
SINGLE-THREADED VS. MULTITHREADED: WHERE SHOULD WE FOCUS?

Joel Emer

Intel

TO CONTINUE TO OFFER IMPROVEMENTS IN APPLICATION PERFORMANCE, SHOULD COMPUTER ARCHITECTURE RESEARCHERS AND CHIP MANUFACTURERS FOCUS ON IMPROVING SINGLE-THREADED OR MULTITHREADED PERFORMANCE? THIS PANEL, FROM THE 2007 WORKSHOP ON COMPUTER ARCHITECTURE RESEARCH DIRECTIONS, DISCUSSES THE RELEVANT ISSUES.

Mark D. Hill

University of Wisconsin

—Madison

Yale N. Patt

University of Texas at

Austin

Joshua J. Yi

Freescale Semiconductor

Derek Chiou

University of Texas at

Austin

Resit Sendag

University of Rhode

Island

Moderator's introduction: Joel Emer

..... Today, with the increasing popularity of multicore processors, one approach to optimizing the processor's performance is to reduce the execution times of individual applications running on each core by designing and implementing more powerful cores. Another approach, which is the polar opposite of the first, optimizes the processor's performance by running a larger number of applications (or threads of a single application) on a correspondingly larger number of cores, albeit simpler ones. The difference between these two approaches is that the former focuses on reducing the latency of individual applications or threads (it optimizes the processor's *single-threaded* performance), whereas the latter focuses on reducing the latency of the applications' threads taken as a group (it optimizes the processor's *multithreaded* performance).

The obvious advantage of the single-threaded approach is that it minimizes the execution times of individual applications (especially preexisting or legacy applications)—but potentially at the cost of longer

design and verification times and lower power efficiency. By contrast, although the multithreaded approach may be easier to design and have higher power efficiency, its utility is limited to specific—highly parallel—applications; it is difficult to program for other, less-parallel applications.

Of course the multiprocessor versus uniprocessor controversy is not a new issue. My earliest recollection of it is from back in the 1970s, when Intel introduced the 4004, the first microprocessor. I remember almost immediately seeing proposals that said that if we could just hook together a hundred or a thousand of these, we would have the best computer in the world, and it would solve all of our problems. I suspect that no one remembers the result of that research, as the long series of increasingly more powerful microprocessors is a testament to our success at (and the value of) improving uniprocessor performance. Yet, with each successive generation—the 4040, the 8080, all the way up to the present—we've seen exactly the same sort of proposal for ganging together lots of the latest-generation uniprocessors.

So the question for these panelists is, why is today's generation different from any of the past generations? Clearly, it is not just that we cannot achieve gains in instructions per cycle (IPC) in direct proportion to the number of transistors used, because that's never been true. If it had been true, we would have IPCs several orders of magnitude larger than those we have today. Our recent rule of thumb has been that processor performance improves as the square root of the number of transistors, and cache-miss rates likewise improve as the square root of the cache size. Yet, despite these sublinear architectural improvements, until recently, uniprocessors have been the preferred trajectory. Why is the uniprocessor trajectory insufficient for today? Are there no ideas that will bring even that sublinear architectural performance gain? Why aren't the order-of-magnitude gains being promised by single-instruction, multiple-data (SIMD), vector, and streaming processors of interest? Is the complexity of current architectures a factor in the inability to push across the next architectural performance step? Or is there a feeling that, irrespective of architecture, we've crossed a complexity threshold beyond which we can't build better processors in a timely fashion?

Looking at multiprocessors, is using full (but simpler) processors as the building blocks the right granularity? Are multiprocessors really such a panacea of simplicity? How much complexity is going to be introduced in the interprocessor interconnect, in the shared cache hierarchy, and in processor support for mechanisms like transactional memory? Much of the challenge of past generations has been coping with the increasing disparity between processor speed and memory speed, or the limits of die-to-memory bandwidth. Does having a multiprocessor with its multiple contexts make this problem worse instead of better? And, even if multiprocessors really are a simpler alternative, what is the application domain over which they will provide a benefit? Will enough people be able to program them?

As I hope is clear, both approaches have advantages and disadvantages. It is, however, unclear which approach will be more

About this article

Derek Chiou, Resit Sendag, and Josh Yi conceived and organized the 2007 CARD Workshop and transcribed, organized, and edited this article based on the panel discussion. Video and audio of this and the other panels can be found at <http://www.ele.uri.edu/CARD/>.

heavily used in the future, and should be the major focus of our research. The goal of this panel was to discuss the merits of each approach and the trade-offs between the two.

The case for single-threaded optimization: Yale N. Patt

The purpose of a chip is to optimally execute the user's desired single application. In other words, the reason for designing an expensive chip, rather than a network of simple chips, is to speed up the execution of individual programs. To the user, optimally executing a desired application means minimizing its execution time. Amdahl's law says the maximum speedup in the execution time is limited by the time it takes to execute the slowest component of the program. Therefore, when we parallelize an application to run multiple threads across multiple cores, the maximum speedup is limited by the time that it takes to execute the serial part of the program, or the one that needs to run on a powerful single core.

For example, suppose a computer architect designs a processor for a specific operating system that needs to periodically execute an old VAX instruction. In the extreme, assume that this instruction shows up only once every two weeks. In that case, because that instruction is not frequently executed, the computer architect does not allocate much hardware to execute that instruction. In the worst case, if it takes the processor three weeks to execute the instruction, then the program will never finish executing, because the processor did not run the slowest part of the program fast enough. Although this example is clearly exaggerated, it motivates the need to continue to focus on improving the single-threaded performance of processors.

One potential objection to continuing to focus heavily on single-threaded microarchitectural research is that there is insufficient

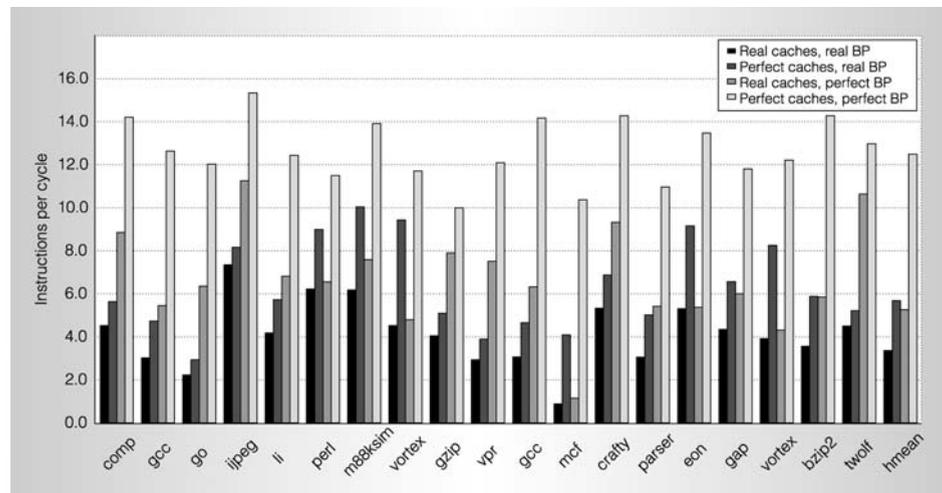


Figure 1. Potential of improving caches and branch prediction.

parallelism at the instruction level to significantly improve processor performance, whereas there may be significantly more parallelism at the thread level that can be exploited instead. However, difficult problems still need to be solved, and the computer architecture community will not solve them if the best and brightest are steered away from those problems. Creativity and ingenuity can solve those problems. For example, in the RISC vs. CISC debate, CISC did not win merely because of legacy code. Rather, CISC won the debate because computer architects had the ingenuity to develop solutions such as out-of-order execution combined with in-order retirement, wide issue, better branch prediction, and so on. For example, Figure 1 shows the headroom remaining for a 16-wide issue processor that uses the best branch predictor and prefetcher available at the time the measurements were made.¹

Take, for example, the creativity of computer architects with respect to branch prediction. Conventional wisdom in 1990 pretty well accepted as fact that instruction-level parallelism on integer codes would never be greater than 1.85 IPC. However, even simple two-level branch predictors could improve the processor's performance beyond 1.85 IPC. And, today, computer architects such as Andre Seznec and Daniel Jimenez have proposed more sophisticated branch predictors that are better than the

two-level branch predictor. There is still quite a bit more unrealized performance to be gotten from better branch prediction and better memory hierarchies (see Figure 1.) The key point is that the computer architecture community should not avoid the difficult problems, but rather should harness its members' creativity and ingenuity to propose solutions and solve those problems.

Another potential objection to the single-threaded focus is power consumption. Power consumption in a 10-billion-transistor processor is a significant problem. However, in a processor with 10 billion transistors, the processor could have several large functional units that remain powered off when not in use, but that are powered up via compiled code when necessary to carry out a needed piece of work specified by the algorithm. I use my "Refrigerator" analogy to explain this type of microarchitecture. William "Refrigerator" Perry was a huge defensive tackle for the 1985 Chicago Bears. While the Bears were on offense, Perry sat on the bench until they were on the one yard line. Then, they would power him up to carry the ball for a touchdown. To me, the key question is, "How much functionality can we put into a processor that remains powered off until it is needed?"

I think the processor of the future will be what I call *Niagara X*, *Pentium Y*. The

Niagara X part of the processor consists of many very lightweight cores for processing the embarrassingly parallel part of an algorithm. The Pentium Y part consists of very few, heavyweight cores—or perhaps only one—with serious hybrid branch prediction, out-of-order execution, and so on, to handle the serial part of an algorithm to mitigate the effects of Amdahl's law. To ensure that the processor's performance does not suffer because of intrachip communication, a high-performance interconnect will need to connect the Pentium Y core to the Niagara X cores. Additionally, computer architects need to determine how transistors can minimize the performance implications of off-chip communication.

The case for multithreaded optimization: Mark D. Hill

For decades, increasing transistor budgets have allowed computer architects to improve processor performance by exploiting bit-level parallelism, instruction-level parallelism, and memory hierarchies. However, several “walls” will steer computer architects away from continuing to design increasingly more powerful single-threaded processors and toward designing less complex, multi-threaded processors.

The first wall is power. Some current-generation processors require large, even absurd, amounts of power. For these processors—and to a lesser degree, other lower-power processors—high power consumption makes it more difficult and expensive to cool the processor or forces the processor to operate at a higher temperature, which can lower the processor's lifetime reliability. In both cases, higher power consumption either increases the operational cost of owning the processor, by increasing the costs of electricity and server room cooling, or decreases the battery life. The current situation is very similar to the one mainframe developers faced about 10 to 15 years ago with emitter-coupled logic. Although mainframe developers were able to switch to an alternative technology that was initially slower but eventually proved better—namely CMOS—the computer architecture community does not currently have a viable alternative technology that we can switch to. Thus, we

must adopt a different approach to designing our processors.

The second wall is complexity. The basic issue with the complexity wall is that it is becoming too difficult, and taking too long, to design and verify next-generation processors, and to manufacture them at sufficiently high yield rates. Although some computer architects think that this is a significant wall, I do not, since I believe that creative people like Joel Emer and Yale Patt can create abstractions to manage the complexity.

The third, and final, wall is memory, which I think is a very significant problem. Currently, accessing main memory requires possibly hundreds of core cycles, which can be hundreds of times slower than a floating-point multiply. Consequently, accessing main memory wastes hundreds, or even thousands, of instruction execution opportunities. How much instruction-level or thread-level parallelism is available for the processor to exploit while waiting for a memory access to complete? And, even if there is a large amount of parallelism, the power and complexity walls limit how the computer architect can design a processor to exploit that parallelism.

Given these three walls, the computer architecture community needs to continue to design processors that exploit parallelism if we want to continue to improve processor performance. However, I do not believe that we can continue to exploit instruction-level parallelism with identical operations, that is, SIMD vectors. Rather, we need to exploit a higher-level parallelism in which we can do slightly different things along each parallel path. Note that this higher-level parallelism could be like the thread-level parallelism that has been spectacularly successful in narrow domains such as database management systems, Web servers, and scientific computing, but less successful for other application spaces. More specifically, instead of exploiting bit-level and instruction-level parallelism with wider, more powerful single-threaded processors, we need to use multicore processors to exploit higher-level parallelism.

My approach to this problem is to advocate the development of new hardware

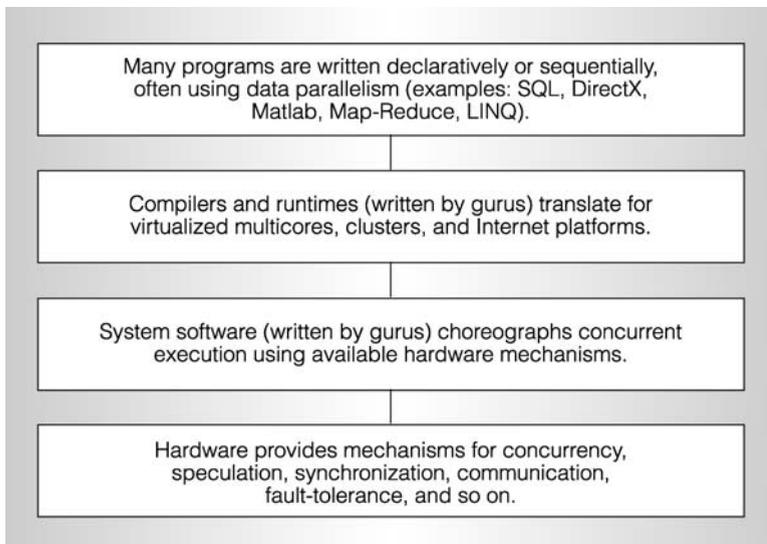


Figure 2. Toward a future hardware/software stack.

and software models (Figure 2). The architecture community needs to raise the level of abstraction at the software layer. Although most people naturally do not think in parallel, there are tremendous success stories, like SQL language for relational databases and Google's MapReduce. In the former case, most people can write declarative queries and let the underlying

system optimize and create great parallelism because of the strong properties of the relational algebra. In the latter case, a limited set of programs allow the user to program even clusters relatively easily. However, these two examples are point solutions only; we need to develop these types of solutions more broadly.

The key at the hardware level is to support heterogeneous operations, and not just strict SIMD. Additionally, the computer architecture community needs to add mechanisms that can improve the performance of software models; examples include transactional memory and speculative parallelism (for example, thread-level speculation). Adding these hardware features helps support the software model, which allows the programmer to continue to think sequentially.

Multicore architecture

Hill: *[removing faux white beard he has been wearing in imitation of Patt; see Figure 3]* Let me take this beard off. It's making me think way too sequentially. So, Yale, is it correct that you think that we should have chips with lots of Niagaras and one beefier core? Are you saying that I've won the debate?



Figure 3. Panelists Yale Patt and Mark Hill and moderator Joel Emer (left to right). Hill removed his beard during the discussion, claiming it caused him to think too sequentially. Patt and Emer declined to remove theirs.

Patt: No.

Hill: You're trying to say that's the uniprocessor argument?

Patt: I'm saying that you need one or more fast uniprocessors on the chip. Without that, Amdahl's law gets in the way. That is, yes, your Niagara thing will run the part of the problem that is embarrassingly parallel. But what about the thing that really needs a serious branch predictor or maybe a trace cache? I have no idea what people are going to come up with in the future. What I *don't* want is for people to get in the "don't-worry-about-it" mind-set.

Hill: Okay, I agree that we need to have multiple cores and that one of those cores should be much more powerful, or alternatively several of those cores should be much more powerful.

Emer: So now Yale has won the debate.

Hill: No, that's multiple cores.

Patt: I'm not suggesting that having multiple cores is not important. What I am suggesting is that we shouldn't forget the part of the problem that requires the heavy-duty single core. You know, you hear a lot of people say, "The time for that has passed." There are so many opportunities to further improve single-core performance. Take the stuff they did at Universitat Politècnica de Catalunya (UPC) with virtual allocate, where, at rename time, they allocate a reorder buffer register but don't actually use it until it is really needed. They save the power from the time they rename it until the time they actually use it: doing things virtually until you really need it, turning off hardware until you really need it... There's a whole lot out there that in fact students in this room will work on, maybe, if they're *not* told, "Forget it; the real action is in multiple cores." The fact of the matter is, *[smiling]* the real action is in Web design.

Hill: So, in your view what is the rate of performance improvement that we can expect to get out of a single core?

Patt: The rate is zero, if we think negatively. Sammy Davis Jr. wrote an autobiography entitled *Yes I Can*. That's what I would like to encourage people to think: "Yes I can"—as opposed to just giving up and saying, "You know, when it's 10 billion transistors, I guess it won't be 20 Pentium 4s; it will be 200 Pentium 4s."

Hill: I guess I should remind you that I absolutely think that we can push uniprocessors forward. We can get improvements, but we're not going to see the improvements that we've seen in the past. And there are markets out there that have grown used to this opium of doubling in performance. That is what we are going to lose going forward.

Emer: How much architectural improvement have we seen?

Patt: We've seen quite a bit. In fact, there was a 2000 Asia and South Pacific Design Automation Conference presentation by a guy at Compaq/DEC [Digital Equipment Corporation] named Bill Herrick, who said that between the [Alpha] EV-4 and EV-8 there was a factor of 55 improvement in performance.² That represented a timeframe of about 10 years. The contribution from technology, essentially in cycle time, was a factor of 7. Thus, the contribution from microarchitecture was more than the contribution from technology. EV-4 was the first 21064, and EV-8, had it been delivered, would have been the 21464.

Take branch predictors. The Alpha 21064 used a last-taken branch predictor. The Alpha 21264 used a hybrid two-level predictor. Or, in-order versus out-of-order execution: The first Alpha was in-order execution. The third one was out-of-order execution. Issue width: You know the first Alpha was two-wide issue. The second one was four-wide issue. I am not saying that we should tell these students, "No multiple cores." I agree, we need to teach thinking and programming in parallel. But we also need to expose them to algorithms and to all of the other levels down to the circuit level.

Audience member: *[interjecting]* Because it's a bad idea. Abstraction is a good thing.

Patt: Abstraction's a good thing? Abstraction is a good thing if you don't care about the performance of the underlying entities. You know, many schools teach freshman programming in Java. So what's a data structure? Who cares? My hero is Donald Knuth, who teaches data structures showing you how data is stored in memory. Knuth says that unless the programmer understands how the data structure is actually represented in memory, and how the algorithm actually processes that data structure, the programmer will write inefficient algorithms. I agree. Ask a current graduate in computer science, "Does it matter whether or not all of the data to be sorted can be in memory at the same time?" How many will say "Yes" and pick their sorting algorithm accordingly?

Programming parallel machines

Audience member: I'd like to pick up on that point. There seem to be a lot of people now who argue that we won't realize the performance benefits of multicore unless programmers explicitly manage data locality. And in some cases that's not so hard if it naturally fits a stream or some other paradigm. But is this a realistic expectation? And what do we do?

Patt: I think there are two types of programmers: yo-yos and serious programmers. And if you really want performance, you don't want yo-yos. So, yes, for every programmer who knows what he or she is doing, there are thousands who don't. Object-oriented isolates the programmer from what's going on, but don't expect performance.

Hill: But I don't think it's as simple as that. I think that there are a lot of cases where people are solving very deep intellectual challenges in their problem domains. Insofar as we can automatically help them create the locality, I think it's a good thing and we should do it.

Patt: Absolutely.

Hill: I am willing to give up some performance. I just don't see people managing the details of locality. I think the

tougher problem is parallelizing the work and coming up with the abstractions somewhere in the tool chain so that the work will be parallelized. I think that locality is also important, but it is not the hardest thing to do.

Patt: Yes, the developer needs tools to break down the work at large levels, but if you want performance, you're going to need knowledge of what's underneath.

Providing simpler programming models

Hill: I still think that even sequential consistency is too complicated for those programmers. You're already reasoning with arbitrary threads and arbitrary nondeterminism and locks, so who wants to also reason about memory reordering and fences?

Patt: You know, I'm hopeful that Mark is right and that someday we will think parallel.

Hill: The key is enabling people to think at a higher level of abstraction. Take SQL: There is a lot of parallelism underneath, but people don't think parallel when they write SQL.

Patt: So, somehow, magically, these intermediate layers will do it for you?

Emer: That's the research that Mark advocates pursuing more aggressively, in lieu of the solutions for higher single-stream performance that you're advocating.

Hill: In addition to! We've got to know a two-handed economist.

Patt: And certainly there are people working on trying to take the sequential abstraction of the program and having the compiler automatically parallelize that code.

Hill: I don't think that's good enough. I mean, just taking C or Java and parallelizing it is not good enough. We need to go higher than that.

Patt: I agree with that. In fact, that's Wenmei [Hwu's] point, that you should have the library available and the heavy-duty logic available, which I agree with also.

Checkpointed processors

Audience member: Any thoughts on checkpointed processors and how they change the trade-off of processor complexity and performance, and perhaps the functionality for multithreading.

Hill: I think checkpointed processors are a very interesting and viable technique that are going to make uniprocessors better, but there are also arguments that they can help multiprocessing. For example, you have a paper accepted to ISCA on bulk sequential consistency (SC) where the checkpointed processors help multiprocessing. I don't see checkpointed processors as fundamentally tipping this debate, but they are a good idea.

Emer: So, you're saying that there's research that applies to both uniprocessor and multiprocessor domains, potentially.

Patt: That's right.

Breaking down the barriers around architects

Audience member: In general, I agree with Mark [Hill], and I'd like to amplify his comments that good abstraction to parallelism using a parallel programming model is probably the most important point. How can architects enable that? In my view, it's not going to happen with just architects working alone. It's not going to happen if the other people are working alone. We really need to work together, and it's very hard to do that. So, the question is, what can we really do to enable that synergy?

Hill: I completely agree. Doing it alone, we can make some progress. But to have the real progress, we have to work all the way up to theoreticians. Well, there are two ways we're going to do this. One is if we can convince our colleagues to do the research. The other is if performance improvement falls flat on its face for a number of years; then they'll start paying attention. I think we want to try to use the former route, because the latter route is not going to be pretty.

Emer: We've been in a situation for a number of years where we've been able

to work behind an architectural boundary to make processors go faster, and software people can be largely oblivious to what we're doing.

Hill: A very concrete example of this is that Microsoft had no interest in architecture research until recently. And suddenly it occurred to them that being ignorant of what's happening on the other side of the interface is no longer a viable strategy.

Emer: They probably did get a lot of performance by being oblivious. That's all the legacy code that we do have.

Audience member: Actually, a good analogy is the memory consistency model issue. The weaker models were exposed to the software community for the longest time, and they chose to ignore them. But in the last five or six years, there's been this huge effort from the software community to get things fixed there. So, I do see a hope that the other people will band together with architects, but I think that something needs to be done proactively to enable that synergy to actually happen.

Patt: Yeah, so what [the audience member] has opened up is whether this should be a sociology debate rather than a hardware one, which I think is right. I think we are uniquely positioned where we are as architects because we're the center. There are the software people up here, and the circuit designers down here, and we—if we're doing our job—we see both sides. We are uniquely positioned to engage those people. Historically, you said that software people are oblivious, and yet they still get performance because we did our job so well. I think we can continue to do our job so well. I don't think we're going to run out of performance if we address Amdahl's bottleneck. For a certain amount of time I think [the audience member] is right, that we need to be engaging people and problems on a number of levels. In fact, I would say, the number one problem today has nothing to do with performance, and that is security.

Hill: I just want to add one more comment. I think what's really tough is to parallelize

code. You may notice that at Illinois too—a lot of software people did some parallel work in the 1980s and early 1990s. But the efforts petered out, and they left! And now there are very few parallel software experts around.

What if there were no power wall?

Audience member: Hypothetically, if we didn't have a power wall, would we be able to continue scaling uniprocessor performance and thus avoid having to go after parallelism?

Hill: There's still the memory wall. The power wall makes it worse because of the high cost in power for many forms of speculation.

Patt: If the biggest problem today is the memory wall, how do we make this bottleneck in the code we want to run still perform well? Again, this is an argument for faster uniprocessors. There will always still be issues with the memory wall, interconnect, and other areas that get in the way of single-threaded performance. The power wall has influenced our thinking and has also helped us to cop out, because architects reason that putting a number of identical low-power cores on the same chip solves the power wall problem.

Hill: I agree that just doubling the number of cores is a cop-out.

Data parallelism?

Audience member: Mark [Hill] has seemed to downplay the potential for data parallelism on the hardware level. It is much simpler to exploit data parallelism from a workload than thread-level parallelism, especially when dealing with a large number of cores (about 1,000) on a chip.

Emer: This kind of parallelism is much more power-efficient than multiple-core parallelism, because one can save all the bookkeeping overhead.

Hill: We need to have data parallelism (not SIMD), but I do not believe that data parallelism is as simple as doing identical operations in the same place due to

conditionals, encryption/decryption, and so on. It's not the same at the lowest level. I'm not convinced that data parallelism is sufficient at the lowest operation level, but it is needed in the programming model.

Vectors are a cool solution, but they haven't seemed to generalize beyond a few classes of applications these past 35 years. The difference between vector processing and multicore processing is that we may have no alternative with the latter. The reason for this is that the uniprocessors will get faster through creative efforts on the part of people like Yale [Patt], but not at a rate of 50 percent a year. We need not only additional cores, but also additional mechanisms to use them, and that is what computer architects have to invent. Simply stamping out the cores may be okay for server land, but not for clients.

Patt: I'm not interested in server land—although that's what made SMT (simultaneous multithreading). If you have lots of different jobs, why not just have simpler chips and let this job run here and that job run there?

Emer: What does SMT have to do with that?

Patt: SMT was a solution looking for a problem, and the problem it found was servers.

Hill: I completely disagree with Yale. Multiple threads are a way to hide memory latency.

Patt: Using SMT to hide memory latency is application dependent, and not always a solution to the memory wall problem. Thus, we come back to Amdahl's law, which we need to continue to address.

Emer: That's what SMT allowed you to do!

The relationship between locality and parallelism

Audience member: You said that locality is less of a problem than parallelism. I would argue that localization is parallelism. If you don't have independent, local tasks, then you can't have parallelism; both are strongly intertwined. Looking at parallel applications,

they did a good job except when parallelism and locality were in conflict, resulting in too much communication.

Hill: Multicore does change things. For example, multicore has much better on-chip, thread-level communication than was previously possible. But multicore chips do have a different type of locality, in that the union of the cores should not thrash the chip's last-level cache. I would like programmers to manage locality, but it seems very difficult to do so.

Audience member: I agree that it's hard, but if you don't, it doesn't work.

Emer: Yale, [the audience member] is supporting you.

Patt: Then I should just sit quietly.

Where should industry put its resources?

Audience member: The business side simply wants applications to run faster. Where should computer manufacturers put their resources? What should they produce?

Emer: This is the fundamental question: How do we allocate our resources as chip designers?

Patt: I would produce Niagara X, Pentium Y. I'd worry about lots of mickey-mouse cores and a few heavy-duty cores with a good interconnection network so you don't have to suffer when you move between cores. Some would argue whether the memory model on the chip is correct, but I think that's the chip of the future.

Hill: I agree with Yale that a multiprocessor like his Niagara X, Pentium Y is a good idea. I would also put effort into hardware mechanisms, compiler algorithms, and programming-model changes that can make it easier to program these machines.

Patt: I agree with that.

MICRO

References

1. R. Chappell, private communication, 2003.
2. B. Herrick, "Design Challenges in Multi-GHz Microprocessors," presentation, 2000.

Asia South Pacific Design Automation Conf. (ASP-DAC 00); <http://www.aspdac.com/2000/eng/ap/herrick2.pdf>.

Joel Emer is an Intel Fellow and director of microarchitecture research at Intel, where he leads the VSSAD group. He also teaches part-time at the Massachusetts Institute of Technology. His current research interests include performance-modeling frameworks, parallel and multithreaded processors, cache organization, processor pipeline organization, and processor reliability. Emer has a PhD in electrical engineering from the University of Illinois. He is an ACM Fellow and a Fellow of the IEEE.

Mark D. Hill is a professor in both the Computer Sciences and Electrical and Computer Engineering Departments at the University of Wisconsin–Madison, where he also coleads the Wisconsin Multifacet project with David Wood. His research interests include parallel computer system design, memory system design, computer simulation, and transactional memory. Hill has a PhD in computer science from the University of California, Berkeley. He is a Fellow of the IEEE and the ACM.

Yale N. Patt is the Ernest Cockrell Jr. Centennial Chair in Engineering at the University of Texas at Austin. His research interests focus on harnessing the expected benefits of future process technology to create more effective microarchitectures for future microprocessors. He is a Fellow of the IEEE and the ACM.

Joshua J. Yi is a performance analyst at Freescale Semiconductor in Austin, Texas. His research interests include high-performance computer architecture, simulation, low-power design, and reliable computing. Yi has a PhD in electrical engineering from the University of Minnesota, Minneapolis. He is a member of the IEEE and the IEEE Computer Society.

Derek Chiou is an assistant professor in the Electrical and Computer Engineering Department at the University of Texas at Austin. His research interests include com-

puter system simulation, computer architecture, parallel computer architecture, and Internet router architecture. Chiou has a PhD in electrical engineering and computer science from the Massachusetts Institute of Technology. He is a senior member of the IEEE and a member of the ACM.

Resit Sendag is an assistant professor in the Department of Electrical and Computer Engineering at the University of Rhode Island, Kingston. His research interests include high-performance computer archi-

itecture, memory systems performance issues, and parallel computing. Sendag has a PhD in electrical and computer engineering from the University of Minnesota, Minneapolis. He is a member of the IEEE and the IEEE Computer Society.

Direct questions and comments about this article to Joel Emer, joel.emer@intel.com.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/csdl>.

Giving You the Edge

IT Professional magazine gives builders and managers of enterprise systems the "how to" and "what for" articles at your fingertips, so you can delve into and fully understand issues surrounding:

- Enterprise architecture and standards
- Information systems
- Network management
- Programming languages
- Project management
- Training and education
- Web systems
- Wireless applications
- And much, much more ...

IT Professional

www.computer.org/itpro

