Automatic Generation of Test and Benchmark Workloads

(Making programs that make programs)

Jozo J. Dujmović
Department of Computer Science
San Francisco State University

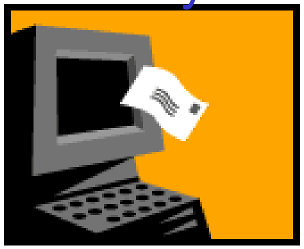
A New Approach to Benchmarking

- BenchMaker – a web oriented tool for generation of benchmark programs
- Benchmark generation procedure:
  - User visits a BenchMaker web site and specifies desired benchmark(s) properties
  - BenchMaker generates specified benchmarks and delivers them to the user by e-mail
- User compiles and executes benchmarks
- Open source

1. Specify benchmarks

2. Send specs to BenchMaker

3. Get benchmarks by e-mail



Contents

1. Classification of benchmarks
2. Industrial benchmarks
3. Benchmark scalability
4. BenchMaker 1 (BM1): Program generator based on the recursive expansion (REX) method
5. BenchMaker 2 (BM2): Program generator based on the kernel insertion (KIN) method
6. Applications of benchmark program generators
7. Work in progress:
    - (a) Towards open source benchmark manufacturing
    - (b) Benchmarking multicore and hyperthreaded systems

Classification of Benchmarks

Basic types of computer workloads

- **Natural** (written by programmers using selected programming languages; they have “semantic identity”, i.e. they are solutions of selected real problems)
- **Synthetic** (generated by code generators using correct language constructs combined according to desired distribution, but without semantic identity)
- **Hybrid** (segments of natural code combined by a code generator in order to create aggregated workloads that have desired size, resource consumption, and semantic identity)

Benchmarks

- Benchmark is any workload that is executed not to get its results, but to measure the speed of execution and the consumption of computer resources
- Benchmark workload must be a semantically correct sequence of service requests
- Goals of benchmarking:
  - Performance measurement of hardware units
  - Performance measurement of software units

Real Workload vs. Benchmark Workload

- **Real workload:** a workload that is the predominant computing activity of an analyzed computer system.
- **Benchmark workload:** a workload that is acceptable as a good representative of a real workload
- **Proof of similarity:** a quantitative proof that a selected benchmark workload is sufficiently similar to the real workload; this proof is a formal prerequisite for benchmarking

Theoretical background for benchmarking (1)

- Status: Benchmarking is usually considered and empirical art, and not an engineering activity based on strict theoretical background
- Consequences: controversial area that is heavily influenced by perception of analysts and by corporate interests:
  - The problem of standards and “standards”
  - SPEC and other industry consortia
  - The role of Internet in distributing incomplete and temporary results
- Ludwig Boltzmann: “There is nothing more practical than a good theory”

Theoretical background for benchmarking (2)

- Program space: Theoretical foundations of space where each point is a program (or another more complex computer workload)
- Program difference metrics: theoretical models of difference/distance between individual computer workloads:
  - White box approach
  - Black box approach
- Cluster analysis: Techniques for grouping similar workloads and replacing groups by one or more best representatives

Six basic types of benchmarks

1. Real workloads used as benchmarks
2. Standard benchmarks
3. Kernels
4. Microbenchmarks
5. Synthetic benchmarks
6. Hybrid benchmarks

1. Real workloads (used as benchmarks)

- **Characteristics:** a selected class of applications in a selected programming environment (100% natural workloads)
- **Advantages:**
  - Represent themselves - used to eliminate or reduce the standard criticism related to differences between the real and benchmark workloads
- **Disadvantages:**
  - Usually too complex and too diversified
  - The problem of the best representative among different programs in real workloads is the same as for any other benchmark
  - The problem of the best representative of input data (e.g. gcc xx; xx=?)
  - Restricted to specific HW/SW environment
  - Regularly modified after the change of HW/SW environment (reducing or eliminating the fundamental advantage of this approach)
  - Low portability of programs (regular use of all HW/SW-specific features)
  - Low portability of data
  - Low scalability
  - Use of proprietary data (data protection problems)
  - Problems related to input from users (interactive workloads, transact. proc.)
  - Low reusability (regularly unique, nonstandard, and non reusable SW)
  - Bottom line: High cost of benchmarking and questionable benefits

2. Standard benchmarks (e.g. SPEC)

- **Characteristics:** selected natural workloads modified to have fixed input, selected resource consumption, and serve as benchmarks
- **Advantages:**
  - Have semantic identity (problems from physics, chemistry, math, etc.)
  - Adjusted to provide high portability
  - Standardization (strict control of workload, conditions of execution and measurement method to secure reproducibility of results and comparison across various HW/SW platforms)
  - Public availability of a database of measurements for the majority of commercially available computers
- **Disadvantages:**
  - The quality of representation problem (representativeness of real workload)
  - Not scalable
  - Need permanent upgrading (short life span)
  - Fixed functionality (limited characterization of natural workloads)
  - No adjustable parameters (fixed resource consumption)
  - Affected by political processes inside consortia (approved by voting)
  - Expensive (high cost of standardization, measurement and renewal)

3. Kernels

- **Characteristics:** Important and frequently used components of natural workloads with easily recognizable semantic identity (matrix operations, sort, search, data compression, etc.)
- **Advantages:**
  - Clearly defined semantic identity
  - High portability
  - Low cost
- **Disadvantages:**
  - The quality of representation problem (representativeness of real workload)
  - Narrow scope of resource utilization
  - Limited scalability
  - Fixed functionality (limited characterization of natural workloads)

4. Microbenchmarks

- **Characteristics:** small natural code segments designed to isolate a specific performance feature and provide reliable performance indicators that characterize the selected HW/SW feature (e.g. the efficiency of recursive calls, the efficiency of array processing, the efficiency of parameter passing, the efficiency of sequential/random disk accesses, etc.)
- **Advantages:**
  - Clearly defined functionality and scope
  - Focused insight into a specific performance feature
  - High portability
  - Low cost
- **Disadvantages:**
  - Very narrow scope
  - Absence of methodology for aggregating microbenchmark results

5. Synthetic benchmarks

- **Characteristics:** HLL programs automatically generated by benchmark generators according to user specification. No natural workloads included.
- **Advantages:**
  - Possibility to specify desired frequencies of available language constructs
  - Fast generation of any size of source code
  - Full portability
  - Suitable for benchmarking compilers
  - No cost
- **Disadvantages:**
  - Fully artificial code (low representativeness of real programs)
  - Limited (rather low) diversity of generated code

6. Hybrid benchmarks

- **Characteristics:** HLL programs automatically generated by benchmark generators as combinations of selected natural code segments according to user specification.
- **Advantages:**
  - Easy adjustment of desired semantic identity
  - Possibility to specify desired frequencies of available natural code segments, and select desired structure of benchmark program
  - Fast generation of any size of source code in variety of languages
  - High scalability
  - Practically unlimited spectrum of functionality
  - Full portability
  - Mostly natural with low synthetic overhead
  - Suitable for wide variety of benchmarking tasks
  - Negligible cost
- **Disadvantages:**
  - The quality of representation problem (representativeness of real workload is based on aggregated semantic identity)

Benchmark Workloads

- Individual benchmark programs
- Benchmark suites
- Benchmark series

Benchmark Suites

- A family of nonredundant benchmark programs having a variety workload characteristics (e.g. numeric [int and/or float] and nonnumeric/combinatorial problems)
- Typical benchmark suites are expected to include a necessary and sufficient variety of workload characteristics that represent a set of expected natural workloads (proof = ?)
- Typical usage: performance evaluation and comparison of competitive computer systems

Benchmark Series

- A sequence of benchmark programs having same workload characteristics but different (increasing) sizes
- Typical series include increasing number of lines of code (or increasing memory consumption)
- Typical usage: compiler performance measurement and analysis

Program Cloning – a Goal for the Future

- Define a set of measurable program parameters
- Extract program parameters from a running natural workload
- Pass the parameters to a program generator
- Specify additional scalability parameters (desired size and resource consumption)
- Generate synthetic workloads according to given specifications (and provide a measure of accuracy)

Industrial Benchmarks

(And Their Relation to Moore's Law)

MOORE'S LAW: Exponential growth of computer performance as a function of time

$$\boxed{q(t) = q_0 2^{t/T}}$$

$$q(0) = q_0$$

$$q(T) = 2q_0$$

$$q(2T) = 4q_0$$

$$q(nT) = 2^n q_0$$

t = time
q = performance (speed, mem., cost)
q_0 = initial performance at time t=0
T = performance doubling time
≅ 18 months for memory capacity
≅ 12 months for performance/price
New problem: Core # doubling time

MOORE'S LAW: current issues

- Limits of clock rate (< 5 GHz)
- Limits of processor power (< 100 W)
- Expansion in the area of parallelism (multiple processor cores, hyperthreading)
- Difficult software problems:
  - How to write/compile/optimize parallel programs?
  - SW developers are not ready to utilize the expected exponential growth of processor cores
- Core doubling time ≠ performance doubling time

Approach currently used by industry [1/2]

"Technology evolves at a breakneck pace. With this in mind, SPEC believes that computer benchmarks need to evolve as well. While the older benchmarks (SPEC CPU95) still provide a meaningful point of comparison, it is important to develop tests that can consider the changes in technology."

http://www.spec.org/osg/cpu2000/

Approach currently used by industry [2/2]

The SPEC CPU Benchmark Search Program

SPEC holds to the principle that better benchmarks can be developed from actual applications. With this in mind, SPEC is once again seeking to encourage those outside of SPEC to assist us in locating applications that could be used in the next CPU-intensive benchmark suite, currently planned to be SPEC CPU2004.

http://www.spec.org/osg/cpu2000/CPU2004/search_program.html

Back of the Envelope Feasibility Analysis

Main memory size = x GB

Lines of source code in 50 MB of memory = 1,000,000

Effort to write 1,000,000 LOC = 6873 person months [intermediate COCOMO]

Time to write 1,000,000 LOC = 55 months = 4.6 years

Number of software engineers = 125

Development cost = $xx Million

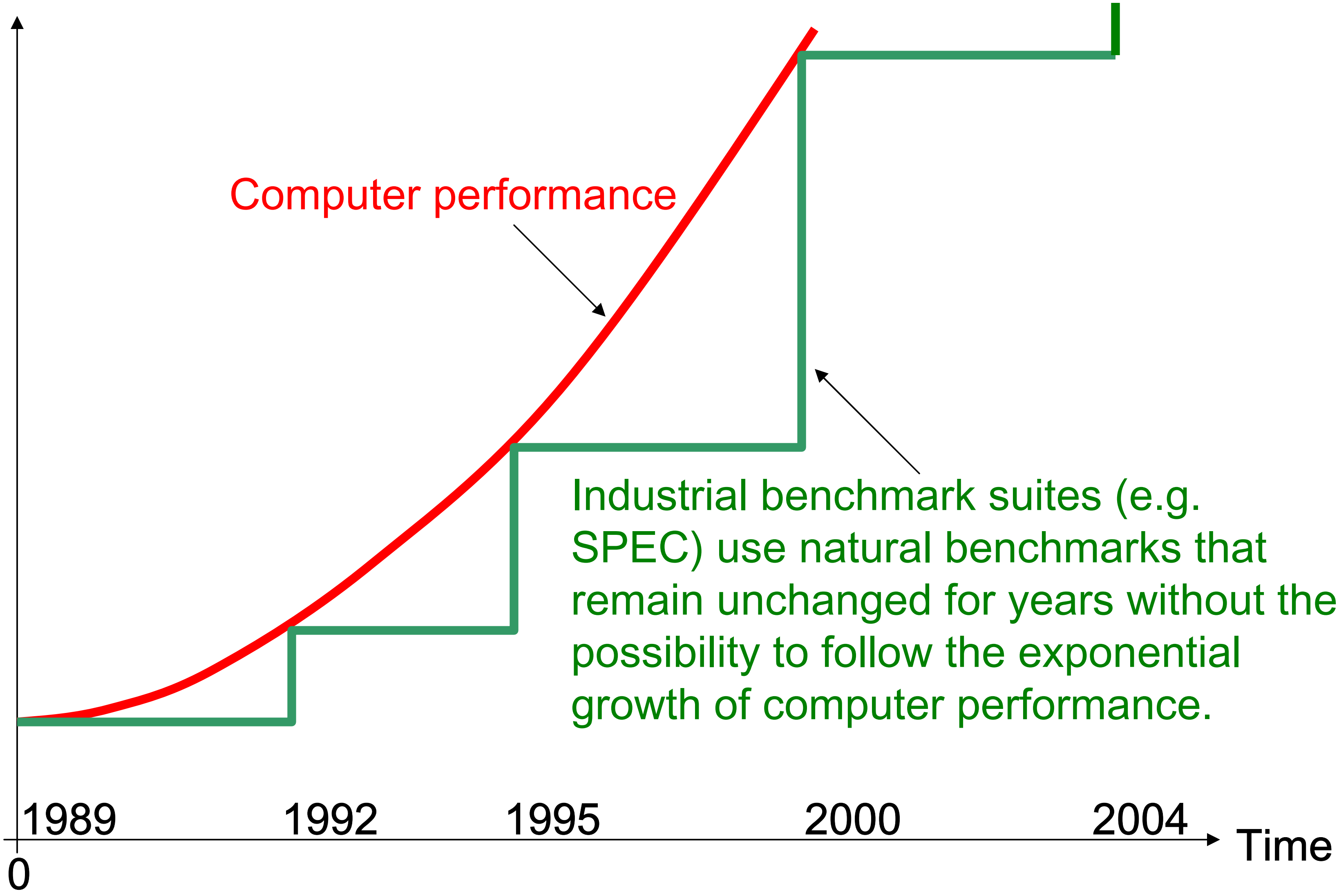
Reward offered by SPEC = $x Thousand

Discrepancy factor = 10000

Natural vs. Synthetic Programs

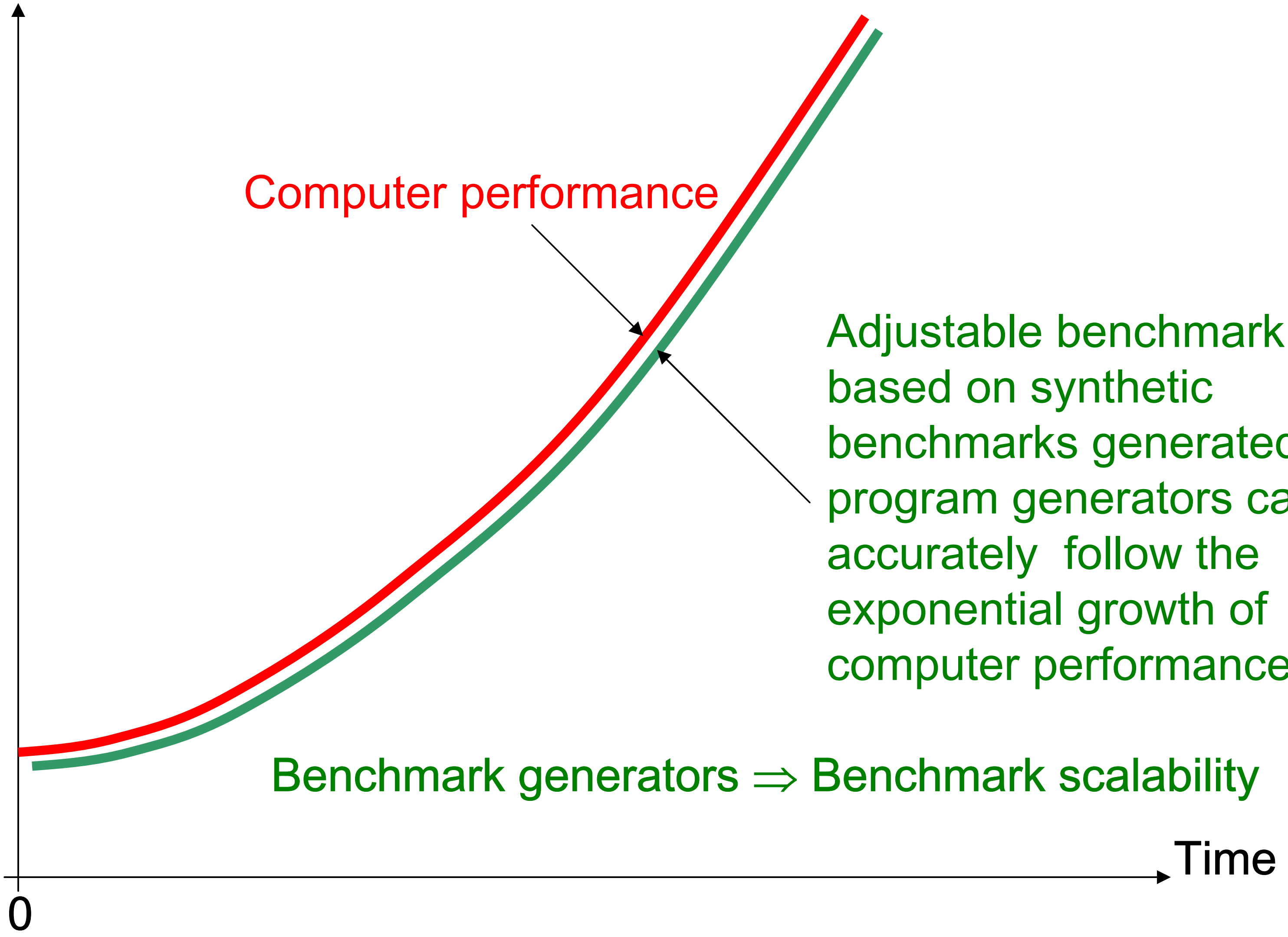
Q:	Is it possible to follow Moore’s law using natural (manually written) benchmark programs?
A:	No!
Q:	Why?
A:	Because the computer performance grows faster than our ability to provide natural, representative, reliable, and permanently increasing large programs.
Q:	How to quickly create benchmark programs having desired properties and desired size?
A:	The only way is to develop techniques and tools for automatic generation of benchmark programs.

Current Performance/Benchmark Relation



Industrial benchmark suites (e.g. SPEC) use natural benchmarks that remain unchanged for years without the possibility to follow the exponential growth of computer performance.

Desired Performance/Benchmark Relation



Computer performance

Adjustable benchmark suites based on synthetic benchmarks generated by program generators can accurately follow the exponential growth of computer performance.

Benchmark generators ⇒ Benchmark scalability

Current Industrial Benchmarks

- Not scalable
- Expensive
- Need permanent upgrading
- Fixed functionality (limited characterization of natural workloads)
- No adjustable parameters (fixed resource consumption)
- Affected by political processes inside consortia (approved by voting)

Desired Features of Industrial Benchmark Programs

Industrial benchmark suites should be able to strictly follow the exponential growth of computer performance and provide:

- ⇨ Adjustable program size
- ⇨ Adjustable memory consumption
- ⇨ Adjustable CPU power consumption
- ⇨ Adjustable functionality

Such Benchmarks must be:

- ⇨ Quickly generated (> 1MLOC/minute)
- ⇨ Able to easily adjust workload properties
- ⇨ Inexpensive and available on the Web

Suggested Approach to Industrial Benchmarks

- Based on generators of scalable synthetic (hybrid) benchmarks
- Adjustable functionality
- Adjustable resource consumption
- Web-oriented
- Produced by the user according to user’s specifications
- Open-source

Currently Available Generators of Benchmark Programs

- **BenchMaker 1** (BM1: generator of compilable programs primarily used for compiler performance measurement and analysis; limited control of executable properties)
- **BenchMaker 2** (BM2: generator of general purpose executable programs, used for computer performance measurements; good control of executable properties)

Benchmark Scalability

(Manufacturing Scalable
Benchmarks)

Benchmark Scalability (1/2)

- Benchmark properties that are relevant for the usability of benchmarks in system performance analysis include resource consumption (processor, memory, disk), functionality (type of processing), program structure, etc.
- Benchmarks are scalable if users can create benchmark workloads having independently adjustable all relevant properties.

Benchmark Scalability (2/2)

- Controlled increase of the consumption of computing resources (memory, processors, etc.) by adding more, or more specific, benchmark program modules
- Support for both upwards and downwards scalability
- Scalable benchmarks are manufactured according to user’s specifications.

Six types of benchmark scalability

1. **Time scalability** (user selects the benchmark run time)
2. **Space scalability** (user adjusts the benchmark size and its memory consumption)
3. **Parametric scalability** (adjustable for each benchmark)
4. **Structural scalability** (benchmarks have adjustable structure; generation of benchmark series and suites)
5. **Functional scalability** (semantic workload characterization: each user can select functions that are similar to an existing or expected user workload)
6. **Mixed software scalability** (user programs can be inserted as a part of benchmark workload)

1. Time Scalability

- Selection of benchmark program run time according to user’s needs
- Implementation:
  - Benchmark program consists of independent program modules (e.g. kernels)
  - By adjusting loop parameters each kernel is calibrated to have a specified run time on a given machine
  - Benchmark run time is adjusted by selecting the number of kernels to be executed

2. Space Scalability

- Selection of benchmark program size (both LOC and MB) according to user’s needs (e.g. from 50 LOC to 5 MLOC; LOC $\in$ {PLOC, LLOC})
- Implementation:
  - Benchmark program consists of independent program modules (typically kernels)
  - By adjusting array parameters each kernel is calibrated to use a desired memory space
  - Benchmark size is adjusted by selecting the number of kernels to be executed

3. Parametric scalability

- Scalability based on adjusting various benchmark program parameters.
- Typical parameters:
  - The number of users (threads)
  - The number of network nodes
  - The size of arrays
  - The run time
  - The number of disk accesses

4. Structural Scalability

- Adjusting of the structure of workload
- Typical components:
  - Selecting the structure of kernel invocations in a benchmark program
  - Selecting network topology for network benchmarks (e.g. ring, star, grid, etc.)

5. Functional Scalability

- Scalability based on semantic characterization of workload
- Selection of kernels that belong to a desired application area. E.g.:
  - Numerical procedural problems
  - Nonnumerical procedural problems
  - Object oriented problems
  - Memory and/or disk access
  - System applications
  - Etc.

6. Mixed software scalability

- In addition to kernels, synthetic benchmark programs can also include selected user programs
- Mixed software scalability refers to the capability to select a desired fraction of benchmark that is based on user’s programs (combining user functions and kernel library functions)

Space scalability details

- The size of program – a fundamental parameter of all benchmark programs
- Program size affects the program development time, production cost, memory consumption, and the run time
- Program size must be precisely defined and there are several different definitions

Program size metrics

- There are various metrics for measuring program size:
  - Only executable lines
  - Executable lines and data definitions
  - Executable lines, data definitions and comment lines
  - Physical lines of code (newlines)
  - Logical lines of code (complete statements)

Benchmark Size Metric for C++

- LLOC = Logical Lines Of Code
- PLOC = Physical Lines of Code

- BM1 creates logical lines of code and the size of programs is specified in desired LLOC
- Approximately: PLOC ≈ 1.6*LLOC

Definition of LLOC for C++

For C++ programs we use the following:

LLOC = # of programming units (functions + main)

+ # of “;” (whole program except comments)

+ # of “=“ (constructor-initializer statements only)

+ # of “if” statements

+ # of “switch” statements

+ # of “while” statements

+ # of “for” statements

Arithmetic

```
int a;            //  Constructor
a = 123;          //  Assignment
                  //  LLOC = 2
```

```
int a = 123;      //  Constructor + assignment
                  //  LLOC = 2
```

```
a = 123;          //  LLOC = 1
```

If

```
if(condition)
  a = 1;                 // LLOC = 2

if(condition)
  a = 1;
else
  b = 2;                 // LLOC = 3
```

Concept = Frame + inserted statements

LLOC += Keyword (if) + # of “ ; “

switch

```
switch (selector)
case 1: a = 1; break;
case 2: b = 2; break;
case 3: c = 3; break;
default: d = 0;                    // LLOC = 8
```

LLOC += Keyword (switch) + # of “ ; “

while

```
while (condition)
{
        a[n] = n;
        b[n] = n++;
}                       // LLOC = 3
```

LLOC += Keyword (while) + # of “ ; “

do

```
do
{
        a[n] = n ;
        b[n] = n++ ;
} while (condition) ;          // LLOC = 3 (not 4)
```

LLOC counter is incremented on “;” but not on keyword “do”

LLOC += # of “ ; “

for

Original for loop:

```
for(j=0 ; j<n ; j++)
{
   a[ j ] = 0;
   b[ j ] = j;
}      // LLOC = 5
```

(# of “;” + 1 (keyword))

For loop transformed to while:

```
j=0;
while (j < n)
{
      a[ j ] = 0;
      b[ j ] = j;
      j++ ;
}            // LLOC = 5
```

Benchmark Generators

(Manufacturing Scalable Benchmarks)

Benchmark Manufacturing

- Production of benchmarks by the user, according to user's specification
- Features: scalability, speed, and low cost
- Production based on a benchmark program generator tool
- Type of benchmark products:
  - Individual benchmarks
  - Benchmark series
  - Benchmark suites

Application Areas and Goals

- Design of industrial benchmark suites
- Reducing the cost of benchmarking
- Increasing the credibility of benchmarking
- Evaluation and comparison of language processors (compilers, VMs, interpreters)
- Computer evaluation and comparison
- Test program generation
- Study of workload properties
- Software metrics and experimentation

Benchmark Generators Design Concepts

BenchMaker1: Based on Recursive Expansion (**REX**) concept of benchmark program development. Program is generated by systematic insertion of blocks into control statements, and statements into blocks.

BenchMaker2: Based on Kernel Insertion (**KIN**) concept. Program is generated by systematic insertion of independent code segments (kernels) from a library.

BenchMaker 1 and the Recursive Expansion Program Generation Method

The concept of BM1

- Sequences, and all control structures have the form of frames where programmers can insert contents
- Synthetic programs can be created in the same way

Block Containing Statements

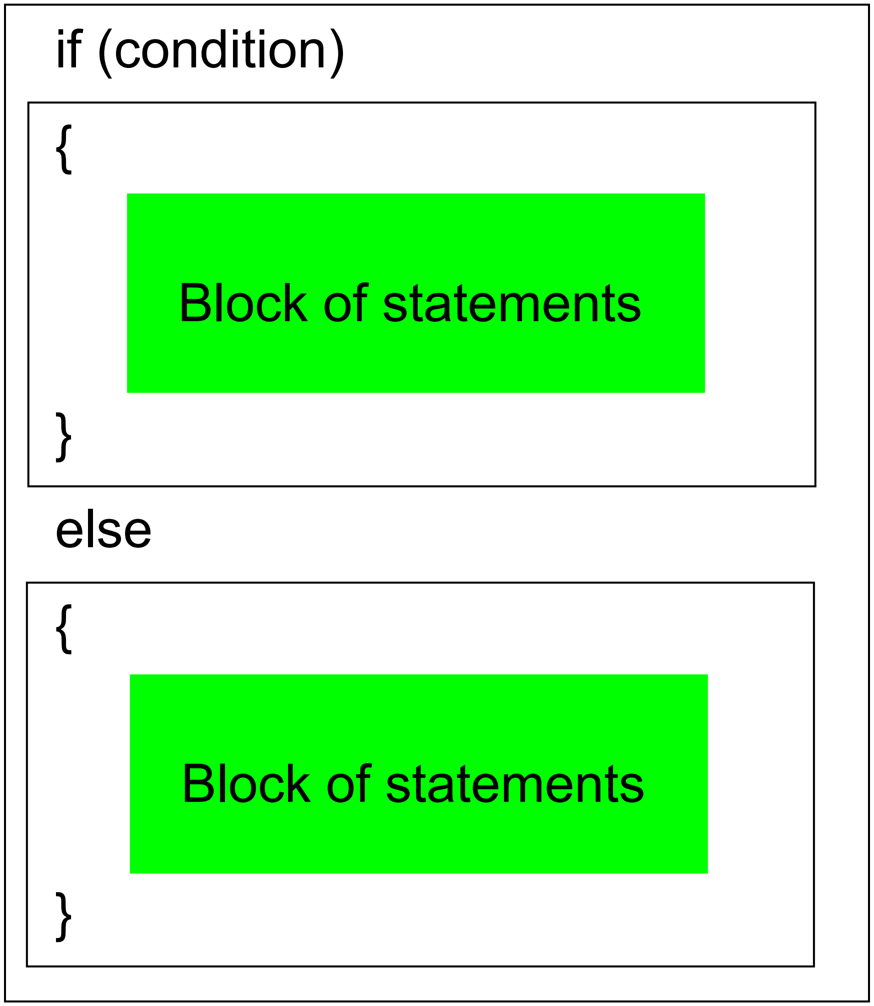
```
int main(arguments)
{   // block
    Statement
    Statement
    Statement
    Statement
}
```

```
int func(arguments)
{   // block
    Statement
    Statement
    Statement
    Statement
}
```

Classification of Statements

- **Expandable statements**: contain frames (blocks) and can be expanded by inserting statements into frames
- **Terminal statements**: fixed contents that cannot be expanded
  - Simple (arithmetic)
  - Compound (fixed blocks, e.g. kernels)

Expandable Statement



if (condition)

{

Block of statements

}

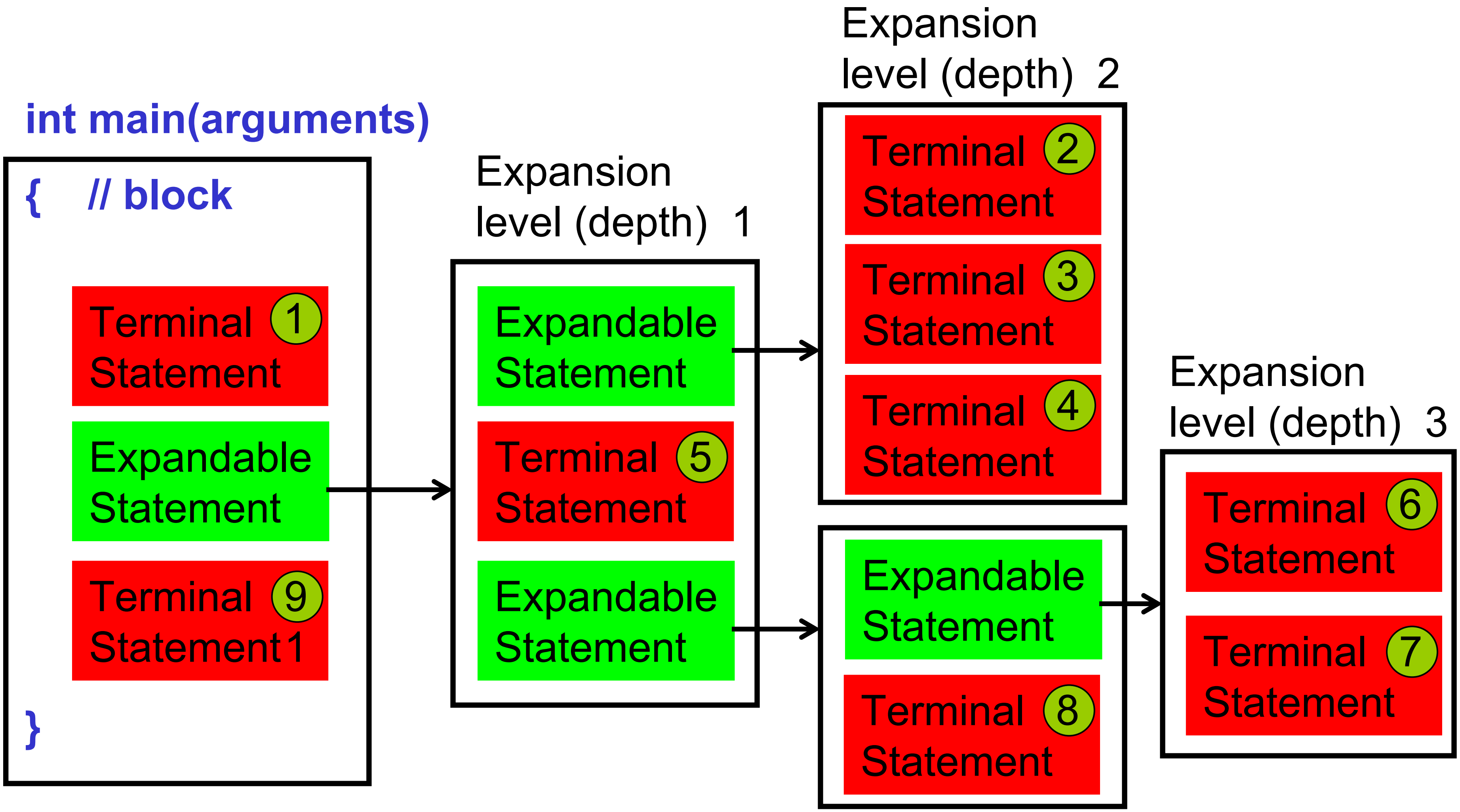
else

{

Block of statements

}

Expansion of Statements



int main(arguments)

{ // block

Terminal ① Statement

Expandable Statement

Terminal ⑨ Statement 1

}

Expansion level (depth) 1

Expandable Statement

Terminal ⑤ Statement

Expandable Statement

Expansion level (depth) 2

Terminal ② Statement

Terminal ③ Statement

Terminal ④ Statement

Expandable Statement

Terminal ⑧ Statement

Expansion level (depth) 3

Terminal ⑥ Statement

Terminal ⑦ Statement

The Concept of Breadth

```
{
      statement;
      statement;
      statement;          //      B = 5
      statement;
      statement;
}
```

The Concept of Depth

```
{  // 0
        {  // 1
                {  // 2
                        statement;          //        D = 2
                }
        }
}
```

REX Program Model

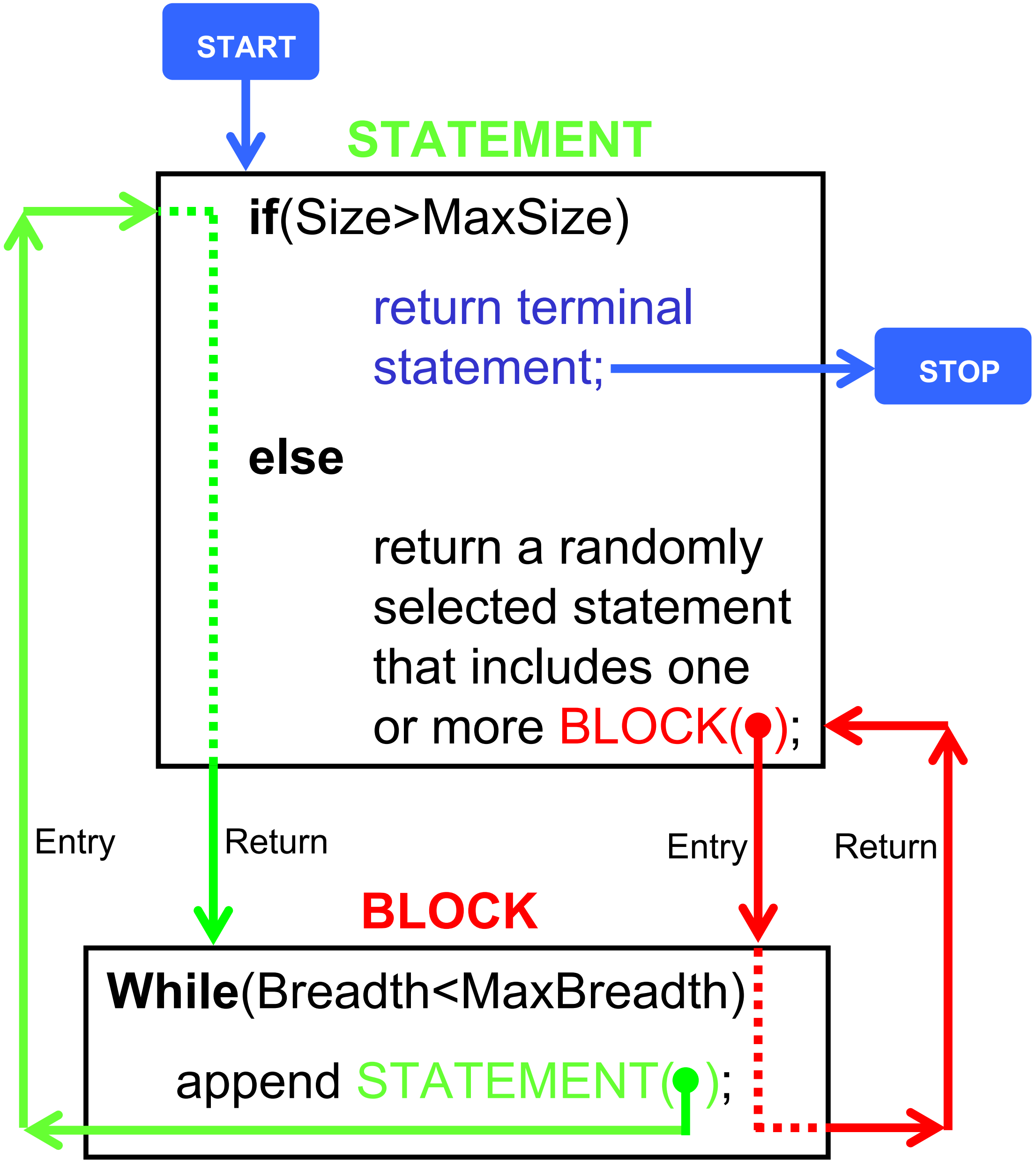
- Each block contains one or more statements.
- Each control statement contains one or more blocks. An example of two blocks:

  if(condition) {block} else {block}

- Create programs by systematically inserting blocks into statements and statements into blocks (stepwise refinement).
- When the generated program attains a desired size, insert a "terminal block" (either an arithmetic statement or an executable kernel).

REX Model Recursion

```
string STATEMENT(…)
{    ……………
     BLOCK(…);
}

string BLOCK(…)
{   ……………………
    STATEMENT(…);
}
```



START

STATEMENT

if(Size>MaxSize)

return terminal statement;

STOP

else

return a randomly selected statement that includes one or more BLOCK(●);

Entry

Return

Entry

Return

BLOCK

While(Breadth<MaxBreadth)

append STATEMENT(●);

A toy REX generator [1/3]

```
string STATEMENT(int D, int B, int selector)  // D = depth, B = breadth
{
  if (++D > maxDepth) selector = 0;       // End of recursive expansion
  switch (selector)
  {
    case 0: return assignment( ) + "\n";  // Assignment terminator
    case 1: return "if" + condition( ) + "\n" + BLOCK(D, B)+ "\n";
    case 2: return "if" + condition( ) + "\n" + BLOCK(D, B) + "\n" +
                    indent(D) + "else\n"  + BLOCK(D, B)+ "\n";
    case 3: return "while" + condition( ) + "\n" + BLOCK(D, B)+ "\n";
    case 4: return "do\n" + BLOCK(D, B) + " while" + condition( )+";\n";
  }
}
```

A toy REX generator [2/3]

```
string BLOCK(int D, int B)     // D = depth, B = breadth
{
  string block = indent(D) + "{\n" ;
  for(int i=0; i<B; i++)
    block += indent(D+1) +
             STATEMENT(D, 1+rand()%maxBreadth, rand()%5);
  return block + indent(D) + "}";
}
```

A toy REX generator [3/3]

```
void main( void )
{
    fstream file;
    srand(time(NULL));  // randomize
    cout << "\n\nToy program generator\n\n"
         << "Maximum Breadth = "; cin  >> maxBreadth;
    cout << "Maximum Depth   = "; cin  >> maxDepth;
    file.open("demo.cc", ios::out);
    file << "void main(void)\n{\n" +
             indent(1) + "int " + init(nvars, ",") + ";\n" +
             indent(1) + init(nvars, "=") + "=1;\n" +
             indent(1) + STATEMENT(0, maxBreadth, 1+rand()%4) + "}\n";
    cout << "demo.cc completed.\n";
}
```

A Sample Program

```
#include<iostream.h>
void main(void)
{
   int I,a,b,c,d,e,f,g,h,i,j,k,l,m,n;
   a=b=c=d=e=f=g=h=i=j=k=l=m=n=1;
   long S=0, G[20000]; for(I=0; I<20000; I++) G[I]=0;
   while(++G[2]%3)  //  1,2,0,1,2,0,…
   {
      if(++G[0]%2)  //  1,0,1,0,1,…
      {
         i = k-a-k*b+f+e+d-d-m*m+h+g-f;
         l = m+d-n-m+n*i+n;
      }
      else
      {
         e = h*f-g-l*f+a+a*m;
         h = a-h*h-l+k*k-l*d+e-l*m;
      }
      while(++G[1]%3) // 1,2,0,1,2,0,…
      {
         b = d-m-j+m-j+k-b+a+e-g-i+f*g;
         j = k*f*m*b*h-d+l+b;
      }
   }
   for(I=0; I<3; S+=G[I], I++)
      cout << G[I] << ((I+1)%10 ? ' ':'\n');
   cout << "\nNumber of control statements = 3";
   cout << "\nExecuted control statements  = " << S << '\n';
}
```

```
$ g++ demo.cc
$ ./a
2 6 3
Number of control statements = 3
Executed control statements  = 11
```

Experiments With Compilable Benchmark Programs [1/2]

```
$ time ./tg


Toy program generator

Maximum Breadth = 7
Maximum Depth   = 7
Loop Repetition = 7
demo.cc completed.

real    0m7.492s
user    0m3.327s
sys     0m0.046s

$ wc -l demo.cc
 100755 demo.cc
```

```
$ time g++ demo.cc

real    13m16.637s
user    7m6.169s
sys     0m10.341s

$ ls -l demo.cc a.exe
2673681 Oct  9 11:00 a.exe
3570094 Oct  9 10:43 demo.cc
```

Density = 26.5 Bytes / PLOC

≈ 70 Bytes / LLOC

Experiments With Compilable Benchmark Programs [2/2]

```
$ time ./tg


Toy program generator

Maximum Breadth = 7
Maximum Depth   = 7
Loop Repetition = 10
demo.cc completed.

real    0m4.907s
user    0m2.936s
sys     0m0.108s

$ wc -l demo.cc
  89675 demo.cc
```

```
$ time g++ demo.cc

real    10m55.547s
user    6m42.356s
sys     0m8.419s

$ ls -l demo.cc a.exe
2586641 Oct  9 12:02 a.exe
3193103 Oct  9 11:49 demo.cc

Time ./a
- - - - - - - - - - - - - - - - - -
Number of control statements = 11603
Executed control statements  = 973081553

real    1m1.831s
user    0m59.686s
sys     0m0.077s
```

Density = 28.8 Bytes / PLOC

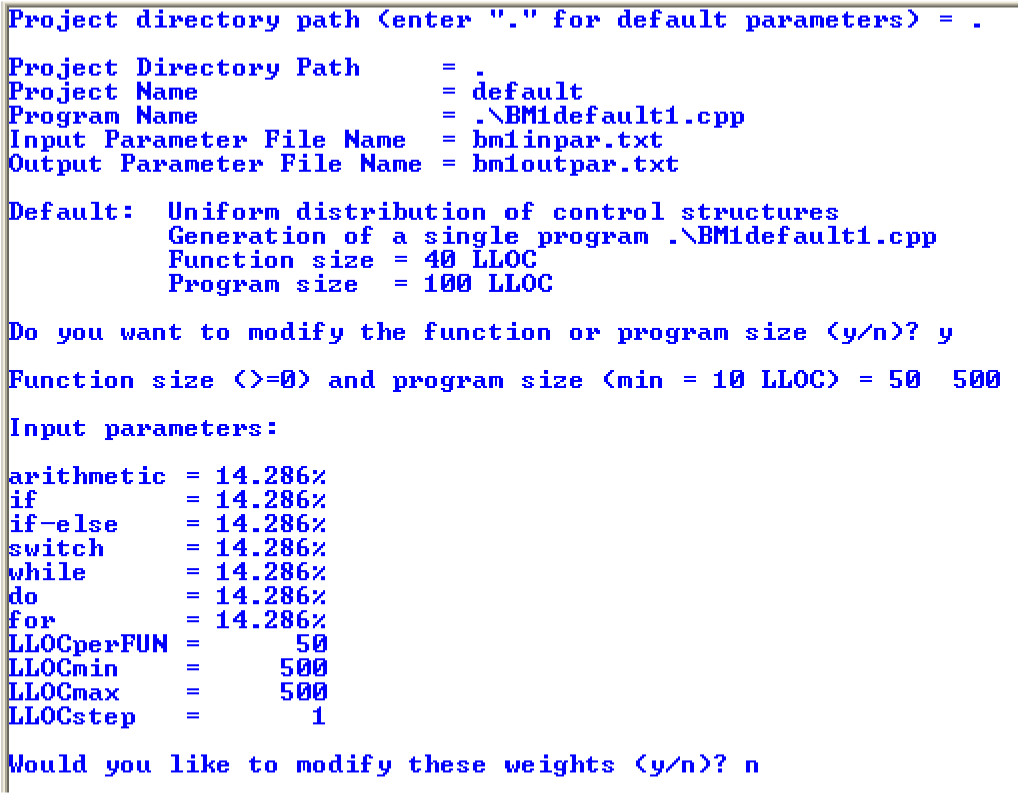
Benchmaker 1.6 demo: Generating C++ programs

bm16.exe

1. Make and execute a **500 LLOC** program: 10 functions, 50 PLOC/function, uniform distribution of control structures
2. Make and execute a **20,000 LLOC** program: 40 functions, 500 LLOC/function, nonuniform distribution of control structures
3. Create a **1,000,000 LLOC** program, uniform distribution of control structures

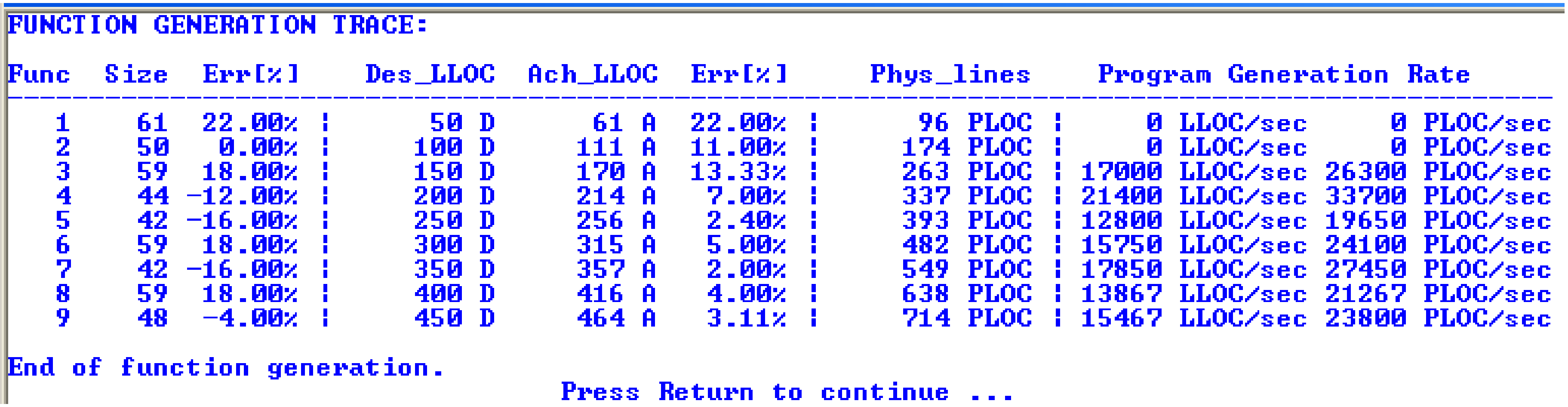
```
BM1 operation modes:
   1. Engine mode (I/O from API files)
   2. Interactive mode (I/O = Keyboard/Screen)

Application areas:
   1. Testing and performance analysis of compilers and computers
   2. Testing of source program analyzers (LOC, complexity, etc.)
   3. Visual demo of the automatic program generation process

Properties of generated benchmark programs:
   1. Program length is expressed in logical lines of code (LLOC).
   2. Generated programs consist of a sequence of functions denoted
      F1(), F2(),...,Fn(), followed by the main program.
   3. All programs contain random expressions and control structures.

The available control structures are:
[1] arithmetic    [2] if            [3] if-else       [4] switch
[5] while         [6] do            [7] for

BenchMaker1 (bm1) is normally called using a command line parameter:
   bm1  project_directory_path
   bm1 "project directory path"
Without the project directory path bm1 enters the interactive mode.

The project directory contains the following files:
   1. bm1inpar.txt  (all bm1 input data)
   2. bm1outpar.txt (names and parameters of generated output files)
   3. All generated source C++ program files (one or more)

The bm1inpar.txt file contains (in any order) the name of parameter
followed by the value of parameter, as in the following example:
   ARITHMETIC  0        :
   IF          1        :  Weights can be any
   IF_ELSE     2        :  nonnegative real values.
   SWITCH      3        :  bm1 will automatically
   WHILE       4        :  check and normalize
   DO          2        :  these values (sum = 1).
   FOR         3        :
   LLOCperFUN  100      :  0 or positive
   LLOCmin     200      :  positive and not less than LLOCperFUN
   LLOCmax     2000     :  not less than LLOCmin
   LLOCstep    200      :  any positive value

Conditions for values of input parameters:
   1. All frequencies must be nonnegative
   2. At least one of input frequencies must be positive (any value)
   3. Input data lines can come in any order
   4. LLOCmin > 0
   5. 0 < LLOCperFUN <= LLOCmin <= LLOCmax
   6. If LLOCperFUN = 0 then only the main program is generated
   7. If LLOCmax < LLOCmin it is automatically set equal to LLOCmin
   8. If LLOCstep < 1 it is automatically set to 1
```

500 LLOC

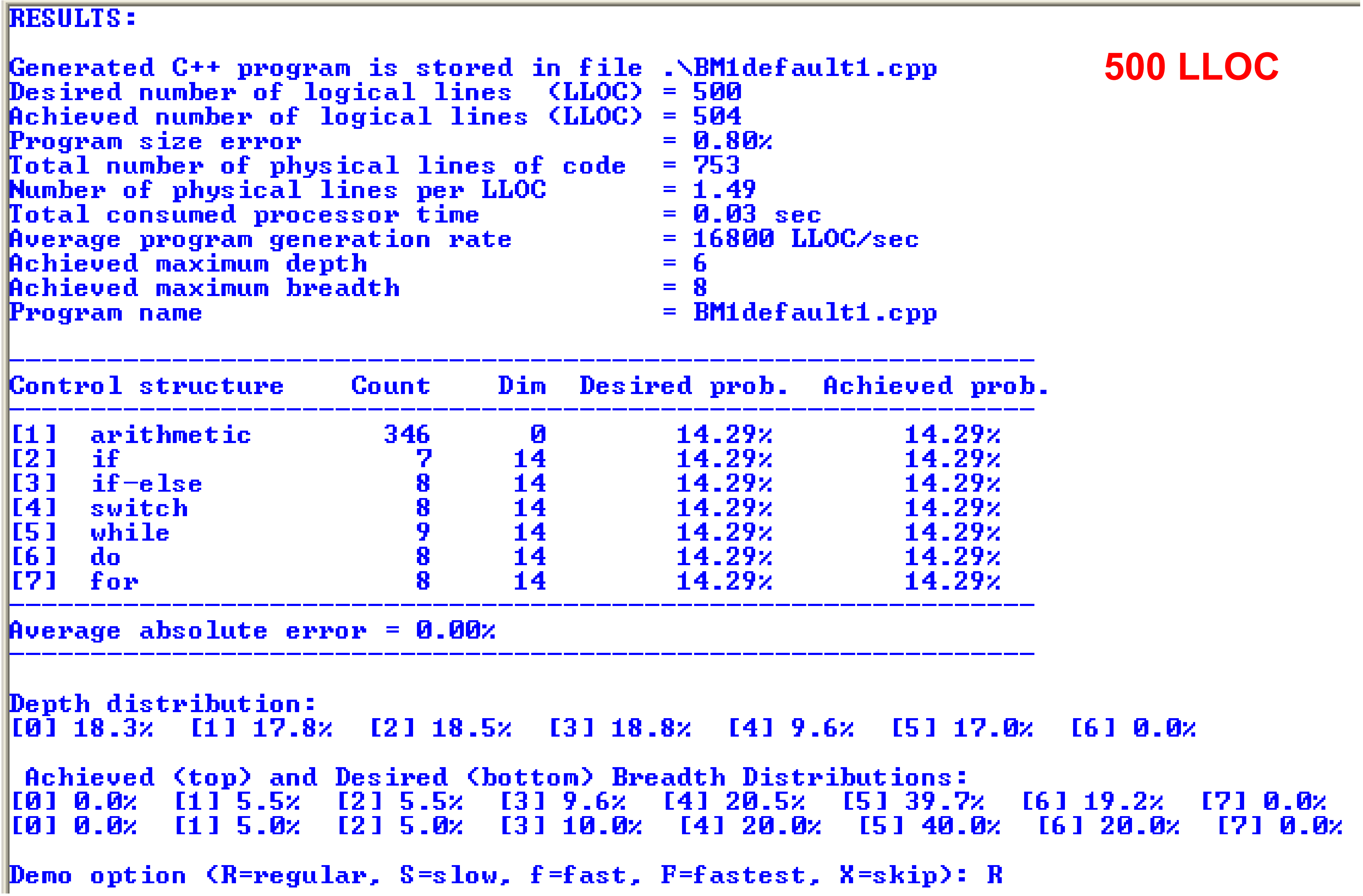


```
Project directory path (enter "." for default parameters) = .

Project Directory Path      = .
Project Name                = default
Program Name                = .\BM1default1.cpp
Input Parameter File Name   = bm1inpar.txt
Output Parameter File Name  = bm1outpar.txt

Default:  Uniform distribution of control structures
          Generation of a single program .\BM1default1.cpp
          Function size = 40 LLOC
          Program size  = 100 LLOC

Do you want to modify the function or program size (y/n)? y

Function size (>=0) and program size (min = 10 LLOC) = 50  500

Input parameters:

arithmetic = 14.286%
if         = 14.286%
if-else    = 14.286%
switch     = 14.286%
while      = 14.286%
do         = 14.286%
for        = 14.286%
LLOCperFUN =       50
LLOCmin    =      500
LLOCmax    =      500
LLOCstep   =        1

Would you like to modify these weights (y/n)? n
```

500 LLOC



```
FUNCTION GENERATION TRACE:

Func  Size  Err[%]    Des_LLOC  Ach_LLOC  Err[%]     Phys_lines    Program Generation Rate
------------------------------------------------------------------------------------------------
   1    61  22.00% |      50 D      61 A  22.00% |      96 PLOC |     0 LLOC/sec     0 PLOC/sec
   2    50   0.00% |     100 D     111 A  11.00% |     174 PLOC |     0 LLOC/sec     0 PLOC/sec
   3    59  18.00% |     150 D     170 A  13.33% |     263 PLOC | 17000 LLOC/sec 26300 PLOC/sec
   4    44 -12.00% |     200 D     214 A   7.00% |     337 PLOC | 21400 LLOC/sec 33700 PLOC/sec
   5    42 -16.00% |     250 D     256 A   2.40% |     393 PLOC | 12800 LLOC/sec 19650 PLOC/sec
   6    59  18.00% |     300 D     315 A   5.00% |     482 PLOC | 15750 LLOC/sec 24100 PLOC/sec
   7    42 -16.00% |     350 D     357 A   2.00% |     549 PLOC | 17850 LLOC/sec 27450 PLOC/sec
   8    59  18.00% |     400 D     416 A   4.00% |     638 PLOC | 13867 LLOC/sec 21267 PLOC/sec
   9    48  -4.00% |     450 D     464 A   3.11% |     714 PLOC | 15467 LLOC/sec 23800 PLOC/sec

End of function generation.
                                        Press Return to continue ...
```

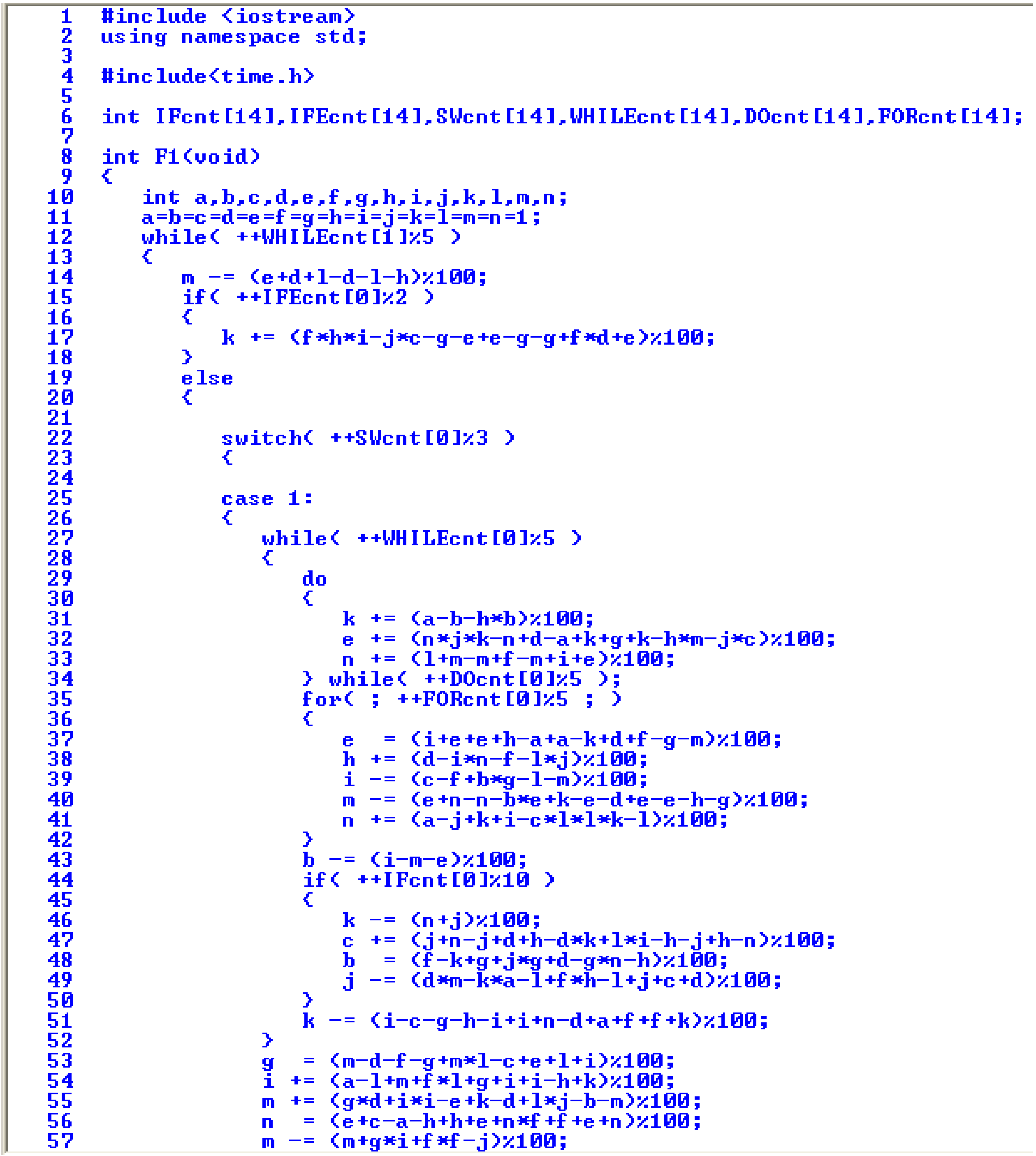


```
RESULTS:

Generated C++ program is stored in file .\BM1default1.cpp
Desired number of logical lines  (LLOC) = 500
Achieved number of logical lines (LLOC) = 504
Program size error                      = 0.80%
Total number of physical lines of code  = 753
Number of physical lines per LLOC       = 1.49
Total consumed processor time           = 0.03 sec
Average program generation rate         = 16800 LLOC/sec
Achieved maximum depth                  = 6
Achieved maximum breadth                = 8
Program name                            = BM1default1.cpp

---------------------------------------------------------------
Control structure    Count    Dim  Desired prob.  Achieved prob.
---------------------------------------------------------------
[1]  arithmetic        346      0      14.29%        14.29%
[2]  if                  7     14      14.29%        14.29%
[3]  if-else             8     14      14.29%        14.29%
[4]  switch              8     14      14.29%        14.29%
[5]  while               9     14      14.29%        14.29%
[6]  do                  8     14      14.29%        14.29%
[7]  for                 8     14      14.29%        14.29%
---------------------------------------------------------------
Average absolute error = 0.00%
---------------------------------------------------------------

Depth distribution:
[0] 18.3%  [1] 17.8%  [2] 18.5%  [3] 18.8%  [4] 9.6%  [5] 17.0%  [6] 0.0%

 Achieved (top) and Desired (bottom) Breadth Distributions:
[0] 0.0%  [1] 5.5%  [2] 5.5%  [3] 9.6%  [4] 20.5%  [5] 39.7%  [6] 19.2%  [7] 0.0%
[0] 0.0%  [1] 5.0%  [2] 5.0%  [3] 10.0%  [4] 20.0%  [5] 40.0%  [6] 20.0%  [7] 0.0%

Demo option (R=regular, S=slow, f=fast, F=fastest, X=skip): R
```

500 LLOC

500 LLOC

Beginning of generated C++ program



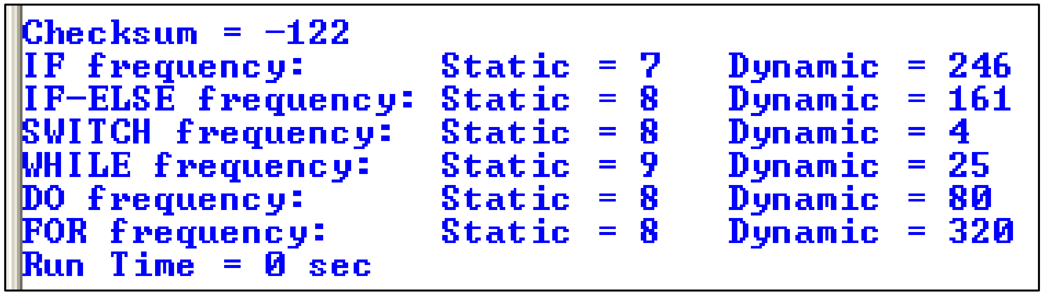
```cpp
#include <iostream>
using namespace std;

#include<time.h>

int IFcnt[14],IFEcnt[14],SWcnt[14],WHILEcnt[14],DOcnt[14],FORcnt[14];

int F1(void)
{
    int a,b,c,d,e,f,g,h,i,j,k,l,m,n;
    a=b=c=d=e=f=g=h=i=j=k=l=m=n=1;
    while( ++WHILEcnt[1]%5 )
    {
        m -= (e+d+l-d-l-h)%100;
        if( ++IFEcnt[0]%2 )
        {
            k += (f*h*i-j*c-g-e+e-g-g+f*d+e)%100;
        }
        else
        {

            switch( ++SWcnt[0]%3 )
            {

            case 1:
            {
                while( ++WHILEcnt[0]%5 )
                {
                    do
                    {
                        k += (a-b-h*b)%100;
                        e += (n*j*k-n+d-a+k+g+k-h*m-j*c)%100;
                        n += (l+m-m+f-m+i+e)%100;
                    } while( ++DOcnt[0]%5 );
                    for( ; ++FORcnt[0]%5 ; )
                    {
                        e  = (i+e+e+h-a+a-k+d+f-g-m)%100;
                        h += (d-i*n-f-l*j)%100;
                        i -= (c-f+b*g-l-m)%100;
                        m -= (e+n-n-b*e+k-e-d+e-e-h-g)%100;
                        n += (a-j+k+i-c*l*l*k-l)%100;
                    }
                    b -= (i-m-e)%100;
                    if( ++IFcnt[0]%10 )
                    {
                        k -= (n+j)%100;
                        c += (j+n-j+d+h-d*k+l*i-h-j+h-n)%100;
                        b  = (f-k+g+j*g+d-g*n-h)%100;
                        j -= (d*m-k*a-l+f*h-l+j+c+d)%100;
                    }
                    k -= (i-c-g-h-i+i+n-d+a+f+f+k)%100;
                }
                g  = (m-d-f-g+m*l-c+e+l+i)%100;
                i += (a-l+m+f*l+g+i+i-h+k)%100;
                m += (g*d+i*i-e+k-d+l*j-b-m)%100;
                n  = (e+c-a-h+h+e+n*f+f+e+n)%100;
                m -= (m+g*i+f*f-j)%100;
```

500 LLOC

End of generated C++ program

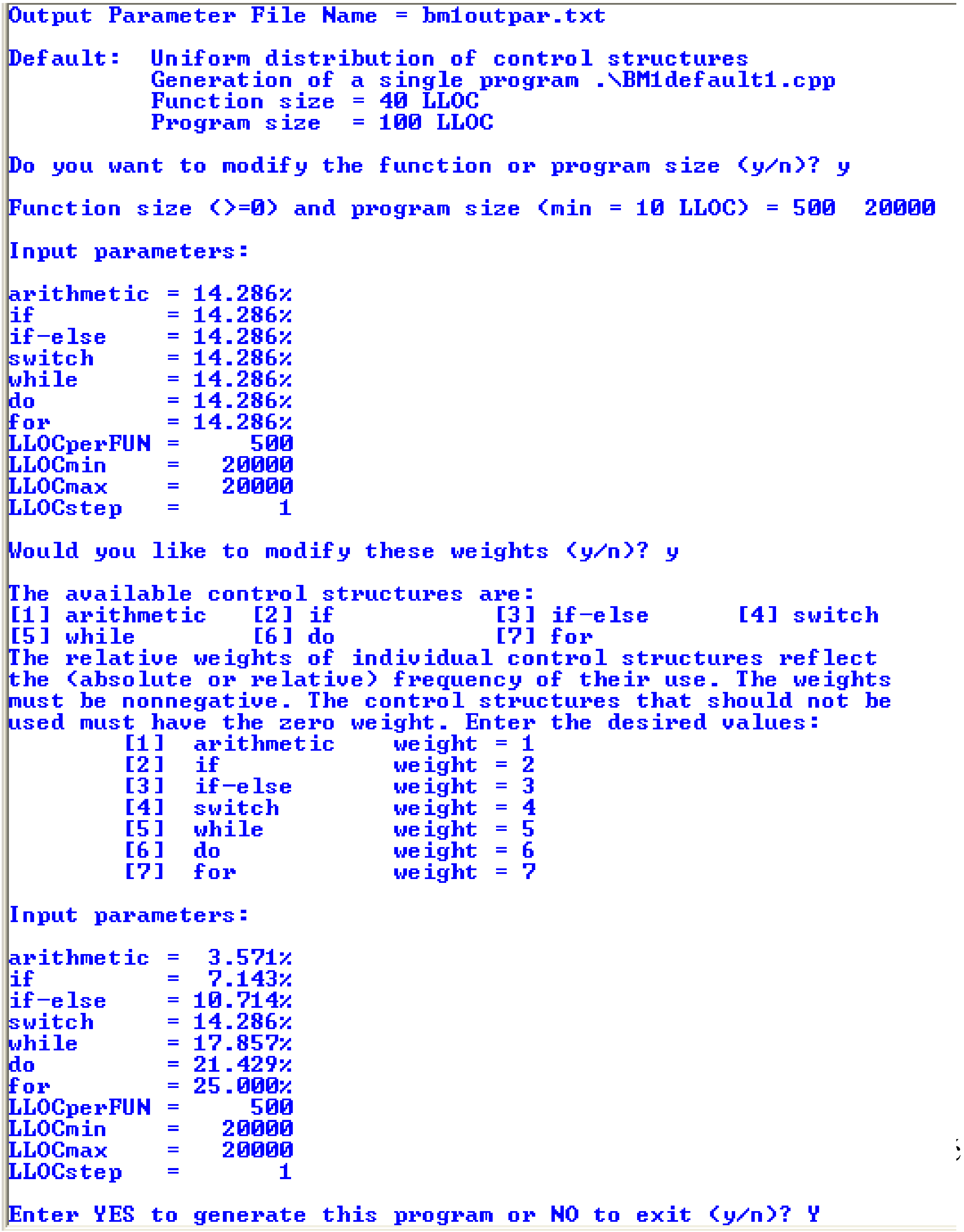

BM1default1.cpp
C++ Source file
20 KB

```
      c -= (g+g+e-i+m+m+g+c*h+l+a+g+l)%100;
      g += (j+c-b*k+e*f)%100;
      b += (l+j+f-m+m-l-j-k-n+k-i)%100;
      a += (b*m*c-f)%100;
   }
   g -= (b+f*a*j-a+n+e+g*j*b+f-a)%100;
   l -= (l*f+n)%100;
   h  = (a+d+k*m+g+h-c-d)%100;
   k += (c-d*a+i+a*c+i*m-n*i+j+h*f*l)%100;
   k += (i-a)%100;
   a += (f*k)%100;
   i -= (j-j)%100;
   return (a+b+c+d+e+f+g+h+i+j+k+l+m+n)%100 ;
}

int main(void)
{
   int I;
   clock_t StartTick = clock();
   for(I=0; I<14; I++) IFcnt[I]   =0;
   for(I=0; I<14; I++) IFEcnt[I]  =0;
   for(I=0; I<14; I++) SWcnt[I]   =0;
   for(I=0; I<14; I++) WHILEcnt[I]=0;
   for(I=0; I<14; I++) DOcnt[I]   =0;
   for(I=0; I<14; I++) FORcnt[I]  =0;
   long int sum=0;

   sum += F1( ) ;
   sum += F2( ) ;
   sum += F3( ) ;
   sum += F4( ) ;
   sum += F5( ) ;
   sum += F6( ) ;
   sum += F7( ) ;
   sum += F8( ) ;
   sum += F9( ) ;

   cout << "\nChecksum = " << sum;
   for(I=sum=0; I<7; I++) sum += IFcnt[I];
   cout << "\nIF frequency:      Static = " << 7 << "   Dynamic = " << sum ;
   for(I=sum=0; I<8; I++) sum += IFEcnt[I];
   cout << "\nIF-ELSE frequency: Static = " << 8 << "   Dynamic = " << sum ;
   for(I=sum=0; I<8; I++) sum += SWcnt[I];
   cout << "\nSWITCH frequency:  Static = " << 8 << "   Dynamic = " << sum ;
   for(I=sum=0; I<9; I++) sum += WHILEcnt[I];
   cout << "\nWHILE frequency:   Static = " << 9 << "   Dynamic = " << sum ;
   for(I=sum=0; I<8; I++) sum += DOcnt[I];
   cout << "\nDO frequency:      Static = " << 8 << "   Dynamic = " << sum ;
   for(I=sum=0; I<8; I++) sum += FORcnt[I];
   cout << "\nFOR frequency:     Static = " << 8 << "   Dynamic = " << sum ;
   cout << "\nRun Time = " << double(clock()-StartTick)/CLOCKS_PER_SEC << " sec\n\n";

   return 0;
}
```



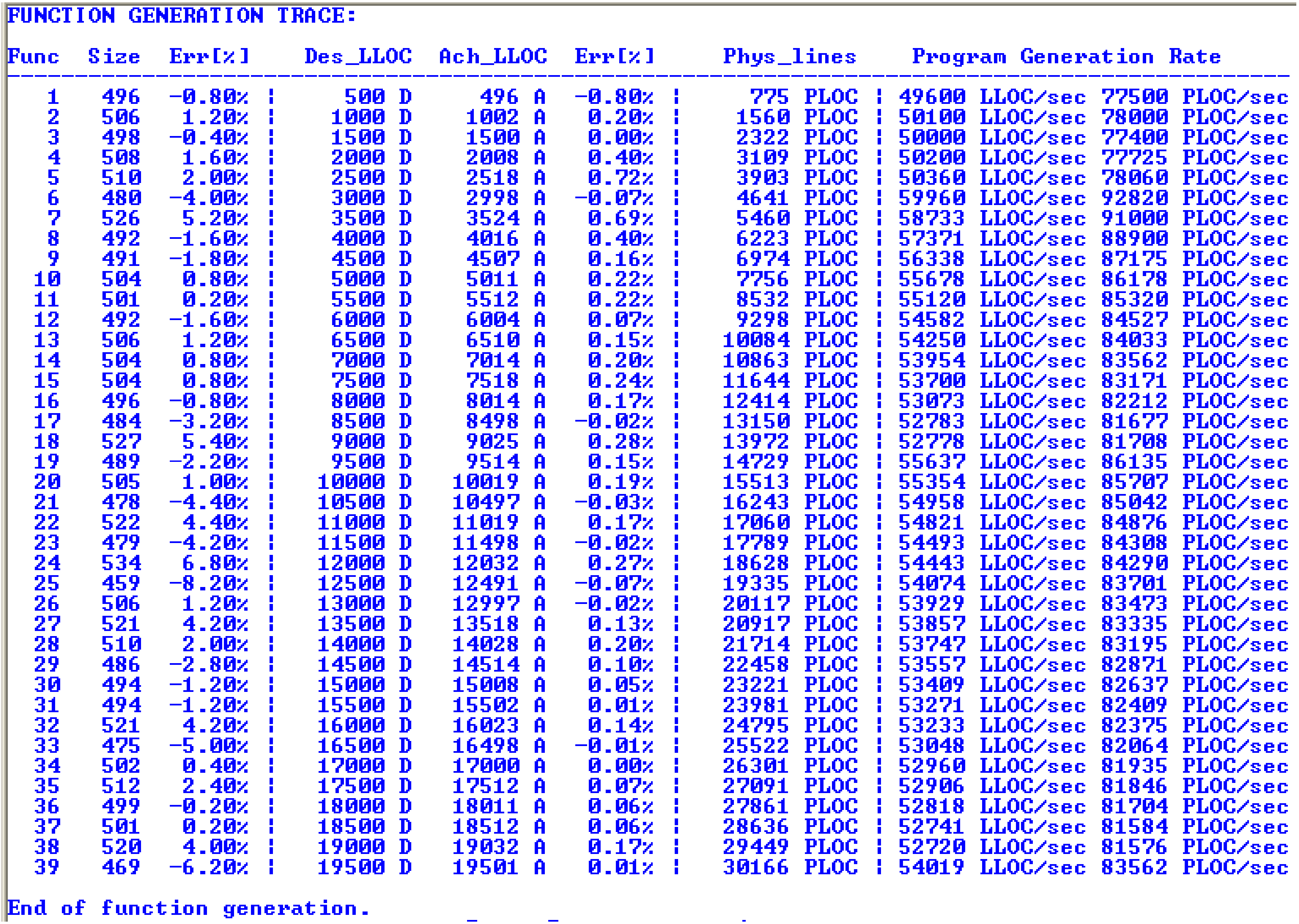
```
Checksum = -122
IF frequency:      Static = 7   Dynamic = 246
IF-ELSE frequency: Static = 8   Dynamic = 161
SWITCH frequency:  Static = 8   Dynamic = 4
WHILE frequency:   Static = 9   Dynamic = 25
DO frequency:      Static = 8   Dynamic = 80
FOR frequency:     Static = 8   Dynamic = 320
Run Time = 0 sec
```

20,000 LLOC

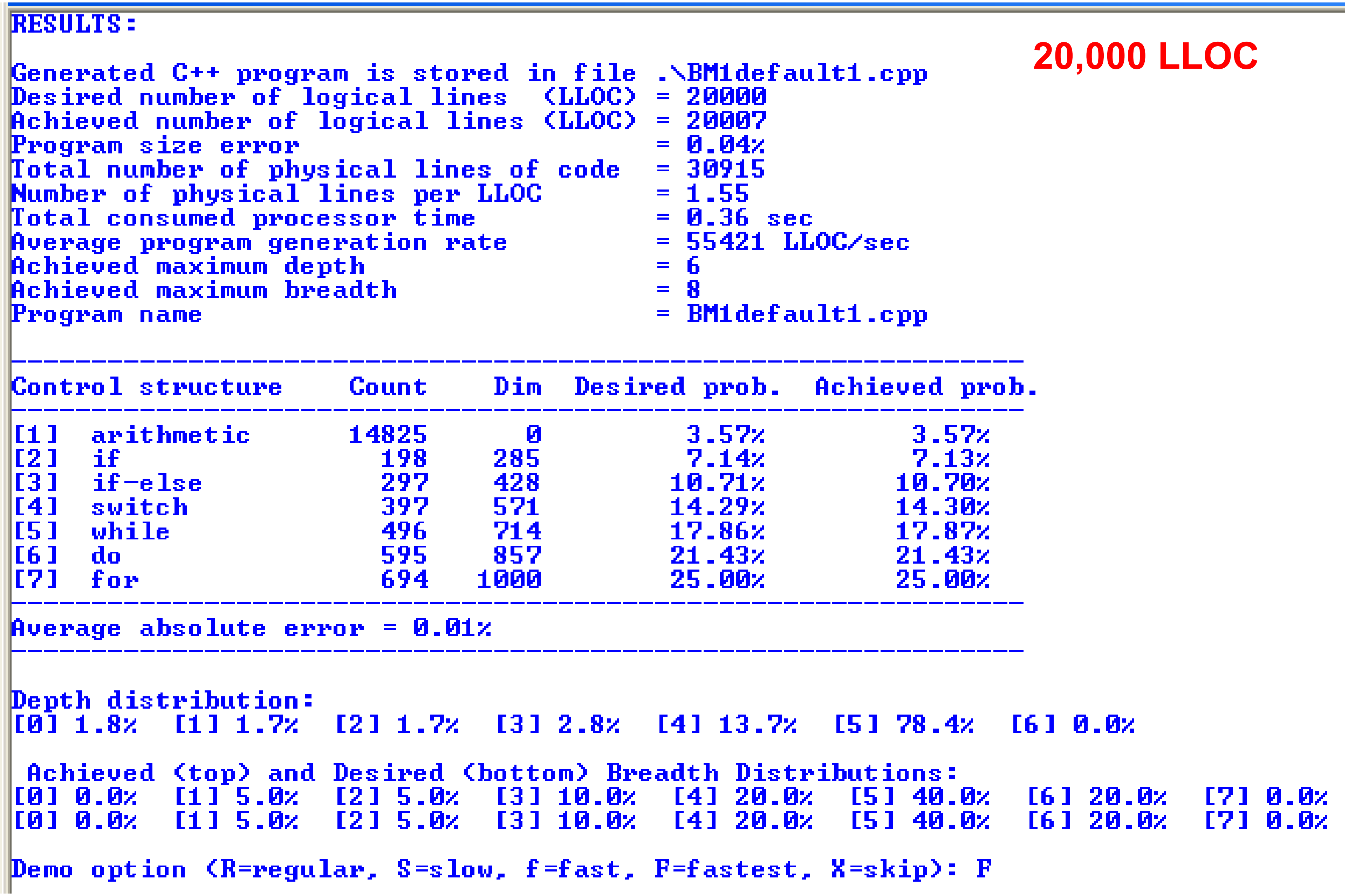


```
Output Parameter File Name = bm1outpar.txt

Default:  Uniform distribution of control structures
          Generation of a single program .\BM1default1.cpp
          Function size = 40 LLOC
          Program size  = 100 LLOC

Do you want to modify the function or program size (y/n)? y

Function size (>=0) and program size (min = 10 LLOC) = 500  20000

Input parameters:

arithmetic = 14.286%
if         = 14.286%
if-else    = 14.286%
switch     = 14.286%
while      = 14.286%
do         = 14.286%
for        = 14.286%
LLOCperFUN =     500
LLOCmin    =   20000
LLOCmax    =   20000
LLOCstep   =       1

Would you like to modify these weights (y/n)? y

The available control structures are:
[1] arithmetic   [2] if           [3] if-else      [4] switch
[5] while        [6] do           [7] for
The relative weights of individual control structures reflect
the (absolute or relative) frequency of their use. The weights
must be nonnegative. The control structures that should not be
used must have the zero weight. Enter the desired values:
        [1]  arithmetic     weight = 1
        [2]  if             weight = 2
        [3]  if-else        weight = 3
        [4]  switch         weight = 4
        [5]  while          weight = 5
        [6]  do             weight = 6
        [7]  for            weight = 7

Input parameters:

arithmetic =  3.571%
if         =  7.143%
if-else    = 10.714%
switch     = 14.286%
while      = 17.857%
do         = 21.429%
for        = 25.000%
LLOCperFUN =     500
LLOCmin    =   20000
LLOCmax    =   20000
LLOCstep   =       1

Enter YES to generate this program or NO to exit (y/n)? Y
```

20,000 LLOC

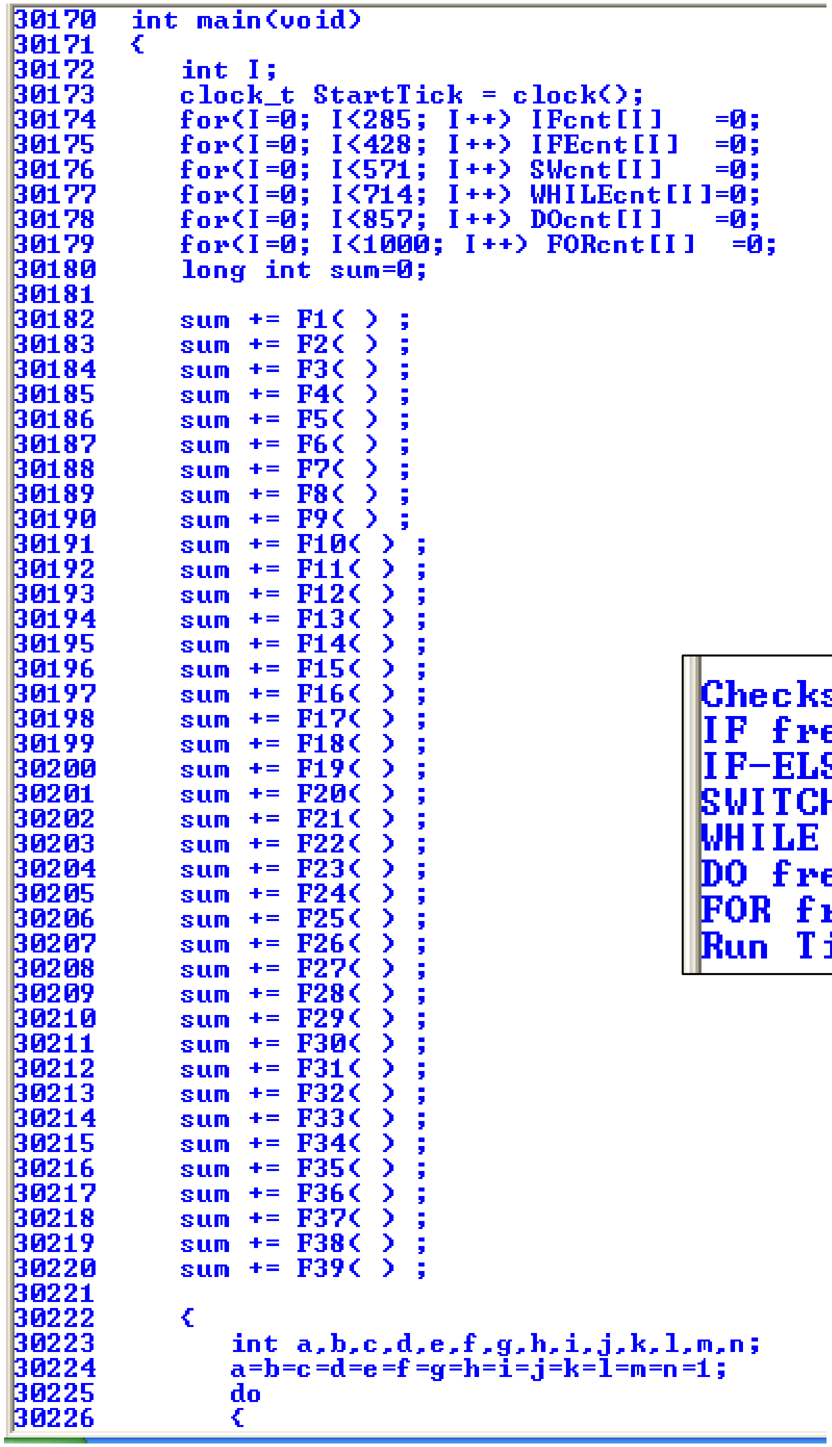


```
FUNCTION GENERATION TRACE:

Func  Size  Err[%]    Des_LLOC  Ach_LLOC  Err[%]     Phys_lines    Program Generation Rate
-----------------------------------------------------------------------------------------------
   1   496  -0.80% ¦     500 D    496 A  -0.80% ¦     775 PLOC ¦ 49600 LLOC/sec 77500 PLOC/sec
   2   506   1.20% ¦    1000 D   1002 A   0.20% ¦    1560 PLOC ¦ 50100 LLOC/sec 78000 PLOC/sec
   3   498  -0.40% ¦    1500 D   1500 A   0.00% ¦    2322 PLOC ¦ 50000 LLOC/sec 77400 PLOC/sec
   4   508   1.60% ¦    2000 D   2008 A   0.40% ¦    3109 PLOC ¦ 50200 LLOC/sec 77725 PLOC/sec
   5   510   2.00% ¦    2500 D   2518 A   0.72% ¦    3903 PLOC ¦ 50360 LLOC/sec 78060 PLOC/sec
   6   480  -4.00% ¦    3000 D   2998 A  -0.07% ¦    4641 PLOC ¦ 59960 LLOC/sec 92820 PLOC/sec
   7   526   5.20% ¦    3500 D   3524 A   0.69% ¦    5460 PLOC ¦ 58733 LLOC/sec 91000 PLOC/sec
   8   492  -1.60% ¦    4000 D   4016 A   0.40% ¦    6223 PLOC ¦ 57371 LLOC/sec 88900 PLOC/sec
   9   491  -1.80% ¦    4500 D   4507 A   0.16% ¦    6974 PLOC ¦ 56338 LLOC/sec 87175 PLOC/sec
  10   504   0.80% ¦    5000 D   5011 A   0.22% ¦    7756 PLOC ¦ 55678 LLOC/sec 86178 PLOC/sec
  11   501   0.20% ¦    5500 D   5512 A   0.22% ¦    8532 PLOC ¦ 55120 LLOC/sec 85320 PLOC/sec
  12   492  -1.60% ¦    6000 D   6004 A   0.07% ¦    9298 PLOC ¦ 54582 LLOC/sec 84527 PLOC/sec
  13   506   1.20% ¦    6500 D   6510 A   0.15% ¦   10084 PLOC ¦ 54250 LLOC/sec 84033 PLOC/sec
  14   504   0.80% ¦    7000 D   7014 A   0.20% ¦   10863 PLOC ¦ 53954 LLOC/sec 83562 PLOC/sec
  15   504   0.80% ¦    7500 D   7518 A   0.24% ¦   11644 PLOC ¦ 53700 LLOC/sec 83171 PLOC/sec
  16   496  -0.80% ¦    8000 D   8014 A   0.17% ¦   12414 PLOC ¦ 53073 LLOC/sec 82212 PLOC/sec
  17   484  -3.20% ¦    8500 D   8498 A  -0.02% ¦   13150 PLOC ¦ 52783 LLOC/sec 81677 PLOC/sec
  18   527   5.40% ¦    9000 D   9025 A   0.28% ¦   13972 PLOC ¦ 52778 LLOC/sec 81708 PLOC/sec
  19   489  -2.20% ¦    9500 D   9514 A   0.15% ¦   14729 PLOC ¦ 55637 LLOC/sec 86135 PLOC/sec
  20   505   1.00% ¦   10000 D  10019 A   0.19% ¦   15513 PLOC ¦ 55354 LLOC/sec 85707 PLOC/sec
  21   478  -4.40% ¦   10500 D  10497 A  -0.03% ¦   16243 PLOC ¦ 54958 LLOC/sec 85042 PLOC/sec
  22   522   4.40% ¦   11000 D  11019 A   0.17% ¦   17060 PLOC ¦ 54821 LLOC/sec 84876 PLOC/sec
  23   479  -4.20% ¦   11500 D  11498 A  -0.02% ¦   17789 PLOC ¦ 54493 LLOC/sec 84308 PLOC/sec
  24   534   6.80% ¦   12000 D  12032 A   0.27% ¦   18628 PLOC ¦ 54443 LLOC/sec 84290 PLOC/sec
  25   459  -8.20% ¦   12500 D  12491 A  -0.07% ¦   19335 PLOC ¦ 54074 LLOC/sec 83701 PLOC/sec
  26   506   1.20% ¦   13000 D  12997 A  -0.02% ¦   20117 PLOC ¦ 53929 LLOC/sec 83473 PLOC/sec
  27   521   4.20% ¦   13500 D  13518 A   0.13% ¦   20917 PLOC ¦ 53857 LLOC/sec 83335 PLOC/sec
  28   510   2.00% ¦   14000 D  14028 A   0.20% ¦   21714 PLOC ¦ 53747 LLOC/sec 83195 PLOC/sec
  29   486  -2.80% ¦   14500 D  14514 A   0.10% ¦   22458 PLOC ¦ 53557 LLOC/sec 82871 PLOC/sec
  30   494  -1.20% ¦   15000 D  15008 A   0.05% ¦   23221 PLOC ¦ 53409 LLOC/sec 82637 PLOC/sec
  31   494  -1.20% ¦   15500 D  15502 A   0.01% ¦   23981 PLOC ¦ 53271 LLOC/sec 82409 PLOC/sec
  32   521   4.20% ¦   16000 D  16023 A   0.14% ¦   24795 PLOC ¦ 53233 LLOC/sec 82375 PLOC/sec
  33   475  -5.00% ¦   16500 D  16498 A  -0.01% ¦   25522 PLOC ¦ 53048 LLOC/sec 82064 PLOC/sec
  34   502   0.40% ¦   17000 D  17000 A   0.00% ¦   26301 PLOC ¦ 52960 LLOC/sec 81935 PLOC/sec
  35   512   2.40% ¦   17500 D  17512 A   0.07% ¦   27091 PLOC ¦ 52906 LLOC/sec 81846 PLOC/sec
  36   499  -0.20% ¦   18000 D  18011 A   0.06% ¦   27861 PLOC ¦ 52818 LLOC/sec 81704 PLOC/sec
  37   501   0.20% ¦   18500 D  18512 A   0.06% ¦   28636 PLOC ¦ 52741 LLOC/sec 81584 PLOC/sec
  38   520   4.00% ¦   19000 D  19032 A   0.17% ¦   29449 PLOC ¦ 52720 LLOC/sec 81576 PLOC/sec
  39   469  -6.20% ¦   19500 D  19501 A   0.01% ¦   30166 PLOC ¦ 54019 LLOC/sec 83562 PLOC/sec

End of function generation.
```



20,000 LLOC

```
RESULTS:

Generated C++ program is stored in file .\BM1default1.cpp
Desired number of logical lines  (LLOC) = 20000
Achieved number of logical lines (LLOC) = 20007
Program size error                      = 0.04%
Total number of physical lines of code  = 30915
Number of physical lines per LLOC       = 1.55
Total consumed processor time           = 0.36 sec
Average program generation rate         = 55421 LLOC/sec
Achieved maximum depth                  = 6
Achieved maximum breadth                = 8
Program name                            = BM1default1.cpp

---------------------------------------------------------------
Control structure    Count    Dim  Desired prob.  Achieved prob.
---------------------------------------------------------------
[1]  arithmetic      14825      0         3.57%         3.57%
[2]  if                198    285         7.14%         7.13%
[3]  if-else           297    428        10.71%        10.70%
[4]  switch            397    571        14.29%        14.30%
[5]  while             496    714        17.86%        17.87%
[6]  do                595    857        21.43%        21.43%
[7]  for               694   1000        25.00%        25.00%
---------------------------------------------------------------
Average absolute error = 0.01%
---------------------------------------------------------------

Depth distribution:
[0] 1.8%  [1] 1.7%  [2] 1.7%  [3] 2.8%  [4] 13.7%  [5] 78.4%  [6] 0.0%

 Achieved (top) and Desired (bottom) Breadth Distributions:
[0] 0.0%  [1] 5.0%  [2] 5.0%  [3] 10.0%  [4] 20.0%  [5] 40.0%  [6] 20.0%  [7] 0.0%
[0] 0.0%  [1] 5.0%  [2] 5.0%  [3] 10.0%  [4] 20.0%  [5] 40.0%  [6] 20.0%  [7] 0.0%

Demo option (R=regular, S=slow, f=fast, F=fastest, X=skip): F
```

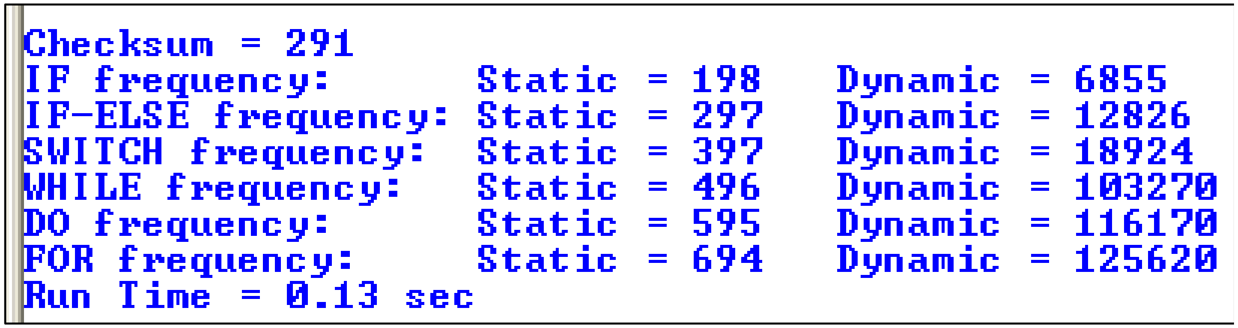


```
int main(void)
{
    int I;
    clock_t StartTick = clock();
    for(I=0; I<285; I++) IFcnt[I]    =0;
    for(I=0; I<428; I++) IFEcnt[I]   =0;
    for(I=0; I<571; I++) SWcnt[I]    =0;
    for(I=0; I<714; I++) WHILEcnt[I]=0;
    for(I=0; I<857; I++) DOcnt[I]    =0;
    for(I=0; I<1000; I++) FORcnt[I]   =0;
    long int sum=0;

    sum += F1( ) ;
    sum += F2( ) ;
    sum += F3( ) ;
    sum += F4( ) ;
    sum += F5( ) ;
    sum += F6( ) ;
    sum += F7( ) ;
    sum += F8( ) ;
    sum += F9( ) ;
    sum += F10( ) ;
    sum += F11( ) ;
    sum += F12( ) ;
    sum += F13( ) ;
    sum += F14( ) ;
    sum += F15( ) ;
    sum += F16( ) ;
    sum += F17( ) ;
    sum += F18( ) ;
    sum += F19( ) ;
    sum += F20( ) ;
    sum += F21( ) ;
    sum += F22( ) ;
    sum += F23( ) ;
    sum += F24( ) ;
    sum += F25( ) ;
    sum += F26( ) ;
    sum += F27( ) ;
    sum += F28( ) ;
    sum += F29( ) ;
    sum += F30( ) ;
    sum += F31( ) ;
    sum += F32( ) ;
    sum += F33( ) ;
    sum += F34( ) ;
    sum += F35( ) ;
    sum += F36( ) ;
    sum += F37( ) ;
    sum += F38( ) ;
    sum += F39( ) ;

    {
        int a,b,c,d,e,f,g,h,i,j,k,l,m,n;
        a=b=c=d=e=f=g=h=i=j=k=l=m=n=1;
        do
        {
```

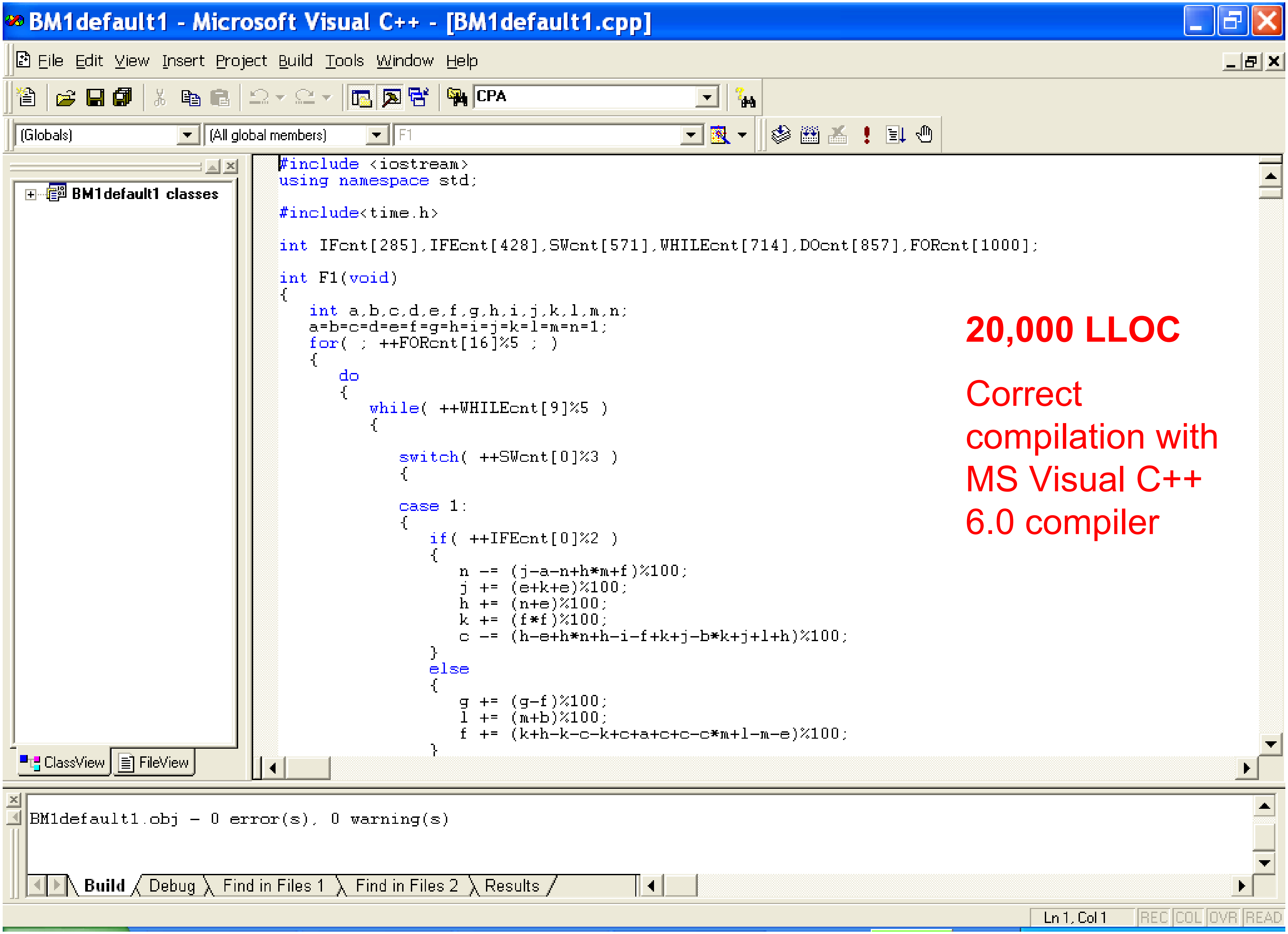
20,000 LLOC

A segment of generated main C++ program

BM1default1.cpp
C++ Source file
978 KB



```
Checksum = 291
IF frequency:       Static = 198    Dynamic = 6855
IF-ELSE frequency:  Static = 297    Dynamic = 12826
SWITCH frequency:   Static = 397    Dynamic = 18924
WHILE frequency:    Static = 496    Dynamic = 103270
DO frequency:       Static = 595    Dynamic = 116170
FOR frequency:      Static = 694    Dynamic = 125620
Run Time = 0.13 sec
```

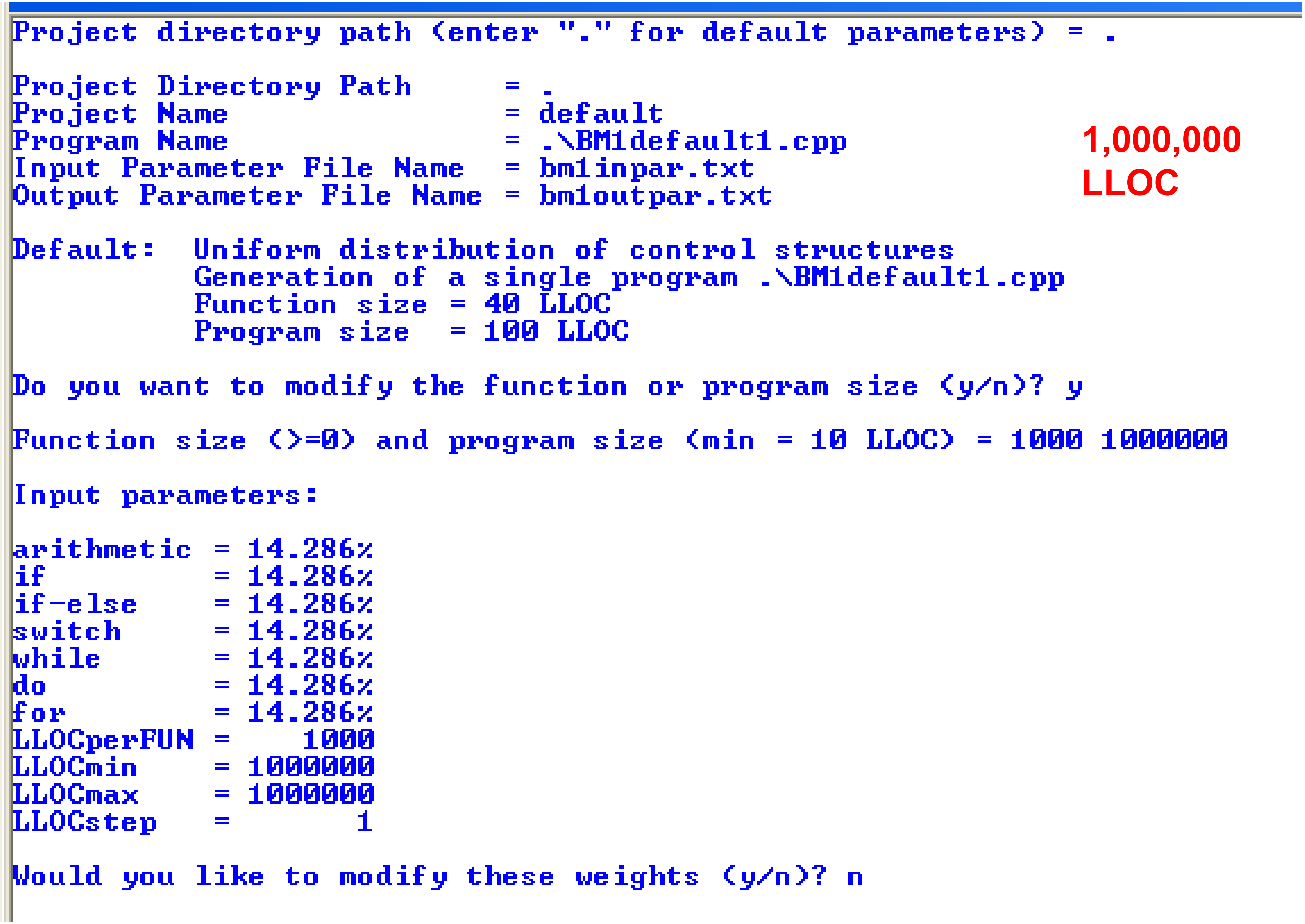


```
#include <iostream>
using namespace std;

#include<time.h>

int IFcnt[285],IFEcnt[428],SWcnt[571],WHILEcnt[714],DOcnt[857],FORcnt[1000];

int F1(void)
{
   int a,b,c,d,e,f,g,h,i,j,k,l,m,n;
   a=b=c=d=e=f=g=h=i=j=k=l=m=n=1;
   for( ; ++FORcnt[16]%5 ; )
   {
      do
      {
         while( ++WHILEcnt[9]%5 )
         {

            switch( ++SWcnt[0]%3 )
            {

            case 1:
            {
               if( ++IFEcnt[0]%2 )
               {
                  n -= (j-a-n+h*m+f)%100;
                  j += (e+k+e)%100;
                  h += (n+e)%100;
                  k += (f*f)%100;
                  c -= (h-e+h*n+h-i-f+k+j-b*k+j+l+h)%100;
               }
               else
               {
                  g += (g-f)%100;
                  l += (m+b)%100;
                  f += (k+h-k-c-k+c+a+c+c-c*m+l-m-e)%100;
               }
```

```
BM1default1.obj - 0 error(s), 0 warning(s)
```

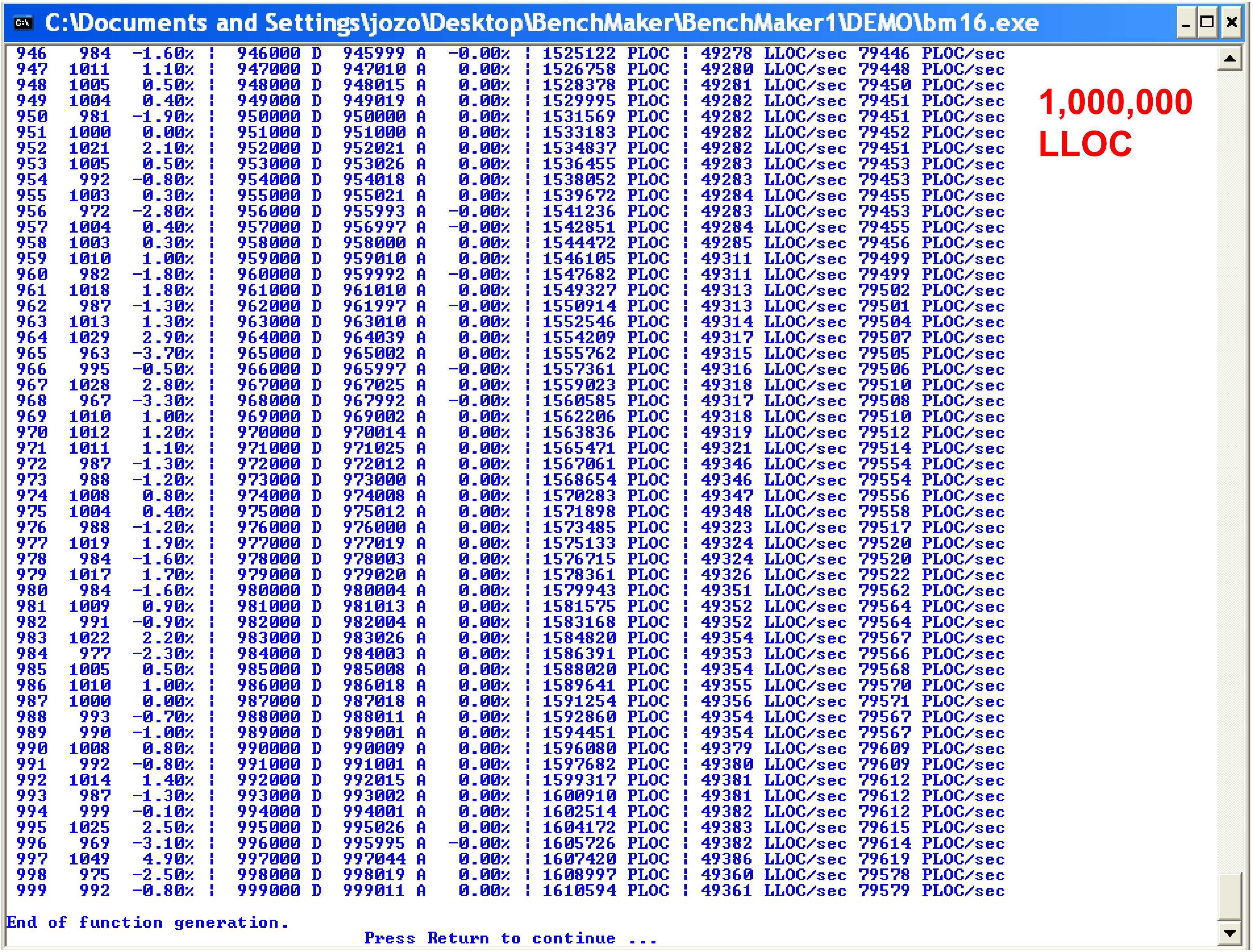
20,000 LLOC

Correct compilation with MS Visual C++ 6.0 compiler



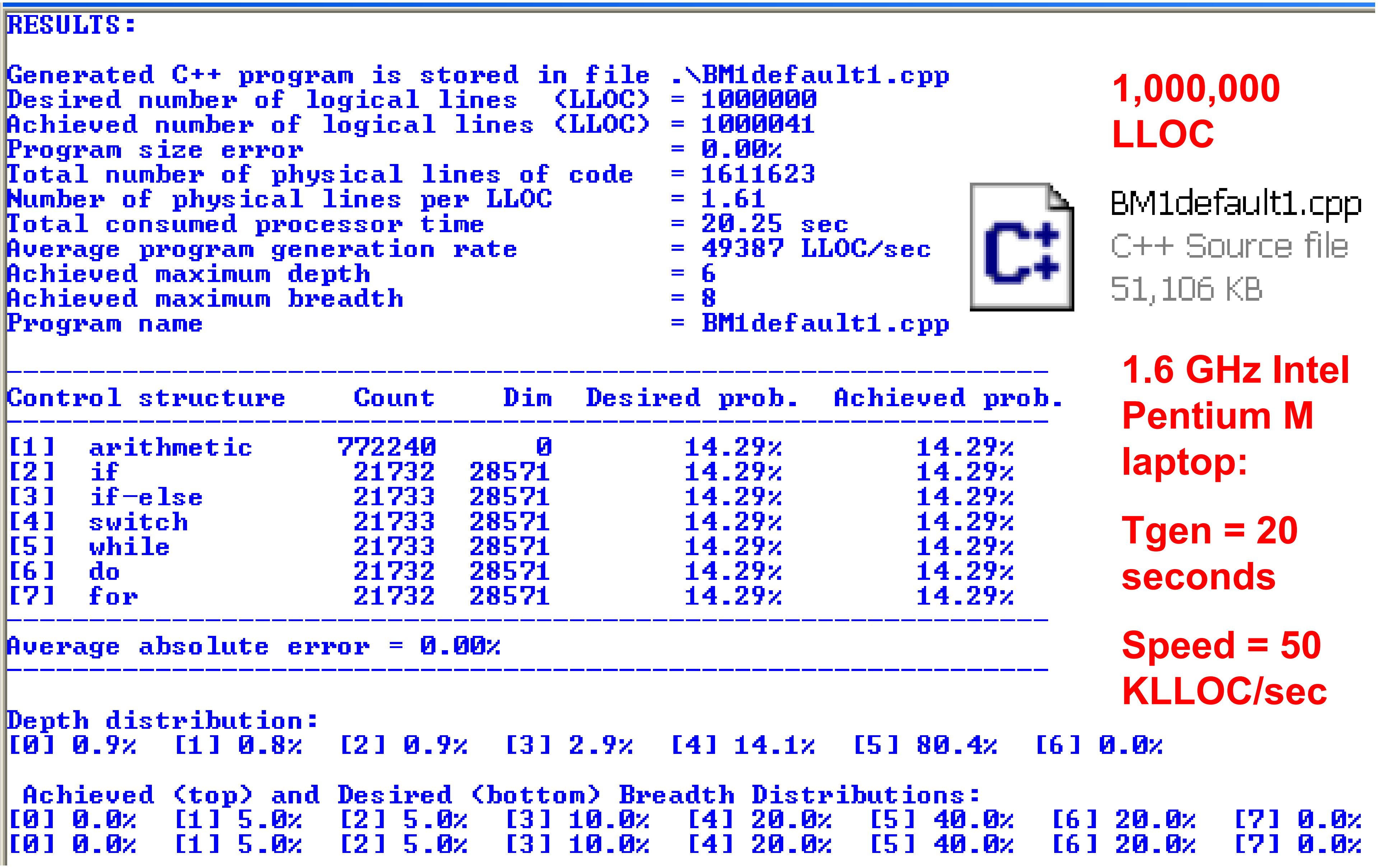
```
Project directory path (enter "." for default parameters) = .

Project Directory Path      = .
Project Name                = default
Program Name                = .\BM1default1.cpp
Input Parameter File Name   = bm1inpar.txt
Output Parameter File Name  = bm1outpar.txt

Default:  Uniform distribution of control structures
          Generation of a single program .\BM1default1.cpp
          Function size = 40 LLOC
          Program size  = 100 LLOC

Do you want to modify the function or program size (y/n)? y

Function size (>=0) and program size (min = 10 LLOC) = 1000 1000000

Input parameters:

arithmetic = 14.286%
if         = 14.286%
if-else    = 14.286%
switch     = 14.286%
while      = 14.286%
do         = 14.286%
for        = 14.286%
LLOCperFUN =    1000
LLOCmin    = 1000000
LLOCmax    = 1000000
LLOCstep   =       1

Would you like to modify these weights (y/n)? n
```

1,000,000
LLOC



```
946    984  -1.60% ¦  946000 D  945999 A  -0.00% ¦ 1525122 PLOC ¦ 49278 LLOC/sec 79446 PLOC/sec
947   1011   1.10% ¦  947000 D  947010 A   0.00% ¦ 1526758 PLOC ¦ 49280 LLOC/sec 79448 PLOC/sec
948   1005   0.50% ¦  948000 D  948015 A   0.00% ¦ 1528378 PLOC ¦ 49281 LLOC/sec 79450 PLOC/sec
949   1004   0.40% ¦  949000 D  949019 A   0.00% ¦ 1529995 PLOC ¦ 49282 LLOC/sec 79451 PLOC/sec
950    981  -1.90% ¦  950000 D  950000 A   0.00% ¦ 1531569 PLOC ¦ 49282 LLOC/sec 79451 PLOC/sec
951   1000   0.00% ¦  951000 D  951000 A   0.00% ¦ 1533183 PLOC ¦ 49282 LLOC/sec 79452 PLOC/sec
952   1021   2.10% ¦  952000 D  952021 A   0.00% ¦ 1534837 PLOC ¦ 49282 LLOC/sec 79451 PLOC/sec
953   1005   0.50% ¦  953000 D  953026 A   0.00% ¦ 1536455 PLOC ¦ 49283 LLOC/sec 79453 PLOC/sec
954    992  -0.80% ¦  954000 D  954018 A   0.00% ¦ 1538052 PLOC ¦ 49283 LLOC/sec 79453 PLOC/sec
955   1003   0.30% ¦  955000 D  955021 A   0.00% ¦ 1539672 PLOC ¦ 49284 LLOC/sec 79455 PLOC/sec
956    972  -2.80% ¦  956000 D  955993 A  -0.00% ¦ 1541236 PLOC ¦ 49283 LLOC/sec 79453 PLOC/sec
957   1004   0.40% ¦  957000 D  956997 A  -0.00% ¦ 1542851 PLOC ¦ 49284 LLOC/sec 79455 PLOC/sec
958   1003   0.30% ¦  958000 D  958000 A   0.00% ¦ 1544472 PLOC ¦ 49285 LLOC/sec 79456 PLOC/sec
959   1010   1.00% ¦  959000 D  959010 A   0.00% ¦ 1546105 PLOC ¦ 49311 LLOC/sec 79499 PLOC/sec
960    982  -1.80% ¦  960000 D  959992 A  -0.00% ¦ 1547682 PLOC ¦ 49311 LLOC/sec 79499 PLOC/sec
961   1018   1.80% ¦  961000 D  961010 A   0.00% ¦ 1549327 PLOC ¦ 49313 LLOC/sec 79502 PLOC/sec
962    987  -1.30% ¦  962000 D  961997 A  -0.00% ¦ 1550914 PLOC ¦ 49313 LLOC/sec 79501 PLOC/sec
963   1013   1.30% ¦  963000 D  963010 A   0.00% ¦ 1552546 PLOC ¦ 49314 LLOC/sec 79504 PLOC/sec
964   1029   2.90% ¦  964000 D  964039 A   0.00% ¦ 1554209 PLOC ¦ 49317 LLOC/sec 79507 PLOC/sec
965    963  -3.70% ¦  965000 D  965002 A   0.00% ¦ 1555762 PLOC ¦ 49315 LLOC/sec 79505 PLOC/sec
966    995  -0.50% ¦  966000 D  965997 A  -0.00% ¦ 1557361 PLOC ¦ 49316 LLOC/sec 79506 PLOC/sec
967   1028   2.80% ¦  967000 D  967025 A   0.00% ¦ 1559023 PLOC ¦ 49318 LLOC/sec 79510 PLOC/sec
968    967  -3.30% ¦  968000 D  967992 A  -0.00% ¦ 1560585 PLOC ¦ 49317 LLOC/sec 79508 PLOC/sec
969   1010   1.00% ¦  969000 D  969002 A   0.00% ¦ 1562206 PLOC ¦ 49318 LLOC/sec 79510 PLOC/sec
970   1012   1.20% ¦  970000 D  970014 A   0.00% ¦ 1563836 PLOC ¦ 49319 LLOC/sec 79512 PLOC/sec
971   1011   1.10% ¦  971000 D  971025 A   0.00% ¦ 1565471 PLOC ¦ 49321 LLOC/sec 79514 PLOC/sec
972    987  -1.30% ¦  972000 D  972012 A   0.00% ¦ 1567061 PLOC ¦ 49346 LLOC/sec 79554 PLOC/sec
973    988  -1.20% ¦  973000 D  973000 A   0.00% ¦ 1568654 PLOC ¦ 49346 LLOC/sec 79554 PLOC/sec
974   1008   0.80% ¦  974000 D  974008 A   0.00% ¦ 1570283 PLOC ¦ 49347 LLOC/sec 79556 PLOC/sec
975   1004   0.40% ¦  975000 D  975012 A   0.00% ¦ 1571898 PLOC ¦ 49348 LLOC/sec 79558 PLOC/sec
976    988  -1.20% ¦  976000 D  976000 A   0.00% ¦ 1573485 PLOC ¦ 49323 LLOC/sec 79517 PLOC/sec
977   1019   1.90% ¦  977000 D  977019 A   0.00% ¦ 1575133 PLOC ¦ 49324 LLOC/sec 79520 PLOC/sec
978    984  -1.60% ¦  978000 D  978003 A   0.00% ¦ 1576715 PLOC ¦ 49324 LLOC/sec 79520 PLOC/sec
979   1017   1.70% ¦  979000 D  979020 A   0.00% ¦ 1578361 PLOC ¦ 49326 LLOC/sec 79522 PLOC/sec
980    984  -1.60% ¦  980000 D  980004 A   0.00% ¦ 1579943 PLOC ¦ 49351 LLOC/sec 79562 PLOC/sec
981   1009   0.90% ¦  981000 D  981013 A   0.00% ¦ 1581575 PLOC ¦ 49352 LLOC/sec 79564 PLOC/sec
982    991  -0.90% ¦  982000 D  982004 A   0.00% ¦ 1583168 PLOC ¦ 49352 LLOC/sec 79564 PLOC/sec
983   1022   2.20% ¦  983000 D  983026 A   0.00% ¦ 1584820 PLOC ¦ 49354 LLOC/sec 79567 PLOC/sec
984    977  -2.30% ¦  984000 D  984003 A   0.00% ¦ 1586391 PLOC ¦ 49353 LLOC/sec 79566 PLOC/sec
985   1005   0.50% ¦  985000 D  985008 A   0.00% ¦ 1588020 PLOC ¦ 49354 LLOC/sec 79568 PLOC/sec
986   1010   1.00% ¦  986000 D  986018 A   0.00% ¦ 1589641 PLOC ¦ 49355 LLOC/sec 79570 PLOC/sec
987   1000   0.00% ¦  987000 D  987018 A   0.00% ¦ 1591254 PLOC ¦ 49356 LLOC/sec 79571 PLOC/sec
988    993  -0.70% ¦  988000 D  988011 A   0.00% ¦ 1592860 PLOC ¦ 49354 LLOC/sec 79567 PLOC/sec
989    990  -1.00% ¦  989000 D  989001 A   0.00% ¦ 1594451 PLOC ¦ 49354 LLOC/sec 79567 PLOC/sec
990   1008   0.80% ¦  990000 D  990009 A   0.00% ¦ 1596080 PLOC ¦ 49379 LLOC/sec 79609 PLOC/sec
991    992  -0.80% ¦  991000 D  991001 A   0.00% ¦ 1597682 PLOC ¦ 49380 LLOC/sec 79609 PLOC/sec
992   1014   1.40% ¦  992000 D  992015 A   0.00% ¦ 1599317 PLOC ¦ 49381 LLOC/sec 79612 PLOC/sec
993    987  -1.30% ¦  993000 D  993002 A   0.00% ¦ 1600910 PLOC ¦ 49381 LLOC/sec 79612 PLOC/sec
994    999  -0.10% ¦  994000 D  994001 A   0.00% ¦ 1602514 PLOC ¦ 49382 LLOC/sec 79612 PLOC/sec
995   1025   2.50% ¦  995000 D  995026 A   0.00% ¦ 1604172 PLOC ¦ 49383 LLOC/sec 79615 PLOC/sec
996    969  -3.10% ¦  996000 D  995995 A  -0.00% ¦ 1605726 PLOC ¦ 49382 LLOC/sec 79614 PLOC/sec
997   1049   4.90% ¦  997000 D  997044 A   0.00% ¦ 1607420 PLOC ¦ 49386 LLOC/sec 79619 PLOC/sec
998    975  -2.50% ¦  998000 D  998019 A   0.00% ¦ 1608997 PLOC ¦ 49360 LLOC/sec 79578 PLOC/sec
999    992  -0.80% ¦  999000 D  999011 A   0.00% ¦ 1610594 PLOC ¦ 49361 LLOC/sec 79579 PLOC/sec

End of function generation.
                                   Press Return to continue ...
```

1,000,000
LLOC



```
RESULTS:

Generated C++ program is stored in file .\BM1default1.cpp
Desired number of logical lines  (LLOC) = 1000000
Achieved number of logical lines (LLOC) = 1000041
Program size error                      = 0.00%
Total number of physical lines of code  = 1611623
Number of physical lines per LLOC       = 1.61
Total consumed processor time           = 20.25 sec
Average program generation rate         = 49387 LLOC/sec
Achieved maximum depth                  = 6
Achieved maximum breadth                = 8
Program name                            = BM1default1.cpp

---------------------------------------------------------------
Control structure     Count     Dim  Desired prob.  Achieved prob.
---------------------------------------------------------------
[1]  arithmetic      772240       0         14.29%         14.29%
[2]  if               21732   28571         14.29%         14.29%
[3]  if-else          21733   28571         14.29%         14.29%
[4]  switch           21733   28571         14.29%         14.29%
[5]  while            21733   28571         14.29%         14.29%
[6]  do               21732   28571         14.29%         14.29%
[7]  for              21732   28571         14.29%         14.29%
---------------------------------------------------------------
Average absolute error = 0.00%
---------------------------------------------------------------

Depth distribution:
[0] 0.9%  [1] 0.8%  [2] 0.9%  [3] 2.9%  [4] 14.1%  [5] 80.4%  [6] 0.0%

 Achieved (top) and Desired (bottom) Breadth Distributions:
[0] 0.0%  [1] 5.0%  [2] 5.0%  [3] 10.0%  [4] 20.0%  [5] 40.0%  [6] 20.0%  [7] 0.0%
[0] 0.0%  [1] 5.0%  [2] 5.0%  [3] 10.0%  [4] 20.0%  [5] 40.0%  [6] 20.0%  [7] 0.0%
```

1,000,000 LLOC

1.6 GHz Intel Pentium M laptop:

Tgen = 20 seconds

Speed = 50 KLLOC/sec

Summary of BM1 properties

- Easy specification of parameters
- Uniform and nonuniform distribution of control structures
- Very fast code generation (even on slow hardware)
- Very accurate control structure distribution
- Very accurate program size
- Correct compilation
- Possible execution
- Generation of individual benchmarks and their series
- Limited diversity of code (e.g. scalar data only, no file input/output, only procedural code)

BenchMaker 2 and the Kernel Insertion Program Generation Method

Goals

- Flexible adjustment of program structure
- Flexible adjustment of program size
- Flexible adjustment of execution time
- Semantic interpretation of workload characteristics
- Evaluation and comparison of compilers for different types of workload
- Evaluation and comparison of computer performance for different types of workload

Kernels

- Kernels are sequential segments of code that have a standardized structure:
  - Data definition and initialization
  - Procedural and OO data processing
  - Verification of correct results
  - Calibrated to have standardized (constant) run time (e.g. 1 sec) in order to be equally significant
- Kernels also have a clear semantic interpretation. They represent recognizable and frequently used operations; e.g.: sort, search, matrix operations (multiplication, inversion), disk operations, etc.

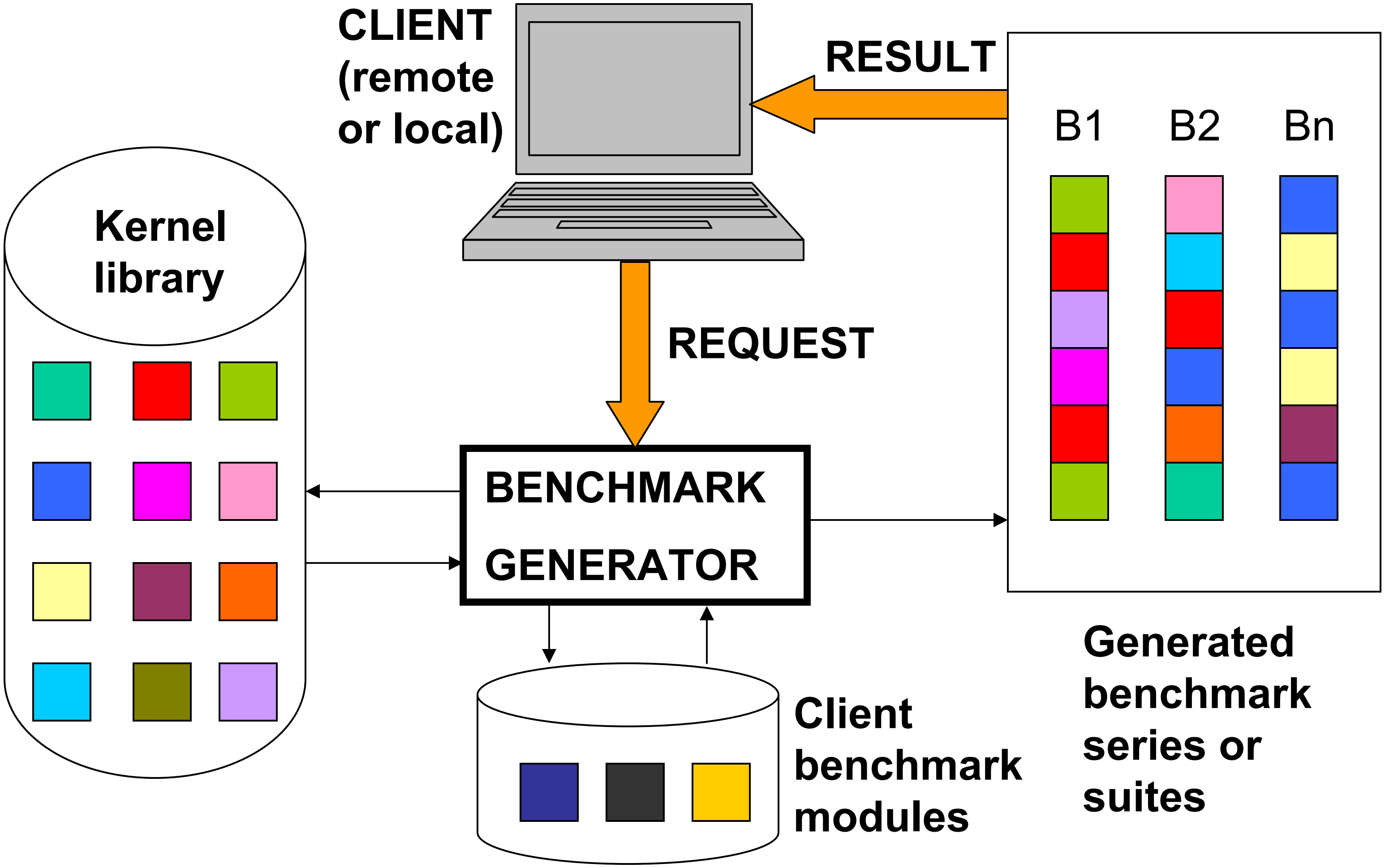
Kernel-Related Issues

- Kernel structure
- Kernel library
- Workload characterization by kernel distribution
- Benchmark workload structure
- Benchmark workload size
- BenchMaker 2 program generator
- Kernel calibration

KIN method

- Create a library of important and frequently used executable program segments called kernels. Kernels must be self contained (generate data, process data, and test the validity of results)
- Select a distribution of kernels that characterizes a desired computer workload.
- Select a desired structure of benchmark workload.
- Select a desired size of benchmark workload.
- Create the benchmark workload by adding kernels according to the selected distribution. Stop when the resulting benchmark program attains the desired size.

The Concept of Kernel Insertion



Kernel Naming and Classification

L A G S # #

L = Programming language code:
    C denotes C++
    B denotes C language
    J denotes Java
    F denotes Fortran
A = Area code (0...9) for main kernel areas
G = Group code (0...9) inside an area
S = Subgroup code (0...9) inside a group
= Kernel ID (00, 01, …) inside the subgroup

Areas of Classification

1. Processor performance kernels
2. Memory access kernels (paging and caching)
3. Disk and peripherals access kernels
4. System kernels
5. User programs

Kernel Classification (1/9)

1 PROCESSOR PERFORMANCE KERNELS
  11 Nonnumerical procedural kernels
    110 Miscellaneous
    111 Control structures and function calls
    112 Arrays (including C-strings)
    113 Strings (the standard class string)
    114 Records/structs
    115 Dynamic lists, queues, and trees
    116 Search, sort, and merge
    117 Recursive nonnumerical problems
    118 Combinatorial problems

Kernel Classification (2/9)

1 PROCESSOR PERFORMANCE KERNELS
  12 Seminumerical procedural kernels
    120 Miscellaneous
    121 Integer arithmetic and counters
    122 Bitwise and integer operations/functions
    123 Graph algorithms
    124 Prime numbers
    125 Random numbers and Monte Carlo methods
    126 Cryptography
    127 Recursive seminumerical problems

Kernel Classification (3/9)

- 1 PROCESSOR PERFORMANCE KERNELS
  - 13 Numerical procedural kernels
    - 130 Miscellaneous
    - 131 Scalar floating-point arithmetic
    - 132 Library and special functions
    - 133 Arrays
    - 134 Polynomials
    - 135 Matrices
    - 136 Integrals and differential equations
    - 137 Recursive numerical problems
    - 138 Statistics

Kernel Classification (4/9)

1 PROCESSOR PERFORMANCE KERNELS
  - 14 Object oriented kernels
    - 140 Miscellaneous
    - 141 Object construction/destruction/manipulation
    - 142 Overloading operators
    - 143 Inheritance and multiple inheritance
    - 144 Polymorphism
    - 145 Abstract classes
    - 146 Templates
    - 147 Exception handling

Kernel Classification (5/9)

2 MEMORY ACCESS KERNELS (PAGING & CACHING)

- 21 Static memory access
  - 210 Miscellaneous
  - 211 Uniform distribution, multiple localities
  - 212 Normal distribution, multiple localities

- 22 Dynamic memory access
  - 220 Miscellaneous
  - 221 Uniform distribution, multiple localities
  - 222 Normal distribution, multiple localities

Kernel Classification (6/9)

3 DISK AND PERIPHERALS ACCESS KERNELS

- 31 Disk access
  - 310 Miscellaneous
  - 311 Sequential access
  - 312 Random access

- 32 Other peripheral kernels
  - 320 Miscellaneous
  - 321 VDU and graphics
  - 322 Archival tape access

Kernel Classification (7/9)

4 SYSTEM KERNELS
- 41 Processes
  - 410 Miscellaneous
  - 411 Process create and delete
  - 412 Multicore
- 42 Threads
  - 420 Miscellaneous
  - 421 Thread create and delete
  - 422 Hyperthreaded
- 43 Signals and alarms
  - 430 Miscellaneous
  - 431 Signals
  - 432 Alarms

Kernel Classification (8/9)

4 SYSTEM KERNELS
- 44 Pipes and other process communication mechanisms
  - 440 Miscellaneous
  - 441 Pipe communication
- 45 Networking and data communication
  - 450 Miscellaneous
  - 451 Socket communication
- 46 File management
  - 460 Miscellaneous
  - 461 Sequential access
  - 462 Random access
  - 463 Indexed access

Kernel Classification (9/9)

5 USER PROGRAMS

- 50 Miscellaneous
  - 500 Miscellaneous

Kernel Design Concepts (1/2)

- Kernels must be self-contained (designed as a block that can be inserted at any place in a benchmark program)
- To secure maximum mobility of kernel code, its dependence on environment should be kept at minimum (usage of only a few global variables).
- Kernels must be resistant to elimination by optimizing compilers.

Kernel Design Concepts (2/2)

- Input data must be internally generated.
- The number of lines of code in a kernel must be limited to secure sufficient granularity of benchmark workload.
- It is necessary to include a validation of results to verify both the correctness of algorithm, and the proper functioning of tested hardware and software.

Standard Kernel Structure

TIME = O(SEC)

```
{ // Definition of local data objects
  char* name = “<kernel code>: <kernel name>”;
  for(I=0; I<SEC; I++)                       // SEC = desired run time in sec
    for(J=0; J<RATE; J++)                    // 1 second calibration loop
    {
       // Local data initialization          // Synthetic data
       // Computation of results             // Any algorithm
       // Validation of results              // Computation of the
       if(results_incorrect)                 // results_incorrect flag
       { // Error message
          exit(1);                           // Abort benchmark execution
       }
    }
  terminator( name );                        // Kernel termination function
}                                            // (kernel/benchmark termination)
```

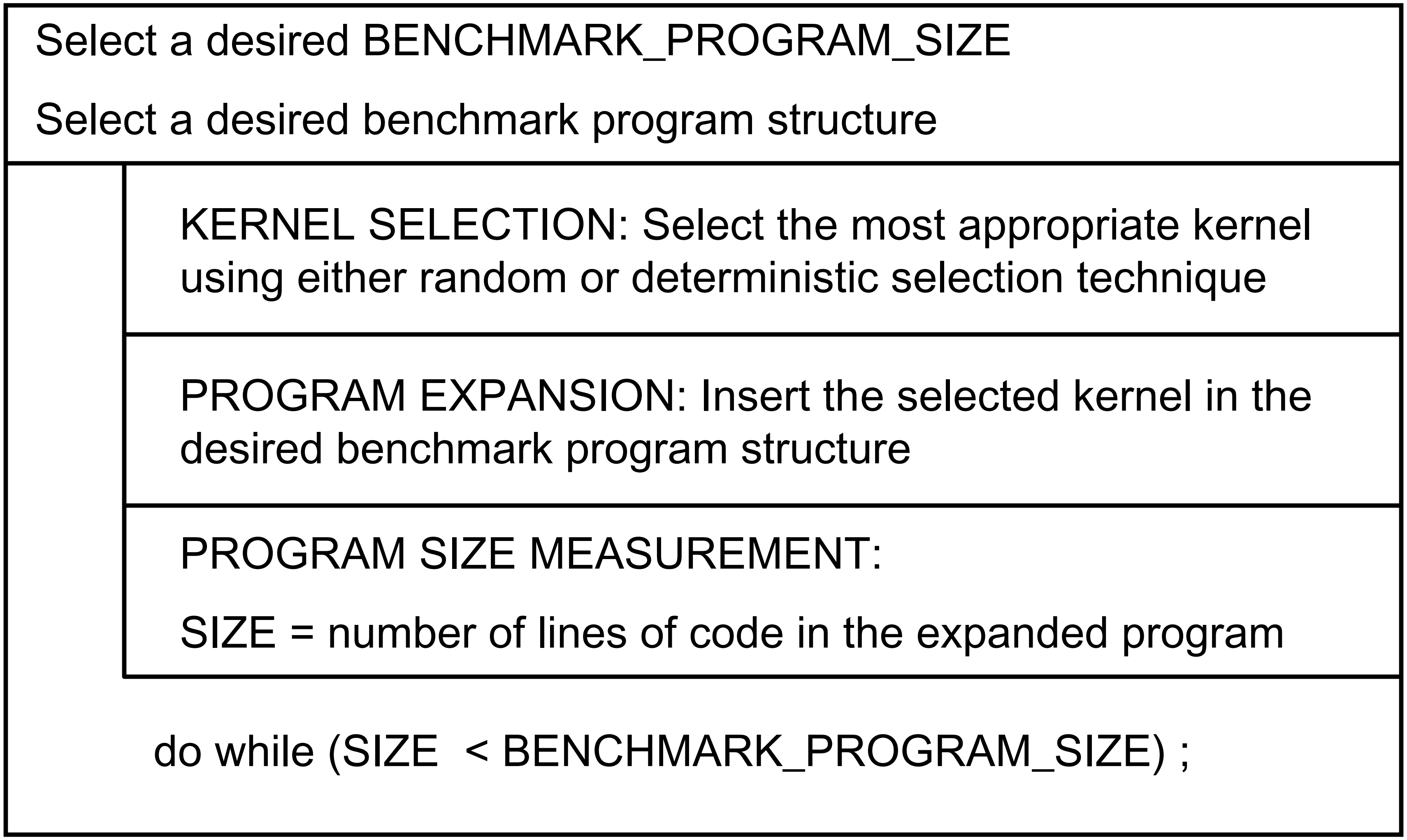
Benchmark Terminator Function

```
void terminator( char name[ ] )
{
  double RunTime= sec( ) - STARTTIME;  // Benchmark run time (from
  KERNEL_COUNT++;                      // start to this point)

  if(TRACE) cout << "Kernel Count = " << KERNEL_COUNT
                         << "  Seconds" << RunTime << "  " << name << endl;

  // End of program test

  if( (MAXKERNEL>0  &&  MAXKERNEL <= KERNEL_COUNT) ||
     (MAXSEC > 0.  &&  MAXSEC <= RunTime) )
  {
    cout << "\n\nNumber of executed kernels = " << KERNEL_COUNT
            <<   "\nRun time [total  seconds]  = " << RunTime
            << "\n\nEnd of measurement\n\n";
    exit(1);
  }
}
```

Global Parameters

- **SEC :** desired kernel run time in seconds
- **MAXSEC :** desired benchmark run time in seconds
- **KERNEL_COUNT :** a counter used by the benchmark program to control the number of executed kernels
- **MAXKERNEL :** desired number of executed kernels
- **RATE :** the number of kernel initialization-computation- validation cycles per second (adjusted during the kernel calibration process)
- **TRACE :** benchmark program trace flag

Benchmark Generation Process



Select a desired BENCHMARK_PROGRAM_SIZE

Select a desired benchmark program structure

KERNEL SELECTION: Select the most appropriate kernel using either random or deterministic selection technique

PROGRAM EXPANSION: Insert the selected kernel in the desired benchmark program structure

PROGRAM SIZE MEASUREMENT:

SIZE = number of lines of code in the expanded program

do while (SIZE < BENCHMARK_PROGRAM_SIZE) ;

Kernel Calibration

- Adjust the kernel SIZE parameter to get a desired use of memory
- Adjust the internal SEC parameter to get a desired run time T = O(SEC)
- Calibration is performed using an independent calibration program tool
- Kernels are stored in kernel library

Calibration parameters

- r = the repetition count
- t = run time that corresponds to r
- T = desired (calibrated) run time
- R = the repetition count value that corresponds to the desired value of T (denoted in programs as RATE, the number of repetitions per second)
- Linear model: t = ar + b, a=const., b=const. (b is usually negligible)

Calibration process

$$t = ar + b, \quad a = const, \quad b = const.$$

$$t_1 = ar_1 + b, \qquad t_2 = ar_2 + b, \qquad T = aR + b$$

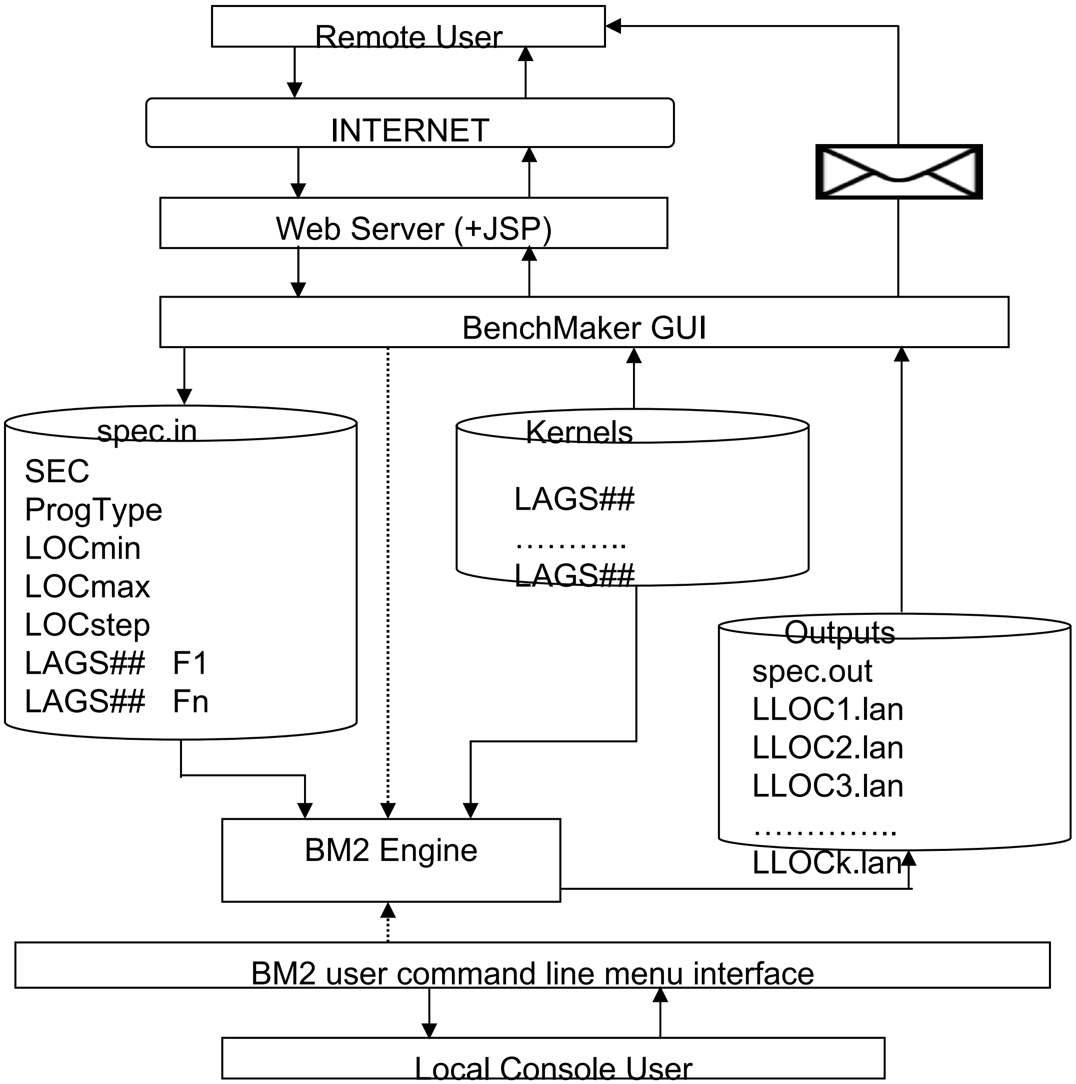
$$t_2 - t_1 = a(r_2 - r_1), \qquad T - t_1 = a(R - r_1),$$

$$a = \frac{t_2 - t_1}{r_2 - r_1} = \frac{T - t_1}{R - r_1}$$

$$R = r_1 + (T - t_1)(r_2 - r_1)/(t_2 - t_1)$$

R should be greater than 100 to provide accurate approximation of T

BM2 System Overview



Remote User

INTERNET

Web Server (+JSP)

BenchMaker GUI

spec.in

SEC
ProgType
LOCmin
LOCmax
LOCstep
LAGS## F1
LAGS## Fn

Kernels

LAGS##
………..
LAGS##

Outputs

spec.out
LLOC1.lan
LLOC2.lan
LLOC3.lan
…………..
LLOCk.lan

BM2 Engine

BM2 user command line menu interface

Local Console User

Workload Characterization

- Representative set of kernels (those that are most similar to user’s expected or existing activities)
- Individual kernel weights (relative frequencies of use of the type of processing implemented by a kernel)
- The length of generated kernel-based benchmark (expressed in logical lines of code, LOC, which are generally defined as high-level language statements)
- Individual kernel run times (SEC, seconds per kernel), that affect the total run time of the generated benchmark.

Benchmark Generation Methods

- Kernel sequence (SEQ) model
- Kernel function (KF) model
- Minimum size canonic (MC) loop-select model
- Adjustable size canonic (AC) loop-select model
- Kernel-terminated recursive expansion (REX) model

SEQ: Kernel Sequence Model

```
void main(void)
{
        { K33 }

        { K17 }

        { K44 }

        { K19 }

        { K33 }

        { K41 }

        { K44 }
        ............
        { K93 }
}
```

Kernels are randomly or deterministically selected according to a desired kernel distribution function

```
while(LOC(main) < desired_SIZE)
{
    Select kernel;
    Append kernel;
}
```

SEQF: Kernel Function Model

```
int ERROR;                    //  Global kernel error code
int F1(void)
{
        { K19 }               //  Randomly selected kernel
        return ERROR ;        //  Kernel error code
}
..............................
int Fn(void)
{
        { K41 }               //  Randomly selected kernel
        return ERROR ;        //  Kernel error code
}
void main(void)
{       long int sum = 0 ;
        sum += F1( ) ;
        .....................
        sum += Fn( ) ;
        cout << sum;
}
```

MC: Minimum Size Canonic Loop-Select Model

```
for(i=0; i<TIME; i++)
          switch( selector( ) )
          {
                    case 00:      { K00 } ; break;
                    case 01:      { K01 } ; break;
                    case 02:      { K02 } ; break;
                    ........................................
                    case 99:      { K99 } ; break;
          }
```

TIME = execution time parameter.

selector( ) = kernel distribution function.

Each kernel appears only once.

AC: Adjustable Size Canonic Loop-Select Model

```
for(i=0; i<TIME; i++)
        switch( uniform( ) )     // 0 ≤ uniform( ) ≤ SIZE
        {       case 0000:  { K19 } ; break;
                case 0001:  { K02 } ; break;
                case 0002:  { K02 } ; break;
                case 0003:  { K02 } ; break;
                case 0004:  { K19 } ; break;
                ........................................
                case SIZE:  { K41 } ; break;
        }
```

TIME = execution time parameter. Kernels may repeat. Their frequency is specified by the desired SIZE and the kernel distribution function.

REX: Kernel-terminated recursive expansion model

```
// G[ ] = global counter array. Initially long G[n]=0,  n=1,…,N
if (++G[13]%2)       //   1, 0, 1, 0, 1, …
{
          while (++G[14]%5)   //   1, 2, 3, 4, 0, 1, 2, 3, 4, 0, …
          {
                    { K19 }             //   Kernel termination
                    if (++G[15]%2)    //   1, 0, 1, 0, 1, …
                    {
                              { K17 }    //   Kernel termination
                    }
          }
}
else
{
          for( ;  ++G[16]%5  ; )        //   1, 2, 3, 4, 0, 1, 2, 3, 4, 0, …
                    if (++G[17]%2)    //   1, 0, 1, 0, 1, …
                              { K64 }    //   Kernel termination
                    else
                              { K17 }    //   Kernel termination
}
```

Workload Characterization by Kernel Distribution

$$K_1, K_2, \ldots, K_n = \text{kernels}$$

$$P_1, P_2, \ldots, P_n = \text{desired kernel probabilities}$$

Kernel selection techniques:

- Minimization of error criterion (math approach)
- Random selection according to given distribution
- Deterministic Optimum Selection (DOS)

Kernel Selection Problem [1/11]

n = total number of available kernels

$K_1, K_2, ..., K_n$ = kernels

$L_1, L_2, ..., L_n$ = kernel sizes [LOC]

$f_1, f_2, ..., f_n$ = kernel frequencies in a given program

$f_1 + f_2 + ... + f_n = F$ = total number of kernels

$f_1 L_1 + f_2 L_2 + ... + f_n L_n$ = total benchmark size

L = desired size of benchmark program [LOC]

$P_1, P_2, ..., P_n$ = desired kernel probabilities

$p_i = f_i / F, \quad i = 1, ..., n$: achieved kernel probabilities

Kernel Selection Problem [2/11]

INPUTS:

$P_1, P_2, ..., P_n =$ desired kernel probabilities

$L =$ desired benchmark size

PROBLEM:

Find optimum kernel frequencies $f_1^*, f_2^*, ..., f_n^*$

so that the resulting benchmark has a desired

size and desired kernel probabilities.

Kernel Selection Problem [3/11]

Statement of the kernel selection problem:

Minimize the kernel distribution error

$$E(f_1, f_2, ..., f_n) = \sum_{i=1}^{n} \left| \frac{f_i}{f_1 + f_2 + ... + f_n} - P_i \right|$$

with the following condition:

$$f_1 L_1 + f_2 L_2 + ... + f_n L_n \cong L$$

Kernel Selection Problem [4/11]

In other words, find $f_1^*, f_2^*, \ldots, f_n^*$ so that

$$E(f_1^*, f_2^*, \ldots, f_n^*) = \min_{f_1, f_2, \ldots, f_n} \sum_{i=1}^{n} \left| \frac{f_i}{f_1 + f_2 + \ldots + f_n} - P_i \right|$$

and

$$f_1^* L_1 + f_2^* L_2 + \ldots + f_n^* L_n \cong L$$

Kernel Selection Problem [5/11]

Approach #1. Minimize a global error criterion function that combines two goals: a desired program size, and a desired kernel distribution.

$$C(f_1, f_2, \ldots, f_n) =$$

$$\left[W\left(\left| f_1 L_1 + \ldots + f_n L_n - L \right| \right)^r + (1-W)\left(\sum_{i=1}^{n} \left| \frac{f_i}{f_1 + f_2 + \ldots + f_n} - P_i \right| \right)^r \right]^{1/r}$$

$0 < W < 1, \quad 1 \le r \le +\infty$ (to simultaneously satisfy both goals)

This function can be minimized using Nelder-Mead algorithm.

Kernel Selection Problem [6/11]

Advantage of the mathematical approach:

- It is possible to generate the exact optimum solution

Disadvantages:

- The solution depends on parameters *W* and *r*. It may be necessary to readjust parameters for different numbers and distributions of kernels.
- Minimization can find a local minimum different from the optimum solution.
- Minimization can be time consuming.

Kernel Selection Problem [7/11]

Approach #2: Random selection according to desired kernel probability distribution.

```
do{
        r = (random integer from 1 to n distributed according to any desired kernel distribution) ;
        Insert kernel $K_r$ in benchmark program;
        size = (number of lines of code after the addition of kernel $K_r$ );
} while (size < L);
```

Kernel Selection Problem [8/11]

Advantages of random selection:

- Simplicity
- Speed (constant kernel selection time)
- Appropriate for very large programs

Disadvantage:

- Large and random distribution errors for small and medium numbers of kernels

Kernel Selection Problem [9/11]

Approach #3: Deterministic Optimum Selection (DOS) according to desired kernel distribution.

```
do{
        r = (integer from 1 to n  selected by DOS according
             to desired kernel distribution) ;
        Insert kernel $K_r$ in benchmark program;
        size =  (number of lines of code after the addition of
                 kernel $K_r$ );
} while (size < L);
```

Kernel Selection Problem [10/11]

DOS Algorithm: In each iteration add kernel that minimizes the kernel distribution error

$$e(j) = \left| \frac{f_j + 1}{f_1 + f_2 + \ldots + f_n + 1} - P_j \right| + \sum_{\substack{i=1 \\ i \neq j}}^{n} \left| \frac{f_i}{f_1 + f_2 + \ldots + f_n + 1} - P_i \right|, \quad 1 \le j \le n$$

Select kernel K_r where $e(r) = \min_{1 \le j \le n} e(j)$

Kernel Selection Problem [11/11]

Advantages of DOS approach:

- Simplicity
- Close to optimum in each insertion step
- Accurate for any program size

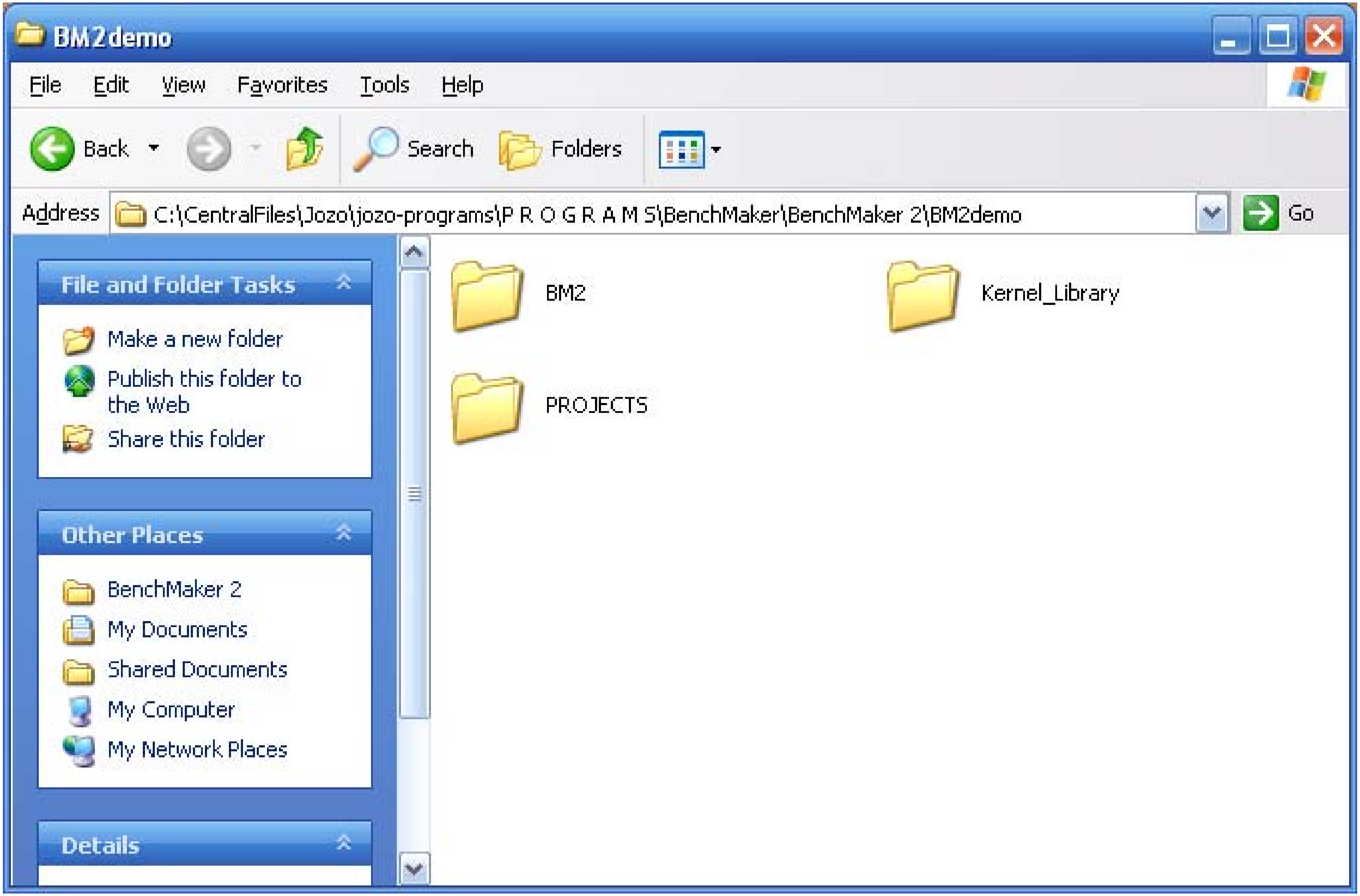
Disadvantage:

- Each kernel selection needs time O(n)

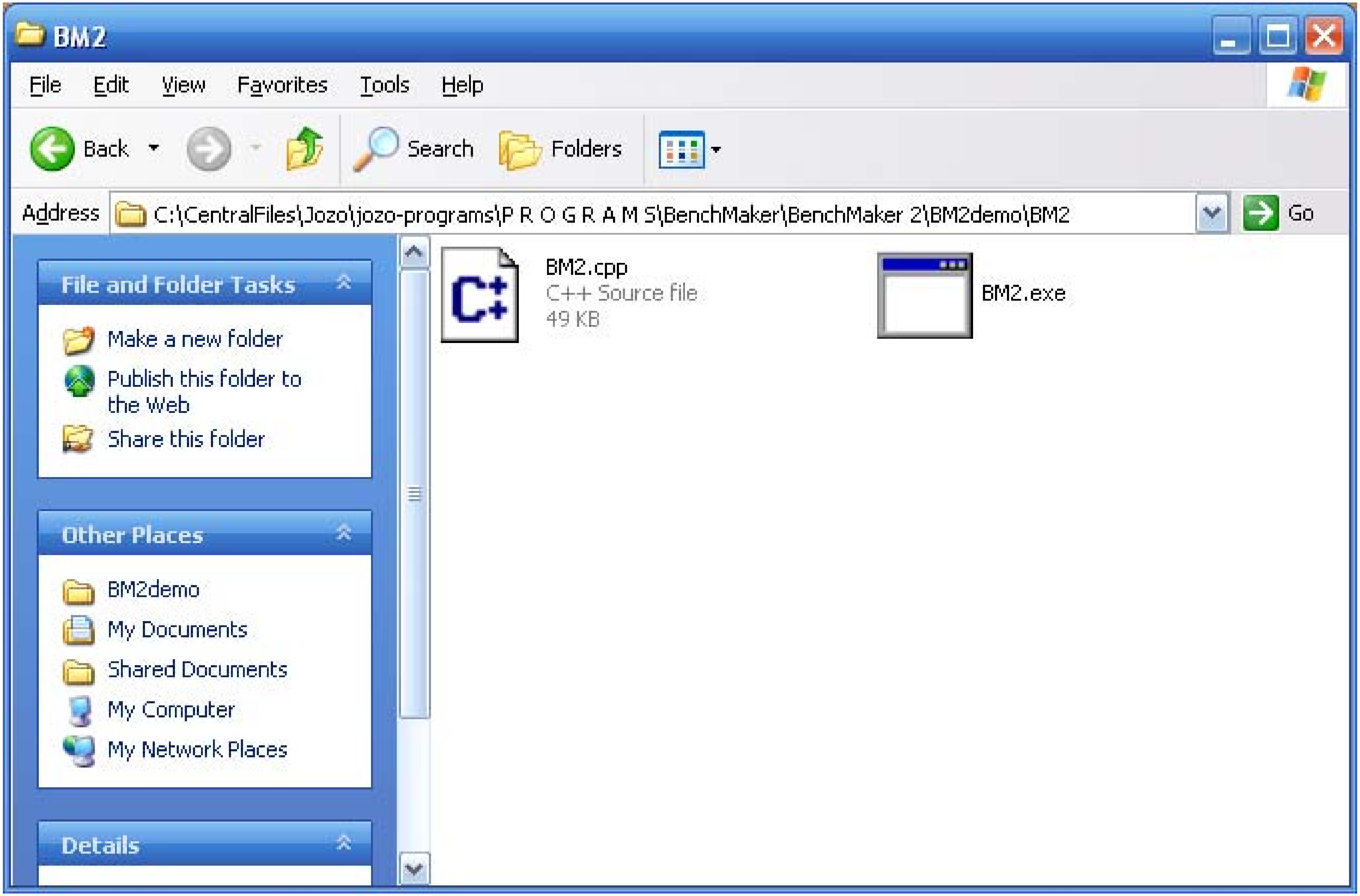
BenchMaker2 Engine

Algorithm

1. Select the structure of the generated program
2. Select the desired size of program (LLOC or K)
3. Select the desired distribution of kernels
4. Select the optimum kernel according to the deterministic selection algorithm (DSA)
5. Insert the selected kernel in the generated program
6. If the desired size is not achieved go to (4). Otherwise, stop.



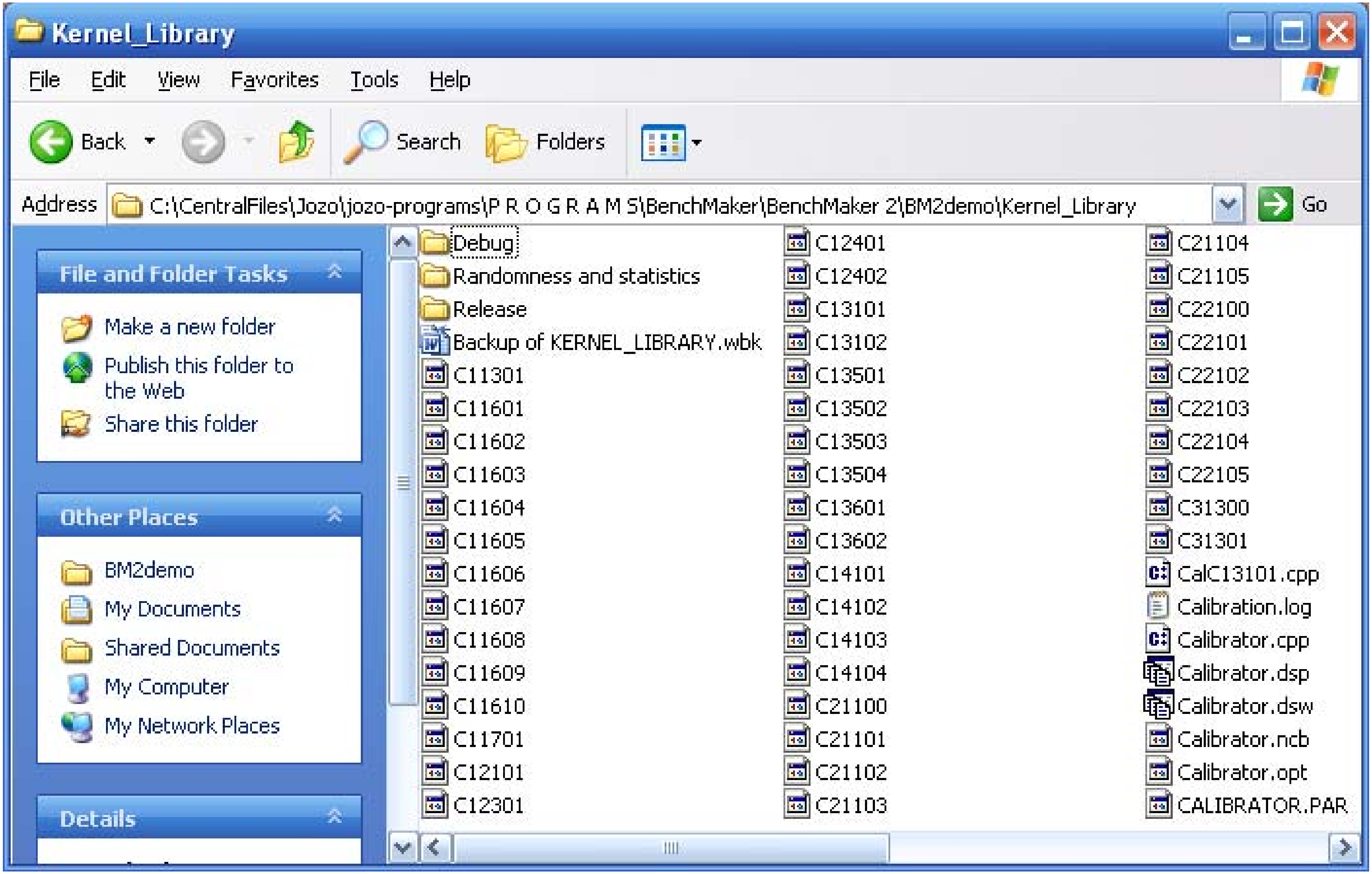
Address C:\CentralFiles\Jozo\jozo-programs\P R O G R A M S\BenchMaker\BenchMaker 2\BM2demo



Address: C:\CentralFiles\Jozo\jozo-programs\P R O G R A M S\BenchMaker\BenchMaker 2\BM2demo\BM2

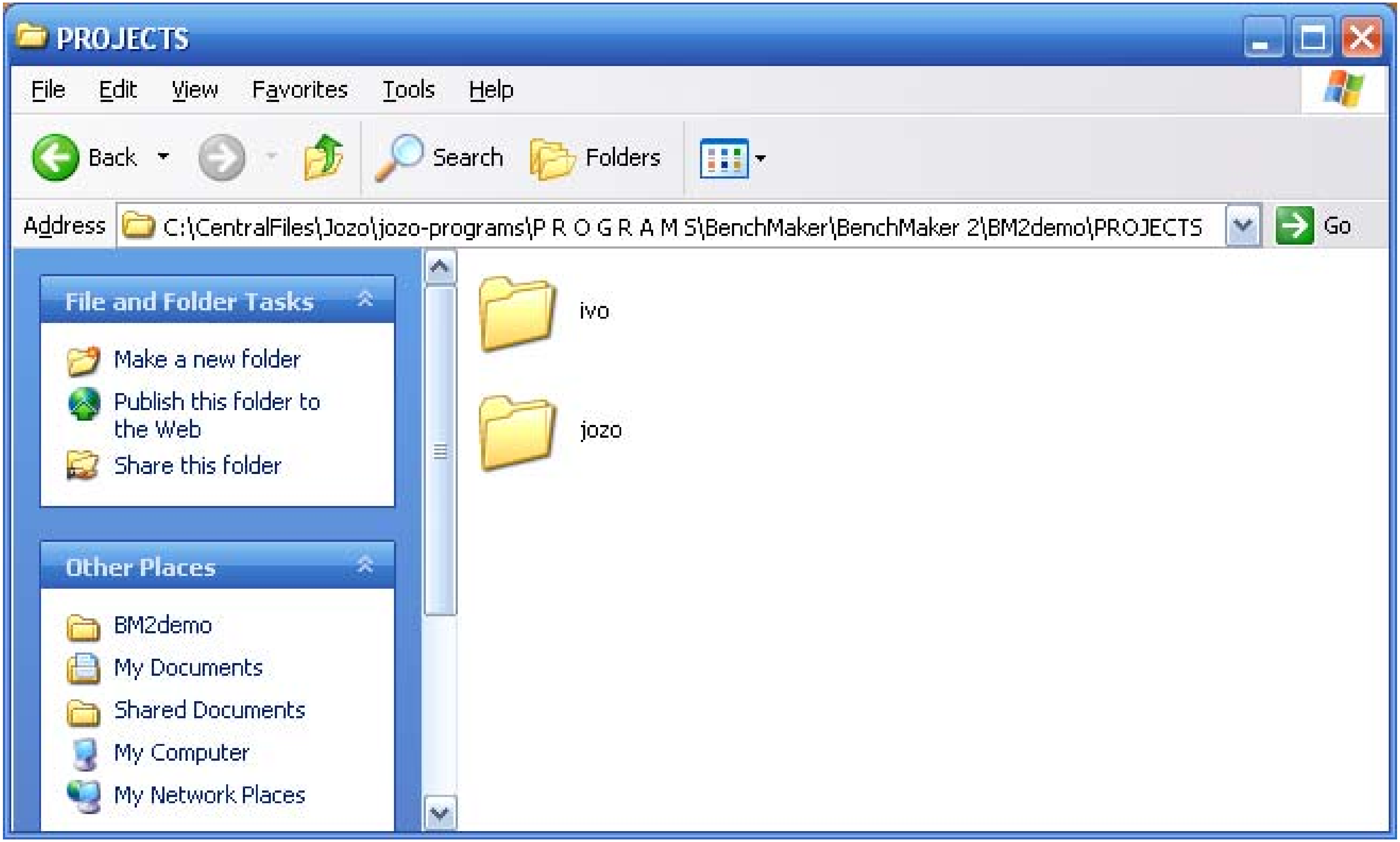
BM2.cpp
C++ Source file
49 KB

BM2.exe

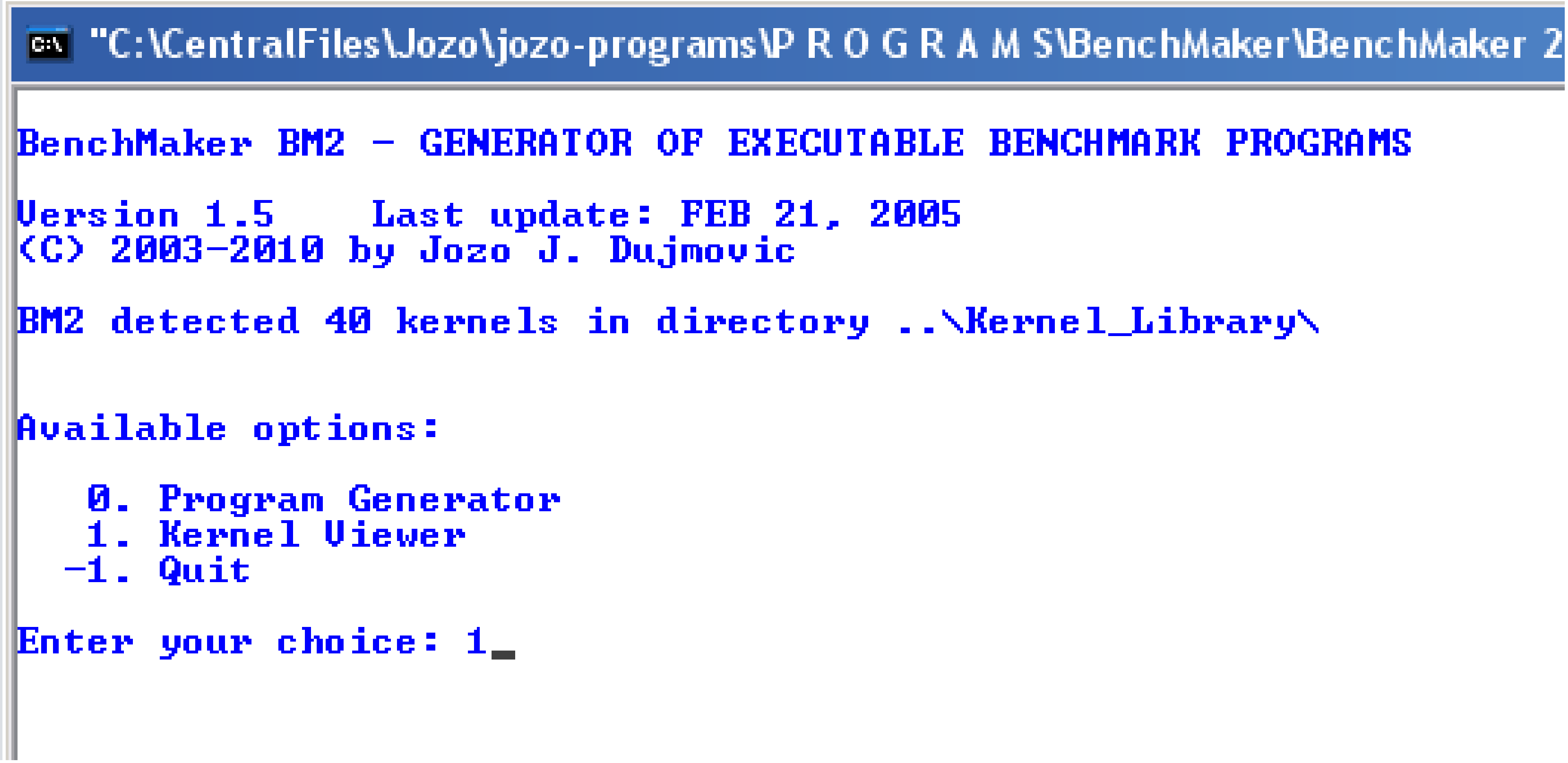


Address: C:\CentralFiles\Jozo\jozo-programs\P R O G R A M S\BenchMaker\BenchMaker 2\BM2demo\Kernel_Library

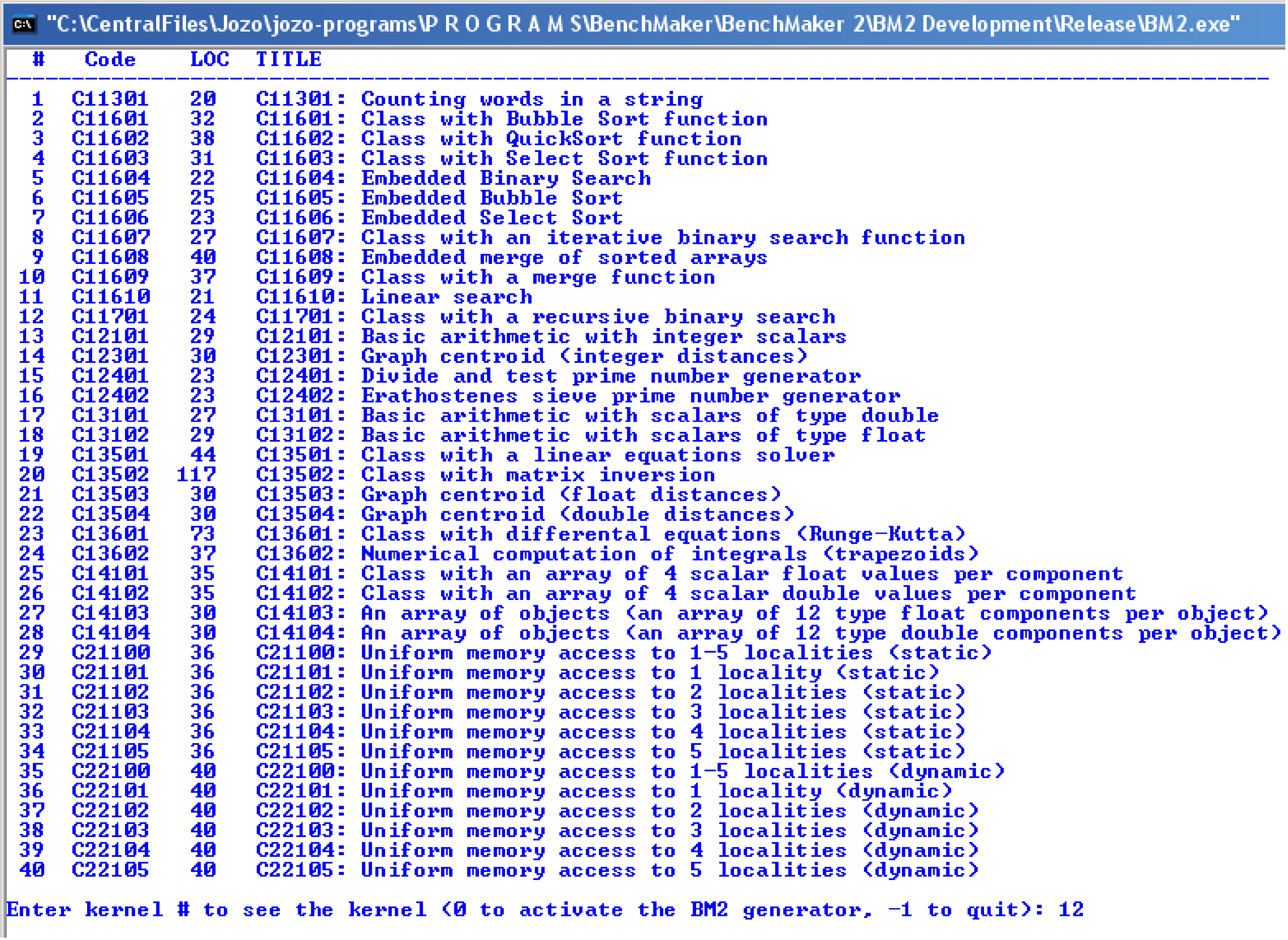
```
Debug
Randomness and statistics
Release
Backup of KERNEL_LIBRARY.wbk
C11301
C11601
C11602
C11603
C11604
C11605
C11606
C11607
C11608
C11609
C11610
C11701
C12101
C12301
C12401
C12402
C13101
C13102
C13501
C13502
C13503
C13504
C13601
C13602
C14101
C14102
C14103
C14104
C21100
C21101
C21102
C21103
C21104
C21105
C22100
C22101
C22102
C22103
C22104
C22105
C31300
C31301
CalC13101.cpp
Calibration.log
Calibrator.cpp
Calibrator.dsp
Calibrator.dsw
Calibrator.ncb
Calibrator.opt
CALIBRATOR.PAR
```



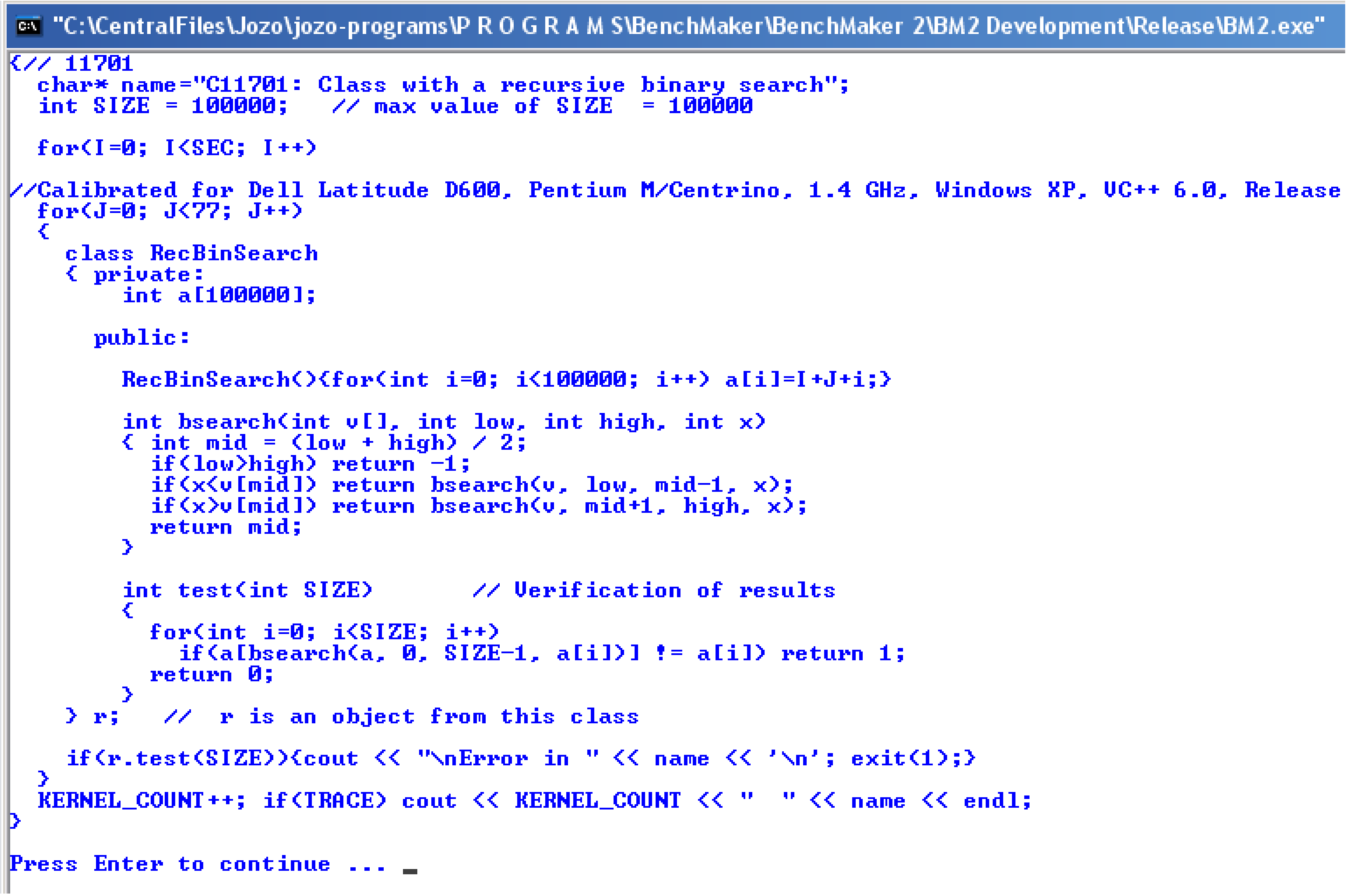
Address: C:\CentralFiles\Jozo\jozo-programs\P R O G R A M S\BenchMaker\BenchMaker 2\BM2demo\PROJECTS



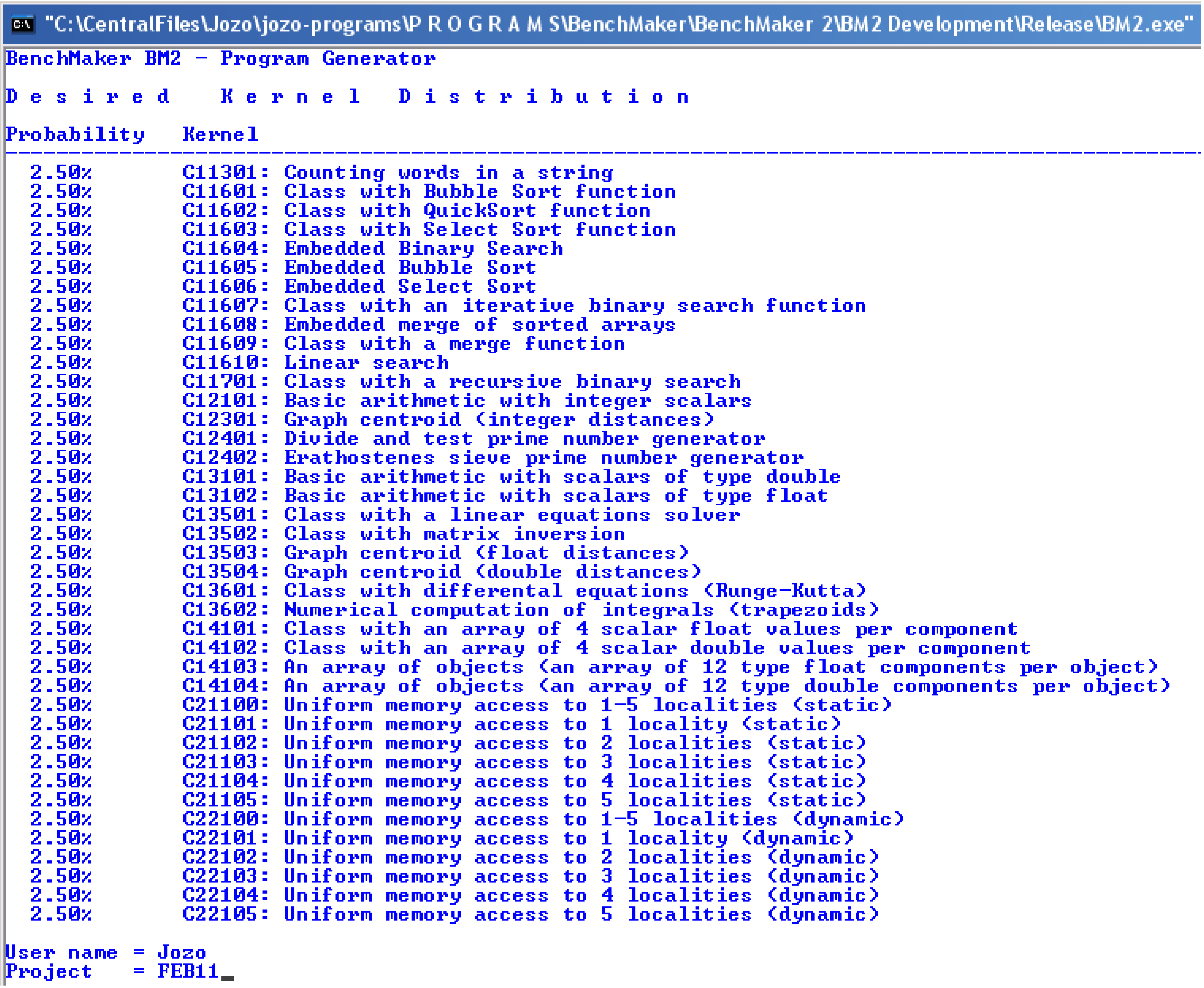
```
BenchMaker BM2 - GENERATOR OF EXECUTABLE BENCHMARK PROGRAMS

Version 1.5    Last update: FEB 21, 2005
(C) 2003-2010 by Jozo J. Dujmovic

BM2 detected 40 kernels in directory ..\Kernel_Library\


Available options:

   0. Program Generator
   1. Kernel Viewer
  -1. Quit

Enter your choice: 1_
```



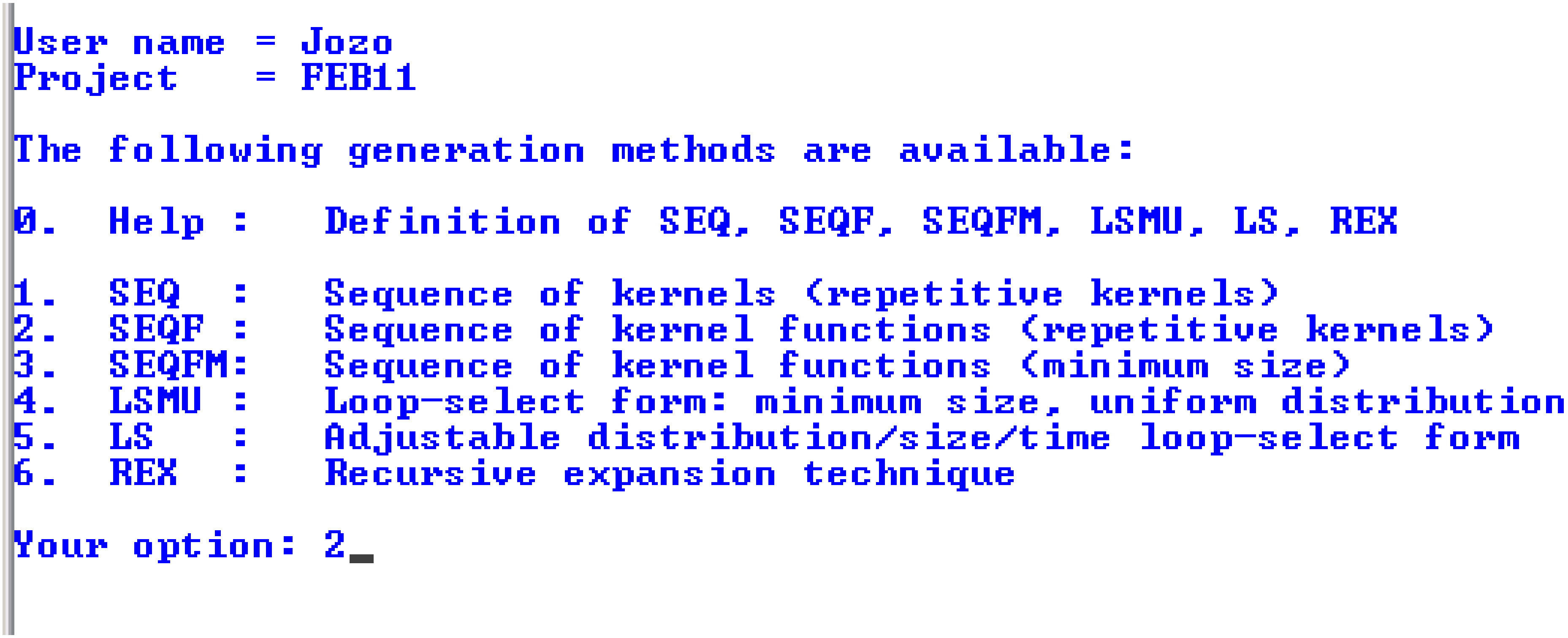
```
  #    Code     LOC  TITLE
--------------------------------------------------------------------------------------------------
  1  C11301   20   C11301: Counting words in a string
  2  C11601   32   C11601: Class with Bubble Sort function
  3  C11602   38   C11602: Class with QuickSort function
  4  C11603   31   C11603: Class with Select Sort function
  5  C11604   22   C11604: Embedded Binary Search
  6  C11605   25   C11605: Embedded Bubble Sort
  7  C11606   23   C11606: Embedded Select Sort
  8  C11607   27   C11607: Class with an iterative binary search function
  9  C11608   40   C11608: Embedded merge of sorted arrays
 10  C11609   37   C11609: Class with a merge function
 11  C11610   21   C11610: Linear search
 12  C11701   24   C11701: Class with a recursive binary search
 13  C12101   29   C12101: Basic arithmetic with integer scalars
 14  C12301   30   C12301: Graph centroid (integer distances)
 15  C12401   23   C12401: Divide and test prime number generator
 16  C12402   23   C12402: Erathostenes sieve prime number generator
 17  C13101   27   C13101: Basic arithmetic with scalars of type double
 18  C13102   29   C13102: Basic arithmetic with scalars of type float
 19  C13501   44   C13501: Class with a linear equations solver
 20  C13502  117   C13502: Class with matrix inversion
 21  C13503   30   C13503: Graph centroid (float distances)
 22  C13504   30   C13504: Graph centroid (double distances)
 23  C13601   73   C13601: Class with differental equations (Runge-Kutta)
 24  C13602   37   C13602: Numerical computation of integrals (trapezoids)
 25  C14101   35   C14101: Class with an array of 4 scalar float values per component
 26  C14102   35   C14102: Class with an array of 4 scalar double values per component
 27  C14103   30   C14103: An array of objects (an array of 12 type float components per object)
 28  C14104   30   C14104: An array of objects (an array of 12 type double components per object)
 29  C21100   36   C21100: Uniform memory access to 1-5 localities (static)
 30  C21101   36   C21101: Uniform memory access to 1 locality (static)
 31  C21102   36   C21102: Uniform memory access to 2 localities (static)
 32  C21103   36   C21103: Uniform memory access to 3 localities (static)
 33  C21104   36   C21104: Uniform memory access to 4 localities (static)
 34  C21105   36   C21105: Uniform memory access to 5 localities (static)
 35  C22100   40   C22100: Uniform memory access to 1-5 localities (dynamic)
 36  C22101   40   C22101: Uniform memory access to 1 locality (dynamic)
 37  C22102   40   C22102: Uniform memory access to 2 localities (dynamic)
 38  C22103   40   C22103: Uniform memory access to 3 localities (dynamic)
 39  C22104   40   C22104: Uniform memory access to 4 localities (dynamic)
 40  C22105   40   C22105: Uniform memory access to 5 localities (dynamic)

Enter kernel # to see the kernel (0 to activate the BM2 generator, -1 to quit): 12
```



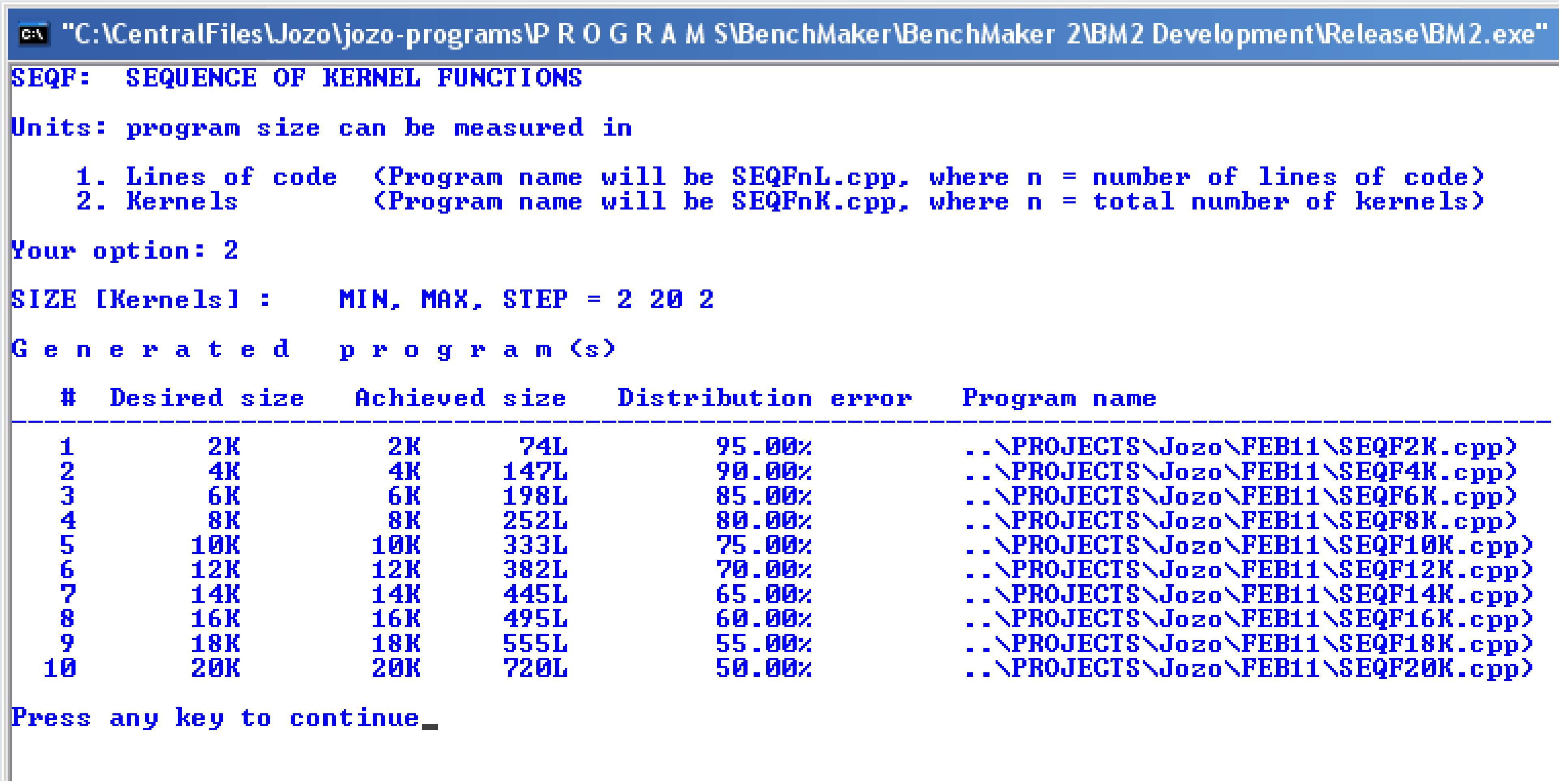
```
{// 11701
  char* name="C11701: Class with a recursive binary search";
  int SIZE = 100000;   // max value of SIZE  = 100000

  for(I=0; I<SEC; I++)

//Calibrated for Dell Latitude D600, Pentium M/Centrino, 1.4 GHz, Windows XP, VC++ 6.0, Release
  for(J=0; J<77; J++)
  {
    class RecBinSearch
    { private:
        int a[100000];

      public:

        RecBinSearch(){for(int i=0; i<100000; i++) a[i]=I+J+i;}

        int bsearch(int v[], int low, int high, int x)
        { int mid = (low + high) / 2;
          if(low>high) return -1;
          if(x<v[mid]) return bsearch(v, low, mid-1, x);
          if(x>v[mid]) return bsearch(v, mid+1, high, x);
          return mid;
        }

        int test(int SIZE)       // Verification of results
        {
          for(int i=0; i<SIZE; i++)
            if(a[bsearch(a, 0, SIZE-1, a[i])] != a[i]) return 1;
          return 0;
        }
    } r;   //  r is an object from this class

    if(r.test(SIZE)){cout << "\nError in " << name << '\n'; exit(1);}
  }
  KERNEL_COUNT++; if(TRACE) cout << KERNEL_COUNT << "  " << name << endl;
}

Press Enter to continue ... _
```



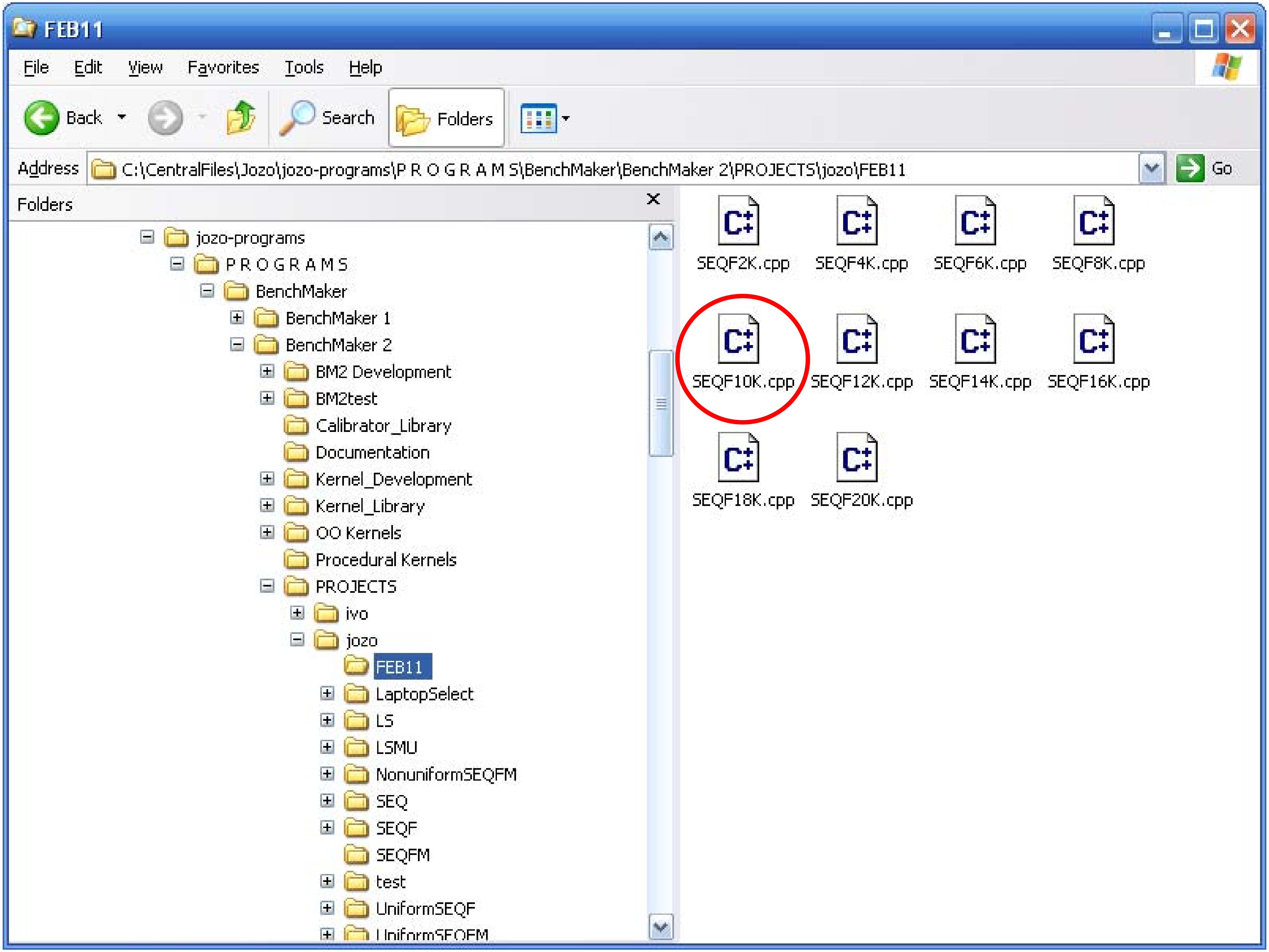
```
BenchMaker BM2 - Program Generator

D e s i r e d   K e r n e l   D i s t r i b u t i o n

Probability    Kernel
-----------------------------------------------------------------------------------------------
  2.50%        C11301: Counting words in a string
  2.50%        C11601: Class with Bubble Sort function
  2.50%        C11602: Class with QuickSort function
  2.50%        C11603: Class with Select Sort function
  2.50%        C11604: Embedded Binary Search
  2.50%        C11605: Embedded Bubble Sort
  2.50%        C11606: Embedded Select Sort
  2.50%        C11607: Class with an iterative binary search function
  2.50%        C11608: Embedded merge of sorted arrays
  2.50%        C11609: Class with a merge function
  2.50%        C11610: Linear search
  2.50%        C11701: Class with a recursive binary search
  2.50%        C12101: Basic arithmetic with integer scalars
  2.50%        C12301: Graph centroid (integer distances)
  2.50%        C12401: Divide and test prime number generator
  2.50%        C12402: Erathostenes sieve prime number generator
  2.50%        C13101: Basic arithmetic with scalars of type double
  2.50%        C13102: Basic arithmetic with scalars of type float
  2.50%        C13501: Class with a linear equations solver
  2.50%        C13502: Class with matrix inversion
  2.50%        C13503: Graph centroid (float distances)
  2.50%        C13504: Graph centroid (double distances)
  2.50%        C13601: Class with differental equations (Runge-Kutta)
  2.50%        C13602: Numerical computation of integrals (trapezoids)
  2.50%        C14101: Class with an array of 4 scalar float values per component
  2.50%        C14102: Class with an array of 4 scalar double values per component
  2.50%        C14103: An array of objects (an array of 12 type float components per object)
  2.50%        C14104: An array of objects (an array of 12 type double components per object)
  2.50%        C21100: Uniform memory access to 1-5 localities (static)
  2.50%        C21101: Uniform memory access to 1 locality (static)
  2.50%        C21102: Uniform memory access to 2 localities (static)
  2.50%        C21103: Uniform memory access to 3 localities (static)
  2.50%        C21104: Uniform memory access to 4 localities (static)
  2.50%        C21105: Uniform memory access to 5 localities (static)
  2.50%        C22100: Uniform memory access to 1-5 localities (dynamic)
  2.50%        C22101: Uniform memory access to 1 locality (dynamic)
  2.50%        C22102: Uniform memory access to 2 localities (dynamic)
  2.50%        C22103: Uniform memory access to 3 localities (dynamic)
  2.50%        C22104: Uniform memory access to 4 localities (dynamic)
  2.50%        C22105: Uniform memory access to 5 localities (dynamic)

User name = Jozo
Project   = FEB11_
```



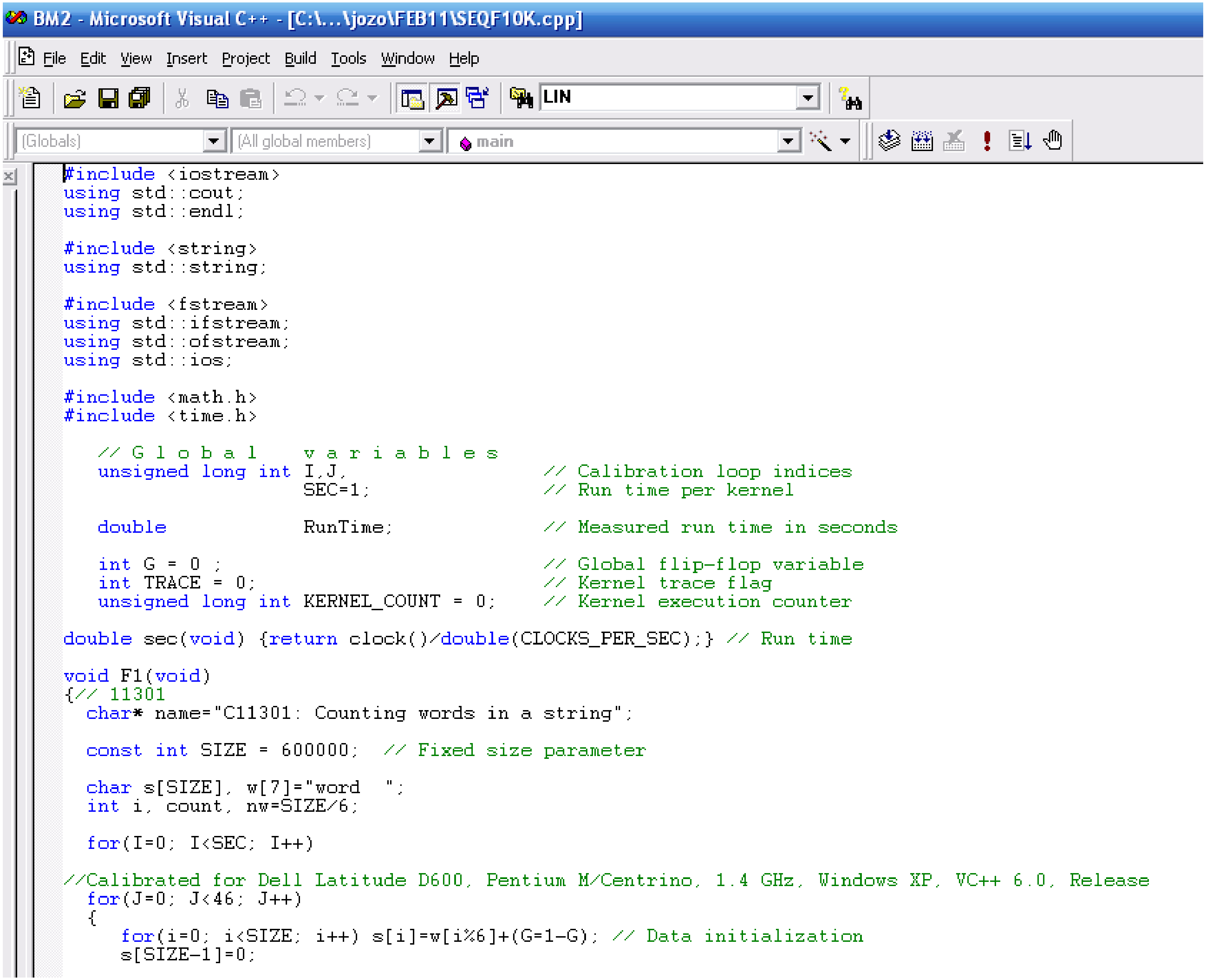
```
User name = Jozo
Project   = FEB11

The following generation methods are available:

0.  Help :   Definition of SEQ, SEQF, SEQFM, LSMU, LS, REX

1.  SEQ  :   Sequence of kernels (repetitive kernels)
2.  SEQF :   Sequence of kernel functions (repetitive kernels)
3.  SEQFM:   Sequence of kernel functions (minimum size)
4.  LSMU :   Loop-select form: minimum size, uniform distribution
5.  LS   :   Adjustable distribution/size/time loop-select form
6.  REX  :   Recursive expansion technique

Your option: 2_
```



```
SEQF:  SEQUENCE OF KERNEL FUNCTIONS

Units: program size can be measured in

    1. Lines of code  (Program name will be SEQFnL.cpp, where n = number of lines of code)
    2. Kernels        (Program name will be SEQFnK.cpp, where n = total number of kernels)

Your option: 2

SIZE [Kernels] :    MIN, MAX, STEP = 2 20 2

G e n e r a t e d   p r o g r a m (s)

   #  Desired size   Achieved size   Distribution error   Program name
----------------------------------------------------------------------------------------------
   1         2K          2K      74L           95.00%        ..\PROJECTS\Jozo\FEB11\SEQF2K.cpp)
   2         4K          4K     147L           90.00%        ..\PROJECTS\Jozo\FEB11\SEQF4K.cpp)
   3         6K          6K     198L           85.00%        ..\PROJECTS\Jozo\FEB11\SEQF6K.cpp)
   4         8K          8K     252L           80.00%        ..\PROJECTS\Jozo\FEB11\SEQF8K.cpp)
   5        10K         10K     333L           75.00%        ..\PROJECTS\Jozo\FEB11\SEQF10K.cpp)
   6        12K         12K     382L           70.00%        ..\PROJECTS\Jozo\FEB11\SEQF12K.cpp)
   7        14K         14K     445L           65.00%        ..\PROJECTS\Jozo\FEB11\SEQF14K.cpp)
   8        16K         16K     495L           60.00%        ..\PROJECTS\Jozo\FEB11\SEQF16K.cpp)
   9        18K         18K     555L           55.00%        ..\PROJECTS\Jozo\FEB11\SEQF18K.cpp)
  10        20K         20K     720L           50.00%        ..\PROJECTS\Jozo\FEB11\SEQF20K.cpp)

Press any key to continue_
```

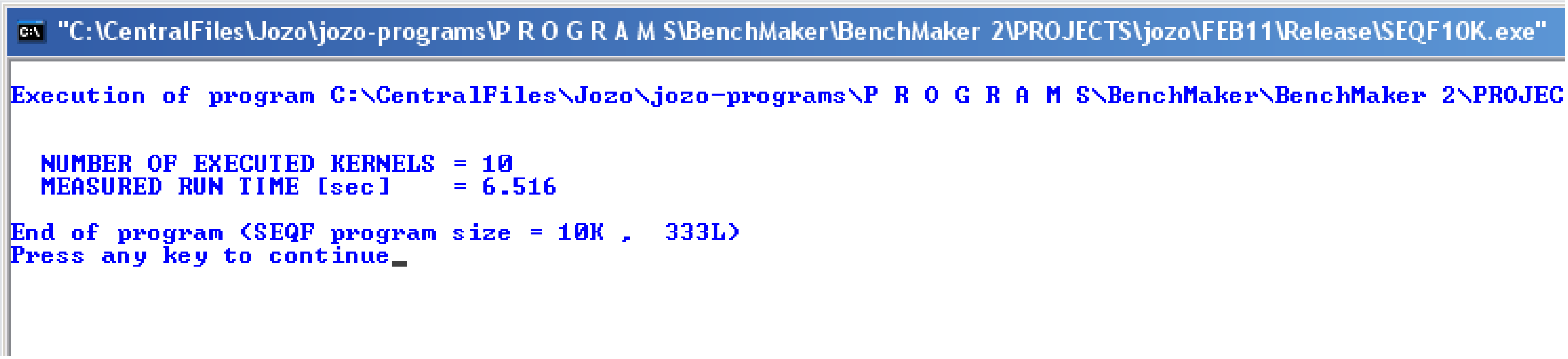


Address: C:\CentralFiles\Jozo\jozo-programs\P R O G R A M S\BenchMaker\BenchMaker 2\PROJECTS\jozo\FEB11



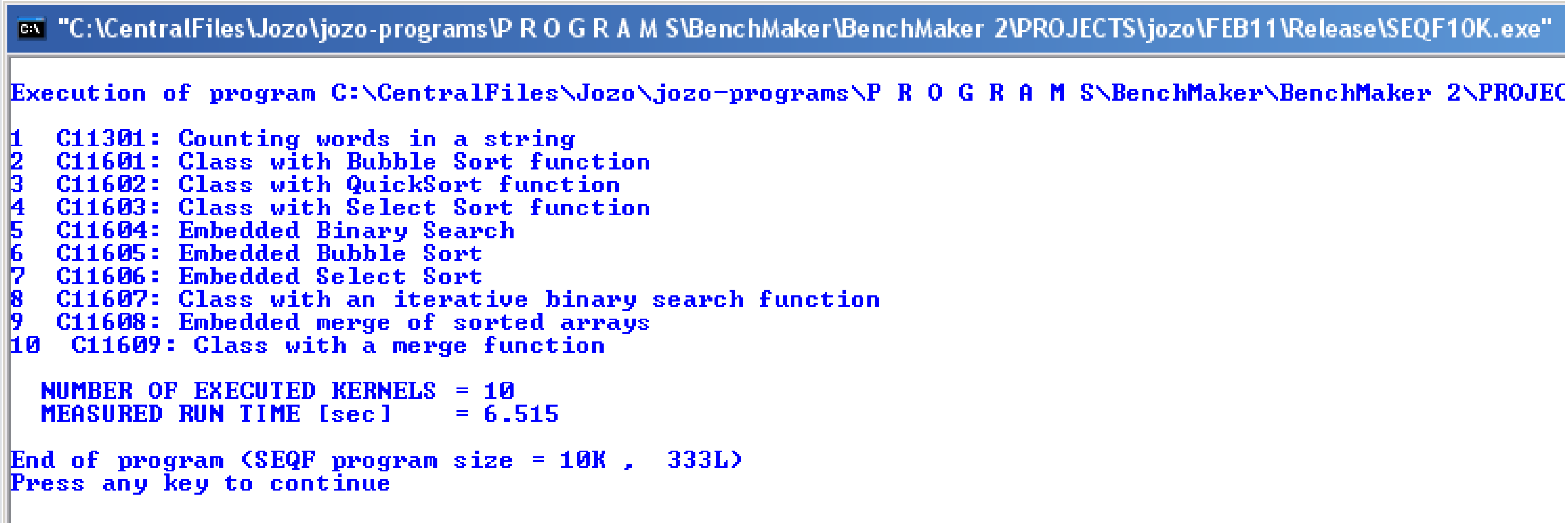
```
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

#include <fstream>
using std::ifstream;
using std::ofstream;
using std::ios;

#include <math.h>
#include <time.h>

   // G l o b a l    v a r i a b l e s
   unsigned long int I,J,                 // Calibration loop indices
                     SEC=1;               // Run time per kernel

   double            RunTime;             // Measured run time in seconds

   int G = 0 ;                            // Global flip-flop variable
   int TRACE = 0;                         // Kernel trace flag
   unsigned long int KERNEL_COUNT = 0;    // Kernel execution counter

double sec(void) {return clock()/double(CLOCKS_PER_SEC);} // Run time

void F1(void)
{// 11301
  char* name="C11301: Counting words in a string";

  const int SIZE = 600000;  // Fixed size parameter

  char s[SIZE], w[7]="word  ";
  int i, count, nw=SIZE/6;

  for(I=0; I<SEC; I++)

//Calibrated for Dell Latitude D600, Pentium M/Centrino, 1.4 GHz, Windows XP, VC++ 6.0, Release
  for(J=0; J<46; J++)
  {
     for(i=0; i<SIZE; i++) s[i]=w[i%6]+(G=1-G); // Data initialization
     s[SIZE-1]=0;
```

Execution of SEQF10K without trace (TRACE=0)



```
Execution of program C:\CentralFiles\Jozo\jozo-programs\P R O G R A M S\BenchMaker\BenchMaker 2\PROJEC


  NUMBER OF EXECUTED KERNELS = 10
  MEASURED RUN TIME [sec]    = 6.516

End of program (SEQF program size = 10K ,  333L)
Press any key to continue
```

Execution of SEQF10K with trace (TRACE=1)



```
Execution of program C:\CentralFiles\Jozo\jozo-programs\P R O G R A M S\BenchMaker\BenchMaker 2\PROJEC

1   C11301: Counting words in a string
2   C11601: Class with Bubble Sort function
3   C11602: Class with QuickSort function
4   C11603: Class with Select Sort function
5   C11604: Embedded Binary Search
6   C11605: Embedded Bubble Sort
7   C11606: Embedded Select Sort
8   C11607: Class with an iterative binary search function
9   C11608: Embedded merge of sorted arrays
10   C11609: Class with a merge function

  NUMBER OF EXECUTED KERNELS = 10
  MEASURED RUN TIME [sec]    = 6.515

End of program (SEQF program size = 10K ,  333L)
Press any key to continue
```

Summary of BM2 properties

- Flexible adjustment of program structure
- Easy adjustment of program size
- Executable programs, easy adjustment of run time
- Semantic interpretation and unlimited adjustment of workload characteristics (procedural, object oriented, file I/O, numeric, nonnumeric, arrays, etc.)
- Almost all code is expertly generated by humans
- Fast code generation and correct compilation
- Scalability and calibration
- Expandability of library kernels
- Suitability for evaluation and comparison of computer performance for different types of workload
- Suitability for open-source development

Towards Open Source Benchmark Manufacturing

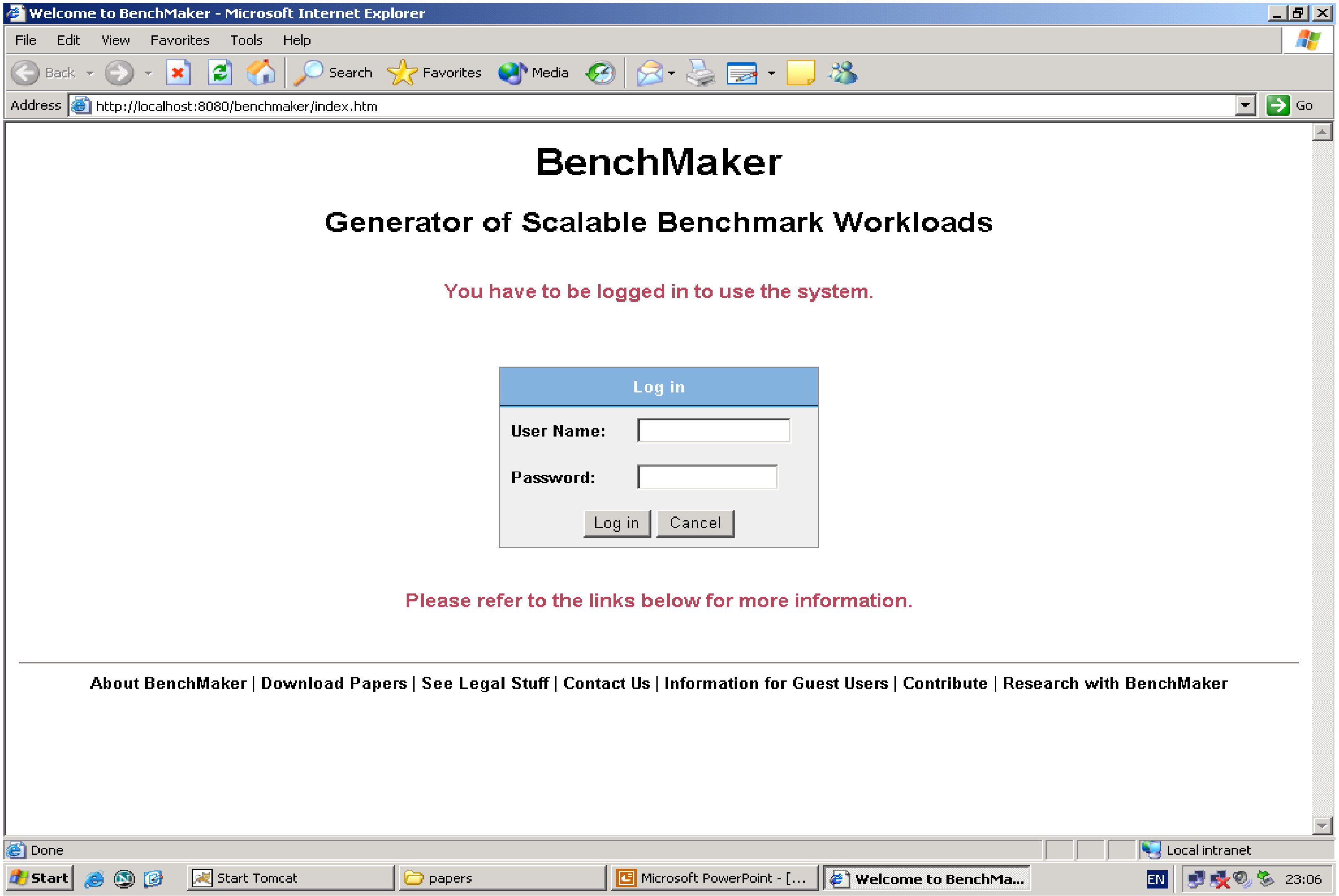
Basic Goals

- Create an environment where users can manufacture scalable benchmark workloads based on their individual needs
- Create a user community that contributes to an open-source kernel library
- Encourage research in the area of workload characterization, benchmark scalability, and program cloning

BenchMaker User Interface (1/9)

- Web based, dynamic interface
- JSP & Java based, outputs are pure HTML
- Most browsers are supported
- Tomcat4.1 on the server side
- List of kernels are read at run-time from configuration files and the interface adapts itself to changes
- Simple to use
- Support for e-mail retrieval of benchmarks
- Supports multiple users and projects

BenchMaker User Interface (2/9)



BenchMaker

Generator of Scalable Benchmark Workloads

You have to be logged in to use the system.

Log in

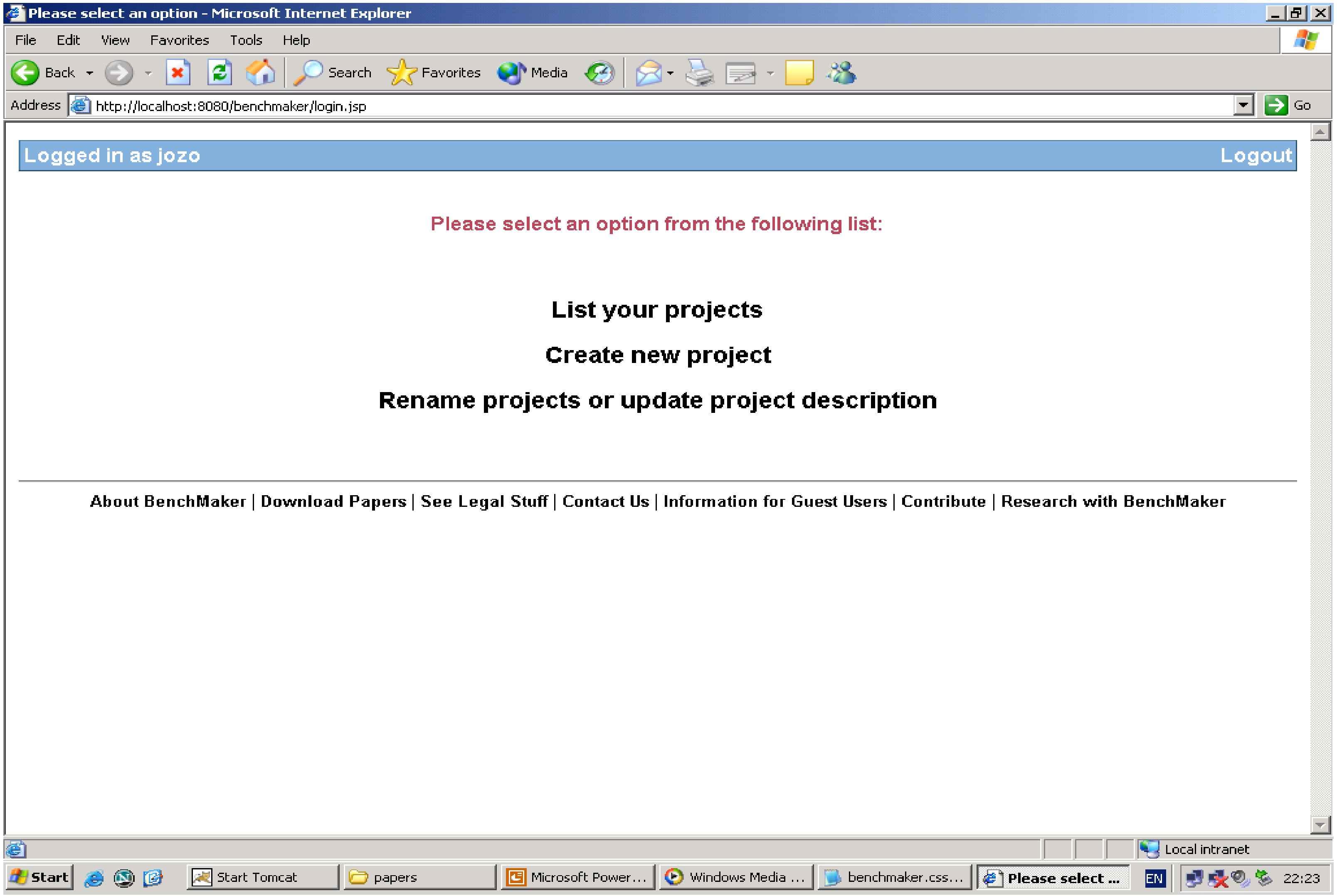
User Name:

Password:

Please refer to the links below for more information.

About BenchMaker | Download Papers | See Legal Stuff | Contact Us | Information for Guest Users | Contribute | Research with BenchMaker

BenchMaker User Interface (3/9)



Logged in as jozo | Logout

Please select an option from the following list:

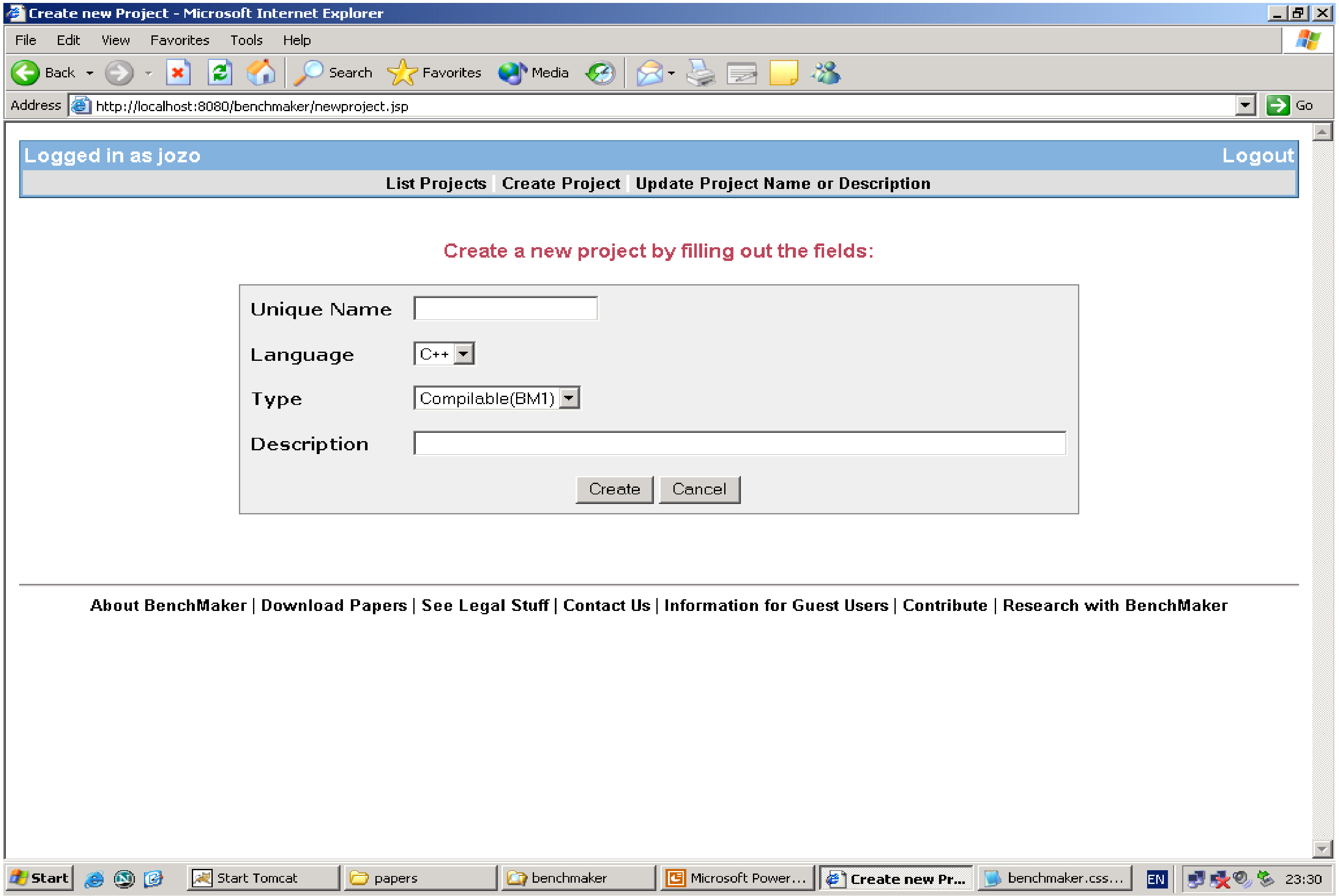
List your projects

Create new project

Rename projects or update project description

About BenchMaker | Download Papers | See Legal Stuff | Contact Us | Information for Guest Users | Contribute | Research with BenchMaker

BenchMaker User Interface (4/9)



Logged in as jozo

Logout

List Projects | Create Project | Update Project Name or Description

Create a new project by filling out the fields:

Unique Name

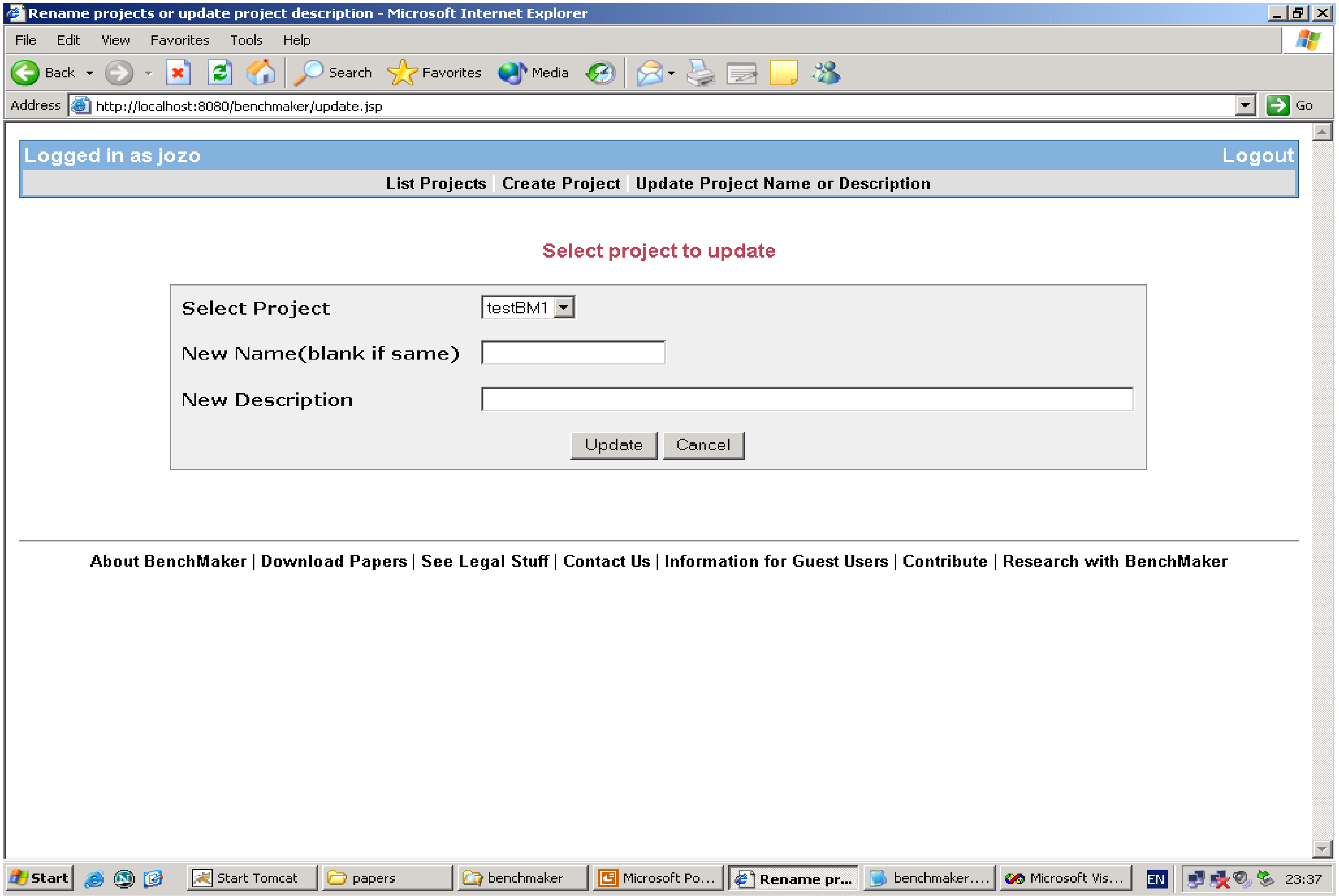
Language C++

Type Compilable(BM1)

Description

About BenchMaker | Download Papers | See Legal Stuff | Contact Us | Information for Guest Users | Contribute | Research with BenchMaker

BenchMaker User Interface (5/9)

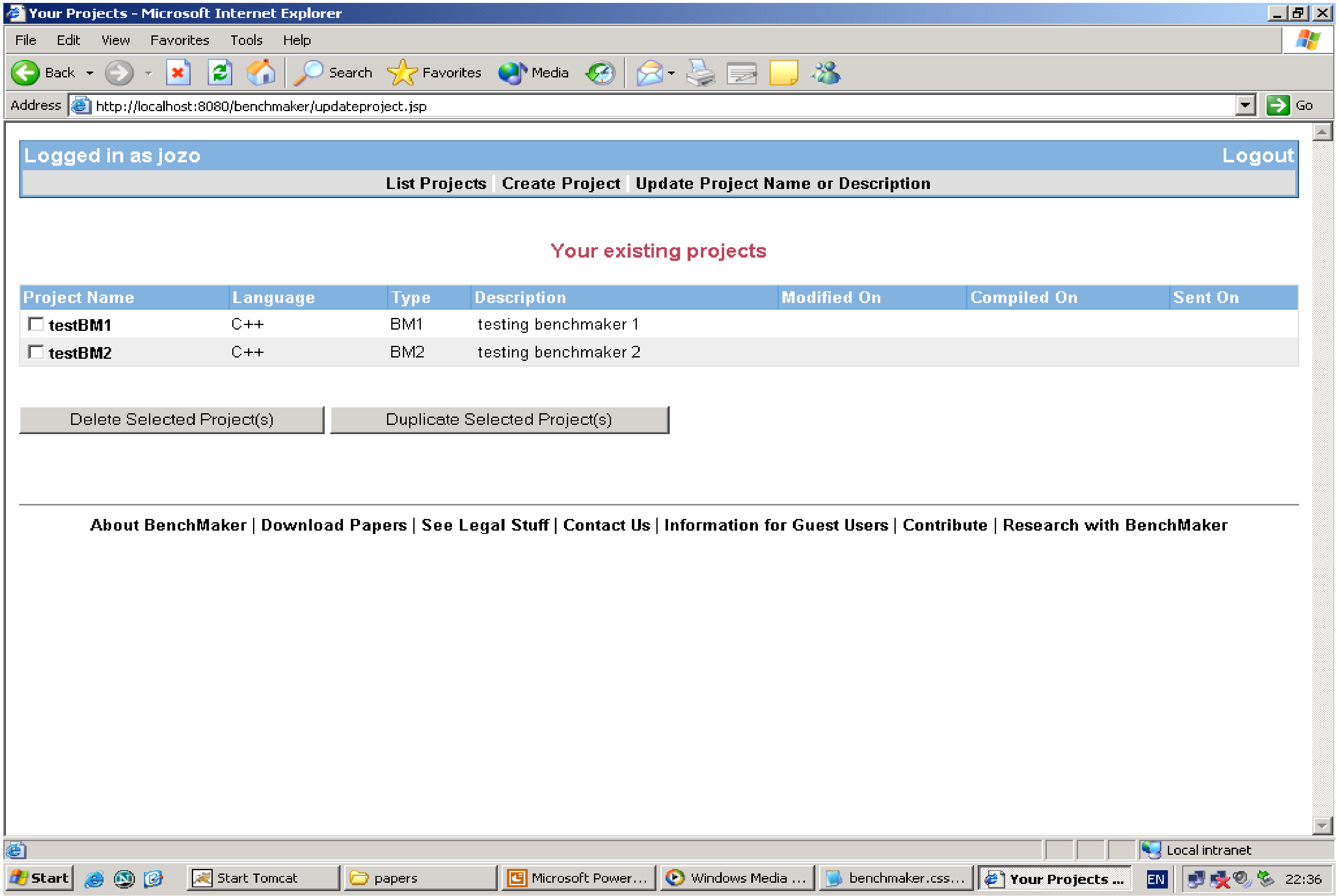


http://localhost:8080/benchmaker/update.jsp

Select project to update

Select Project	testBM1
New Name(blank if same)	
New Description	

BenchMaker User Interface (6/9)



Logged in as jozo

Logout

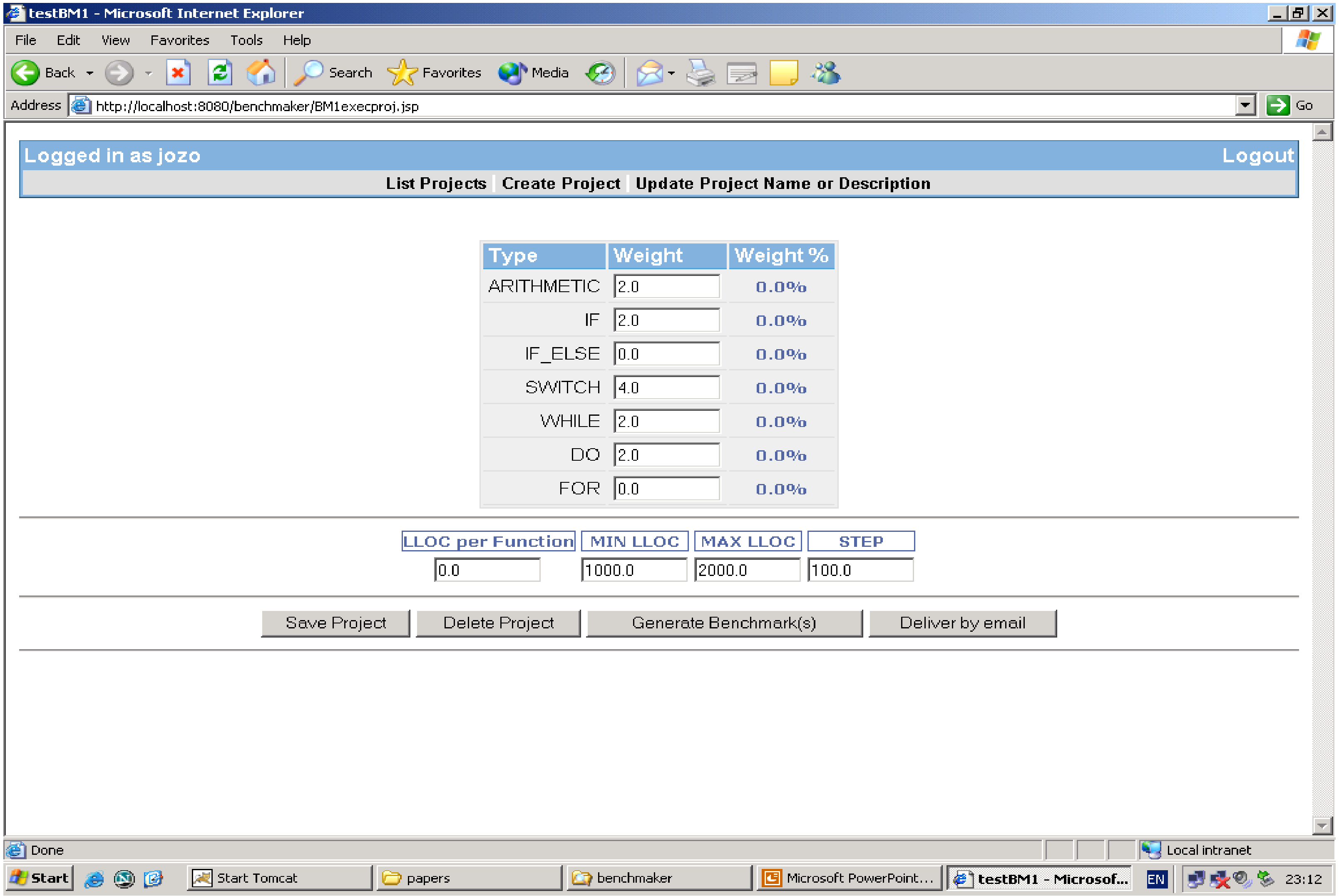
List Projects | Create Project | Update Project Name or Description

Your existing projects

Project Name	Language	Type	Description	Modified On	Compiled On	Sent On
☐ **testBM1**	C++	BM1	testing benchmaker 1			
☐ **testBM2**	C++	BM2	testing benchmaker 2			

About BenchMaker | Download Papers | See Legal Stuff | Contact Us | Information for Guest Users | Contribute | Research with BenchMaker

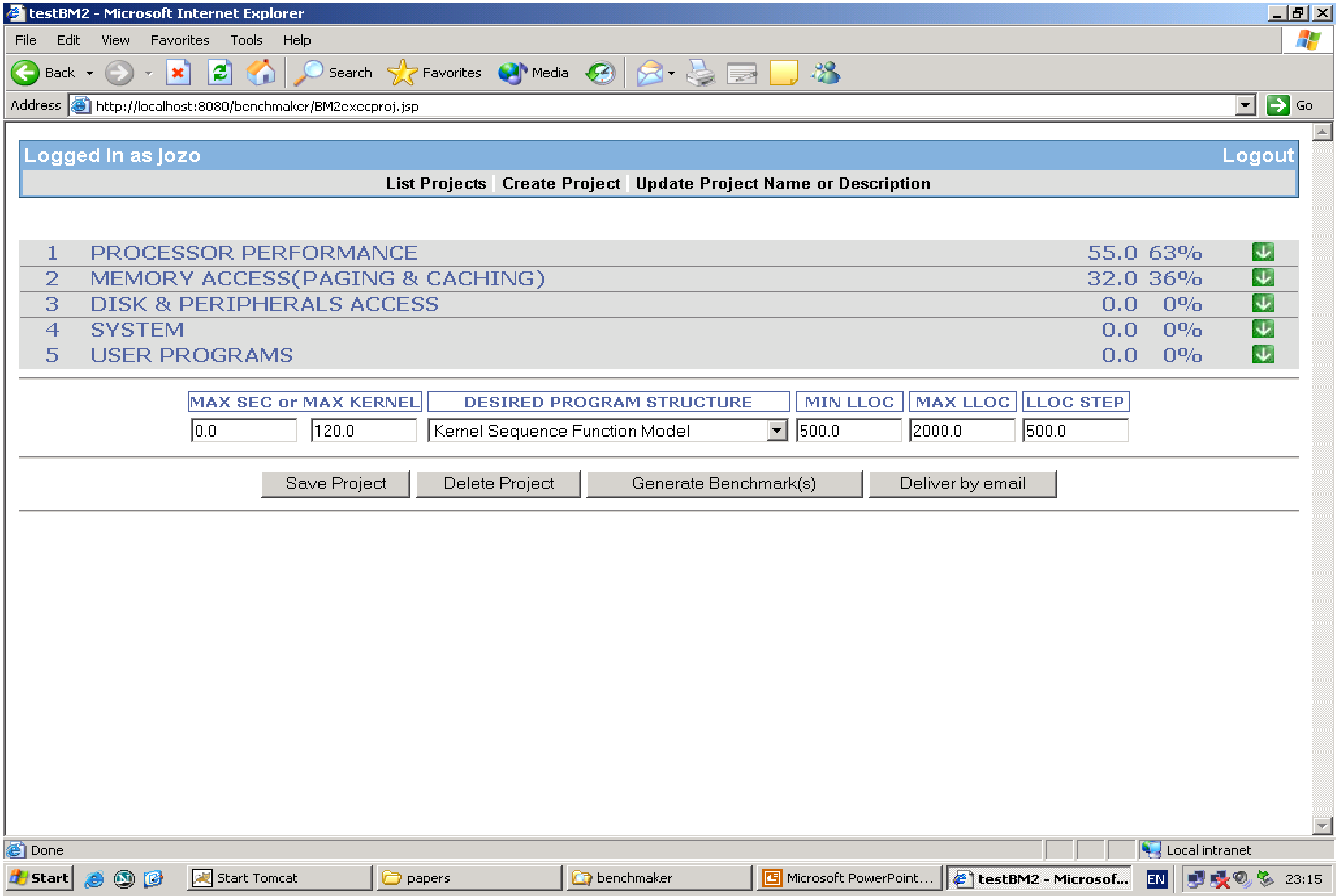
BenchMaker User Interface (7/9)



Type	Weight	Weight %
ARITHMETIC	2.0	0.0%
IF	2.0	0.0%
IF_ELSE	0.0	0.0%
SWITCH	4.0	0.0%
WHILE	2.0	0.0%
DO	2.0	0.0%
FOR	0.0	0.0%

LLOC per Function	MIN LLOC	MAX LLOC	STEP
0.0	1000.0	2000.0	100.0

BenchMaker User Interface (8/9)



1	PROCESSOR PERFORMANCE	55.0	63%
2	MEMORY ACCESS(PAGING & CACHING)	32.0	36%
3	DISK & PERIPHERALS ACCESS	0.0	0%
4	SYSTEM	0.0	0%
5	USER PROGRAMS	0.0	0%

MAX SEC or MAX KERNEL		DESIRED PROGRAM STRUCTURE	MIN LLOC	MAX LLOC	LLOC STEP
0.0	120.0	Kernel Sequence Function Model	500.0	2000.0	500.0

BenchMaker User Interface (9/9)

1	PROCESSOR PERFORMANCE	55.0	52%		
2	MEMORY ACCESS(PAGING & CACHING)	32.0	30%		
3	DISK & PERIPHERALS ACCESS	18.0	17%		
31	Disk Access	15.0	14%		
310	Miscellaneous	5.0	4%	Preset	Custom
31001	Miscellaneous			○ ○ ○ ○ ○ ● ○	5.0
311	Sequential Access	5.0	4%	Preset	Custom
31101	Sequential Access			○ ○ ○ ○ ○ ● ○	5.0
312	Random Access	5.0	4%	Preset	Custom
31201	Random Access			○ ○ ○ ○ ○ ● ○	5.0
32	Miscellaneous Peripheral Access	3.0	2%		
320	Miscellaneous	3.0	2%	Preset	Custom
32001	Miscellaneous			○ ○ ○ ● ○ ○ ○	3.0
321	VDU & Graphics	0.0	0%	Preset	Custom
32101	VDU & Graphics			● ○ ○ ○ ○ ○ ○	0.0
322	Archival Tape Access	0.0	0%	Preset	Custom
32201	Archival Tape Access			● ○ ○ ○ ○ ○ ○	0.0
4	SYSTEM	0.0	0%		
5	USER PROGRAMS	0.0	0%		

MAX SEC or MAX KERNEL		DESIRED PROGRAM STRUCTURE	MIN LLOC	MAX LLOC	LLOC STEP
0.0	120.0	Kernel Sequence Function Model	500.0	2000.0	500.0

Applications of Benchmark Program Generators

(Compiler Performance and Computer Performance)

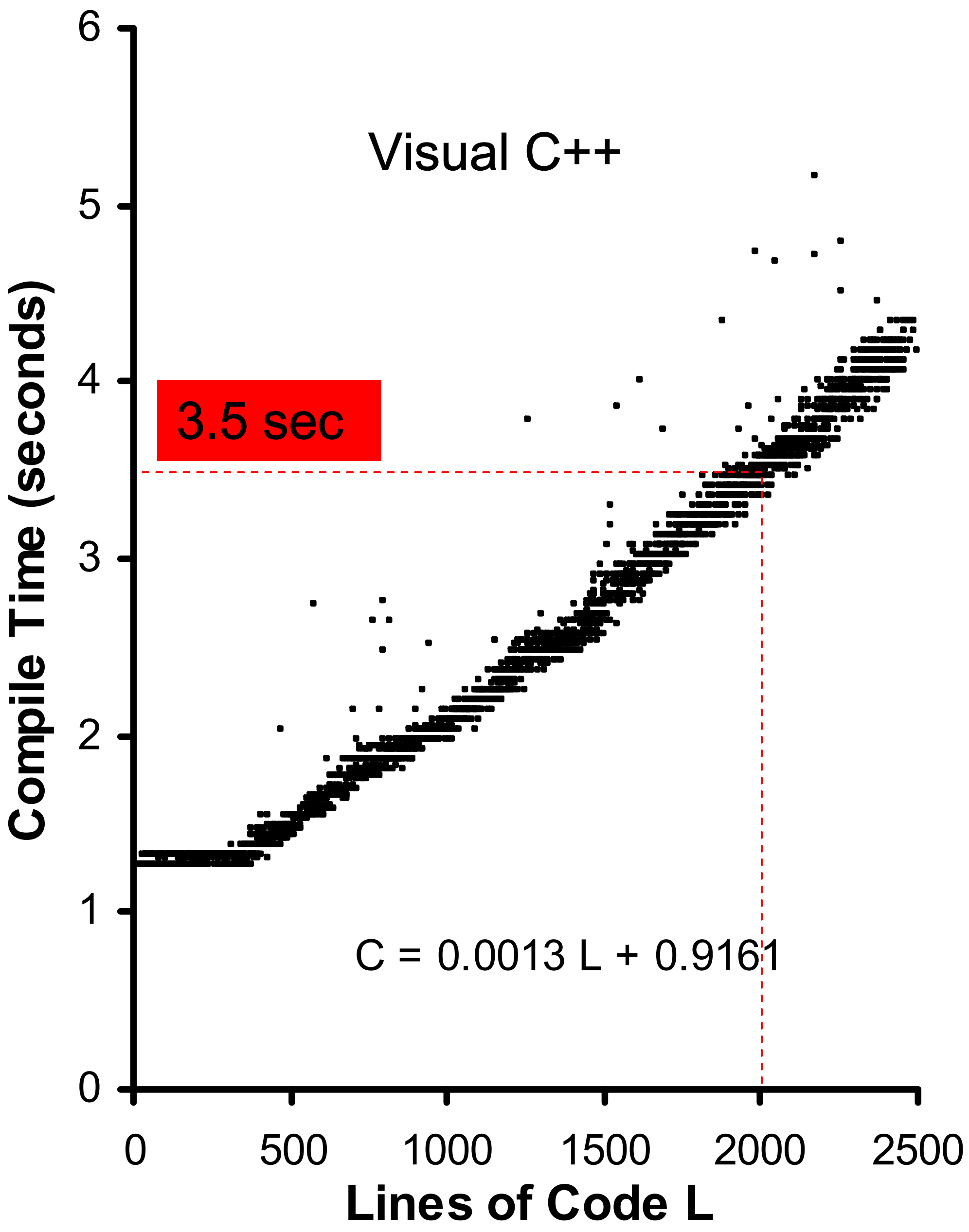
Compiler Performance Analysis

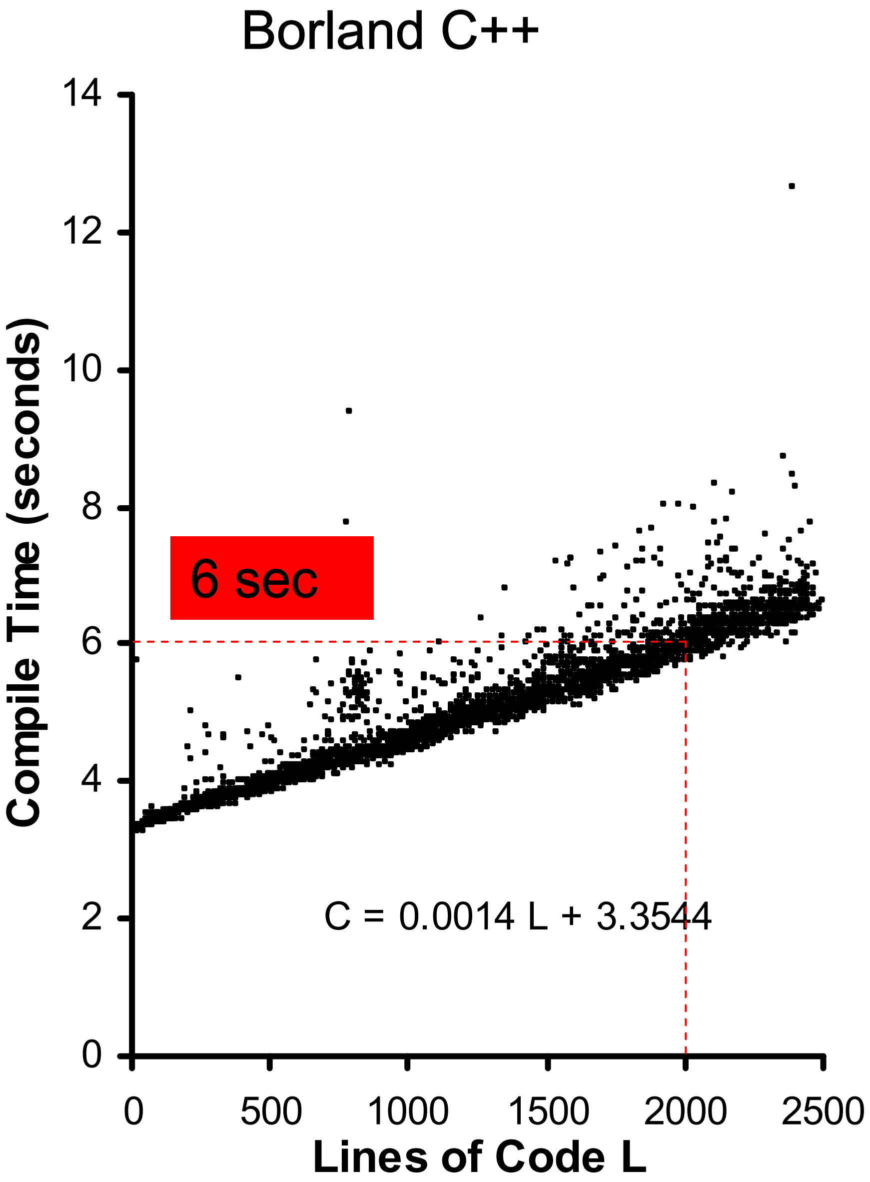
- Compile time
- Memory consumption
  - Object program
  - Executable program
- Maximum program size
- Nonlinear phenomena
- Execution time

Compile Time (C) as a Function of Program Size (L)

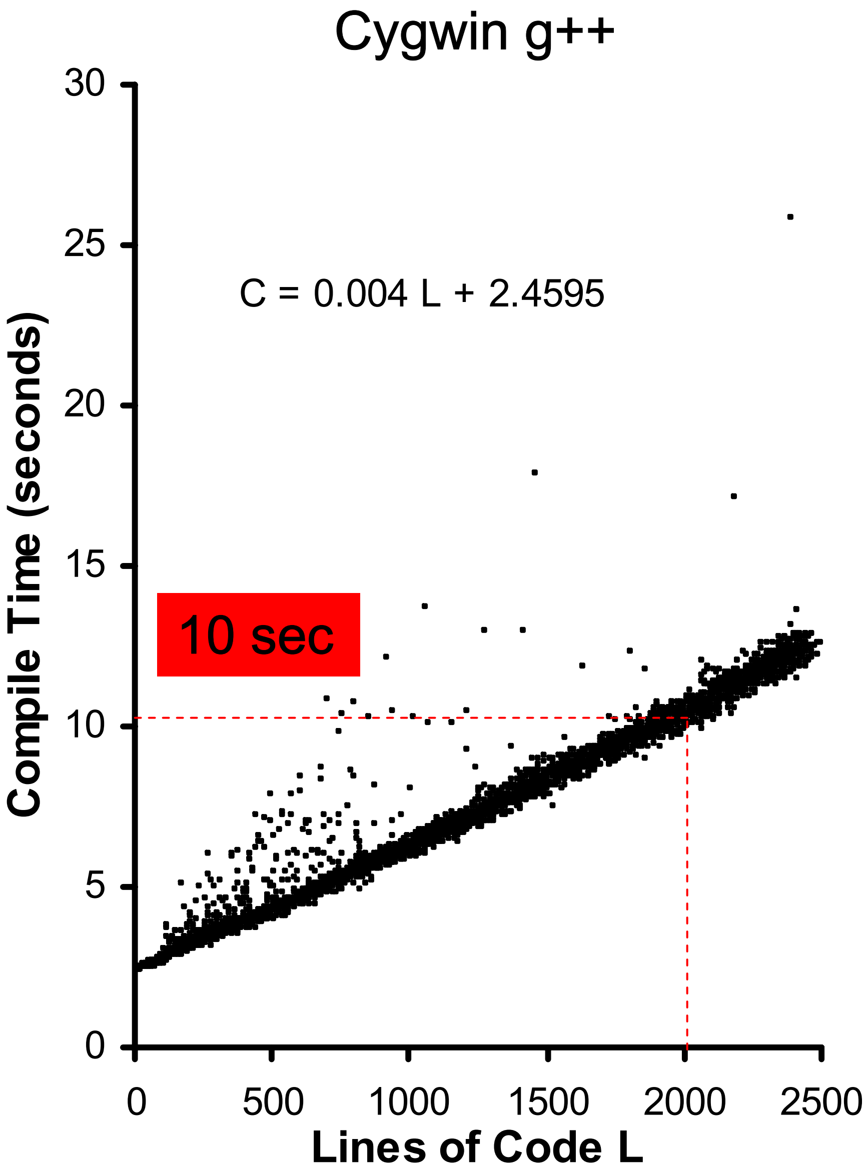
$$C = t_0 + t_1 L^q, \quad q \geq 1$$

This analysis is based on 3500 synthetic benchmark programs generated using the BM1 program generator

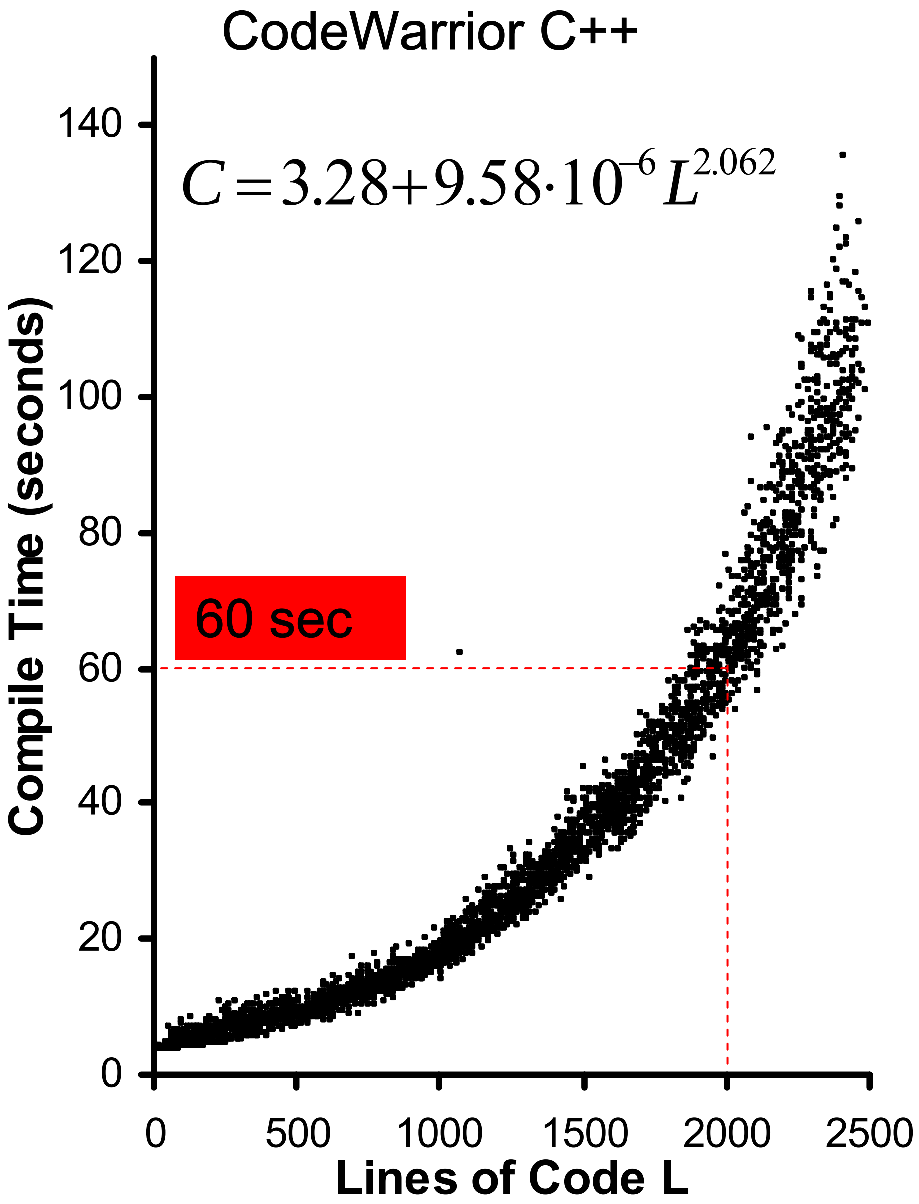




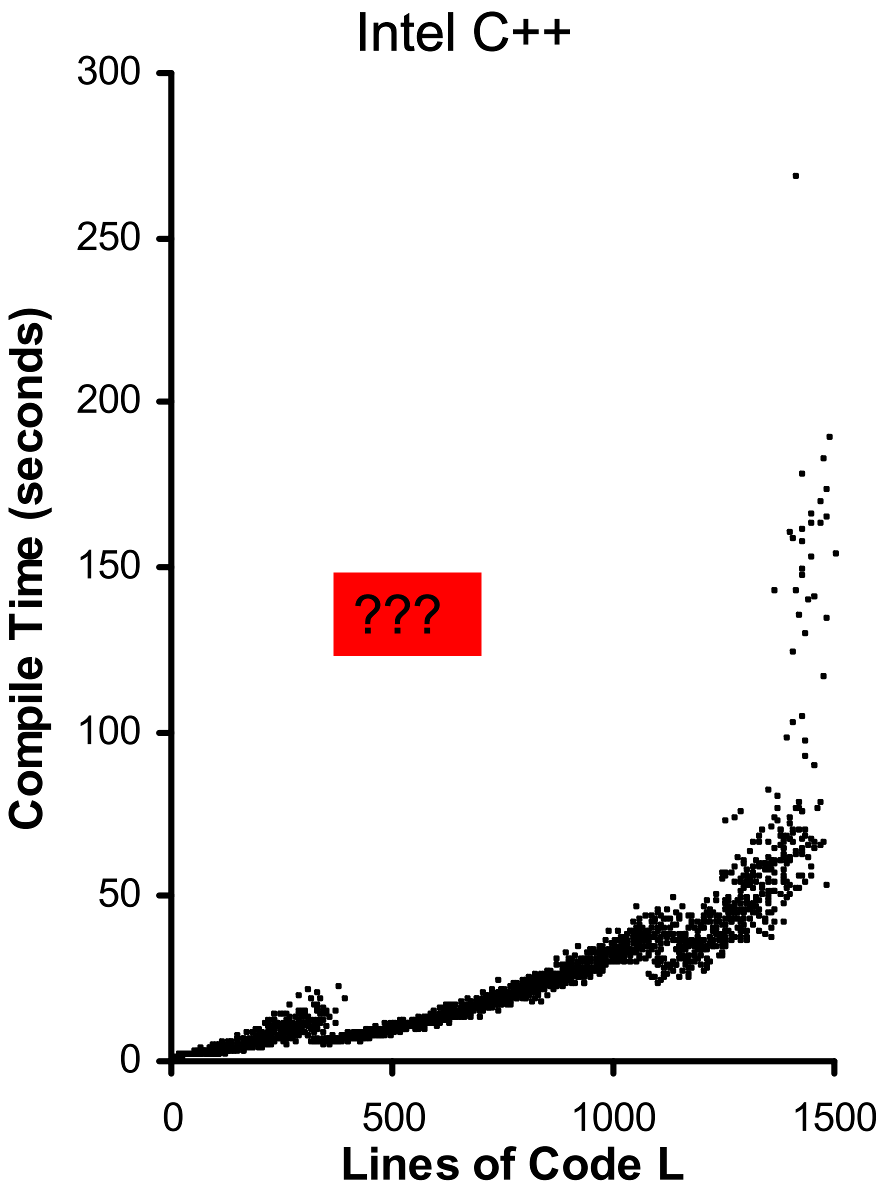
Borland C++



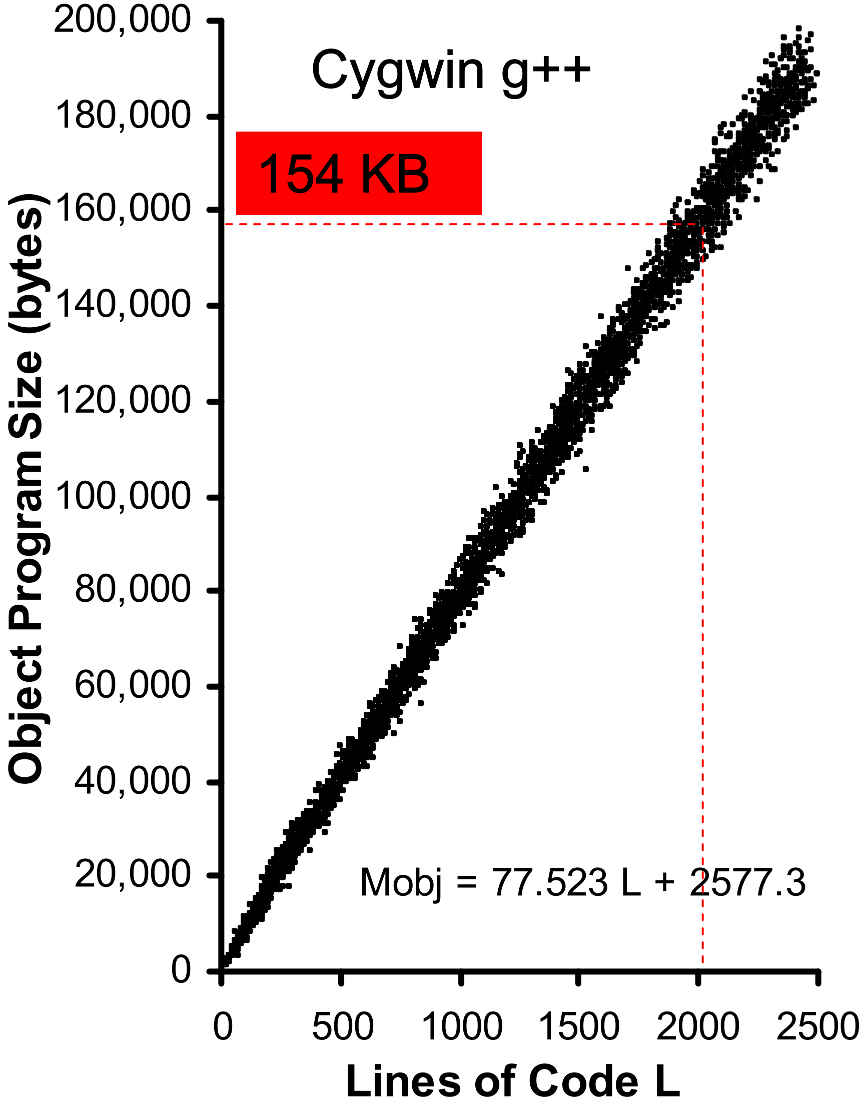
Cygwin g++

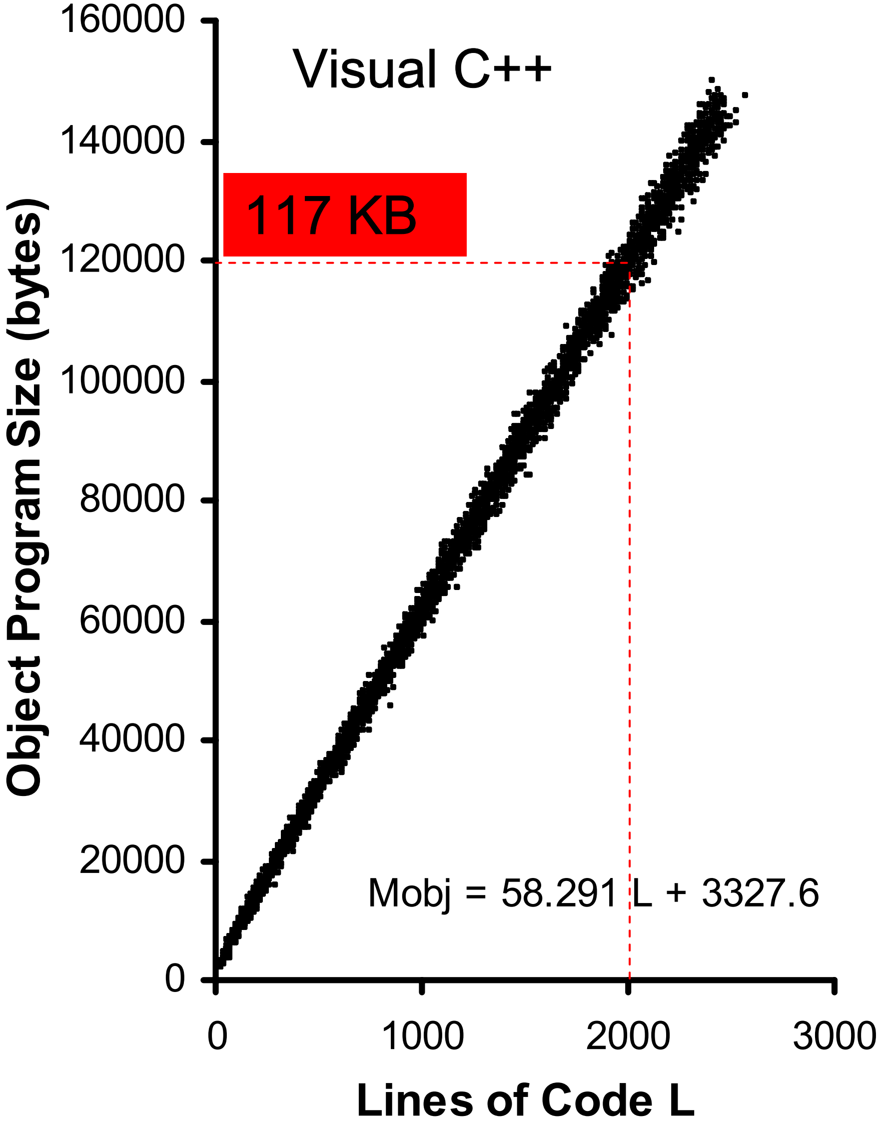


$$C = 3.28 + 9.58 \cdot 10^{-6} L^{2.062}$$



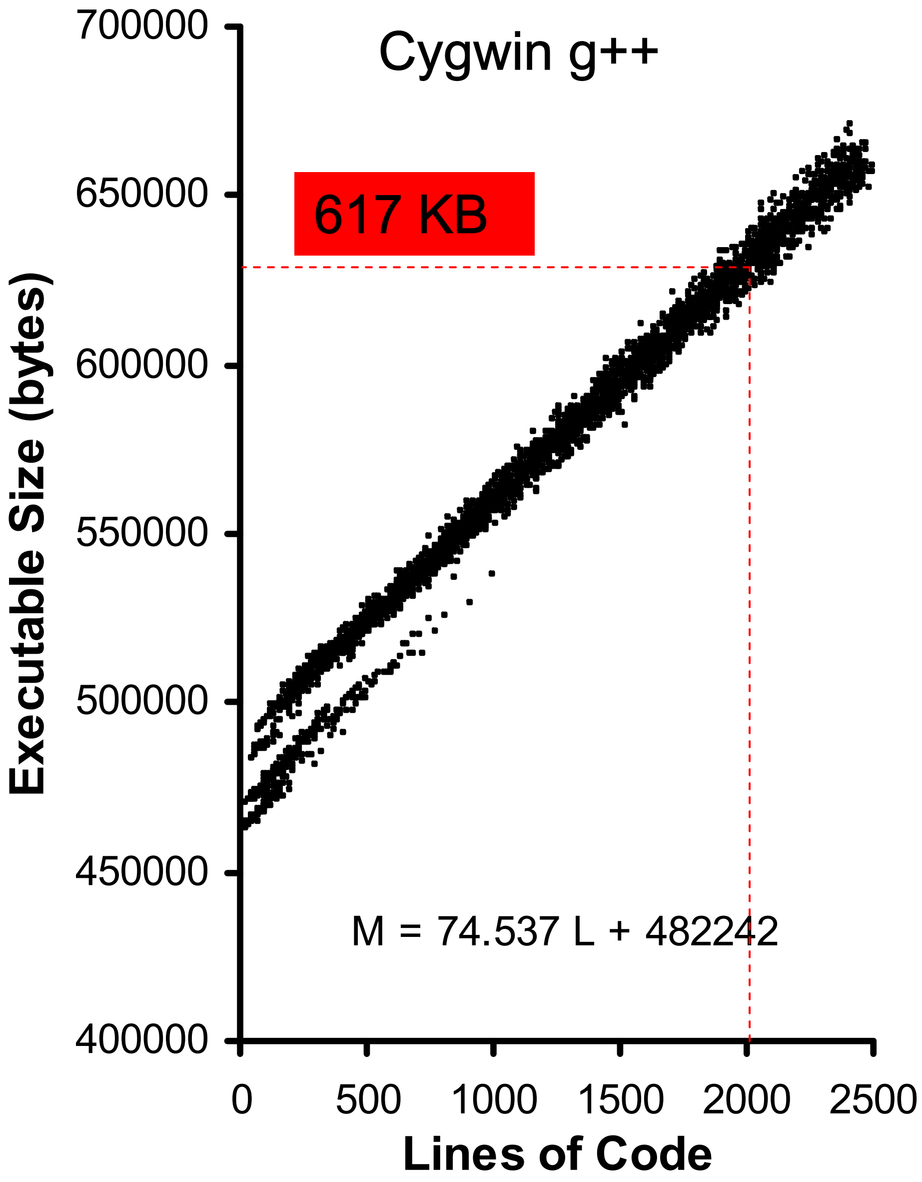
Comparison of Object Program Sizes



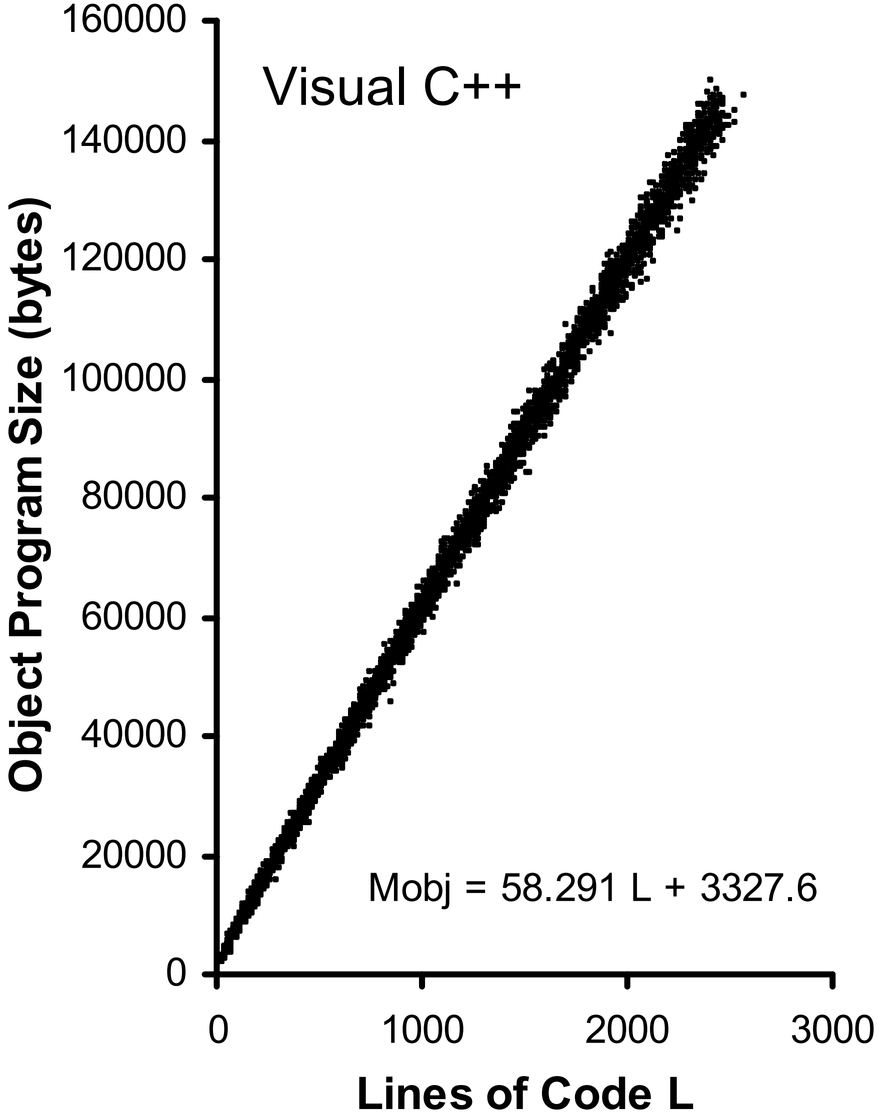


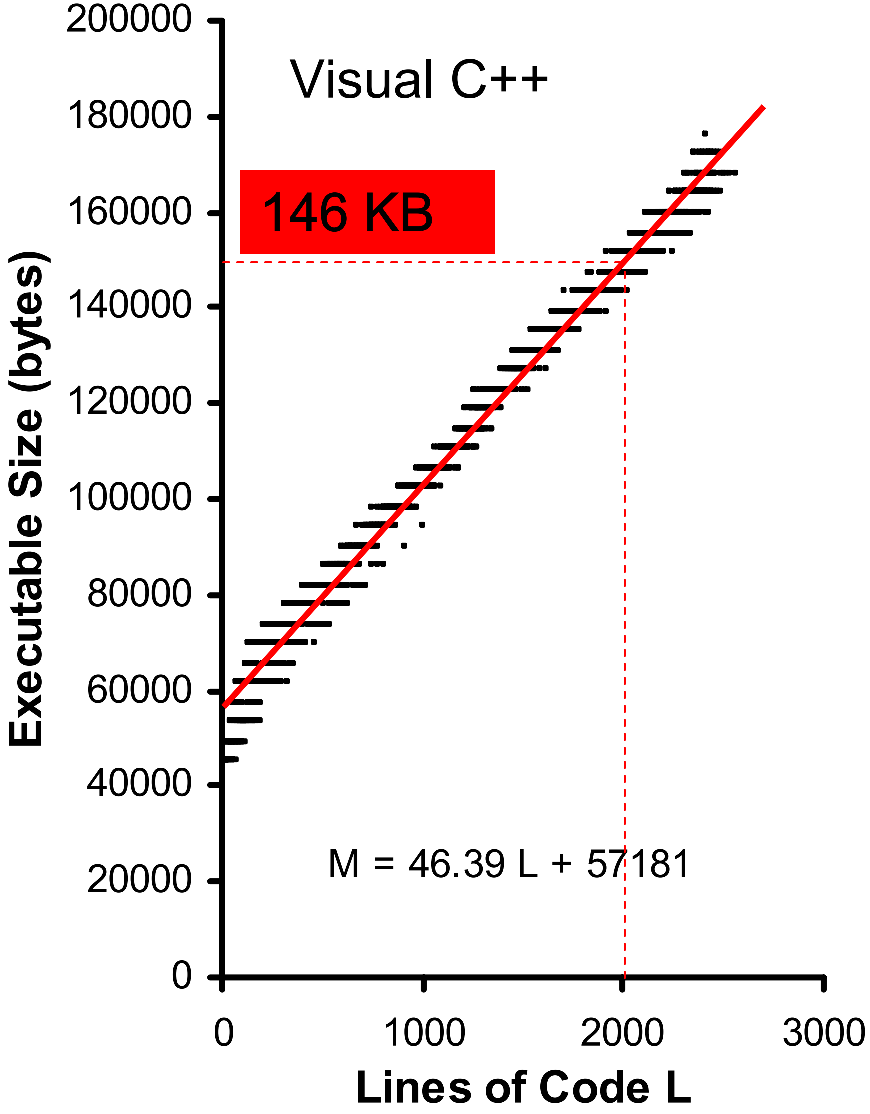
Memory Consumption (M) as a Function of Program Size (L)

$$M = m_0 + m_1 L$$

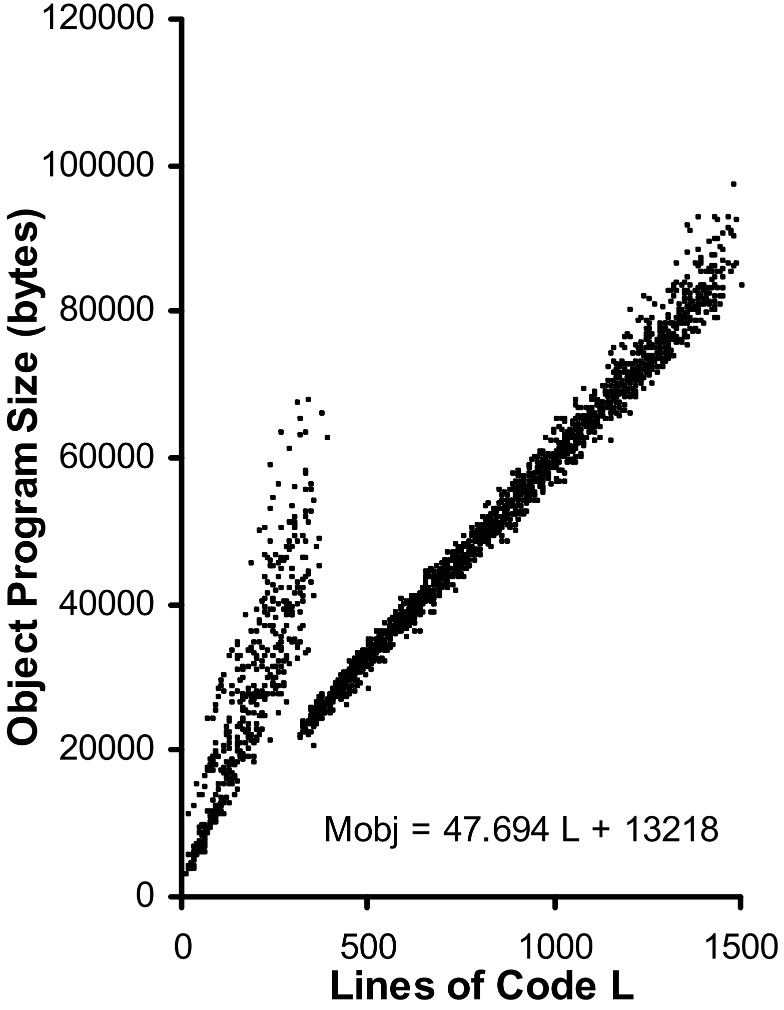


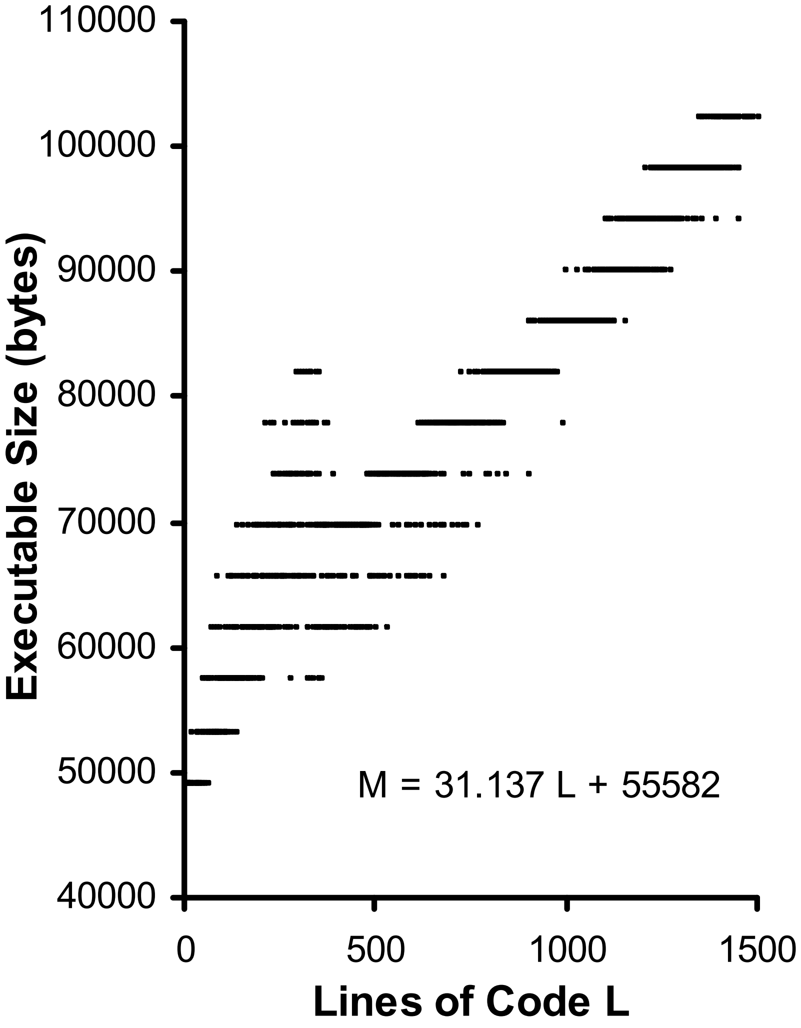
Object Program Size vs. Executable Program Size



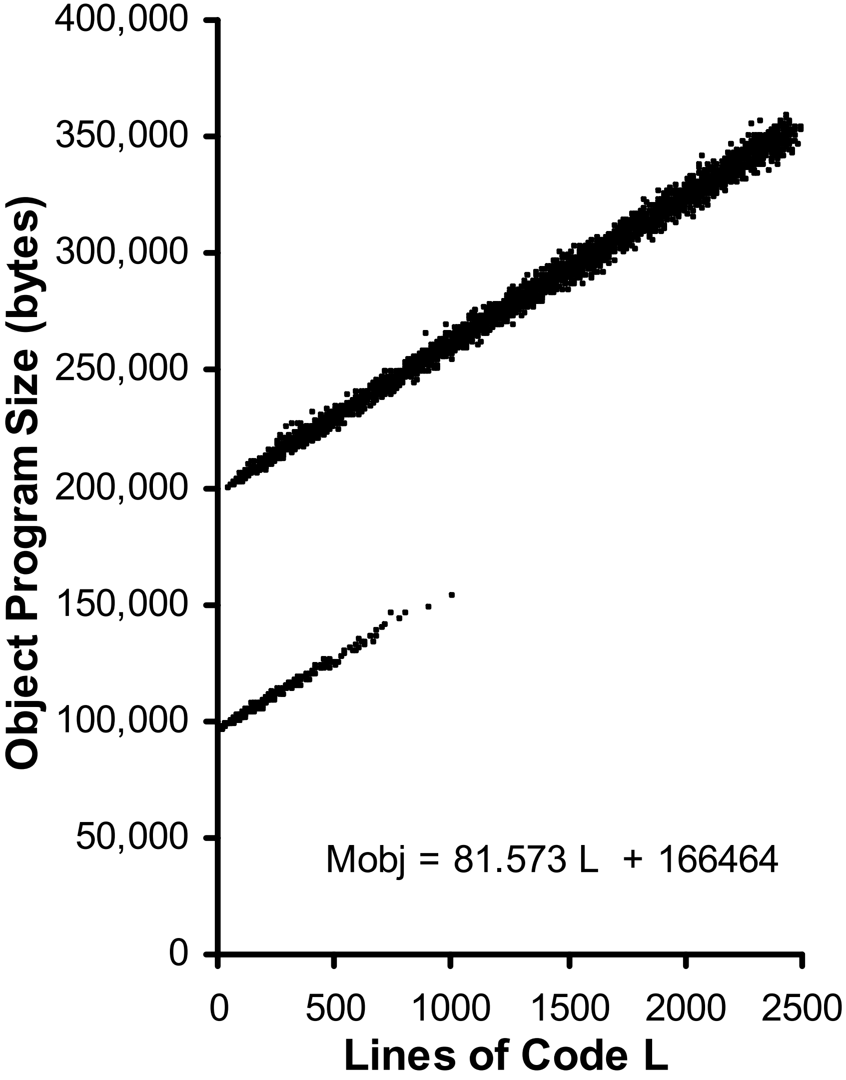


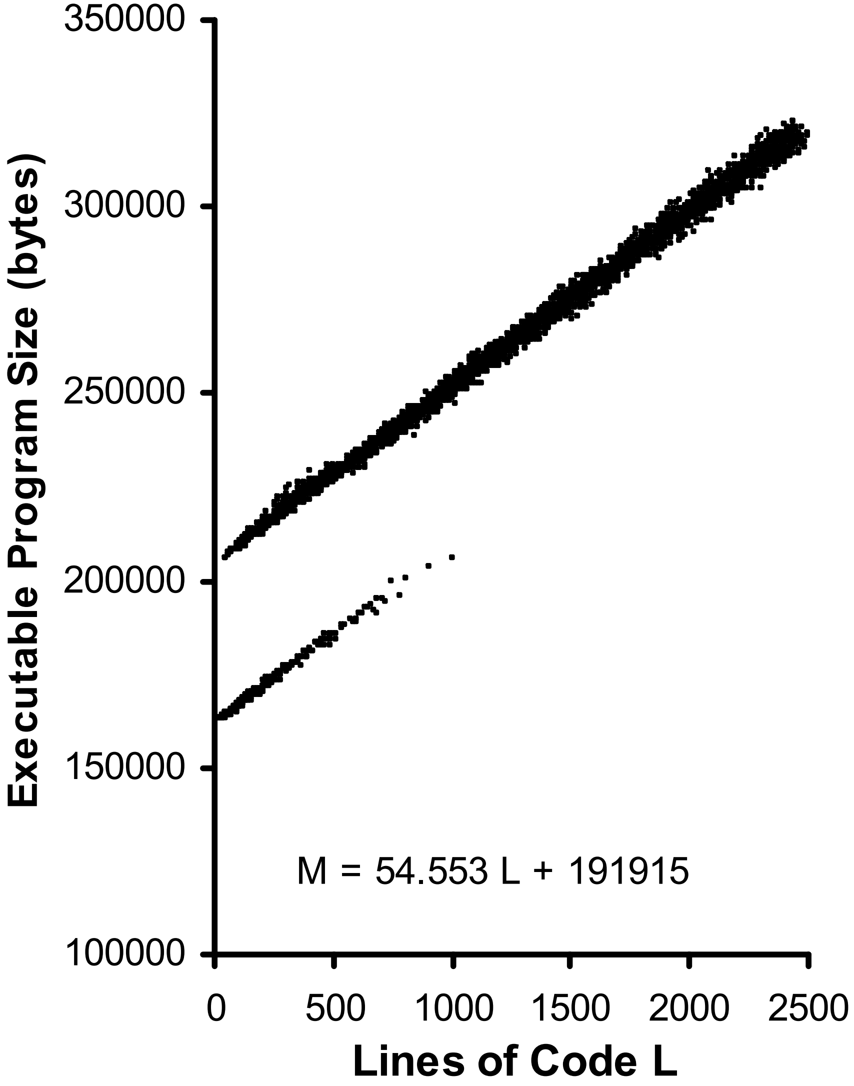
Nonlinear Phenomena – Intel C++ Compiler





Nonlinear Phenomena – Metrowerks CodeWarrior

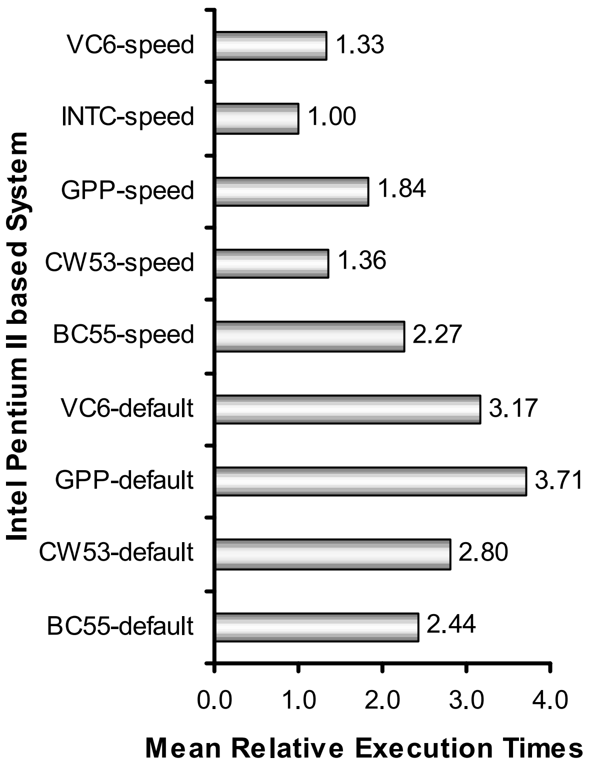


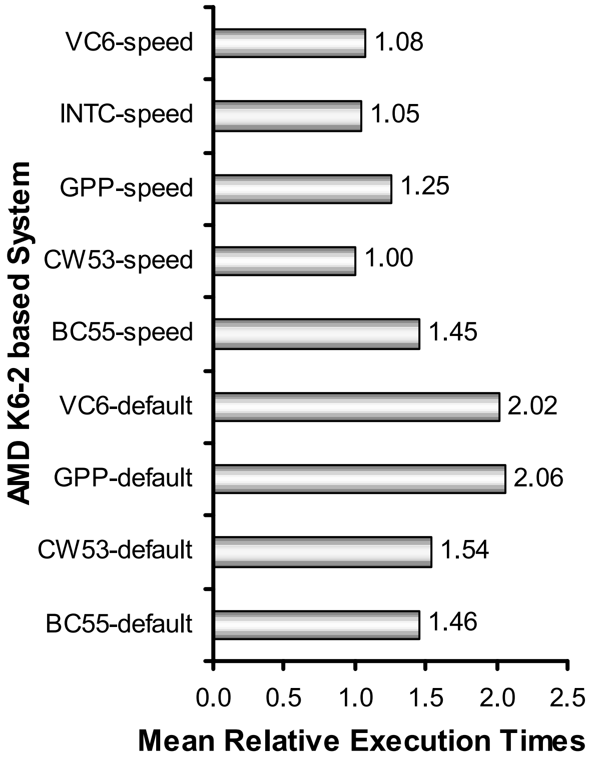


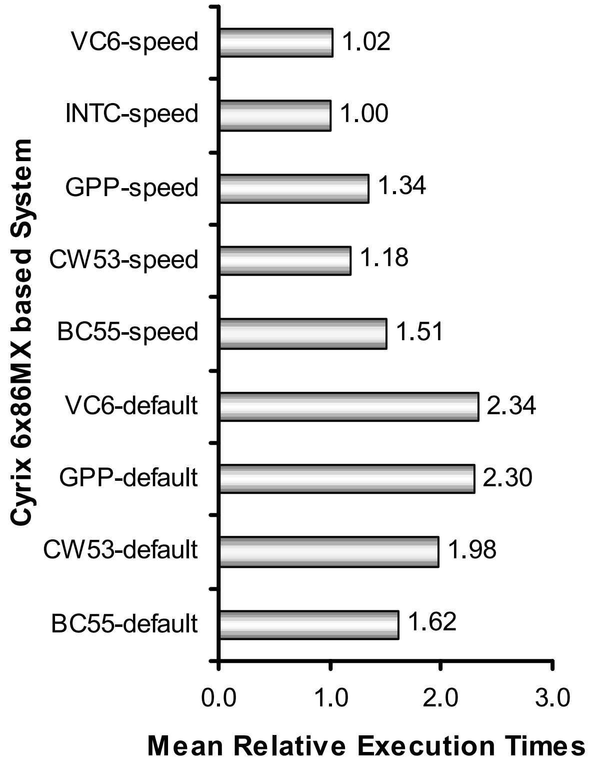
Execution Time Comparison

Compilers: *Imprise Borland C++ 5.5, Intel C/C++ Compiler 4.5, Metrowerks CodeWarrior 5.3, Microsoft Visual C++ 6.0, and Redhat Cygwin b20 (based on GNU compiler tools)*

Processors: Intel Pentium II 300 , AMD K6-2 350 , Cyrix 6x86MX-PR166







Performance ranking of compilers using a Pentium based system

Execution time ratio:

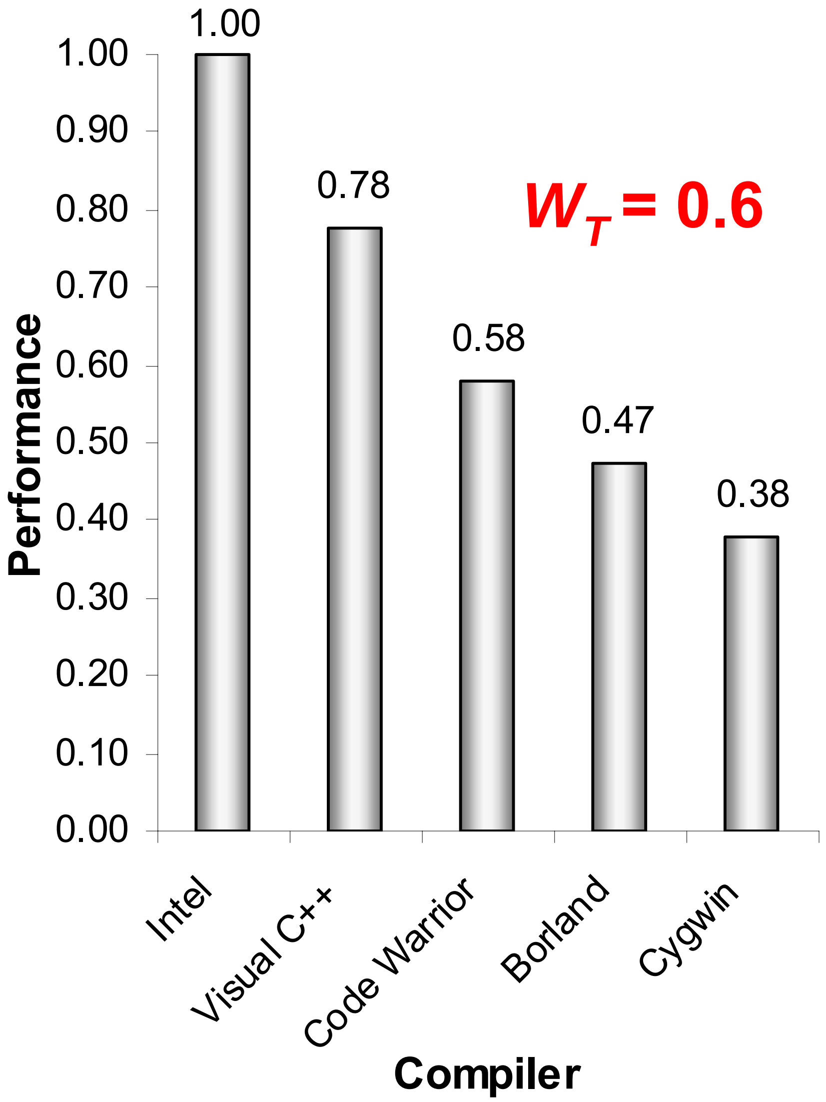
$$r = \left(\frac{T_{1A}}{T_{1B}} \cdot \frac{T_{2A}}{T_{2B}} \cdots \frac{T_{nA}}{T_{nB}} \right)^{1/n}$$

Global criterion:

$$R = r^{W_T} \left(\frac{m_{0A}}{m_{0B}} \right)^{W_{m0}} \left(\frac{m_{1A}}{m_{1B}} \right)^{W_{m1}} \left(\frac{t_{0A}}{t_{0B}} \right)^{W_{t0}} \left(\frac{t_{1A}}{t_{1B}} \right)^{W_{t1}}$$

Release criterion (compilation speed omitted):

$$R = r^{W_T} \left(\frac{m_{0A}}{m_{0B}} \right)^{(1-W_T)/2} \left(\frac{m_{1A}}{m_{1B}} \right)^{(1-W_T)/2}, \quad 0 \le W_T \le 1.$$



Performance Comparison Model

$$P_{ij} = 100 \prod_{k=1}^{n} \left(\frac{R_{ik}}{R_{jk}} \right)^{W_k} \qquad [\%]$$

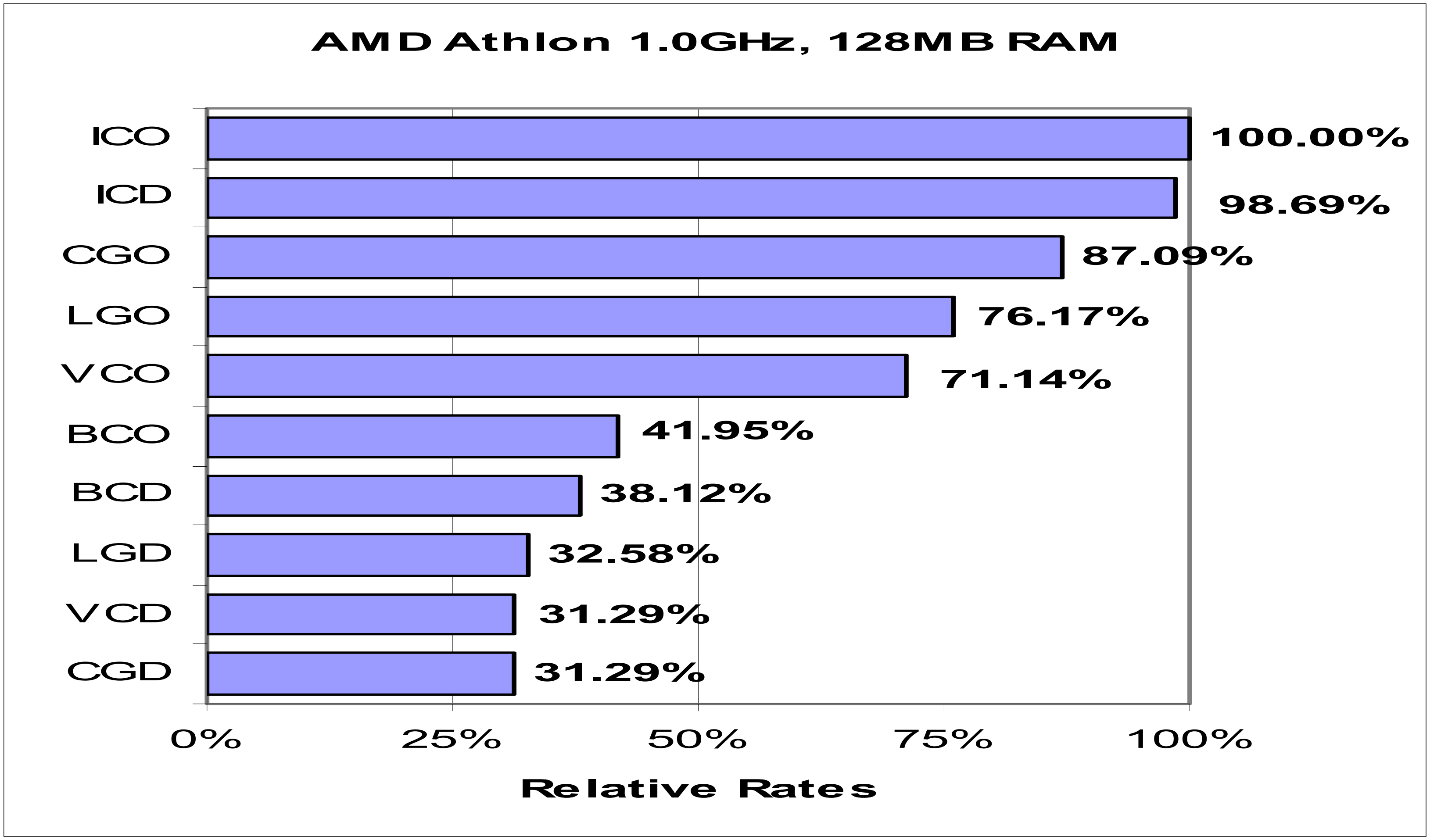
$$\sum_{k=1}^{n} W_k = 1\,, \quad 0 < W_k < 1,\ k = 1,\ldots,n\ .$$

A general comparison of compilers can be based on using the geometric mean with equal rates ($W_1 = \ldots = W_n = 1/n$).

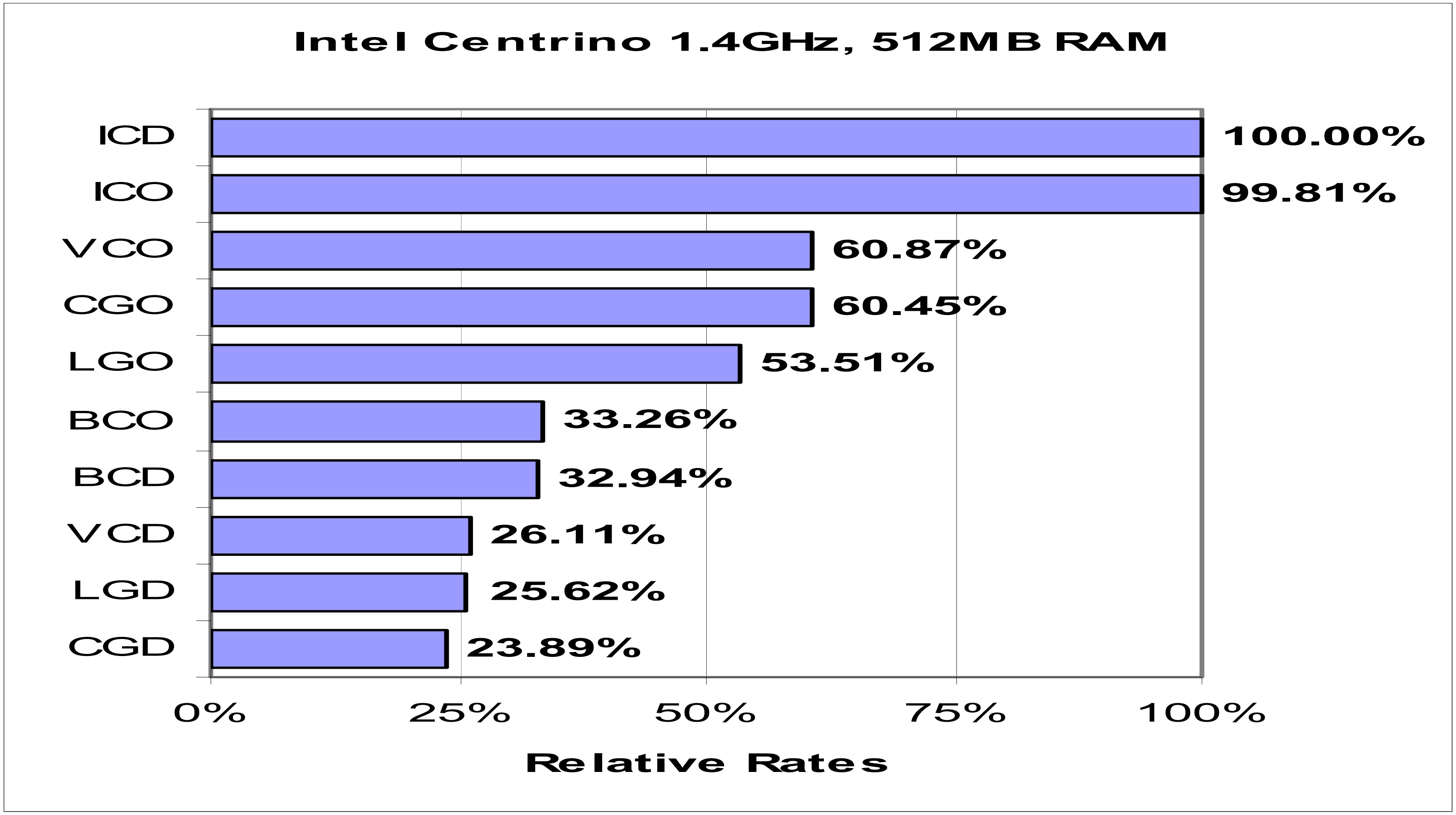
Using Calibration for Performance Comparison (1/3)

- VCO = Microsoft Visual C++ 6.0, release version
- VCD = Microsoft Visual C++ 6.0, debug version
- ICO = Intel C++ 7.1, optimized version
- ICD = Intel C++ 7.1, default version
- BCO = Borland C++ 5.5, optimized version
- BCD = Borland C++ 5.5, default version
- CGO = Cygwin g++ 3.2, -O3 optimized version
- CGD = Cygwin g++ 3.2, default version
- LGO = Linux g++ 3.2.2, -O3 optimized version
- LGD = Linux g++ 3.2.2, default version

Using Calibration for Performance Comparison (2/3)



Using Calibration for Performance Comparison (3/3)



Observations (1/3)

- Various software environments offer a wide spectrum of different performance levels. On the same hardware the proper selection of compiler can sometimes produce dramatic speedup. Optimum versions of compilers can differ in performance up to **3** times. Versions with different parameters can differ up to **4** times.
- Debug versions of compilers substantially slow down the execution process (**typically 2 to 3 times**).

Observations (2/3)

- Intel C++ compiler consistently outperforms competitors on both tested machines.
- Intel C++ compiler advantage over other compilers is bigger for Centrino (Pentium M) then for AMD.
- One of unexpected results is that on measured machines the Cygwin environment with GNU C++ outperforms the native Linux environment. In the case of AMD we used Red Hat Linux, and in the case of Centrino we used Mandrake Linux.

Observations (3/3)

- Some C++ compilers (e.g. Intel) use default version that is close to the most optimized version.
- Some compilers have default and/or debug versions significantly slower than the optimized version.

Conclusions

- Exponential growth of computer performance causes a need for fast development of new benchmarks
- Benchmark program generators are tools that provide:
  - High speed and low cost of test and benchmark program generation
  - Flexibility in workload characterization
  - Scalability of resulting workloads
  - A way towards program cloning

Primary source

Dujmović, J.J., Automatic Generation of Benchmark and Test Workloads. Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering, ISBN 978-1-60558-563-5, pp. 263-273, San Jose, CA, USA Jan 28-30, 2010.

Other publications

Dujmović, J.J., E. Horvath, H. Lew, **Benchmark Program Generator for Compiler Performance Analysis.** The 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems. CMG 99 Proceedings, Vol. 2, pp. 838-847, 1999.

Lew, H. and J.J. Dujmović, **Performance Evaluation and Comparison of C++ Compilers**. The 26th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems. CMG 2000 Proceedings, Vol. 1, pp. 241-252, 2000.

Dujmović, J.J. and H. Lew, **A Method for Generating Benchmark Programs**. The 26th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems. CMG 2000 Proceedings, Vol. 1, pp. 379-388, 2000.

Dujmović, J.J. and M. Cengiz, **A Kernel Library for Benchmark Program Generators**. CMG 2003 Proceedings, Vol. 2 pp. 609-618, 2003.

Thanks!

Questions?