Catalyzing Computing Podcast Episode 35 – Computer Architecture with Mark D. Hill (Part 1)

The transcript below is lightly edited for readability. Listen to “Computer Architecture with Mark D. Hill (Part 1)” here.

[Intro - 00:10]

Khari: Hello, I'm your host, Khari Douglas, and welcome to Catalyzing Computing, the official podcast of the Computing Community Consortium. The Computing Community Consortium, or CCC for short, is a programmatic committee of the Computing Research Association. The mission of the CCC is to catalyze the computing research community and enable the pursuit of innovative, high-impact research.

In this episode I interview Dr. Mark D. Hill, a Professor Emeritus of Computer Sciences at the University of Wisconsin-Madison. Mark recently joined Microsoft as a Partner Hardware Architect. His research interests include parallel computer system design, memory system design, computer simulation, deterministic replay and transactional memory. He is the Chair Emeritus of the CCC Council. In

this episode Mark discusses the importance of computer architecture, the "3C model of cache behavior", and overcoming the end of Moore's law. Enjoy.

[What is Computer Architecture? - 1:01]

Khari: So we're here today with Mark Hill, former chair of the Computing Community Consortium and now retired from the University of Wisconsin-Madison, moving to Microsoft.

How are you doing today?

Mark: I'm doing very good. It's a pleasure to be here with you, even though here is cyberspace.

[Laughter]

Khari: So could you tell me a little bit about yourself? What is your background and how did you get involved with computer science?

Mark: So I grew up in Detroit. Neither of my parents had bachelor's degrees, but they made it clear to my sister and I that we had to go to college. I never questioned that and I kind of liked math and science. I thought math was not a good way to earn a living and as a lower middle class kid, earning a living was very important to me; so I picked engineering, and computing was particularly fascinating.

I had early success in ninth grade with a science fair project. I actually built a mechanical adder using numbers that you might put on your door to give the address of your house [see the photo here]. So, computing fascinated me, I guess, as I went further along, because when I told a computer to do something it did exactly that and no one else in my life did. In fact, it did exactly that even when I told it to do the wrong thing, it faithfully did the wrong thing. But the process of then trying to make it right, called debugging, is kind of like detective work, which I actually liked as well.

Computing has worked out really well. Early on — you know, this was in the 1970s when I was in high school — most people didn't really know what computers were because the personal computer hadn't come out yet. Computers were these mysterious things that you saw on television.

Khari: One of your primary research interests is computer architecture. So for people who don't know what is computer architecture?

Mark: So computer architecture is the big picture of computer hardware. Computer hardware has many complicated things and somebody has to deal with the big picture. The name architect comes from the analogy of a building architect who also has to handle the big picture of the building, even though others may be greater experts in the plumbing or the electrical system. In both cases, you have goals that you want to maximize, let's call it “performance,” and you have to do that within cost constraints, physical constraints, and standards and things like that. Buildings have to meet certain electrical standards and computers have to meet certain communication standards.

Computer architecture is low down in the computing stack, so it's not very visible to society. I would liken it to the foundation of a skyscraper. That's a very essential thing, and if you want to build a taller building you need to build a better foundation, but it's not something that people tend to notice. We computer architects have been all about taking what technology gives us with Moore's Law and more transistors and turning it into better and faster computers.

By the way, another question you could ask me is why did I choose computer architecture within computer science? And my answer to that is I just loved how we computer architects get to cheat. We get to make things that are better than the pieces that we make them out of. I deal with caches — which we will maybe come up later — which makes the memory appear faster than it is, and other people do processors that make it appear faster than they would be if you just use the underlying technology in a straightforward fashion. I found that fascinating.

Khari: Has most of your work been with hardware or the exchange between hardware and software? Some combination?

Mark: Much of the work has been hardware, but it deals also with low-level software. But even when you do hardware, there are theories going back to Alan Turing that all hardware can compute the same thing, so the difference is how well and efficiently it does. You need to pay attention to software and what the software is doing in order to do good hardware.

[Three C Model of Cache Behavior - 5:40]

Khari: So you're credited as "the inventor of the widely used 'three C model'" of cache behavior. So what is a cache and what is the 3C model?

Mark: So a cache is a small, transparent buffer that holds the contents of a larger, slower memory. By transparent I mean that the user of the cache of larger memory just thinks they're getting a fast memory. They don't really see the difference between the cache and the slower memory. The cache just sort of "automagically" has things which the user tends to ask, making the whole thing go faster. Cache comes from a word like, a pirate may have a cache of buried treasure somewhere. It's hidden and valuable.

Khari: Ok. So what is the 3C model of cache behavior?

Mark: So when you make an access to this cache-memory combination, if it's found in the cache it's called a hit and if it's not found it's called a miss, kind of like baseball. What the three C's was trying to do was to get some insights into these misses because they're expensive.

Some misses you just have to do because you've never referenced that before — called "compulsory misses" in the model. Some misses happen because the cache can only be so big and still be fast — those are called "capacity misses" if you exceed the

capacity. Finally, to be able to look up things and find things faster, caches get divided into smaller pieces called sets. If your set overflows and there's too many things there, that's called a “conflict miss.”

By the way, I actually spent some time with a thesaurus to come up with an alliteration, everything starting with C, and that made it much more memorable. And people have found this to be quite intuitive to figure out what is going on in it. So it was a nice result in my PhD dissertation.

Khari: Ok. So what happens in a computer when a cache misses.

Mark: When a cache misses it then goes to the larger memory to obtain the data that it didn't have and brings that data into the cache. If the cache is full, it also has to evict some other data. You're betting that the newly referenced data is more likely to be a good thing to be in the cache than the old data, because programs tend to reference what they've more recently referenced.

Khari: So if there's a miss, then it's slower for the computer to access the data.

Mark: Exactly. And this is a big deal because without caches — and we actually do caches on caches on caches — computers would be a hundred or more times slower. So your smartphone would not really be fast enough to do anything that you love without those caches.

[Laughter]

Khari: So I saw one presentation where you're talking about the 3C model and you said it was “wrong.” Could you expand on that? What do you mean by it being wrong?

Mark: So the hand wave of the model says that all misses fall in these three categories, compulsory, capacity, and conflict, but the reality is actually more complicated, because

when you divide the cache into these smaller pieces called sets you could, for example, have a pattern in which the sets actually are a good thing and do better than a fully flexible cache that could put things anywhere. These would be like anti-conflict misses and that's not counted in the model. That's one example.

The thing to remember with a model: a model is always a simplified version of a complex thing. George Box, the great statistician, said essentially “all models are wrong, but some are useful.” So even though the model is wrong relative to the full detail, it's useful because it provides very good intuition. People have told me.

[Log-based Transactional Memory - 9:58]

Khari: So you're also part of a team that created a transactional memory system called log-based transactional memory or LogTM. So what is that?

Mark: Right. So, in the old days computers always had one processing core that did all the work and we moved to systems that have multiple processing cores, and software has to sort of coordinate how fast one processor core is going versus another and that coordination can be hard. One thing that helps is if you could just say, “Hey, I want to do this unit of work atomically. You know, get my act together, get it finished, and then go forward.” That's called a transaction. So LogTM was a system to do transactions to ease programming. It's so named because it uses something called a log.

Khari: What are the implications of using the LogTM system as opposed to other established systems that existed?

Mark: Well, the benefit is that it can potentially ease programming relative to using something called locks, which are much lower-level. We won’t go into the details, but it does add some complexity to the hardware. So far there has been some limited implementation of more restrictive transactional memory systems, but, sort of, not broad success.

LogTM is even more complicated, so it hasn't been done, but it inspired a lot of thinking; and what we see in computer architecture often is ideas sometimes gestate for many decades before they actually go somewhere. So I'm still optimistic.

Khari: Are there any historical architectures of note that you could discuss that, sort of, came out, didn't have much impact, and then down the line played a big role in changing how computing is done?

Mark: Sure. One of the things that's critically important right now to supporting deep neural networks or AI are general purpose graphics processing units (GPUs) and within them they do something called data-level parallelism where a single operation implies multiple manipulations. For example, one instruction — I should say, instruction not operation — one instruction might take 64 numbers and add it to 64 corresponding numbers for 64 results.
Ok. This idea has gone by many names and was first proposed in the 1960s with systems like the Illinois ILLIAC IV and had some niche successes like in Cray supercomputers, but just kind of hung around as a specialized thing for like four decades before it exploded into being a gigantic success in graphics processing units.

Khari: Wow. So in one of the presentations I watched of you talking about the LogTM system, you had this four quadrant chart with deferred and eager conflict detection as, sort of, two potential settings, like on the left hand side, and on the top of this box you had deferred and eager version management as two other alternatives.

Could you talk about those things like what are deferred and eager conflict detection and version management?

Mark: Ok. Well, so first of all, when you do transactions it's possible that there is a concurrent transaction that is incompatible with it. If everything's ok you commit, and if there's a problem you might have to abort one of the transactions. Think of a transaction which is doing a move, like you're taking something from one data structure to another,

if it commits you both remove it from the first data structure and add to the second, if it aborts you do nothing, but you never are partially done, like you've removed it and then it just never appeared anywhere.

Khari: Mhmm.

Mark: So what a transactional memory system has to do is first detect whether there's any conflicts requiring an abort, and they can do so as the transactions execute, that's called" eager," or it can do it at the end of the transaction when it's just sort of double checking if everything's ok, and that's called "lazy conflict detection."

Orthogonal to that, which two by two gives you the fourth quadrant, is lazy version management versus eager version management. So, what lazy management is about is that within a transaction you sometimes overwrite an old piece of data, and when you're all done which do you need: your old piece or the new piece?

Well, you need the new piece if you commit and you need the old data if you abort. So you need both, but only one can sort of be in the final resting place, the other has to be in like a buffer on the side, a temporary scratch place. Eager version management says, "Hey, I think things are good. I'm going to put the old value on the side and put the new value in place." That way when you commit everything's fast, whereas lazy version management says, "Well, I'm not sure I'm going to use the new value, so I'll leave the old value there until I know the new value is good." LogTM was one of the first to really do a very rich eager version management system in a transactional memory system.

Khari: Ok. Have there been more eager version management systems that have come out since then?

Mark: Uhh, no it hasn't really caught on that much. Most of the systems that have happened have been very limited. They don't allow big transactions, so it was easier to just do some temporary buffering of the new values.

But at another level in database management systems, which also do transactions — in fact, transactions long before transactional memory — they do the equivalent of eager version management in many cases.

[End of Moore's Law - 16:36]

Khari: Ok. You sort of mentioned this earlier, but I want to bring this up: what is Moore's Law and what does it's end mean for computing and for society broadly?

Mark: So there's actually two Moore's Laws. The real Moore's Law was Gordon Moore's observation in, I think it was, 1965 that the optimal number of transistors per chip is going to double every 18 months or two years. The reason for this was that if we got better at making the transistors we could do a bigger chip, and if we could do bigger chips a system might be able to be built with fewer chips; so at any given time, how big a chip should be there would be a happy point in the middle. That happy point would keep doubling every two years, he predicted and that prediction was right for more than three decades. Four decades really. But it's slowing down now because, even though we can sometimes double the number of transistors, we can't completely use them for various energy reasons and their cost is not going down like it used to. So Moore's Law is dramatically slowing down.

The implication of that is that Moore's Law, coupled with computer architecture, heretofore made computers much faster at the same cost, and that allowed software people to both run older programs much faster without any changes, without thinking about the hardware and, more importantly, to dream up new things that ran like pigs on the old hardware, but on the faster hardware would run well, so they could go bigger and bigger and more ambitious. That was a great thing.

Now that Moore's Law is slowing greatly down, if we want to do more ambitious things like this artificial intelligence we want to do, we can't just do it by waiting and hoping for the hardware to get better. We have to figure out other ways, which is going to require a lot of creativity in software and a lot of creativity across the hardware-software boundary

Khari: Hmm. So what are some possible ways that you've seen people propose to overcome this end of Moore's Law?

Mark: Well, the way that is always hoped for is that somehow there will be a new Moore's Law or there will be a new type of transistor, or other kind of switch, which resumes the process, and that has happened in the past. Unfortunately, we don't have a good candidate for the future right now.

A second thing that can help somewhat is that currently chips are relatively two-dimensional. They're like flat plains, like northern Wisconsin not too many high-rise buildings; and we can start building high-rise chips, which, if we can get the heat out, can improve performance by communicating faster. Just like people in Manhattan don't have to travel great distances to encounter other people. But there is a limit to that.

I think then the next thing we need to do is we take software, which has previously been implemented in many layers using abstractions or approximations between the layers to help manage complexity, and sort of dive down and cross-optimized to get the fat out. Finally, I see a lot of promise in specialized accelerators, which target specific types of computation to be extra efficient at. Like instead of being a general-purpose processor that can compute anything you might be an accelerator that's good at video decoding.

[Hardware accelerators - 20:40]

Khari: Ok, so could you explain what an accelerator is and how that works?

Mark: As I said, an accelerator is a specialized hardware block that can't do everything but does some computation particularly well. It's named an "accelerator" because it's faster, in many cases it's only somewhat faster. But it can really do the computation with ten or one hundred times less energy, and energy matters. Obviously, it matters on your smartphone because your battery doesn't run out, but it also matters in data centers

where they use a tremendous amount of power and if they can save power that's a good thing.

How does it work? Well, I think the right thing to say is that the beauty of a general purpose processor is that it can do any computation to be named later. But for that generality you get a certain amount of overhead. Maybe you can liken it to a situation where, you know, there's a lot of management that allows great flexibility to go in many directions, whereas another organization might have a very flat system where there's just one person telling everybody what to do, in which case you just can't do everything that way, you can only do the specific things people were previously trained to do. That's kind of the way an accelerator works.

Khari: Ok, so I know you've done a lot of work with accelerator-level parallelism. Could you talk a little bit about that work and maybe some of the other kinds of parallelism that exist within computing architecture? I know you mentioned data-level parallelism already.

Mark: So accelerator parallelism is a term that Vijay Janapa Reddy and I coined. It's not generally accepted, but we were observing the fact that smartphones had dozens of accelerators and in any given use case, like recording or decoding video you might be using a handful of accelerators and you use them in parallel.

This use of transistors in parallel has a long history, because when you get more transistors and they are also faster transistors, how do you go even faster? Well, the only way to go even faster is to use the additional transistors in parallel with the other one, so they're getting it done faster. This is what we've been doing since the beginning, but the problem is that as you go from thousands of transistors to ten thousands, to hundred thousands, to millions, and billions of transistors, the same ways of using them in parallel don't work anymore.

Kind of like if you are told as a student, “Congratulations, your income doubles,” you say, “Oh, great. I get to buy some small thing.” Many maybe doublings later you're going to have to figure out how to buy yachts.

[Laughter]

That would be a good problem, but in computer architecture we have that problem where we've had this many times doubling. Initially we did something called bit-level parallelism, where you just manipulated bigger numbers or you like did the multiply in fewer steps then with a smaller number of transistors.

Then we did instruction-level parallelism, where a processor core logically executes one instruction after another, but we cheat — this is one of the examples of cheating — and we execute many at the same time, like recent processors from Intel and others might have 100 instructions in partial execution and still look serially. That's instruction-level parallelism.

Thread-level parallelism is when we say, “Oh, we can't make the process go that much faster, let's use multiple processors and let software have to coordinate the multiple processors.” They could coordinate with Log TM, for example. Data-level parallelism, as I mentioned, has now manifested with great success in GPUs, but goes back to vectors and SIMD extensions and things like that. That's where you have a single instruction that causes many operations to happen.

We think accelerator parallelism is an example of what's going to happen next. It's clearly happened in smartphones and I think it's going to happen to make, for example, self-driving vehicles much more efficient, and using cloud services where many of the services are specialized, like your processing video and things like that.

Khari: Could you discuss a common accelerator use case?

Mark: For example, if you are using your smartphone to record video, there is a camera sensor that is producing a stream of information — and we will focus on the video, not the audio although the audio is done as well. You need to go through an image signal processor, might have to go off chip to be buffered in DRAM or memory, and then it can be processed by the GPU, maybe again to the instruction processor, the GPU, etc. In the end it can go out to the display so that you can see it and it also might be stored in non-volatile memory or flash so that you can save it for later.

That's something that is an interesting thing because you're not trying to make it as fast as possible, you're trying to make it fast enough to record the video and then you're trying to do it with as little energy as possible to both save the battery and avoid the phone getting too hot. The way that phones work is that you have a number of these use cases, maybe 20, that are important, and you have to make sure that each one works sufficiently well. And I think many computer systems are going to move toward this kind of model.

Khari: Ok. Are there any major differences between what's on a cell phone or another mobile device and a computer in terms of how the computer architecture and the hardware works?

Mark: Well Khari, that's actually a great question. In 2018, I did a sabbatical at Google with a group looking at silicon for these kinds of things, and I thought, "What do I know about these kinds of things? Because I have never operated in the space, I've only operated with big computers." But I had operated with big computers for almost 30 years at that time and these devices have grown up to be real computers, so they have all the standard components of compute, interconnect, memory, and storage, but they have them in different proportions and they had this abundance of accelerators, which classical computers have not had at this point, but I think will going forward. There was a recent Apple iPhone that had 42 accelerators [see a photo of that chip here].

Khari: So prior to the iPhone with the 42 accelerators, what was a typical number for a phone to have?

Mark: Typical is hard to say because they were changing pretty fast, but if you look at a series of Apple iPhones from 2010 to 2016, the number of accelerators went from nine, 12, 17, 20 to 29, 31, 36. You can see it's climbing fast.

Khari: Wow. So are most of these accelerators being used for things like the video processing you gave in your example or is it a wider variety of functions?

Mark: It's a wider variety of functions so that in any given use case, like recording video, you are not using 42 accelerators, you're using a relatively small number, like five or seven. But presumably these were all put on there because there's some important use case that needs them.

One of the things that I think we need to evolve is some of this stuff was put on there, and maybe when you put it on the chip you don't know quite what the phone is going to be used for when it comes out, so you might put things on unnecessarily. I think we need a better science for designing chips with multiple accelerators and accelerator-level parallelism. We tried to help a little bit with a model called Gables.

[The Gables Model - 29:41]

Khari: Could you discuss the Gables model? What is that?

Mark: Ok. So when you have a chip with 42 accelerators, how do you decide which ones to have, which ones to select, how to size them, and things like that? It's just very complicated. It would be nice to have a simplified picture to get a first answer, not a final answer. Gables was our attempt to do that and it builds on something called Roofline.

The Roofline model was for a homogeneous multicore chip, which is a chip that every processor core is the same, and it modeled the chip with a peak computation performance and a peak off-chip communication bandwidth in a plot that looks like a roofline. So what Gables did is it said, "We can't model our 42 accelerator's that way,

but we can perhaps do a roofline for each accelerator” — because each accelerator is different, they're heterogeneous — “and then find a way to combine them to model the whole chip.” That's what Gables does, and Gables gets its name by a roof that has many rooflines.

Khari: Ok. So have you used this model on any products or technologies that have shown promise?

Mark: I have not used it on products. However, I'm aware of at least two instances, one where it was incorporated in the tool chain of a major tool provider — I'm not sure I’m allowed to say the name — and I know it was used for the preliminary development of some products at a major IT company. And they're still using it, but I think that's confidential.

But I will say it, I think it's better than the community thinks it is at this point, because it's not like, you know, super popular.

[Laughter]

Khari: Yeah, maybe now people will think more about it. Anything else you want to say about the Gables model?

Mark: Um, I'll just say this: I think it's important when you encounter a complicated system to try to figure out a way to get your head around it, and Gables was our attempt to do that. I think that model has value even if you don't believe the numbers, because it gives you a way to frame your thinking about the system and pay attention to these communication and these performance things and how they interact. That's valuable even if you don't believe the output of the number.

[Three Other Models of Computer System Performance - 32:24]

Khari: Yeah, that makes a lot of sense. So talking about models, I saw you wrote a paper in 2018 called *Three Other Models of Computer System Performance*, and in that you argue for the use of more simple models beyond Amdahl's law, such as bottleneck analysis, Little's Law and M/M/1 Queue.

So could you talk about that paper? What was the argument you were making there?

Mark: Right. So, I have found that computer architects are too quick to go to simulation. Simulation is where you write a computer program that mimics the system that you're studying, potentially very well, but really very slowly. In fact, some of our most cited papers are simulation. I think you can also complement this with these simple models that are maybe less accurate but give you a lot of insight.

Amdahl's law is an example of that. It's fairly well known that you have a part of a computation that you're speeding up and a part that you're not, and Amdahl's law shows you that that part that you're not speeding up really limits what you can do unless it's vanishingly small.

These other three models I find equally helpful for getting these first answers, not the final answers. Bottleneck analysis looks at, sort of, information flow. Think of a computer system as a whole bunch of pipes, and you're trying to figure out how fast the water can flow. It turns out that Roofline and Gables are actually just specific instances of bottleneck analysis.

Little's law is broadly applicable, it relates the number of items in a system to the product of the average time an item stays in the system times the average rate that items come in. That's very abstract, but, for example, if there are four hundred current

students and one hundred students arrive every year then the average student stays four years. And what's powerful about Little's law is it relates those three numbers, and if you can easily figure out two of them you can use Little's law to find the other. David Wood and I used to do a lot of consulting, and we joke that we just kept applying Little's law to systems to fix performance bugs

Finally, an M/M/1 Queue has to deal with the fact that when things arrive at random, whether they're some request in a computer system or people arriving at a store at random, is not exactly equal. One way to model randomness is something called Markovian, which basically says when I arrive at the store is independent of when anyone else decided to arrive at the store. And what an M/M/1 Queue says is if people are coming in at random — Markovian, that's the first M — and they're serviced in about a random amount of time — that's the second M — when we service one at a time, it shows you how well you can...how long people are going to have to wait depending on how busy things are. It shows, for example, that you can't make a system that is always busy and has very low waiting times, it's a severe trade off, under the assumption of these random arrivals. That's why doctors, for example, don't allow random arrivals. They make you schedule appointments. But this is a very powerful law to think about. It is an example of a rich field called queuing theory.

[Outro - 36:15]

Khari: That's it for this episode of the podcast. We'll be back next week for part two of my interview with Mark Hill. In that episode, Mark discusses the importance of hardware security, the impact of AI on hardware and working in academia versus industry. Until then, remember to like, subscribe, and rate us five stars wherever you get your podcast.

Learn more about the work of the CCC on our website at cra.org/ccc and find us on social media to stay up to date on all our latest activities. Until next time, peace.