Select Font Size: A A A

Sponsored By SPECTRUM

RAM for Free

By **Lei Yang**, **Robert P. Dick**, **Haris Lekatsas**, and **Srimat Chakradhar**

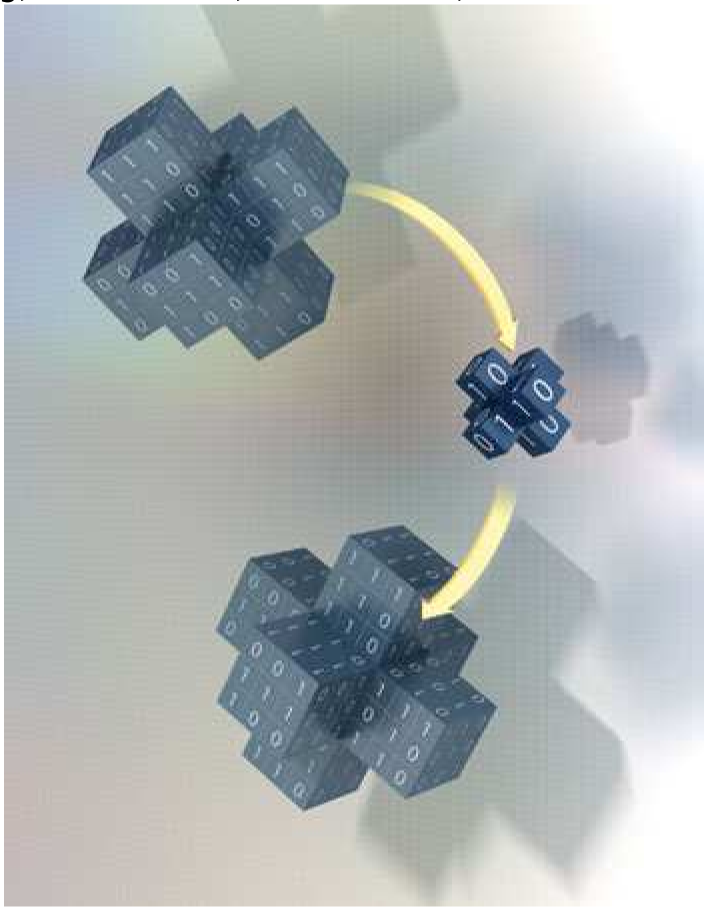

IMAGE: EMILY COOPER

Give us a reading on the 1202 program alarm," radioed Neil Armstrong to Mission Control in July 1969, seemingly about to lose his famously cool demeanor. He was busy trying to steer his spacecraft to the first-ever manned landing on the moon and was worried that this error message from his guidance computer meant serious trouble.

Fortunately, a young computer engineer at Mission Control had the insight to realize that this error was not as ominous as it seemed, and on his signal the Apollo 11 landing went forward. Within hours of the astronauts' safe touchdown, it became clear what had happened: the lunar module's rendezvous radar had remained switched on during descent, when only the landing radar was needed, and the craft's navigation computer had become overtaxed trying to process radar data from the two sources. The system's programmers had, however, built in a fail-safe mechanism that would shed the less critical tasks so that the computer could do its main job.

The Apollo Guidance Computer was arguably the first example of an embedded system, one that incorporates a special-purpose computer dedicated to a single function. Embedded systems have become the predominant form of computing, exemplified by the microcomputer-operated brains inside your microwave oven, your MP3 player, your cellphone, even your refrigerator, to name just a few. Although many such devices now have 10 000 times as much random-access memory (RAM) as the mere 4 kilobytes of their Apollo-era great-granddaddy, memory constraints continue to dog their designers.

Cost is one reason: although the prices of RAM have plummeted fast, the need for memory has expanded faster still. Another concern is all the energy that RAM requires. Manufacturers of mass-market products, especially portable devices like music players and mobile phones, must therefore take care to add no more memory than the software needs to operate. Hewing to that line is no easy trick. To speed a new product to market, hardware and software engineers have to work along parallel tracks, which means that neither side can know quite what the other has in store.

If software designers yield to temptation and ask for more memory than they could possibly need, they risk wasting a lot of money—even pennies matter when you're producing millions of units. Or the product could end up being too power hungry. Yet if they skimp on RAM, they may prevent the unit from running some new killer app that would allow the gadget to beat the competition. Such mistakes sometimes force companies to redesign their hardware, a process that is enormously costly and time-consuming.

We have spent the better part of three years trying to give designers of embedded systems a third option: to increase effective memory by compressing the data stored in RAM using just software.

Data compression is standard fare in other parts of the computer business, of course. DiskDoubler was a top-selling software utility during the early days of the Apple Macintosh, for example. By encoding regular patterns in the data in more compact form, it gave users the feeling they'd doubled the space on their disks—something well appreciated at a time when hard drives held only a few tens of megabytes.

Conventional wisdom held that such data compression required specialized hardware

Unlike the “lossy” compression employed, say, for encoding JPEG images, the compression used to store programs and associated data has to be “lossless”: it can't drop a single bit. Conventional wisdom held that additional special-purpose chips were required for this (or the functions of those chips had to be designed into the processor, at great expense). Otherwise, the thinking went, operations would be too slow for memory compression, and in portable systems the energy usage would become prohibitive.

Speed and power are indeed often critical for embedded computers, as is consistency of response time. So it's no wonder that their designers have shied away from data compression and just added RAM as needed. But it occurred to us that if you could use data-compression software to control the way embedded systems store information in RAM, and do it in a way that didn't sap performance appreciably, the payoff would be enormous.

We began our investigations by taking a long, hard look at existing compression techniques. The most promising appeared to be something called Lempel-Ziv-Oberhumer, or LZO, one of the family of widely used Lempel-Ziv algorithms invented in the late 1970s by the Israeli computer scientists Abraham Lempel and Jacob Ziv. In the mid-1990s, an Austrian programmer named Markus Oberhumer wrote the LZO variant in ANSI C, designing it expressly for speed.

Once we had selected the compression algorithm, we needed to devise ways to determine which of the data in memory should be compressed and when. We also had to come up with the means to expand the compressed data when they were needed. While this might seem nightmarishly difficult, in fact, getting LZO compression to work reliably in our test-bed system—a PDA—proved straightforward. We were able to make use of the virtual memory feature of the PDA's Linux operating system, which swaps infrequently used data between RAM and disk storage. We modified this mechanism to compress such data when memory requirements exceed physical RAM capacity. But instead of writing the compressed data to disk, the system stores it on a virtual device—a portion of the available RAM.

When an application requires the previously compressed information, our software locates and decompresses it, then moves it back to the uncompressed part of the PDA's memory so that the application can continue to work normally. What's more, our scheme allots space for the compressed data as needed, so that applications that work fine without extra RAM aren't slowed down at all.

Programmed in this way, our PDA could use far less memory to run various applications—games, office tools, even media apps, whose compressed sound and image data are normally expanded when they are in RAM. Attempts to operate these same applications using reduced physical RAM but without data compression generally crashed the system—or made it grind along at an intolerably slow pace. To ensure that the comparison was valid, we wrote special software to monitor and then replay user input with identical timing properties, with and without data compression. And to fully quantify how much the compression degraded performance, we ran some benchmark tests that didn't require any user interaction at all. Execution time went up by almost 10 percent on average and by nearly 30 percent in the worst case. Recognizing that some situations might not tolerate such lethargy, we sought to do better.

The challenge, then, was to come up with an algorithm that would rival LZO's degree of compression but be much quicker. We managed to satisfy both requirements by exploiting regularities in the kinds of data that are typically stored in RAM, a scheme we dubbed pattern-based partial match. It resembles many other compression techniques in that it replaces frequently used patterns with short codes and rarely used patterns with longer codes. The basic strategy is not unlike what Samuel Morse adopted when he translated the alphabet into a series of dots and dashes—which explains why the Morse code for the commonly used letter E is just a single dot, while the rarer J requires a dot and three dashes.

Pattern-based partial match takes advantage of the fact that much of the RAM in embedded systems is wasted. For example, in a system with 32-bit (4-byte) data "words," numerical values often demand just 4, 8, or perhaps 16 of these bits. The rest of the bits are zeros. An integer variable, for example, is normally just 16 bits wide. And when a particular integer is a small number—say, the hour or minute value you set on your cellphone alarm—most of those 16 bits, too, will be zeros. And even when all 32 bits are really required (to index an arbitrary spot in memory, for instance), nearby words often hold similar values, because they point to adjacent memory locations. So such a word can be encoded compactly merely by keeping track of how much it differs from a neighbor.

The extent to which you can compress data depends on the patterns in the input. For example, a 32-bit word whose 3 most significant bytes are zeros is packed into just 12 bits in our scheme. What's more, our system maintains a dictionary of frequently used data words. If a word of the input exactly matches something currently stored in the dictionary, those 32 bits get squeezed down into just 6 bits.

The dictionary is constructed on the fly, so it can adjust if the statistical properties of the data change—say, when a pattern that was rare becomes common. You might think this would make decompression impossible, but in fact, this neat trick really works. The key is that the dictionary is not saved for later lookups. Rather, when it comes time to decompress things, the compressed data themselves are used to re-create the dictionary—and to update it in a manner that keeps it matched with what was used to compress the data in the first place.

Jon Louis Bentley of Bell Labs, along with three colleagues, invented this technique for data compression in the mid-1980s, an approach that has since become known as move-to-front coding, because newly encountered patterns are placed at the front of the dictionary. Our version allows for 16 possible matches (or partial matches) to the contents of eight separate two-entry dictionaries, a somewhat odd arrangement that allows the lookups to be made at blazing speed.

Our investigations of the data held in RAM showed plenty of opportunities to squeeze things down. The algorithm we devised for this can't match LZO or some other well-known techniques for general-purpose data compression—say, for compacting the contents of an image file—because most files don't have the same tendencies as the data typically found in an embedded system's RAM. A 32-bit word used to encode the color of a particular pixel, for example, is not likely to be chock-full of zeros. But for compressing RAM, our system excels. It reduces the space needed by about 60 percent, and it's startlingly fast. Indeed, our testing revealed that ripping out this much memory results in a performance penalty of just 0.2 percent on average and 9.2 percent in the worst case. That is, this software effectively gives an embedded system more than twice the memory it had originally—essentially for free.

Translating these gains from the lab bench to the marketplace has not been a trivial undertaking, however. In January 2007 we filed for patents on the process, which we dubbed CRAMES, for Compressed RAM for Embedded Systems. We were keen to use it to address a real-world problem: industry's seemingly continual need to redesign embedded systems like mobile phones so that they can run ever more complex applications. (Who would have thought that high- school students would be using their phones to study for the SAT?)

NEC, which together with the U.S. National Science Foundation sponsored our project, wanted to reuse existing hardware for its next-generation cellphone applications, and our compression scheme allowed it to do just that: the company's Foma N904i smart phone, released in September 2007, uses CRAMES. But plenty of practical hurdles sprang up along the way.

For example, in real life an application sometimes terminates when its data are still in the compressed part of memory. Because we initially designed our software not to know the “owner” of any particular piece of compressed data, it was not possible to find out which parts of memory belonged to defunct software processes. So over time, useless data would pile up.

We licked this problem by keeping track of the compressed data and the applications that created them so that the system could free up orphaned chunks of memory. This change required slight modifications to the operating system, however. The production version of CRAMES remains highly modular, but it is not quite as elegant as the stand-alone operating-system add-on we first envisioned it to be.

Initially, we tested CRAMES on a cellphone prototyping board that ran applications that didn't require access to the airwaves. We measured performance with and without CRAMES switched on, while either cutting back on the amount of memory available or starting more applications than is normally possible.

Later, NEC engineers in Japan gave CRAMES the acid test by running it on hardware connected to a telecom network. Because there was very little time before the first product was to ship, they couldn't test our system of pattern-based partial match completely. So they chose to install a version of CRAMES that uses older, slower, but more thoroughly proven Lempel-Ziv compression code.

We were thinking only of embedded systems when we first engineered CRAMES. But since then we've also been working on some exciting possibilities involving general-purpose computers (which might allow us to compress our acronym for this data compression, fittingly enough, to CRAM).

Consider the recent trend of putting more than one processor core on a chip. Doing so boosts computing power, sure, but it sacrifices the space on the chip that's available for cache memory, which can hamstring an application whenever it requires frequent access to main memory. We're now designing pattern-matching hardware to compress the data in cache memory to speed up applications.

Another exciting prospect on the horizon involves computers at the other end of the spectrum: tiny ones found in the least-expensive, lowest-cost systems, like those in kitchen appliances. Some of these systems have only 4 KB of RAM—matching what Armstrong and Aldrin had at their disposal in 1969. We've developed software that manages the memory of such tiny computer systems and compresses data when space gets tight.

Although we don't envision our memory-compression technique being used for anything as dramatic as a lunar landing, it's ready for use in equipment that's every bit as critical to safety: sensor networks that monitor the structural integrity of buildings or bridges. These applications will require a great deal of testing, of course, because the software must be able to handle unexpected situations without failing catastrophically. People accept that their desktop computers will crash every once in a while, but such embedded systems must be robust enough not to cause real crashes—as the designers of the very first one, the Apollo Guidance Computer, knew well.

About the Author

LEI YANG and her colleagues explain how some clever software can replace costly hardware in “RAM for Free”. Yang is a research assistant at Northwestern University, where she studied with ROBERT P. DICK, an assistant professor of electrical engineering and computer science. For two summers, Yang interned at NEC Laboratories America, where she worked with SRIMAT CHAKRADHAR, head of the labs' systems architecture division, and with HARIS LEKATSAS, who is a consultant to designers of embedded systems.

To Probe Further

For a listing of the authors' related articles, see http://robertdick.org/tools.html.