# PowerPC Adopts Code Compression

## *IBM PowerPC 405 Core's CodePack Compresses Entire Blocks of Object Code*

*by Jim Turley*

EMBEDDED PROCESSOR FORUM

Code compression, long a feature of ARM and MIPS chips, is coming to PowerPC. IBM's new 405 will be the first PowerPC chip to have this important feature. Perhaps thumbing its nose at the other code-compression developers, IBM believes that CodePack will squeeze PowerPC code by an average of 40%, and that it can do so without resorting to an alternative, shorthand instruction set. CodePack will be included in the PowerPC 405 when it first appears in 2Q99.

At the Embedded Processor Forum earlier this month, Tom Sartorius, IBM's principal architect for embedded PowerPC, lifted the wraps on the company's latest processor core for ASIC designs. The PowerPC 405 is faster, smaller, and more power efficient than the current 403 core, and it offers some additional features as well.

### Real Compression, Not Compaction

CodePack is fundamentally different from both Thumb (see MPR 3/27/95, p. 1) and MIPS-16 (see MPR 10/28/96, p. 40). The latter two are actually separate 16-bit instruction sets that duplicate a subset of the host chip's native 32-bit instruction set. CodePack, on the other hand, offers compression in the literal sense: it compresses blocks of object code using run-length encoding that are then decompressed in real time with symbol tables generated by the compression utility. Using CodePack is like running PKZip on PowerPC programs.

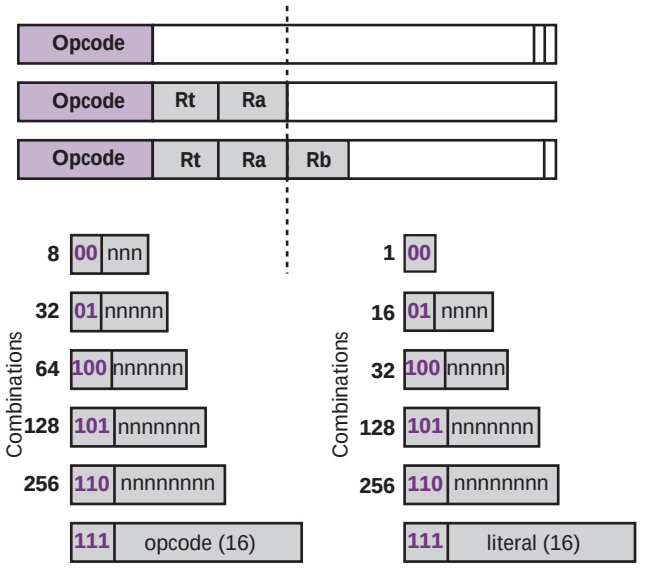



**Figure 1.** IBM's CodePack splits each 32-bit instruction into halves and compresses each half separately, into a 2- or 3-bit tag followed by a variable-length symbol. The most frequently occurring instructions are encoded into the fewest bits. Infrequent patterns might be "compressed" into more bits than they had originally.

There is no secondary CodePack instruction set. Programs are simply written, compiled, assembled, and linked in a perfectly ordinary manner. Then, before the executable code is burned into ROM, stored onto disk, or downloaded, it passes through a compression postprocessor. The compression utility scans the entire object file, analyzes it for frequency distribution, and outputs a compressed version of the program.

This compressed object file would be completely useless and unexecutable without a CodePack-equipped processor to run it. At run time, the processor decompresses the object code in real time, restoring each instruction to its original condition as it is fetched and loaded into the instruction cache. The underlying concept is deceptively simple; the devil lies in the details of IBM's implementation.

### Code Is Cached in Uncompressed Form

CodePack works entirely "outside" of the caches, so its effect (indeed, its presence) is completely undetectable to the processor core. CodePack conceptually lies much closer to the chip's bus interface than to its cache or pipeline.

Because CodePack works its magic before code reaches the caches, packed code is cached in its uncompressed form. Both MIPS-16 and Thumb, on the other hand, cache instructions in compacted form, effectively increasing the capacity of their instruction caches by 25% or more (but only when executing compacted code).

The PowerPC 405 chip, therefore, does not enjoy the same benefits from increased code-cache capacity as do the compacted-mode processors. On the other hand, CodePack does not in any way affect the internal design of the processor pipeline, so it can easily be applied to any PowerPC CPU, including existing cores such as the 401 or 603e. CodePack could also be applied to future PowerPC cores without compromising their basic performance or compatibility.

### Algorithm Not New; Hardware Implementation Is

IBM's compression algorithm is patented, although it relies heavily on prior art and other generally available compression algorithms, such as Huffman or Lempel-Ziv. IBM's system makes no real contributions to the art and science of data compression. Rather, it marks the first time that decompression has been done in real time, in hardware.

For its compression algorithm, IBM was able to assume the input data (the symbol set, for the initiated) was

PowerPC object code. That being the case, CodePack's algorithm does not have to be general purpose or deliver consistent results across a random set of inputs data. Because IBM knew about the inputs CodePack would receive, it was able to fine-tune the algorithm to produce better results.

The upper half of each PowerPC instruction always holds the opcode, usually with two or more register specifiers, while the lower half usually holds immediate data or a branch displacement, as Figure 1 shows. The opcodes tend to be encoded in a regular and predictable manner, a vestige of their RISC-inspired heritage.

Data, inconveniently, tends to be random. Immediate data displays neither the regularity nor the predictability so characteristic of opcodes. IBM's frequency analysis showed that the upper and lower halves of typical PowerPC instructions have quite different frequency distributions, shown in Figure 2, so the two halves are handled with slightly different algorithms.

## Compression Changes From Program to Program

After evaluating the entire program to be compressed, the CodePack compression utility develops a frequency profile and assigns the most frequently occurring patterns to the shortest compressed tokens (Huffman encoding). That frequency distribution changes from one program to the next, so CodePack does not use a single, fixed encoding scheme as Thumb or MIPS-16 does.

Instead, the CodePack compression utility determines the best, most efficient encoding for each program individually. After the program is compressed, the utility also produces two symbol, or decompression, tables, one each for the opcodes and for data. (The compression utility can also be used to analyze multiple programs and produce a single, shared set of decompression tables.)

These two symbol tables must stay with the program, for they are the only key to decompressing it properly. There is no universal key to decompressing CodePack programs; each key is different. These symbol tables, which are 2K in size, are downloaded into RAM cells within the CodePack-equipped processor by the bootstrap loader (which must obviously be uncompressed code).

IBM believes that CodePack can reduce object code size by 35–40%, which is about equal to the claims of MIPS-16 and Thumb devotees. Unlike the others, CodePack can compress all code, including interrupt handlers, systems functions, and OS calls. Like the others, CodePack also has a somewhat unpredictable effect on performance. In all three cases, data structures are not compressed at all.

IBM is considering producing chips with hardwired symbol tables built into the CodePack logic, in lieu of SRAM lookup tables. Such hardwired tables would be cheaper and smaller than SRAM tables, but they would force a fixed encoding for all programs the chip might run. If the code base is known ahead of time, the proper table could be designed in; if not, the table would have to be supplied as input to the compression utility for all future programs.

## Common Patterns Are Given Short Tags

CodePack treats the upper 16 bits differently than the lower 16 bits, as Figure 1 shows. The upper 16 bits (the opcode and registers) can be compressed into as few as 5 bits or as many as 19. Frequently occurring patterns in the lower half might be compressed into as few as 2 bits or as many as 19 bits. In the best case, an entire 32-bit PowerPC instruction can be compressed into 7 bits, or about 22% of its original size.

In contrast, infrequently used combinations of opcode and immediate data might each be "compressed" to greater than their original size. The worst-case compression of a 32-bit instruction would be into two 19-bit fields, or 120% of its original size.

As the figure shows, compressed fields (either upper or lower half) are identified by one of six tags. The "00" tag identifies the shortest encoding for the most frequently occurring symbols. For upper-half (opcode field) encoding, the 00 tag is followed by a three-bit field identifying the eight most commonly occurring opcodes; 01 identifies the 32 next-most frequently occurring symbols, and so on.

After the 488 most common symbols have been encoded, CodePack handles the remainder (potentially another 65,047 permutations) as literals, and it labels them as such by tacking 111 on the front. For lower-half (immediate field) encoding, the 00 tag stands alone. IBM found that zero occurs frequently enough to warrant its own extra-short tag.

## Compression Done in 64-Byte Chunks

Using CodePack is not an all-or-nothing proposition for programmers (or processors). Some parts of a program may be compressed and other parts left uncompressed. As mentioned previously, some sections of code may actually swell when "compressed." In such cases, the CodePack compression

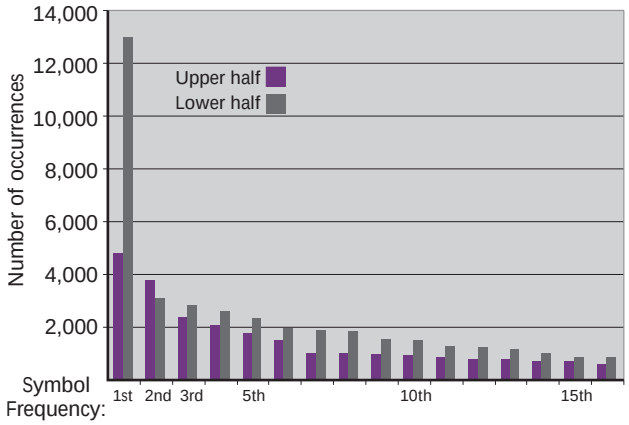



**Figure 2.** IBM's frequency analysis for typical embedded PowerPC code reveals that the distribution for the upper half (opcode) rolls off much more slowly than does the lower half (immediate data). The heavy usage of the value zero is also clear from the chart.

utility will not compress those portions of the program, opting instead to leave them in their natural state.

Programs to be compressed are scanned in 64-byte (16-instruction) blocks. If compression will result in a smaller block, those 16 instructions are compressed. If the block would get larger, or not compress at all, those instructions are left alone, and the compression utility moves on to the next 64-byte block. The entire program is eventually evaluated, 64 bytes at a time. By the end, perhaps three-quarters of the blocks might actually be compressed.

This invites the question: How does the the PowerPC 405 know if the code it's executing has been compressed or not? This quandary is handled by the MMU.

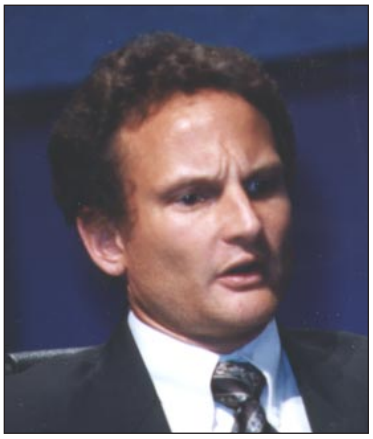
A previously reserved attribute bit in the MMU's lookup tables marks each page as either compressed or uncompressed. (Pages may be from 1K to 16M in size.) Fetching code from an uncompressed page bypasses CodePack entirely and avoids its performance penalty, as if nothing had happened.

Instructions in a compressed page are not necessarily compressed, but they may be. A tag at the beginning of each block identifies its compression status.

If a block is uncompressed, it will bypass CodePack when it is fetched. The penalty for checking and then discarding the header is one access to an index table. Thus, there is some incentive for programmers to store uncompressed code in uncompressed pages.

If the block is, in fact, compressed, the entire 64-byte block is transferred in a burst from memory and passes through the CodePack decompression hardware. The block is then decoded into an internal buffer before passing to the instruction cache or execution unit.

This process repeats indefinitely, for as long as the program executes. Serial execution of compressed code is relatively straightforward. Handling compressed branches, on the other hand, presents a new set of problems.



MICHAEL MUSTACCHI

**Tom Sartorius of IBM describes the PowerPC 405 CPU core at the Embedded Processor Forum.**

## Change of Flow Causes Problems

A drawback to IBM's approach to compressing code is locating the exact address of a compressed instruction. When the 405 branches to compressed code, the CodePack logic must translate the target address to the actual location of the instruction in compressed memory. This problem doesn't arise with either Thumb or MIPS-16, because those extensions use fixed-length instructions whose location is known to the compiler, assembler, and linker.

CodePack-equipped processors handle this problem with a translation stage independent of the MMU. The CodePack logic maintains a hash table of entries, each 32 bits in size, that map decompressed addresses to their actual, compressed locations. In each entry, 26 bits point to the beginning of the compressed block in physical memory. The remaining six bits point to the beginning of the next compressed block relative to the first one.

As with MMU tables, the compressed-to-decompressed translation table is kept in memory. The 405 must fetch a table entry from memory to do a lookup. Only the most recently used entry is maintained in a "cache" on the chip.

Having located the target block of instructions, the 405 fetches it from memory and begins decompressing the entire block into an internal 64-byte buffer. Finally, the target instruction is forwarded to the processor core for execution.

The duration of this ordeal varies widely and depends on many things, including the instruction's position within a block. IBM's compression algorithm forces CodePack to decompress instructions serially, starting from the beginning of the block. It cannot decompress instructions "critical word first." In the current implementation, CodePack decompresses two instructions per clock, or eight clock cycles for the entire 64-byte block. If the target instruction is the sixteenth one in the block, it will be decompressed last. On the other hand, if the compressed instruction is the first in its block, the 405 will receive it eight clocks sooner.

IBM claims that the performance of CodePacked programs will be within ±10% of the performance of uncompressed programs. The positive value (i.e., a speedup) is possible when code is executed directly from very slow memory (such as ROMs), or over a narrow data bus, because blocks of compressed code can be fetched in less time. Both Thumb and MIPS-16 offer the same advantage, executing faster over a narrow bus than uncompressed code would.

## Apart From CodePack, Little Is New With 405

In his presentation, Sartorius described the PowerPC 405 pipeline and the core's interfaces, although CodePack was the highlight of the design. Like ARM's 740T (et al., see MPR 4/20/98, p. 10), the 405 includes the instruction and data caches as part of its base definition; the core cannot be used without them. This is only logical, as any 200-MHz ASIC design would need sizable caches to get decent performance.

IBM has set a minimum target frequency of 200 MHz for its 405 core in 0.25 micron, but Sartorius feels that speeds of 275 MHz or more are likely. The 2.5-V core is expected to cover just 2.0 $mm^2$ and draw 400 mW.

Interestingly, the 405's five-stage pipeline is not significantly different from that of the PowerPC 403 (see MPR 5/9/94, p. 1), yet it runs at four times the clock rate. The 405 achieves this improvement largely by reversing many of the decisions made with the 403, which was designed to minimize size and power usage. That, and its better process technology (0.25-micron vs. 0.35-micron), allowed the speedup.

The 405 executes all the standard "Book 1" integer instructions, including multiply-accumulate functions. Although the core includes a hardware multiply-accumulate (MAC) unit, multiply latencies range from 2 to 5 clock cycles, depending on the magnitude of the operands. Divide instructions finish in a fixed 35 clock cycles.

The 405's hardware MAC is viewed as an auxiliary processor unit (APU, or coprocessor) by software. The MAC unit has 24 instructions of its own, all variations on the basic multiply-accumulate function. Signed, saturating, and modulo arithmetic options fill out the 24 instructions.

### Price & Availability

IBM's PowerPC 405 core will appear first in the 405GP microprocessor, which will begin sampling in 2Q99. Pricing for the processor has not been announced.

For more information, contact IBM (Research Triangle Park, NC) via fax at 415.855.4121 or visit IBM at *www.chips.ibm.com/products/embedded/index.html*.

### Binary Compatibility Is Somewhat Compromised

CodePack's after-the-fact compression means that existing PowerPC binaries can be compressed and run on a new CodePack-equipped processor, even if the original source code is not available. And as far as compilers and other software-development tools are concerned, CodePack doesn't exist. It's invisible to the programmer and the compiler.

CodePack is nearly—but not entirely—invisible to hardware tools as well. The big hangup will be with debug systems that examine or modify memory without passing through the processor. These tools will have to be aware of CodePack's existence or risk corrupting program code. At the very least, disassembling code from memory will require knowledge of CodePack's decompression algorithm and access to the program's specific decode tables. Inserting a breakpoint into already compressed code would be virtually impossible without upsetting the rest of the program.

CodePack exacts its toll in performance, as do Thumb and MIPS-16. Decompressing the object code takes time, of course, but the hit is felt only during instruction-cache misses and appears as unusually long memory latency. On the plus side, fewer bytes of code need to be transferred during a miss, offsetting some of that extra latency.

Unlike other compaction schemes, CodePack doesn't require "mode switching" between the compressed and uncompressed states. Indeed, the processor might switch between these two states every instruction, without the programmer's knowledge or complicity.

While it's not clear yet whether this might be an advantage or a disadvantage, CodePack could make programs less portable. Every program will compress differently, yielding different characteristic decompression tables. Without those tables, compressed programs are not binary compatible among PowerPC chips. A multithreaded system would need to use the same symbol tables for all executables, or waste time swapping keys along with the code.

This characteristic could be carried over to serve as encryption for CodePack programs. Without the individual decompression tables, it would be very difficult to decompress a CodePacked program. Reconstructing the lower-half (immediate-operand) table would be especially challenging.

The emergence of CodePack might also lead to some previously unthought of benchmarking of compilers. IBM has established that different compilers, given the same source code, will produce programs with different frequency distributions. These, in turn, will compress with varying degrees of success. (This is one reason IBM did not develop a single "approved" compression/decompression symbol table for PowerPC binaries.) That being the case, some compilers might be preferred not for their "native" code density but for how well they typically pack with CodePack.

### Very Different Technology, Similar Results

The 405 core holds its own against other forthcoming embedded cores, such as the ARM10 and MIPS cores from LSI Logic and others. As nice as the 405 core is, its significance is overshadowed by IBM's innovative CodePack compression system.With CodePack, IBM may be asymptotically approaching the ideal code-compression system for microprocessors.

The absolutely best compression would come from treating the entire program—data and all—as a single block to be compressed, using an adaptive algorithm (such as the Lempel-Ziv). But such systems are designed for all-or-nothing decompression, not for extracting partial results or intermediate symbols. Total compression makes it impossible to branch into the middle of compressed code, something that all programs need to do. So IBM compromised by chunking up the data set every 64 bytes, using the best algorithm for that block size. This makes it possible to jump into the middle of anywhere with some hope of decompressing and executing the instruction in a reasonable amount of time.

CodePack is a big step to one side in the technology of code compression. Its claimed compression ratio is about the same as that for less elaborate alternatives, but it has the not-insignificant advantages of compressing unmodified executable programs, leaving the core pipeline untouched, and being compatible with past, present, and future PowerPC processors.

Code Pack is also interesting for the pitfalls that it highlights: decompressing code in real time, relying on multiple levels of translation, and tying decompression tables to individual programs. In the end, CodePack will certainly become widespread within IBM's ASIC business. It's also an interesting highway post on the road toward greater integration, where transistors are almost free, and features like CodePack pay off in smaller memory arrays. M