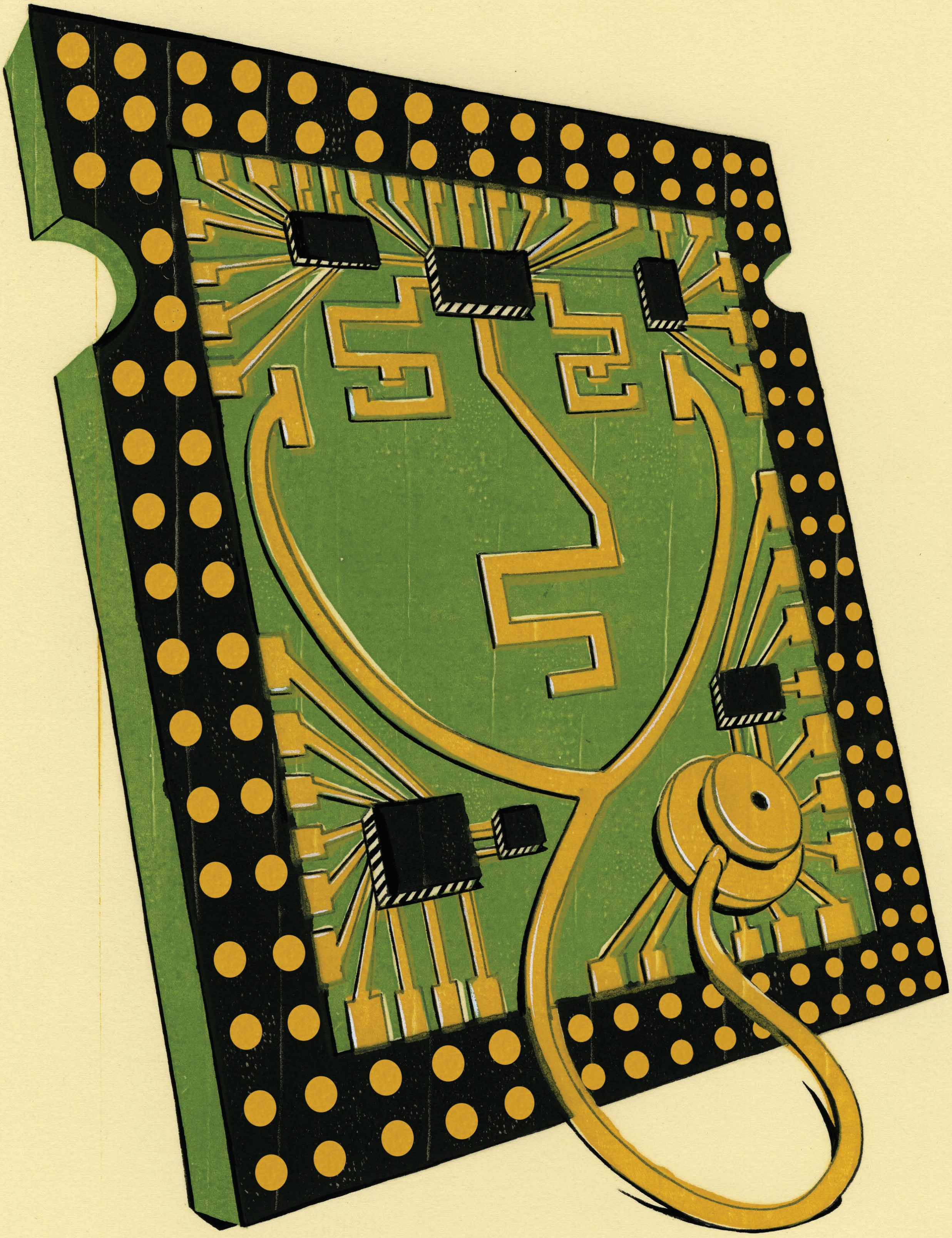


A FAULT-MONITORING MICROPROCESSOR DESIGN CAN SAVE POWER OR ALLOW OVERCLOCKING

BY DAVID BLAAUW & SHIDHARTHA DAS

CPU, Heal Thyself

IN THE OLD DAYS, COMPUTER VENDORS WOULD OFTEN pull a fast one. They would tell you their system had the latest microprocessor when it actually had a cheaper, slower version running faster than the chip's rating permitted. So the shiny, new 500-megahertz system you thought you were buying might contain only an overclocked 300-MHz CPU. But the computer worked fine; indeed, it might have operated perfectly for years, with you none the wiser. And you perhaps replaced it only because a good buy on a 1-gigahertz machine eventually came along.

How did that poor 300-MHz processor cope with such abuse? The short answer is that the manufacturer had set the clock speed low to ensure that its products would function without fault despite the inevitable variations among chips and among their different operating environments. Shady overclockers took advantage of that conservatism, inviting unpredictable failures when they eliminated the chipmaker's prudent safety margins.

Lately, overclocking has gone mainstream. You can, for example, find competitions on the Web in which hardware hackers vie for top honors in this domain. Even chip manufacturers themselves are doing it in public trials to show off how blazingly fast their processors can run under the right conditions—like when they are being cooled with liquid helium to within a few kelvins of absolute zero.

DANIEL BEJAR

Engineers at Advanced Micro Devices, of Sunnyvale, Calif., did just that this past April to prove that the company's Phenom II CPU could break the 7-GHz barrier. In theory, they could have used the same approach to reduce the voltage at which this chip runs at its normal clock speed. That in turn would have significantly diminished the power it consumed.

While saving a few watts is not so important in a desktop system, it's critical for smartphones, mobile Internet devices, and other such gadgetry, which now have to handle glitzy graphics, video, Web access, and gaming without burning too quickly through their tiny batteries. And reducing the amount of power that a CPU uses translates to an enormous amount of money saved for the companies that deploy vast numbers of microprocessors in large-scale server farms.

The problem is that if you use anything less than their normal voltage, some of these chips, on rare occasions, will fail to produce the correct results. That might happen when a laptop is turned on after being left to bake in a hot car, for example. The resulting miscalculations could be catastrophic—or maybe not.

What if a microprocessor could check its output and correct any error on the fly? Suppose further that the chip could slow itself down or turn up its voltage slightly when it noticed it was flubbing up too often. Experiments we and our colleagues at the University of Michigan in Ann Arbor have carried out show that adding those capabilities to a microprocessor can slash energy use by more than a third.

DANIEL BEJAR

THIS POWER-SAVING TRICK can be very effective, but only if the chip is intentionally designed to fail at times. That's because every chip is different—both in how it comes off the production line and how it is ultimately used—so you need to push a given chip to its limits to truly know what those limits are. But the thought of operating something without a comfortable safety buffer makes most engineers shudder. Perhaps that's one reason why for the past decade or so circuit designers have attempted to conserve battery power in less radical ways.

One easy approach, called clock gating, disables the clock signal in the circuitry that isn't working on a given operation. That way, the bypassed transistors won't use energy while switching on and off. A variation of this theme not only disables the clock but also cuts off the power being fed to the unused components. Doing so can substantially improve the energy efficiency of cellphones and similar mobile devices, which typically idle for long periods, interrupted by short bursts of activity.

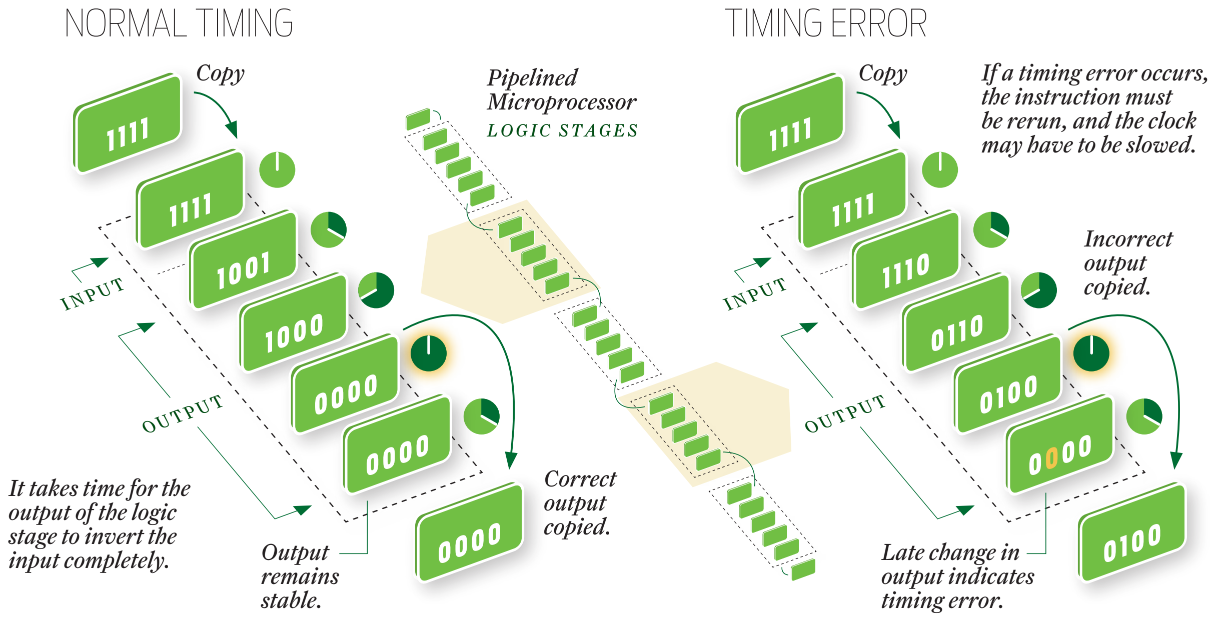
In 2003, we began exploring more ambitious ways to reduce the power required to run a microprocessor—work done together with Todd Austin and Trevor Mudge, our colleagues at the University of Michigan, and with the help of many students. We were aware that chip designers routinely compensate for manufacturing variations, as well as for high-temperature or low-voltage conditions (which can vary even within a single chip), by specifying an operating voltage that's higher than it really needs to be. We knew, too, that manufacturers have had to become more and more conservative in this regard, because it has been increasingly difficult to control the operating characteristics of transistors as they get smaller—a factor of two safety margin is not uncommon. Yet only a few of these transistors will ever experience problematic temperature or voltage conditions for very long. So most of the time, the built-in safety margin just squanders energy.

How, we wondered, could we minimize this waste? One day while we were chatting about possible ways to do that, one member of our research group noted that our basic goal was to shave the safety margin to a minimum. So, fittingly enough, we named the hardware modifications we were working on Razor.

Our idea—inspired by the adage "If you're not failing some of the time, you're not trying hard enough"—was to reduce the operating voltage until the chip would sometimes stumble; then we'd give it a way to recover. Although this tactic had been proposed before for a few very specialized applications (for within-chip communications and in certain digital-signal processors), our Razor project was the first to apply it to a general-purpose microprocessor.

POWER SAVINGS IN THE PIPELINE

The authors' Razor circuitry saves power by reducing the microprocessor's operating voltage. This slows the processor's many transistors, increasing the chance of a timing error, but Razor includes a safety net. Consider one logic stage of a pipelined microprocessor running normally [below, left]. In this example, logical 1s are transformed to 0s, although the signal lines do not change all at the same time. If the transistors involved in the operation switch too slowly [right], incorrect results are copied, but the subsequent change in the output indicates a timing error.


NORMAL TIMING

TIMING ERROR

It takes time for the output of the logic stage to invert the input completely.

Output remains stable.

Correct output copied.

If a timing error occurs, the instruction must be rerun, and the clock may have to be slowed.

Incorrect output copied.

Late change in output indicates timing error.

We knew up front that it would take some energy to monitor for failures and correct them—an overhead that would need to be kept small. But we didn't know how common those failures would be as we reduced the operating voltage. Would the first failures be sporadic or would nearly all the instructions go haywire? Armed only with some very approximate theoretical analysis and crude experiments, we took it mostly on faith that the failure rate would be reasonably low—say, one instruction in 10 000, making error correction feasible.

What we found after we built and tested our first Razor chip was that as the voltage dropped to where the chip just started to err, it in fact erred far less often than we had guessed—typically only once in every 100 million operations. The rarity of failures meant that we could more than make up for the cost of error correction on the millions of instructions that executed properly at low voltage.

The Razor system we devised manages the trade-off between system voltage and error rate. It monitors for failure and automatically tunes the supply voltage to achieve the error rate that saves the most energy. In a sense, the chip maintains its own health.

WHAT'S THE MOST ENERGY-EFFICIENT way for a microprocessor to determine that it has messed up? And how can it reliably correct its mistakes? To understand the system we've engineered to do those things, you need to know a little about how modern CPUs work.

To speed processing, most of these chips use a strategy called instruction pipelining. Although the name conjures up a water pipe, the better analogy is to a bucket brigade, where one person fills a pail with water and passes it to a second person, who then passes it to a third, and so forth.

Continued on page 52

CPU, Heal Thyself

Continued from page 43

All the while, the first person is filling and handing off more buckets.

A pipelined microprocessor owes its high speed to the same strategy of breaking down each operation into a series of discrete steps. For a simple processor, there are often five: Fetch the instruction to be carried out from memory, decode it, execute it, determine the address in memory where the result is to be written, and write it there. High-end microprocessors might extend this strategy to a couple of dozen separate pipeline stages.

Pipelining works only because these different functions can all be carried out at the same time. For example, while one of the programmed instructions is being executed, the following one can be decoded, and the one after that can be fetched from memory. Each step is carried out by a specialized circuit that takes the input provided to it, reacts to it in some fashion, and then presents the results to the next stage in the logic pipeline.

As with an actual bucket brigade, these operations need to take place with a regular rhythm. Here, the microprocessor's clock provides the necessary timing. At some designated instant—say, when the clock signal switches from low voltage to high voltage—each processing stage makes a copy of the data on its input lines. Each stage then works with its copy to produce a result.

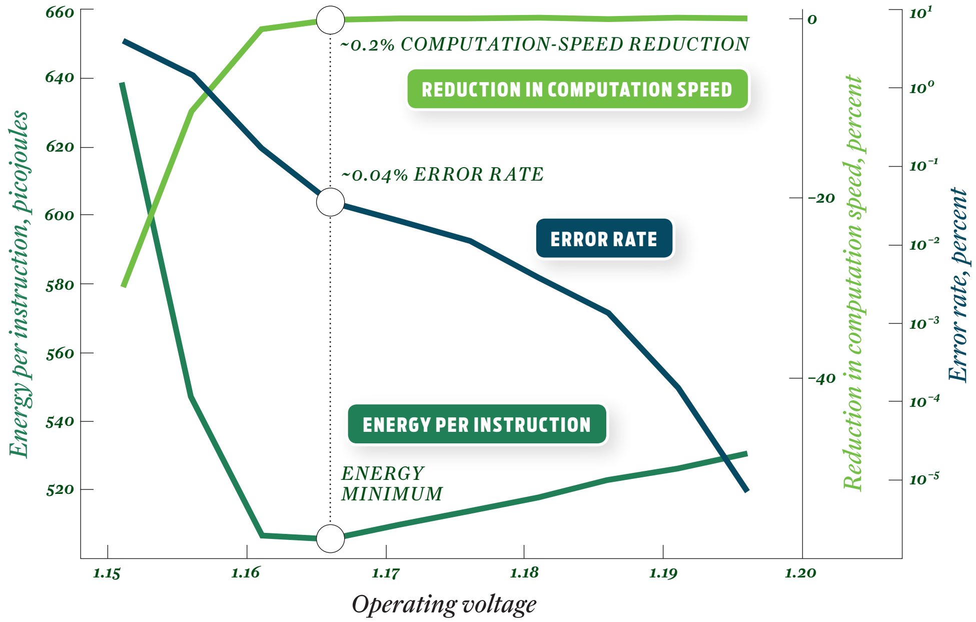
The time it takes for the input of any stage to be translated into the corresponding output depends on how long it takes the different transistors involved to switch states. The processor's clock is normally set to run slowly enough to ensure that the output will be correct by the time the clock next switches from low to high—which is to say, when the output from one stage becomes the input for the next one. As long as the transistors are finished switching states by the time the next low-to-high clock signal comes around, everything works well.

Now suppose you turn down the supply voltage so that the microprocessor's many transistors can't switch logic states quite so fast. One or more slowpoke transistors within some critical calculation pathway may cause an output to switch states *after* the clock has commanded the following stage of circuitry to copy the data presented to it. Working with the wrong input data, that next stage would, of course, produce an erroneous output, which would wreck whatever operation is flowing through the chip's instruction pipeline. This could easily cause the application—or even the whole computer—to crash. Razor provides a way to avoid such a fiasco.

With our latest version of Razor, each copying circuit is modified so that it includes a transition detector,

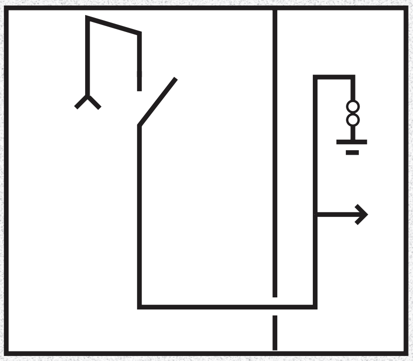
HITTING THE SWEET SPOT

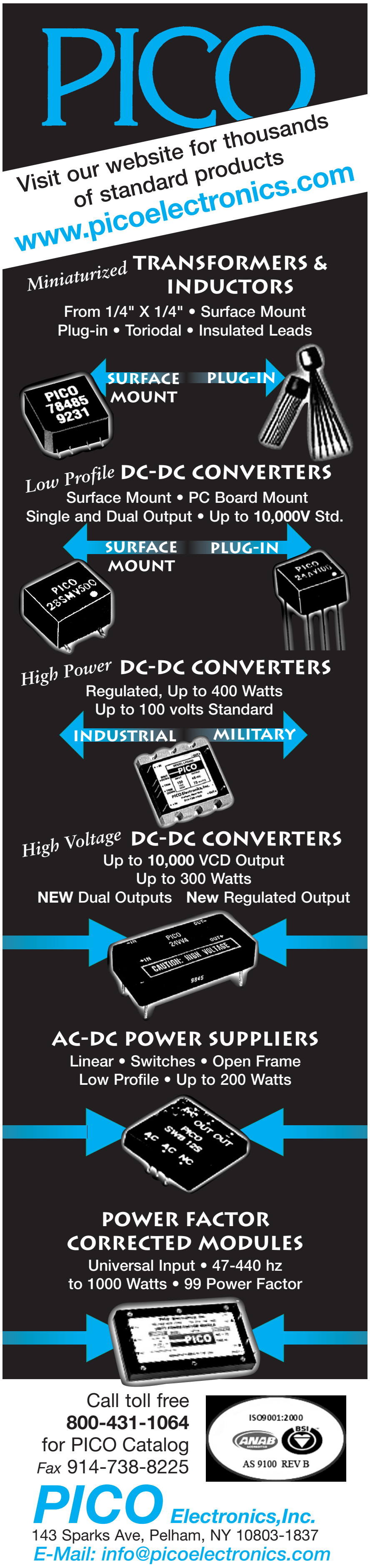
The Razor circuitry is used to maintain an error rate of 0.04 percent, which keeps energy use at a minimum while barely affecting computation speed [right].



which is sensitive to changes in the output for a short period of time after each tick of the clock. If the output is not yet valid at the clock tick, the next logic stage will be working with the wrong data. But catastrophe can still be averted, because the correct data will arrive slightly later, triggering the transition detector, which flags the event as a timing error.

When this occurs, a special error controller executes the problematic instruction again. Although it rarely happens in practice, it's possible that this particular instruction will produce an error on the next attempt, too—maybe even on many repeated attempts. To avoid such a deadlock, the controller we've designed tries only a handful of times. If the error persists, the controller





circuitry cuts the processor's clock frequency in half during the next attempt to ensure adequate time for the error-free operation of the problematic instruction. The correction process might seem cumbersome, but as the first iteration of our Razor system has shown, this chain of events occurs so infrequently that it slows the average computation speed by only a fraction of a percent.

Ironically enough, the biggest challenge in designing the Razor system has been to prevent the microprocessor's circuitry from working too quickly. The reason this can be a problem is that the transition-detection circuitry is dumb: When it sees a signal line change state shortly after a clock tick, it doesn't know whether this is old data from the previous clock cycle arriving late or new data from the current clock cycle arriving early. So the transition detector could mistakenly flag the early arrival of valid data as an error. And such an event might well occur again and again during attempts at recovery, even with a slower clock.

To prevent this from happening, we had to introduce some extra delay into the microprocessor's speedier circuitry. This ensures that the output of a given pipeline stage doesn't switch states while the transition detector connected to it is still sensitive to changes. Adding delay to the faster circuits consumes some power, but it doesn't diminish the processor's overall speed, which remains limited by how quickly the slowest circuits can operate.

You might guess that we also needed to add some very complex circuitry to the microprocessor to enable it to repeat an operation after a timing error occurs. In fact, we did very little, because most of the replay circuitry was already there. It's required to deal with one of the subtle drawbacks of instruction pipelining—the dependence that one instruction often has on the outcome of the previous instruction. That's a problem for a pipelined microprocessor, which must begin processing the second instruction before the result of the first one is known.

In such instances, the microprocessor often guesses what the result will be. The answer could determine, for example, whether to jump to some other part of the program. If the processor guesses correctly, all is well. If not, the microprocessor executes the instruction once more using the correct result as input. This was just the mechanism we needed to force the microprocessor to replay operations when a timing error occurs.

THE INITIAL VERSION OF Razor included both error-detection and correction circuitry within each copying circuit, more than doubling the number of transistors needed to copy a stage's input data. The most recent version of Razor, which we described publicly in 2008, adds only error detection to the various pipeline stages. It relies entirely on the microprocessor's existing replay circuitry for error correction. This approach significantly reduces the area on the chip and the amount of energy the new circuitry consumes.

We've also limited the additional area and power needed by adding Razor elements only to those portions of the circuitry that are prone to fail when the operating voltage is reduced. This requires a careful analysis to be done up front for each microprocessor design. But that effort pays off handsomely, because only 10 to 30 percent of a proces-

TO PROBE FURTHER

An overview of adaptive techniques for microprocessor design (including Razor) is presented in the chapter "Architectural Techniques for Adaptive Computing," by Shidhartha Das, David Roberts, David Blaauw, David Bull, and Trevor Mudge, in *Adaptive Techniques for Dynamic Processor Optimization: Theory and Practice*, Alice Wang and Sam Naffziger, editors, Springer, 2008.

A more detailed description of the early version of Razor is available in "A Self-Tuning DVS Processor Using Delay-Error Detection and Correction," by S. Das, D. Roberts, S. Lee, S. Pant, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, *IEEE Journal of Solid-State Circuits*, Vol. 41, no. 4, April 2006.

The newer strategy for Razor is described more fully in "RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance," by S. Das, C. Tokunaga, S. Pant, W.-H. Ma, S. Kalaiselvan, K. Lai, D. M. Bull, and D.T. Blaauw, *IEEE Journal of Solid-State Circuits*, Vol. 44, no. 1, January 2009.

sor's copying circuits typically require Razor's protection against failure.

Although we've tried to minimize the additional power used as much as possible, the Razor circuitry does use a small amount of energy at all times. So if you were to run the modified microprocessor at its standard operating voltage, it would use slightly more power than normal. But the Razor circuitry doesn't do that; it lowers the operating voltage until the error rate is about 0.04 percent. This saves far more energy than the additional transistors consume. Running the chip at still lower voltages would, however, make the error rate surge, requiring many replays, which would slow computation appreciably and use more energy overall. So the Razor circuitry senses the error rate and adjusts the supply voltage to keep the chip at its optimum operating point—which depends not only on the environment and manufacturing variations but also on the calculations being performed at that time. As they say, your mileage may vary, but on average you can expect a 35 percent savings in energy use.

Since our team first described Razor, in 2005, several semiconductor companies have started to investigate this approach. For example, at the IEEE International Solid-State Circuits Conference in February, researchers from Intel presented a Razor-inspired mechanism for improving high-performance chips. Instead of reducing the supply voltage to save energy, the Intel team keeps that voltage constant and uses a Razor-like technique to boost computation speed by 25 to 32 percent. And one of us (Das) now works full time with other researchers at ARM, a British company that designs and licenses reduced-instruction-set computer processors, in an effort to harness Razor's ability to make processors faster or less power hungry.

Razor is one of the more far-out attempts to push microprocessors to their limits—but we're sure it's not going to be the last of its kind. As transistors continue to shrink, we can expect chip designers to invest a growing portion of their transistor budget in circuitry to monitor, analyze, predict, correct, and adapt. Computers might never obtain anything resembling consciousness, but the microprocessors they contain will surely become increasingly self-aware. ❑