Reflections and Research Advice Upon Receiving the 2019 Eckert-Mauchly Award

Mark D. Hill

University of Wisconsin-Madison

This essay presents reflections and research advice based on my June 2019 Eckert-Mauchly award speech at the International Symposium for Computer Architecture (ISCA). It begins by discussing simulation, a prominent method in computer architecture, great for testing and refining hypotheses, but best for only late steps in the scientific method. The essay then backs up to delve into the less-discussed methods of using thinking, models, and taxonomies for developing initial insights and a first hypothesis. It then backs up further to examine how to pick a problem to work on in the first place. It concludes with thanks and a charge to give forward.

Introduction

I am honored, grateful, and humbled to receive the 2019 Eckert-Mauchly award.[1] I am humbled to have my name associated with the luminaries that have preceded me. I have known many prior recipients, from Sir Maurice Wilkes to my graduate student officemate Susan Eggers. Although I'm the recipient of this award, the work is really "our" work, as it stems from the creativity and perspiration of more than 160 co-authors. Figure 1 shows a word cloud with the co-author's names sized roughly logarithmically with the number of papers.

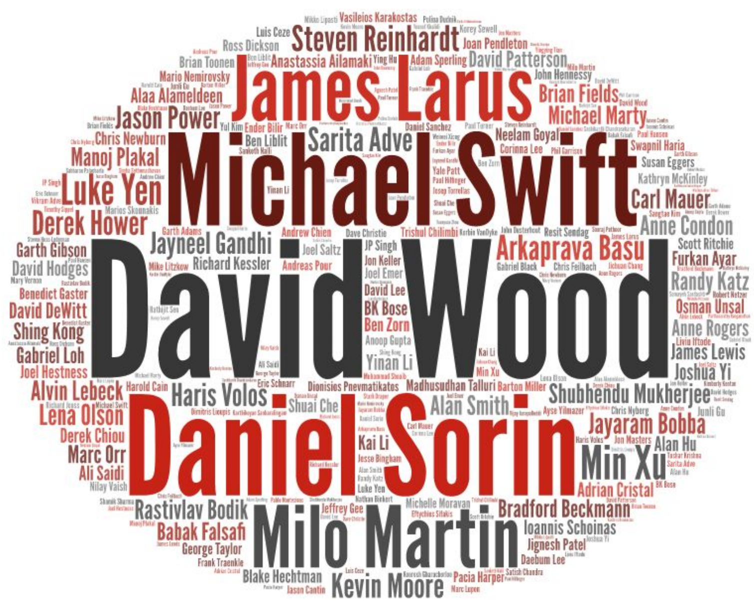

Figure 1: Word Cloud of Co-Authors with Mark D. Hill

[1] https://awards.acm.org/award_winners/hill_2155109

Rather than a historical tour, this essay--and the talk it is based on[2]--gives forward-looking advice on methods that we find valuable for doing research, illustrated with past examples.

Simulation for Testing and Refining Hypotheses

A theme of our work is developing **simulators** to explore questions previously out of reach of existing tools. First, as Ph.D. student with co-advisors David Patterson and Alan Jay Smith, we developed the Dinero uniprocessor trace-driven simulator[3]. Its ease of use and license facilitated its distribution to dozens of universities and a few companies. Second, as mostly assistant professors, we created the *Wisconsin Wind Tunnel* [R93]. WWT simulated a cache coherent shared memory computer on a non-shared memory Thinking Machine CM-5. It was execution-driven--so that memory behavior could influence program execution--but had three flaws (in retrospect): the CM-5 did not get faster while Moore's Law accelerated software-only simulators; we couldn't share it much as the CM-5 was rare; and the simulator didn't model operating system behavior. Third, we developed *GEMS* [M05], which booted an OS (initially Solaris) and included I/O devices. We did this hard task, because repeated feedback at our annual industrial affiliates meetings strongly advocated for full-system simulation. To ease work and speedup development, we implemented the performance model of GEMS but had it do functional simulation by working symbiotically with initially-beta commerical Virtutech SimICS. This symbiosis was a blessing--it worked--but also a curse as it limited the spread of our influence, as not all wanted or could license SimICS. Fortunately, the (former) Michigan folks had implemented full-system functionality in their m5 simulator and proposed that we merge GEMS+m5 to form gem5 [B11].

Our simulators have helped many do better research and have been cited 5,000 times, including some citations where authors explain why they are not using them. We achieved difficult innovations, but we were also "creatively lazy" (which we advocate) wherein we did only what was needed ***after leveraging the work of others***, e.g., SimICS and m5. After Dinero, wrote approximately zero lines of code for these amazing simulators. Key contributors can be found in the author lists at the end of this essay.

You might think that simulation is the most important method in computer architecture, based on both the above paragraph and what you find reading many papers. In fact, simulation is important for testing and refining hypotheses. While it is often the step most visible in papers, this is only one step in the Scientific Method[4] [P64]:

1. Pick a problem.
2. Develop insight and first hypothesis.
3. Test and refine hypotheses.
4. Repeat steps as needed.

[2] Slides at http://pages.cs.wisc.edu/~markhill/papers/markhill_eckert-mauchly_2019.pptx (and .pdf) with unofficial audio http://www.cs.wisc.edu/~markhill/papers/markhill_eckert-mauchly_2019.m4a and almost-complete video https://youtu.be/kqrhBTK6SHE.
[3] 1980s Dinero predated web pages but was later re-released: http://www.cs.wisc.edu/~larus/warts.html
[4] https://en.wikipedia.org/wiki/Novum_Organum

The Scientific Method's use underlies modern science, and its value to computer architecture is no less, even as we sometimes seem to only implicitly apply it. Let's discuss its other steps in reverse order.

Develop Insight and First Hypothesis

1980s 3C Cache Misses. In the 1980s, we were fascinated by the memory hierarchy. The good news for these hierarchies is that--if properly designed--they provide cost-performance that far exceeds that of the technology levels they are created from. The bad news is that their proper design requires setting numerous parameters, often informed by a data deluge. How could we channel the deluge, following Hamming who said, "The purpose of computing is insight, not numbers."? Our intuition said that this was a good problem and turned out to be correct. It is not clear if we were good, lucky, or both. Still, in our experience, intuition matters for choosing research directions among many options.

In my Ph.D. thesis [H87] (or easier to find subset [HS89]) with co-advisors David Patterson and Alan Jay Smith, we sought a **taxonomy** or **model** to give insight into cache misses and used a thesaurus to develop a memorable name. The result was the *3C Model* with *conflict misses* for too little associativity, *capacity misses* for too small cache size, and *compulsory misses* for never previously accessing a block/line. The 3Cs had explicit influence on Norm Jouppi developing victim caches and stream buffers a year later [J90], and went on to join the undergraduate canon. It did so, in part, because my Ph.D. co-advisor co-wrote a popular textbook (Patterson). While the 3C Model was simple relative to other contemporary cache models, this simplicity is a factor in its longevity. We thus learned to prize simplicity.

1990s Memory Consistency with Sequential Consistency for Data-Race Free Programs. In the 1990s, we were sure that shared-memory multiprocessors had "arrived". It turns out we were off by a decade as Moore's Law facilitated microprocessor performance improvements that allowed most markets to avoid multiprocessors, but we were correct that it would eventually happen. This illustrates that good research should anticipate trends, but need not get timing right, as is required for products. Sarita Adve and I--and others--wanted to put the correctness of multiprocessors on a firmer foundation. We saw that cache coherence could make caches invisibile, but what then? Leslie Lamport's *sequential consistency* model was elegant, but most real machines did not obey it. These multiprocessors exposed write buffers, out-of-order execution, and what we now call non-atomic stores. We instead wished to follow Einstein who said, "Everything should be made as simple as possible, but not simpler."

A breakthrough started with a talk by Bart Miller on software datarace detection--and there is a lesson here about how attending talks can lead to new connections. We developed new intuition that there might be a connection between dataraces and the weak or relaxed memory models of the era. Nevertheless, it took hard **thinking** to make the connection both explicit and simple: specify a system to provide sequential consistency (SC) to data-race-free (DRF) programs. The SC for DRF model enabled a "have your cake and eat it too" situation. Hardware could do aggressive reordering for performance between synchronization operations, while almost all programmers could reason with relatively simple sequential consistency. Adve and others subsequently used SC for DRF as the cornerstone of the Java and C++ high-level language memory models, and many of us still use it when specifying memory consistency in

heterogeneous systems with both CPUs and general-purpose GPUs. All this from insight from a talk plus three decades of work.

2000s LogTM Transaction Memory. In the 2000s, we were again sure that shared-memory multiprocessors had "arrived." It turns out that this time we were right, but for a reason whose timing we did not predict: the end of Dennard scaling causing a "right turn" to multicore chips. We--initially Kevin Moore, David Wood, and I--were interested in multiprocessor programmability, now that correctness was arguably under control. The prevailing programming method coordinated threads using locks that were known to not compose and are subject to deadlock. For deadlock, consider a simple method that moves an item between two data structures by obtaining a lock first at the source and then a second lock at the destination. If one thread sought to move an item from A to B while a concurrent thread sought to move it from B to A, deadlock could occur with each thread holding one lock and unable to get the other. *Transactional memory* (TM) offered a potentially elegant solution. With TM, each thread could ask that a method be an atomic transaction and the TM system would "make it so," sometimes having to abort and retry transactions. Existing TM systems, however, allowed micro-architectural elements (e.g., write buffer size and cache associativity) to affect what transactions could commit, required substantial changes to conventional systems, or both.

We sought a TM solution where the micro-architecture would not limit which transactions could commit and required at most a modest change to existing cache-coherent multiprocessors. To drive thinking on what to do, we first developed a **taxonomy**, recreated in Figure 2. In one dimension, TM systems had to detect conflicts among concurrent transactions, either when a transaction sought to commit (lazy) or earlier when reads and writes first occur (eager). Orthogonally on a write, all TM systems must do "version management" to keep the new value for possible commit and the old value for possible abort. Lazy version management puts the new value "on the side" (e.g., in a write buffer) and leaves the old value "in place" (in coherent memory) until commit. Conversely, eager version management saves the old value "on the side" and puts the new value "in place."

		Version Management	
		Lazy	Eager
Conflict Detection	Lazy	DBMSs w/ optimistic CC Stanford TCC	none
	Eager	MIT LTM Intel/Brown VTM	DBMSs w/ locking CC MIT UTM Wisconsin LogTM [new]

Figure 2: A 2006 Taxonomy of Hardware Transactional Memory Systems (adapted from [M06])

This taxonomy assisted us in developing LogTM [M06]. In particular, we decided to focus on the quadrant that modeled commercially-successful database management systems with locking concurrency control (CC). Consequently, LogTM combined eager conflict detection (using coherence) with eager version management (a per-thread log). LogTM--and its successor LogTM-SE [Y07]--enabled unbounded transactions with modest core changes and trivial

memory system changes. While LogTM and other academic papers developed many promising ideas, and limited TM hardware is supported by Intel, IBM, and recently ARM, it is not universal. This is, in part, because, although multicore use is now ubiquitous (e.g., in the cloud), only a relatively few experts program directly with thread-level parallelism. Nevertheless, we should take the long view, as ideas can take time to flourish. For example, SIMD and vectors developed over decades of niche successes before their broader success with general-purpose GPUs and the SIMT model. More generally, we recommend developing taxonomies to structure deep thinking, recalling the "mother of all scientific taxonomies:" Mendeleev's periodic table of the elements that focused efforts for uncovering missing elements.

2010s SoC Gables and Accelerator-Level Parallelism. In the 2010s, Systems of a Chip (SoCs) grew up to have heterogeneous CPUs, GPUs, dozens of accelerators, interconnects, coherence, virtual memory, and even virtualization. Accelerators make specific computations faster, more predictable, and more energy-efficient. As a Google intern (during my 2017-18 sabbatical), my host Albert Meixner charged that we should make SoC design "more scientific." Gasp! Thus, I fell into SoCs by luck after deciding to do another mind-expanding sabbatical in industry, putting myself in a position to get lucky.

To make some progress on this grandiose charge and frame early SoC thinking, Vijay Janapa Reddi developed a simple SoC **model** called Gables [HR19] for SoC hardware and software use cases. Specifically, Gables models each accelerator with a "roofline" (previously used for a whole multicore chip), including the important parameter of "operational intensity" that speaks to whether communication or computation is the bottleneck. George Box said, "All models are wrong; some are useful." The community has yet to decide if the newly-proposed Gables is useful. Nevertheless, it has already led to the important hypothesis that mobile SoCs, in particular--and arguably computing, more generally--must now deal with *accelerator-level parallelism* (ALP) wherein multiple accelerators are concurrently active [HR19b]. Broad ALP success will require the research community--maybe you!--to develop better "best practices" for targeting accelerators, managing accelerator concurrency, choreographing inter-accelerator communication, and productively programming them.

Pick A (Good) Problem

A greatly underappreciated aspect of influential research is the first step of picking a problem from the infinite set of possible problems. This is creative and important, but rarely discussed. Good research problems fall at the intersection of two criteria, "If you can do it, people will care." and "You can do (some of) it," as illustrated by Figure 3.

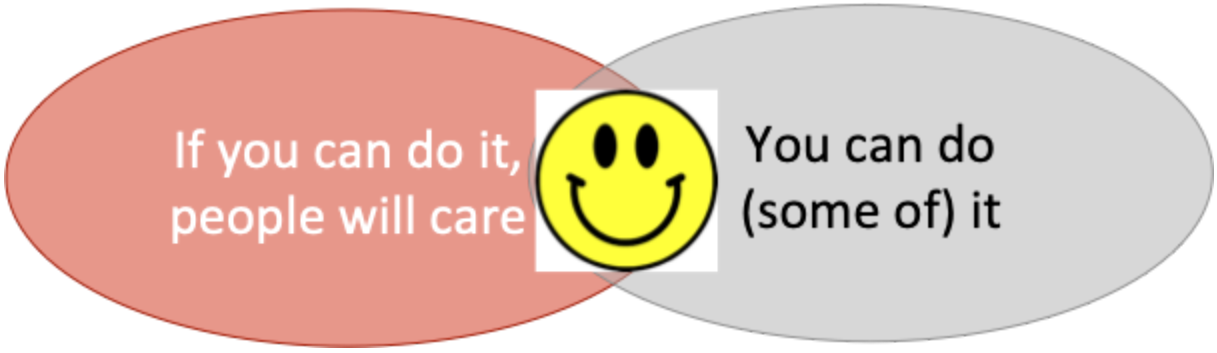

Figure 3: Pick Problems at the Smile

In our experience, one should devote considerable thinking and time to picking problems to work on. One should not play Jeopardy, which is an American TV game show in which contestants are shown the answer and challenged to develop the question. You may laugh, but we have seen this in research: "I have a cool mechanism, so let me figure out what it is good for." In our experience, this approach rarely produces groundbreaking and lasting research. Ask first, "What problem am I trying to solve and why?"

Operationally, we recommend looking for change. You can do good work without change, but you have to be smarter and more creative than all that have preceded you, which is hard for important problems. In computer architecture, change arises from (a) software and applications above, e.g., exploding machine learning and augmented reality, (b) technology changes below, e.g., 3D chip stacking and emerging non-volatile memory technologies, and (c) influences from other (sub-)fields, e.g., miraculous progress in SAT solvers and using ML to optimize hardware.

For a concrete example, consider the work of David Patterson, Garth Gibson, and Randy Katz on redundant arrays of inexpensive disks or RAID [P88] that I observed as a graduate student. You might think that the most creative part of the work was taking erasure codes, applying them to blocks, and rotating parity to avoid bottlenecks. Although this was creative, in our opinion, the greatest creativity in the project came first in recognizing the problem. Historically, the most cost-effective way to store large data was on large washing-machine-sized disks. When personal computers exploded into the world, they soon adopted small disks, and the sales volume made these disks the most cost-effective place to store data. The problem: Can we use small PC disks to store large data? The problem within the problem: Without innovation, an array of PC disks is too unreliable. A single PC disk is not that reliable (the market wanted inexpensive), and an array of these disks will lose data in days. With this setup, the innovation of RAID now seems creative but not super-human. The hard thinking to pick a good problem was well rewarded with a seminal paper with over 4,000 citations and three test-of-time awards.

We conclude this section with four other comments regarding picking problems. First, spend considerable time and energy on picking the problem. Avoid jumping to solutions too fast, as you might solve the wrong problem. Second, seek simple ideas, especially for interfaces. In computer architecture and systems, even simple ideas get more complex when actually deployed. Be proud of simple ideas, as I tout with a web page.[5] Woe to those in industry who have to deploy something that was already complex in the academic paper. Third, collaborate broadly with professional colleagues and students. It is not clear I ever unilaterally developed an idea that did not benefit from interactions with one or more of my 160 coauthors. ***Don't unduly worry about dividing credit, as credit often multiplies and collaboration usually enables something worthy of greater credit.*** Fourth, keep academic-industry connections strong, as computer architecture and systems are about influence, not intrinsic beauty. Impact and effort seem to vary proportionally from the smaller--talking to people at conferences and holding industrial affiliate meetings--to the larger--student internships and sabbaticals in industry (for me at Sun, AMD, and Google).

[5] http://pages.cs.wisc.edu/~markhill/includes/simple.html

Thanks and Giving Forward

I have been blessed to work with many great people--beginning with my Ph.D. co-advisors David Patterson and Alan Jay Smith--and continuing with 160 co-authors, with the biggest word cloud font sizes in Figure 1 going to David Wood, Daniel Sorin, Michael Swift, James Larus, and Milo Martin. This work has occurred at three great public universities--Michigan for undergraduate, Berkeley for graduate, and Wisconsin where I am faculty--with funding largely from the U.S. National Science Foundation, most recently with grants CCF-161782 and CNS-1815656. As Newton said, "We can see further, because we stand on the shoulders of giants." For my students and me, we have discovered that these giants go back at least as far as 990 AD in Constantinople.[6]

For this, one cannot give back, but one can--and should--give forward. I have sought to give forward through more than thirty years of teaching, three years as department chair, and through service in several ways, including with ACM SIGARCH and, mostly-recently, the Computing Community Consortium (CCC).[7] Your opportunities and predilections may be different, but please give forward to honor those who gave to you.

I conclude by thanking my family: wife Sue, children Nicole and Greg, granddaughter Zeynep, sister Kathryn, and parents Toivo and Maria. My strong mother Maria passed away the morning after I received the 2019 Eckert-Mauchly award. She learned of it a few weeks before, valued it, but cared more that her children sought to live with integrity and to follow the golden rule. May Maria rest in peace.

References

[B11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D A. Wood, "The gem5 simulator," ACM SIGARCH Computer Architecture News, vol. 39, no. 2,pp. 1–7, 2011.

[HR19] M. D. Hill and V. Janapa Reddi, "Gables: A roofline model for mobile socs," in IEEE 25th International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2019.

M. D. Hill and V. J. Reddi, "Accelerator-level parallelism," 2019, ArXiv: https://arxiv.org/abs/1907.02064v1

[H87] M. D. Hill, Aspects of Cache Memory and Instruction Buffer Performance. PhD thesis, University of California, Berkeley, November 1987, http://www.cs.wisc.edu/multifacet/theses/mark_hill_phd.pdf

[HS89] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches,"IEEE Transactions on Computers, vol. 38, no. 12, pp. 1612–1630, 1989.

[J90] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by theAddition of a Small Fully-Associative Cache and Prefetch Buffers,"ACM SIGARCH Computer Architecture News, vol. 18, no. 2SI, pp. 364–373, 1990.

[M05] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," SIGARCH Comput. Archit. News, vol. 33, pp. 92–99, Nov. 2005.

[M06] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: Log-based transactional memory," in The Twelfth International Symposium on High-Performance Computer Architecture, 2006., pp. 254–265, IEEE, 2006.

[6] http://pages.cs.wisc.edu/~markhill/Academic/Genealogy_Hill_Mark.pdf
[7] https://cra.org/ccc/

[P64] J. R. Platt, “Strong inference,”Science, vol. 146, no. 3642, pp. 347–353,1964.

[P88] D. A. Patterson, G. Gibson, and R. H. Katz, A case for redundant arrays of inexpensive disks (RAID), vol. 17. ACM, 1988.

[R93] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, “The wisconsin wind tunnel: Virtual prototyping of parallel computers,” in Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, SIGMETRICS’93, (New York, NY, USA), pp. 48–60, ACM, 1993.

[Y07] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M.Swift, and D. A. Wood, “Logtm-se: Decoupling hardware transactional memory from caches,” in 2007 IEEE 13th International Symposium on High Performance Computer Architecture, pp. 261–272, IEEE, 2007.