Patrick Trinkle
CMSC611
Extra Credit

"Compressed Lossless Texture Representation and Caching"

1)

The more immediate parallel to what we've been studying in class is the idea that memory is slow, so any method to require less memory access is a good thing. In this case and in our class this idea of slow memory is tackled by more or less avoiding the problem by supplying superfast caches. A modern CPU utilizes a minimum of 2 levels of cache before main memory is hit. Each level of cache has slightly different characteristics, which more closely model its specific purpose. The higher level of cache is often smaller and faster than a lower level. CPU caches tend to be actual data, whereas these individuals have done something rather clever. Instead of structuring progressively larger caches in a chain, each holding strictly data (possibly in a subset to superset framework or mutually exclusive sets), for their purpose they have actually structured their data into a B-Tree of compressed data blocks. With this they have 1 cache of actually decompressed texture tiles, the Tile cache. Then two other caches closely modeling the B-Tree stored in main memory: an index cache, which holds decompressed index blocks, and a leaf cache that is further down. Interestingly enough caching the recent blocks from a B-Tree can maximize efficiency with the already rather efficient balanced tree. This caching also implies that the data texture tiles are repeatedly used in a non-random method. This is also demonstrated by the usage of the Least Recently Used replacement scheme for the 3 caches. Another aspect we glanced over that they're taking advantage of is multiple in-flight memory requests; as these will reduce the overall access latency. The initial access latency might be hit once for multiple requests, each carrying the normal read latency and transmission time.

Another large problem is bandwidth. They have tackled this partially with data compression. The paper indicates that compressing instructions is not uncommon; and such compressed instructions are read-only—therefore only ever needing to be decompressed. This idea of read only memory, versus read-write follows what we've studied in class as well. Also the expansion of the idea to use this read-only, read-write paradigm to actually enforce security, such as non-execute, etc, was discussed. Their paper detailed the decompression overhead to only 2 cycles. With memory latency considerably higher, it might be worth investigating compression for more than read-only memory sections (instructions) or to consider allowing compilers to request data pages that are read only, and therefore can more easily be accessed with regard to compression/decompression. For an example if you can compress the program data's image in main memory, the decompression execution time should prove considerably faster than the actual access latencies—which is pretty cool. If program data is ASCII there are many

fairly strong compression algorithms, which likely could be implemented in hardware.

2)

Actually forgot there was a part 2 until too late to get it done.