Microarchitecture-Independent Workload Characterization Studies Using Pin

Pin tutorial @ IISWC-2007

September 29th 2007
Boston, MA, USA

Kenneth Hoste and Lieven Eeckhout
kenneth.hoste@elis.ugent.be - lieven.eeckhout@elis.ugent.be

ELIS, Ghent University, Belgium
Why microarchitecture-independent?

Prevalent workload characterization:
- simulation or hardware performance counters (IPC, cache miss rates, ...)
- single machine configuration

Problems:
- specific to chosen configuration(s)
- can be highly misleading!
Pitfall in prevalent workload characterization

Workloads seem similar using dependent metrics, but are clearly different using microarch-indep. metrics.
MICA to the rescue!

Microarchitecture-Independent Characterization of Applications

Pin tool which extracts µarch-indep. program characteristics, i.e., independent of:

- cache configuration
- branch predictor
- number of functional units
- ...
MICA: types of characteristics

- **itypes**: instruction mix
- **ppm**: taken rate, transition rate, Markov-chain based branch prediction
- **reg**: distribution of register dependency distances, avg. number of input registers, degree of use
- **stride**: distribution of memory access address distances
- **memfootprint**: memory footprint (# blocks/pages touched)
- **ilp**: amount of available inherent ILP
Using MICA is easy

Collect μarch-indep. chars for /bin/ls:

```
pin -t mica full all -- ls
```

results in a number of *pin.out files (6) containing program characteristics
Data extraction, data processing, insight!

*Data extraction (instrumentation):*
- how to extract instruction info using Pin?

*Data processing:*
- how to compute program characteristics?

*Insight:*
- how to gain insight?
Part 1: Data extraction

- **itypes**: instruction type
- **ppm**: branch ID, taken/non-taken
- **reg**: register reads/writes, register ID
- **stride**: memory reads/writes addresses
- **memfootprint**: see **stride**
- **ilp**: see **reg + stride**
VOID Instruction(INS ins, VOID *v){
    char cat[50]; char opcode[50];
    strcpy(cat, CATEGORY_StringShort(INS_Category(ins)).c_str());
    strcpy(opcode, INS_Mnemonic(ins).c_str());
    if(strcmp(cat,"COND_BR") == 0)
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)condBr_ins, IARG_END);
    if(strcmp(opcode,"MUL") == 0)
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)mul_ins, IARG_END);
}

VOID Fini(INT32 code, VOID *v){
    fprintf(stderr,"%lld cond. branches, %lld muls\n", cond_branches, muls);
}
VOID Instruction(INS ins, VOID *v) {
    char cat[50];
    strcpy(cat, CATEGORY_StringShort(INS_Category(ins)).c_str());
    if(strcmp(cat, "COND_BR") == 0) {
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)br, IARG_UINT32, br_cnt, IARG_END);
        br_cnt++;
    }
}

VOID Fini(INT32 code, VOID *v) {
    UINT32 i;
    for(i=0; i < br_cnt; i++) {
        fprintf(stderr, "branch %d: %lld\n", i, branch[i]);
    }
}

UINT32 br_cnt;
INT64* branch;
VOID br(UINT32 id) {
    branch[id]++;
}
void Instruction(INS ins, VOID *v) {
    char cat[50];
    strcpy(cat, CATEGORY_StringShort(INS_Category(ins)).c_str());
    if (strcmp(cat, "COND_BR") == 0) {
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)br,
                       IARG_ADDRINT, INS_Address(ins), IARG_END);
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)br,
                       IARG_ADDRINT, INS_Address(ins), IARG_END);
    }
}

VOID Fini(INT32 code, VOID *v) {
    UINT32 i;
    for (i = 0; i < br_cnt; i++) {
        fprintf(stderr, "branch %d: %lld\n", i, branch[i]);
    }
}
VOID Instruction(INS ins, VOID *v){
    UINT32 id;
    char cat[50]; strcpy(cat, CATEGORY_StringShort(INS_Category(ins)).c_str());
    if(strcmp(cat,"COND_BR") == 0){
        id = lookup(INS_Address(ins));
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)br, IARG_UINT32, id,
                       IARG_END);
    }
}

VOID Fini(INT32 code, VOID *v){ int UINT32 i;
    for(i=0; i < br_cnt; i++) { fprintf(stderr,"branch %d: %lld\n", i, branch[i]); }
}
Data extraction: branch (not-)taken rate

```c
ADDRTINT ba; INT64 brCnt, t, nt; BOOL last_br;
VOID br(ADDRTINT na){ brCnt++; ba = na; last_br = true; }
VOID instr(ADDRTINT a) { if(last_br) { if (a != ba) { t++; } else { nt++; } 
    last_br = false; }
}

VOID Instruction(INS ins, VOID *v){
    char cat[50]; strcpy(cat, CATEGORY_StringShort(INS_Category(ins)).c_str());
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)instr,
        IARG_ADDRINT, INS_Address(ins), IARG_END);
    if(strcmp(cat,"COND_BR") == 0)
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)br,
            IARG_ADDRINT, INS_NextAddress(ins), IARG_END);
}

VOID Fini(INT32 code, VOID *v){
    fprintf(stderr,%"lf taken, %lf not taken\n", (double)t/brCnt, (double)nt/brCnt);
}
```
VOID Instruction(INS ins, VOID *v){
    UINT32 i; UINT32 max = INS_MaxNumRRegs(ins);
    for(i=0; i < max; i++) {
        const REG reg = INS_RegR(ins, i);
        if( REG_valid(reg)){ // ALL registers (segment, fp, gen. purp., ...)
            INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)regRead, IARG_END);
        }
    }
}

VOID Fini(INT32 code, VOID *v){
    fprintf(stderr, “%lld register reads
”,reg_reads);
}

INT64 reg_reads;
VOID regRead() { reg_reads++; }

PROCESSING

INSTRUMENTATION

OUTPUT
Data extraction: register ID

INT64* reg_reads;
VOID reg_read(UINT32 id) { reg_reads[id]++;
}

VOID Instruction(INS ins, VOID *v){
    UINT32 i; UINT32 max = INS_MaxNumRRegs(ins);
    for(i=0; i < max; i++){
        const REG reg = INS_RegR(ins, i);
        if( REG_valid(reg)){
            INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)reg_read,
                IARG_UINT32, reg, IARG_END);
        }
    }
}

VOID Fini(INT32 code, VOID *v){
    UINT32 i;
    for (i=0; i < MAX_REG; i++) {
        if (reg_reads[i] > 0) fprintf(stderr,"[%s] %lld\n",
            REG_StringShort((REG)i).c_str(), reg_reads[i]);
    }
}

INT64* reg_reads;
VOID reg_read(UINT32 id) { reg_reads[id]++;
}
Data extraction: memory reads/writes

```c
INT64 ri, wi, rs;
VOID read_ins() { ri++; rs++; } VOID read() { rs++; } VOID write_ins() { wi++; }

VOID Instruction(INS ins, VOID *v){
    if( INS_IsMemoryRead(ins) ){
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)read_ins, IARG_END);
        if( INS_HasMemoryRead2(ins) )
            INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)read, IARG_END);
    }
    if( INS_IsMemoryWrite(ins) ){
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)write_ins, IARG_END);
    }
}

VOID Fini(INT32 code, VOID *v){
    fprintf(stderr, "%lld load ins (%lld loads), %lld store ins\n", ri, rs, wi);
}
```

**INSTRUMENTATION**

**OUTPUT**
Data extraction: memory addresses

```c
VOID r_mem(ADDRINT a) { fprintf(stderr,"memory read @ %x\n",a); }
VOID w_mem(ADDRINT a) { fprintf(stderr,"memory write @ %x\n",a); }

VOID Instruction(INS ins, VOID *v) {  
    if ( INS_IsMemoryRead(ins) ) {  
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)r_mem,  
                        IARG_MEMORYREAD_EA, IARG_END);  
        if ( INS_HasMemoryRead2(ins) )  
            INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)r_mem,  
                           IARG_MEMORYREAD2_EA, IARG_END);  
    }  
    if ( INS_IsMemoryWrite(ins) )  
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)w_mem,  
                       IARG_MEMORYWRITE_EA, IARG_END);  
}

VOID Fini(INT32 code, VOID *v) { }
```
Part 2: Data processing

Detailed description for:

- Markov-chain based PPM predictor
- instruction-level parallelism (ILP)

How much time is needed to collect the data?
Data processing: PPM-predictors

Prediction by Partial Match (PPM)
branch direction is predicted using a set of Markov chains;
longest matching branch history delivers prediction;
idealistic model for most common branch predictors

for each conditional branch:

- previous prediction correct?
- compare branch direction with previous prediction, keep track of number of mispredictions
- update pattern history tables for different predictors
- adjust saturating counter for branch history (global or local history, per-address or global table)
- predict next branch direction for each predictor

base prediction on branch history and value of saturating counter
( > 0 ⇒ taken, < 0 ⇒ not-taken)

more details, see “Analysis of Branch Prediction via Data Compression” by Chen et al., ASPLOS 1996
**Data processing: PPM-predictors**

<table>
<thead>
<tr>
<th>hist. length</th>
<th>history</th>
<th>value</th>
<th>pred.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-2</td>
<td>NT</td>
</tr>
<tr>
<td>1</td>
<td>“0”</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>“1”</td>
<td>-1</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>“00”</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>“01”</td>
<td>-3</td>
<td>NT</td>
</tr>
<tr>
<td></td>
<td>“10”</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>“11”</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

PAg (per-address history, global table)

Branch history:
1: 01010
2: 001001
3: 01

# pred.: 13
# mispred.: 4

predictor: 2
Data processing: PPM-predictors

<table>
<thead>
<tr>
<th>hist. length</th>
<th>history</th>
<th>value</th>
<th>pred.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-2</td>
<td>NT</td>
</tr>
<tr>
<td>1</td>
<td>“0”</td>
<td>(\times) 2</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>“1”</td>
<td>-1</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>“00”</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>“01”</td>
<td>-3</td>
<td>NT</td>
</tr>
<tr>
<td></td>
<td>“10”</td>
<td>(\times) -1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>“11”</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

branch history
1: 010100
2: 001001
3: 01

# pred.: \(\sqrt{13}\) 14
# mispred.: \(\sqrt{14}\) 5

predictor: 1
Data processing: amount of inherent ILP

**inherent ILP**

amount of instruction-level parallelism while assuming perfect caches, perfect branch prediction, etc.; only limiting factors are data dependencies and instruction window size

**INPUTS**
- memory read/write addresses
- register read/write ids

**OUTPUTS**
- amount of inherent ILP for various instruction window sizes (32, 64, 128, 256)

**for each instruction:**

- **register/memory read**
  - adjust issue time for this instr. according to time when reg./mem. block is available

- **register/memory write**
  - set time when reg./mem. block is available to current issue time + 1 clock cycle

- add instruction to tail of instruction window
  - if window is full:
    - increment clock time
    - commit instructions which are ready from head of instr. window
**Data processing: amount of inherent ILP**

**instruction stream**

- i1: read 0xDE; write r1
- i2: read r1; write r2
- i3: read 0xAD; write r3
- i4: read r2, r3; write 0xDE
- i5: read r1, r2; write r3

**clock:** 0 cycles  
**instr. count:** 1 instr.  
**issue time:** 0

**instruction window:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: 0</td>
<td></td>
</tr>
</tbody>
</table>

**register**

<table>
<thead>
<tr>
<th>Register</th>
<th>Time Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r2</td>
<td>0</td>
</tr>
<tr>
<td>r3</td>
<td>0</td>
</tr>
</tbody>
</table>

**mem. addr.**

<table>
<thead>
<tr>
<th>Address</th>
<th>Time Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xAD</td>
<td>0</td>
</tr>
<tr>
<td>0xDE</td>
<td>0</td>
</tr>
</tbody>
</table>
Data processing: amount of inherent ILP

Instruction stream:

- i1: read 0xDE; write r1
- i2: read r1; write r2
- i3: read 0xAD; write r3
- i4: read r2, r3; write 0xDE
- i5: read r1, r2; write r3

Clock: 0 cycles
Instruction count: 2 instr.
Issue time: 1

Instruction window:

| i1: 0 | i2: 1 |

Register availability:

<table>
<thead>
<tr>
<th>register</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r2</td>
<td>2</td>
</tr>
<tr>
<td>r3</td>
<td>0</td>
</tr>
</tbody>
</table>

Memory address availability:

<table>
<thead>
<tr>
<th>mem. addr.</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OxAD</td>
<td>0</td>
</tr>
<tr>
<td>OxDE</td>
<td>0</td>
</tr>
</tbody>
</table>
Data processing: amount of inherent ILP

Instruction stream:
- i1: read 0xDE; write r1
- i2: read r1; write r2
- i3: read 0xAD; write r3
- i4: read r2, r3; write 0xDE
- i5: read r1, r2; write r3

Clock:
- 0 cycles

Instruction count:
- 3 instr.

Issue time:
- 0

Instruction window:
- i1: 0
- i2: 1
- i3: 0

Register table:

<table>
<thead>
<tr>
<th>register</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r2</td>
<td>2</td>
</tr>
<tr>
<td>r3</td>
<td>1</td>
</tr>
</tbody>
</table>

Memory address table:

<table>
<thead>
<tr>
<th>mem. addr.</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xAD</td>
<td>0</td>
</tr>
<tr>
<td>0xDE</td>
<td>0</td>
</tr>
</tbody>
</table>


**Data processing: amount of inherent ILP**

Instruction stream:

- i1: read 0xDE; write r1
- i2: read r1; write r2
- i3: read 0xAD; write r3
- i4: read r2, r3; write 0xDE
- i5: read r1, r2; write r3

<table>
<thead>
<tr>
<th>register</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r2</td>
<td>2</td>
</tr>
<tr>
<td>r3</td>
<td>1</td>
</tr>
</tbody>
</table>

Memory address:

<table>
<thead>
<tr>
<th>mem. addr.</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OxAD</td>
<td>0</td>
</tr>
<tr>
<td>OxDE</td>
<td>0</td>
</tr>
</tbody>
</table>

Clock:

- 1 cycles

Instruction count:

- 3 instr.

Issue time:

- 0

Instruction window:

- i2: 1
- i3: 0
Data processing: amount of inherent ILP

Instruction stream:

- i1: read 0xDE; write r1
- i2: read r1; write r2
- i3: read 0xAD; write r3
- i4: read r2, r3; write 0xDE
- i5: read r1, r2; write r3

Clock: 1 cycles
Instruction count: 4 instr.
Issue time: 0

Instruction window:

<table>
<thead>
<tr>
<th>register</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r2</td>
<td>2</td>
</tr>
<tr>
<td>r3</td>
<td>1</td>
</tr>
</tbody>
</table>

Memory addresses:

<table>
<thead>
<tr>
<th>mem. addr.</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xAD</td>
<td>0</td>
</tr>
<tr>
<td>0xDE</td>
<td>3</td>
</tr>
</tbody>
</table>
Data processing: amount of inherent ILP

instruction stream

i1: read 0xDE; write r1
i2: read r1; write r2
i3: read 0xAD; write r3
i4: read r2, r3; write 0xDE
i5: read r1, r2; write r3

clock: 2 cycles
instr. count: 4 instr.
issue time: 0

instruction window:

<table>
<thead>
<tr>
<th>register</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r2</td>
<td>2</td>
</tr>
<tr>
<td>r3</td>
<td>1</td>
</tr>
</tbody>
</table>

mem. addr. | time avail.
------------|-------------|
0xAD        | 0           |
0xDE        | 3           |
Data processing: amount of inherent ILP

Instruction stream:

- i1: read 0xDE; write r1
- i2: read r1; write r2
- i3: read 0xAD; write r3
- i4: read r2, r3; write 0xDE
- i5: read r1, r2; write r3

Clock:
- 2 cycles

Instruction count:
- 5 instr.

Issue time:
- 2

Instruction window:

Register table:

<table>
<thead>
<tr>
<th>register</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r2</td>
<td>2</td>
</tr>
<tr>
<td>r3</td>
<td>3</td>
</tr>
</tbody>
</table>

Memory address table:

<table>
<thead>
<tr>
<th>mem. addr.</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xAD</td>
<td>0</td>
</tr>
<tr>
<td>0xDE</td>
<td>3</td>
</tr>
</tbody>
</table>
**Data processing: amount of inherent ILP**

Instruction stream:

- i1: read 0xDE; write r1
- i2: read r1; write r2
- i3: read 0xAD; write r3
- i4: read r2, r3; write 0xDE
- i5: read r1, 0xAD; write r2

**Clock:** 3 cycles
**Instruction count:** 5 instr.
**ILP:** 1.666

Instruction window:

<table>
<thead>
<tr>
<th>register</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r2</td>
<td>4</td>
</tr>
<tr>
<td>r3</td>
<td>1</td>
</tr>
</tbody>
</table>

Memory address:

<table>
<thead>
<tr>
<th>mem. addr.</th>
<th>time avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xAD</td>
<td>0</td>
</tr>
<tr>
<td>0xDE</td>
<td>3</td>
</tr>
</tbody>
</table>
Data processing: how long does it take?
Part 3: Insight!

Performance estimation

- how can we use benchmarks to learn something about our own application of interest?
- how do µarch.-indep. program characteristics relate to performance metrics?

Comparing benchmark suites

- how can we identify key program characteristics?
- how can we easily gain insight into inherent program behavior?

Future work
**Insight:** Performance estimation

What does benchmarking tell us?

Excel
(spreadsheet)

Photoshop
(image processing)

Virtual PC
(Windows on Mac)

R
(statistics)

MacBook Performance Benchmarks
SPEC performance: Up to five times faster than the iBook G4.(2)

<table>
<thead>
<tr>
<th></th>
<th>iBook G4 1.42GHz</th>
<th>MacBook Core Duo 2.0GHz</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECint_rate_base2000</td>
<td>5.7</td>
<td>29.1</td>
<td>5.1X</td>
</tr>
</tbody>
</table>

|= 4.3| 24.7 | 5.7X |

Insight: Performance estimation
Can we do the benchmarking ourselves?

✧ Porting

✧ Hardware availability

✧ Time constraints
**Insight: Performance estimation**

Estimate performance for application of interest

- Application of interest
- Benchmark space
- Benchmarks
Insight: Performance estimation

Performance estimation framework

- based on program similarity
- relate program characteristics to performance to scale benchmark space
- estimation allows finding the best machine for a given application
- more details, see
  - “Performance Prediction Based on Inherent Program Similarity” (PACT’06)
  - “Analyzing Commercial Processor Performance Numbers for Predicting Performance of Applications of Interest (SIGMETRICS’07)
**Insight: Comparing benchmark suites**

Comparing benchmarks is easy... right?

<table>
<thead>
<tr>
<th>Microarchitecture-dependent characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPI on Alpha 21164</strong></td>
</tr>
<tr>
<td>CPI on Alpha 21264</td>
</tr>
<tr>
<td>L1 D-cache misses per instruction</td>
</tr>
<tr>
<td>L1 I-cache misses per instruction</td>
</tr>
<tr>
<td>L2 cache misses per instruction</td>
</tr>
<tr>
<td>Microarchitecture-independent characteristics</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td><strong>data working set in 32-byte blocks</strong></td>
</tr>
<tr>
<td>data working set in 4KB pages</td>
</tr>
<tr>
<td>instr. memory footprint in 32-byte blocks</td>
</tr>
<tr>
<td>instr. memory footprint in 4KB pages</td>
</tr>
<tr>
<td>probability for a local load stride = 0</td>
</tr>
<tr>
<td>probability for a local store stride = 0</td>
</tr>
<tr>
<td>probability for a global load stride ≤ 64</td>
</tr>
<tr>
<td>probability for a global store stride ≤ 64</td>
</tr>
</tbody>
</table>
**Insight:** Comparing benchmark suites

Time is of the essence, insight is what we aim for

- Measuring microarchitecture-independent program characteristics takes a lot longer than collecting data using hardware performance counters…
- … but they give you a lot more insight into inherent program behavior!
- To close the gap regarding needed time:
  
  *identify key microarchitecture-independent program characteristics*

![Graph showing Pearson correlation coefficient vs. number of retained characteristics]
**Insight:** Comparing benchmark suites
Visualizing program behavior

- prob. register dep. dist ≤ 16
- prob. local load stride = 0
- prob. global load stride ≤ 8
- prob. local store stride ≤ 8
- prob. local store stride ≤ 4096

PAg PPM
% multiply operations
data mem. footprint (32-byte block level)

mean + std. dev.
mean - std. dev.

key characteristics reveal inherent program behavior
**Insight:** Comparing benchmark suites

Visualizing program behavior

inherent behavior might be very similar across inputs, somewhat different, or very different
Insight: Comparing benchmark suites
Visualizing program behavior

extreme behavior is easy to spot

more details, see

“Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics” (IISWC’06)
“Microarchitecture-Independent Workload Characterization” (IEEE Micro Hot Tutorials May/June 2007)
**Insight: Future work**

**What else do we have up our sleeve?**

- **phase-level performance estimation**
  collect program characteristics and IPCs for intervals of instructions, use machine learning to improve current methodology

- **comparing benchmarks with real applications**
  how different are commonly used applications from benchmarks used by academia?

- **study multithreaded applications**
  characterize multithreaded applications using thread-safe MICA (and additional characteristics?)

- **the next level: ISA-independent (LLVM?)**
Obtaining and using MICA

http://www.elis.ugent.be/~kehoste/mica

released under BSD license

do what you want with it, just don’t pretend it’s yours

updates and news: see website

only tested on Linux/x86

bug reports/fixes welcome (SVN coming soon)
BACKUP
**Data processing: register traffic**

**register dependency distance**
Distance (in number of dynamic instructions) between production of a register value and consumption of the same register value.

**Inputs**
- Instruction reg. op. cnt
- Register ids
- Read/write?

**Outputs**
- Average degree of use
- Average number of input reg. operands
- Probability dependency distance < D, \( D = 2^n, n = [0,6] \)

**Process**
- **All instructions**
  - Count number of register operands, add to total over all instructions
- **Register read**
  - Increase bucket counter for dependency distance
  - Increase degree of use for this register
- **Register write**
  - Set time stamp for register write
  - Add degree of use to total over all instructions

**Output**
- Determine dependency distance count < D
- Divide by total number of register dependencies (\( D = 2^n, n = [0,6] \))
Data processing: distr. of mem. acc. strides

**global memory stride**

difference in memory addresses between two consecutive memory accesses by any two instructions

**local memory stride**

difference in memory addresses between two consecutive memory accesses by the same static instruction

(measured separately for memory reads and writes)

**INPUTS**
- memory addresses
- read/write?

**OUTPUTS**
- probability memory stride < D, $D = 2^{3n}$, $n = [0,6]$
- both for local and global memory strides

for each memory read/write:

- compute difference with previous memory address (local/global)
- increase count for resulting memory stride

output:

- determine local/global memory read/write stride count < D
- divide by total number of memory reads/writes ($D = 2^{3n}$, $n = [0,6]$)
Data processing: touched blocks/pages

**memory footprint:** set of memory locations used by program (cache blocks or main memory pages)

**INPUTS**
- memory read/write addresses

**OUTPUTS**
- number of 32-byte blocks / 4KB pages touched by data/instr. mem. accesses

for each memory access (data/instr.):
- determine address for 32-byte block
- determine address for 4KB page
- set ‘touched’ bit in hash tables (mem. efficiency)

output:
- count number of blocks/pages touched by running over touched bits in hash tables