UNIVERSITEIT GENT

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?

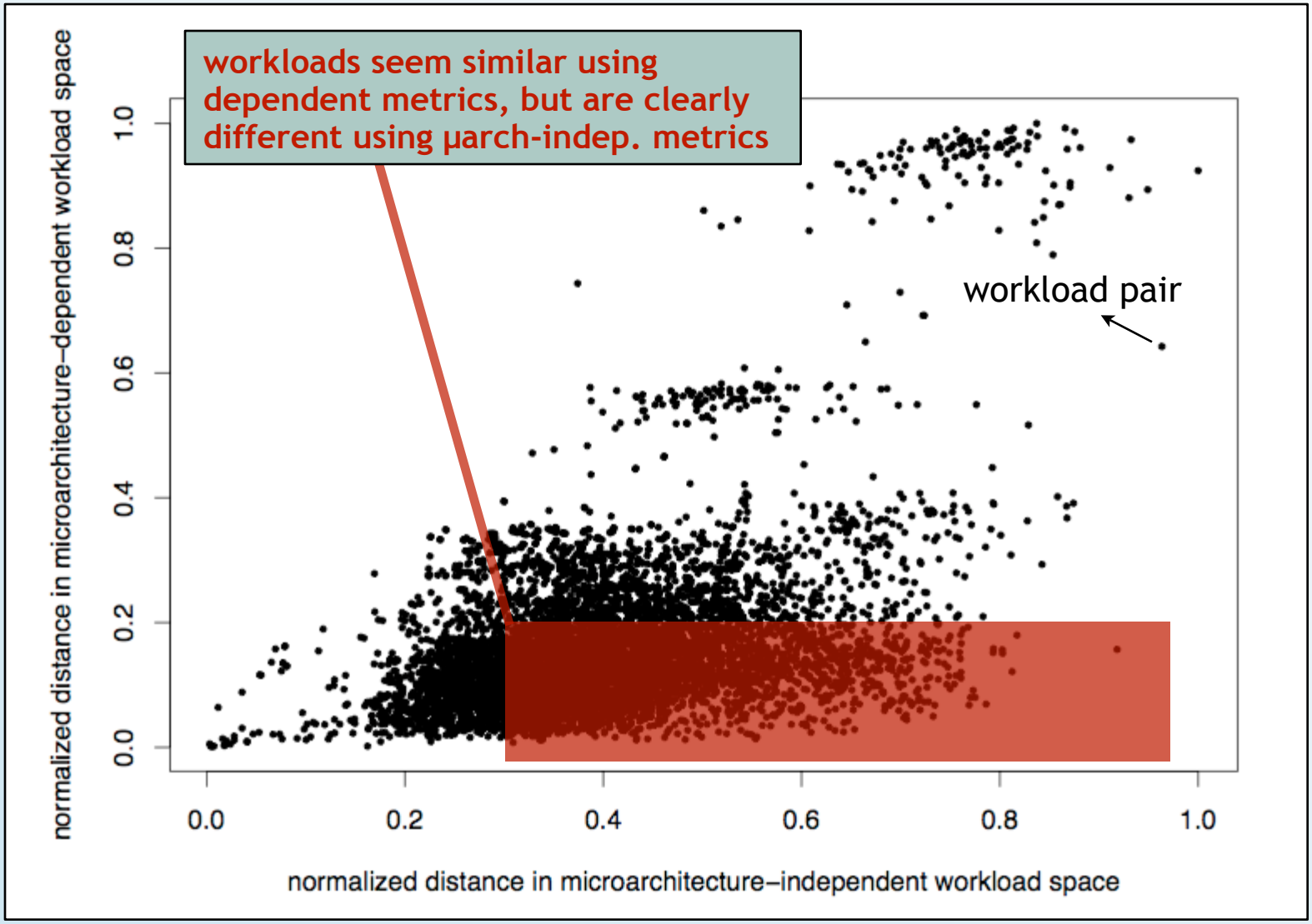
Prevalent workload characterization:

- 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



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*

Data extraction: instruction type

PROCESSING

```
INT64 cond_branches, muls;
VOID condBr_ins(){ cond_branches++; }
VOID mul_ins(){ muls++; }
```

INSTRUMENTATION

```
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);
}
```

OUTPUT

```
VOID Fini(INT32 code, VOID *v){
   fprintf(stderr,"%lld cond. branches, %lld muls\n", cond_branches, muls);
}
```

MAIN

Data extraction: branch ID

PROCESSING

```
UINT32 br_cnt; INT64* branch;
VOID br(UINT32 id){ branch[id]++; }
```

INSTRUMENTATION

```
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++;
    }
}
```

OUTPUT

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

MAIN

Data extraction: branch ID (corrected)

PROCESSING

```
UINT32 br_cnt; INT64* branch;
UINT32 lookup(ADDRINT a) { // find index for instr. address or create new }
VOID br(ADDRINT a){ UINT32 id = lookup(a); branch[id]++; }
```

INSTRUMENTATION

```
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);
    }
}
```

OUTPUT

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

MAIN

Data extraction: branch ID (faster)

PROCESSING

```
UINT32 br_cnt; INT64* branch;
UINT32 lookup(ADDRINT a) { // find index for instr. address or create new }
VOID br(UINT32 id){ branch[id]++; }
```

INSTRUMENTATION

```
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);
    }
}
```

OUTPUT

```
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]); }
}
```

MAIN

Data extraction: branch (not-)taken rate

PROCESSING

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

INSTRUMENTATION

```
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);
}
```

OUTPUT

```
VOID Fini(INT32 code, VOID *v){
    fprintf(stderr,"%lf taken, %lf not taken\n", (double)t/brCnt, (double)nt/brCnt);
}
```

MAIN

Data extraction: register reads/writes

PROCESSING

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

INSTRUMENTATION

```
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);
      }
   }
}
```

OUTPUT

```
VOID Fini(INT32 code, VOID *v){
   fprintf(stderr,"%lld register reads\n",reg_reads);
}
```

MAIN

Data extraction: register ID

PROCESSING

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

INSTRUMENTATION

```
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);
        }
    }
}
```

OUTPUT

```
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]); }
}
```

MAIN

Data extraction: memory reads/writes

PROCESSING

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

INSTRUMENTATION

```
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);
   }
}
```

OUTPUT

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

MAIN

Data extraction: memory addresses

PROCESSING

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

INSTRUMENTATION

```
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);
}
```

OUTPUT

```
VOID Fini(INT32 code, VOID *v){ }
```

MAIN

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

INPUTS

- branch id
- taken/not taken?

OUTPUTS

- misprediction rate using:
  - per-address/global predictor tables with global/local history
  - different history lengths (4/8/12)

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

1	0	1					0	1					***0***	0
2			0	0	1				0	0	1			
3						0						1		

branch history
1: 01010
2: 001001
3: 01

PAg (per-address history, global table)

hist. length	history	value	pred.
0	-	-2	NT
1	“0”	3	T
	“1”	-1	NT
2	“00”	2	T
	“01”	~~2~~ -3	NT
	“10”	0	-
	“11”	0	-

pred.: ~~12~~ 13
mispred.: 4

predictor: 2

Data processing: PPM-predictors

1	0	1					0	1					0	***0***
2			0	0	1				0	0	1			
3						0						1		

PAg (per-address history, global table)

hist. length	history	value	pred.
0	-	-2	NT
1	“0”	~~3~~ 2	T
	“1”	-1	NT
2	“00”	2	T
	“01”	-3	NT
	“10”	~~0~~ -1	-
	“11”	0	-

branch history
1: 010100
2: 001001
3: 01

pred.: ~~13~~ 14
mispred.: ~~4~~ 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:

i1: 0		

register	time avail.
r1	1
r2	0
r3	0

mem. addr.	time avail.
OxAD	0
OxDE	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: 0 cycles
instr. count: 2 instr.
issue time: 1

instruction window:

i1: 0	i2: 1	

register	time avail.
r1	1
r2	2
r3	0

mem. addr.	time avail.
OxAD	0
OxDE	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: 0 cycles
instr. count: 3 instr.
issue time: 0

instruction window:

i1: 0	i2: 1	i3: 0

register	time avail.
r1	1
r2	2
r3	1

mem. addr.	time avail.
OxAD	0
OxDE	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
instr. count: 3 instr.
issue time: 0

instruction window:

i2: 1	i3: 0	

register	time avail.
r1	1
r2	2
r3	1

mem. addr.	time avail.
OxAD	0
OxDE	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
instr. count: 4 instr.
issue time: 0

instruction window:

i2: 1	i3: 0	i4: 2

register	time avail.
r1	1
r2	2
r3	1

mem. addr.	time avail.
OxAD	0
OxDE	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
instr. count: 4 instr.
issue time: 0

instruction window:

i4: 2		

register	time avail.
r1	1
r2	2
r3	1

mem. addr.	time avail.
OxAD	0
OxDE	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
instr. count: 5 instr.
issue time: 2

instruction window:

i4: 2	i5: 2	

register	time avail.
r1	1
r2	2
r3	3

mem. addr.	time avail.
OxAD	0
OxDE	3

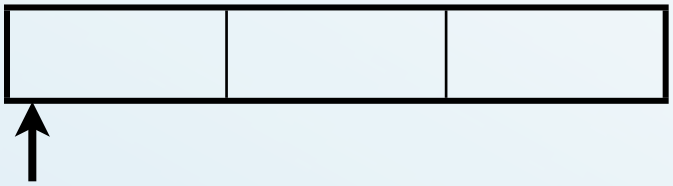
Data processing: amount of inherent ILP

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, 0xAD; write r2~~

clock: 3 cycles
instr. count: 5 instr.
ILP: 1.666

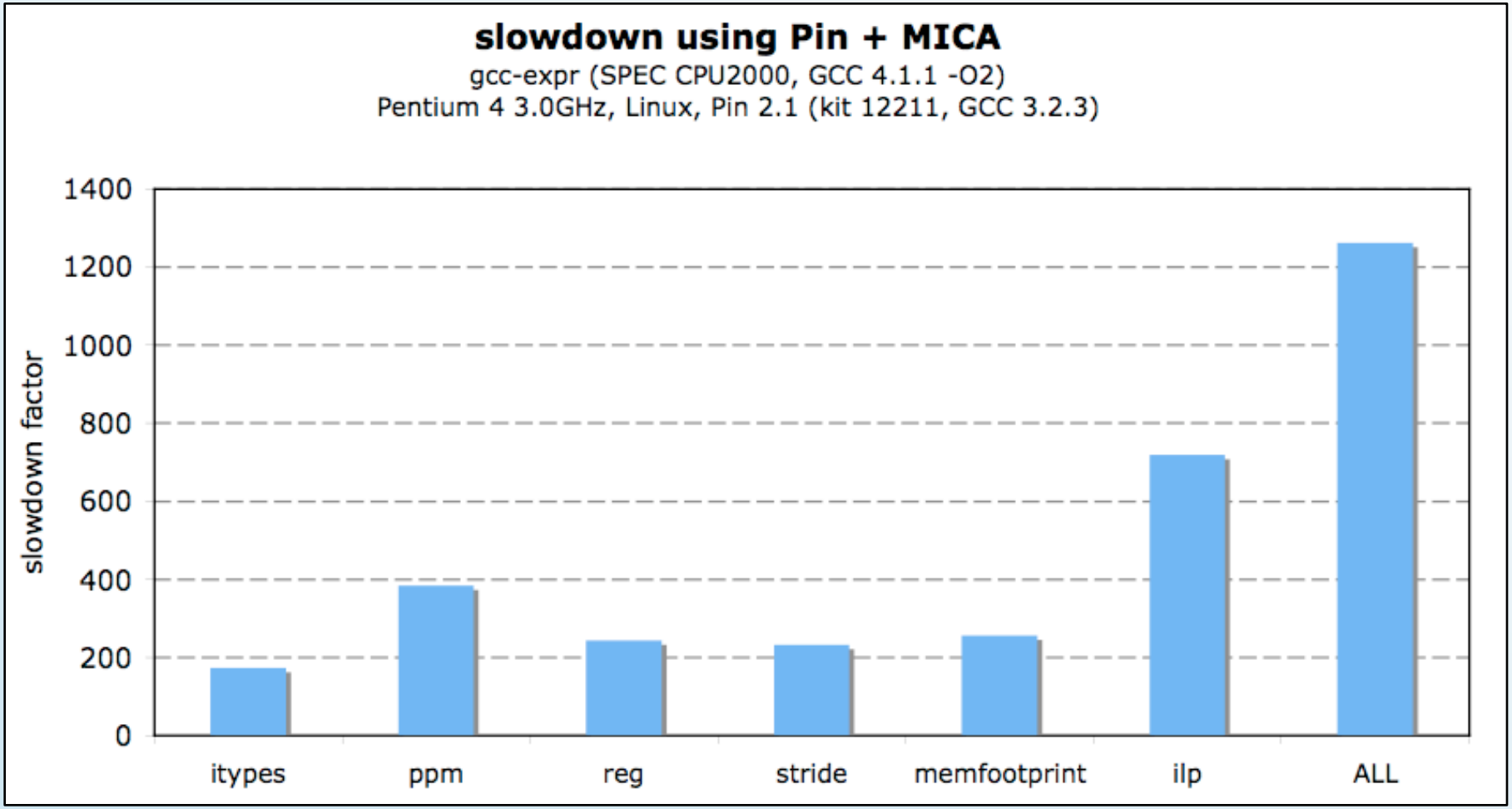
instruction window:



register	time avail.
r1	1
r2	4
r3	1

mem. addr.	time avail.
OxAD	0
OxDE	3

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

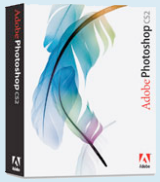
Insight: Performance estimation

What does benchmarking tell us?

Excel
(spreadsheet)

R
(statistics)

Virtual PC
(Windows on Mac)



Photoshop
(image processing)

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

iBook G4 1.42GHz	MacBook Core Duo 2.0GHz	Δ
5.7	29.1	5.1X
SPECint_rate_base2000 Integer calculation (estimate)		

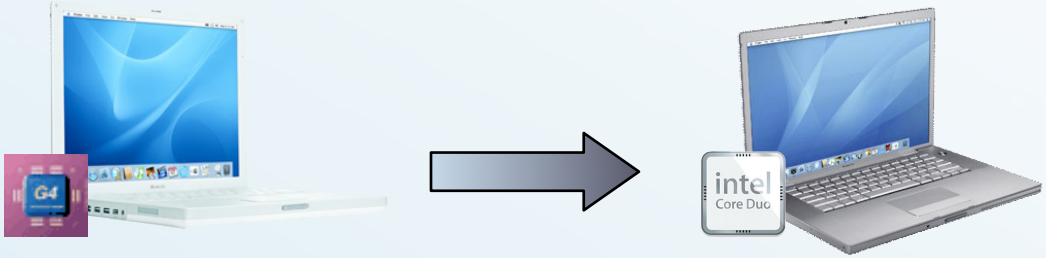
iBook G4 1.42GHz	MacBook Core Duo 2.0GHz	Δ
4.3	24.7	5.7X
SPECfp_rate_base2000 Floating-point calculation (estimate)		

http://www.apple.com/macbook/intelcoreduo.html, May 2006

Insight: Performance estimation

Can we do the benchmarking ourselves?

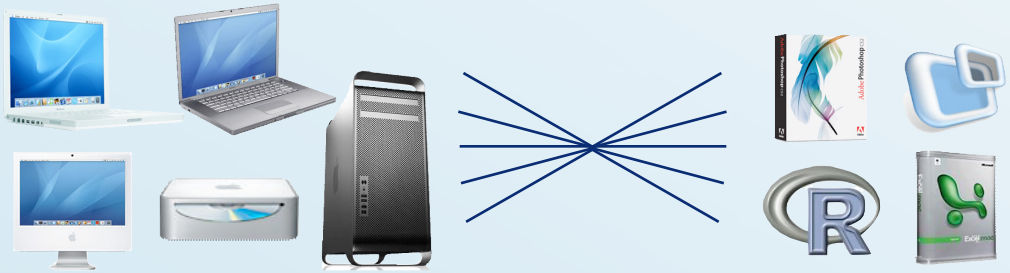
- Porting



- Hardware availability

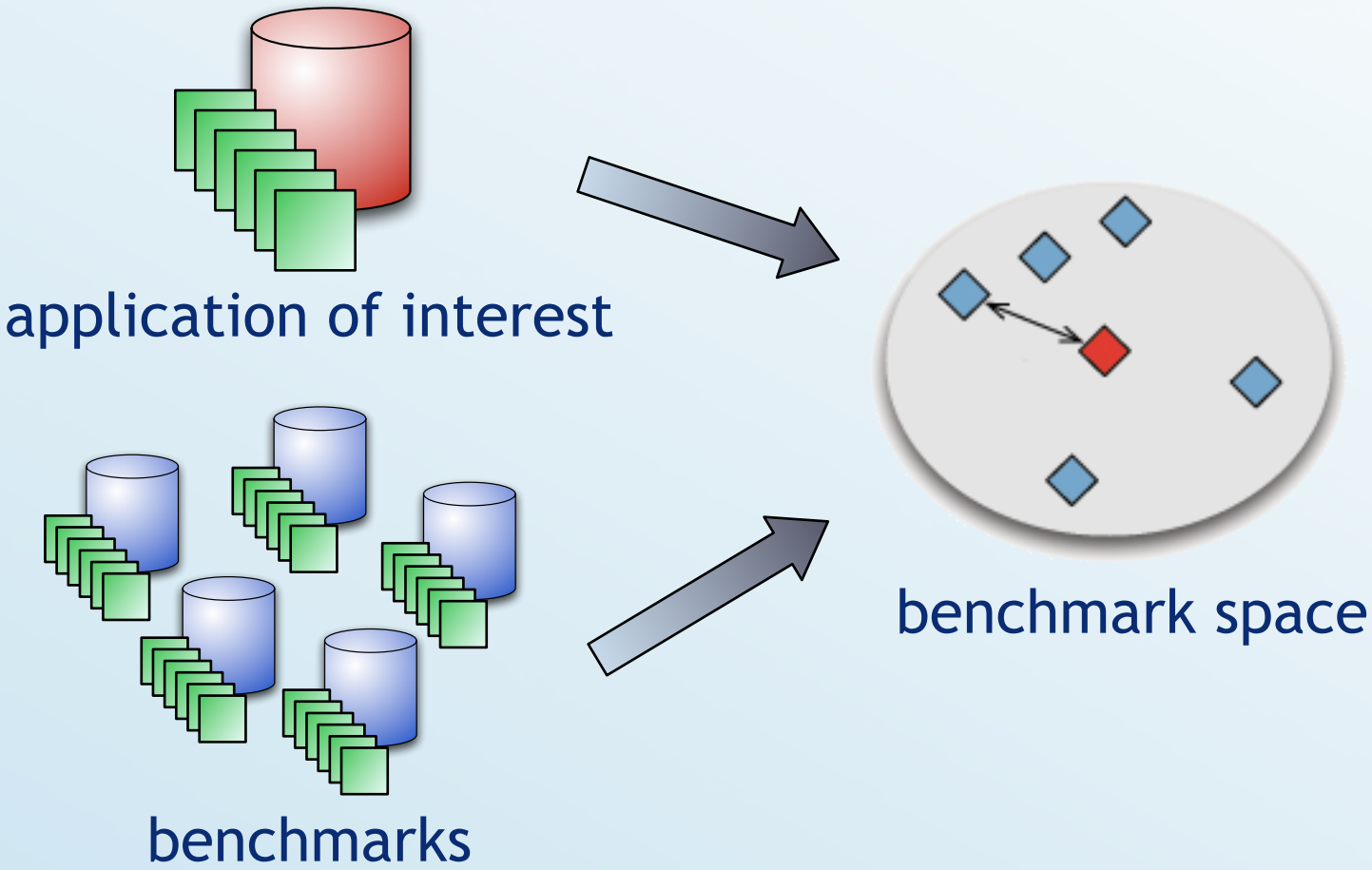
Playstation 3

- Time constraints



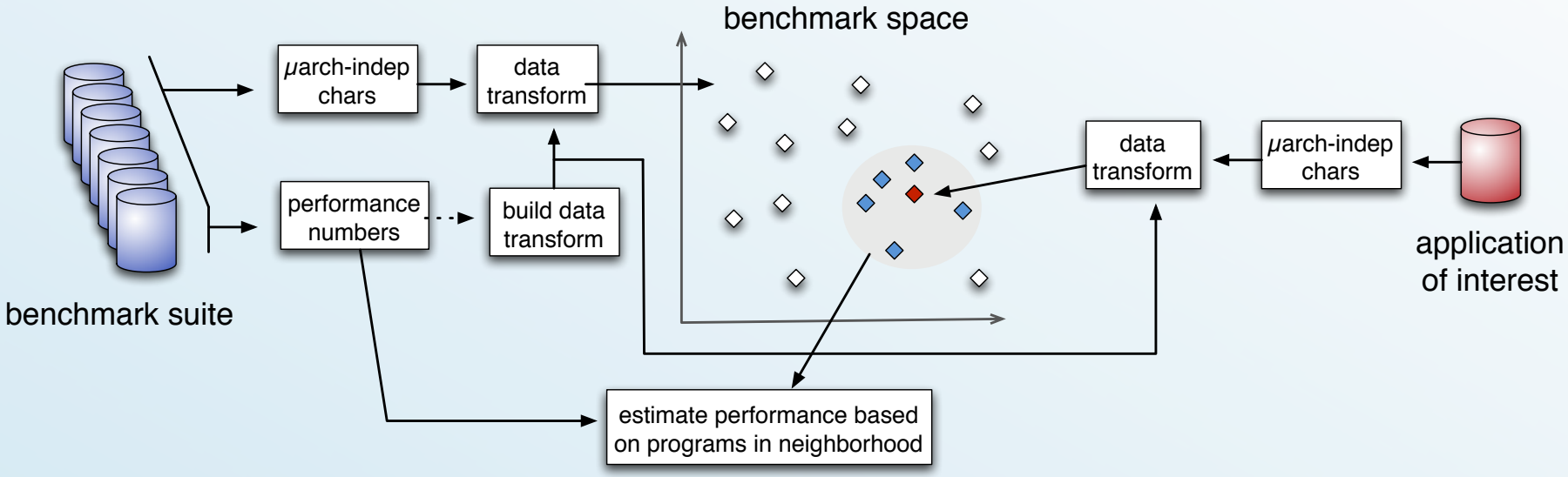
Insight: Performance estimation

Estimate performance for application of interest



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?

Microarchitecture-dependent characteristics			
	gzip	fasta	maximum
CPI on Alpha 21164	1.01	0.92	14.04
CPI on Alpha 21264	0.63	0.66	5.22
L1 D-cache misses per instruction	1.61%	1.90%	22.58%
L1 I-cache misses per instruction	0.15%	0.18%	6.44%
L2 cache misses per instruction	0.78%	0.25%	17.59%

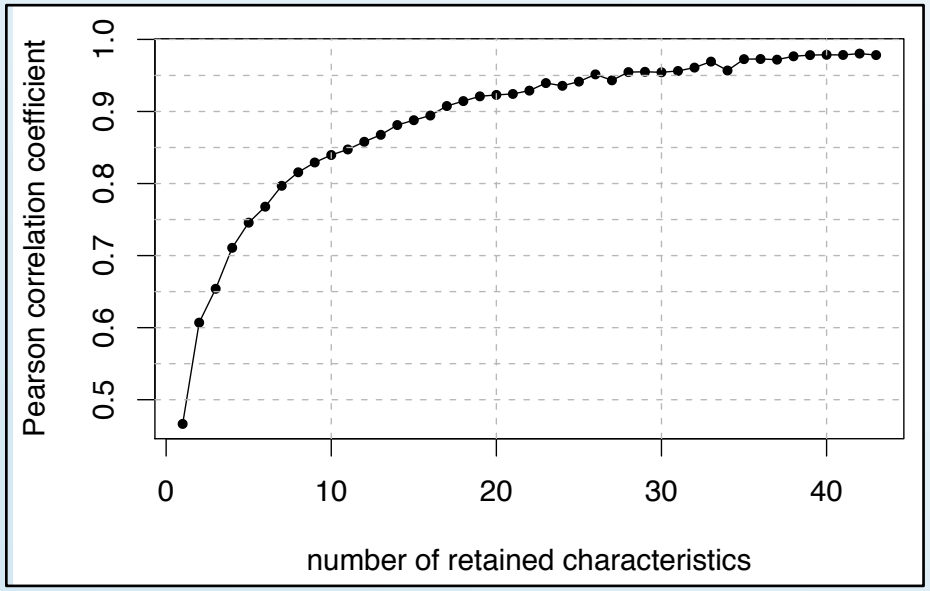
Microarchitecture-independent characteristics			
	gzip	fasta	maximum
data working set in 32-byte blocks	3,857,693	438,726	31,709,065
data working set in 4KB pages	46,199	4,058	248,108
instr. memory footprint in 32-byte blocks	1,394	3,801	24,377
instr. memory footprint in 4KB pages	33	79	341
probability for a local load stride = 0	0.67	0.30	0.91
probability for a local store stride = 0	0.64	0.05	0.99
probability for a global load stride $\leq$ 64	0.26	0.18	0.86
probability for a global store stride $\leq$ 64	0.35	0.93	0.99

Insight: Comparing benchmark suites

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

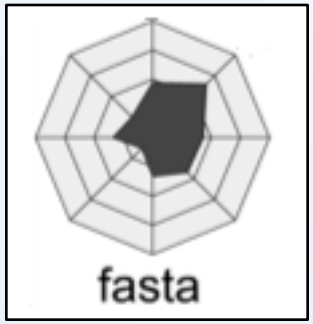
- Measuring microarchitecture-independent program characteristics takes a lot longer than collecting data using hardware performance counters...
- ... but they give you a lot more insight into inherent program behavior!
- To close the gap regarding needed time:

  identify key microarchitecture-independent program characteristics

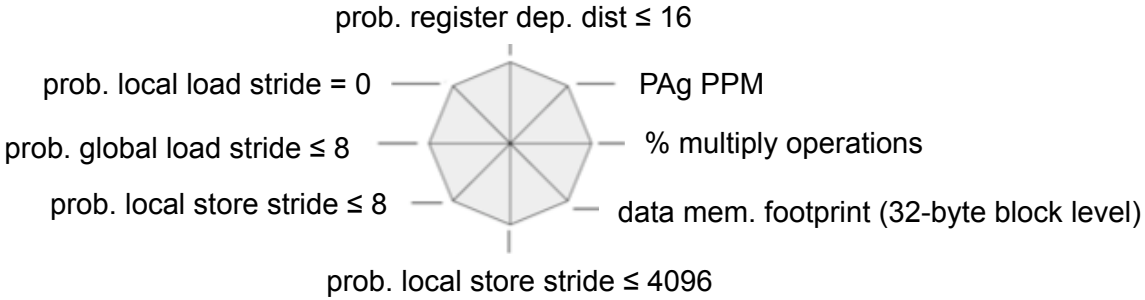


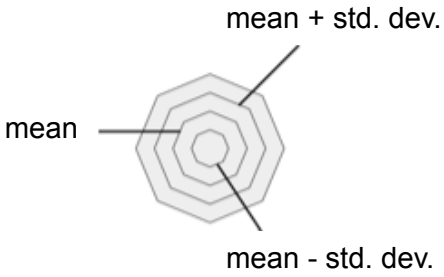
Insight: Comparing benchmark suites

Visualizing program behavior



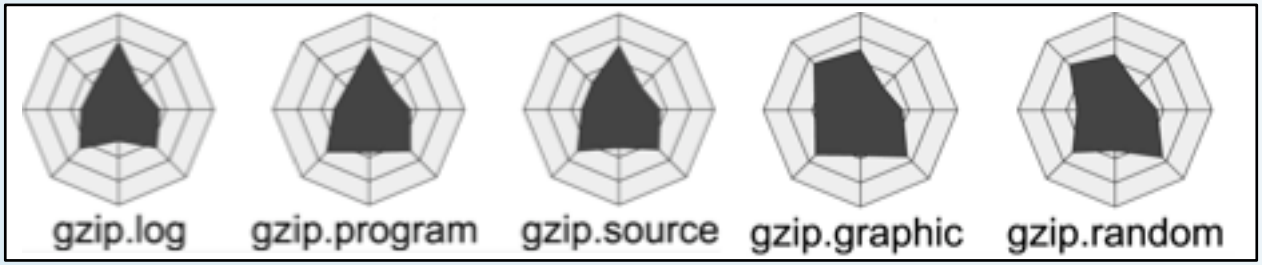
key characteristics reveal inherent program behavior

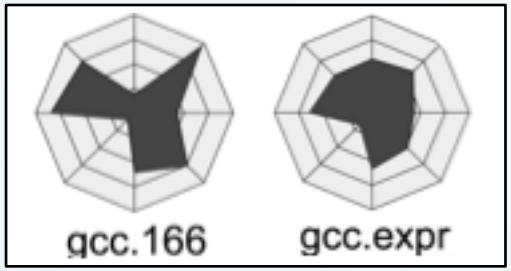




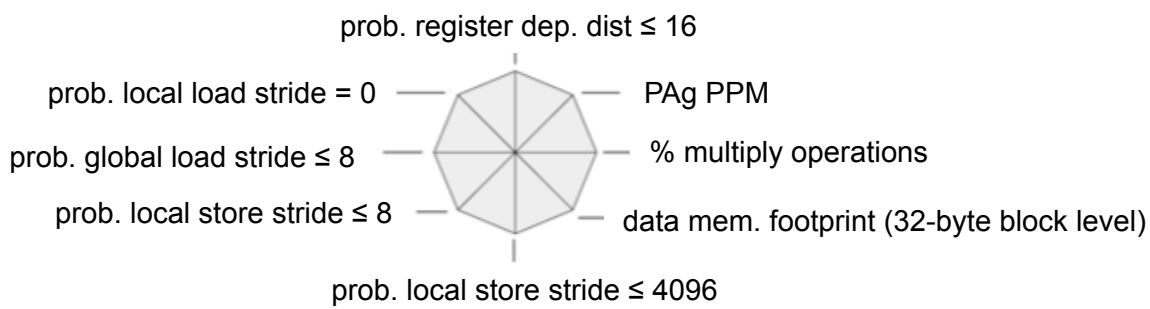
Insight: Comparing benchmark suites

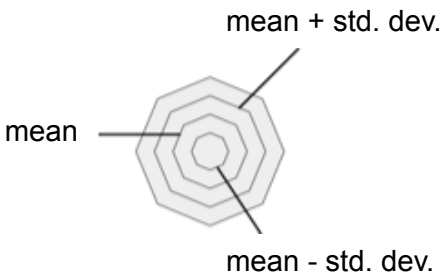
Visualizing program behavior





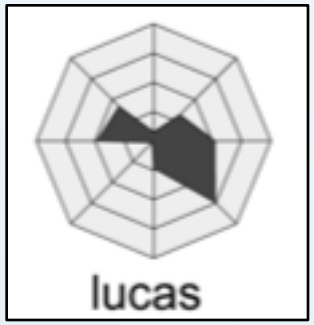
inherent behavior might be very similar across inputs,
somewhat different,
or very different

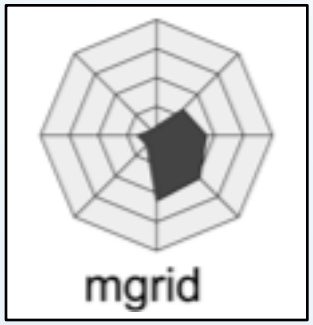




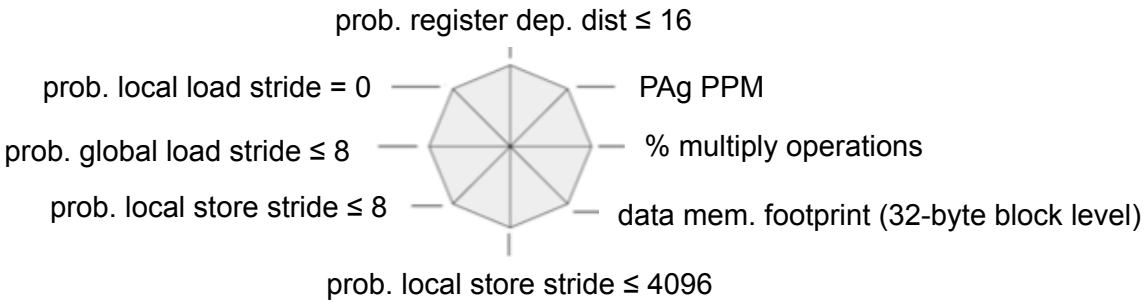
Insight: Comparing benchmark suites

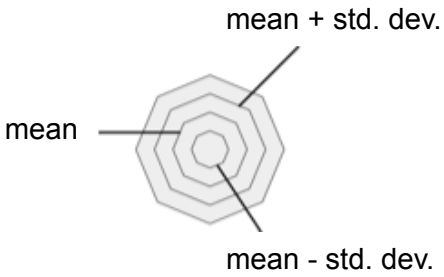
Visualizing program behavior





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`

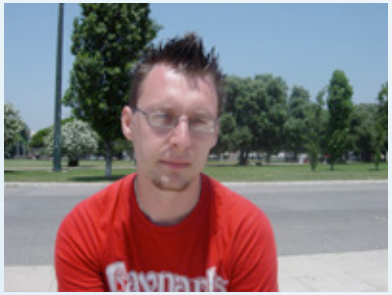
- released under BSD license

  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)



Kenneth Hoste (kehoste@elis.ugent.be)

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

?

Lieven Eeckhout (leeckhou@elis.ugent.be)

http://www.elis.ugent.be/~leeckhou

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, $D = 2^n$, $n = [0,6]$

- 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