

## CHAPTER IV

### THE MEMORY PROBLEM

#### Memory Accesses and Performance

##### Bottlenecks

As discussed in the last chapter, EPIC provides methods to reduce branch latency that results in increased computing performance. However, DRAM memory accesses can also present serious performance bottlenecks when executing blocks of code containing no branches. Clock rates for microprocessors are continuing to increase, but DRAM memory access time is not increasing as quickly to keep up with the processors. Therefore, DRAM memory access time (if measured in processor cycles) is taking longer as the clock rate for processors increase. Data caches, used in almost all current microprocessors, help to reduce performance degradation caused by this increasing main memory latency. However, in exceptional cases, hardware-managed caches can degrade processor performance to even below that of a system without a cache<sup>1</sup>. A classical example of this would be a program that moves so much data around, it continuously displaces data out of cache that it will soon need access to again.

To help combat the performance problems generated by main memory latency, EPIC provides hardware mechanisms that allow the compiler to explicitly control the data flow through the cache. The main mechanisms, which EPIC utilizes for

this purpose, are cache specifiers, data speculation, and memory disambiguation. These mechanisms can override the default hardware policies to help increase performance.

### Cache Specifiers

When an instruction is encountered to load a register from a memory location, the amount of time it will take to complete the read depends on the cache level at which the data is stored, or if not in cache, it must be retrieved from main memory. EPIC utilizes a source cache specifier that the compiler provides to tell the hardware where the desired data is likely to be found in memory<sup>1</sup>. From this specifier, the hardware can then implicitly deduce the latency associated with the memory access. In order for this technique to work in an efficient manner, the compiler must do a good job of predicting the latency of a memory load, and then convey this to the hardware using the source cache specifier. By employing analytical and cache miss profiling techniques, the compiler is able to deduce with good probability where in the memory hierarchy the desired data is located.

Along with a source cache specifier, EPIC also uses a target cache specifier for load and store operations. The purpose of the target cache specifier is to give the compiler control over which level of cache data should be stored after it is accessed from main memory. This way, the compiler can either promote or demote the data requested in memory for future memory loads or stores. This reduces the amount of misses encountered in first and second-level caches because the compiler has direct control over where the data will be stored. With this capability, the compiler can exclude data from being written to cache that will not be used frequently in the program. Also, the compiler

can remove data from any cache level if it is determined that the data will not be used anymore, or becomes out of scope.

One last cache specifier used in EPIC is the data prefetch cache (fully associative). The data prefetch cache is used when a request for data with poor temporal locality is made. A program uses a non-binding load to specify the prefetch cache as the target to load the data into. The sole purpose of a non-binding load is just to move data around in the cache hierarchy. This way, data with poor temporal locality can be successfully loaded into a special cache without displacing any contents from the first-level cache. Non-binding loads can also specify other cache levels as the prefetch target, so the loads are not limited to the prefetch cache.

### Data Speculation

As previously discussed, EPIC relies on the compiler to create an efficient plan of execution for instructions to be successfully executed in parallel. However, a compiler cannot always determine that memory references are to actual distinct locations, and must assume that memory aliases exist. The term alias refers to a condition that two pointers with different names could share the same memory location. A classic example of this is a C function with the format:

```
void foo(int& x, int&y);
```

Because `x` and `y` are address parameters in the function, it is not known at compile time whether they point to different memory locations, or the same (in which case they would alias each other). Compilers must assume that aliases do exist, and this assumption can result in a breakdown of efficiency in the compiler-generated plan of execution. To help

deal with this issue, EPIC employs a method called data speculation, which allows the compiler to generate a plan of execution where it would not matter if memory references were discrete, or aliases.

Data speculation breaks up a load operation into two different components: the data-speculative load and the data-verifying load<sup>6</sup>. First, the compiler schedules data-speculative loads well in advance of accessing potential alias memory locations. This way, the load can begin in a timely manner within the plan of execution. Then, the compiler will schedule the data-verifying load after the potential alias memory accesses. Hardware is used to detect if an alias was actually encountered. If it is determined that an alias did not occur, the read from memory during the data-speculative load is correct, and no action is required from the data-verifying load. The execution of the program then proceeds without further issue. However, if an alias was determined to have occurred, the data-verifying load operation performs corrective action. First, the entire load operation is re-executed because the desired data could possibly be stale. The processor is then stalled momentarily so that the desired data will return in time for the re-load, or any other subsequent loads in the plan of execution.

### Memory Disambiguation

Another issue which must be addressed is determining flow dependency between memory operations. Memory operations are flow dependent on one another if one operation writes data to a location, and other operations might potentially read data from the same location. In parallel and pipelined processor implementations, it becomes quickly apparent why flow dependency is important. If it can be proven that memory

operations will not access the same memory locations, they are said to be independent. However, if this dependency cannot be proven, then these operations are said to be ambiguous. When a compiler encounters an ambiguous set of memory operations, it makes an educated guess as to whether or not a dependency exists, using previous statistics and trends to help make the decision.

When a load operation occurs after a store operation, this situation poses a problem if there is a dependency between the two operations. If the compiler has guessed that there is no dependency where one actually does exist, some type of action is required to recover from the situation. One possibility would be to embed this code in software using compare and branch instructions (for example, if pointers “x” and “y” are equal). However, this is a less than desirable method due to code bloating and performance degradation. Instead, the hardware is used to deal with ambiguous situations. For this purpose, a Memory Conflict Buffer (MCB) is employed, and preload and check instructions are used from the ISA<sup>6</sup>.

When dependent memory operations cause a clash, all data operations must be re-executed. The MCB first records address information for each preload instruction. The addresses of any subsequent stores are then compared with the addresses in the MCB to determine if any conflicts exist. If a conflict is detected, it is recorded in the MCB. When the check instruction is executed, the hardware will check the MCB for conflicts, and transfer control to a block of correction code supplied by the compiler if conflicts were detected.

The MCB is constructed of a preload array and a conflict vector. The preload array consists of register numbers (these are address registers which correspond to the memory locations) and valid bits to indicate whether entries contain valid data. The conflict vector contains information for each register number stored in the preload array. The elements of the conflict vector are simply line numbers to the preload array (to access register information), and a conflict bit indicating whether that entry in the preload array was determined to have a conflict.

One final point of interest is the check instruction. It is important to note that the correction code for dealing with memory conflicts is generated by the compiler. In the ISA, the format for the check instruction is *check Rs, Label*. *Rs* is the register to look up in the MCB and determine if a conflict exists. *Label* is the address of the first instruction for the correction code supplied by the compiler. If the conflict bit is set for this register, execution picks up at the address specified by the label, and the conflict bit is reset. Otherwise, the branch falls through as no error handling is necessary.