

3 Gekko the Dolphin

by Fiora

3.1 The Porpoise of Dolphin

Dolphin is one of the most popular emulators, supporting games and other applications for the GameCube and Wii game consoles. Featuring a highly optimized just-in-time (JIT) compiler and graphics unit that translates GPU opcodes into vertices, textures, and shaders, Dolphin is able to emulate almost all GameCube and Wii games at high speeds on a modern x86 CPU.

Instead of trying to do a detailed anatomy of the entire system, much of which is beyond my current understanding, in this PoC||GTFO article I'm going to focus on some particularly evil assembly optimizations and interesting bug fixes in the Dolphin JIT from the past two months—some large and dramatic, others small and elegant (or horrifically hacky, depending on your perspective!) But first, let's do a quick overview of how Dolphin works and some of the biggest difficulties inherent in Gamecube/Wii emulation.

Dolphin's JIT is superficially similar to a typical PowerPC emulator, but things are not nearly so simple as they appear. The GameCube Gekko CPU (and the extremely similar Broadway CPU on the Wii) has a number of particularly odd features that aren't present on a typical PowerPC.

- A “paired singles” SIMD unit, somewhat similar to 3DNow! but complicated by some of PowerPC's inherent weirdnesses with floating-point (32 bit floats are represented as 64 bit internally, similar to x87).
- Built-in “graphics quantization” registers, which allow quantized loads and stores based on runtime-variable parameters, up to and including the data type to be converted to and from.
- A complex memory layout with mirrored regions and a slew of MMIO features, including a memory-mapped FIFO usually connected to the GPU, but which can also be repurposed for other uses by games.
- The ability to directly access—and modify—the active GPU frame buffer.
- Complex cache manipulation features, such as the ability to enable a “locked cache” and access memory as cached or uncached.
- A floating point unit with its own very unique definition of the word “multiply.”

Making emulation even more difficult, games tend to abuse every aspect of the system imaginable, from the precise rounding of every floating point instruction to self-modifying code to behavior that isn't even defined in IBM's specification for the CPU. Additionally, games typically run in supervisor mode, giving them the ability to abuse a wide variety of features user-mode applications can't. All of this leads to severe limits on the shortcuts Dolphin can take; the most benign-seeming optimization often results in a slew of unintended consequences. Dolphin can't even reorder memory loads; an attempt to do this resulted in a real game failing because of exception handling semantics not being maintained.³

³Dolphin-Emu issue 5864

NEW! DS-CAPS
\$89.00*

A Unique Keyboard or Program Activated Data Switch for the IBM PC or Any MS-DOS System.



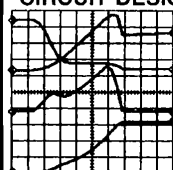
This compact, self-powered switch is software controlled at the keyboard to program to direct parallel data from the computer to peripherals. Eliminates the time and frustration of re-cabling to use different peripherals. To install, connect between computer and peripheral, plug into power, boot support software etc. and you are ready to code select and direct data flow between the two devices.

Also available is a complete line of manual and electronic switches plus converters to interface and direct data between CPUs and programs.

U/A WEST, Inc.
 The Interface Company
 154 North Shore Hwy., Tucson AZ 85705
 (602) 633-5716

*All units shipped freight collect. Add \$10.00 for shipping. Please allow 2-3 weeks for delivery. Quantities discount available. All residents add 7% state and local taxes. Data conversion manual.

CIRCUIT DESIGN TOOLS FOR PC'S

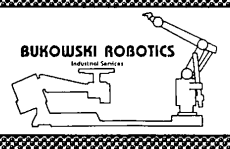


IS_SPICE \$95
 SOFT_SCOPE \$175

Perform AC, DC and Transient analysis with IS_SPICE. View, manipulate and plot data with Soft_Scope. Requires 640K RAM, coprocessor, fixed disk and color graphics adapter.

Write or call **intusoft**
 PO BOX 6607 (213) 833-0710
 San Pedro, CA 90734-6607

BUKOWSKI ROBOTICS



APPLE II™ I/O ROBOTIC CONTROLLER

WE KNEW THE TRUE ENTHUSIAST WOULD FIND THIS ARTICLE WAY BACK HERE IN THE BACK OF BYTE.

THE BUKOWSKI ROBOTICS I/O CARD IS A LOW COST APPLE COMPATIBLE ROBOTICS CONTROLLER CARD THAT MAY BE USED STAND ALONE, OR IN AN APPLE SLOT AS AN INTELLIGENT PERIPHERAL CARD. THE CARD FEATURES AN ONBOARD 65C02 MICROPROCESSOR, UP TO 4K I/O LINES, 2 TIMERS, AND 8K NON VOLATILE RAM. SHIPPED WITH TONS OF SOFTWARE AND SUPPORT. \$129.00

BUKOWSKI ROBOTICS
 1555 W UNIVERSITY #105
 TEMPE AZ 85281 (602) 966-6230

	OOAA AAAA 0000 0BBB 00CC CCCC 0000 0DDD
AAAAAA	6 bit code representing the quantization factor (2^{-32} to 2^{31}) for loads.
BBB	3 bit code representing the data type for loads (float, S8, U8, S16, or U16).
CCCCCC	6 bit code representing the quantization factor (2^{-32} to 2^{31}) for stores.
DDD	3 bit code representing the data type for stores (float, S8, U8, S16, or U16).

Figure 1: GQR Register Format

Yes, there are applications that require precise emulation of MMU mechanics, including post-exception rollback. Yes, there are applications that intentionally try to execute an address of `0x00000001` to trigger a custom exception handler, and won't run unless this behavior is properly emulated. Yes, there are applications that modify code without properly flushing the CPU instruction cache and rely on the mere hope that the old code will have been since replaced in the cache. And yes, there are applications that may do many of these things with the intent of sabotaging Dolphin emulation.

Yet we still have to emulate a 729 MHz PowerPC CPU on a 2-3 GHz x86 CPU, all while trying to run programs that may very well be trying to prevent us from doing so.

3.2 Reserved bits are really just shy

A number of games were breaking in mysterious fashion with the JIT implementation of “paired singles” quantized loads and stores. Some crashed, while others had wildly broken lighting effects or other strange artifacts. Yet, even upon very close inspection, the JIT implementation was nearly identical to the (order-of-magnitude slower) interpreter implementation, which worked correctly. What could games possibly be doing here to break the JIT?

To understand this bug, it is crucial to understand the precise layout of the Gekko CPU's eight graphics quantization registers (GQRs). Each quantized load and each quantized store references one of these eight registers to act as its parameters. Figure 1 describes the format of the GQR registers.

The manual describes the other bits as being zero, but unfortunately, that isn't quite true. They were assumed to be zero, but the CPU never enforced this. Games could—and half a dozen games did—smuggle flag bits through these reserved register bits. Whether this was a bug, or perhaps done for some attempt at anti-emulation code, or even a strange sort of thread-local storage, we may never know.

The JIT's flawed assumption caused the implementation to either read out of bounds in the quantization array or even outright jump to an invalid function pointer. Fortunately, masking out those bits was just a single and operation; the main cost of this glitch was days of debugging by puzzled developers.

Since resolving this issue, I've written hardware tests to test reserved bits in other system registers too, which revealed all sorts of strange behavior. For example, the XER (fixed-point exception register), is laid out as follows.

1	[SO][OV][CA]0 0000 0000 0000 0000 0000 0AAA AAAA
---	--

SO is the summary overflow flag, OV is the overflow flag, and CA is the carry flag, with AAAAAAA being a 7 bit control code for string load/store instructions.

But on the Gekko, the actual bits that the CPU allowed to be set in XER were `0xE000FF7F`; it apparently supported setting the 8 bits in XER[16-23] even though it doesn't support the associated instruction, the string compare instruction `lscbx` (load string and compared byte indexed, similar to `rep cmpsb` on x86). I sincerely doubt any games used those bits in XER, but one can never be quite certain of such a thing.

3.3 Practice your multiplication, or you might become a GameCube CPU when you grow up!

For as long as it's existed, Dolphin has had trouble with replays, like those in racing games (Mario Kart, F-Zero) and fighting games (Super Smash Brothers). Emulation often desynced dramatically within seconds of the start of a console-recorded replay, with cars flying off the racetrack or Mario tripping off the side of the stage. The same happened in reverse, when emulator-recorded replays were transferred to a physical console. This was particularly dramatic in the case of Mario Kart's ghost feature, in which the game let you play against "ghosts" recorded by the developers of the game. The ghosts would very quickly drive into a wall, making victory quite trivial, if not very satisfying.

The source of this strange yet consistent desyncing was the way these games recorded replays. Instead of recording the movement of the karts or characters, the games record the player's input. This is a much more compact representation, but unfortunately, it means the most minuscule error on playback can accumulate until the result desyncs completely. To make replays, ghosts, and other similar features function correctly, Dolphin's floating point unit would have to match the Gekko's to the last bit of rounding.

For many months the Dolphin developer Magumagu exhaustively attempted to reverse-engineer the hardware FPU and make a software implementation. One by one, precise versions of instructions were implemented. Among the first victims were `frsqrite`, approximate inverse square root, and `fres`, the approximate reciprocal, which were replaced with table-driven versions matching the actual Gekko hardware. But it still wasn't enough; replays still constantly desynced, and bizarrely, the trouble seemed to trace back to the multiply instruction.

Some consoles do use non-IEEE floating point, like the Playstation 2; the curiosities of emulating this could make for an article of its own. Yet the Gekko was supposedly equipped with an IEEE-compatible floating point unit, denormals and all! How could multiplies on a GameCube give different results than on a typical desktop PC even with identical rounding flags set?

The problem, as Magumagu discovered, traced back to exactly how the floating point unit's internals were implemented. A double-precision float has 53 bits of mantissa; combined with three guard bits, this makes a 56 bit input. Accordingly, the Gekko had a 56x28 bit multiply and performed double-precision multiplies by combining the results of two 56x28 bit multiplies. Single precision multiplies were done with just one execution of the multiply unit.

But on the Gekko, all floating-point numbers are stored as 64 bit doubles. Single precision operations have reduced output precision and clamp their output to 32 bit precision, but are still stored as 64 bit doubles. Technically, according to the manual, you're not supposed to perform single-precision operations on double-precision values; the result is supposedly undefined. But, of course, countless games did it all over the place, so we still have to emulate it in a way that matches the behavior of the hardware.

Most single-precision operations seemed to be fine with double-precision input; a single-precision floating-point add, for example, seemed to be identical to performing a double-precision add and then rounding to single-precision. But, as Magumagu discovered, multiplies were their own unique brand of bizarre: they rounded the right hand side operand's mantissa to 25 bits of precision (for 28 including guard bits), then performed a 56x28 bit multiply. Note that 25 bits gives neither single nor double precision; it's something in between.

Fortunately, it took just four SSE instructions to perform this rounding operation for each multiply:

```
1 movapd xmm1, xmm0
  pand   xmm0, [truncate_mantissa] ; 0xFFFFFFFF80000000
3 pand   xmm1, [round_bit]         ; 0x0000000008000000
  paddq  xmm0, xmm1
```

The overall performance loss was barely measurable compared to the literally dozens of games with fixed replays or physics, ranging from *Zelda: The Wind Waker* to *Donkey Kong Country*.

As Dolphin's primary tester, Justin Chadwick, once said, "Fiora, I hate how in your build the AI no longer bounces off the track in *Mario Kart Wii*. It makes it a lot harder to win."

3.4 Dolphin intentionally makes thousands of segfaults

Emulating one CPU’s virtual memory subsystem on another CPU is hard. Doing so quickly is even harder. A direct approach would be to map one host page to each emulated page, but that’s impossible on Windows because the Alpha AXP CPU didn’t have a “load 32 bit integer” instruction. I’m not making this up.⁴ The existence of MMIO, VRAM being directly mapped into CPU memory, and mirrored sections of the memory map certainly don’t help.

The simplest approach would be to send every load and store through software address translation, but this proves to be fantastically slow. (Remember, we can only spend about three or four x86 cycles per Gekko CPU cycle!) Dolphin does support a variant of this as “full MMU emulation mode,” which a few games with particular complex memory layouts do require. But for most games, it gets away with a vastly more elegant—or horrific—solution. Which one applies to you depends on how you feel about intentionally triggering thousands of segfaults.

For every memory access, Dolphin first tries to perform address constant propagation—if we know which area of memory an address is in, we can directly pass off the load or store to wherever it’s supposed to go; usually a direct RAM access or a push to the FIFO. For the rest of the memory accesses, it shouts “YOLO” and just goes for it, with seemingly no care for what might happen if the access isn’t to valid RAM.

But Dolphin has an ace up its sleeve: it’s replicated the rough address space layout of the Gekko CPU in virtual memory using the operating system’s shared memory features. Yes, that’s a four gigabyte chunk of contiguous address space, including mirrored sections. (Addresses 0x8010000 and 0x0010000 map to the same place due to mirroring.) Sections that aren’t directly mapped to physical RAM are marked as inaccessible.

When the “YOLO” access fails, a segfault is thrown by the operating system and caught by Dolphin’s handler, which proceeds to backpatch the x86 code that caused the segfault to jump to a trampoline which then redirects to the slow, safe memory access handler. Thus, only the few memory accesses that actually go to non-RAM addresses take the slow route, while the rest are simply a `mov` and `bswap`.

This feature, called “fastmem,” isn’t at all new to Dolphin, but is nevertheless among a core reservoir of hacks that keep Dolphin’s JIT fast. Tests suggest it provides at least a 15-20% CPU performance benefit over runtime address range checking.

3.5 Wasting all your cache is a good way to go bankrupt

As mentioned in the previous section, a few games make sufficient use of the GameCube’s fancy MMU features that they need to take the slow path—full MMU emulation. While address translation (which is hopelessly unoptimized in Dolphin) is a significant cost, the greatest speed cost actually comes from the other consequences of full MMU mode. One of these is that it must check exceptions manually after every single memory operation, and if so, flush the register state, revert any address update that occurred in the load, and jump to the handler. It’s all rather painful and an optimizer’s worst nightmare, as it generates massive code bloat and places great constraints on instruction reordering and other aspects of optimization.

Because of all this, full MMU games tend to require incredible amounts of CPU power to emulate. While a few are at least playable on a very fast PC, others aren’t so lucky. Rogue Squadron 2, for example, was developed by Factor 5, a game developer notorious for their ability to squeeze performance never thought possible out of consoles. In the Nintendo 64 era, they rewrote the GPU firmware to render five times more polygons than it was ever meant to. In Rogue Squadron 2, their incredible stressing of the Gamecube has led to a game that runs at half-speed in Dolphin on a 4 Ghz Intel Haswell CPU.

In addition, likely due to Dolphin’s incomplete MMU implementation, a number of full MMU games simply don’t boot at all: Rogue Squadron 3, Toy Story 3, and Disney Infinity among them. Particularly in the case of the latter, this might very well be anti-emulation code.

Profiling Rogue Squadron 2 with VTune suggested L1 instruction cache misses occurred at a rather high rate. The cost of cache misses is hardly a new topic in the optimization world, but code cache misses tend to be glossed over. Modern x86 CPUs have vast instruction fetch bandwidth, long pipelines to absorb fetch miss

⁴unzip pocorgtfo06.pdf 64k.txt

bubbles, and while performance can certainly be improved by reducing code size, it's often not considered a major factor.

Regardless of this, I figured I would see how much could be gained. I created a “far code buffer” in which to stuff all the rarely-used generated code (like exception handling and recovery for each memory access) instead of having it inline. Maybe this would get us a few percent of a speed increase?

With one rather simple commit, Rogue Squadron 2 sped up over 30% on my Ivy Bridge. The bloating of the generated code had cost so much that the CPU spent roughly 40% of its time sitting idle, waiting for new instructions to come in. The gain was even larger—over 50%—on another developer's Haswell, most likely because the Haswell has even higher instructions per clock-cycle count, and is thus even more susceptible to being front-end bound. Even in POV-Ray, a heavily floating-point-bound benchmark that doesn't use the MMU and was hardly known for its binary size, the gain was roughly 6% overall.

Never underestimate the value of instruction cache on modern CPUs. With a Haswell's four ALUs, two load units, and one store unit, it might very well be able to chew through instructions much, much faster than you can feed it.

3.6 It's normally abnormal for denormals to renormalize

I mentioned previously how the Gekko CPU internally stores all its floats—even 32 bit ones—as 64 bit doubles. This means that Dolphin has to convert floats to 64 bit on load, and convert back to 32 bit on store, at least if the `lfs` (load float single) and `stfs` (store float single) instructions are used. Hypothetically, if a value was loaded immediately and then stored, an optimizing recompiler could remove the conversion, but this can only sometimes be proven safely.

This wouldn't be an issue normally, outside of the small speed cost of a single extra conversion operation on each load and store. But unfortunately, yet again, games are not so kind. A strangely large number of games use `lfs` and `stfs` to copy integer data, which means the conversion process of float-to-double-to-float must be lossless, regardless of input. This would normally work, but at the same time, a large number of games also set the flush-to-zero (FTZ) floating point flag, which causes denormal floating point results to be set to zero by the CPU. Unfortunately, this also applies to our float-to-double and double-to-float conversions, so any game copying integer data that happens to look like a denormal float will have its data corrupted.

We can't turn off FTZ, because that would result in floating point arithmetic errors of the same sort that motivated the multiplication rounding changes mentioned previously. We also can't toggle FTZ off then back on again; the floating point control registers on x86 take upwards of fifty cycles to modify. The initial solution was to set rounding flags for SSE2, then do the load/store conversions using x87 (which, conveniently, doesn't even support FTZ). The one tricky part was fixing up the NaN flags afterward, as x87 handles NaN differently from SSE2, setting an exception flag instead. This is what the double-to-float code looked like.

```
movsd [temp64], xmm0
2 movsd   xmm1, xmm0
fld   [temp64]
4 ptest  xmm1, [double_exponent] ; 0x7FF0000000000000
fstp  [temp32]
6 movss  xmm0, [temp32]
jnc .dont_reset_qnan_bit
8 pandn  xmm1, [double_qnan_bit] ; 0x0008000000000000
psrlq  xmm1, 29
10 vpandn xmm0, xmm1, xmm0
.dont_reset_qnan_bit:
```

This is better than fifty cycles per load and store, but it's still inefficient and gross enough to make x86 assembly writers everywhere squirm in discomfort. The overall speed penalty was around 20% on Super

Smash Brothers Melee—but there was little choice, since the alternative was inaccurate emulation that broke many games.

Fortunately, there is one other way. What if we just checked for denormals, passed them off to a slow, rarely-taken code path, and sent everything else through SSE? This has the bonus effect of not needing to fix up the NaN bit, since only denormals (not NaNs) would take the x87 path. The resulting code looks like the following.

```
1 movq    rax, xmm0
  shr    rax, 55
3 sub     al, 0x6D
  cmp    al, 3
5 jbe    .x87conversion
  cvtsd2ss xmm0, xmm0
7 jmp    .continue
  movsd [temp64], xmm0
9 fld    [temp64]
  fstp  [temp32]
11 movss xmm0, [temp64]
  .continue:
```

The comparison at the top is a bit tricky and designed to minimize code size, since this code will be duplicated countless times throughout generated JIT code. The only actual exponents that need to take the slow path are those in the range [0x369, 0x380], but sending a few more to minimize the size of the comparison has negligible effect on performance (in this case, [0x368, 0x387]). The comparison could be simpler if zeroes are also sent to the slow path, but testing shows that there’s a very large proportion of zeroes—as many as a third of the inputs. With the check shown here, only 0.01% of floats take the slow path and the overall performance penalty for this change drops from 20% to 2%.

As a side note, the official IBM manual claims that the Gekko/Broadway CPU uses denormals-are-zero (DAZ) in addition to FTZ when the non-IEEE (NI) flag is set. Curiously, actual hardware testing shows that the CPU doesn’t ever seem to actually do this.

3.7 Hey I just RET you, and this is crazy, but here’s my address, so CALL me maybe?

Modern x86 CPUs typically have a built-in return stack, designed to predict where a `ret` instruction is heading, with the assumption that every call is paired with exactly one `ret`. This is a pretty good assumption, and in the rare cases where it fails, the performance cost is typically equivalent to a branch misprediction. Without this prediction, a return would be relatively costly and difficult to predict—little different from an indirect branch `jmp [rsp]` or similar.

PowerPC has its own similar call and return instructions: `b \hat{l}` (branch with link) and `b1r` (branch to link register). The first jumps to a location and stores the old location in the link register (the return address), while the latter jumps to the location stored in the link register. When emulating `b1r`, Dolphin treats it as an indirect jump to the link register. This is the natural translation for such an instruction, but it is costly from a branch misprediction standpoint, since such a branch is extremely difficult to predict correctly. Profiling shows a non-trivial number of micro-ops lost to branch mispredictions.

Comex’s idea was to re-use the CPU’s existing return prediction stack. On a `b1` instruction, instead of jumping to the target function, he would push the emulated destination address onto the stack and then `call` the target JIT’d function. When emulating a `b1r` instruction, instead of jumping to the given link register, he compares the link register against the one stored on the stack at `[rsp+8]`, and if the two match, returns with `ret`. If functions call and return as expected, this approach should give near-perfect branch prediction. Despite the seeming increase in instruction count, this led to roughly an eight percent overall speed increase across nearly every game merely from improved return prediction.

The one danger of this is the possibility of the stack overflowing. If a game uses `b1` without an associated `b1r`, the return stack will continually grow until Dolphin crashes. Comex’s first solution was to clear the

stack whenever a misprediction occurred; this reduces the problem to the pure evil case of an application that used `bl` hundreds of thousands of times in a row without any `blr`. Out of curiosity and being a bit pedantic about correctness, he decided to support this case as well, writing a short test case that triggered the problem and setting up guard pages and extending the signal handler to catch any failure.

The core concept of this optimization is not too different from `fastmem`. Hijack a hardware CPU feature (in that case, memory protection, in this case, return address prediction) and use it to help emulate the same feature of the target CPU, even if it wasn't really intended for that purpose.

3.8 Through the SUBFIC and the SRAW we carry on

Like x86, PowerPC has a number of instructions that set flags based on their result. Unlike x86, there are two ways in which this can happen. There's condition flags (`GT`, `LT`, `EQ`, `SO`) which can be set by a comparison operation or an arithmetic instruction with the `Rc` bit set. This is a lot more convenient than x86, because one can generally avoid clobbering the flags when they're not needed, which makes code more efficient and, coincidentally, emulation easier.

Carry flags, on the other hand, are not quite so friendly. Some common instructions set carry unconditionally (`subfic`, `sraw`, `srawi`), enough so that carry calculation becomes a significant cost even in code that doesn't make heavy use of carry bits. The calculation of carry bits for `sraw` and `srawi` in particular is a bit non-trivial, easily requiring a half-dozen or so extra instructions on x86 to emulate.

The first step to optimizing carries was to enhance `PPCAnlyst`, the class that performs dependency analysis on instructions. If an instruction calculates a carry bit, but that bit is overwritten before being used or before reaching a JIT block exit, we can omit the calculation of that carry bit entirely.

`PPCAnlyst` also has an instruction reordering pass that uses dependency information to reorder instructions wherever it can be sure doing so is safe. This was originally just used to move comparison instructions next to branches so the two can be merged, but it can be extended to support a wide variety of operations.

I modified the instruction reordering pass to attempt to "stick" pairs of carry-using instructions next to each other. A large number of common PPC idioms use sequences such as `subc+subfe`; not merely arithmetic on variables larger than the register size. One example is `r0 = (r1 != r2)`.

```
2 subf r3, r1, r2
  addic r0, r3, -1
  subfe r0, r0, r3
```

The PowerPC Compiler Writer's Guide lists a number of these in the appendix.⁵

The third and final step was to take advantage of this; if the next instruction is going to consume the carry bit, take advantage of the x86 carry flag instead of storing the carry bit in the emulated CPU state. This is a slightly tricky (and limited) optimization, since it requires the instructions to follow each other directly, since most instructions will clobber the x86 flags.

Combined with the "sticky" reordering, these changes were able to drastically reduce instruction count in carry-heavy code; some recompiled sequences dropped in size by a factor of two or more. Some games, such as Virtual Console games (an emulator inside an emulator!) went as much as 12% faster just with these carry optimizations.

An interesting future optimization might be to recognize some of the aforementioned multi-instruction compiler idioms and transform them into equivalent idiomatic x86 code; this could be even better than merely optimizing the individual instructions!

3.9 Capturing performance from the flags

As mentioned in the previous section, many integer operations, such as comparisons and operations with the `Rc` (record) bit set, have the ability to set result flags in the PowerPC condition register. The condition

⁵<https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF7785256996007558C6>

register is split into eight 4 bit sections, each of which represents one result, consisting of the LT, GT, EQ, and SO flags. This is in sharp contrast to x86, for which most instructions set flags unconditionally. It only has a single condition flags register instead of eight.

Emulating operations on these flags efficiently is critical to performance in Dolphin. It's often difficult to prove that an update to the flags register won't be used again following its most immediate use (e.g. a conditional branch), so the relevant calculations can't be omitted.

Delroth and Calc84maniac discovered a brilliant way to optimize Dolphin's internal flag representation to minimize the work required to set and read flag bits. These two operations represent the vast majority of operations on flags; everything else, such as boolean operations between flag bits and reading out the flags register, is practically a rounding error by comparison. In addition, reading out flag bits is done almost entirely by conditional branch operations.

The flag representation they invented involves the flags being stored as a 64 bit integer. Bit 63 is equal to !GT, bit 62 equal to LT, bit 61 equal to SO (a flag not fully emulated by Dolphin, but also rarely used except as the output of a boolean flag operation), bit 32 always set, and bits 0-31 set to zero if EQ.

This representation has the useful property that it can be calculated using a single instruction from the result of any integer operation; a 32- >64 bit sign extend (`movsxd` on x86_64). Individual flags can also be read out with single operations:

```
1 GT = (s64)CR > 0
  LT = CR & (1 << 62)
3 EQ = (s32)CR == 0
  SO = CR & (1 << 61)
```

While this dramatically complicated operations such as loading the flags register, the overall performance effect was tremendous. Performance improvements in typical games ranged from six to fourteen percent merely from being able to omit most of the instructions (and code bloat) involved in flag calculation. This change also inspired later optimizations, like splitting carry bits into their own emulated register instead of storing them in XER. There's no requirement that an emulator maintain the same data representations the ISA describes, so long as it transparently performs whatever conversions are necessary for correct emulation.

3.10 With Dolphin, Wii have a bright future

Dolphin still has a long way to go. The graphics engine is imperfect and still missing a few rather difficult features, like zfreeze and OpenGL line-width support. Dual-core mode is still sometimes a bit finicky with timing-sensitive games. GPU to CPU data transfer can be a speed issue, as well as vertex loading for geometry-heavy games. There are still many driver issues, like the long compilation times for shaders, that cause unwanted stutter and slowness.

The HLE audio engine is good but not perfect, with some games still requiring low-level emulation to avoid glitches. Countless minor bugs, from subtle depth buffer issues to issues with non-normal floating point numbers and console glitches not being reproducible in Dolphin, still exist. On the CPU side, even with many optimizations, some games are still slow, and a few still don't even boot properly.

But improvements like these are a start. Already, many games that were far too slow to be playable on all but the fastest overclocked Haswell CPUs are accessible to a much wider audience. And while Dolphin is not and probably never will be a perfectly cycle-accurate emulator (in fact, because of DVD read times and NAND write times, no two physical consoles will even produce identical results!), it may now be accurate enough to create at least some console-verifiable replays and speed runs.

Figure 2 gives some examples of the performance improvements, measured on a variety of synthetic benchmarks and games known for being performance-intensive, between revision 2301 (late July of 2014) and revision 3378 (late September of 2014), as measured on my Ivy Bridge CPU.

Dolphin is hardly a new project; it was open-sourced six years ago and developed as a closed-source project for many years before that. It's far too easy to assume that relatively stable, mature projects don't

POV-Ray	62%	faster
LUA “binary trees” benchmark	48%	faster
Sonic Colors	39%	faster
Rogue Leader	103%	faster
F-Zero GX	110%	faster
The Last Story	38%	faster
Xenoblade Chronicles	40%	faster

Figure 2: Dolphin Performance Improvements

have much room for improvement; as new contributors, we have to resist the urge to shy away from projects like this, because often there are still vast gains to be had.

Thank you so much to Comex and Delroth for their part in these two months of incredible CPU emulation performance improvements. Thanks also to Justin Chadwick (JMC4789) for his unmatched testing and bug bisection skills across hundreds of games, as well as the monthly Dolphin progress report writeups. And thanks to all the other devs: Ryan Houdek, Skidau, Lioncash, Shuffle2, Magumagu, Calc84maniac, Rachel Bryk and many others, for their tireless work on the other aspects of Dolphin, bug fixes, and assistance with the endless ignorant questions I asked on the way to learning the inner workings of Dolphin’s CPU emulation engine.

Dolphin has been the most approachable project of any I’ve yet tried to contribute to, from the helpful developers to the relatively clean codebase. I somehow managed to become the go-to woman for the JIT in a mere six or so weeks, despite having never conceived before that I could ever contribute meaningfully to an open source project.

For anyone looking to contribute, there’s an abundant supply of interesting (or terrifying, depending on your perspective) emulation bugs just itching for someone to attack with the single-step debugger and `printf` hammer. Plus, with the brand new 64 bit ARM JIT, there are countless instructions that still need implementations—and there are certainly lots of missing optimizations for the x86 JIT too. Drop by #dolphin-dev on Freenode or drop us a pull request—any help is always appreciated!



У НАС ЛИШЕ ОДЕРЖИТЕ ФОНОГРАМ

НА БЕЗПЛАТНУ ПРОБУ.

ВИСИЛАЄМО КОЖДОМУ НАШІ ФОНОГРАФИ НА 90 ДНІВ БЕЗПЛАТНОЇ ПРОБИ.

ПРЕКРАСНИЙ ФОНОГРАФ З 24 НАВАЛКАМИ РУСЬКИХ СПІВАНОК. ЛИШЕ \$22.50.

Плати по \$1.00 місячно наолесь задоволений.

НАЙБІЛЬШИЙ СКЛАД РЕКОРДІВ У ВСІХ ЯЗИКАХ.

Пишіть по ваші прекрасні ілюстровані каталоги, котрі висилаємо цілком безплатно, або відвідайте нас в нашій складі, отвертій так в будві дни як і в недлі і свята, завжди до 10-ої години вечером. Ів. Руденський, Дер. 83.

International Phonograph Co. 196 E. Houston St., New York, N. Y.