. The PRAM (even the exclusive read, exclusive write variety) is difficult to simulate without a large loss of algorithmic efficiency between the PRAM model and the actual computation time. However, Balkan, Qu, and Vishkin [2] give strong evidence that connecting processors and first-level caches with a mesh-of-trees network can result in acceptable performance. It seems reasonable to believe that a PRAM computational model will yield good application programmer productivity with acceptable performance for applications that require *low-latency*. Examples may include *complex*, real-time robotic control, some large scientific computations, such as the N-body problem, and exact solution to NP-complete problems.

Some applications require high-throughput, but do not require low-latency (e.g., [multiple independent] stream processing). Examples include non-recursive digital signal processing (e.g., audio and video streams), routing, and perhaps database transaction processing, and specific web service processing. In such applications, the computational model may be a directed acyclic graph (DAG) of processes. An important goal in such a model is to efficiently schedule the processes onto a set of processors. This may be done with or without help from the application programmer. If an aspect of the application is both NP-hard to optimize and central to the computation's efficient execution, it may be appropriate to expose it to the application programmer. In very high throughput applications, the topology of the DAG is important to optimize, for example, to minimize the longest edge in a planar embedding. In such circumstances, exposing the DAG to the application programmer allows him to creatively bring application knowledge to bear on the central optimization problem. It may be that a library of topological classes (e.g., 3D meshes, butterfly network [4], banyan networks, fat-trees, cube-connected cycles, shuffle-exchange networks [14], mesh-of-trees) could be developed, whose planar embeddings have been optimized by experts [3, 18]. But, the application programmer may be in the best position to identify the most appropriate topology class in the library of topological classes: This is an example, where we may want the computational model to expose *process topology* to the application programmer.

The compiler, using the computational model, maps the application program to the compute and communicate substrates. Compiler construction is typically the province of computer scientists. But, clearly there is the potential for synergy between the compiler writers and the computer architects.

Should the computational model be, in any way, *reconfigurable*. That is, should it expose any aspect of a reconfigurable compute substrate or communicate substrate to the application programmer?

Compute substrate This is garnering a lot of attention, due to the paradigm shift from increasing clock speeds to increasing concurrency. The EDGE (2D VLIW) architecture will continue to be explored and refined.

The interesting new aspect is hardware reconfigurability: What is the optimal amount? What aspect[s] of the compute[communicate] substrate should be reconfigurable? What is the best way to expose the reconfigurable aspects of the compute[communicate] substrate to the compiler? Should hardware reconfigurability occur only between applications or also within an application? If the former, the compiler may explore a discrete space of compute[communicate] substrates, selecting what it determines is likely to be the best combination of compute and communicate substrates. If hardware reconfigurability occurs within an application, the compiler may need to identify program *phases* such that the time delay of hardware reconfigurability can be amortized over a sufficiently long program phase to be "profitable." In this case, application program *annotations* may help the compiler estimate the length of a program phase (e.g., the *likely* length of an FFT) whose length is data-dependent.

Communicate substrate The desire for increasing on-chip communication concurrency and bandwidth

has led to network-on-chip architectures [22], whose modular architectures promise scalable performance increases. All the questions concerning the compute substrate have analogs for the communicate substrate. Regarding reconfigurability, one imagines that there is a discrete set of compute substrates and a discrete set of communicate substrates. Substrate design desiderata: The compute and communicate substrates are *independent*: Every point in the product space of substrates is a valid computer architecture.

Reconfigurable hardware We expect to see unabated progress in this area. Some specific areas of rapid progress include: FPGA architecture, FPGA application development environments, the speed with which reconfiguration occurs (which affects how and when it is used), and the ASIC cores that populate it. Indeed, FPGAs may become a primary market for ASICs, and affect the richness of FPGA application development environments.

As the complexity of fabrication technology increases, the expertise needed to fully exploit it increases. FPGA designers will have that expertise. Their job is to pass advances in fabrication to computer architects in the form of better FPGAs. This is the multidisciplinary area of computer engineers and electrical engineers.

We have asked above if it would be advisable to expose some degree of hardware reconfigurability to the compiler. A way to do this would be to design *compiler reconfiguration interfaces* (CRI) for the compute and communicate substrates, exposing a carefully designed part of the FPGA development interface to the compiler, whose target then is not entirely static. Advances in compiler technology for reconfiguring hardware then can be delivered to applications by simply recompiling the application. Of course, compiler reconfiguration interface *design* is key, a ripe area of interdisciplinary research involving compilers, computer architecture, and FPGA design.

We similarly could design a computational model reconfiguration interface, exposing a carefully designed part of the compute and communicate substrate reconfiguration interfaces to the application programmer. Application Program Reconfiguration Interfaces (APRI) is a ripe area of interdisciplinary research involving programming languages, computational models, and computer architecture.

Static hardware In the heterogeneous space of partially reconfigurable, partially static circuits, there is the question as to what part should be static. In some cases, it will be all or part of compute substrates, in others it will be all or part of communicate substrates. As the demand for a circuit increases the feasibility of a custom design increases. As tools for the development of custom circuits increase designer productivity, the feasibility of custom circuits increase. As the cost of silicon (and the need for silicon reuse) decreases, the feasibility of custom circuits increase. While these are somewhat simplistic characterizations, we can be assured that the dance between reconfigurable hardware and custom circuitry will go on.

The design of hardware design tools, while multidisciplinary, will always include electrical engineers, whose understanding of the fabrication technology will always be key.

Fabrication technology Work in electrical/material engineering continues apace. Apart from the "disruptive" technologies, such as quantum dots, nanotubes, and RNA-substrates, advances in silicon-based circuit substrates continues to impress. Indeed, embedding lasers in silicon may produce wonderful new toys that change the rules for communicating within and between chips. There is every reason to believe that the next 20 years in electrical/material engineering will greatly surpass the advances of the last 20 years.

The multi-layer design process, depicted in Fig. 13 is intended to emphasize the specialization of labor that results from the increasing complexities throughout the entire process of specifying a computation to executing that computation in hardware. Each layer acts as an articulation point for the layer above and the layer below. For example, the compute and communicate substrates act as an articulation point between compilers, on the one hand, and FPGAs and custom circuits on the other. Changes in an FPGA layer can be hidden from the compiler by the substrates, or it can be exposed to the compiler by the substrates. In general, each layer can control the propagation of changes to neighboring layers. This control enables layers to advance concurrently without being chaotic. This control pattern might be referred to as an *iterated adaptor pattern*; it allows *controlled adaptation* in rapidly changing multi-layer environments.

The expertise dependencies are not a one way street. For example, circuit design involves placement and routing tools and [programming] languages. The underlying design problems translate into large-scale computationally complex problems that are the province of computer science and/or operations research. Beware the person who derides electrical engineers, computer scientists, or any other discipline. We all need each other. The winning culture is inclusive, not tribal.

5. Conclusion

Special-purpose processors must push the limits of technology to achieve the performance that justifies their cost. Systolic arrays embrace VLSI circuit technology to achieve high performance via parallelism, honoring the scarcity of power and resistive delay by using a communication topology with no long inter-processor wires. The communication topologies require chip area that is linear in the number of processors. They also embrace the design *economics* of special-purpose processors by reusing a processor design in an array of (generally) homogeneous processors.

These same design forces increasingly are reflected in today's GPP architectures. Power scarcity has pushed GPPs to multi-core architectures, where a processor design is reused in ever larger *arrays* of cores.

Increasing processor complexity increases processor design cost, while fragmenting the GPP market reduces the available design dollars. These forces imply the need to reuse designs. Multi-core arrays thus comprise a fixed number of core *types* (e.g., the Cell processor's SPE and PPE), independent of the *number* of cores in the array. Tile architectures, introduced in RAW and adopted in TRIPS, fully embrace these design forces.

Regarding the need to limit resistive delays and localize communication, we expect that buses, which still are used today in multi-core architectures (even the Cell), will give way to a communication network that permits higher communication concurrency but which scales, such as is used in the RAW architecture. We expect to see networks use of at least a *2D torus* topology, since its worst case latency is *half* that of a simple 2D mesh but still admits a linear area embedding in the plane. We expect similar communication topology enhancements in the execution tile arrays of EDGE (2D VLIW) architectures, reducing hyperblock completion times. In the RAW architecture, first-level cache is interleaved with the processors, enabling compilers to increase communication locality. EDGE architectures use a 2D geometry for the execution tiles, but a 1D geometry for the registers and data caches. If the execution tile array is increased significantly, issues of locality may require a 2D geometry for register and cache tiles. Doing so presents compiler challenges/opportunities.

There is no question that the highest performance special-purpose processors will continue to aggressively exploit the underlying technology with custom circuits. We expect this to bear fruit for regular computation on streams, as has always been the principle province of systolic arrays. However, to the extent that applications get less regular and market demand is smaller (than, say, for graphics processing), purely custom circuits are prohibitively costly. We see a sweet spot in the middle of the 2D taxonomy of Fig. 11. This sweet spot is for chips that are less costly than purely custom circuits but which perform better than those designed on an entirely reconfigurable substrate. They are more generally applicable than a single-purpose chip but perform better than a GPP. This area, we believe, is likely to see a lot of research and development.

Increasing complexity, in both applications and technology, requires increased specialization of labor. This appears to result in a multi-layered design process involving expertise of many varieties. The layering allows progress in each area to proceed *concurrently* (e.g., in compiler technology and FPGA technology). The layering also allows activities in one layer to adapt to advances in other layers in a controlled manner. Thus, the entire vertical process can advance quickly without becoming chaotic (from unwanted inter-layer interaction).

Hardware reconfigurability will be increasingly exploited. The RAW architecture's programmable network presages an important area of research into programmable communication substrates. For now, this programmability is exposed only to the compiler. We see such programmability being exposed, in some form, at the application level, especially to deal with issues that are most tractable at that level. We also see the compute substrate evolving from a static

architecture to a reconfigurable architecture that is exposed to the compiler, with attendant challenges and opportunities for efficient, effective compilation. Extending a reconfigurable architecture, in some form, to the application program's computational model presents exciting opportunities. We expect hardware reconfigurability to yield increasingly intense interdisciplinary research in programming languages, computational models, algorithms, compilers, computer architecture, and FPGA design.

Advances in materials may be more dramatic in the next 20 years than they were in the last 20 years. They will continually change the "fundamental" rules of our game. Since, indeed, every layer in the design process is likely to advance significantly in the next 20 years, their combined affect may be much more dramatic in the next 20 years.

Finally, although increased complexity requires increased specialization of labor, the winning culture shuns tribalism and abhors every notion of sacred. It is marked by open-mindedness, rationality, and inclusiveness. Our disciplines and organizations must have boundaries. But, boundaries are a playground for our creative spirits.

References

1. A. Agarwal, "Raw Computation," *Sci. Am.*, vol. 281, no. 20, August 1999, pp. 44–47.
2. A. O. Balkan, G. Qu, U. Vishkin, "A Mesh-of-trees Interconnection Network for Single-chip Parallel Processing," in *Proc. IEEE 17th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, September 2006, pp. 73–80.
3. S. N. Bhatt, F. T. Leighton, "A Framework for Solving VLSI Graph Problems," *J. Comput. Syst. Sci.*, vol. 28, 1984, pp. 300–343.
4. S. N. Bhatt, F. R. K. Chung, J.-W. Hong, T. Leighton, A. L. Rosenberg, "Optimal simulations by butterfly networks," in *The 20th Annual Symp. on Theory of Computing*, ACM Press, Chicago, May 1986, pp. 192–204.
5. P. Cappello, "Multicore Processors as Array Processors: Research Opportunities," in *Proc. IEEE 17th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, September 2006, p. 169.
6. P. R. Cappello, "Towards an FIR filter tissue," in *Proc. Int. Conf. Acoustics, Speech, and Signal Processing*, Tampa, March 1985, pp. 276–279.
7. C. Clos, "A Study of Non-blocking Switching Networks," *Bell Syst. Tech. J.,* vol. 32, 1953, pp. 406–424.
8. CoWare, "SANYO Selects CoWare's LISATek Products & Services for Custom DSP Design," Accessed on 2007/4/7: http://www.coware.com/news/press449.htm.
9. W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. A., N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, I. Buck, "Merrimac: Supercomputing with Streams," in *SC'03*, Phoenix, AZ, November 2003.
10. B. R. Engstrom, P. R. Cappello, "The SDEF programming system," *J. Parallel Distrib. Comput.*, vol. 7, 1989, pp. 201–231.
11. P. Gachet, C. Mauras, P. Quinton, Y. Saouter, "Alpha du centaur: a prototype environment for the design of parallel regular algorithms," in *Proc. of the 3rd Int. Conference on Supercomputing*, 1989, pp. 235–243.
12. J. Gebis, S. Williams, C. Kozyrakis, D. Patterson, "VIRAM1: A Media−Oriented Vector Processor with Embedded DRAM," in *Proc. DAC*, June 2004, http://iram.cs.berkeley.edu/papers/dac2004.pdf.
13. IBM, "Cell Broadband Engine Programming Tutorial," downloadable PDF, December 2006, version 2.
14. D. Kleitman, F. T. Leighton, M. Lepley, G. L. Miller, "An Asymptotically Optimal Layout for the Shuffle-Exchange Graph," *J. Comput. Syst. Sci.*, vol. 26, 1983, pp. 339–361.
15. R. Kumar, V. Zyuban, D. M. Tullsen, "Interconnections in Multicore Architectures: Understanding Mechanisms, Overheads and Scaling," in *Proc. 32nd Int. Symp. on Computer Architecture (ISCA'05)*, June 2005, pp. 408–419, http://ieeexplore.ieee.org/iel5/9793/30879/01431574.pdf?isnumber=&arnumber=1431574.
16. H.-T. Kung, C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings 1978*, I. S. Duff and G. W. Stewart (Eds.), SIAM, 1979, pp. 256–282.
17. S. Y. Kung, S. C. Lo, S. N. Jean, J. N. Hwang, "Wavefront Array Processors—Concept to Implementation," *Computer*, vol. 20, no. 7, July 1987, pp. 18–33, IEEE Computer Society Press.
18. F. T. Leighton, "New Lower Bound Techniques for VLSI," in *Proc. IEEE 22nd Annual Symp. of Foundations of Computer Science*, Nashville, TN, 1981.
19. C. E. Leiserson, "Fat-Trees: Universal Networks for Hardware Efficient Supercomputing," *IEEE Trans. Comput.*, vol. 34, no. 10, October 1985, pp. 892–901.
20. R. McDonald, D. Burger, S. Keckler, "TUTORIAL: Design and Implementation of the TRIPS EDGE Architecture," in *ISCA*, August 2005, http://www.cs.utexas.edu/ trips/talks.html.
21. C. Mead, L. Conway, "*Introduction to VLSI* Systems," Addison-Wesley Publishing Co., Menlo Park, CA, 1980.
22. NOCS, "International Symposium on Networks-on-Chip," May 2007, Accessed on 2007/4/7: http://nocsymposium.org/.
23. R. Santos, R. Azevedo, G. Araujo, "2D-VLIW: An Architecture Based on the Geometry of Computation," in *Proc. IEEE 17th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, September 2006, pp. 87–92.
24. M. Taylor, "Evaluating the Raw Microprocessor: Scalability and Versatility," *in Int. Symp. on Computer Architecture (ISCA)*, June 2004, http://cag-www.lcs.mit.edu/raw/documents/talks.html.
25. M. B. Taylor, W. Lee, S. Amarasinghe, A. Agarwal, "Scalar Operand Networks: On-chip Interconnect for ILP in Partitioned Architectures," in *Proc. Int. Symp. High Performance Computer Architecture*, February 2003.
26. Tensilica, "The Xtensa 7 Processor for SOC Design," Accessed on 2007/4/7: http://www.tensilica.com/.
27. Wikipedia, "Kolmogorov complexity," January 2007, http://en.wikipedia.org/wiki/search:Kolmogorovcomplexity.
28. Xilinx, "Xilinx Virtex-II Pro FPGAs," Accessed 2007/4/7: http://www.xilinx.com/products.

Peter Cappello received his Ph.D. degree in Computer Science from Princeton University in 1982. He is a Professor of Computer Science at the University of California, Santa Barbara. His research interests include cluster computing, parallel processing, multiprocessor scheduling, market-based resource allocation, and VLSI architectures for digital signal processing.