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
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 \textit{same workload characteristics} but \textit{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

\[
q(t) = q_0 2^{t/T}
\]

- \( q(0) = q_0 \)
- \( q(T) = 2q_0 \)
- \( q(2T) = 4q_0 \)
- \( q(nT) = 2^n q_0 \)

- \( t = \) time
- \( q = \) performance (speed, mem., cost)
- \( q_0 = \) initial performance at time \( t=0 \)
- \( T = \) performance doubling time
  - \( \approx 18 \) months for memory capacity
  - \( \approx 12 \) 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/
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.

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.
Computer performance

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

Benchmark generators $\Rightarrow$ 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 ∈ \{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: \( PLOC \approx 1.6 \times LLOC \)
Definition of LLOC for C++

For C++ programs we use the following:

\[ \text{LLOC} = \# \text{ of programming units (functions + main)} \]
+ \# 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:

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

( # of ";"); + 1 (keyword))

For loop transformed to while:

```c
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

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

```c
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

```c
int main(arguments) {
    // block
    Terminal Statement
    Expandable Statement
    Terminal Statement 1
}
```

Expansion level (depth) 1

Expansion level (depth) 2

Expandable Statement
Terminal Statement
Terminal Statement
Terminal Statement

Expansion level (depth) 3

Expandable Statement
Terminal Statement
Terminal Statement
Expandable Statement
Terminal Statement
Terminal Statement
Terminal Statement

Terminal Statement 1
Terminal Statement 2
Terminal Statement 3
Terminal Statement 4
Terminal Statement 5
Terminal Statement 6
Terminal Statement 7
Terminal Statement 8
Terminal Statement 9
The Concept of Breadth

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

The Concept of Depth

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

BenchMaker 1&2 Copyright © 2010 by Jozo Dujmović
REX Program Model

• Each block contains one or more statements.
• Each control statement contains one or more blocks. An example of two blocks:
  \[
  \text{if(condition) \{block\} else \{block\}}
  \]
• 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).
string STATEMENT(…)
{
    ..............
    BLOCK(…);
}

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

if (Size > MaxSize)
    return terminal statement;
else
    return a randomly selected statement that includes one or more BLOCK( );

While (Breadth < MaxBreadth):
    append STATEMENT( );
A toy REX generator [1/3]

```c
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

```c
#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 ? ' ':'
        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

$ 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

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 \( F_1(), F_2(), ..., F_n() \), followed by the main program.
3. All programs contain random expressions and control structures.

The available control structures are:

BenchMaker1 (bm1) is normally called using a command line parameter:
\[ \text{bm1} \text{ project directory path} \]
\[ \text{bm1} \text{ "project directory path"} \]
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  
IF_ELSE 2  
SWITCH 3  
WHILE 4  
DO 2  
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
Project directory path (enter "." for default parameters) = .
Project Directory Path = .
Project Name = default
Program Name = \BM\default1.cpp
Input Parameter File Name = bminpar.txt
Output Parameter File Name = bmdloutpar.txt

Default: Uniform distribution of control structures
  Generation of a single program \BM\default1.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

FUNCTION GENERATION TRACE:

<table>
<thead>
<tr>
<th>Func</th>
<th>Size</th>
<th>Err[%]</th>
<th>Des_LLOC</th>
<th>Ach_LLOC</th>
<th>Err[%]</th>
<th>Phys_lines</th>
<th>Program</th>
<th>Generation</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>61</td>
<td>22.00%</td>
<td>50 D</td>
<td>61 A</td>
<td>22.00%</td>
<td>96 PLOC</td>
<td>0 LLOC/sec</td>
<td>0 PLOC/sec</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>00.00%</td>
<td>100 D</td>
<td>111 A</td>
<td>11.00%</td>
<td>174 PLOC</td>
<td>0 LLOC/sec</td>
<td>0 PLOC/sec</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>59</td>
<td>18.00%</td>
<td>150 D</td>
<td>170 A</td>
<td>13.33%</td>
<td>263 PLOC</td>
<td>17000 LLOC/sec</td>
<td>26300 PLOC/sec</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>44</td>
<td>-12.00%</td>
<td>200 D</td>
<td>214 A</td>
<td>7.00%</td>
<td>337 PLOC</td>
<td>21400 LLOC/sec</td>
<td>33700 PLOC/sec</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>42</td>
<td>-16.00%</td>
<td>250 D</td>
<td>256 A</td>
<td>2.40%</td>
<td>393 PLOC</td>
<td>12800 LLOC/sec</td>
<td>16300 PLOC/sec</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>59</td>
<td>18.00%</td>
<td>300 D</td>
<td>315 A</td>
<td>5.00%</td>
<td>482 PLOC</td>
<td>15750 LLOC/sec</td>
<td>24000 PLOC/sec</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>42</td>
<td>-16.00%</td>
<td>350 D</td>
<td>357 A</td>
<td>2.00%</td>
<td>549 PLOC</td>
<td>17850 LLOC/sec</td>
<td>27450 PLOC/sec</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>59</td>
<td>18.00%</td>
<td>400 D</td>
<td>416 A</td>
<td>4.00%</td>
<td>638 PLOC</td>
<td>13867 LLOC/sec</td>
<td>21267 PLOC/sec</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>48</td>
<td>-4.00%</td>
<td>450 D</td>
<td>464 A</td>
<td>3.11%</td>
<td>714 PLOC</td>
<td>15467 LLOC/sec</td>
<td>23800 PLOC/sec</td>
<td></td>
</tr>
</tbody>
</table>

End of function generation. Press Return to continue ...
## RESULTS:

Generated C++ program is stored in file `BM1default1.cpp`

<table>
<thead>
<tr>
<th>Desired number of logical lines (LLOC)</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Achieved number of logical lines (LLOC)</td>
<td>504</td>
</tr>
<tr>
<td>Program size error</td>
<td>0.80%</td>
</tr>
<tr>
<td>Total number of physical lines of code</td>
<td>753</td>
</tr>
<tr>
<td>Number of physical lines per LLOC</td>
<td>1.49</td>
</tr>
<tr>
<td>Total consumed processor time</td>
<td>0.03 sec</td>
</tr>
<tr>
<td>Average program generation rate</td>
<td>16800 LLOC/sec</td>
</tr>
<tr>
<td>Achieved maximum depth</td>
<td>6</td>
</tr>
<tr>
<td>Achieved maximum breadth</td>
<td>8</td>
</tr>
<tr>
<td>Program name</td>
<td><code>BM1default1.cpp</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control structure</th>
<th>Count</th>
<th>Dim</th>
<th>Desired prob.</th>
<th>Achieved prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] arithmetic</td>
<td>346</td>
<td>0</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[2] if</td>
<td>7</td>
<td>14</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[3] if-else</td>
<td>8</td>
<td>14</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[4] switch</td>
<td>8</td>
<td>14</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[5] while</td>
<td>9</td>
<td>14</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[6] do</td>
<td>8</td>
<td>14</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[7] for</td>
<td>8</td>
<td>14</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
</tbody>
</table>

Average absolute error = 0.00%

### Depth distribution:

<table>
<thead>
<tr>
<th>Depth</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>18.3%</td>
</tr>
<tr>
<td>[1]</td>
<td>17.8%</td>
</tr>
<tr>
<td>[2]</td>
<td>18.5%</td>
</tr>
<tr>
<td>[3]</td>
<td>18.8%</td>
</tr>
<tr>
<td>[4]</td>
<td>9.6%</td>
</tr>
<tr>
<td>[5]</td>
<td>17.0%</td>
</tr>
<tr>
<td>[6]</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Achieved (top) and Desired (bottom) Breadth Distributions:

<table>
<thead>
<tr>
<th>Breadth</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0.0%</td>
</tr>
<tr>
<td>[1]</td>
<td>5.5%</td>
</tr>
<tr>
<td>[2]</td>
<td>5.5%</td>
</tr>
<tr>
<td>[3]</td>
<td>9.6%</td>
</tr>
<tr>
<td>[4]</td>
<td>20.5%</td>
</tr>
<tr>
<td>[5]</td>
<td>39.7%</td>
</tr>
<tr>
<td>[6]</td>
<td>19.2%</td>
</tr>
<tr>
<td>[7]</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Breadth</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0.0%</td>
</tr>
<tr>
<td>[1]</td>
<td>5.0%</td>
</tr>
<tr>
<td>[2]</td>
<td>5.0%</td>
</tr>
<tr>
<td>[3]</td>
<td>10.0%</td>
</tr>
<tr>
<td>[4]</td>
<td>20.0%</td>
</tr>
<tr>
<td>[5]</td>
<td>40.0%</td>
</tr>
<tr>
<td>[6]</td>
<td>20.0%</td>
</tr>
<tr>
<td>[7]</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

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

---
```cpp
#include <iostream>
using namespace std;
#include <time.h>
int IFcnt[14],IFECnt[14],SWEcnt[14],WHILEcnt[14],DOcnt[14],FORcnt[14];
int P1(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[0]%5 )
{
    m -= (e+d+1-d-1-h)%100;
    if( ++IFcnt[0]%2 )
    {
        k += (f*h*i-j*m*c-g-e+n+f*d+e)%100;
    }
    else
    {
        switch( ++SWEcnt[0]%3 )
        {
        case 1:
            while( ++WHILEcnt[0]%5 )
            {
                do
                {
                    k += (a-h*h*b)%100;
                    e += (n*j*h-n+d+a+k+g+h*m-j*c)%100;
                    n += (1+m+f-m+i+e)%100;
                } while( ++DOcnt[0]%5 );
            for( ; ++FORcnt[0]%5 ; )
            {
                e = (i*e+h*a+a-k+d+f-g-m)%100;
                h += (d-i*n-f-l*j)%100;
                i -= (c-f*h*g-l-m)%100;
                n -= (e*n-n-b*e+k-e-d+e-h-g)%100;
                n += (a-j+k+i-c*l*l*i*k-l)%100;
            }
            h -= (i-m-e)%100;
            if( ++IFcnt[0]%10 )
            {
                k -= (n+j)%100;
                c += (j*n-j+d+h-d*k+1*i-h-j+h-n)%100;
                b = (f-k+g*j*m*g+d-g*m-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+k)%100;
        }
g = (m-d-f-g*m*1-c+e+1+i)%100;
i += (a-l+m*f*1+g+i+i+h+k)%100;
m += (g*d+i*e-k*d*l*j-h-m)%100;
n = (e+c-a+h+e+n+f*e+n)%100;
m -= (m+g*i+f*i+e-j)%100;
```
End of generated C++ program

500 LLOC

BM1default1.cpp
C++ Source file
20 KB

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

return 0;
Output Parameter File Name = bmloutpar.txt

Default: Uniform distribution of control structures
Generation of a single program ./BMLdefault1.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:

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
## FUNCTION GENERATION TRACE:

<table>
<thead>
<tr>
<th>Func</th>
<th>Size</th>
<th>Err [%]</th>
<th>Des_LLOC</th>
<th>Ach_LLOC</th>
<th>Err [%]</th>
<th>Phys_lines</th>
<th>Program Generation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>496</td>
<td>-0.80%</td>
<td>500 D</td>
<td>496 A</td>
<td>-0.80%</td>
<td>775 PLOC</td>
<td>49600 LLOC/sec</td>
</tr>
<tr>
<td>2</td>
<td>506</td>
<td>1.20%</td>
<td>1000 D</td>
<td>1002 A</td>
<td>0.20%</td>
<td>1560 PLOC</td>
<td>50100 LLOC/sec</td>
</tr>
<tr>
<td>3</td>
<td>498</td>
<td>-0.40%</td>
<td>1500 D</td>
<td>1500 A</td>
<td>0.00%</td>
<td>2322 PLOC</td>
<td>50000 LLOC/sec</td>
</tr>
<tr>
<td>4</td>
<td>508</td>
<td>1.60%</td>
<td>2000 D</td>
<td>2008 A</td>
<td>-0.40%</td>
<td>3109 PLOC</td>
<td>39360 LLOC/sec</td>
</tr>
<tr>
<td>5</td>
<td>510</td>
<td>2.00%</td>
<td>2500 D</td>
<td>2518 A</td>
<td>0.72%</td>
<td>3903 PLOC</td>
<td>50360 LLOC/sec</td>
</tr>
<tr>
<td>6</td>
<td>480</td>
<td>-4.00%</td>
<td>3000 D</td>
<td>2998 A</td>
<td>-0.07%</td>
<td>4641 PLOC</td>
<td>59960 LLOC/sec</td>
</tr>
<tr>
<td>7</td>
<td>526</td>
<td>5.20%</td>
<td>3500 D</td>
<td>3524 A</td>
<td>0.69%</td>
<td>5460 PLOC</td>
<td>58733 LLOC/sec</td>
</tr>
<tr>
<td>8</td>
<td>492</td>
<td>-1.60%</td>
<td>4000 D</td>
<td>4016 A</td>
<td>0.40%</td>
<td>6223 PLOC</td>
<td>57371 LLOC/sec</td>
</tr>
<tr>
<td>9</td>
<td>491</td>
<td>-1.80%</td>
<td>4500 D</td>
<td>4507 A</td>
<td>0.16%</td>
<td>6974 PLOC</td>
<td>56338 LLOC/sec</td>
</tr>
<tr>
<td>10</td>
<td>504</td>
<td>0.80%</td>
<td>500 D</td>
<td>511 A</td>
<td>0.22%</td>
<td>7756 PLOC</td>
<td>55678 LLOC/sec</td>
</tr>
<tr>
<td>11</td>
<td>501</td>
<td>0.20%</td>
<td>5500 D</td>
<td>5512 A</td>
<td>0.22%</td>
<td>8532 PLOC</td>
<td>55120 LLOC/sec</td>
</tr>
<tr>
<td>12</td>
<td>492</td>
<td>-1.60%</td>
<td>6000 D</td>
<td>6004 A</td>
<td>0.07%</td>
<td>9298 PLOC</td>
<td>54582 LLOC/sec</td>
</tr>
<tr>
<td>13</td>
<td>506</td>
<td>1.20%</td>
<td>6500 D</td>
<td>6510 A</td>
<td>0.15%</td>
<td>10084 PLOC</td>
<td>54250 LLOC/sec</td>
</tr>
<tr>
<td>14</td>
<td>504</td>
<td>0.80%</td>
<td>7000 D</td>
<td>7014 A</td>
<td>0.20%</td>
<td>10863 PLOC</td>
<td>53954 LLOC/sec</td>
</tr>
<tr>
<td>15</td>
<td>504</td>
<td>0.80%</td>
<td>7500 D</td>
<td>7518 A</td>
<td>0.24%</td>
<td>11644 PLOC</td>
<td>53700 LLOC/sec</td>
</tr>
<tr>
<td>16</td>
<td>496</td>
<td>-0.80%</td>
<td>8000 D</td>
<td>8014 A</td>
<td>0.17%</td>
<td>12414 PLOC</td>
<td>53073 LLOC/sec</td>
</tr>
<tr>
<td>17</td>
<td>484</td>
<td>-3.20%</td>
<td>8500 D</td>
<td>8498 A</td>
<td>-0.02%</td>
<td>13150 PLOC</td>
<td>52783 LLOC/sec</td>
</tr>
<tr>
<td>18</td>
<td>527</td>
<td>5.40%</td>
<td>9000 D</td>
<td>9025 A</td>
<td>0.28%</td>
<td>13972 PLOC</td>
<td>52778 LLOC/sec</td>
</tr>
<tr>
<td>19</td>
<td>489</td>
<td>-2.20%</td>
<td>9500 D</td>
<td>9514 A</td>
<td>0.15%</td>
<td>14729 PLOC</td>
<td>55637 LLOC/sec</td>
</tr>
<tr>
<td>20</td>
<td>505</td>
<td>1.00%</td>
<td>10000 D</td>
<td>10019 A</td>
<td>0.19%</td>
<td>15513 PLOC</td>
<td>55354 LLOC/sec</td>
</tr>
<tr>
<td>21</td>
<td>478</td>
<td>-4.40%</td>
<td>10500 D</td>
<td>10497 A</td>
<td>-0.03%</td>
<td>16243 PLOC</td>
<td>54958 LLOC/sec</td>
</tr>
<tr>
<td>22</td>
<td>522</td>
<td>4.40%</td>
<td>11000 D</td>
<td>11019 A</td>
<td>0.17%</td>
<td>17060 PLOC</td>
<td>54821 LLOC/sec</td>
</tr>
<tr>
<td>23</td>
<td>479</td>
<td>-4.20%</td>
<td>11500 D</td>
<td>11497 A</td>
<td>-0.02%</td>
<td>17789 PLOC</td>
<td>54493 LLOC/sec</td>
</tr>
<tr>
<td>24</td>
<td>534</td>
<td>6.80%</td>
<td>12000 D</td>
<td>12032 A</td>
<td>0.27%</td>
<td>18628 PLOC</td>
<td>54443 LLOC/sec</td>
</tr>
<tr>
<td>25</td>
<td>459</td>
<td>-8.20%</td>
<td>12500 D</td>
<td>12491 A</td>
<td>-0.07%</td>
<td>19335 PLOC</td>
<td>54074 LLOC/sec</td>
</tr>
<tr>
<td>26</td>
<td>506</td>
<td>1.20%</td>
<td>13000 D</td>
<td>12997 A</td>
<td>-0.02%</td>
<td>20117 PLOC</td>
<td>53929 LLOC/sec</td>
</tr>
<tr>
<td>27</td>
<td>521</td>
<td>4.20%</td>
<td>13500 D</td>
<td>13518 A</td>
<td>0.13%</td>
<td>20917 PLOC</td>
<td>53857 LLOC/sec</td>
</tr>
<tr>
<td>28</td>
<td>510</td>
<td>2.00%</td>
<td>14000 D</td>
<td>14028 A</td>
<td>0.20%</td>
<td>21714 PLOC</td>
<td>53747 LLOC/sec</td>
</tr>
<tr>
<td>29</td>
<td>486</td>
<td>-2.80%</td>
<td>14500 D</td>
<td>14514 A</td>
<td>0.10%</td>
<td>22458 PLOC</td>
<td>53557 LLOC/sec</td>
</tr>
<tr>
<td>30</td>
<td>494</td>
<td>-1.20%</td>
<td>15000 D</td>
<td>15008 A</td>
<td>0.05%</td>
<td>23221 PLOC</td>
<td>53409 LLOC/sec</td>
</tr>
<tr>
<td>31</td>
<td>494</td>
<td>1.20%</td>
<td>15500 D</td>
<td>15502 A</td>
<td>0.01%</td>
<td>23981 PLOC</td>
<td>53271 LLOC/sec</td>
</tr>
<tr>
<td>32</td>
<td>521</td>
<td>4.20%</td>
<td>16000 D</td>
<td>16023 A</td>
<td>0.14%</td>
<td>24795 PLOC</td>
<td>53233 LLOC/sec</td>
</tr>
<tr>
<td>33</td>
<td>475</td>
<td>-5.00%</td>
<td>16500 D</td>
<td>16498 A</td>
<td>-0.01%</td>
<td>25522 PLOC</td>
<td>53048 LLOC/sec</td>
</tr>
<tr>
<td>34</td>
<td>502</td>
<td>0.40%</td>
<td>17000 D</td>
<td>17000 A</td>
<td>0.00%</td>
<td>26301 PLOC</td>
<td>52960 LLOC/sec</td>
</tr>
<tr>
<td>35</td>
<td>512</td>
<td>2.40%</td>
<td>17500 D</td>
<td>17512 A</td>
<td>0.07%</td>
<td>27091 PLOC</td>
<td>52906 LLOC/sec</td>
</tr>
<tr>
<td>36</td>
<td>499</td>
<td>-0.20%</td>
<td>18000 D</td>
<td>18011 A</td>
<td>0.06%</td>
<td>27861 PLOC</td>
<td>52818 LLOC/sec</td>
</tr>
<tr>
<td>37</td>
<td>501</td>
<td>0.20%</td>
<td>18500 D</td>
<td>18512 A</td>
<td>0.06%</td>
<td>28636 PLOC</td>
<td>52741 LLOC/sec</td>
</tr>
<tr>
<td>38</td>
<td>520</td>
<td>4.00%</td>
<td>19000 D</td>
<td>19032 A</td>
<td>0.17%</td>
<td>29449 PLOC</td>
<td>52720 LLOC/sec</td>
</tr>
<tr>
<td>39</td>
<td>469</td>
<td>-6.20%</td>
<td>19500 D</td>
<td>19501 A</td>
<td>0.01%</td>
<td>30166 PLOC</td>
<td>54019 LLOC/sec</td>
</tr>
</tbody>
</table>

End of function generation.
### 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`

<table>
<thead>
<tr>
<th>Control structure</th>
<th>Count</th>
<th>Dim</th>
<th>Desired prob.</th>
<th>Achieved prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] arithmetic</td>
<td>14825</td>
<td>0</td>
<td>3.57%</td>
<td>3.57%</td>
</tr>
<tr>
<td>[3] if-else</td>
<td>297</td>
<td>428</td>
<td>10.71%</td>
<td>10.70%</td>
</tr>
<tr>
<td>[4] switch</td>
<td>397</td>
<td>571</td>
<td>14.29%</td>
<td>14.30%</td>
</tr>
<tr>
<td>[5] while</td>
<td>496</td>
<td>714</td>
<td>17.86%</td>
<td>17.87%</td>
</tr>
<tr>
<td>[6] do</td>
<td>595</td>
<td>857</td>
<td>21.43%</td>
<td>21.43%</td>
</tr>
<tr>
<td>[7] for</td>
<td>694</td>
<td>1000</td>
<td>25.00%</td>
<td>25.00%</td>
</tr>
</tbody>
</table>

Average absolute error = 0.01%

Depth distribution:

1.0% 1.7% 1.7% 2.8% 13.7% 78.4% 0.0%

Achieved (top) and Desired (bottom) Breadth Distributions:

0.0% 5.0% 5.0% 10.0% 20.0% 40.0% 20.0% 0.0%

0.0% 5.0% 5.0% 10.0% 20.0% 40.0% 20.0% 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++) IF__cnt[I] = 0;
    for(I=0; I<571; I++) SWcnt[I] = 0;
    for(I=0; I<274; I++) WHILEcnt[I] = 0;
    for(I=0; I<852; 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
    {
        // Code...
    }

    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

    © 2010 by Jozo Dujmović
#include <iostream>
using namespace std;

#include <time.h>

int IFcnt[295], 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( ++IFcnt[0] % 2 )
                        {
                            n = (j - a - n + b * 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
                        {
                            q = (q - f) % 100;
                            l = (m + b) % 100;
                            f = (k + h - k - c - k + c + c + c + c * n + l - m - e) % 100;
                        }
                    }
                }
            }
        }
    }
}

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
<table>
<thead>
<tr>
<th>LLOC</th>
<th>1,000,000 LLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOC</td>
<td>Press Return to continue ...</td>
</tr>
</tbody>
</table>
RESULTS:

Generated C++ program is stored in file `BM1default1.cpp`
Desired number of logical lines \(|\text{LLOC}|\) = 1,000,000
Achieved number of logical lines \(|\text{LLOC}|\) = 1,000,041
Program size error = 0.00%
Total number of physical lines of code = 1,611,623
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`

<table>
<thead>
<tr>
<th>Control structure</th>
<th>Count</th>
<th>Dim</th>
<th>Desired prob.</th>
<th>Achieved prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] arithmetic</td>
<td>772240</td>
<td>0</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[2] if</td>
<td>21732</td>
<td>28571</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[3] if-else</td>
<td>21733</td>
<td>28571</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[4] switch</td>
<td>21733</td>
<td>28571</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[5] while</td>
<td>21733</td>
<td>28571</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[6] do</td>
<td>21732</td>
<td>28571</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
<tr>
<td>[7] for</td>
<td>21732</td>
<td>28571</td>
<td>14.29%</td>
<td>14.29%</td>
</tr>
</tbody>
</table>

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 \(<\text{top}>\) and Desired \(<\text{bottom}>\) Breadth Distributions:

1,000,000 LLOC

BM1default1.cpp

C++ Source file
51,106 KB

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 library

CLIENT (remote or local)

REQUEST

BENCHMARK GENERATOR

RESULT

Generated benchmark series or suites

B1 B2 Bn

Client benchmark modules
Kernel Naming and Classification

\[ \text{L} \quad \text{A} \quad \text{G} \quad \text{S} \quad \# \quad \# \]

- **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**
   1.3 Numerical procedural kernels
      1.3.0 Miscellaneous
      1.3.1 Scalar floating-point arithmetic
      1.3.2 Library and special functions
      1.3.3 Arrays
      1.3.4 Polynomials
      1.3.5 Matrices
      1.3.6 Integrals and differential equations
      1.3.7 Recursive numerical problems
      1.3.8 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
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)

<table>
<thead>
<tr>
<th>4</th>
<th>SYSTEM KERNELS</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>Processes</td>
</tr>
<tr>
<td>410</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>411</td>
<td>Process create and delete</td>
</tr>
<tr>
<td>412</td>
<td>Multicore</td>
</tr>
<tr>
<td>42</td>
<td>Threads</td>
</tr>
<tr>
<td>420</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>421</td>
<td>Thread create and delete</td>
</tr>
<tr>
<td>422</td>
<td>Hyperthreaded</td>
</tr>
<tr>
<td>43</td>
<td>Signals and alarms</td>
</tr>
<tr>
<td>430</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>431</td>
<td>Signals</td>
</tr>
<tr>
<td>432</td>
<td>Alarms</td>
</tr>
</tbody>
</table>
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

```c
{ // 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)
```

\( \text{TIME} = O(\text{SEC}) \)
Benchmark Terminator Function

```c
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
             << " Run time [total seconds] = " << RunTime
             << " End 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

<table>
<thead>
<tr>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select a desired BENCHMARK_PROGRAM_SIZE</td>
</tr>
<tr>
<td>Select a desired benchmark program structure</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>KERNEL SELECTION</strong>: Select the most appropriate kernel</td>
</tr>
<tr>
<td>using either random or deterministic selection technique</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>PROGRAM EXPANSION</strong>: Insert the selected kernel in the</td>
</tr>
<tr>
<td>desired benchmark program structure</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>PROGRAM SIZE MEASUREMENT</strong>:</td>
</tr>
<tr>
<td>SIZE = number of lines of code in the expanded program</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>do while (SIZE &lt; BENCHMARK_PROGRAM_SIZE) ;</td>
</tr>
</tbody>
</table>
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

\[
t = ar + b, \quad a = \text{const}, \quad b = \text{const.}
\]

\[
t_1 = ar_1 + b, \quad t_2 = ar_2 + b, \quad T = aR + b
\]

\[
t_2 - t_1 = a(r_2 - r_1), \quad T - t_1 = a(R - r_1),
\]

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

\[
R = r_1 + (T - t_1)(r_2 - r_1)/(t_2 - t_1)
\]

R should be greater than 100 to provide accurate approximation of T
BM2 System Overview
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

```c
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 }
    return ERROR;  // Randomly selected kernel
                     // Kernel error code
}

..............................

int Fn(void)
{
    { K41 }
    return ERROR;  // Randomly selected kernel
                     // 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]

\[ n = \text{total number of available kernels} \]

\[ K_1, K_2, \ldots, K_n = \text{kernels} \]

\[ L_1, L_2, \ldots, L_n = \text{kernel sizes [LOC]} \]

\[ f_1, f_2, \ldots, f_n = \text{kernel frequencies in a given program} \]

\[ f_1 + f_2 + \ldots + f_n = F = \text{total number of kernels} \]

\[ f_1L_1 + f_2L_2 + \ldots + f_nL_n = \text{total benchmark size} \]

\[ L = \text{desired size of benchmark program [LOC]} \]

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

\[ p_i = \frac{f_i}{F}, \quad i = 1, \ldots, n : \text{achieved kernel probabilities} \]
Kernel Selection Problem [2/11]

INPUTS:

\( P_1, P_2, \ldots, P_n = \) desired kernel probabilities
\( L = \) desired benchmark size

PROBLEM:

Find optimum kernel frequencies \( f_1^*, f_2^*, \ldots, f_n^* \)
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

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

with the following condition:

\[ f_1 L_1 + f_2 L_2 + \ldots + f_n L_n \approx L \]
Kernel Selection Problem [4/11]

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

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

and

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

\[ C(f_1, f_2, \ldots, f_n) = \]

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

\[ 0 < W < 1, \quad 1 \leq r \leq +\infty \] (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.

\[
\begin{align*}
\text{do} & \{ \\
& \quad r = \text{(random integer from 1 to n distributed according to any desired kernel distribution)} ; \\
& \quad \text{Insert kernel } K_r \text{ in benchmark program;} \\
& \quad \text{size} = \text{(number of lines of code after the addition of kernel } K_r \text{);} \\
& \text{\} while (size < L); \\
\end{align*}
\]
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.

\[
do{
  r = \text{(integer from 1 to } n \text{ selected by DOS according to desired kernel distribution)};
  \text{Insert kernel } K_r \text{ in benchmark program};
  \text{size } = \text{(number of lines of code after the addition of kernel } K_r \text{)};
}
\text{while (size } < L\text{)};
\]
Kernel Selection Problem [10/11]

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

\[ e(j) = \left| \frac{f_j + 1}{f_1 + f_2 + \ldots + f_n + 1} - P_j \right| + \]

\[ + \sum_{i=1}^{n} \left| \frac{f_i}{f_1 + f_2 + \ldots + f_n + 1} - P_i \right|, \quad 1 \leq j \leq n \]

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

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.
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\n
Available options:

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

Enter your choice: 1
<table>
<thead>
<tr>
<th>#</th>
<th>Code</th>
<th>LOC</th>
<th>TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C11301</td>
<td>20</td>
<td>C11301: Counting words in a string</td>
</tr>
<tr>
<td>2</td>
<td>C11601</td>
<td>32</td>
<td>C11601: Class with Bubble Sort function</td>
</tr>
<tr>
<td>3</td>
<td>C11602</td>
<td>30</td>
<td>C11602: Class with QuickSort function</td>
</tr>
<tr>
<td>4</td>
<td>C11603</td>
<td>31</td>
<td>C11603: Class with Select Sort function</td>
</tr>
<tr>
<td>5</td>
<td>C11604</td>
<td>22</td>
<td>C11604: Embedded Binary Search</td>
</tr>
<tr>
<td>6</td>
<td>C11605</td>
<td>25</td>
<td>C11605: Embedded Bubble Sort</td>
</tr>
<tr>
<td>7</td>
<td>C11606</td>
<td>23</td>
<td>C11606: Embedded Select Sort</td>
</tr>
<tr>
<td>8</td>
<td>C11607</td>
<td>27</td>
<td>C11607: Class with an iterative binary search function</td>
</tr>
<tr>
<td>9</td>
<td>C11608</td>
<td>40</td>
<td>C11608: Embedded merge of sorted arrays</td>
</tr>
<tr>
<td>10</td>
<td>C11609</td>
<td>37</td>
<td>C11609: Class with a merge function</td>
</tr>
<tr>
<td>11</td>
<td>C11610</td>
<td>21</td>
<td>C11610: Linear search</td>
</tr>
<tr>
<td>12</td>
<td>C11701</td>
<td>24</td>
<td>C11701: Class with a recursive binary search</td>
</tr>
<tr>
<td>13</td>
<td>C12101</td>
<td>29</td>
<td>C12101: Basic arithmetic with integer scalars</td>
</tr>
<tr>
<td>14</td>
<td>C12301</td>
<td>30</td>
<td>C12301: Graph centroid (integer distances)</td>
</tr>
<tr>
<td>15</td>
<td>C12401</td>
<td>23</td>
<td>C12401: Divide and test prime number generator</td>
</tr>
<tr>
<td>16</td>
<td>C12601</td>
<td>29</td>
<td>C12602: Erathostenes sieve prime number generator</td>
</tr>
<tr>
<td>17</td>
<td>C13101</td>
<td>27</td>
<td>C13101: Basic arithmetic with scalars of type double</td>
</tr>
<tr>
<td>18</td>
<td>C13102</td>
<td>29</td>
<td>C13102: Basic arithmetic with scalars of type float</td>
</tr>
<tr>
<td>19</td>
<td>C13501</td>
<td>44</td>
<td>C13501: Class with a linear equations solver</td>
</tr>
<tr>
<td>20</td>
<td>C13502</td>
<td>117</td>
<td>C13502: Class with matrix inversion</td>
</tr>
<tr>
<td>21</td>
<td>C13503</td>
<td>30</td>
<td>C13503: Graph centroid (float distances)</td>
</tr>
<tr>
<td>22</td>
<td>C13504</td>
<td>30</td>
<td>C13504: Graph centroid (double distances)</td>
</tr>
<tr>
<td>23</td>
<td>C13601</td>
<td>37</td>
<td>C13601: Class with differential equations (Runge-Kutta)</td>
</tr>
<tr>
<td>24</td>
<td>C13602</td>
<td>37</td>
<td>C13602: Numerical computation of integrals (trapezoids)</td>
</tr>
<tr>
<td>25</td>
<td>C14101</td>
<td>35</td>
<td>C14101: Class with an array of 4 scalar float values per component</td>
</tr>
<tr>
<td>26</td>
<td>C14102</td>
<td>35</td>
<td>C14102: Class with an array of 4 scalar double values per component</td>
</tr>
<tr>
<td>27</td>
<td>C14103</td>
<td>30</td>
<td>C14103: An array of objects (an array of 12 type float components per object)</td>
</tr>
<tr>
<td>28</td>
<td>C14104</td>
<td>30</td>
<td>C14104: An array of objects (an array of 12 type double components per object)</td>
</tr>
<tr>
<td>29</td>
<td>C21100</td>
<td>36</td>
<td>C21100: Uniform memory access to 1-5 localities (static)</td>
</tr>
<tr>
<td>30</td>
<td>C21101</td>
<td>36</td>
<td>C21101: Uniform memory access to 1 locality (static)</td>
</tr>
<tr>
<td>31</td>
<td>C21102</td>
<td>36</td>
<td>C21102: Uniform memory access to 2 localities (static)</td>
</tr>
<tr>
<td>32</td>
<td>C21103</td>
<td>36</td>
<td>C21103: Uniform memory access to 3 localities (static)</td>
</tr>
<tr>
<td>33</td>
<td>C21104</td>
<td>36</td>
<td>C21104: Uniform memory access to 4 localities (static)</td>
</tr>
<tr>
<td>34</td>
<td>C21105</td>
<td>36</td>
<td>C21105: Uniform memory access to 5 localities (static)</td>
</tr>
<tr>
<td>35</td>
<td>C22100</td>
<td>40</td>
<td>C22100: Uniform memory access to 1-5 localities (dynamic)</td>
</tr>
<tr>
<td>36</td>
<td>C22101</td>
<td>40</td>
<td>C22101: Uniform memory access to 1 locality (dynamic)</td>
</tr>
<tr>
<td>37</td>
<td>C22102</td>
<td>40</td>
<td>C22102: Uniform memory access to 2 localities (dynamic)</td>
</tr>
<tr>
<td>38</td>
<td>C22103</td>
<td>40</td>
<td>C22103: Uniform memory access to 3 localities (dynamic)</td>
</tr>
<tr>
<td>39</td>
<td>C22104</td>
<td>40</td>
<td>C22104: Uniform memory access to 4 localities (dynamic)</td>
</tr>
<tr>
<td>40</td>
<td>C22105</td>
<td>40</td>
<td>C22105: Uniform memory access to 5 localities (dynamic)</td>
</tr>
</tbody>
</table>

Enter kernel # to see the kernel (0 to activate the BM2 generator, -1 to quit): 12
```c
// C11701
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 ...
```
**Desired Kernel Distribution**

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

**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

Generated program(s)

<table>
<thead>
<tr>
<th>#</th>
<th>Desired size</th>
<th>Achieved size</th>
<th>Distribution error</th>
<th>Program name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2K</td>
<td>2K</td>
<td>74L</td>
<td>95.00%</td>
</tr>
<tr>
<td>2</td>
<td>4K</td>
<td>4K</td>
<td>147L</td>
<td>90.00%</td>
</tr>
<tr>
<td>3</td>
<td>6K</td>
<td>6K</td>
<td>198L</td>
<td>85.00%</td>
</tr>
<tr>
<td>4</td>
<td>8K</td>
<td>8K</td>
<td>252L</td>
<td>80.00%</td>
</tr>
<tr>
<td>5</td>
<td>10K</td>
<td>10K</td>
<td>333L</td>
<td>75.00%</td>
</tr>
<tr>
<td>6</td>
<td>12K</td>
<td>12K</td>
<td>382L</td>
<td>70.00%</td>
</tr>
<tr>
<td>7</td>
<td>14K</td>
<td>14K</td>
<td>445L</td>
<td>65.00%</td>
</tr>
<tr>
<td>8</td>
<td>16K</td>
<td>16K</td>
<td>495L</td>
<td>60.00%</td>
</tr>
<tr>
<td>9</td>
<td>18K</td>
<td>18K</td>
<td>555L</td>
<td>55.00%</td>
</tr>
<tr>
<td>10</td>
<td>20K</td>
<td>20K</td>
<td>720L</td>
<td>50.00%</td>
</tr>
</tbody>
</table>

Press any key to continue...
#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>

// Global variables
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++)
  {
    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\PROGRAMS\BenchMaker\BenchMaker 2\PROJECTS\jozo\FEB11\Release\SEQF10K.exe

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\PROGRAMS\BenchMaker\BenchMaker 2\PROJECTS\jozo\FEB11\Release\SEQF10K.exe

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.

Please refer to the links below for more information.
BenchMaker User Interface (3/9)

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
Create a new project by filling out the fields:

Unique Name: 
Language: C++ 
Type: CompileTo(BM1) 
Description: 

Create | Cancel

About BenchMaker | Download Papers | See Legal Stuff | Contact Us | Information for Guest Users | Contribute | Research with BenchMaker
BenchMaker User Interface (5/9)
BenchMaker User Interface (6/9)

Logged in as jozo

List Projects  Create Project  Update Project Name or Description

Your existing projects

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Language</th>
<th>Type</th>
<th>Description</th>
<th>Modified On</th>
<th>Compiled On</th>
<th>Sent On</th>
</tr>
</thead>
<tbody>
<tr>
<td>testBM1</td>
<td>C++</td>
<td>BM1</td>
<td>testing benchmaker 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testBM2</td>
<td>C++</td>
<td>BM2</td>
<td>testing benchmaker 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Delete Selected Project(s)  Duplicate Selected Project(s)

---

About BenchMaker | Download Papers | See Legal Stuff | Contact Us | Information for Guest Users | Contribute | Research with BenchMaker
BenchMaker User Interface (9/9)

Logged in as jozo

<table>
<thead>
<tr>
<th>List Projects</th>
<th>Create Project</th>
<th>Update Project Name or Description</th>
<th>Logout</th>
</tr>
</thead>
</table>

1. PROCESSOR PERFORMANCE  55.0 52%
2. MEMORY ACCESS(PAGING & CACHING)  32.0 30%
3. DISK & PERIPHERALS ACCESS  16.0 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
0 120 0 500 0 200 0 500 0

Save Project  Delete Project  Generate Benchmark(s)  Deliver by email
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++

\[ C = 0.0014 L + 3.3544 \]

Cygwin g++

\[ C = 0.004 L + 2.4595 \]

Compile Time (seconds)

Lines of Code L

6 sec

10 sec
$C = 3.28 + 9.58 \cdot 10^{-6} L^{2.062}$
Comparison of Object Program Sizes

**Cygwin g++**

- Object Program Size (bytes) vs. Lines of Code
- Linear Regression: $M_{obj} = 77.523L + 2577.3$
- Estimated Program Size: 154 KB

**Visual C++**

- Object Program Size (bytes) vs. Lines of Code
- Linear Regression: $M_{obj} = 58.291L + 3327.6$
- Estimated Program Size: 117 KB
Memory Consumption (M) as a Function of Program Size (L)

\[ M = m_0 + m_1 L \]
Object Program Size vs. Executable Program Size

Visual C++

Object Program Size vs. Executable Program Size

Visual C++

Object Program Size (bytes)

Executable Size (bytes)

Lines of Code $L$

$M_{obj} = 58.291 L + 3327.6$

$M = 46.39 L + 57181$

$146$ KB
Nonlinear Phenomena – Intel C++ Compiler

Object Program Size (bytes)

Lines of Code $L$

$$M_{obj} = 47.694L + 13218$$

Executable Size (bytes)

Lines of Code $L$

$$M = 31.137L + 55582$$
Nonlinear Phenomena – Metrowerks CodeWarrior

Object Program Size (bytes)

Mobj = 81.573 L + 166464

Executable Program Size (bytes)

M = 54.553 L + 191915
**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

<table>
<thead>
<tr>
<th>Processor Type</th>
<th>System Type</th>
<th>Mean Relative Execution Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC6-speed</td>
<td>Intel Pentium II</td>
<td>1.33</td>
</tr>
<tr>
<td>INTC-speed</td>
<td></td>
<td>1.00</td>
</tr>
<tr>
<td>GPP-speed</td>
<td></td>
<td>1.84</td>
</tr>
<tr>
<td>CW53-speed</td>
<td></td>
<td>1.36</td>
</tr>
<tr>
<td>BC55-speed</td>
<td></td>
<td>2.27</td>
</tr>
<tr>
<td>VC6-default</td>
<td></td>
<td>3.17</td>
</tr>
<tr>
<td>GPP-default</td>
<td></td>
<td>3.71</td>
</tr>
<tr>
<td>CW53-default</td>
<td></td>
<td>2.80</td>
</tr>
<tr>
<td>BC55-default</td>
<td></td>
<td>2.44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor Type</th>
<th>System Type</th>
<th>Mean Relative Execution Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC6-speed</td>
<td>AMD K6-2 based System</td>
<td>1.08</td>
</tr>
<tr>
<td>INTC-speed</td>
<td></td>
<td>1.05</td>
</tr>
<tr>
<td>GPP-speed</td>
<td></td>
<td>1.25</td>
</tr>
<tr>
<td>CW53-speed</td>
<td></td>
<td>1.00</td>
</tr>
<tr>
<td>BC55-speed</td>
<td></td>
<td>1.45</td>
</tr>
<tr>
<td>VC6-default</td>
<td>AMD K6-2 based System</td>
<td>2.02</td>
</tr>
<tr>
<td>GPP-default</td>
<td></td>
<td>2.06</td>
</tr>
<tr>
<td>CW53-default</td>
<td></td>
<td>1.54</td>
</tr>
<tr>
<td>BC55-default</td>
<td></td>
<td>1.46</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor Type</th>
<th>System Type</th>
<th>Mean Relative Execution Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC6-speed</td>
<td>Cyrix 6x86MX based System</td>
<td>1.02</td>
</tr>
<tr>
<td>INTC-speed</td>
<td></td>
<td>1.00</td>
</tr>
<tr>
<td>GPP-speed</td>
<td></td>
<td>1.34</td>
</tr>
<tr>
<td>CW53-speed</td>
<td></td>
<td>1.18</td>
</tr>
<tr>
<td>BC55-speed</td>
<td></td>
<td>1.51</td>
</tr>
<tr>
<td>VC6-default</td>
<td>Cyrix 6x86MX based System</td>
<td>2.34</td>
</tr>
<tr>
<td>GPP-default</td>
<td></td>
<td>2.30</td>
</tr>
<tr>
<td>CW53-default</td>
<td></td>
<td>1.98</td>
</tr>
<tr>
<td>BC55-default</td>
<td></td>
<td>1.62</td>
</tr>
</tbody>
</table>
Performance ranking of compilers using a Pentium based system

Execution time ratio:

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

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):

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

![Graph showing relative rates for AMD Athlon 1.0GHz, 128MB RAM]

- ICD: 98.69%
- CGO: 87.09%
- LGO: 76.17%
- VCO: 71.14%
- BCD: 41.95%
- BCO: 38.12%
- LGD: 32.58%
- VCD: 31.29%
- CGD: 31.29%
Using Calibration for Performance Comparison (3/3)

Intel Centrino 1.4GHz, 512MB RAM

<table>
<thead>
<tr>
<th></th>
<th>Relative Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICD</td>
<td>100.00%</td>
</tr>
<tr>
<td>ICO</td>
<td>99.81%</td>
</tr>
<tr>
<td>VCO</td>
<td>60.87%</td>
</tr>
<tr>
<td>CGO</td>
<td>60.45%</td>
</tr>
<tr>
<td>LGO</td>
<td>53.51%</td>
</tr>
<tr>
<td>BCO</td>
<td>33.26%</td>
</tr>
<tr>
<td>BCD</td>
<td>32.94%</td>
</tr>
<tr>
<td>VCD</td>
<td>26.11%</td>
</tr>
<tr>
<td>LGD</td>
<td>25.62%</td>
</tr>
<tr>
<td>CGD</td>
<td>23.89%</td>
</tr>
</tbody>
</table>
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

Other publications


Thanks!
Questions?