Clouds and manycores share common challenges. Most of those challenges were identified and pointed out in the FOS paper. They mentioned scalability -- it is problematic to scale computation to take advantage of newly available units capable of computation. Existing programming models and, equally importantly, the toolchains supporting them, are finely tuned to the single processor, von Neumann programming model. Scaling that to multiple independent processing units is a challenge for both multicores and clouds.

Managing elasticity of demand and supply. What does it mean. When I think of elasticity, I think of both spacial and temporal aspects of it. Demand elasticity could refer to spikes in demand for resources, such as a surge in web server requests. Responding to that should ideally be the elasticity of available resources, accommodating demand spikes.

Fault Isolation is certainly a shared concern. In the context of cloud computing, this has resulted in the popularity of virtualization -- many refer to this technology as the enabler of cloud computing. It is uncertain what the solution is for multicores -- but it is fairly obvious to me that we can re-use the benefits of virtualization for multicores as well.

Programmability -- as mentioned, current models of computation are fine-tuned to sequential execution. A fundamental redesign of both programming models, and the toolchains around them is called for to take advantage of distributed resources available both in cloud computing and on a manycore chip.

Finally, our research is poised to contribute an additional item to this list -- namely workload scheduling.

To recap, we’ve seen that there will be a ton of execution threads to schedule. Factored OS is a recent body of work that pointed out space-division multiplexing replacing time-division multiplexing, with the number of threads to schedule increasing with the number of cores (at least).

What space-division multiplexing means is that the thread is going to spend longer periods of time running on the same core. It will no longer be swapped out every 10ms. As a result, ensuring an OPTIMAL RELATIVE PLACEMENT of threads and data is becoming increasingly important.

Now, I would like to place an emphasis on optimal relative placement here. Naturally, one way to accomplish this is to concentrate on data placement. But this is not the only way. To optimize thread-to-memory communication costs, you can move around data or you can move around threads.

Problem formulation
There are two major terms in this equation:
(a) the sum of weighted edges in the logical application graph
(b) the count of links contended for by the mapping of edges onto the NoC paths.

L(map(e_ij)) is the set of links on the onchip network that should be traversed for core I to talk to core J once mapped. In other words, each of the edges translates into a routing path on the onchip network. We take the number of those links, weight it by the weight of the original edge and do that for all edges and sum it all up. This is the cost of communication between the cores.

The contention component is somewhat less well-presented in the paper. The use of math is quite liberal. For example, in equation one we’re missing a summation.

The intersection here refers to the following. We pick two unique edges, we map them to the onchip route, translate those routes into sets of router-to-router links and intersect those sets. The intersection reveals the set of contended links. If we count them all up -- it will give us an idea of how much contention may result from a given map() function.

Finally, there’s a constraint. We take a given link l_k, we have a binary variable that is set if this link is on the path between tile mapping of c_i and tile mapping of c_j. If it is, we increment the bandwidth consumption on that link by the bandwidth of the edge between c_i and c_j. And we do that for all edges. This sum of edge bandwidth contributions to a given link l_k must not exceed the maximum bandwidth capacity of the link.

m variable is a boolean variable that equals to 1 if core c_i maps to tile r_s.
the p variable is also a boolean variable that is equal to one if three conditions hold:
* c_i -> r_s
* c_j -> r_t
* c_i, c_j is an edge

relaxing the ILP problem into LP removes the constraint that these variables should be boolean, so they can be completely arbitrary. And step 1 chooses the core with the largest value to schedule

The important thing to notice is that, although the formulation of the problem incorporates the contention-awareness and minimization, the heuristic algorithm does not.