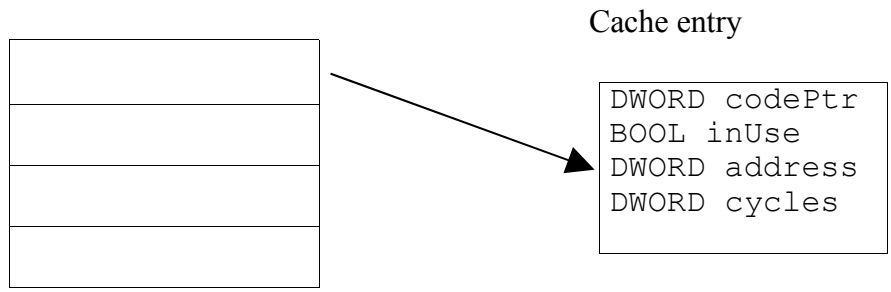
MIPS R2000 dynamic recompilation emulation
- A brief guide -

Mic, 2004
stabmaster_@hotmail.com

Introduction
This paper describes the implementation of a dynamic recompiler (dynarec) emulator for the MIPS R2000 processor.

The implementation of a dynarec compiler does not differ dramatically from an interpretive emulator. Instead of fetching opcodes and calling an interpreting function, you compile the interpretive code at run-time and write it to a buffer. The code in this buffer is then executed. The main issues to deal with in a dynarec emulator is synchronization and translation overhead. Synchronization was not considered in this emulator, since code that require cycle-perfect timing is not as common on newer processors as it used to be back in the ‘80s or ‘90s.

Translation cache
To avoid translating a block of code every time it is to be executed, a translation cache is maintained. The layout of the translation cache is as follows:



In this figure the cache only has four entries, or lines. The actual emulator uses a cache with sixty four entries. Bits 2-7 of an address determines which entry to use. The members of each cache entry are explained below:

`codePtr`	Points to the start of a memory buffer containing the generated x86 code.
`inUse`	Signals whether the block is in use or not.
`address`	Holds the address of the first (MIPS) instruction in the block.
`cycles`	Hold the number of (MIPS) cycles used by the instructions in the block.

Code translation and execution
When a block of code is to be excuted, the following happens:

1. The cache entry indexed by the current program counter is checked. If it’s not in use, some memory is allocated to hold the generated x86 code and the translation phase starts.

2. If the cache block is already in use, the `address` entry is compared to the current program counter. If they match it means this exact block has already been translated, otherwise its contents are implicitly discarded and the translation phase starts.
3. During the translation phase, opcodes are read from memory and passed on to their corresponding translation function. The translation functions output x86 code to a buffer. Once a branch instruction is encountered a counter is set to 2 (two) to signal a pending branch. This counter is decremented for every iteration of the translation loop until it reaches zero, at which point the value of the program counter is written back to memory and a `RET`-like instruction is inserted. The reason the counter starts at 2 is so that the instruction in the delayed branching slot will be translated before the loop exits.
4. Once a code block has been translated – or if it was already translated – the generated x86 code is called. Upon return from the x86 code, the block's cycle count is added to the global cycle count.
5. This is all repeated until the global cycle count is greater than or equal to a threshold set by the caller.

Speed improvements

Some possible speed improvements include mapping frequently used MIPS registers to x86 registers, and merging MIPS instructions that often occur together – such as `LUI` and `ORI`, and `MULTU` and `MFLO`.