Automatic Generation of Benchmark Workloads

(Making programs that make programs)

Jozo J. Dujmović
Department of Computer Science
San Francisco State University
jozo@cs.sfsu.edu
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
Contents

1. Industrial benchmarks
2. Benchmark scalability
3. Generators of compilable programs (Recursive expansion model - BenchMaker 1)
4. Generators of executable programs (Kernel insertion model - BenchMaker 2)
5. Applications of benchmark program generators
6. Towards open source benchmark manufacturing
Basic Types of Benchmark Workloads

- Individual benchmark programs
- Benchmark suites
- Benchmark series
Benchmark Suites

- A family of nonredundant benchmark programs having a variety workload characteristics
- Typical benchmark suites are expected to include a necessary and sufficient variety of workload characteristics that represent a set of expected natural workloads
- Typical usage: global comparison of competitive computer systems
Benchmark Series

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

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

(And Their Relation to Moore’s Law)
MOORE’S LAW: Exponential growth of computer performance

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

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

q = performance
\( t \) = time
q_0 = initial performance at time \( t=0 \)
T = performance doubling time
\( \cong 18 \) months for memory capacity
\( \cong 12 \) months for performance/price
Approach Currently Used by Industry

[1/2]

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

http://www.spec.org/osg/cpu2000/
Approach Currently Used by Industry [2/2]

The SPEC CPU Benchmark Search Program

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

Feasibility Analysis

Back of the Envelope

Typical memory of a standard PC = 256 MB

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

Typical memory of a standard PC = 256 MB

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

Minimum estimated cost = $41 Million

Reward offered by SPEC = $5,000
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.
Industrial benchmark suites (e.g. SPEC) use natural benchmarks and remain unchanged for years without possibility to follow the exponential growth of computer performance.
Desired Performance/Benchmark Relation

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

Benchmark generators ⇒ Benchmark scalability
Current Industrial Benchmarks

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

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

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

Such Benchmarks must be:

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

- Based on generators of scalable synthetic 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** (generator of compilable programs primarily used for compiler performance measurement and analysis)
- **BenchMaker 2** (generator of general purpose executable programs, used for computer performance measurements)
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.
Types of Benchmark Scalability

- **Time scalability** (user selects the benchmark run time)
- **Space scalability** (user adjusts the benchmark size and its memory consumption)
- **Parametric scalability** (adjustable for each benchmark)
- **Structural scalability** (benchmarks have adjustable structure; generation of benchmark series and suites)
- **Functional scalability** (semantic workload characterization: each user can select functions that are similar to an existing or expected user workload)
- **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, called 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, called 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
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).
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
- Compiler evaluation and comparison
- 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, called kernels.
Generation of Compilable Programs for Compiler Performance Analysis

(BenchMaker 1 and the Recursive Expansion Program Generation Model)
Block Containing Statements

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

int func(arguments)
{
    // block
    Statement
    Statement
    Statement
    Statement
}
```
Classification of Statements

• **Expandable statements** (contain blocks and can be expanded by inserting statements into blocks)
• **Terminal statements** (fixed contents that cannot be expanded)
  – Simple (arithmetic)
  – Compound (fixed blocks called kernels)
Expandable Statement

If (condition)
{
Block of statements
}

Else
{
Block of statements
}
int main(int arguments)
{
    // block
}

Expansion of Statements

Expansion level (depth) 1

Expansion level (depth) 2

Expansion level (depth) 3

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

Expansion level (depth) 3
REX Program Model

- Each block contains one or more statements.
- Each control statement contains one or more blocks. An example of two blocks:
  
  ```
  if(condition) {block} else {block}
  ```
- Create programs by systematically inserting blocks into statements and statements into blocks (stepwise refinement).
- When the generated program attains a desired size, insert a “terminal block” (either an arithmetic statement or an executable kernel).
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(…);
The Concept of Breadth

{

  statement;

  statement;

  statement;  // B = 5

  statement;

  statement;

}

The Concept of Depth

{ 
  
  { 
    
    { 
      
      statement; // D = 2
    
  
  
  } 

}
A Toy REX Generator [1/3]

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

BenchMaker

Copyright © 2004 by Jozo Dujmović
A Toy REX Generator [2/3]

```c
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]

```c
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";
}
```
#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-m*m+h+g-f;
            l = m+d-n-m+n*i+n;
        }
        else
        {
            e = h*f-g-l*f+a+a*m;
            h = a-h*h-l+k*k-l*d+e-l*m;
        }
    }
    while(++G[1]%3) // 1,2,0,1,2,0,…
    {
        b = d-m-j+m-j+k-b+a+e-g-i+f*g;
        j = k*f*m*b*h-d+l+b;
    }
    for(I=0; I<3; S+=G[I], I++)
        cout << G[I] << ((I+1)%10 ? ' ':'\n');
    cout << "%nNumber of control statements = 3";
    cout << "%nExecuted control statements = " << S << '
';
}
Experiments With Compilable Benchmark Programs [1/2]

$ time ./tg

Toy program generator

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

$ wc -l demo.cc
100755 demo.cc

$ time g++ demo.cc

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

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

Density = 26.5 Bytes / PLOC
≈ 70 Bytes / LLOC
Experiments With Compilable Benchmark Programs [2/2]

```
$ time ./tg

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

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

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

```
$ time g++ demo.cc

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

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

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

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

**Density = 28.8 Bytes / PLOC**
Generators of Executable Programs

(BenchMaker 2 and the Kernel Insertion Program Generation Model)
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
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.
Kernel-Related Issues

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

CLIENT (remote or local)

REQUEST

RESULT

B1 B2 Bn

Generated benchmark series or suites

BENCHMARK GENERATOR

Client benchmark modules

Generated benchmark series or suites

Kernel library
Kernel Naming and Classification

L A G S ##

L = Programming language code:
C denotes C++
B denotes C language
J denotes Java
F denotes Fortran

A = Area code (0...9) for main kernel areas

G = Group code (0...9) inside an area

S = Subgroup code (0...9) inside the subgroup

## = 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)

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>121</td>
<td>Integer arithmetic and counters</td>
</tr>
<tr>
<td>122</td>
<td>Bitwise and integer operations/functions</td>
</tr>
<tr>
<td>123</td>
<td>Graph algorithms</td>
</tr>
<tr>
<td>124</td>
<td>Prime numbers</td>
</tr>
<tr>
<td>125</td>
<td>Random numbers and Monte Carlo methods</td>
</tr>
<tr>
<td>126</td>
<td>Cryptography</td>
</tr>
<tr>
<td>127</td>
<td>Recursive seminumerical problems</td>
</tr>
</tbody>
</table>
### Kernel Classification (3/9)

#### 1 PROCESSOR PERFORMANCE KERNELS

13 Numerical procedural kernels
   - 130 Miscellaneous
   - 131 Scalar floating-point arithmetic
   - 132 Library and special functions
   - 133 Arrays
   - 134 Polynomials
   - 135 Matrices
   - 136 Integrals and differential equations
   - 137 Recursive numerical problems
   - 138 Statistics
Kernel Classification (4/9)

1  PROCESSOR PERFORMANCE KERNELS
14 Object oriented kernels
   140 Miscellaneous
   141 Object construction/destruction/manipulation
   142 Overloading operators
   143 Inheritance and multiple inheritance
   144 Polymorphism
   145 Abstract classes
   146 Templates
   147 Exception handling
# Kernel Classification (5/9)

<table>
<thead>
<tr>
<th>2</th>
<th>MEMORY ACCESS KERNELS (PAGING &amp; CACHING)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>Static memory access</td>
</tr>
<tr>
<td>210</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>211</td>
<td>Uniform distribution, multiple localities</td>
</tr>
<tr>
<td>212</td>
<td>Normal distribution, multiple localities</td>
</tr>
<tr>
<td>22</td>
<td>Dynamic memory access</td>
</tr>
<tr>
<td>220</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>221</td>
<td>Uniform distribution, multiple localities</td>
</tr>
<tr>
<td>222</td>
<td>Normal distribution, multiple localities</td>
</tr>
</tbody>
</table>

BenchMaker Copyright © 2004 by Jozo Dujmović
Kernel Classification (6/9)

3 DISK AND PERIPHERALS ACCESS KERNELS

31 Disk access
   310 Miscellaneous
   311 Sequential access
   312 Random access

32 Other peripheral kernels
   320 Miscellaneous
   321 VDU and graphics
   322 Archival tape access
<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>SUB-CATEGORIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes</td>
<td>41 Processes</td>
</tr>
<tr>
<td></td>
<td>410 Miscellaneous</td>
</tr>
<tr>
<td></td>
<td>411 Process create and delete</td>
</tr>
<tr>
<td>Threads</td>
<td>42 Threads</td>
</tr>
<tr>
<td></td>
<td>420 Miscellaneous</td>
</tr>
<tr>
<td></td>
<td>421 Thread create and delete</td>
</tr>
<tr>
<td>Signals and alarms</td>
<td>43 Signals and alarms</td>
</tr>
<tr>
<td></td>
<td>430 Miscellaneous</td>
</tr>
<tr>
<td></td>
<td>431 Signals</td>
</tr>
<tr>
<td></td>
<td>432 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 // Abort benchmark execution
                        exit(1);
                    }
            }
    terminator( name );                           // Kernel termination function
}                                                             // (kernel/benchmark termination)

TIME = O(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 kernel calibration process
- **TRACE**: benchmark program trace flag
Benchmark Generation Process

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Select a desired BENCHMARK_PROGRAM_SIZE</td>
</tr>
<tr>
<td>2</td>
<td>Select a desired benchmark program structure</td>
</tr>
<tr>
<td>3</td>
<td>KERNEL SELECTION: Select the most appropriate kernel using either random or deterministic selection technique</td>
</tr>
<tr>
<td>4</td>
<td>PROGRAM EXPANSION: Insert the selected kernel in the desired benchmark program structure</td>
</tr>
<tr>
<td>5</td>
<td>PROGRAM SIZE MEASUREMENT:</td>
</tr>
<tr>
<td></td>
<td>SIZE = number of lines of code in the expanded program</td>
</tr>
<tr>
<td></td>
<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

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

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

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

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

\[ R = r_1 + \frac{(T - t_1)(r_2 - r_1)}{(t_2 - t_1)} \]

\[ G = \frac{T(RATE + 1) - T(RATE)}{T(RATE)} = \frac{1}{RATE} \]
BM2 System Overview

Remote User

INTERNET

Web Server (+JSP)

BenchMaker GUI

spec.in

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

spec.out

Kernels

LAGS##

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

BM2 Engine

BM2 user command line menu interface

Local Console User
Workload Characterization

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

- Kernel sequence (KS) 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
void main(void) {
    { K33 }
    { K17 }
    { K44 }
    { K19 }
    { K33 }
    { K41 }
    { K44 }
    ............
    { K93 }
    while(LOC(main) < desired_SIZE) {
        Select kernel;
        Append kernel;
    }
}
KF: Kernel Function Model

```c
int ERROR; // Global kernel error code
int F1(void)
{
    { K19 } // Randomly selected kernel
    return ERROR ; // Kernel error code
}

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

int Fn(void)
{
    { K41 } // Randomly selected kernel
    return ERROR ; // Kernel error code
}

void main(void)
{
    long int sum = 0 ;
    sum += F1( ) ;
    ....................
    sum += Fn( ) ;
    cout << sum;
}
```

BenchMaker Copyright © 2004 by Jozo Dujmović
MC: Minimum Size Canonic Loop-Select Model

```c
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]

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

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

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

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

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

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

\[ p_1, p_2, \ldots, p_n = \text{kernel probabilities} \]

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

INPUTS:

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

\[ L = \text{desired benchmark size} \]

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

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

and

$$f_1^* L_1 + f_2^* L_2 + ... + f_n^* L_n \approx L$$
Kernel Selection Problem [5/11]

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

\[
C(f_1, f_2, \ldots, f_n) = \left[ W \left( |f_1L_1 + \ldots + f_nL_n - L| \right)^r + (1 - W) \left( \sum_{i=1}^{n} \frac{f_i}{f_1 + f_2 + \ldots + f_n} - P_i \right)^r \right]^{1/r}
\]

where \(0 < W < 1\), \(1 \leq r \leq +\infty\) (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.

\[
do{
\begin{align*}
    r &= \text{(random integer from 1 to } n\text{ distributed according} \\
    &\quad \text{to any desired kernel distribution)}; \\
    \text{Insert kernel } K_r \text{ in benchmark program;} \\
    \text{size} &= \text{(number of lines of code after the addition of} \\
    &\quad \text{kernel } K_r); \\
\end{align*}
\}
\text{while (size < } L);
Kernel Selection Problem [8/11]

Advantages of random selection:

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

Disadvantage:

- Large and random distribution errors for small and medium numbers of kernels
Kernel Selection Problem [9/11]

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

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

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

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

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 $O(n)$
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++

\[ C = 0.0014L + 3.3544 \]

Cygwin g++

\[ C = 0.004L + 2.4595 \]

6 sec

10 sec
CodeWarrior C++

\[ C = 3.28 + 9.58 \times 10^{-6} L^{2.062} \]

Intel C++
Comparison of Object Program Sizes

Cygwin g++

Object Program Size (bytes)

Lines of Code L

Mobj = 77.523 L + 2577.3

154 KB

Visual C++

Object Program Size (bytes)

Lines of Code L

Mobj = 58.291 L + 3327.6

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 (bytes)

Executable Size (bytes)

Lines of Code L

Mobj = 58.291 L + 3327.6

M = 46.39 L + 57181

Visual C++

146 KB
Nonlinear Phenomena – Intel C++ Compiler

\[ \text{Mobj} = 47.694 \cdot L + 13218 \]

\[ \text{M} = 31.137 \cdot L + 55582 \]
Nonlinear Phenomena – Metrowerks CodeWarrior

Object Program Size (bytes)

Lines of Code L

Mobj = 81.573 L + 166464

Executable Program Size (bytes)

Lines of Code L

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
Performance ranking of compilers using a Pentium based system

Execution time ratio:

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

Global criterion:

\[ R = r^{W_T} \left( \frac{m_{0A}}{m_{0B}} \right)^{W_{m_0}} \left( \frac{m_{1A}}{m_{1B}} \right)^{W_{m_1}} \left( \frac{t_{0A}}{t_{0B}} \right)^{W_{t_0}} \left( \frac{t_{1A}}{t_{1B}} \right)^{W_{t_1}} \]

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)

AMD Athlon 1.0GHz, 128MB RAM

<table>
<thead>
<tr>
<th>Name</th>
<th>Relative Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICO</td>
<td>100.00%</td>
</tr>
<tr>
<td>ICD</td>
<td>98.69%</td>
</tr>
<tr>
<td>CGO</td>
<td>87.09%</td>
</tr>
<tr>
<td>LGO</td>
<td>76.17%</td>
</tr>
<tr>
<td>VCO</td>
<td>71.14%</td>
</tr>
<tr>
<td>BCO</td>
<td>41.95%</td>
</tr>
<tr>
<td>BCD</td>
<td>38.12%</td>
</tr>
<tr>
<td>LGD</td>
<td>32.58%</td>
</tr>
<tr>
<td>VCD</td>
<td>31.29%</td>
</tr>
<tr>
<td>CGD</td>
<td>31.29%</td>
</tr>
</tbody>
</table>
Using Calibration for Performance Comparison (3/3)

Intel Centrino 1.4GHz, 512MB RAM

<table>
<thead>
<tr>
<th>Relative Rates</th>
<th>ICD</th>
<th>ICO</th>
<th>VCO</th>
<th>CGO</th>
<th>LGO</th>
<th>BCO</th>
<th>BCD</th>
<th>VCD</th>
<th>LGD</th>
<th>CGD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>100.00%</td>
<td>99.81%</td>
<td>60.87%</td>
<td>60.45%</td>
<td>53.51%</td>
<td>33.26%</td>
<td>32.94%</td>
<td>26.11%</td>
<td>25.62%</td>
<td>23.89%</td>
</tr>
</tbody>
</table>

Copyright © 2004 by Jozo Dujmović
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 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 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.
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)
Developed by S. Murat Cengiz

- Web based, dynamic interface
- JSP & Java based, outputs are pure HTML
- Most browsers are supported
- Tomcat4.1 on the server side
- List of kernels are read at run-time from configuration files and the interface adapts itself to changes
- Simple to use
- Support for e-mail retrieval of benchmarks
- Supports multiple users and projects
BenchMaker User Interface (2/9)

BenchMaker

Generator of Scalable Benchmark Workloads

You have to be logged in to use the system.

Log in

User Name: 
Password: 
Log in Cancel

Please refer to the links below for more information.

About BenchMaker | Download Papers | See Legal Stuff | Contact Us | Information for Guest Users | Contribute | Research with BenchMaker
BenchMaker User Interface (3/9)

Please select an option from the following list:

- List your projects
- Create new project
- Rename projects or update project description
BenchMaker User Interface (4/9)

Create a new project by filling out the fields:

- Unique Name
- Language
- Type
- Description

Create Cancel
BenchMaker User Interface (6/9)
BenchMaker User Interface (7/9)
BenchMaker User Interface (8/9)

<table>
<thead>
<tr>
<th></th>
<th>PROCESOR PERFORMANCE</th>
<th>MEMORY ACCESS (PAGING &amp; CACHING)</th>
<th>DISK &amp; PERIPHERALS ACCESS</th>
<th>SYSTEM</th>
<th>USER PROGRAMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>55.0 63%</td>
<td>32.0 36%</td>
<td>0.0 0%</td>
<td>0.0 0%</td>
<td>0.0 0%</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAX SEG or MAX KERNEL</th>
<th>DESIRED PROGRAM STRUCTURE</th>
<th>MIN LLOC</th>
<th>MAX LLOC</th>
<th>LLOC STEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>Kernel Sequence Function Model</td>
<td>500.0</td>
<td>2000.0</td>
<td>500.0</td>
</tr>
</tbody>
</table>

Save Project  | Delete Project  | Generate Benchmark(s) | Deliver by email
BenchMaker User Interface (9/9)

Logged in as jozo

<table>
<thead>
<tr>
<th>Project Category</th>
<th>Score</th>
<th>Weight</th>
<th>Custom?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 PROCESSOR PERFORMANCE</td>
<td>55.0</td>
<td>52%</td>
<td></td>
</tr>
<tr>
<td>2 MEMORY ACCESS (PAGING &amp; CACHING)</td>
<td>32.0</td>
<td>30%</td>
<td></td>
</tr>
<tr>
<td>3 DISK &amp; PERIPHERALS ACCESS</td>
<td>18.0</td>
<td>17%</td>
<td></td>
</tr>
<tr>
<td>31 Disk Access</td>
<td>15.0</td>
<td>14%</td>
<td></td>
</tr>
<tr>
<td>310 Miscellaneous</td>
<td>5.0</td>
<td>4%</td>
<td>Custom</td>
</tr>
<tr>
<td>31101 Miscellaneous</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311 Sequential Access</td>
<td>5.0</td>
<td>4%</td>
<td>Custom</td>
</tr>
<tr>
<td>312 Random Access</td>
<td>5.0</td>
<td>4%</td>
<td>Custom</td>
</tr>
<tr>
<td>31201 Random Access</td>
<td>5.0</td>
<td>4%</td>
<td>Custom</td>
</tr>
<tr>
<td>32 Miscellaneous Peripheral Access</td>
<td>3.0</td>
<td>2%</td>
<td></td>
</tr>
<tr>
<td>320 Miscellaneous</td>
<td>3.0</td>
<td>2%</td>
<td>Custom</td>
</tr>
<tr>
<td>32001 Miscellaneous</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>321 VDU &amp; Graphics</td>
<td>0.0</td>
<td>0%</td>
<td>Custom</td>
</tr>
<tr>
<td>32101 VDU &amp; Graphics</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>322 Archival Tape Access</td>
<td>0.0</td>
<td>0%</td>
<td>Custom</td>
</tr>
<tr>
<td>32201 Archival Tape Access</td>
<td>0.0</td>
<td>0%</td>
<td>Custom</td>
</tr>
<tr>
<td>4 SYSTEM</td>
<td>0.0</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>5 USER PROGRAMS</td>
<td>0.0</td>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>

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

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

BenchMaker  Copyright © 2004 by Jozo Dujmović  122
Conclusions

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


