CHAPTER V

LIMITS OF EPIC

Static Scheduling

EPIC employs many advanced and complex methods to help increase its computing throughput. However, even with all of these innovative methods, there are some inherent limitations. Some of these limitations play a role in the current implementation of the architecture, while some will eventually inhibit further advancements in EPIC over the long term. First off, EPIC is based on static scheduling, meaning that the compiler dictates the ordering of instructions to be executed in parallel. However, run-time events are dynamic in nature and cannot be accounted for during compile-time. Even with very sophisticated methods of profiling, analyzing, and predicting, the compiler still cannot take full advantage of the dynamic nature of run-time events. For example, the cache hit/miss ratio may vary for a memory load instruction. By profiling, the compiler then might schedule other loads too far in advance where not necessary, thus producing worse performance results.

Penalties from Code Bloating

EPIC performance can suffer serious penalties from code bloat. As described previously, EPIC allows a load instruction to be scheduled far in advance of when the actual data is needed. This is to help reduce the amount of latency time a
memory access generates. However, if a considerable latency difference exists between accessing cache memory and main memory, the amount of instructions the preload must be scheduled before can become impractical. For example, consider an EPIC processor running at 2 gigahertz that can issue 6 instructions in parallel, with a second level cache latency of 10 clock ticks, and a main memory latency of 105 clock ticks. In accessing the cache, we would need to insert our preload $6 \times 10$, or 60 instructions before the actual data is needed to assure it is loaded in time. This is a considerable amount of pre-scheduling, but it can be accomplished using code transformation, loop un-rolling, etc. However, the main memory latency becomes a real issue, as it would require $12 \times 105$, or 1260 instructions to schedule a load in advance of. In this situation, the code would become significantly bloated in order to accommodate such a load in advance. The limits for EPIC in this area are reached when the penalties from code bloating become higher than the gains the EPIC architecture provides.

**Pointer Manipulations**

Another area where EPIC performance can really suffer is during extensive pointer manipulations, like working with many dynamic data structures¹. The problem emerges because many cache misses can occur when searching over a largely populated dynamic data structure. As previously discussed, the difference in latencies between cache and main memory can be fairly significant. Throw in many accesses to main memory, and performance will begin to suffer significantly. When traversing, say, a linked list, usually a pointer manipulation is embedded within some sort of looping structure. If the manipulation is generating many cache misses, this leaves the compiler to
try and insert useful work within the loop so that the latency will not just generate idle
time. However, it is extremely difficult even for the most advanced compiler to insert
useful instructions within such a loop.