

CHAPTER III

THE BRANCH PROBLEM

Branch Intensive Applications

There are a relatively large number of applications which are very branch intensive, meaning they have frequent transfers of control to different blocks of executed code. These could be either conditional (e.g., if-else statements) or unconditional branches (like the goto statement). While being a very logical way to think about writing code, it does create performance degradation during the execution of the code. The problem stems from the fact that there can be a considerable latency in code execution because of branch instructions.

Expressed in terms of processor cycles, branch latency will actually *increase* as clock speeds increase. The reason for this increase is because branch operations have a memory access latency that begins when the program counter is loaded with the branch address, and ends when the first instruction at the branch target address is executed. Let us now briefly examine why this latency occurs.

When a branch instruction is encountered, several actions happen that cause the branch latency. First, the hardware computes a branch condition and forms a target address based on the result of the condition. After this, the processor fetches instructions from either the fall-through of the branch (the next sequential instruction), or from the memory location of the computed target address. Finally, it then decodes and executes

the appropriate instruction. Even though branch instructions are usually specified as a single operation instruction, the instruction at the target address may not be in cache. When this happens, the instruction must be fetched from main DRAM memory. Also, there may be a number of instructions in the pipeline which must be nullified if the prediction is incorrect.

While it may seem from the surface that this latency might not be such a detriment to performance, it is important to note that it is not possible to overlap many operations with branches. Because of this, performance greatly suffers with branch-intensive applications. To overcome this, EPIC utilizes unbundled branches, predicated execution, and control speculation to help increase performance on such branch-intensive applications. These will be described in the following sections.

Unbundled Branches

When a branch instruction is encountered in the EPIC architecture, it is “unbundled” into three main components¹. The first component is a “prepare to branch”, which will compute the effective address of the branch target. The next component is a compare, where the condition of the branch is computed. Finally, the last component is the actual branch, which specifies when control is to be transferred by loading the target address into the program counter.

In the prepare-to-branch and compare stages, it is the responsibility of the compiler to schedule prepare-to-branch and compare operations. Usually this is done well in advance of the actual branch so that information can be promptly acted upon by the hardware. Upon receiving the prepare-to-branch information, the hardware will then

speculatively pre-fetch instructions from the target of the branch. After this is done, the hardware will execute the compare and determine if the branch will be taken. If it is determined that the branch will not be taken, the hardware will then discard any unnecessary speculative pre-fetched instructions, and launch non-speculative pre-fetches. Finally, if the branch is taken then it will start executing the pre-fetched instructions from the target location.

Predication

While unbundled branches aim to reduce the inherent latency generated by branch instructions, significant penalties can still exist in branch-intensive applications if there are no other instructions that can be executed in parallel. Another dimension of branch instructions, which carries a significant penalty, is branch misprediction. In this case, the speculated target address of the branch proves to be incorrect, and corrective measures must be taken in order to compute the correct address. This misprediction usually carries a hefty branch penalty, because of nullified instructions and rescheduling. In order to help overcome this associated penalty, the reduction of the frequency of branch instructions becomes important. This is where predication comes in to play, to help eliminate branches when possible.

Predicated execution uses Boolean logic to compute a “predicate” associated with an instruction⁵. When a compare operation is encountered, a Boolean value is computed upon the operands, and usually stored as a bit in a predicate register. The predicate value is computed to be true if the comparison evaluates to true (for example, if the contents of register 4 is less than the contents of register 6), otherwise the predicate is

false. Then, simple if statements can be used on the computed predicate instead of branching to different segments of code.

To help illustrate this concept, consider the following code fragment:

```
int x=2;
int y=6;
if (x < y)
    x = x + 10;
else
    y = y + 2;
```

If predication is not used, the above code segment would translate into assembly

language which would look something like this:

```
load immediate x, 2
load immediate y, 6
branch on greater or equal x, y, next
add immediate x, x, 10
branch to final
next: add immediate y, y, 2

final: .....
```

Here we can see that no matter which case occurs during the comparison, one branch will be executed. By applying predication techniques, we can eliminate the need for branches.

Consider now the target code using predication:

```
load immediate x, 2
load immediate y, 6
p = x < y          ←P is a particular bit in the predicate register
if (p): add immediate x, x, 10
if (!p): add immediate y, y, 2
```

The above example illustrates the “if-conversion” method of predication, because it is a direct translation of an “if” statement. By eliminating the need for branches in this case, any branch penalties which would have been incurred normally are now gone completely.

Predication is especially effective when branches are unbiased (meaning it is no more likely to go one way than another), and conditional clauses contain relatively few operations.

Control Speculation

Control speculation is a mechanism, which helps to overcome main memory latency because of branch instructions. Normally, if an instruction to load a value from memory were to occur as a result of a conditional branch being taken, the load would not be initiated until control of the program had reached that instruction. Reading from main memory carries a latency usually around 10 to 20 CPU clock cycles, thus reducing the performance of the code. An example of this would be if the data requested is not in any of the cache levels. In this case, a cache miss is encountered, and fetching from main memory always takes more clock cycles than fetching from cache.

Control speculation aims to help reduce the latency generated from main memory reads by allowing load instructions at the target branch address block of code to be scheduled before a conditional branch is encountered in a program⁵. This way, the hardware can begin to speculatively load the information from memory so that when the branch is reached, the required information has already been read from memory. While this greatly enhances instruction level parallelism, it can create problems when dealing with exceptions.

The problem with exceptions arises when an incorrectly speculative loaded block of code generates an exception, such as a page fault. In this case, an exception is reported to have occurred, even though the code generating the exception would not have

been executed in the program (if the execution of this code block proved to be incorrect). To overcome this obstacle, hardware assistance is needed to help handle such exceptions. EPIC avoids erroneous exceptions by employing speculative opcodes and tagged operands. If a speculatively loaded operation generates an exception, it is not immediately acted upon. Instead, it is tagged by hardware by setting a flip-flop for later use. Then, if the branch outcome determines the speculation is correct, the bit is turned off and the exception is acted upon. If the speculative load is not used, the exception is simply ignored.

To summarize, EPIC utilizes a number of methods aimed at reducing the latency generated by branches. Some are aimed at increasing the performance during branch instructions (branch unbundling, speculative loads and instruction fetches), while some are aimed at eliminating the branching statements completely where possible (predication). Also, all of the above listed methods rely on the compiler for information, which is a key element of the EPIC architecture.