Dark Memory and Accelerator-Rich System Optimization in the Dark Silicon Era

Ardavan Pedram, Stephen Richardson, and Mark Horowitz
Stanford University

Sameh Galal
Citadel LLC

Shahar Kvatinsky
Technion

Editor's note:
Unlike traditional dark silicon works that attack the computing logic, this article puts a focus on the memory part, which dissipates most of the energy for memory-bound CPU applications. This article discusses the dark memory state and present Pareto curves for compute units, accelerators, and on-chip memory, and motivates the need for HW/SW codesign for parallelism and locality.
—Muhammad Shafique, Vienna University of Technology

Even though Dennard in 1974 showed how to scale CMOS devices for constant power density as the feature size scaled down by a factor $\alpha =$ (newSize/prevSize) [1], the power density of CMOS processor chips grew exponentially from the mid-1980s to the late 1990s. This power growth resulted both from scaling clock frequency faster than $1/\alpha$ and voltages slower than α [2]. By the mid-2000s, this growing power meant that all computing systems, even high-end servers, had become power limited.

Unfortunately, during this period, voltage scaling essentially stopped. Now, when moving to a technology with feature size scaled by α with respect to the previous generation, gate energy scales by at best α (not α^3 as before). So even when we do not scale clock frequency at all and just try to build α^{-2} processors (to use all transistors available in the same area), the power will increase by α^{-1} which will exceed the power budget. This inability to use, or at least use concurrently, all the gates you can create on a silicon die gave rise to the term "dark silicon" [3].

Today the key challenge in improving performance is how to leverage transistors when they cannot all be used at the same time. Taylor, in his "Four horsemen of dark silicon" paper, characterized the work in this field into four different approaches: shrink, dim, specialize, and technology magic [4].

The simplest approach is to simply not build transistors that cannot be used continuously: only build the number of gates that you can operate concurrently under a given power constraint. Since this number is growing slowly, the resulting die area will shrink with technology scaling. This is the shrink horseman. While the power density of the silicon die does go up as area shrinks, getting power out of the die is not the main problem, e.g., heat pipes work well for this. The main problem is getting the power out of the complete system, whose form factor does not change when the die shrinks. The shrink

Digital Object Identifier 10.1109/MDAT.2016.2573586
Date of publication: XX XXXX *XXXX; date of current version:* *XX XXXX XXXX.*

2168-2356/16 © 2016 IEEE

approach makes the computing device cheaper to manufacture, but significantly limits the performance improvement.

Dim tries to leverage all the possible gates/transistors by making some or all of them dissipate less power than before. This dimming generally reduces the performance per unit area, so it must be done in a way that results in better overall performance than simple die shrinking. Two common dimming techniques are lowering the supply voltage to reduce gate energy, and increasing the numbers of gates in a clock cycle to decrease the clock energy and the number of gate evaluations per second.[1] Dimming techniques have been widely used to create today's multicore processors, and have grown quite sophisticated. For example, many processors dynamically adapt their supply voltage so aggressively that they have to lower their clock frequency when they detect small power supply glitches [5]. We show in the "Metrics for energy constrained computing" section that these techniques create Pareto curves in the energy efficiency and compute density metric space. These curves together with the design power, performance, and area constraints can be used to determine the optimal amount of dimming.

The next horseman, specialization, uses the extra transistors to create compute engines highly optimized to specific applications. This specialization can dramatically improve energy efficiency which, in a power-limited world, enables higher performance. Since they run only specific applications, these engines are idle, or dark, most of the time, a perfect fit for dark silicon constraints. Specialized accelerators are widely used in modern processor systems-on-chip (SoCs) and many of these are orders of magnitude more energy efficient than a CPU or a GPU. This dramatic improvement in energy efficiency has led many people to think that this approach is the key to designing a dark silicon chip.

Yet when you look at power dissipation in a CPU chip, around half is in the on-chip memory system [6]. Remembering that most power limitations are really system and not chip-level power limitations, this actually understates the memory problem, since we should really include external DRAM power as well. Thus, the memory system contributes well over 50% of the total system power. So, given Amdahl's law, changing the compute engine without improving the memory can only have a modest (less than two times) change in energy efficiency. The next section explores this issue in more detail, explaining why memory fetches are expensive, how their energy costs grow with memory size, and how to compute the lower bound on an application's energy consumption from the locality of the running application. The unavoidable conclusion is that high performance requires the DRAM and most of the lower levels of memory hierarchy (e.g., last level cache) to be dark almost all of the time. We call this idle memory "dark memory." Given this insight, the "Algorithmic optimization" section describes the critical task for dark silicon systems: optimizing algorithms to maximize their exploitable locality.

Finally, the Deus Ex Machina horseman deals with dark silicon by hoping for a dramatic change in the underlying device technology. While it would be great if a new and better technology/approach was found, we have at least two reasons not to count on it. First, all new technologies take time to reach the manufacturing scale needed to affect computing; even if a new technology is created, it is a decade away from affecting volume computing devices. Given that there is no serious competitor today, computing will use CMOS for at least another decade. Second, waiting for a new "magic" technology abdicates our role in helping to continuously improve computing performance. So the rest of the article focuses on existing mainstream silicon computing, though we will also look briefly at the effect of potential new technologies.

The "Metrics for energy constrained computing" section ties everything together by describing two simple metrics, energy/op and mm^2/(op/s) which enable us to bring all these techniques into a single framework, and thus determine what amount of shrink, dim, and specialization is best for a given design, as well as quantifying the importance of keeping the memory dark and finding optimal cache hierarchy sizes for a given workload. One can use this framework to trade off memory and specialized processors, as well as comparing two applications with different compute and locality patterns.

Why dark memory is essential

The lesson that one quickly learns doing chip design today is that most of the energy is consumed not in computation but in moving data to and from

[1] Lowering the clock speed decreases the number of gate evaluations per second, but, of course, also lowers the performance. The performance loss is often less than the change in clock frequency since the shorter pipeline generally has higher architectural efficiency and thus better energy efficiency.

Table 1 Energy per op, in pJ, for various ops in 45 nm. The second column in each group shows energy multiple versus a single add operation.

Operation	16 bit (int)		64 bit (dp)	
	E/op	vs. Add	E/op	vs. Add
Add	0.18	1.0x	5	1x
Multiply	0.62	3.4x	20	4x
16-word RF	0.12	0.7x	0.34	0.07x
64-word RF	0.23	1.3x	0.42	0.08x
4K-word SRAM	8	44x	26	5.2x
32K-word SRAM	11	61x	47	9.4x
DRAM	640	3556x	2560	512x

memory, as can be seen in Table 1.[2] In ASIC-style design, given, e.g., the computational graph shown in Figure 1, links between functional blocks are implemented as wires. As the communication complexity grows, the hardware block gets bigger, making the wires longer, increasing the communication energy. Computers avoid this wire problem by serializing the computation, computing a few operations each cycle. However, it now needs to store the intermediate results, which used to flow in wires from one logic unit to the next, so it can access them when needed. Thus, the size of the memory needed is related to the complexity of the communication in the algorithm, so the energy increases with communication complexity.

Memory energy

While the access energy of a memory depends on many factors, to first order it grows as the square root of its size, which roughly corresponds to the length of the wires that need to transport the address and data values across the memory array. This was noted long ago by Amrutur and Horowitz [8], citing even earlier work by Evans and Franzon [9]. The memory energy also depends on the fetch width, but that dependence is much weaker than you might expect. For example, moving from 16- to 64-b fetches only changes the energy by 1.5x, so wider fetches are generally more efficient in terms of energy per byte.[3] This means that for a 16-b machine, a fetch from even a 4K-entry memory block costs over ten times the energy of a multiply operation, as we saw in Table 1.

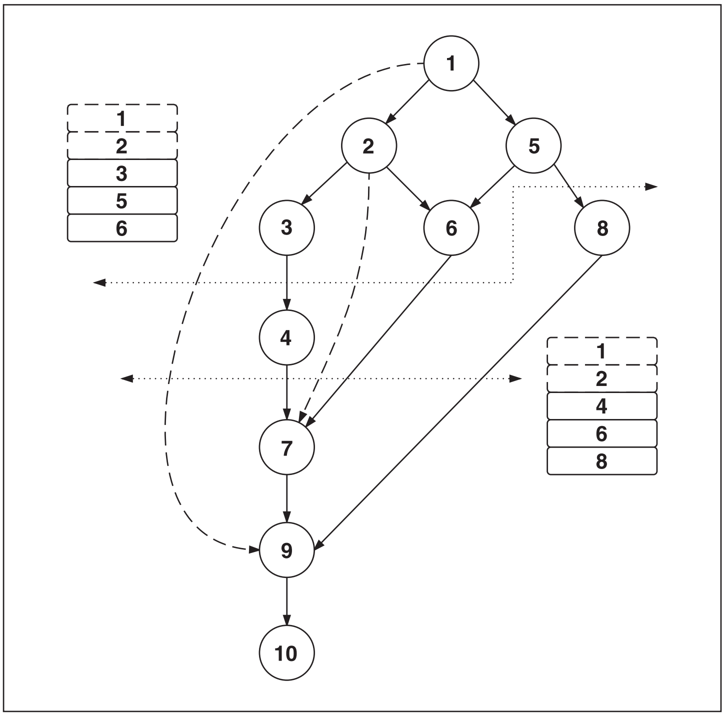

Figure 1. Example data flow graph showing why wire length grows as communication become more complex. When this algorithm executes sequentially, the values on the wires that are still live in the execution trace are named and stored in a memory so they can be accessed when needed. The size of this live set sets the size of the required memory. When new dependences are added from nodes 1 to 9 and nodes 2 to 7 (dashed lines), this adds two new registers 1 and 2 to the required memory at each indicated cut.

To minimize memory energy costs and improve performance, we create memory hierarchies, so that most of the accesses can be satisfied by small local memories. Given that a normal operation requires three data fetches—two operands and one result—it is essential that the register file energy be as small as possible. The register energy is significant: for 16-b arithmetic, the cost of these three fetches, 36 pJ, exceeds the cost of a simple add operation, 18 pJ, even when using a small 16-entry register file.

The situation is actually worse than this since the actual cost of the operand fetch is higher than just the register file energy. It also took energy to load the value into, and store the result from, the register file. This additional energy is set by the

[2] Energy for memory and integer ops come from Verilog, placed and routed using commercial tools. Energy for floating-point ops come from the Galal thesis [7], which also used data from placed and routed designs.

[3] Internally, most SRAMs fetch 64–256 b on each access, so returning a small number of bits increases the effective energy cost per bit. To address this issue, you could create a SIMD machine and fetch the 16-b data for four lanes from a single SRAM. While this is more efficient, it also makes the memory four times larger, since it now needs to hold four lanes' worth of working set, so the benefit is modest.

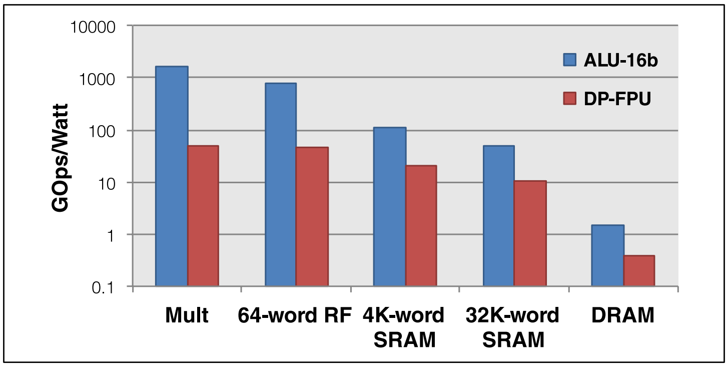

Figure 2. Effective number of ops/s/W (ops/J) if one operand for that operation is fetched from the indicated memory, and the others come from the register file. For 16-b ops even a small 4K word memory throttles the performance per watt.

number of ops performed per register file load instruction, and grows as the register file gets smaller. This additional energy cost from needing to "load/store" values from/to a lower (slower) level in the memory hierarchy exists until you get to DRAM, and can be significant: since the energy of a DRAM access is often two orders of magnitude larger than a local memory access, the overall hit rate of the on-chip memory system needs to be better than 99% for the DRAM not to dominate the overall memory energy.

While this seems to argue that larger memory hierarchies are better, both die cost and leakage constrain memory size. The problem is that while idle SRAM may be dim, it is never completely dark. Each memory cell has a small leakage current such that SRAM dissipates static power, which can be a large issue for a battery-operated device. If the average activity of the device is low, minimizing this leakage moves the optimal point to smaller memory sizes, which increases DRAM activity and results in a higher energy cost for each memory access.[4] Leakage energy and access energy both increase as the memory gets larger, and this leads to a minimum memory cost, which is set by the application's locality. The "Algorithmic optimization" section shows methods to improve the locality of the algorithm we use, and the "Metrics for energy-constrained computing" section shows how to find an optimal memory hierarchy for this improved algorithm.

Another way to view memory's energy constraint is shown in Figure 2, derived from the energy numbers of Table 1. Figure 2 plots the maximum number of operations per second for a watt of power, assuming that one of the operands needs to be fetched from the memory indicated. Fetching one operand essentially assumes that the operations perfectly cascade, so the output of the operation is stored into the register file and then read out as the other operand for the next operation. For simple 16-b operations, accesses to even a small memory are very costly (10x GOPS/W when going from Mult to 4K SRAM in the table), while for more expensive 64-b operations, first level cache accesses only triple the energy cost (from about 45 GOPS/W Mult to 15 GOPS/W 4K SRAM). For 64-b FP, it is the last level cache and DRAM accesses that have a dramatic effect. It is important to remember that this limitation is independent of the degree of parallelism of the application or the hardware. For memory, parallelism does not change the energy/access, and thus does not change the peak bandwidth in a power-limited system.

Emerging memory technologies

Recently there has been an increasing interest in new memory technologies fueled by the possibility that more radical developments in memory or interconnect technology will emerge. Examples of these technological changes include increasing on-die memory using existing or emerging technologies such as eDRAM [10], STT-MRAM [11], RRAM [12], PCM [13] or 3-D Xpoint [14], to using RRAM, PCM, or Xpoint to replace DRAM or adding an additional level after DRAM in the memory hierarchy. Most of these technologies are nonvolatile so have a low leakage state, and can be stacked to yield very high densities. These new technologies are proposed for creating large memories, and these large memories will need long wires to distribute the address and data. Thus, while the length of these wires might be shorter than in DRAM, they will still be long enough to require significant energy compared to computation, and must be used infrequently. Hence, the need for dark memory is an inherent issue in the design of the system for any reasonable memory solution in the foreseeable future.

[4] Another option is to power down the on-chip memory during idle periods, but this too increases overall memory energy since now the dirty cache data need to be written to DRAM on power-down, and additional DRAM fetches are needed to bring the data back into the cache when it is powered back on.

Given the criticality of keeping the memory hierarchy—especially the DRAM—dark, the first part of accelerator design, is not about the hardware: it is to find a way to execute the application using an algorithm that minimizes DRAM accesses and has high chip-level locality, especially when parallelized, as described in the next section.

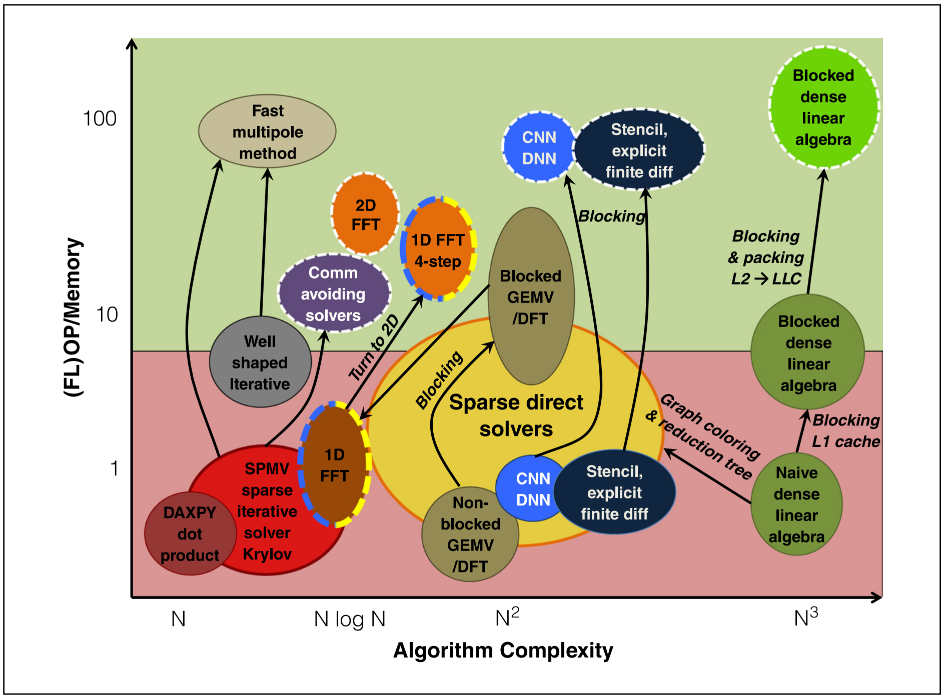

Figure 3. Complexity versus computation/memory-access ratio for several algorithms. Dashed algorithms increase algorithm complexity for efficient implementation.

Algorithmic optimization

Given the high cost of memory accesses, algorithm optimization primarily focuses on minimizing DRAM and low-level cache accesses, and secondarily creating parallelism that can be exploited on chip. The simplest optimizations involve blocking, which splits and reorders loops to increase locality. In this context, it is possible to unroll a loop in hardware, creating parallelism for the hardware to exploit. Often these methods are not enough, however, and a new lower communication approach to the problem is needed. That approach can have a higher computation cost, but if the energy is communication dominated, it is still more energy efficient.

Exploiting locality and blocking

We will use GEneral Matrix Multiplication (GEMM) $A \times B + = C$ as an example to see how blocking can reduce DRAM accesses and consequently save energy.[5] At first GEMM looks like it should be computation dominated, since for n by n matrices it accesses $3n^2$ memory locations (read two and write one matrix) and performs $2n^3$ operations. The problem arises with the required working set of a naive implementation, since to create one row of the output requires reading the entire B matrix, which can be very large. As a result, this matrix must be reread n times, leading to n^3 memory operations and low FLOPS per DRAM access as depicted in Figure 3 (as "naive dense linear algebra").

However, by reordering the computation, we can greatly increase the locality. If we view each matrix as composed of a number of smaller $b \times b$ matrices, each entire submatrix can be stored in a $b \times b$ block of memory on-chip. Now if we iterate over these submatrices, we need to refetch the B matrix only n/b times, reducing the DRAM accesses down to $2n^3/b + 2n^2$ accesses [15]. This technique can be applied recursively, blocking each submatrix into a higher level of the memory hierarchy, with the highest level blocked into the register file. Adding this on-chip memory increases the area and power dissipated by the chip, but causes the system power to greatly decrease by keeping the DRAM dark. As Figure 3 demonstrates, blocking can improve many computations, including algorithms for dense linear algebra [15], [16], [17], convolutional neural networks [18], the four-step fast Fourier transform (FFT) [19], [20], [21], and many others.

Sequential to parallel

Locality is also critical when mapping an application to parallel hardware, since it is best if the parallel executions use mostly local data. Both data and task parallelism can be exploited in hardware design, which often requires small algorithmic changes to

[5] As part of the BLAS scientific computing library, GEMM is essential to innumerable applications, including data parallel applications.

remove minor dependencies in the sequential code. Data parallelism is often exploited by taking one of the blocked loops and unrolling it so each loop iteration is done by a different piece of hardware, while task parallelism is exploited by building a hardware block for each task, and using wires to handle the producer/consumer communication.

Parallel execution generally decreases the energy required for memory that is strictly local to the unit, since in this case the original memory is partitioned into many smaller memories with one memory embedded into each parallel unit. The energy required for memory storing shared data generally goes up, since now these data must be communicated to all the cores, which are large in size due to their private memory. We will again use GEMM to demonstrate this issue. To create a parallel GEMM execution, we distribute the rows of A to different cores and broadcast the columns of B to all the cores so each core produces unique rows of C. Since the A and C matrices are partitioned among the cores, the working set in each core is smaller, since it only needs to hold a fraction of the total matrix. The memory required for the B matrix remains the same size, but now its output needs to be broadcast to all the cores [22]. The energy required to distribute this information is proportional to the square root of the area that all the cores occupy, which is related to the total memory used in all the cores (plus the overhead of the hardware), and is often larger than the energy needed to fetch B from its memory. This overhead makes it critical for parallel algorithms to limit the total communication between parallel units, or restrict them to physically adjacent units.

Changing the nature of the algorithm

While it may be possible to get the required locality and parallelism through blocking, sometimes a very different approach is needed to reach the desired performance. Here the application developer needs to take broader look at the problem, to see if there are problem symmetries or simplifications that can be exploited, different approaches to try, or constraints that can be relaxed. For example, in linear algebra, different variants of algorithms show different behaviors in various levels of the memory hierarchy so the specific choice of variant affects locality and performance [23], [24]. Another example is the FFT, which exploits symmetries in the DFT to dramatically reduce the complexity of computing a Fourier transform [25].

A classical example depicted in Figure 3 is the solution of sparse systems. The most straightforward method is to use expensive $O(N^3)$ dense direct methods that do not take advantage of sparsity in the data structure. Sparse direct solvers use techniques such as reordering the data, graph coloring [26], and constructing dependence trees to preserve nonzero patterns in the matrix and so avoid performing computations with zeros, all while improving parallelism [27]. This drops the computations[6] down to at most $O(N^2)$ in spite of various overheads for extra complexity. In contrast, iterative solvers reduce computations by performing a sequence of improving approximate solutions that are much cheaper in complexity [e.g., $O(N^2)$] and (for well-conditioned matrices) converge after a few iterations [28]. However, each iteration consists of low-performance memory-bound kernels such as (sparse) matrix–vector multiplication. Communication-avoiding algorithms can replace these memory-bound kernels with GEMM-like kernels to improve the locality and performance at the cost of slightly slower convergence rate and more computations [29], [30].

Other approaches relax some constraints in the original problem. For example, iterative refinement techniques use high precision arithmetic for lower order residual computation and then use lower precision arithmetic for high-order less sensitive linear solve kernels [31]. This method can speed the computation by up to two orders of magnitude and can be generalized for solving linear least square problems, eigenvalue/singular value computations, and sparse solutions such as conjugate gradient [32]. Or parallel applications can allow cores that update shared state to be stochastic with respect to other processors. Both of these methods sacrifice convergence rate to decrease communication for each computation round.

This reduction of constraints is widely used in applications that use randomized algorithms, which are becoming popular especially in domains such as machine learning and principal components analysis (PCA) where approximate but fast results are desired. Such methods select a random subset of the initial input data and reduce substantial parts of the computation while still managing to converge on a desired result [33]–[35].

[6] For matrices whose graphs can be embedded in at most three dimensions.

Metrics for energy-constrained computing

To formalize the tradeoffs discussed in the previous sections we will assume that we are building a system on chip (SoC) with specialized hardware designed to solve a data parallel problem, and that we have constraints on, or want to optimize combinations of performance, power, and chip area.[7] To solve this optimization problem, we can place every possible design combination in a 3-D space, where the x-axis is chip area, the y-axis is power, and the z-axis is performance. In this space, it is easy to remove designs that can never be optimal: designs with the same area and power as another design but lower performance, designs with the same performance and area but higher energy, or designs with the same performance and energy by larger area. Removing these suboptimal designs will leave a 2-D surface of designs that might be optimal.

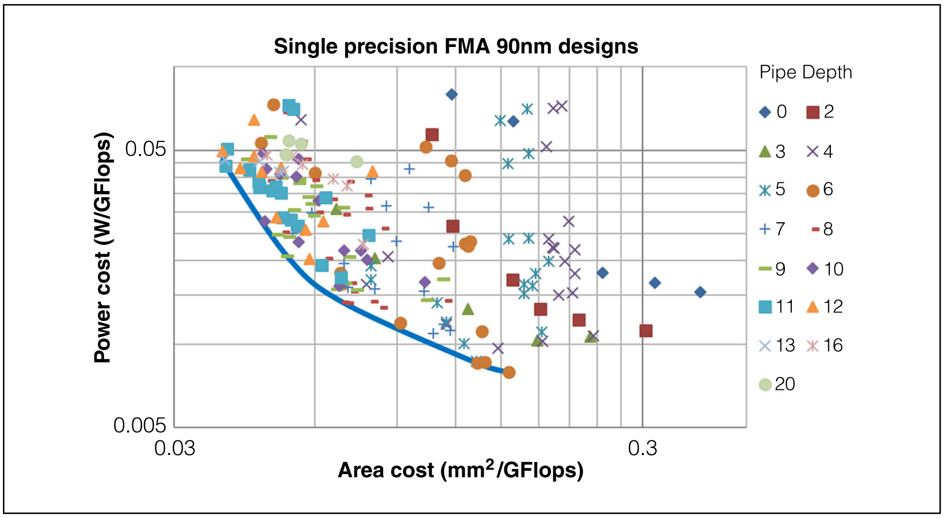

Figure 4. Mapping of a large design space of fused FMADD designs. Each dot represents a different variation on the base design; for example, all the diamond shapes represent various unpipelined versions, squares have a pipe depth of 2 and so on up to a 20-deep pipeline. Most of the designs are strictly worse in the sense that they either take more area or more energy than one of the other designs. The left-hand edge is the edge of the feasible space, and these designs are optimal for some design constraints [36].

Fortunately, we can simplify this space further by recognizing that we are solving a data parallel problem. In this type of problem, we assume you can double the throughput (the performance) by doubling the hardware (the power and area). What this means is that each design is not a point in the 3-D performance space, but a line. To convert a design back to a point, we divide the area and power axes by the performance of the design (since both of these parameters are proportional to performance) and end up with a 2-D metric space: power/performance, or energy/op; versus area/performance, or mm^2/op/s.

Joules/op and mm^2 /(ops/s) metrics

As in the 3-D case, it is easy to find nonoptimal designs. Any design that has a higher energy/op with the same compute density as another design can never be the best design. Similarly, if two designs have the same energy/op, the one with a higher mm^2/(ops/s) cost cannot be optimal. Figure 4 shows the result of evaluating the design space for an FP fused mult-add unit, and exploring different microarchitecture, pipeline depth, gate sizing, cell libraries, and Vdd settings. From an energy-efficient design perspective, we can completely characterize this design space, which includes the effect of dimming, by the shape of its Pareto curve (the left-hand edge of the feasible space), which is shown in Figure 5a.

These two metrics nicely capture many of the tradeoffs we have discussed previously. As we dim the silicon, we create designs with lower energy/op, but they will also operate slower, which moves along the Pareto curve. Similarly, adding a level in the memory hierarchy may decrease the energy of an access, but will also increase the area required, contributing another design point to the Pareto curve.

To show why Figure 5a is so powerful, Figure 5b plots the power and area of an accelerator, and shows some possible design constraints. Note that the lines of constant performance shown in this plot are simply the Figure 5a curve scaled by different throughput numbers. So finding the maximum performance point for $P < P_{\max}$ and $A < A_{\max}$ is the

[7] Talk of free transistors aside, die area is still important to consider. It strongly affects cost when you sell parts in large (10^6) volumes, and low volume parts still have area constraints they cannot exceed.

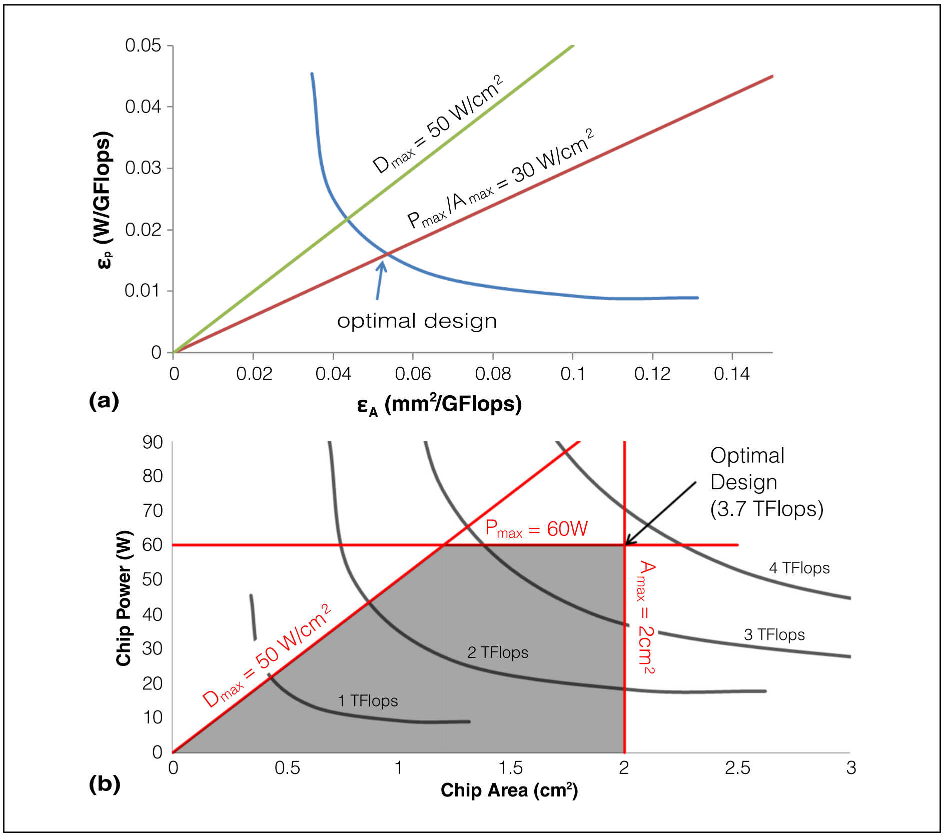

Figure 5. (a) Determining the optimal design point from a throughput-energy tradeoff curve and constraints. (b) Contour map of achievable throughputs versus area and power. Constraints of A_{max} = 2 cm², P_{max} = 60 W, and D_{max} = 50 W/cm² are indicated [7].

same as finding the point (ε_A, ε_P) in Figure 5a where (energy/op)/(mm²/(ops/s)) $= P_{\max}/A_{\max}$, and the resulting performance is $A_{\max}/\varepsilon_A$. Other optimization objectives can be mapped to a curve in this space, allowing them to be solved as well, including optimizing for total cost of ownership. For more details see Galal's work on energy-efficient FPU design [36].

If the algorithm is fixed, one can use any definition of an op in these metrics, since this optimization does not change the number of ops. However, if we need to compare designs across different algorithmic approaches, it is essential to define op to be something that is invariant across the different implementations. For example, using FLOPs to compare sparse and dense algorithms would be a bad idea, since a dense implementation would have much lower energy/FLOP and area/FLOP/s, but would require many more FLOPs than a sparse solver, and would look worse on the curve. Similarly, when trading off among different possible implementations, it is important that they all use the same op definition.

Accelerator optimization

Another advantage of using Pareto curves rather than a specific design point is that the curve provides information about marginal cost in area or energy if you need to change the design. While these marginal costs assume you can add fractional compute units to get fractional performance, which is clearly wrong, they do provide the insight needed to create efficient solutions. To demonstrate how they can be used for accelerator evaluation, assume our application is running on a scalable machine and we want to minimize this machine's power by adding some specialized accelerators while staying within the chip's current area and performance budget. Since we are assuming the base machine and accelerator area scale with performance, moving computation from the base machine to the accelerator will provide area that the accelerator can use. The accelerator will improve the energy of the machine if it has a lower energy/op when operating at the same mm²/(op/s) as the base machine. Since the compute density is the same, this new solution should require the same area as before.

The previous step verified that the accelerator can reduce energy/op versus the original system, but the resulting design is not necessarily optimal: to ease the comparison we chose points that had the same compute density, and left the base design alone. We need to change both to get the optimal power. Fortunately, like most constrained optimization problems, the optimal area allocation can be found by balancing marginal costs: at the optimal point, the change in energy/op per change in mm²/(op/s) in the two compute units must be the same. Moving an increment of work lowers the energy of the unit losing the work by its marginal cost, while the unit gaining the work increases its energy by its marginal cost. If these are not the same, moving work from the unit with higher marginal cost to the one with lower marginal cost will save energy (or if the work cannot move because the accelerator is specialized, simply move silicon area in the other direction).

Nonscalable objects

While this method clearly shows how Pareto information lets us optimally allocate area between two compute engines, its assumption of finely partitioned engines is rarely the case. In most designs, the area of a block cannot be smoothly changed. Processors/accelerators can be scaled by duplication, but since each unit contains compute/control/memory they are generally of significant size. The result is one cannot really incrementally move area from one unit to the other. Instead you can only make much coarser grain moves. This quantization makes finding the exact answer harder, since now we need to solve a mixed integer program; but the basic intuition remains the same: If the marginal cost of an accelerator A_1 is lower than a second unit A_2, test

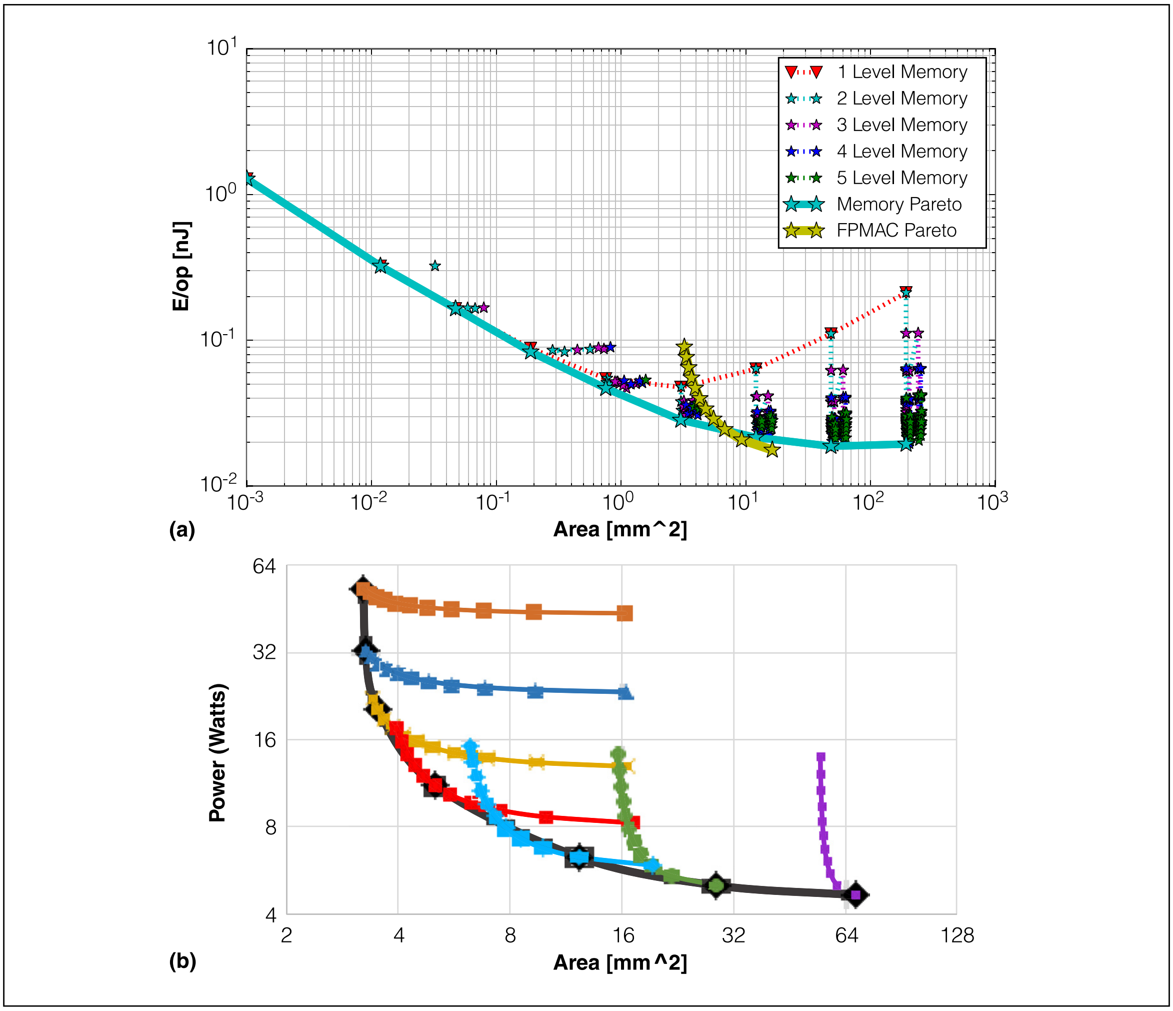

Figure 6. (a) Effect of multiple layers of on-chip memory on the energy and area tradeoffs for GEMM. As the area grows, more memory levels are needed in the optimal design. One level memory is registers and DRAM, two levels has registers, local memory and DRAM, etc. (b) Pareto curve of 256-GFLOP GEMM accelerator, shown in black. This was generated by finding the FMADD design that matched the margin cost of the memory system. Also drawn are the systems that would result by pairing different FMADD tradeoff choices to the optimal memory design points showing other potential designs, most of which are highly suboptimal.

to see if you can reduce the size of A_1 enough to give A_2 enough area so it can move to a more energy-efficient design. This might involve lowering the performance of each existing A_2 compute unit, and then adding a new one to maintain aggregate throughput. If enough area cannot be created, the best alternative is to try to use the area in A_1 to reduce its energy cost.

Dealing with the memory system adds a new challenge. While the register files and first level caches are duplicated with the compute units, the levels in the memory hierarchy closer to DRAM (last level cache, and sometimes even the L2) are shared and so their area is not proportional to the computing throughput. Fortunately, like a compute unit, one can create a Pareto curve for a memory system. The y-axis remains energy/op, but now it represents the average memory energy used for each processor op. Since area does not scale with performance, the x-axis is just area. Like compute units, the different memory configurations will collectively generate a single Pareto curve, where larger area reduces the average memory cost, by filtering out more of the DRAM accesses.

This memory Pareto curve has exactly the form we need to find the optimal allocation between memory and computation. We just scale the compute curve by the desired aggregate performance so its Pareto curves also indicate the tradeoff between area and energy/op, and the energy optimal design will balance the marginal cost between the two units.

Figure 6 shows how this is done for a GEMM accelerator. Using the known access pattern of the algorithm, the required memory energy per fused multiply/add is found for all possible memory configurations. We explore 1–5 levels of on-chip memory hierarchy in addition to the DRAM, and try many different potential memory sizes for each level. Most of these configurations are not optimal, but a few form the Pareto curve (in turquoise). This curve shows how the memory energy changes from 1 nJ/FMADD to around 20 pJ/FMADD as the area changes from 0 to 100 mm^2. Also shown in Figure 6a is the Pareto curve of an FMADD running at 256 GFLOPS. To generate the power and area curve for the entire system, we add the energy and area cost of the FMADD design at each point in the memory Pareto curve. This results in the many curves shown in Figure 6b. Overlaid on these curves is the overall Pareto curve, which is shown in black which uses the FMADD design which matches the marginal cost of the memory system. Not surprisingly, the small area solutions chose high compute density FMADD solutions, since the memory system dominates the energy, while large memory area solutions use low energy, and area-inefficient FMADD. The result is that even though the total power ranges by nearly 10x, in most of these designs, the compute energy and memory energy are roughly 50/50.

The large energy cost of memory fetches limits the overall efficiency of applications no matter how efficient the accelerators are on the chip. As a result, the most important optimization must be done at the algorithm level, to reduce off-chip memory accesses, to create dark memory. The algorithms must first be (re)written for both locality and parallelism before one tailors the hardware to accelerate them.

Using pareto curves in the energy/op and mm^2/(op/s) space allows one to quickly evaluate different accelerators, memory systems, and even algorithms to understand the tradeoffs between performance, power, and die area. This analysis is a powerful way to optimize chips in the dark silicon era. ■

References

[1] R. H. Dennard, F. H. Gaensslen, H. N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted MOSFETs with very small physical dimensions,” *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256 – 268, 1974.

[2] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, “CPU DB: Recording microprocessor history,” *Commun. ACM*, vol. 55, no. 4, pp. 55–63, 2012.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Proc. IEEE 38th Annu. Int. Symp. Comput. Architect.*, 2011, pp. 365–376.

[4] M. B. Taylor, “Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse,” in *Proc. ACM 49th Annu. Design Autom. Conf.*, 2012, pp. 1131–1136.

[5] A. Grenat, S. Pant, R. Rachala, and S. Naffziger, “Adaptive clocking system for improved power efficiency in a 28nm x86-64 microprocessor,” in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, 2014, pp. 106–107.

[6] M. Horowitz, “Computing’s energy problem (and what we can do about it),” in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, Feb. 2014, pp. 10–14.

[7] S. Galal, "Energy efficient floating-point unit design," Ph.D. dissertation, Stanford Univ., Stanford, CA, USA, 2012.

[8] B. S. Amrutur and M. A. Horowitz, "Speed and power scaling of SRAM's," *IEEE J. Solid-State Circuits*, vol. 35, no. 2, pp. 175–185, 2000.

[9] R. J. Evans and P. D. Franzon, "Energy consumption modeling and optimization for SRAM's," *IEEE J. Solid-State Circuits*, vol. 30, no. 5, pp. 571–579, 1995.

[10] S. Narasimha et al., "22nm high-performance SOI technology featuring dual-embedded stressors, epi-plate high-K deep-trench embedded DRAM and self-aligned via 15LM BEOL," in *Proc. IEEE Int. Electron Devices Meeting*, Dec. 2012, pp. 3.3.1–3.3.4.

[11] H. Yoda et al., "Progress of STT-MRAM technology and the effect on normally-off computing systems," in *Proc. IEEE Int. Electron Devices Meeting*, Dec. 2012, pp. 11.3.1–11.3.4.

[12] H.-S. Wong et al., "Metal oxide RRAM," *Proc. IEEE*, vol. 100, no. 6, pp. 1951–1970, Jun. 2012.

[13] H.-S. Wong et al., "Phase change memory," *Proc. IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.

[14] R. Merritt, "3D XPoint steps into the light," *EE Times*, Jan. 14, 2016.

[15] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimization of blocked algorithms," *ACM SIGOPS Oper. Syst. Rev.*, vol. 25, no. Special Issue, pp. 63–74, 1991.

[16] L. Renganarayana and S. Rajopadhye, "A geometric programming framework for optimal multi-level tiling," in *Proc. ACM/IEEE Conf. Supercomput.*, 2004, p. 18.

[17] J. J. Navarro, T. Juan, and T. Lang, "MOB forms: A class of multilevel block algorithms for dense linear algebra operations," in *Proc. 8th Int. Conf. Supercomput.*, 1994, pp. 354–363.

[18] Y. Chen et al., "DaDianNao: a machine-learning supercomputer," in *Proc. IEEE 47th Annu. IEEE/ACM Int. Symp. Microarchitect.*, 2014, pp. 609–622.

[19] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, Philadelphia, PA, USA: SIAM, 1992.

[20] D. H. Bailey, "FFTs in external or hierarchical memory," in *Proc. ACM/IEEE Conf. Supercomput.*, 1989, pp. 234–242.

[21] D. Takahashi, "High-performance parallel FFT algorithms for the Hitachi SR8000," in *Proc. 4th Int. Conf./Exhibit. High Performance Comput. Asia-Pacific Region*, May 2000, vol. 1, pp. 192–199 vol.1.

[22] A. Pedram, R. Van de Geijn, and A. Gerstlauer, "Codesign tradeoffs for high-performance, low-power linear algebra architectures," *IEEE Trans. Comput.*, vol. 61, no. 12, pp. 1724–1736, 2012.

[23] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn, "The science of deriving dense linear algebra algorithms," *ACM Trans. Math. Softw.*, vol. 31, no. 1, pp. 1–26, 2005.

[24] E. Anderson and J. Dongarra, "Evaluating block algorithm variants in LAPACK," Dept. Comput. Sci., Univ. Tennessee, 1990.

[25] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, p. 297, 1965.

[26] M. Naumov, P. Castonguay, and J. Cohen, "Parallel graph coloring with applications to the incomplete-LU factorization on the GPU," Nvidia Corp., Tech. Rep. NVR-2015-001, 2015.

[27] M. T. Heath, E. Ng, and B. W. Peyton, "Parallel algorithms for sparse linear systems," *SIAM Rev.*, vol. 33, no. 3, pp. 420–460, 1991.

[28] R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, Philadelphia, PA, USA: SIAM, 1994, vol. 43.

[29] M. Hoemmen, "Communication-avoiding Krylov subspace methods," Ph.D. dissertation, Univ. California Berkeley, Berkeley, CA, USA, 2010.

[30] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM J. Sci. Comput.*, vol. 34, no. 1, pp. A206–A239, 2012.

[31] J. H. Wilkinson, "Rounding errors in algebraic processes," Courier Corp., 1994.

[32] J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)," in *Proc. ACM/IEEE SC Conf.*, 2006, pp. 50–50.

[33] B. Scholkopf et al., "Input space versus feature space in kernel-based methods," *IEEE Trans. Neural Netw.*, vol. 10, no. 5, pp. 1000–1017, Sep. 1999.

[34] B. Moore, "Principal component analysis in linear systems: Controllability, observability, and model reduction," *IEEE Trans. Autom. Control*, vol. 26, no. 1, pp. 17–32, Feb. 1981.

[35] P.-G. Martinsson, G. Quintana-Orti, N. Heavner, and R. van de Geijn, "Householder QR factorization: Adding randomization for column pivoting," Dept. Comput. Sci., Univ. Texas at Austin, Tech. Rep. FLAME Working Note #78, Dec. 2015.

[36] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 913–922, 2011.

Ardavan Pedram is currently a Research Associate at Stanford University, Stanford, CA, USA. His research interests include high-performance computing and computer architecture. He specifically works on hardware/software codesign (algorithm for architecture) of special purposed accelerators for high-performance energy-efficient linear algebra, machine learning, and signal processing. Pderam has a PhD in computer engineering from The University of Texas at Austin, Austin, TX, USA. He is a member of the IEEE.

Stephen Richardson is currently a Research Associate at the Electrical Engineering Department, Stanford University, Stanford, CA, USA. He has worked in industry at Weitek and MIPS, as well as at Sun Microsystems and Hewlett-Packard research labs. Richardson has a PhD in electrical engineering from Stanford University. He is a member of the IEEE.

Mark Horowitz is the Yahoo! Founders Professor at Stanford University, Stanford, CA, USA and was Chair of the Electrical Engineering Department from 2008 to 2012. He cofounded Rambus, Inc. in 1990. His research interests are quite broad and span using electrical engineering and computer science analysis methods to problems in molecular biology to creating new design methodologies for analog and digital VLSI circuits. He is a Fellow of the IEEE and the Association for Computing Machinery (ACM) and a member of the National Academy of Engineering and the American Academy of Arts and Science.

Sameh Galal currently works at Citadel LLC, Chicago, IL, USA. His research interests include energy efficiency and floating-point unit design. Galal has a PhD in electrical engineering from Stanford University, Stanford, CA, USA. He is a member of the IEEE.

Shahar Kvatinsky is an Assistant Professor at the Electrical Engineering Department, Technion—Israel Institute of Technology, Haifa, Israel. From 2006 to 2009, he was with Intel as a Circuit Designer and was a Postdoctoral Fellow at Stanford University, Stanford, CA, USA, from 2014 to 2015. His current research is focused on circuits and architectures with emerging memory technologies and design of energy-efficient architectures. Kvatinsky has a PhD in electrical engineering from Technion. He is a member of the IEEE.

■ Direct questions and comments about this article to Ardavan Pedram, Stanford University, Stanford, CA 94305 USA; e-mail: perdavan@gmail.com.