Floating point is widely used in different applications from high-performance computing to finance. Due to the variety of numerical errors that can occur, floating point programs are hard to debug and test. In order to minimize the chances of numerical errors, developers tends to use highest precision throughout the program. However, this practice drag down the performance significantly. In the paper *Precimonious: Tuning Assistant for Floating-Point Precision*, the author provide a dynamic analysis tool – Precimonious to help developers tuning the precision of floating-point program. Precimonious perform a search which is based on delta-debugging algorithm on the variables and theirs types of floating point programs trying to reduce their precision while satisfying the accuracy and performance constrains. The author evaluate Precimonious on several functions from GNU Scientific Library, two NAS Parallel Benchmarks(SNU NPB Suite, C translation), a arclength program, a Simpson's rule program, and infinite sum program. The result shows that by using Precimonious to reduce precision, the performance improved as high as 41%.

The main contribution of this paper is introduce a automated approach for recommending variables' types so that leads to better performance in floating point programs. Instead of manual turning: exploring the variable types, apply change and check result/performance, Precimonious automated the process which greatly alleviate the cost of turning the numerical programs. Comparing to the work from Lam et al. which tries to use single precision instructions to replace double precision instructions, Precimonious considers the accuracy and speedup together as a goal when reducing the precision. Furthermore, Precimonious changes are at bitcode level of the source code which is easier to map to the source code.

The limitations of the work are: 1) The type configuration suggested by Precimonious need to be mapped to the source code of the program which is been done manually. When apply Precimonious on large programs or applications, the amount of manual work could be very large. As suggested by the author, developing a front-end tool which directly modify the source code instead of perform lower-level program transformation would be more useful. 2) Precimonious does not take into account how variable interact with each other. When some variables are often been used together as operands, changing one of them would lead to additional casting that could slow down the program. Group variables together by performing an initial static analysis before the searching could avoid such situation.

The future works of Precimonious are:

1) Extend Precimonious to other languages beside C. Since Precimonious uses LLVM IR, it can be easily extended to other LLVM frontend languages such as Fortran, Swift, Julia, Ruby, and Scala which are widely used in all applications.
2) Precimonious paper is done on Intel Core i7-3700K 3.5Ghz CPU, it would be interesting to see how well it works on floating point specific hardware(certain GPUs).
3) The paper is done in 2013, LLVM and Clang has evolved several versions. In order to use Precimonious on recently programs, Precimonious need adapt to the different versions of LLVM and Clang.