Although it is certainly true that requiring data to be persistent is a major difference in the architecture, and that accessing data through the Data Store will introduce considerable latencies into the architecture, we believe that the approach we have taken will be more than competitive for a number of reasons. First, we believe that we can make the difference between accessing data in main memory and accessing it through the data store much smaller than is generally believed. Although conceptually every object that lasts longer than a single task needs to be read from and written to persistent storage, the implementation of such a store can utilize the years of research in database caching and coherence to minimize the data access latencies incurred by the approach.

This is especially true if we can localize the access to particular sets of objects on a particular server. If the only tasks that are making use of a particular set of objects are run on a single server, then the cache on that server can be used to give near main-memory access and write times for the objects (subject to whatever durability constraints need to be met). Tasks can be identified with particular players or users in the virtual world. And here we can utilize the requirement that data access and communications go through services provided by the infrastructure to gather information about the data access patterns and the communication patterns taking place in the game or world at a particular time. Given this information, we believe that we can make very accurate estimations of which players should be co-located with other players. Since we can move players to any server that we wish, we can maximize the co-location of players in an active fashion, based upon the runtime behavior that we observe. This should allow us to make use of standard caching techniques that are well-known in the database world to minimize the latencies of accessing and storing the persistent information.

This sounds very much like the geographic decomposition that is currently used in large-scale games and virtual worlds to allow scaling. There, the server developers decompose the world into areas that are assigned to servers, and the various areas act as localization devices for the players. Players in the same area are more likely to interact than those in other areas, and so co-location on a server is enhanced. The difference is that current geographic decompositions occur as part of the development of the game and are reified in the source code to the server. Our co-location is based on runtime information, and can be dynamically tuned to the actual patterns of play or interaction that are occurring at the time of placement. This is analogous to the difference between compile-time optimization and just-in-time optimization. The former seeks to optimize for all possible runs of a program, whereas the latter attempts to optimize for the current run.

We don't believe that we can make the difference between main-memory access and persistent access disappear, but we also don't think that this is necessary in order to end up with performance that is better than that of infrastructures that make use of main memory. Remember that by making all of the data persistent, we are enabling the use of multiple threads (and therefore the multiple cores) within the server. Although we don't believe that the concurrency will be perfect (that is, that for each additional core we will get complete use of that core), we do believe (and preliminary results encourage this belief) that there is a