COSC 340: Software Engineering

Debugging with Dynamic Binary Analysis & Instrumentation

Michael Jantz
Program Analysis

• Programmers often use *analysis tools* to improve program quality

• Static Analysis
  – Analyze program source code or machine code without running it
  – Often performed by compilers (type checking, optimizations)

• Dynamic Analysis
  – Analyze a client program as it runs
  – Profilers, error checkers, execution visualizers
  – DA instruments the client program with analysis code
  – May change depending on program input
Source Analysis vs. Binary Analysis

• Source Analysis
  – Analyze program at level of source code
  – Generally performed in terms of programming language constructs (functions, statements, expressions, variables)
  – Example: control-flow graph

• Binary Analysis
  – Analyze program at the level of machine code: object code (pre-linking) or executable code (post-linking)
  – Includes executable intermediate representations (bytecodes)
  – Generally performed in terms of machine entities (procedures, instructions, registers, memory locations)
## Analysis Properties

<table>
<thead>
<tr>
<th>Source</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Considers all execution paths&lt;br&gt;Platform independent&lt;br&gt;Access to high-level information (functions, expressions, variables)</td>
<td>Considers run-time information, but only for one execution path&lt;br&gt;Platform independent&lt;br&gt;Access to high-level information (functions, expressions, variables)</td>
</tr>
<tr>
<td>Binary</td>
<td>Considers all execution paths&lt;br&gt;Does not require source&lt;br&gt;Access to low-level information (registers, memory locations)</td>
<td>Considers run-time information, but only for one execution path&lt;br&gt;Does not require source&lt;br&gt;Access to low-level information (registers, memory locations)</td>
</tr>
</tbody>
</table>
## Analysis Properties

<table>
<thead>
<tr>
<th>Source</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Considers all execution paths</td>
<td>Considers run-time information, but only for one execution path</td>
</tr>
<tr>
<td></td>
<td>Platform independent</td>
<td>Platform independent</td>
</tr>
<tr>
<td></td>
<td>Access to high-level information (functions, expressions, variables)</td>
<td>Access to high-level information (functions, expressions, variables)</td>
</tr>
<tr>
<td>Binary</td>
<td>Considers all execution paths</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Does not require source code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Access to low-level information (registers, memory locations)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instrumentation for Dynamic Binary Analysis

• Static Binary Instrumentation:
  – Before program is run, rewrite the binary

• Dynamic Binary Instrumentation
  – Code is injected into the client process at runtime (either by a program that is grafted onto the client, or by an external process)

• Focus on DBI:
  – Advantages:
    • Does not require preparation of the client program
    • Easy to naturally cover all the reached code
  – Disadvantages
    • Cost of instrumentation incurred at run-time
    • May be difficult to implement
Valgrind

• DBI framework for building *heavyweight* analysis
  – Every instruction is instrumented
  – Tools track a lot of metadata (e.g. every register is memory value is shadowed by a metavalue)

• Valgrind Tools
  – MemCheck
  – Cachegrind
  – Massif
MemCheck

• Memory error detector for C and C++ programs
• Can help detect the following problems:
  – Accessing memory you shouldn't, e.g. overrunning heap blocks, overrunning
    the top of the stack, and accessing memory after it has been freed.
  – Using undefined values
  – Double-freeing heap blocks
  – Overlapping \textit{src} and \textit{dst} in \textit{memcpy} and related functions
  – Passing negative values as the size to \texttt{malloc}
  – Memory leaks
MemCheck Example

```c
#include <stdlib.h>
#include <stdio.h>

void f(void)
{
    int* x = malloc(10 * sizeof(int));
    x[10] = 0; // problem 1: heap block overrun
    // problem 2: memory leak -- x not freed
}

int main(void)
{
    int y;
    printf("y = %d\n", y); // problem 3: y used before initialized
    f();
    return 0;
}
```
MemCheck Example

• Running MemCheck
valgrind --leak-check=full ./mc
==27967== Memcheck, a memory error detector
==27967== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==27967== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==27967== Command: ./mc

• Error seen when MemCheck detects an uninitialized value
==27967== Conditional jump or move depends on uninitialised value(s)
==27967== at 0x4E7CE12: vfprintf (in /usr/lib64/libc-2.17.so)
==27967== by 0x4E86C98: printf (in /usr/lib64/libc-2.17.so)
==27967== by 0x4005C1: main (mc.c:13)
MemCheck Example

• Error seen when it detects a write to an invalid address
  ==27967== Invalid write of size 4
  ==27967==   at 0x40059E: f (mc.c:7)
  ==27967==   by 0x4005C6: main (mc.c:15)

• Error seen when it detects a memory leak
  ==27967== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
  ==27967==   at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
  ==27967==   by 0x400591: f (mc.c:6)
  ==27967==   by 0x4005C6: main (mc.c:15)
  ==27967==
CacheGrind

• Simulates how your program interacts with the cache hierarchy and branch predictor
  – Includes simulation for first-level instruction and data caches (I1 and D1) and the last level (LL) cache
  – Does not simulate caches between first and last level
CacheGrind

• Collects the following:
  – I cache reads (Ir which is the same as the instructions executed), I1 cache read misses (I1mr) and LL cache reads and misses (ILmr)
  – D cache reads (Dr, which equals the number of memory reads), D1 cache read misses (D1mr) and LL cache data read misses (DLmr)
  – D cache writes (Dw, which equals the number of memory writes), D1 cache write misses (D1mw) and LL cache data write misses (DLmw)
  – Conditional branches executed (Sc) and conditional branches mispredicted (Scm)
  – Indirect branches executed (Bi) and indirect branches mispredicted (Bim)
Cachegrind Example: Matrix Multiply

class MM {
    public:
        vector < vector <double> > M1;
        vector < vector <double> > M2;
        vector < vector <double> > P;
        int Print;
        void Multiply();
        void PrintAll();
};

void MM::Multiply()
{
    int i, j, k;

    for (i = 0; i < P.size(); i++) {
        for (j = 0; j < P[0].size(); j++) {
            for (k = 0; k < M2.size(); k++) P[i][j] += (M1[i][k] * M2[k][j]);
        }
    }
}
Cachegrind Example: Matrix Multiply

- Solution: store the second matrix with rows as columns / columns as rows (i.e. store its transpose)

```cpp
void MM::Multiply()
{
    int i, j, k;

    for (i = 0; i < P.size(); i++) {
        for (j = 0; j < P[0].size(); j++) {
            for (k = 0; k < M1[0].size(); k++) P[i][j] += (M1[i][k] * M2[j][k]);
            // This is the change: ^^^^^
        }
    }
}
```
Cachegrind Example: Matrix Multiply

[~/teaching/matrix]
mrjantz@com1643$ valgrind --tool=cachegrind ./mm-plain 600 600 600 1 n
==28382== Cachegrind, a cache and branch-prediction profiler
==28382== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==28382== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==28382== Command: ./mm-plain 600 600 600 1 n
==28382== warning: L3 cache found, using its data for the LL simulation.
Time: 11.8845
==28382==    I refs:  2,441,613,670
==28382==    I1 misses:  1,674
==28382==    L1 misses:  1,662
==28382==    I1 miss rate:  0.00%
==28382==    L1 miss rate:  0.00%
==28382==    D refs:  889,692,360 (665,023,032 rd + 224,669,328 wr)
==28382==    D1 misses:  325,995,459 (325,767,145 rd + 228,314 wr)
==28382==    LLd misses:  317,352 ( 89,647 rd + 227,705 wr)
==28382==    D1 miss rate:  36.6% ( 48.9% +  0.1% )
==28382==    LLd miss rate:  0.0% ( 0.0% +  0.1% )
==28382==    LL refs:  325,997,133 (325,768,819 rd + 228,314 wr)
==28382==    LL misses:  319,014 ( 91,309 rd + 227,705 wr)
==28382==    LL miss rate:  0.0% ( 0.0% +  0.1% )
Cacheigrind Example: Matrix Multiply

[~/teaching/matrix]
mrzjantz@com1643$ valgrind --tool=cacheigrind ./mm-transpose 600 600 600 1 n
==28384== Cacheigrind, a cache and branch-prediction profiler
==28384== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==28384== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==28384== Command: ./mm-transpose 600 600 600 1 n
==28384==
==28384== warning: L3 cache found, using its data for the LL simulation.
Time: 7.0942
==28384==
==28384== I refs: 2,010,691,576
==28384== I1 misses: 1,652
==28384== LLI misses: 1,638
==28384== I1 miss rate: 0.00%
==28384== LLI miss rate: 0.00%
==28384==
==28384== D refs: 674,051,405 (449,382,209 rd + 224,669,196 wr)
==28384== D1 misses: 28,008,372 (27,465,213 rd + 543,159 wr)
==28384== LLd misses: 314,211 (89,619 rd + 224,592 wr)
==28384== D1 miss rate: 4.1% (6.1% + 0.2%)
==28384== LLd miss rate: 0.0% (0.0% + 0.0%)
==28384==
==28384== LL refs: 28,010,024 (27,466,865 rd + 543,159 wr)
==28384== LL misses: 315,849 (91,257 rd + 224,592 wr)
==28384== LL miss rate: 0.0% (0.0% + 0.0%)
Massif: Heap Profiler

• Measures how much heap space your program uses
  – Useful space and extra bytes for alignment
  – Can also measure stack space (but does not do so by default)

• Collects 'snapshots' of memory usage at certain point in time

• Also tells you how much heap data is allocated at each allocation site
Massif Example

```c
#include <stdlib.h>

void g(void)
{
    malloc(4000);
}

void f(void)
{
    malloc(2000);
    g();
}

int main(void)
{
    int i;
    int a[10];
    for (i = 0; i < 10; i++) {
        a[i] = malloc(1000);
    }
    f();
    g();
    for (i = 0; i < 10; i++) {
        free(a[i]);
    }
    return 0;
}
```
Massif Example

• Run massif:
  – > valgrind --tool=massif ./prog

• View information collected by massif
  – > ms_print massif.out.PID

• By default, snapshots measured in instructions executed
  – For short-running programs, use --time-unit=B to measure snapshots in bytes allocated
Massif Output Graph

Number of snapshots: 25
Detailed snapshots: [9, 14 (peak), 24]
Massif Output Graph

```
[~/teaching/massif]
majantz@com16435 ms_print massif.out.27329
Command: ./pro
Massif arguments: --time-unit=8
ms_print arguments: massif.out.27329
```

Number of snapshots: 25
Detailed snapshots: [9, 14 (peak), 24]
Snapshot Details

Number of snapshots: 25
Detailed snapshots: [9, 14 (peak), 24]

<table>
<thead>
<tr>
<th>n</th>
<th>time(B)</th>
<th>total(B)</th>
<th>useful-heap(B)</th>
<th>extra-heap(B)</th>
<th>stacks(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1,016</td>
<td>1,016</td>
<td>1,000</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2,032</td>
<td>2,032</td>
<td>2,000</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3,048</td>
<td>3,048</td>
<td>3,000</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4,064</td>
<td>4,064</td>
<td>4,000</td>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5,080</td>
<td>5,080</td>
<td>5,000</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6,096</td>
<td>6,096</td>
<td>6,000</td>
<td>96</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7,112</td>
<td>7,112</td>
<td>7,000</td>
<td>112</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8,128</td>
<td>8,128</td>
<td>8,000</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>9,144</td>
<td>9,144</td>
<td>9,000</td>
<td>144</td>
<td>0</td>
</tr>
</tbody>
</table>

98.43% (9,000B) (heap allocation functions) malloc/new/new[], -alloc-fns, etc.

99.09% (20,000B) (heap allocation functions) malloc/new/new[], -alloc-fns, etc.

<table>
<thead>
<tr>
<th>n</th>
<th>time(B)</th>
<th>total(B)</th>
<th>useful-heap(B)</th>
<th>extra-heap(B)</th>
<th>stacks(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10,160</td>
<td>10,160</td>
<td>10,000</td>
<td>160</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>12,168</td>
<td>12,168</td>
<td>12,000</td>
<td>168</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>16,176</td>
<td>16,176</td>
<td>16,000</td>
<td>178</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>20,184</td>
<td>20,184</td>
<td>20,000</td>
<td>184</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>20,184</td>
<td>20,184</td>
<td>20,000</td>
<td>184</td>
<td>0</td>
</tr>
</tbody>
</table>

99.54% (10,000B) 0x4005BE: main (prog.c:20)

| ->39.6% (8,000B) 0x40058c: g (prog.c:5) |
| ->19.82% (4,000B) 0x4005a1: f (prog.c:11) |
| ->19.82% (4,000B) 0x4005da: main (prog.c:23) |
| ->19.82% (4,000B) 0x4005df: main (prog.c:25) |
| ->09.91% (2,000B) 0x40059c: f (prog.c:10) |
| ->09.91% (2,000B) 0x4005da: main (prog.c:23) |
Intel Pin

• Another tool for dynamic binary instrumentation
• Allows users to write their own DBI tools
  – Insert 'arbitrary' code in 'arbitrary' places in the executable
  – Uses *dynamic compilation* to instrument executables as they are running
  – Provides a platform-independent API for doing useful things with DBI
• Some example tools written in Pin
  – Memory tracers
  – Call site profilers
  – Cache simulators
Pin's Software Architecture
Pin Tool for Tracing Memory Writes

FILE * trace;

// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr, UINT32 size) {
    fprintf(trace,"%p: W %p %d\n", ip, addr, size);
}

// Called for every instruction
VOID Instruction(INS ins, VOID *v) {
    // instruments writes using a predicated call,
    // i.e. the call happens iff the store is
    // actually executed
    if (INS_IsMemoryWrite(ins))
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, AFUNPTR(RecordMemWrite),
            IARG_INST_PTR, IARG_MEMORYWRITE_SIZE, IARG_END);
}

int main(int argc, char *argv[]) {
    PIN_Init(argc, argv);
    trace = fopen("atrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram(); // Never returns
    return 0;
}
Output of pinatrace (Memory Tracing Tool)

1 0x7f39f273c433: W 0x7ffdd0d97a208
2 0x7f39f273f940: W 0x7ffdd0d97a200
3 0x7f39f273f944: W 0x7ffdd0d97a1f8
4 0x7f39f273f946: W 0x7ffdd0d97a1f0
5 0x7f39f273f948: W 0x7ffdd0d97a1e8
6 0x7f39f273f94a: W 0x7ffdd0d97a1e0
7 0x7f39f273f94f: W 0x7ffdd0d97a1d8
8 0x7f39f273f95f: R 0x7f39f295cdf0
9 0x7f39f273f966: W 0x7f39f295cc18
10 0x7f39f273f977: R 0x7f39f295cfa0
11 0x7f39f273f981: W 0x7f39f295d9a8
12 0x7f39f273f988: W 0x7f39f295d998
13 0x7f39f273f9df: W 0x7f39f295da48
14 0x7f39f273f9e7: R 0x7f39f295ce00
15 0x7f39f273f9e9: W 0x7f39f295da00
16 0x7f39f273f9e2: R 0x7f39f295ce10
17 0x7f39f273f9f0: W 0x7f39f295dab0
18 0x7f39f273f9e7: R 0x7f39f295ce20
19 0x7f39f273f9e6: W 0x7f39f295d9f8
20 0x7f39f273f9e7: R 0x7f39f295ce30
21 0x7f39f273fc33: W 0x7f39f295d3c0
22 0x7f39f273f9e7: R 0x7f39f295ce40
23 0x7f39f273f9e6: W 0x7f39f295da00
24 0x7f39f273f9e7: R 0x7f39f295ce50
25 0x7f39f273f9e6: W 0x7f39f295da08
26 0x7f39f273f9e7: R 0x7f39f295ce60
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Image</th>
<th>Address</th>
<th>Calls</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BZ2_hbCreateDecodeTables</td>
<td>bzip2</td>
<td>408590</td>
<td>66</td>
<td>972564</td>
</tr>
<tr>
<td>BZ2_hbAssign Codes</td>
<td>bzip2</td>
<td>408540</td>
<td>66</td>
<td>840460</td>
</tr>
<tr>
<td>BZ2_hbMakeCodeLengths</td>
<td>bzip2</td>
<td>408090</td>
<td>264</td>
<td>26536685</td>
</tr>
<tr>
<td>BZ2_decompress</td>
<td>bzip2</td>
<td>4088F0</td>
<td>738</td>
<td>1348674514</td>
</tr>
<tr>
<td>BZ2_compressBlock</td>
<td>bzip2</td>
<td>4066F0</td>
<td>11</td>
<td>2340829139</td>
</tr>
<tr>
<td>bPutbInt32</td>
<td>bzip2</td>
<td>406670</td>
<td>14</td>
<td>350</td>
</tr>
<tr>
<td>bswo</td>
<td>bzip2</td>
<td>4065E0</td>
<td>5460796</td>
<td>139696616</td>
</tr>
<tr>
<td>BZ2_bzReadGetUnused</td>
<td>bzip2</td>
<td>406180</td>
<td>3</td>
<td>51</td>
</tr>
<tr>
<td>BZ2_bzRead</td>
<td>bzip2</td>
<td>405F30</td>
<td>1260</td>
<td>113190</td>
</tr>
<tr>
<td>BZ2_bzReadClose</td>
<td>bzip2</td>
<td>405e00</td>
<td>3</td>
<td>54</td>
</tr>
<tr>
<td>BZ2_bzReadOpen</td>
<td>bzip2</td>
<td>405b40</td>
<td>3</td>
<td>189</td>
</tr>
<tr>
<td>BZ2_bzWriteClose64</td>
<td>bzip2</td>
<td>405F80</td>
<td>3</td>
<td>4522</td>
</tr>
<tr>
<td>BZ2_bzWrite</td>
<td>bzip2</td>
<td>405780</td>
<td>1260</td>
<td>74374</td>
</tr>
<tr>
<td>BZ2_bzWriteOpen</td>
<td>bzip2</td>
<td>405620</td>
<td>3</td>
<td>180</td>
</tr>
<tr>
<td>BZ2_bzWriteOpen</td>
<td>bzip2</td>
<td>405590</td>
<td>3</td>
<td>96</td>
</tr>
<tr>
<td>BZ2_bzDecompressEnd</td>
<td>bzip2</td>
<td>4047C0</td>
<td>1984</td>
<td>157496191</td>
</tr>
<tr>
<td>BZ2_bzDecompressInit</td>
<td>bzip2</td>
<td>404670</td>
<td>3</td>
<td>150</td>
</tr>
<tr>
<td>BZ2_bzCompressEnd</td>
<td>bzip2</td>
<td>4045F0</td>
<td>3</td>
<td>108</td>
</tr>
<tr>
<td>BZ2_bzCompress</td>
<td>bzip2</td>
<td>404470</td>
<td>1985</td>
<td>46149</td>
</tr>
<tr>
<td>BZ2_bzCompressInit</td>
<td>bzip2</td>
<td>404220</td>
<td>3</td>
<td>3375</td>
</tr>
<tr>
<td>handle_compress.isra.2</td>
<td>bzip2</td>
<td>403db0</td>
<td>1985</td>
<td>323506368</td>
</tr>
<tr>
<td>default_bzalloc</td>
<td>bzip2</td>
<td>403da0</td>
<td>18</td>
<td>94</td>
</tr>
<tr>
<td>default_bzfree</td>
<td>bzip2</td>
<td>403d80</td>
<td>18</td>
<td>72</td>
</tr>
<tr>
<td>add_pair_to_block</td>
<td>bzip2</td>
<td>403c40</td>
<td>33894</td>
<td>1506675</td>
</tr>
<tr>
<td>uncompressStream</td>
<td>bzip2</td>
<td>403920</td>
<td>3</td>
<td>25386</td>
</tr>
<tr>
<td>compressStream</td>
<td>bzip2</td>
<td>4035C0</td>
<td>3</td>
<td>24084</td>
</tr>
<tr>
<td>myfeof</td>
<td>bzip2</td>
<td>403430</td>
<td>1266</td>
<td>16434</td>
</tr>
<tr>
<td>BZ2_blockSort</td>
<td>bzip2</td>
<td>4031c0</td>
<td>11</td>
<td>29973150</td>
</tr>
<tr>
<td>mainSort</td>
<td>bzip2</td>
<td>403200</td>
<td>11</td>
<td>176057843</td>
</tr>
<tr>
<td>mainGUT</td>
<td>bzip2</td>
<td>402b0</td>
<td>312452</td>
<td>4120033586</td>
</tr>
<tr>
<td>fallbackSort</td>
<td>bzip2</td>
<td>4017b0</td>
<td>9</td>
<td>940379636</td>
</tr>
<tr>
<td>spec_FWwrite</td>
<td>bzip2</td>
<td>401600</td>
<td>1993</td>
<td>65769</td>
</tr>
<tr>
<td>spec_Reset</td>
<td>bzip2</td>
<td>401510</td>
<td>6</td>
<td>84</td>
</tr>
<tr>
<td>spec_ungetc</td>
<td>bzip2</td>
<td>401420</td>
<td>3751</td>
<td>108779</td>
</tr>
</tbody>
</table>

COSC 340: Software Engineering
Cache Simulation with Pin

• Multiple cache simulation tools ship with Pin
  – dcache: simulates a simple L1 data cache
  – allcache: simulates instruction, data caches and TLB's

• Pin cache tools are configurable
  – Tune total size, line size, associativity
  – Allows HW/SW design space exploration

• Can combine with other tools for advanced analysis
  – For example, combine with memory tracing, allocation site profiling to
    understand cache locality of particular application data
Tuning Cache Size with Pin

- Evaluate D1 cache miss rate with matrix multiply
  - Multiply two 800x800 matrices of doubles
  - Use Pin to vary cache size with 32KB, 64KB, and 128KB of D1 cache

<table>
<thead>
<tr>
<th>D1 cache size</th>
<th>mm-plain D1 miss %</th>
<th>mm-transpose D1 miss %</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 KB</td>
<td>47.55</td>
<td>9.9</td>
</tr>
<tr>
<td>64 KB</td>
<td>10.74</td>
<td>9.03</td>
</tr>
<tr>
<td>128 KB</td>
<td>7.51</td>
<td>8.62</td>
</tr>
</tbody>
</table>