Caching in on Aggregation (extended abstract)

Michael Ferguson, Laboratory for Telecommunication Sciences

The Chapel programming language features implicit communication, which has many advantages – such as allowing an implementation to separate data distribution from algorithm. One disadvantage of this model is that it can be difficult for a Chapel programmer to create large and efficient network messages. New compiler optimization can solve this problem in some cases, but such optimizations are difficult to write and only apply in certain circumstances.

Take the following program as an example. Suppose A is a remote array of 64-bit ints, and f is a computationally-intensive procedure.

```
// naive version, assume A is distributed
for i in 1..n {
   A[i] = f(i);
}
```

```
// whole-array assignment version, A is distributed
var B:[1..n] int;
for i in 1..n {
   B[i] = f(i);
}
B = A;
```

With the current compiler, each of the assignment operations in the loop in the naive version will generate a separate small message communicating a single 64-bit integer value, wait for it to be received and acknowledged, and then move on to the next value. The situation for reading from A is analogous.

However, if Chapel's whole-array operations are used, the program can be further optimized to the whole-array assignment version listed above. There are two problems with this approach. First, the obvious solution here might run out of memory if n is very large; in that case a programmer will need to instead create a more complicated tiled loop in order to bound the new local memory used. The second problem is that it hurts the Chapel promise of separating algorithm from data distribution since this modification has to be made in the algorithm code itself - where we would like the body of the algorithm code to be able to ignore whether or not A is distributed.

In addition to the above difficulty in aggregating communications, a Chapel programmer has a difficult time providing explicit prefetching hints to the runtime. In the code below, the version on the left is the naive version, and the version on the right might be a version optimized for a single-processor, using the prefetch instructions that performance-minded programmers are already familiar with.

```
for i in 1..n {
   sum +=   A[f(i)];
}
```

```
for i in 1..n {
   prefetch(A[i+8]);
   sum += A[f(i)];
}
```

Wouldn't it be nice if the same optimization technique (adding prefeteches) also applied to the distributed memory version, where the prefetches would cause the remote data to be fetched before it was needed?

We have developed a cache for remote data that is capable of aggregating writes, automatic prefetch for strided access, and prefetching data based on programmer-provided prefetch hints. In this talk, we will describe in detail the design of this cache and discuss how various design choices impact performance and the amount of communication. In addition, the most difficult part about adding such a cache for remote data is ensuring that Chapel programs still operate according to a reasonable memory consistency model. Through the process of developing this cache for remote data, we believe that we have the experience necessary to describe more fully the Chapel memory consistency model. We will informally describe the desired memory consistency model and then informally show how our caching scheme conforms to that model.