Arend, Zach
DuFrene, Andy
CPE 431-01
6/10/15

Programming Languages II Paper

Overall Architecture

In order to generate x86 assembly code, we ran the mini file through a series of stages that would provide a list of assembly instruction data types, which we wrote to a file according to x86 syntax. The first step in our compiler was to run the input file through a generated parser, which created an abstract syntax tree, using data types that we had previously designed. We had datatypes for most of the non-terminal symbols, and some of the terminal symbols.

After we had the AST, we traversed it in order to type-check statements and functions. Due to our use of a lazy-evaluation language, we couldn't throw exceptions within our type-checking, but needed to return an error value if type-checks failed. The next step was transforming the AST into an intermediary ILOC language, that was independent of the final architecture of the assembly. We created this ILOC by iterating through the list of statements in each function, and creating basic blocks made of ILOC statements directly mapped from statements in the AST. A basic block is a piece of code where every instruction in the block is guaranteed to run if the first instruction is run. All of the basic blocks within a functions were also linked together based on the code paths to create a Control Flow Graph. Because each CFG represented a single function, we transformed our AST into a list of Control Flow Graphs.

Next, we applied at most three optimizations on our iloc instructions depending on passed in flags. The first optimization performed was Local Value Numbering, where we tried to eliminate redundant expressions with assignments if the value was previously computed. The next optimization applied was Copy Propagation, which ideally replaced unnecessary assignments with uses of the original value. Finally, we applied Useless Code Removal, which removed any ILOC instructions that were not required in producing the final output of the program.

The final step in our compiler was to transform the ILOC instructions into x86 instructions, and manage the call stack. In order to perform this, we first mapped each of our ILOC instructions to one or more x86 assembly instructions, represented as new data types. Throughout all of our instructions thus far, we had used virtual register values, so the next step was to replace our virtual registers with real registers using register allocation. This was performed by creating an interference graph per function that showed which virtual registers were in use at the same time, meaning that those virtual registers could not be mapped to the same real register. Next we deconstructed the interference graph and assigned real x86 registers to each virtual register in a lookup table. Finally, we replaced every use of a virtual register with the correct x86 register, based on the generated lookup table.

After all of these steps, we were left with a list of assembly datatypes, which we wrote out to a file that could be assembled with an x86 assembler to create an executable.

Data Architecture

We had three primary data types that were used to represent our program at different times in compilation. First, we used a datatype to represent our program in Abstract Syntax Tree form. Because we used a functional language, we were able to create data types that mapped pretty directly to the given EBNF grammar. We created datatypes for the nonterminals in the grammar, with data constructors composed of the corresponding mix of terminals and nonterminals. This resulted in a single program value that was the root of the entire AST.

Second, we created a datatype to represent our ILOC instructions. The ILOC datatype had data constructors corresponding to the individual iloc instructions that we used (add, sub, div, mov). We used unbounded integers to represent our virtual registers, and strings for the labels of each basic block in the CFG. We were also able to create a show instance (equivalent to 'toString()' in java) for our ILOC datatype that allowed us to easily print our list of ILOC instructions.

Third, we again created a datatype to represent our x86 assembly instructions. Almost identically to the ILOC representation, we used a single data type with data constructors corresponding to the x86 instructions that we used. Again, we created a show instance that let us easily print our data in the correct format. In order to represent registers, we had to use a more complicated data representation. We first defined an AsmReg data type with data constructors for each x86 register, and data constructors corresponding to stack variables, and virtual registers that had not yet been allocated. Then we needed different instruction argument types that would take an AsmReg and some other data to create the appropriate argument (e.g. mov can take a register, an immediate value, an address, or a register and an offset, so we had to create a datatype with constructors for each of those cases). We ended up with four such datatypes that represented the possible arguments for assembly instructions. Labels were again represented using strings.

In order to represent graph data, such as the Control Flow Graph and the Interference Graph, we used a haskell library that represented the graph as an adjacency list using an array of lists of Integers, which represented the vertex of each graph. In order to store more information in our graph, such as the basic block of each vertex in the CFG, we needed to create a tuple of the graph with a hashmap mapping vertices to basic blocks. We also created some very dense and messy functions to "improve" our given graph interface.

Optimizations

Local Value Numbering

The first optimization run in our compiler, (and the last optimization implemented,) is Local Value Numbering. The goal of LVN is to reuse an expression if it has been calculated previously, by replacing the second calculation with a copy of the original register that the expression was stored in. This optimization required two passes over each basic block; the first pass walked the instructions top to bottom, and created a number for each register and expression that was not previously computed, and replaced any redundant expression with a mov instruction from the virtual register that the expression was mapped to. The second pass involved walking the instructions backwards, and adding and removing from a list of active virtually numbered instructions. If we found a mov into an lvn register, then we added that local value number into a list of numbers that should be preserved. If we found an expression that was mapped to a local value number that was in our list, then we added a mov instruction from the target register of the expression to the lvn register. Any code rendered useless by this optimization was ideally removed by useless code removal.

Copy Propagation

The second optimization that gets run by our compiler is Copy Propagation. The purpose of this optimization is to remove redundant assignments by replacing uses of the new register with the original register, when applicable. This is performed by walking through a whole function and finding any copies between registers. Next, and subsequent uses of that register is replaced with the original register that was copied, until the original register is overwritten with a new value. Then, the copied register remains, and the original copy was necessary. Just like LVN, Copy Propagation should ideally present opportunities for Useless Code Removal to optimize effectively.

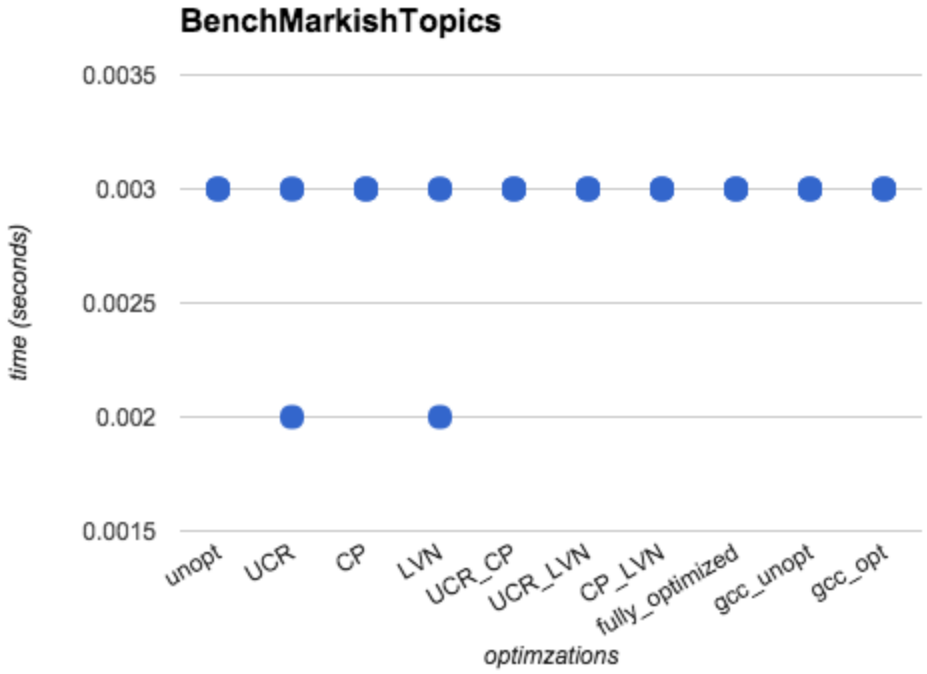
Useless Code Removal

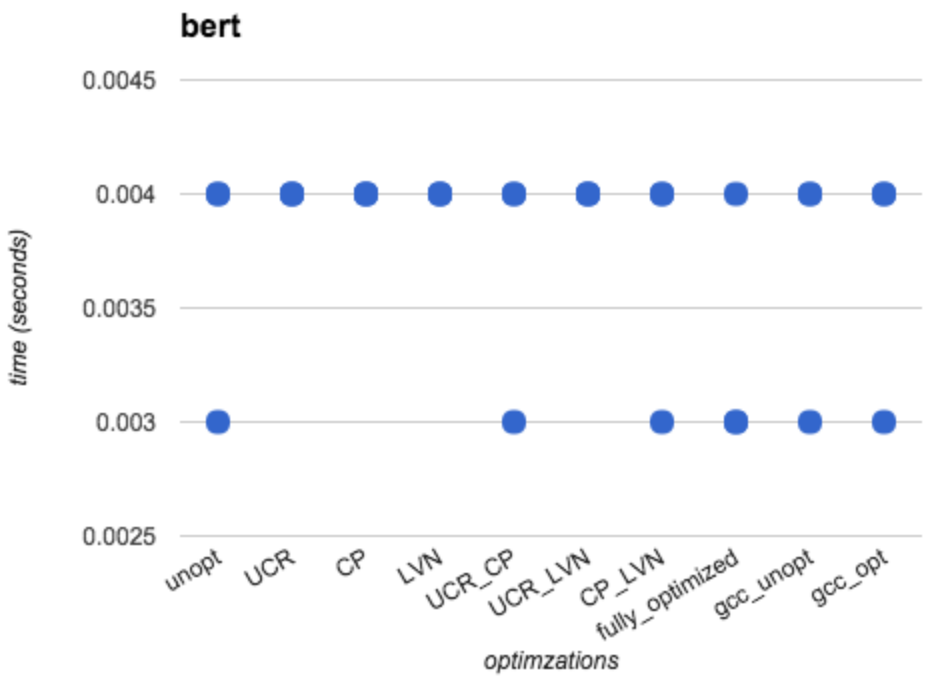
The final optimization run is Useless Code Removal. The goal of this optimization is to find instructions that are not used in calculations that could affect the output of the program. This is done by first locating instructions that are “critical” to the program’s execution, and “marking” them. These instructions include print statements, function calls, writing to global variables, and setting return values. Next a list of reaching definitions must be created for each basic block in the function. The reaching definitions is created by finding, for each register used in each block, the block and instruction where that register was assigned the value that it contains at the current block in the code. After all the reaching definitions are computed, we walk through each critical instruction that was marked, and trace any source registers back to their definitions by either using the reaching definitions of each block, or walking backward in the block to the correct assignment. After the source instructions are found, these instructions are “marked” and added to a list of instructions to trace sources of. After this list is exhausted, we have a function with “marked” and “unmarked” instructions. The final step in Useless Code Removal is to filter out all the unmarked instructions, leaving only necessary instructions in the code.

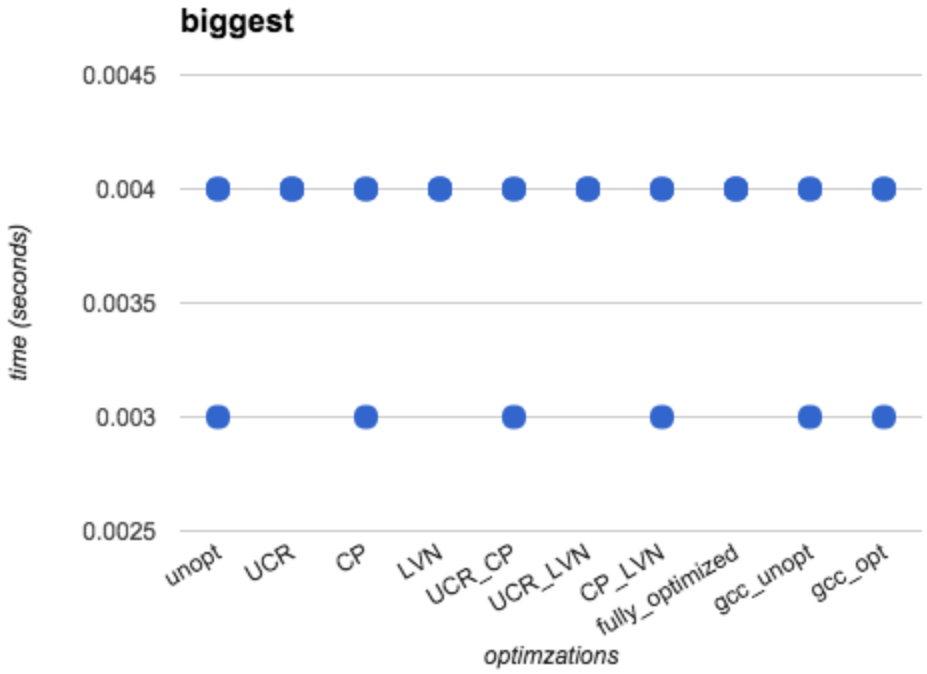
We ran into some annoying issues with this optimization because we did not have references in our language. As a result, our instructions in the reaching definitions had to be represented by a series of indices into basic blocks to locate the correct instruction.

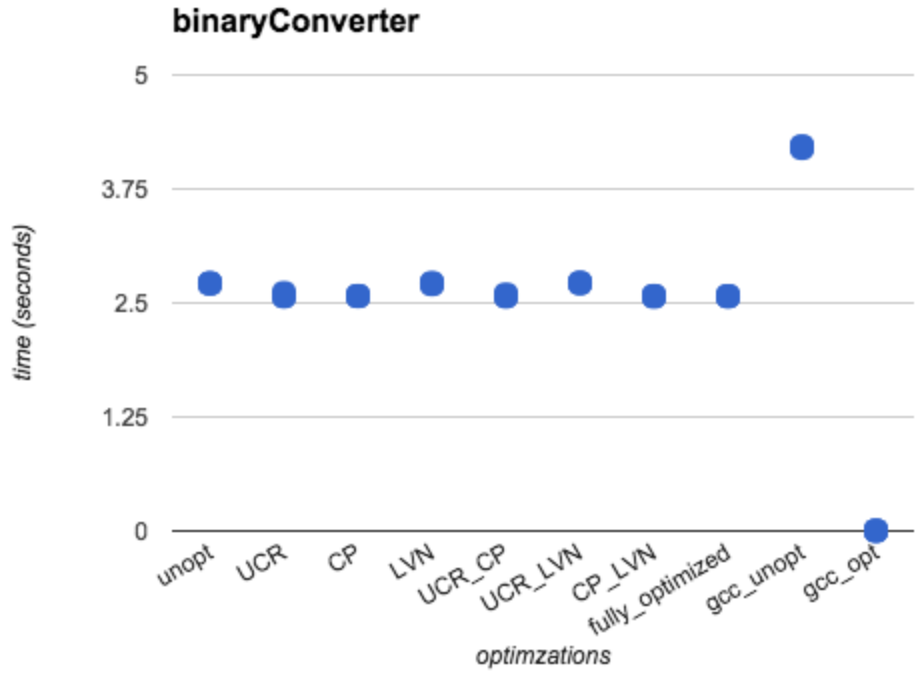
Results

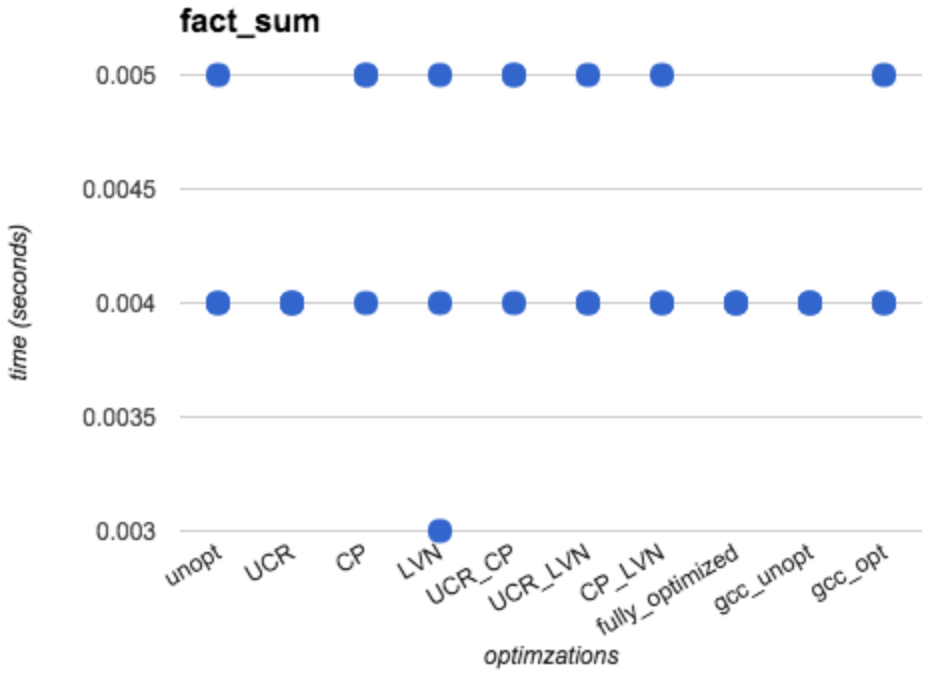
Below are the results of running our compiler with no optimizations, some optimizations and all of the optimizations. The C equivalent code compiled with gcc with and without optimization is also provided for comparison. We ran each test three times on the shared, school server unix13. Many of these benchmarks did not provide significant results because they finish very quickly, and the time command only takes to a hundredth of a second

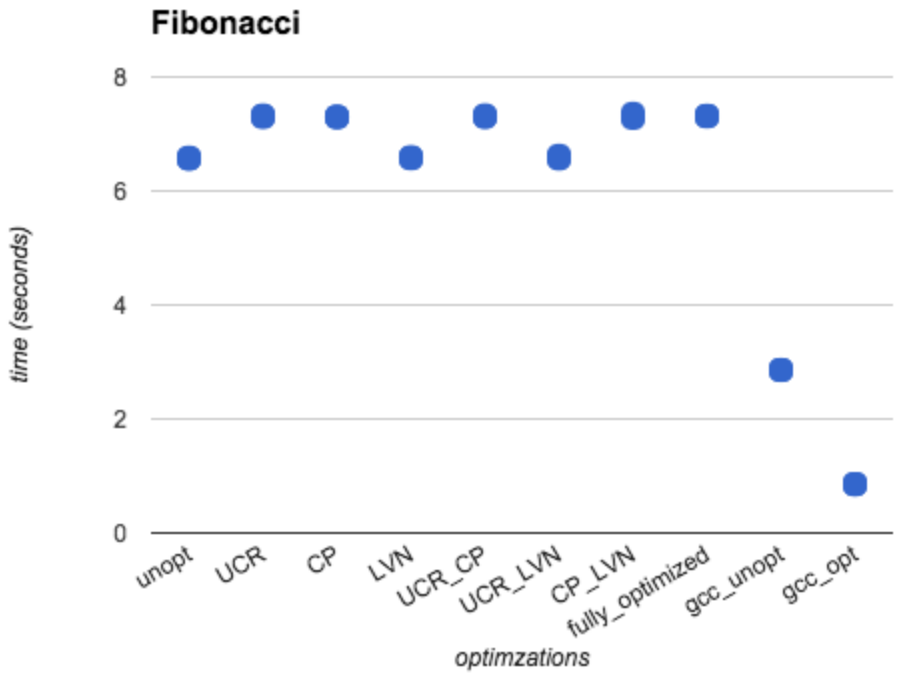

BenchMarkishTopics
0.0035
0.003
0.0025
0.002
0.0015
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

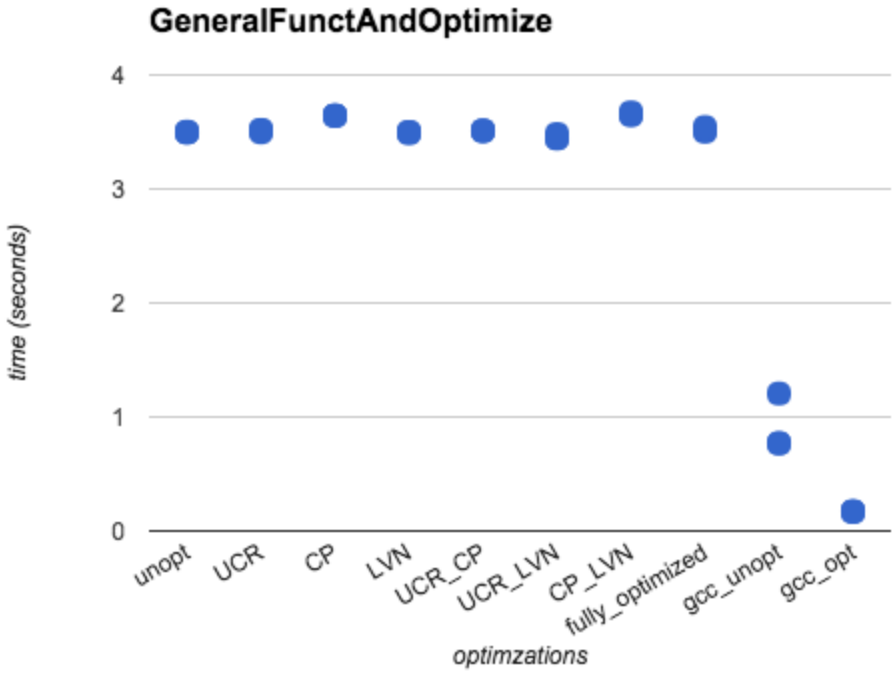

bert
0.0045
0.004
0.0035
0.003
0.0025
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

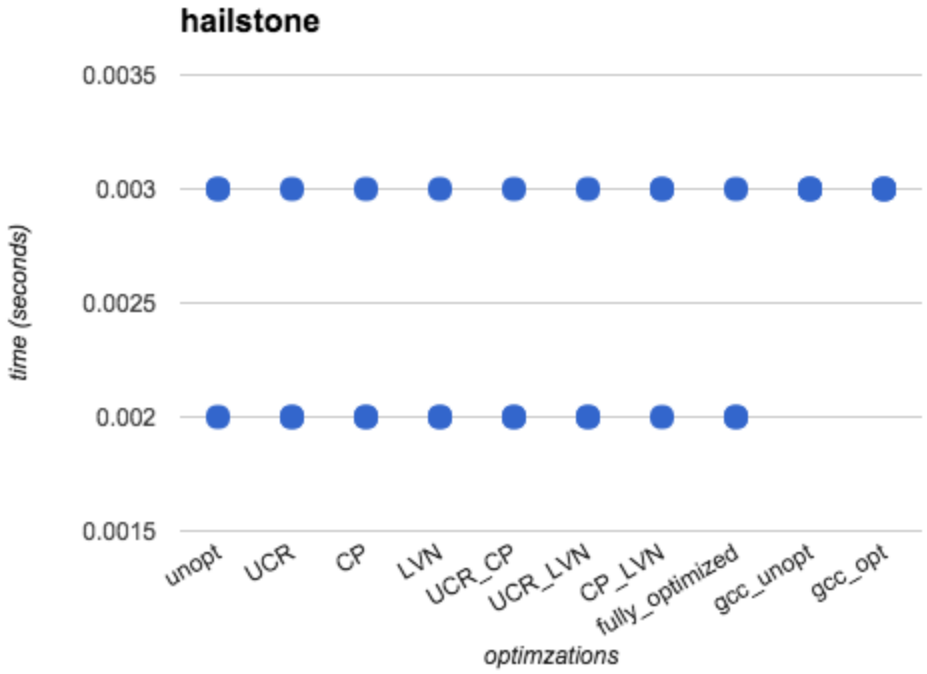

biggest
0.0045
0.004
0.0035
0.003
0.0025
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

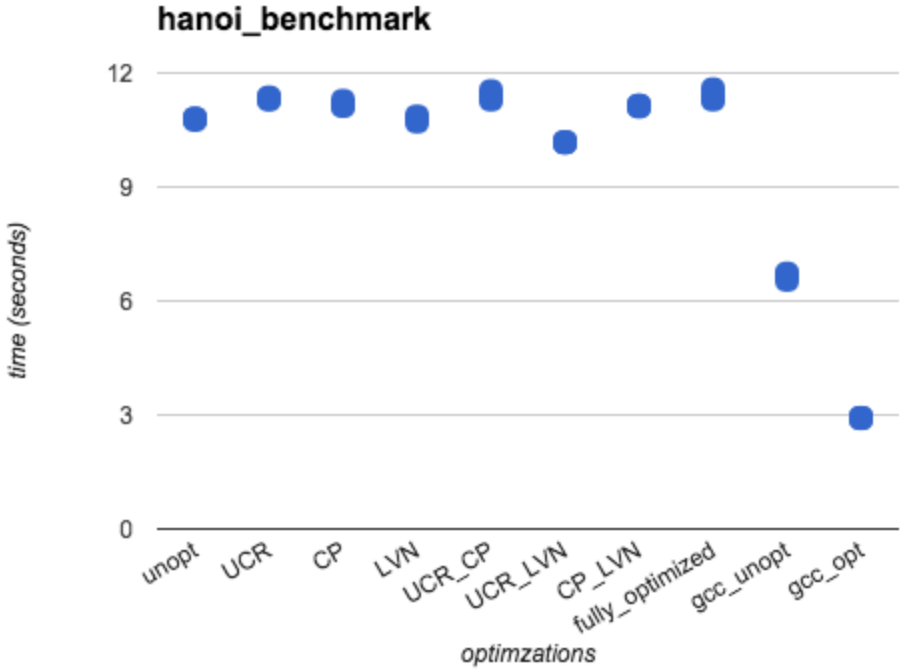

binaryConverter
5
3.75
2.5
1.25
0
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

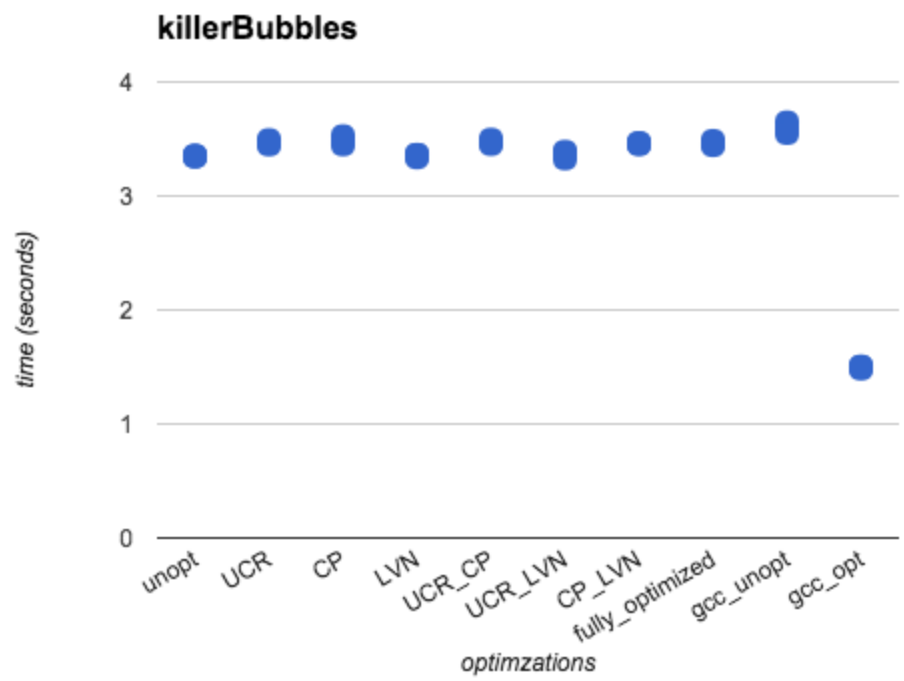

fact_sum
0.005
0.0045
0.004
0.0035
0.003
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

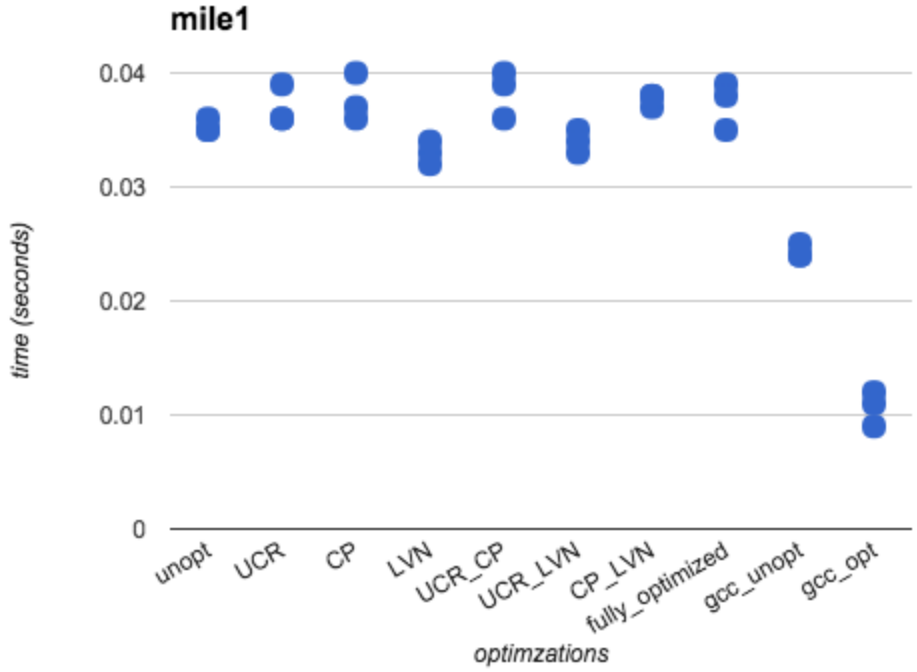

Fibonacci
8
6
4
2
0
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

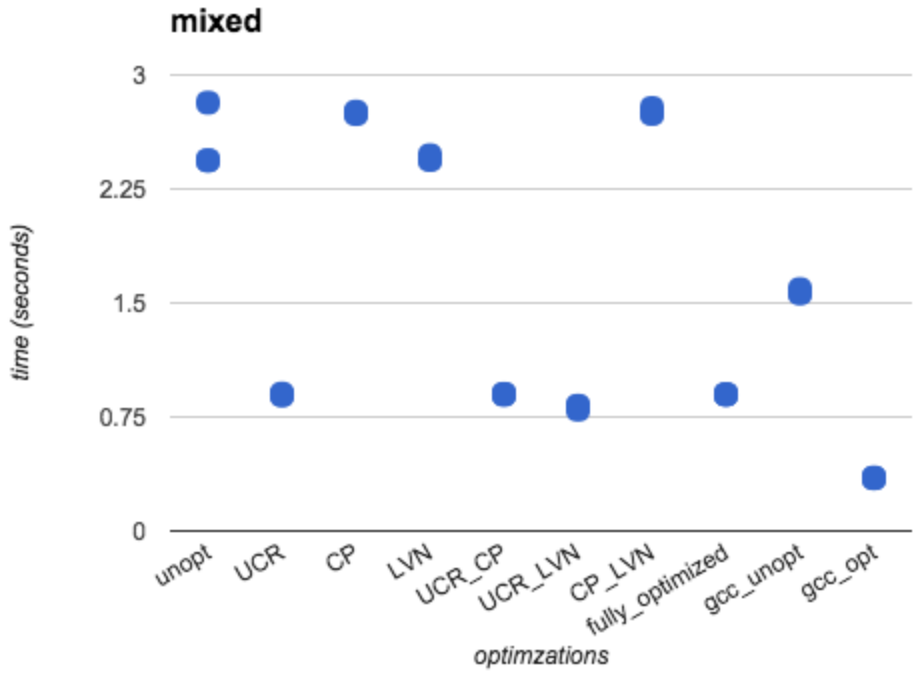

GeneralFunctAndOptimize
4
3
2
1
0
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

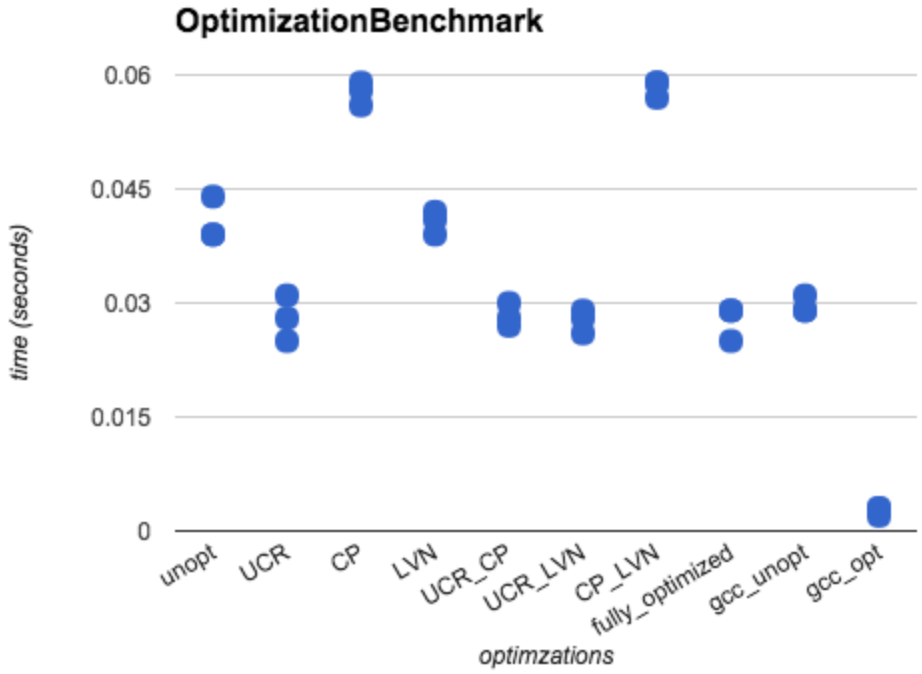

hailstone
0.0035
0.003
0.0025
0.002
0.0015
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

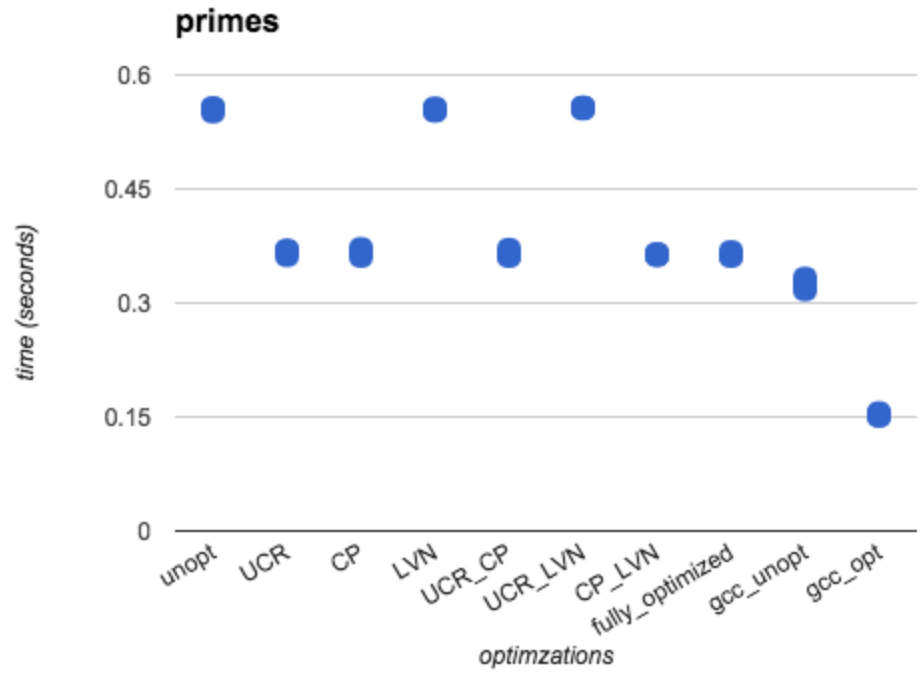

hanoi_benchmark
12
9
6
3
0
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

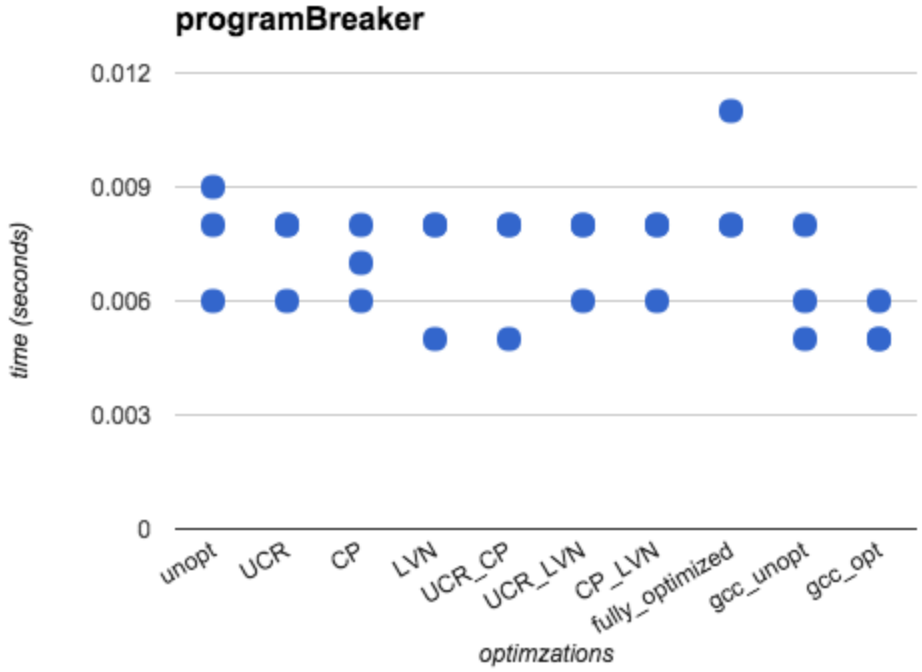

killerBubbles
4
3
2
1
0
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

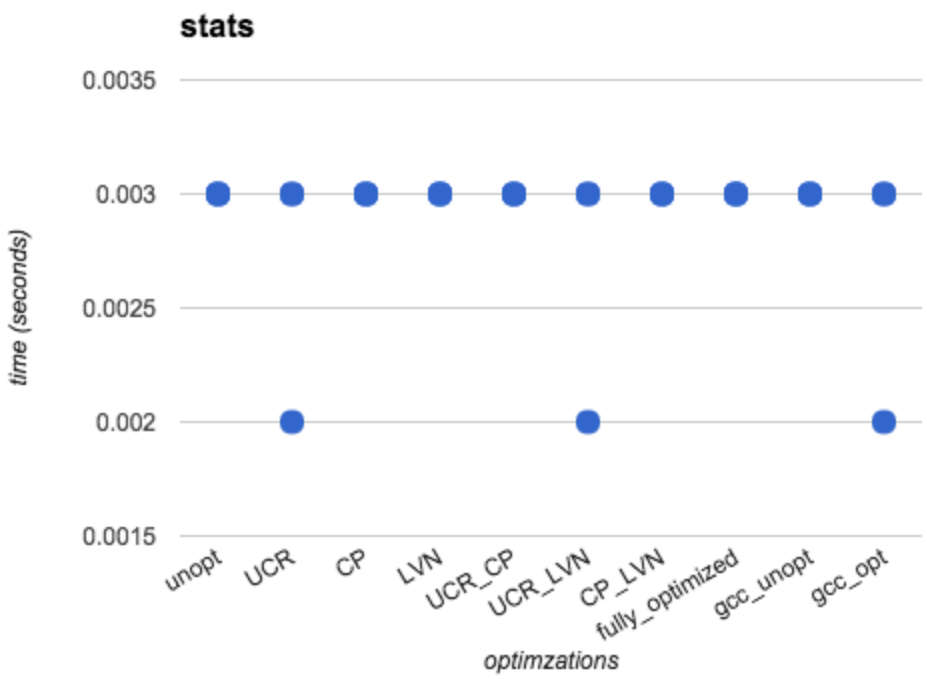

mile1
0.04
0.03
0.02
0.01
0
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

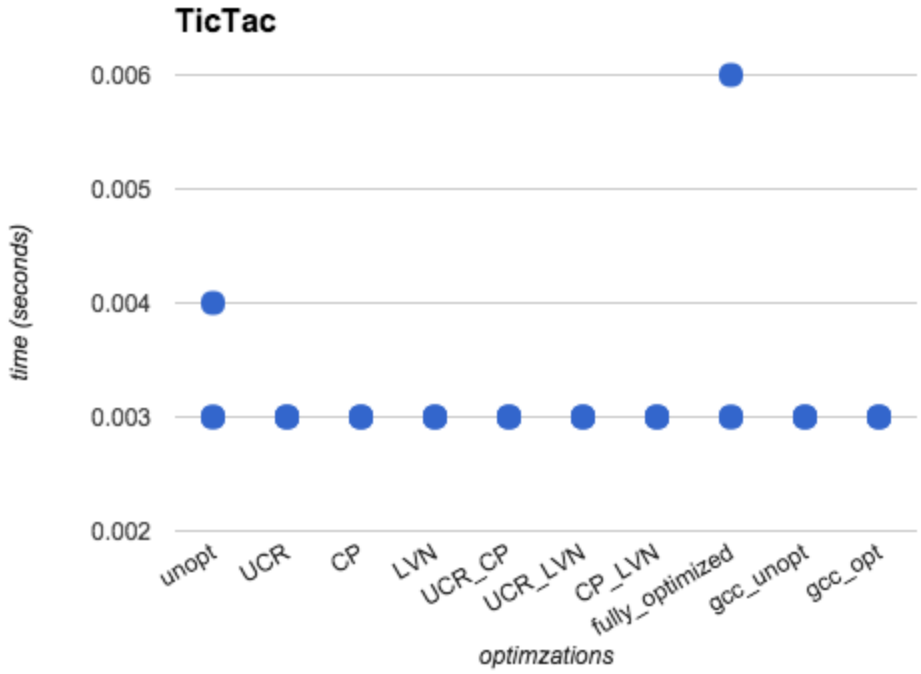

mixed
3
2.25
1.5
0.75
0
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

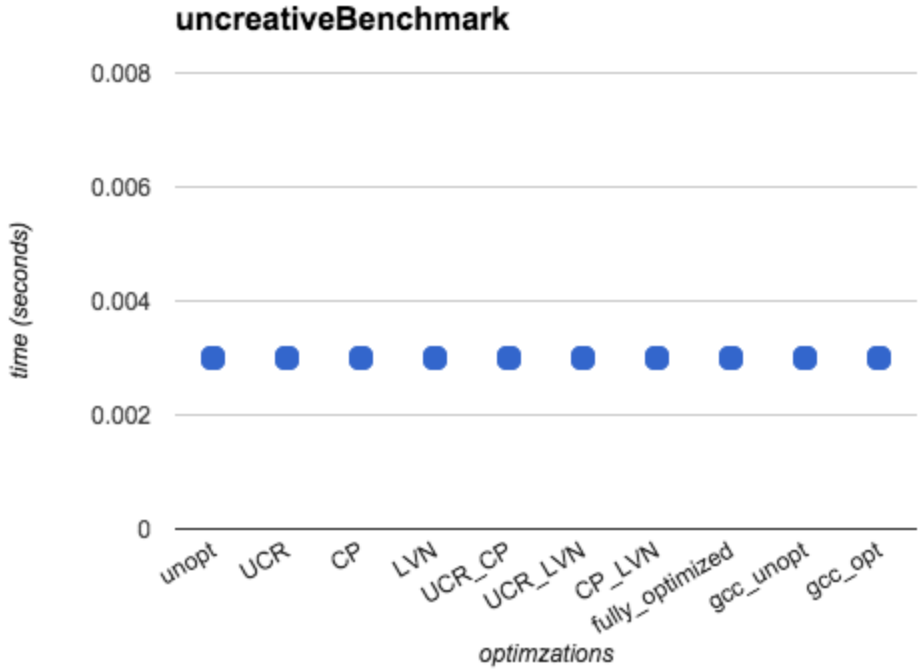

OptimizationBenchmark
0.06
0.045
0.03
0.015
0
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations

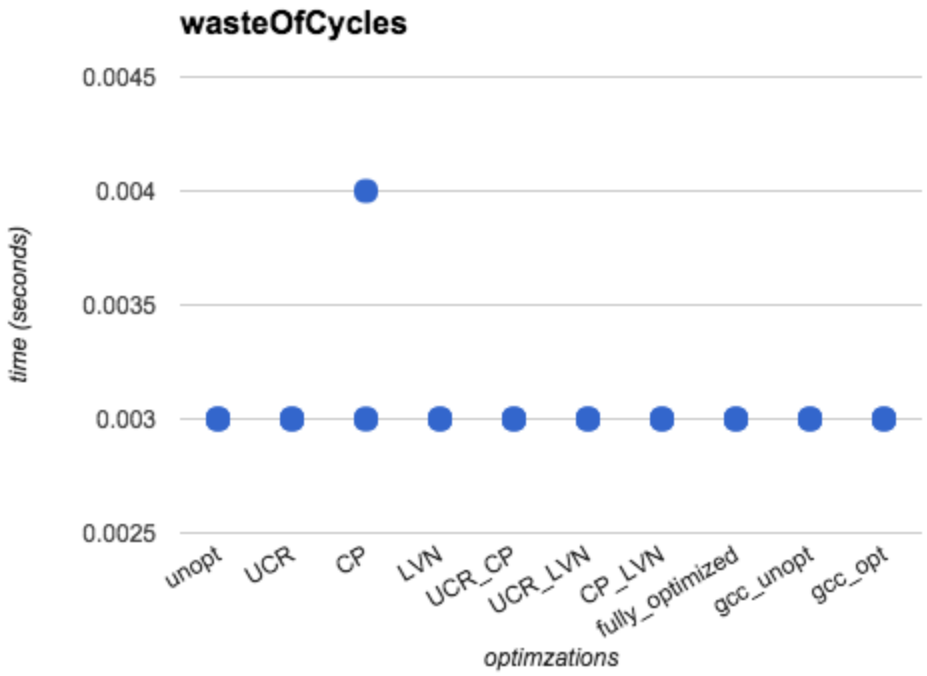

primes
0.6
0.45
0.3
0.15
0
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations


programBreaker
0.012
0.009
0.006
0.003
0
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations


stats
0.0035
0.003
0.0025
0.002
0.0015
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations


TicTac
0.006
0.005
0.004
0.003
0.002
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations


uncreativeBenchmark
0.008
0.006
0.004
0.002
0
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations


wasteOfCycles
0.0045
0.004
0.0035
0.003
0.0025
time (seconds)
unopt
UCR
CP
LVN
UCR_CP
UCR_LVN
CP_LVN
fully_optimized
gcc_unopt
gcc_opt
optimzations