1 Detailed Description of the Proposed Research

We now describe the key goals of our research in more detail.

Research goal 1: *Identify the right specification of NVMM.*
All the correctness properties discussed in previous section focus on keeping the object or the data structure consist in case of a failures. However, none of which address the problem of the returned value by the failed operation. Consider for example a write operation. Since write is atomic it obviously satisfies strict linearizability - either the write took effect before the crash failure, or it did not. However, once the process is recovered it has no knowledge which of the two is the case, since the acknowledgement returned by the write is saved to the local memory of the process, which is lost due to the failure. Assuming the process is in a middle of executing an operation which it desired to complete once it is recovered, it must know whether the write took effect (in which case it can continue and execute the code), or that it did not (in which case it needs to rewrite). Skipping the write is not an option, as it might cause an undesired behaviour, which in turn might violate the requirements of the object.

For this case, we present a different approach called *recoverable response linearizability* (RR-linearizability). In case of a failure, once the process is recovered it is "aware" of the failure, in the sense that it runs a designated recovery code. In a sense, this recovery code "extends" the interval of the interrupted operation. We require that the pending operation linearizes between it's invocation and the end of the recovery code, or aborts. In addition, in case of linearization, the returned value of the operation is known to the process. Notice that this definition introduce a relaxation which is not intuitive neither trivial - in case that a process operation took affect before the crash, but the response of it is lost due to the crash, we allow the process, upon recovering, to extend the interval of that operation using the recovery code, such that at the end of it the process knows the returned value or aborts. This gives the process the flexibility to move the linearization point to after the crash, if needed. In addition, although the definition allows a process to abort the interrupted operation, it is equivalent to a definition in which a process must complete the operation - once a process knows the operation was aborted, it can reissue it.

There are also various flavours for the model. We first consider the standard abstract model, a simple well-studied model. The system consist of a set of asynchronous unreliable processes $P = \{p_1, \ldots, p_n\}$ that communicate by applying operations on shared *base objects* that support atomic operations such as reads, writes, and read-modify-write primitives. All base objects are non-volatile, i.e, in case of a crash the values stored in the base objects are preserved. In order to implement a more complex object, such as stack and counter, each process is supplied with access procedures that simulate each operation on the implemented object using operations on base objects. Each process is

equipped with a local volatile memory. We assume no caches, thus writes goes directly to the main memory. In addition, upon a process crash, we assume that the system control knows at which line of the code the failure occurs, or more precisely, at the course of which atomic operation the process failed. However, there is no induction whether the atomic operation took affect or not before the crash.

Another point to consider is how to model failures. Clearly, the whole system can fail, but we will consider a more relaxed model in which individual processes can also fail.

What is the right progress condition?

Research goal 2: *Derive algorithms, lower bounds and complexity tradeoffs*

First, we look for RR-linearized implementation for base object. This direction attract our focus thanks to the following observation: a system which supply recoverable primitives can be used to implement any object in a recoverable way. In case of a crash the process will simply recover the last atomic operation it executed before the crash. In case the operation is aborted, the process will reissue it. After that, it can continue and execute the remaining code safely.

Some base objects have an RR-linearzied implementation. For example, read has a simple implementation - the process will simply abort the read upon its recovering. Since read does not affect the rest of the processes, this can be done safely. However, not every primitive has an RR-linearized implementation.

Test-and-Set (TAS) is a known object supported as a primitive in many modern architectures. A TAS object contains a binary value initialized to 0, and support a single atomic operation TAS which set the value of the object to 1 and returns the old value. We now show that there is no bounded wait-free RR-linearizable implementation for TAS even for two processes, assuming the system support only read, write and TAS primitives. We give a high level description for the proof. The construction also implies that any wait-free implementation need to use an unbounded number of TAS objects.

Assume by contradiction that there is such an implementation. Denote the set of processes by $P = \{p, q\}$. First, notice that the code implementing the TAS operation must contain a TAS object O_1 which accessed by the two processes and determine their output, otherwise we can solve consensus for two processes using reads and writes only, contradicting to FLP result [*]. We let both processes run to the point where they both about to access O_1, and then perform a TAS one after the other, followed by a crash of p only.

Once p is recover, it needs to execute a recovery code, since the value returned from O_1 is lost. We look at a solo run of p starting from this point, and consider the possible return values. If p returns 1, then we look at the execution where q was the first to access O_1 before the crash. If p returns 0 or abort, we look at the execution where q is the second to access O_1. Notice that after the crash, p does not able to distinguish between the different executions, thus it performs the same run after any of these. TAS can be seen as a consensus for two processes, the one that returns 1 wins. In any case we get a

bivalet configuration - both processes want to return the same value (we consider abort as 0), while only one can return 1 and the other must return 0. Thus, it must be that both processes access another TAS object O_2. We can continue and use the same argument, constructing an execution where more and more different TAS objects are accessed.

Future research will focus on implementing additional primitives, as write and compare-and-swap (which have an RR-linearized implementation). In addition, the complexity of those implementations is another research direction. Is there a tradeoff between normal operation and recovery? Can one achieve the best on both? Is there a tradeoff between space and time for such implementations?

Another research direction is looking for implementation for special object. Although the approach of using RR-linearized base objects is universal, it might be that certain objects as stack or counter can be implemented directly more efficiently without the use of RR-linearized base object, or with reducing the use of such base objects.

For RME, there are still several open problems. Is $logn/loglogn$ RMRs optimal for CC model? What can be done in the DSM model?

Research goal 3: *Incorporate novel architecture aspects.*

The model presented in goal 1 is very simple and abstract. The model is used in order to investigate the right specification for NVRAM systems. As described earlier, even under this simple model there are impossibility results, and non trivial implementation. However, the model neglect some of the complexities of real world systems. Current architectures contains volatile cache in addition to main memory. In such systems, write does not goes directly to the main memory, but rather to a cached copy. Therefore, in case of a failure, the write can be lost due to the cache lost. Moreover, such systems introduce reordering of instruction. Fence instruction is used to prevent such reordering. While it guarantees that all writes preceding it will become visible, in practice it does not necessary means writes goes to the main memory. In such case, a crash after a fence instruction still may cause a lost of writes.

A more realistic model should take the above architectures and the complications they present into account. Also, real world system may contain a combinations of NVRAM and ordinary memory, and a new model needs to be devised for such systems. One more practical property that NVRAM has is a big difference between the cost of a read and the cost of write. So far the research has neglected this property, treating both operations as the same cost. Weather we can reduce the number of writes in certain implementation, even in the cost of additional reads, is another research direction.