Automatic Generation of Benchmark and Test Workloads

Jozo Dujmović
San Francisco State University
1600 Holloway Ave.
San Francisco, CA 94132
1-415-338-2207

jozo@sfsu.edu

ABSTRACT
In this tutorial, we describe techniques for automatic generation of benchmark and test workloads. Generated programs have adjustable parameters that are used to select the program size and structure, as well as the relative frequencies of basic operations (or program modules) that characterize the workload.

Categories and Subject Descriptors
D.2.8 **Metrics**

General Terms
Measurement, Performance

Keywords
Benchmarking, metrics, program generators

1. INTRODUCTION
A traditional goal of benchmarking is to determine (or predict) the performance of a given computer system running a specific real workload. There are many other benchmarking/testing problems, such as benchmarking selected hardware units, language processors, database systems, operating systems, and application software. The most frequent reason for benchmarking is the process of performance evaluation organized in the context of comparison and selection of complex computer systems.

Exponential growth of computer performance is the fundamental property of computer industry. If q denotes a performance indicator (e.g. the CPU transistor count, memory capacity, disk capacity, pixels per Dollar, processor speed, or performance/price ratio) and t denotes time, then $q(t) = q_0 2^{(t-t_0)/T}$ (Moore's law). Here T denotes the "performance doubling time": if the initial performance is $q(t_0) = q_0$ and $t - t_0 = T$ then $q = 2q_0$. In other words, performance doubles after each T time units. For example, according to [16], the CPU transistor count in 1971 was 2300, and in 2003 it was approximately 10^8 yielding

$$T = \frac{(t-t_0)\log 2}{\log q(t) - \log q_0} = \frac{(2003-1971)\log 2}{\log 100{,}000{,}000 - \log 2300} = 2.08 \text{ years}$$

Consequently, the chip density increases two times every two years. According to [8] the memory capacity doubles each 18 months, and since 1985 the performance/price ratio doubles every 12 months. The clock speed, and power dissipation (without expensive cooling) reached their saturation points (around 5 GHz and 100 W respectively) approximately in 2005 [11][1]; however, the exponential growth is expected to continue with the number of processor cores.

Industrial benchmark programs, such as SPEC benchmarks [15] [4], are used in an environment that is permanently and rapidly changing. Under such conditions industrial benchmarks should strictly follow the exponential growth of computer performance, and this cannot be achieved using the current benchmark suites that consist of natural workloads that are updated once in several years. For example, SPEC updates its CPU benchmark suites approximately once in three to six years (1989, 1992, 1995, 2000, 2006). However, in three years performance/price can increase 8 times! It is perfectly clear that as long as benchmarking is based on selecting natural workloads and keeping them unchanged for several years there will always be a substantial gap between the current state of technology and the current state of industrial benchmarks. There is a permanent pressure to replace obsolete benchmarks, to cover a spectrum of applications (e.g. SPEC CPU 2006 includes 29 programs!), develop new benchmark suites, and permanently supply new measurements for almost all commercially available computers. It is obvious that this approach to benchmarking must be expensive.

An alternative approach to benchmarking can be based on automatic generators of benchmark programs. Main advantages of this approach are the high flexibility of workload characteristics, a fast response to changes of available hardware/software resources, easy customizing, and the low cost.

There are two basic prerequisites for this approach to benchmarking. The first prerequisite is the development of theoretical background for quantitative analysis, characterization and design of synthetic workloads [2][3]. The second prerequisite includes programming techniques and tools for implementation of benchmark generators [5][6][7]. Both prerequisites need more research and development efforts and this tutorial is prepared for those who want to move in that direction. Our main goal is to present basic concepts of design and implementation of benchmark generators.

2. WORKLOAD DECOMPOSITION AND AGGREGATION

2.1 A Hierarchy of Benchmark Programs
Benchmark programs come in various sizes and various levels of credibility. Going top-down benchmark programs form the following hierarchy:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOSP/SIPEW'10, January 28–30, 2010, San Jose, California, USA.
Copyright 2010 ACM 978-1-60558-563-5/10/01...$10.00.

- Real workloads
- Standard benchmarks
- Synthetic workloads
- Kernels
- Microbenchmarks

Real workloads are at the top level; they represent themselves and only they can avoid the standard criticism that claims (usually without any proof) that a given benchmark does not represent the real workload. Unfortunately, real workloads are regularly too complex, too specific in hardware/software resource demands, insufficiently portable, based on proprietary data, or simply not available (incomplete or currently in development/updating), so that they cannot be directly used for performance measurement. In addition, real workloads regularly change when ported from an existing platform to a new platform. That yields an unavoidable level of uncertainty in real workload definition.

Since real workloads are rarely suitable for benchmarking, the second best approach might be to measure performance using standard industrial benchmark suites, such as those provided by SPEC [15][13]. Standard benchmarks are selected real applications of substantial size and complexity (e.g. the gcc compiler, the games of chess and Go, or a linear programming solver). The main advantage of this approach is the public availability of a database of measurements for the majority of commercially available computers. Not surprisingly, it is left to users to prove why some of standard benchmarks should be used as sufficiently good representatives of a real workload.

Synthetic workloads [10] are composed of frequently used basic operations (or kernel functions) but they are not real applications. They have the advantage of fast and automatic generation, and the problem of proving representativeness similar to standard benchmarks. As opposed to standard benchmarks which have a fixed structure, synthetic workloads have adjustable parameters and can be easily customized to serve as an approximation of a real workload.

Kernels are frequently used important functions (e.g. matrix inversion, sorting, searching etc.) that can be used either individually or as building blocks of synthetic benchmarks

Microbenchmarks are small code segments designed to isolate a specific performance feature and provide reliable performance indicators that characterize the selected feature (e.g. Fibonacci number generator and Ackermann's function for evaluation of the efficiency of recursive calls, matrix multiplication for evaluation of the efficiency of array processing, etc.)

2.2 A Linear Sequential Workload Decomposition Model

The process of generation of synthetic programs always consists of aggregating various program components and modules to create a complex workload. Therefore, it is useful to start with the simplest way of decomposing and aggregating program components.

In the simplest case let us assume a sequential machine that is designed to execute a set of k basic operations $I_1,...,I_k$. These operations can be machine instructions of a specific processor, or bytecodes implemented by a virtual machine, or any other more complex basic operations. We also assume that the basic operations have average execution times $t_1,...,t_k$. If a benchmark program B_i consists of sequential execution of basic operations, then the total execution time of B_i is $T_i = f_{i1}t_1,...,f_{ik}t_k$ where $f_{i1},...,f_{ik}$ denote frequencies of individual basic operations.

The benchmark program B_i can be characterized by the frequency distribution $f_{i1},...,f_{ik}$. For scaling purposes B_i can be executed multiple times, for example using a loop

```
for(m=0; m<M; m++) { B_i }
```

Multiple repetitions of B_i do not change its characteristics. Similarly, the execution time can be normalized:

$$\overline{T_i} = p_{i1}t_1,...,p_{ik}t_k, \quad p_{ij} = f_{ij} / F_i, \quad j = 1,...,k$$
$$F_i = \sum_{j=1}^{k} f_{ij}, \quad \sum_{j=1}^{k} p_{ij} = 1, \quad T_i = F_i\overline{T_i}$$

The workload is now characterized using the probability distribution $p_{i1},...,p_{ik}$.

Let B be a given (possibly complex) workload that can be characterized using the frequency distribution $f_1,...,f_k$, the probability distribution $p_1,...,p_k$, the total number of executed operations F and the run time $T = F(p_1t_1,...,p_kt_k)$. Suppose now an ideal case where we have n component benchmark programs $B_1,...,B_n$ that satisfy the condition $p_j = W_1p_{1j} + ... + W_np_{nj}$, $j = 1,...,k$, $0 \le W_i \le 1$, $W_1 + ... + W_n = 1$. The weights $W_1,...,W_n$ denote the relative importance of component benchmarks $B_1,...,B_n$. In such a case we have

$$T = F\sum_{j=1}^{k}\sum_{i=1}^{n} W_i p_{ij} t_i = F\sum_{i=1}^{n} W_i \sum_{j=1}^{k} p_{ij} t_i = F\sum_{i=1}^{n} W_i \overline{T_i} = F\overline{T}$$

Therefore, if $p_j = W_1p_{1j} + ... + W_np_{nj}, \quad j = 1,...,k$ then the normalized run time of workload B is a weighted arithmetic mean of the normalized run times of benchmarks $B_1,...,B_n$:

$$\overline{T} = \sum_{i=1}^{n} W_i \overline{T_i}$$

Furthermore, the run time of workload B is a linear combination of the run times of workloads $B_1,...,B_n$:

$$T = F\sum_{i=1}^{n} W_i \frac{T_i}{F_i} = \sum_{i=1}^{n} \frac{FW_i}{F_i} T_i = \sum_{i=1}^{n} Q_i T_i, \quad Q_i = \frac{FW_i}{F_i}$$

If the component benchmarks are tuned to (approximately) satisfy $F = F_i$, $i = 1,...,n$ then the run time of workload B can be computed in the same way as the normalized run time. In this idealized case the component benchmarks can be interpreted as vectors that can be added to yield exactly the complex workload. In other words, the complex workload can be exactly decomposed into component benchmarks, and a well selected group of component benchmarks can be used to estimate the behavior of various complex workloads.

Of course, real computers rarely behave in linear and sequential way. There is always a degree of parallelism in the execution of basic operations, as well as nonlinear phenomena caused by the processor and memory architectures, control and interfacing of peripheral units, and the operating system activities. In real cases the sequential linear model can only be used as an approximation. Instead of satisfying the condition $p_j = W_1 p_{1j} + ... + W_n p_{nj}$ we can minimize the absolute difference between the desired value p_j and its approximation $\tilde{p}_j = W_1 p_{1j} + ... + W_n p_{nj}$. Using differences $| p_j - \tilde{p}_j |$, $j = 1,...,k$, we can select the weights $W_1,...,W_n$ that minimize the error function

$$E(W_1,...,W_n) = 50 \sum_{j=1}^{k} | p_j - W_1 p_{1j} - ... - W_n p_{nj} | \ [\%]$$
$$0 \le E(W_1,....W_n) \le 100\%$$

Alternative forms of the error function are:

$$E_1(W_1,...,W_n) = \sum_{j=1}^{k} (p_j - W_1 p_{1j} - ... - W_n p_{nj})^2$$
$$E_2(W_1,...,W_n) = \max_{1 \le j \le k} | p_j - W_1 p_{1j} - ... - W_n p_{nj} |$$

If n>2 the optimum weights can be found using the Nelder-Mead simplex algorithm [14]. If n=2 then the problem is trivial and can be solved by simple linear search. For example, the Java bytecode grouping technique presented in [9] identified a hierarchical decomposition from 2 to 17 characteristic groups of bytecodes. For simplicity, let us characterize workload using four distinct groups of operations: (1) calculation (arithmetic and logic operations, data type conversion), (2) compare/branch operations, (3) OO operations (object create/access, method invoke/return), and (4) operand stack manipulation. Suppose that we have component programs B_1, B_2 that are used to model a given workload B as follows:

B_1: $[p_{11}, p_{12}, p_{13}, p_{14}] = [0.4, 0.3, 0.2, 0.1]$
B_2: $[p_{21}, p_{22}, p_{23}, p_{24}] = [0.2, 0.35, 0.25, 0.2]$
B: $[p_1, p_2, p_3, p_4] = [0.25, 0.3, 0.3, 0.15]$
$E(W_1,W_2) = 50(| p_1 - W_1 p_{11} - W_2 p_{21} | + | p_2 - W_1 p_{12} - W_2 p_{22} |$
$+ | p_3 - W_1 p_{13} - W_2 p_{23} | + | p_4 - W_1 p_{14} - W_2 p_{24} |), \quad W_1 + W_2 = 1$

The search of all values of W_1 yields the following optimum results that correspond to the minimum error:

$$W_1 = 0.25, \quad W_2 = 0.75, \quad E(0.25, 0.75) = 3.75\% \text{ (min error)},$$
$$\tilde{p}_1 = 0.25, \quad \tilde{p}_2 = 0.3375, \quad \tilde{p}_3 = 0.275, \quad \tilde{p}_4 = 0.1375$$

The presented linear sequential workload decomposition shows that complex workloads consist of simpler components that can be identified and organized as specialized program modules. In the case of real workloads that use various functions (interpreted as kernels) the linear decomposition can be used as an approximation at the level of kernels. In other words, the workload can be characterized by the kernel frequency distribution. If we maintain a kernel library containing trustworthy program modules, then we can develop systematic procedures for building complex workloads as suitable combinations of program modules from the kernel library. Benchmark program generators are tools that implement such procedures.

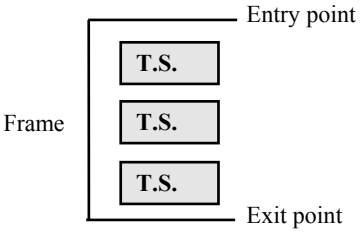
3. A RECURSIVE EXPANSION METHOD FOR PROGRAM GENERATION

3.1 Characteristics of Procedural Programs

The first step in the generation of benchmark programs is to understand the process of creating simple procedural programs containing control structures and arithmetic statements. Our approach to generating simple synthetic procedural benchmark programs is generally applicable in the majority of programming languages. For simplicity, we will use the notation of C/C++. Following is the list of basic control structures that can be used in a program generator:

1. Sequence (block): `{ block }`
2. Selections
   2.1 Skip (branch with one block):
   ```
   if (condition) { block }
   ```
   2.2 Branch with two blocks:
   ```
   if (condition) { block } else { block }
   ```
   2.3 Branch with multiple conditions and *n* blocks:
   ```
   if (condition1) { block }
   else if (condition2) { block }
   else if (condition3) { block }
   ……
   else { block }
   ```
   2.4 Multiple alternatives with selector (*n* blocks):
   ```
   switch (selector) {
   case a1: case a2: … :{block}; break;
   case b1: case b2: … :{block}; break;
      ……
   case z1: case z2: … :{block}; break;
   default { block } }
   ```
3. Loops
   3.1 Loop with the exit at the top (one block):
   ```
   while (condition) { block }
   ```
   3.2 Loop with the exit at the bottom (one block):
   ```
   do { block } while (condition)
   ```
   3.3 Loop with the exit in the middle (two blocks):
   ```
   do{
     { block }
     if (condition) break ;
     { block } }
   while (1);
   ```
   3.4 Loop with counter (one block):
   ```
   for(initialize; condition; update){block}
   ```

These control structures are encountered in almost all programming languages. Each control structure has a single entry point, a single exit point, and includes one or more *frames*. The frame is an empty containers denoted as a pair of braces { } in which we insert *blocks*. Each block consists of one or more statements. Each statement can contain one or more *frames* or it can be a *terminal statement*. The terminal statement is a statement that contains no frames where we could insert blocks; it is either an arithmetic statement or an I/O statement. Of course, each frame should be filled with statements. This process continues as long as we have frames that must be filled with blocks and terminates after inserting terminal statements in each block. A trivial example of a frame with three terminal statements is shown in Fig. 1.




```
void main (void)
{
   int a=123;
   int b=456;
   cout << a*b;
}
```

Figure 1. A frame with three terminal statements (T.S.)

The simplest way to organize a procedural program generator is to simulate the process of top-down procedural programming. This process in the case of a single module (either a function or a main program) starts by selecting the initial statement (either function or main) that contains a single frame. Then the frame is filled with appropriate statements. Nonterminal statements include frames. In the next step the frames are filled with other statements, and the process terminates when there are no more empty frames. The size of the resulting program depends on two parameters: the number of statements we insert in each frame and the maximum level of nesting of frames. Program generators can be organized to insert statements in frames, and when they reach the desired number of statements (or the desired maximum level of nesting) all frames are filled with terminal statements yielding a program that can be compiled and executed.

3.2 The Concept of Breadth and Depth Distribution

The breadth of a block is defined as the number of statements in a block and denoted *B*. The depth, *D*, is defined as the level of nesting where the initial frame, either a function or a main program, is interpreted as the initial level zero. For example, the following swap function

```
void swap(int & a, int & b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

and the main program in Fig. 1 have exactly the same structure, an initial frame with three statements, yielding B=3, D=1. Both breadth and depth are defined for each block, and the depth of the initial block is the depth of the whole program. The initial block of a program is considered the zero level of depth. For breadth and depth we count only executable statements, i.e. the statements that generate executable machine code. For example, the sort program shown in Fig. 2 has the depth D=4 and the breadth B=1. Only the "null program"

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

has D=B=0. Terminal statements are always counted and the following version of the null program

```
int main(void){ return 0; }
```

is assumed to have D=B=1.

Real programs have variable breadth and depth in various blocks and we can define the frequency distributions of breadth $F_b(B)$ and depth $F_d(D)$, as well as the corresponding probability distributions $P_b(B)$ and $P_d(D)$ for each program.

```
void sort(int v[ ], int n)
{
  int i, j;
  for(i=0; i<n-1; i++)
    for(j=i+1; j<n; j++)
      if(v[i]>v[j])
        swap(v[i], v[j]);
}
```

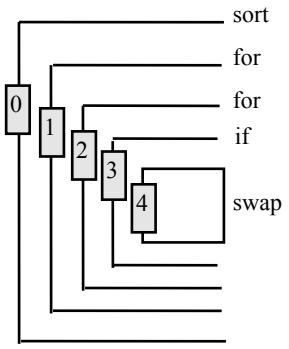


Figure 2. The depth of a sort program

To exemplify the concept of breadth and depth distribution, let us consider a traditional binary search function:

```
int bsearch(int v[ ], int n, int x)
{  int low, high, mid;
   low = 0;
   high = n-1;
   while(low <= high)
    { mid = (low+high)/2;
      if(x < v[mid]) high = mid-1;
      else if(x > v[mid]) low = mid+1;
      else return mid;
    }
    return -1;
}
```

The underlined statements in this program are the terminal statements and the breadth and depth analysis is shown in Fig. 3.

0	**1**	**2**	**3**	**4**	**D**	**(depth)**
{						
	T. S.					// low=0
	T. S.					// high=n-1
	{	T. S.				// mid=(...)/2
		{	T. S.			// high=mid-1
			{	T. S.		// low=mid+1
				T. S.		// return mid
			} B=2			
		} B=2				
	} B=2					
	T. S.					// return -1
} B=4						
	3	**1**	**1**	**2**	**F_d(D)**	**(frequency)**

B	1	2	3	4
F_b(B)	0	3	0	1
P_b(B)	0	0.75	0	0.25

D	1	2	3	4
F_d(D)	3	1	1	2
P_d(D)	0.43	0.14	0.14	0.29

Figure 3. The analysis of breadth and depth distribution for the binary search program

For each block the breadth is denoted after the closing brace and $F_b(B)$ denotes the number of frames that contain B statements. Similarly, $F_d(D)$ denotes the number of terminal statement at the depth level D. Consequently, the structure of bsearch function can be characterized by the breadth and depth absolute and relative frequencies ($F_b(B)$, $F_d(D)$, $P_b(B)$ and $P_d(D)$) shown in Fig 3.

From the standpoint of benchmarking, programs that have similar $P_b(B)$ and $P_d(D)$ distributions can be considered as similar programs. The difference between programs P1 and P2 can be defined using the following breadth and depth difference metrics:

$$D_b(P1,P2) = \frac{1}{2}\sum_{i=1}^{n} |P_{b1}(i) - P_{b2}(i)| \quad , \quad 0 \le D_b(P1,P2) < 1$$

$$D_d(P1,P2) = \frac{1}{2}\sum_{i=1}^{n} |P_{d1}(i) - P_{d2}(i)| \quad , \quad 0 \le D_d(P1,P2) < 1$$

Synthetic benchmark programs should be created so that they have the same breadth and depth distributions as the real programs they represent. In such a way a synthetic benchmark program can be a representative (or even a clone) of one or more real programs. Automatic generators of procedural code should generate programs having desired breadth and depth distributions.

3.3 Recursive Generator of Procedural Code

The traditional top-down program development process consists of stepwise refinements realized by inserting blocks of statements into frames until all frames are filled with terminal blocks that contain only terminal executable statements. We call this process the recursive expansion process (REX). The REX process of benchmark program generation is an automatic implementation of the top-down program development process and includes the following main steps:

1. **Initialization:** Create an initial block either as a function or a main program frame.
2. **Expansion:** In each existing frame insert a given number of statements with new frames. Increase the size of program by repeating this step a necessary number of times.
3. **Termination:** If the size of the generated program is sufficient insert in each frame a terminal statement.

The concept of REX generator is presented in Fig. 4 and Fig. 5. There are two functions, *STATEMENT* and *BLOCK*, which call each other. The function *BLOCK* fills a frame with random statements. The function *STATEMENT* generates a random statement that can be a terminal (arithmetic assignment) statement, or a statement that includes one or more frames. For statements with frames the function *STATEMENT* calls *BLOCK* in order to fill the frame. The termination of recursive calls occurs when we reach the desired size of generated program.

The basic tool we use for building these programs is a class *string* that supports an overloaded string assignment operator (=), and a string concatenation operator (+). The main components of a toy REX generator are presented in Fig. 5. In addition to main functions *STATEMENT* and *BLOCK* Fig. 5 also includes the main program that generates the synthetic program demo.cc.

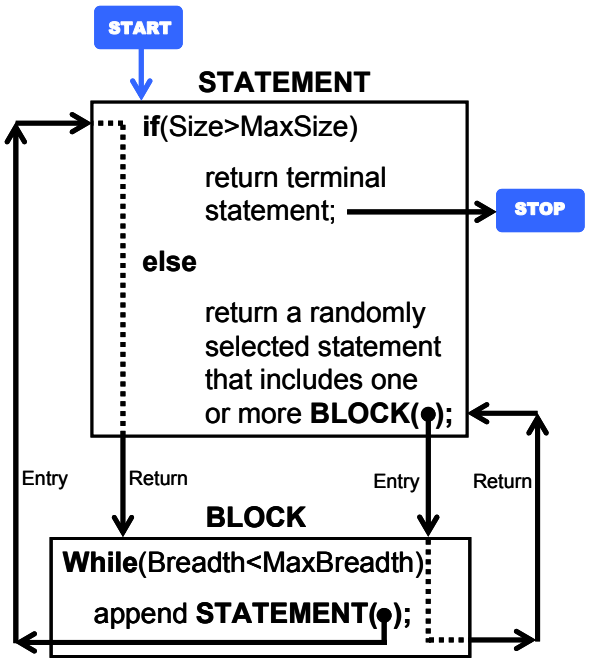


Figure 4. The REX model recursion

```
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";
 }
}
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) + "}";
}
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";
}
```

Figure 5. Main components of a toy REX generator

The presented toy generator uses auxiliary functions that are not shown in Fig. 5: *init* (function that defines and initializes variables used by other functions), *indent* (function that inserts spaces and new lines to properly indent the resulting code), *assignment* (function that creates an assignment using random arithmetic expression) and *condition* (function that creates a random condition that is used by control statements if, while and do). The number of scalar variables used in this program is adjusted by the global parameter *nvars*.

A fully expanded version of the REX generator is called BenchMaker1 (BM1) and it can generate 2 million lines of code per minute on a typical PC. BM1 creates C++ code as a sequence of functions that include control structures distributed according to user's input, and a main program that calls these functions.

```
#include<iostream.h>
void main(void)
{
  int l,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(l=0; l<20000; l++) G[l]=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(l=0; l<3; S+=G[l], l++)
    cout << G[l] << ((l+1)%10 ? ' ':'\n');
  cout << "\nNumber of control statements = 3";
  cout << "\nExecuted control statements  = " << S << '\n';
}
```

Figure 6. A small demo main program created by BenchMaker1

A small synthetic main program without functions, created by BM1, is shown in Fig. 6. In order to make such programs executable we use counters in conditions so that *if* statements uniformly branch and uniformly use both blocks, and loops perform a given number of iterations. At the end we display the basic statistics of executed instructions.

Synthetic programs of the type shown in Fig. 6 are primarily used as inputs for testing compilers [12]. The measured parameters include the compilation speed, the density of generated code, and optimizing features. These programs can also be executed as synthetic benchmarks, in cases where their structure is considered acceptable. However, this form of generators can also be used in a more sophisticated way, if instead of (some or all) arithmetic terminal statements we use function calls in a library of trustworthy kernels. In such cases the complexity of synthetic benchmark can match the complexity of real workloads and still have advantages of fast, flexible and low cost code generation. Such generators, called BenchMaker2 (BM2), are described in the next section.

4. A KERNEL INSERTION METHOD FOR PROGRAM GENERATION

4.1 The Concept of Kernel Insertion Generator

The basic component of BM2 is a library of independent expertly written programs called kernels, which provide a wide spectrum of traditional data processing functions. Each kernel is self-contained: it generates input data, performs desired processing, and verifies that the obtained results are correct. Benchmark programs are organized using a kernel insertion (KIN) process: the user first selects the basic structure of the synthetic benchmark and the benchmark generator then inserts kernels in that structure according to desired distribution. The result is a scalable synthetic benchmark of desired size and characteristics, which combines the quality code from the kernel library and the systematic growth of code provided by BM2.

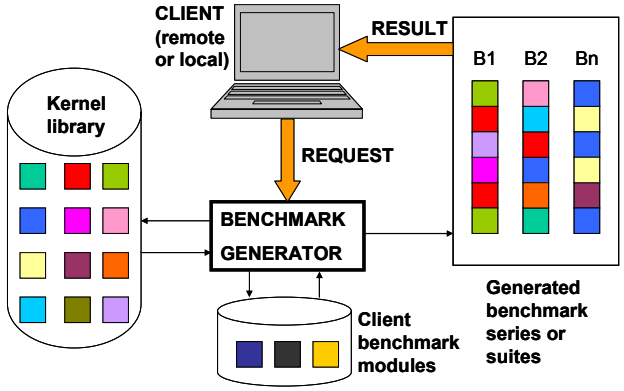


Figure 7. KIN process and the organization of BM2

The concept of BM2 is shown in Fig. 7. BM2 is designed primarily to serve remote users over the Internet. The users get the graphical interface that is used to specify the structure, functions, size, and the number of desired benchmark programs. BM2 server receives user's request, generates benchmarks, and delivers them to the user as e-mail attachments containing source code. The whole system can be organized as open source, serving a large user community. To achieve ultimate flexibility BM2 can integrate library kernels with client's benchmark modules providing a possibility to combine trustworthy programs from various sources.

The basic goal of BM2 is to provide scalable benchmarks. Benchmark scalability can be interpreted in several ways. We differentiate (1) time scalability, (2) space scalability, (3) parametric scalability, (4) structural scalability, and (5) functional scalability. Time scalability is the simplest form, where the run

time can be adjusted according to user needs. Similarly, the space scalability is the possibility to adjust the memory consumption and disk space consumption of benchmarks as well as the intensity and type of memory and disk accesses. In the case of parametric scalability, it is possible to adjust parameters of benchmark workload to increase the number of users, the number of nodes etc., while keeping the same type and structure of workload. The structural scalability refers to cases where the structure of workload can be modified and adjusted, as in the case of network benchmarks. Functional scalability is based on semantic characterization of workload: users can select workload functions that perform similar type and volume of processing as their expected workload.

4.2 Kernel Library

It is convenient to identify kernel programs using standard codes that consist of six characters, LAGS##, which can be interpreted as follows:

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

Our classification of areas includes processor performance kernels (nonnumerical, seminumerical, numerical, and object-oriented), memory access kernels (aimed at caching and paging performance analysis), disk and peripheral kernels, system kernels, and an open-ended area of user programs. The BM2 program generator can combine library kernels and user kernels to produce compound benchmarks. Following are the basic areas, groups and subgroups of kernels:

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 processing
115 Dynamic lists, queues, and trees
116 Search, sort, and merge
117 Recursive nonnumerical problems
118 Combinatorial problems and games

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 and data compression
127 Recursive seminumerical problems

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

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

2 MEMORY ACCESS KERNELS (PAGING AND 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 accesses
312 Random access

32 Other peripheral kernels
320 Miscellaneous
321 VDU and graphics
322 Archival tape access (backup and restore)

4 SYSTEM KERNELS

41 Processes
410 Miscellaneous
411 Process create and delete

42 Threads
420 Miscellaneous
421 Thread create and delete

43 Signals and alarms
430 Miscellaneous
431 Signals
432 Alarms

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

5 USER PROGRAMS

50 Miscellaneous
500 Miscellaneous

For example, according to this classification C12304 denotes a kernel written in C++ that is processor-bound, seminumerical, implements a graph algorithm, and is the fourth program in this subgroup.
The basic idea of the presented classification is to uniformly cover a wide spectrum of possible typical operations, enabling users to select kernels that have the highest level of similarity with their specific applications. Of course, the presented initial classification can be expanded and/or modified.

4.3 Kernel Design Concepts

Kernels are components that are compiled and executed as parts of synthetic benchmark workloads that have substantial size and complexity. Many copies of the same kernel may be used in various parts of a benchmark, and they have to behave in a stable and fully controlled way. This role implies that kernels must satisfy the following conditions:

(a) Kernels must be self-contained (designed as a block that can be inserted at any place in a benchmark program)
(b) To secure maximum mobility of kernel code, its dependence on environment should be kept at minimum (usage of only a few indispensable global variables).
(c) Kernels must be resistant to elimination by optimizing compilers.
(d) All input data must be internally generated.
(e) The number of lines of code in a kernel must be limited to secure sufficient granularity of benchmark workload.
(f) 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.
(g) To provide the equality of impact all kernels must be calibrated to run approximately same time.

Kernels that satisfy the above criteria have the standard structure shown in Fig. 8. Following is the list of the most important global parameters and a short description of their role:

- **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 kernel calibration process
- **TRACE** = benchmark program trace flag
- **STARTTIME** = start of measurement time

```
{  // 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)
```

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

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

  // End of program test

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

Figure 8. The standard kernel structure

The most important global parameters of these functions are SEC and KERNEL_COUNT. The parameter RATE specifies the number of kernel initialization-computation-validation cycles per second, and its value is adjusted during the calibration process so that the kernel run time is 1 second. Faster machines have larger values of RATE than slower machines. Consequently, RATE is a suitable indicator of the speed of a calibrated computer in the area of a specific kernel.

An example of a short kernel code that belongs to the object construction/destruction/manipulation group is shown in Fig. 9. This kernel uses global variables I, J, SEC, RATE, G, and KERNEL_COUNT. Variable G is initially set to 0. The expression G=1-G in the loop generates the sequence 1, 0, 1, 0,… that is used for initializing objects. These values and conditional displaying of error message prevent optimizing compilers to detect that the kernel does not generate results and can be simplified or eliminated.

4.4 The Structure of BM2 Benchmarks

The operation of BM2 generator is summarized in Fig. 10. BM2 supports the following five benchmark generation models:

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

```
{ // C14102
char* name="C14102: An array of objects";
const int SIZE = 10000, N=6;
for(I=0; I<SEC; I++)
for(J=0; J<RATE; J++)
{
  class SumArray
  {
   private:
     double a[2*N];
   public:
     SumArray( ) { }
     void Init(int N)
     { G = 1 - G;      //  0, 1, 0, 1, 0, …
       for(int i=0; i<2*N; i++) a[i] = G;
     }
     double Sum(int N)
     { double sum=0.;
       for(int i=0; i<2*N; i++) sum += a[i];
       return sum;     // Result: sum = N
      }
  } array[SIZE];

  int i;
  double sum=0.;

  // Initialize an array of objects
  for(i=0; i<SIZE; i++)    array[i].Init(N);

  // Process
  for(i=0; i<SIZE; i++) sum += array[i].Sum(N);

  // Validate (the correct result is sum = N * SIZE)
  if(sum != N*SIZE)
  { cout << "\nError in " << name
         << "\nsum = "  << sum  << '\n' ;
    exit(1);
   }
 }
terminator ( name ) ;
}
```

Figure 9. A sample short array processing kernel

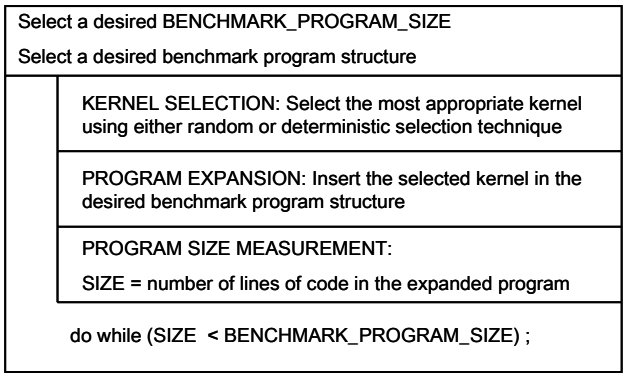


Figure 10. The operation of BM2 generator

The structure of the benchmark program generated by BM2 is specified by a model selection parameter (ProgType). The simplest form is a sequence, where kernels are directly inserted in the main benchmark program as exemplified in Fig. 11. Some kernels are inserted multiple times according to the desired kernel probability distribution.

Another version of sequence is a sequence of functions, where we typically use one kernel per function, as exemplified in Fig. 12.

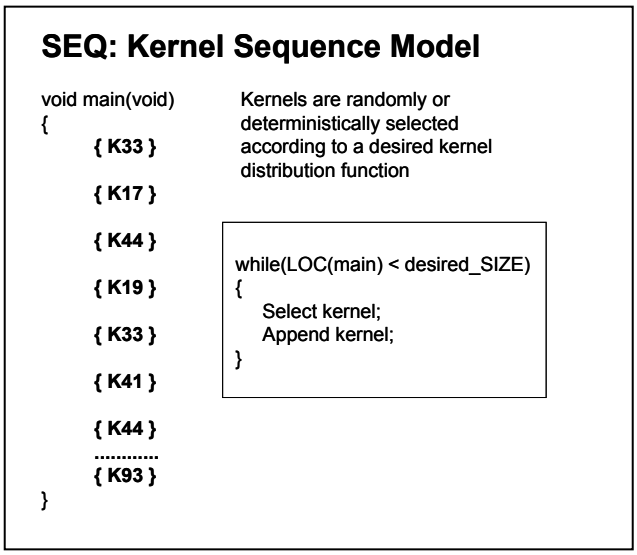


Figure 11. An example of the kernel sequence model

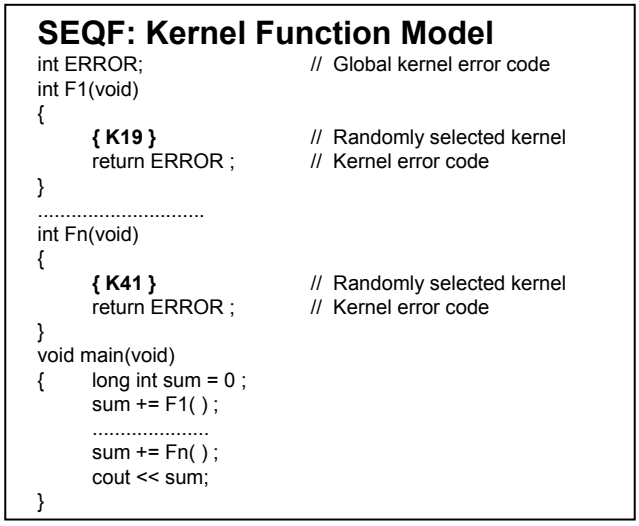


Figure 12. An example of the kernel function sequence model

In all kernel sequence models we assume the use of workload characterization by kernel distribution, i.e. we must select kernels according to a desired kernel probability distribution. If the selection is based on random number generators the resulting distribution will substantially differ from the desired distribution. It is much better to use a deterministic optimum selection (DOS)

method. In each step the DOS algorithm selects the kernel which minimizes the kernel distribution error. Suppose that we have the following parameters:

n = total number of available kernels

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

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

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

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

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

L = desired size of benchmark program [LOC]

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

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

According to DOS algorithm before adding a kernel we compute all distribution errors:

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

$1 \le j \le n$

Then, we select kernel K_r where $e(r) = \min_{1 \le j \le n} e(j)$. The advantages of the DOS approach are (1) simplicity, (2) close to optimum distribution in each insertion step (i.e. the program can terminate at any time), and (3) good accuracy for any program size. A minor disadvantage is that each kernel selection needs time O(n). The corresponding benchmark generation process is:

do{
  r = (integer from 1 to n selected by the DOS algorithm
    according to the desired kernel distribution) ;
  Insert kernel K_r in the benchmark program;
  $size = f_1 L_1 + f_2 L_2 + ... + f_n L_n$ (the number of lines of code
    after the addition of kernel K_r);
} while (size < L);

If we want to generate the minimum size benchmark program, it is suitable to use the canonic loop-select form exemplified in Fig. 13. In this program each kernel appears only once. If the kernels are calibrated so that they run exactly one second, then the parameter TIME specifies the execution time in seconds. The selector( ) function determines the desired kernel distribution.
The same loop-select approach can be used to generate programs of any desired size if we allow kernels to repeat inside the switch-case structure. This approach is exemplified in Fig 14.
In cases where it is desirable to have a modular structure of the benchmark program, the loop-select concept can be implemented using the kernel functions (e.g. void Fi(void) {kernel}), and then the canonic loop-select models include only kernel function calls and not the whole functions. This approach yields the minimum size, similar to the MC model shown in Fig 13.

The KIN concept can also be combined with the recursive expansion (REX) method as exemplified in Fig. 15. The resulting technique is called the kernel-terminated REX because instead of terminal statements we insert either the complete kernels (as shown in Fig. 15) or kernel function calls.

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.

Figure 13. An example of the minimum size loop-select model

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.

Figure 14. An example of the adjustable size loop-select model

Kernel Terminated REX 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
}
```

Figure 15. An example of the kernel-terminated REX model

5. CONCLUSIONS

Benchmark program generators solve the problem of fast and inexpensive production of large benchmark programs. Such programs can be tuned to model a spectrum of natural workloads and can easily adapt to changing hardware and software environments created by the exponential growth of computer performance. Therefore, instead of searching for benchmarks that satisfy specific requirements, it is possible to produce benchmarks using automatic benchmark generators. This can significantly reduce the cost of benchmarking.

The recursive expansion model of benchmark programs is suitable for modeling procedural programs in the majority of high level languages. The stepwise nesting of blocks can be terminated by inserting either arithmetic terminal statements or kernels.

The size of automatically generated synthetic programs is not limited and their characteristics are easily adjusted by selecting structural properties of programs, such as breadth and depth distributions, the control structure distribution, and the semantic properties expressed by an appropriate distribution of kernels.

The use of kernel libraries for automatic generation of benchmark and test programs has a number of advantages that include flexibility, scalability, simplicity, convenience, and cost reduction.

The kernel insertion method yields flexible workloads because the kernel libraries can be easily updated, expanded and improved. The size, contents, and the number of kernels can grow to cover any specific area of interest.

The scalability of synthetic benchmarks has various forms. It is easy to adjust any desired number of lines of code of the resulting test and benchmark programs, as well as any run time, to match the widest range of computer power. The structure of the generated benchmarks can also be conveniently adjusted. However, the most important scalability feature is the possibility to select or quickly modify the desired functionality of resulting benchmarks, what can position resulting benchmarks in any area that is represented by the kernel library.

The cost of benchmarking is the most important factor in evaluating the presented approach. In the case of standard industrial benchmarks, the cost of benchmarking is rather high because it includes permanent search for new benchmarks in an effort to follow Moore's law. When new standard benchmarks are introduced, computer manufacturers must promptly test all their products with the new benchmarks. Finally, computer users must also follow the changes in standard benchmark suites and eventually pay all benchmark development and maintenance costs.

The automatic benchmark generation method is suitable for users, because it comes both as a tool and as a service. The concept of e-mail delivery of benchmark suites from a benchmark server is convenient for many users, particularly those who need fast, flexible, and inexpensive solutions.

6. REFERENCES

[1] Asanovic, K. et al., A View of the Parallel Computing Landscape. CACM, Vol. 52, No. 10, pp. 56-67, 2009.

[2] Dujmović, J.J., *Evaluation and Design of Benchmark Suites*. Chapter 12 in *Advanced Computer Performance Modeling and Simulation*, Edited by K. Bagchi, J. Walrand, and G.W. Zobrist, Gordon and Breach, 1998, pp. 278-323.

[3] Dujmović, J.J., Universal Benchmark Suites – A Quantitative Approach to Benchmark Design. *Performance Evaluation and Benchmarking with Realistic Applications,* Edited by Rudolf Eigenmann, MIT Press, pp. 257-287, 2000.

[4] Dujmović, J.J and I. Dujmović, *Evolution and Evaluation of SPEC Benchmarks*. Performance Evaluation Review, Vol. 26, No. 3, pp. 2-9, 1998.

[5] Dujmović, J.J., E. Horvath, and H. Lew, *Benchmark Program Generator for Compiler Performance Analysis*. CMG99 Proceedings, Vol. 2, pp.838-847, 1999.

[6] Dujmović, J.J. and Howard Lew, *A Method for Generating Benchmark Programs.* CMG 2000 Proceedings, Vol. 1, pp. 379-388, 2000.

[7] Dujmović, J.J. and Murat Cengiz, A Kernel Library for Benchmark Program Generators. CMG 2003 Proceedings, Vol. 2 pp. 609-618, 2003.

[8] Gray, J. *What Next? A Dozen Information-Technology Research Goals*. Microsoft Research Technical Report MS-TR-99-50, 1999.

[9] Herder, C. and J.J. Dujmović, *Workload Characterization Using Metrics Based on Instruction Grouping*. International Journal of Computer and Information Sciences, Vol. 5, No. 1, March 2004.

[10] Jain, R., *The Art of Computer Systems Performance Analysis.* John Wiley, 1991.

[11] Kubiatowicz, J., Introduction to Parallel Architectures. http://parlab.eecs.berkeley.edu/bootcampagenda , 2009

[12] Lew, H. and J.J. Dujmović, Performance Evaluation and Comparison of C++ Compilers. CMG 2000 Proceedings, Vol. 1, pp. 241-252, 2000.

[13] Mirghafori, N., M. Jacoby, and D. Patterson, *Truth in SPEC Benchmarks*. Computer Architecture News, Vol. 23, No. 5, pp. 34-42, December 1995.

[14] Nelder, J.A. and Mead, R. (1965), "A simplex method for function minimization", *Comput. J.*, **7**, pp. 308–313. (See http://www.scholarpedia.org/article/Nelder-Mead_algorithm)

[15] SPEC, The Current Benchmarks. http://www.spec.org/osg/ , 2009.

[16] Wikipedia, Moore's law. http://en.wikipedia.org/wiki/Moore's_law