Future Microprocessors: What Must We do Differently if We Are to Effectively Utilize Multi-core and Many-core Chips?

Patt, N., Yale

Abstract—*The microprocessor f rst surfaced in 1971 with 2300 transistors. Today, some microprocessors have more than a billion transistors on a single chip, and Moore's Law predicts 50 billion transistors in a very few years. Yet this continuing exponential growth in on-chip resources has not resulted in a corresponding improvement in performance. I believe we need to do things differently if we are to take advantage of what process technology has provided. This paper looks at multi-core chips, describes how we got to where we are, exposes some of the myths being promulgated, and explores some of the ways we might do things differently if we are to exploit these increased resources.*

1 Introduction

The microprocessor first surfaced in 1971 with 2300 transistors. Today, some microprocessors have more than a billion transistors on a single silicon die, and Moore's Law predicts 50 billion transistors in a very few years. This continuing exponential growth has unsurprisingly already resulted in the multi-core chip consisting of two, four, or eight cores (processors) on a single chip, and we will soon see the many-core chip consisting of 64 and more cores. Conventional wisdom predicts thousands of cores on a single die in less than a decade. Yet, we have not seen corresponding improvement in performance.

The pseudo-gurus blame it on the "energy wall," claiming that the sum of the energies dissipated by all the cores at even current frequencies is too high and unsustainable. I agree that if we continue to do things as we have, that is true. But I also believe that if we make some fundamental changes to what we provide in the hardware and what we expect of the software, and what we expect of the programmer, we can continue to improve performance at our previous high rate.

First we need to understand how we got to where we are. Why are the most recent offerings from chip manufacturers multi-core and not single core chips? It is also worth exploring some of the multi-core nonsense being advanced. We also need to understand the real benefits and costs of abstraction, and not simply accept abstraction as a fundamental good. Finally, we need to ask whether parallel programming is possible, or whether we should throw up our hands and give up.

• *Y. Patt is Professor of Electrical and Computer Engineering, Professor of Computer Sciences, and the Ernest Cockrell, Jr. Centennial Chair in Engineering at The University of Texas at Austin, Austin, TX 78712-0240.*
E-mail: patt@ece.utexas.edu

2 Why Multi-core Happened

The first microprocessor showed up in 1971 as the Intel 4004, consisting of 2300 transistors, running at a clock frequency of 106 KHz. Process technology continued to improve in two dimensions: our understanding of the process of growing transistors which resulted in larger and larger fault-free dies, and our understanding of photolithography allowing smaller and smaller geometries which resulted in smaller and smaller transistors. Smaller geometries meant higher frequencies. Smaller geometries and larger dies meant more transistors on each die. Gordon Moore initially predicted the number of transistors on each chip would double every year. Ten years later, he revised his prediction to doubling every two years. No matter, the exponential growth in transistor count per die was in place, and has remained true for the past 40 years. Pat Gelsinger, a high-level VP at Intel says the pace will continue to geometries of 10 nanometers – Moore's Law is alive and well!

Initially, this growth in transistor count went to improving the performance of the single-processor core. Motorola introduced a 256 byte Instruction cache on their MC 68020 in the early 1980s. Pipelining enabled intra-instruction concurrency, which quickly resulted in separate on-chip first level instruction and data caches. It also demonstrated the need for branch prediction, which in turn showed the value of speculative execution. Functional units became more substantial, with the x86 floating point unit being installed on the processor chip in the Intel 486. As the number of transistors on a chip grew to tens of millions, we saw more and more sophisticated branch predictors, a larger and larger second level cache, more and more on-chip functional units, the arrival of the multi-media instruction set, trace caches, and simultaneous multithreading.

However, as the turn of the century saw the number of transistors on a chip pass into the hundreds of millions, it just got too hard to design more and more improvement into that uniprocessor core. Some engineers tried and failed. A much easier solution – put the extra transistors into the second level (L2) cache. This resulted in the next round of microprocessors having larger and larger L2 caches until the size of the L2 cache dwarfed the processor it was supporting. The first Pentium M chip had 77 million transistors, 50 million of which was allocated to the L2 cache. The next Pentium M chip had 140 million transistors, with 117 million allocated to the L2 cache. Bigger L2 caches provide some benefit, but is it enough to justify the use of these enormous resources?

The next step was the obvious and easiest next step: use the transistors to replicate the core. A larger, more effective branch predictor or trace cache would take a lot of thought. Replicating the core takes comparatively little thought. So, Intel came out with Core2Duo, Core2Quad, AMD produced the four core Barcelona, IBM gave us P4, P5, P6, and will soon give us P7.

My reason for describing multi-core as such is not to take a cheap shot at processor designers. On the contrary, doing anything other than what they have done would have been very, very hard. And, it is not clear that the corporate world would have allowed them to do it. But, because we have today multi-cores of their current type is no reason we should accept them as the correct multi-cores. I would rather we adopt the position that we can do better. In section 4, I will suggest how.

3 Multi-core begets Multi-nonsense

I think the biggest problem with multi-core is that, given its existence, too many people have come up with nonsensical pronouncements justifying its existence instead of critically looking at current multi-core chips, and asking how we can do better.

3.1 ILP is dead

The first justification for multi-core is that the benefits of single-core performance have been exhausted. As evidence, one is usually shown performance figures for single core performance as a function of the number of transistors on the chip. We double the number of transistors; performance goes up by 5%. "Hardly worth it," the naysayers claim. A closer look, as stated in Section 2, provides the insight that those extra transistors did not go into the core, they went into the L2 cache. So, we use the doubled transistor count to increase the size of the L2 cache and then blame the processor for not doing any better. Perhaps we have the wrong culprit.

Studies have shown that better branch predictors get much better performance. Microarchitecture research has come a long way from the 2 bit saturating counter of 1981 which showed up on the Intel Pentium chip ten years later to some of the sophisticated two-level predictors like TAGE [1] that have appeared in the literature in the past few years.

Tailored accelerators also improve performance. Even the anemic AMD 29000 had a Find First instruction in the mid-1980s that used a simple priority encoder as an accelerator to speed up the Find First operation from a code fragment taking tens of instruction cycles to a single cycle instruction.

I submit that ILP is far from dead if one wants to seriously address what can be accomplished with a sophisticated single core.

3.2 Make the cores simple

Pseudo-gurus are touting multi-cores with thousands of identical simple cores. The mantra seems to be: the more cores the better, and the way to get more is to keep them simple. We actually played that tune in the 1980s where designers attempted to design many, very simple cores into a single chip. The transistor count was much smaller then – barely a million, so the simple cores were really simple. But the mentality remains.

I would argue for two things: accelerators and an asymmetric chip with many simple cores and a very few (perhaps only one) heavy-weight cores. I call such a chip an Asymmetric Chip MultiProcessor (ACMP). The many simple cores are for the embarrassingly parallel parts of parallel algorithms. The heavyweight core is for the serial bottleneck first identified by Gene Amdahl (Amdahl's Law), and for executing code in critical sections where executing fast and thereby leaving the critical section more quickly provides great performance benefit [2], [3].

As to accelerators, there is no limit to what we can provide on the chip for handling important functions, if we have the ability to power off those accelerators when not in use. Transistor count is freely available. Transistors that are allowed to sit by idly draining leakage current are not freely available. The bottom line: we identify, like the AMD engineers did 20 years ago with Find First, a set of accelerators that are really useful when they are needed, and we design them into the chip. When not needed, they sit there powered off, doing no harm. When needed, they provide great benefit.

Even the classical processing elements can be made to perform greater benefit if we invest more of the transistor budget in making them more sophisticated. Subordinate simultaneous multithreading [4] provides an opportunity for improving the performance of on-chip resources, Runahead execution [5] provides a mechanism for concurrently satisfying off-chip memory accesses, DIP [6] provides a better cache replacement policy.

The point is that if we assign the transistor budget to improving the performance of a heavyweight core, rather than adopting the mantra that more simple cores are better, there are lots of places where engineering ingenuity may prevail.

3.3 Make the interface high

Another mantra we hear constantly now that we are in the era of multi-core is that the interface to the software needs to be high. Otherwise, most programmers will be overwhelmed. It is not my place to indict most programmers. However, there are programmers – perhaps 5 to 10 percent of them, who can take advantage of the underlying hardware. For them, not providing a low level interface denies them the ability to practice their craft.

The answer, I think, is quite straightforward: multi-core demands multiple interfaces, each tuned to the skill set of a set of programmers. Certainly we need to provide a high level interface for those programmers who need it. But, can't we do that while also providing the low level interface for those programmers who can use it? And, while we are at it, we need that middle layer of software to bridge the gap between what the high level programmers require to be effective and what the low level programmers can effectively use.

4 Abstraction: A double-edged sword

Conventional wisdom dictates that abstraction is an absolute good. The higher the level of abstraction one works at, the more productive one is. I would add: True if you do not care about performance.

I have lots of examples from technology and from non-technology (i.e., life) to demonstrate this. My favorite example was a taxi ride I once had in the financial district of Manhattan, in the heart of New York City. I needed to get to JFK airport. I got into a cab, and announced, "Take me to JFK airport." What could be a higher level of abstraction than that. I did not tell the driver how to drive the automobile. At a higher level, I did not tell him what route to take. I just gave him five words – take me to JFK airport. In fact, I probably could have reduced my instructions to one word: "JFK"! The problem was that since I did not deal with any lower level of abstraction, the driver was free to take me along whichever route he wished. In this instance, it was over the Triboro Bridge, which substantially increased my fare, and almost made me miss my plane.

A second example comes from the early VLSI design days of the late 1970s, when designers first started designing circuits while being oblivious to how transistors actually worked. In one instance that I am very familiar with, the result was a disaster. Then the engineers looked at each other, noted that they really understood how transistors worked, small-signal models and all, and set out to redesign the chip, putting their lower level knowledge to work. The result: a working chip.

The point is that while it is true that raising the level of abstraction increases productivity, it puts one at the mercy of everything below the level one is working at. That is, everything below the level must work perfectly. It also prevents one from taking advantage of what goes on at the lower levels.

Raising the level of abstraction and improving performance are fundamentally at odds with each other. To show how this is relevant to the chip multiprocessor, I need to introduce the transformation hierarchy.

4.1 The transformation hierarchy

The transformation hierarchy (see figure 1) is the name I gave 25 years ago to the mechanism that converts problems stated in natural language (English, Serbian, Hindi, Japanese, etc.) to the electronic circuits of the computer that actually do the work of producing a solution. The problem is first transformed from a natural language description into an algorithm, and then to a program in some mechanical language, then compiled to the ISA of the particular processor, which is implemented in a microarchitecture, built out of circuits. At each step of the transformation hierarchy, there are choices. These choices enable one to optimize the process to accommodate some optimization criterion. Usually, that criterion is microprocessor performance.

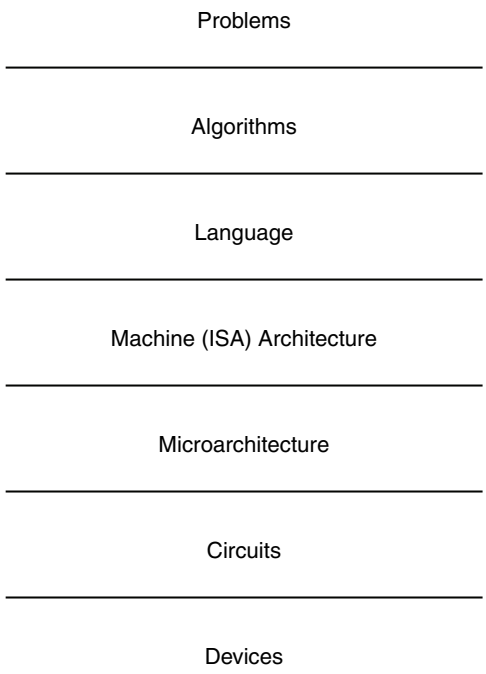

Fig. 1. Levels of Transformation

Up to now, optimizations have been done mostly within each of the layers, with artificial barriers in place between the layers. It has not been the case (with a few exceptions) that knowledge at one layer has been leveraged to impact optimization of other layers. One could argue that such is as it should be, since this approach, working within one's layer of abstraction allows one to become an expert at that layer. It also provides great comfort since most people know very little outside their layer.

Ask computer science graduates if their choice of sorting algorithm would depend on whether all the data to be sorted can reside in memory at the same time or whether it has to be piecemeal brought in from the disk. Their answers more often than not would be that it does not matter. Even a casual reading of Knuth, vol 3 identifies the differences between "internal sorts" and "external sorts."

4.2 What if we break the layers?

If we break the layers, and introduce discomfort to those who have operated within their own level of abstraction, we open up new opportunities to improve performance.

We have already seen this with respect to the Compiler and Microarchitecture. Predicated execution can get rid of conditional branches if the microarchitecture supports conditional execution. The Block-structured ISA [7] compiled programs into instructions having the granularity of a basic block. With microarchitecture support for this instruction, operations within an instruction can be reordered, resulting in decreasing the number of bypass networks. With explicit linkages between operations within a basic block, register pressure can be decreased dramatically. The net effect is increased performance.

In a larger scope, if those working at the algorithm layer talked more to those working at the microarchitecture layer, accelerators could be more effectively identified. The algorithm people know what they want, the microarchitects know whether they can provide it.

If people working at the circuit layer talked to the microarchitects, things like the ill effects of soft errors due to increased frequencies perhaps could be eliminated.

5 Parallel programming

Everyone says, "Thinking in parallel is hard." Perhaps "thinking is hard." Is thinking in parallel hard, or are people told it is hard often enough that it is a self-fulling prophesy.

I tried an experiment in the first year course "Intro to Computing" that I teach. I gave the class – on the spot, with no warning, the following problem: Suppose it takes a computer five units of time to perform a multiply. How many units of time (approximately) to compute 10 factorial? Answer: approximately 40 units, since eight multiplies are required. Unsurprisingly, everyone got it correct. Then I said, suppose you have two processors working together, how many units approximately? More than half, with very little effort came up with 25 units, noting that if processor A directs processor B to compute 5!, processor A, after multiplying 10x9x8x7x6 can multiply its result by 5! supplied by processor B.

My point is a simple one. Certainly, considering all asynchronous actions that can go wrong in parallel processing is not easy, but I wonder how much better we can do if we introduce parallel thinking early in the computer scientists' education before they are convinced that it is hard.

6 Some final observations

My point in this paper has been to call attention to this new world of microprocessors we have recently entered, called multi-core, and ask how we are to continue to exploit the enormous potential it can provide. First, I argue that the existing multi-core paradigm – lots of identical cores – is not the right answer. I submit that a better solution would be ACMP with serious accelerators for doing serious work.

Second, I would argue that our current approach to provide high level interfaces where all programmers can retain their current level of abstraction will not best utilize the chips that we could produce. I recognize that people's desire to remain within the comfort zones provided by their respective levels of abstraction and their hesitancy to embrace parallel thinking is real and pervasive. However, I believe this provides a tremendous opportunity for education. Education is never a short term solution, but it is a solution we need to embrace immediately.

Finally, I believe that if we break with current trends, the multiprocessor of the future will be a many core processor with lots of simple cores to handle the embarrassingly parallel parts of parallel algorithms, but with at least one heavyweight core to handle the serial bottleneck called Amdahl's Law and the execution of critical sections. I believe that this heavyweight core will continue to acquire structures needed to continue to improve the performance of single instruction stream programs. I think the chip will have accelerators identified by algorithms people, and support for structures identified by compiler people. I believe the chip will provide multiple interfaces for multiple types of programmers. But I believe none of this can happen unless we break with the past and at least some of us do things differently.

Acknowledgments

I don't know about other professors, but for me, my abilities are constantly challenged and improved by my interaction with very bright, very focused graduate students, starting with my one PhD student at NC State in the early 1970s, my PhD students at Berkeley in the 1980s, at Michigan in the 1990s, and at Texas since 1999. Since space does not allow me to name them all, I must name none of them since each in some unique way is my "best student," and I gratefully acknowledge them all.

References

[1] A. Seznec, "A 256 kbits l-tage branch predictor," Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2), vol. 9, 2007

[2] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," In Proceeding of the 14th international Conference on Architectural Support For Programming Languages and Operating Systems (ASPLOS'09), 2009.

[3] M. A. Suleman, Y. N. Patt, E. A. Sprangle, A. Rohillah, A. Ghuloum, and D. Carmean, "ACMP: Balancing Hardware Efficiency and Programmer Efficiency," HPS Technical Report, TR-HPS- 2007-001, The University of Texas at Austin, 2007.
[4] R. S. Chappell, J. Stark, S. K. Reinhardt, Y. N. Patt, S. P. Kim, "Simultaneous Subordinate Microthreading (SSMT)," In Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99), 1999.
[5] O. Mutlu; H. Kim; Y. N. Patt, "Techniques for efficient processing in runahead execution engines," In Proceedings of the 32nd International Symposium on Computer Architecture (ISCA '05), 2005.
[6] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," In Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07), 2007.
[7] S. Melvin, and Y. N. Patt, "Exploiting fine-grained parallelism through a combination of hardware and software techniques," In Proceedings of the 18th Annual International Symposium on Computer architecture (ISCA '91), 1991.