An Empirical Study: mems as a Static Performance Metric

Liwei Zhang, Baoquan Cui, Xutong Ma, Jian Zhang

ISCAS Inria

QRS 2025, Hangzhou

mems=?

mems=memory access

The mems Metric

- *《The Art of Computer Programming》*

can be followed by 4397028651 before we get stuck again.

In Section 7.2.3, we'll study ways to estimate the behavior of such searches, without actually performing them. Such estimates tell us in this case that the Paige–Tompkins method essentially traverses an implicit search tree that contains about 2.5×10^{18} nodes. Most of those nodes belong to only a few levels of the tree; more than half of them deal with choices on the right half of the sixth row of M, after about 50 of the 90 blanks have been tentatively filled in. A typical node of the search tree probably requires about 75 mems (memory accesses) for processing, to check validity. Therefore the total running time on a modern computer would be roughly the time needed to perform 2×10^{20} mems.

Parker, on the other hand, went back to the method that Euler had originally used to search for orthogonal mates in 1779. First he found all of the so-called *transversals* of L, namely all ways to choose some of its elements so that there's exactly one element in each row, one in each column, and one of each value. For example, one transversal is 0859734216, in Euler's notation, meaning that we choose the 0 in column 0, the 8 in column 1, ..., the 6 in column 9. Each transversal that includes the k in L's leftmost column represents a legitimate way to place the ten k's into square M. The task of finding transversals is, in fact, rather easy, and the given matrix L turns out to have exactly 808 of them; there are respectively (79, 96, 76, 87, 70, 84, 83, 75, 95, 63) transversals for $k = (0, 1, \ldots, 9)$

Algorithmic Complexity Analysis: practically using Knuth's Ordin
Operations (oops) and Memory Operations (mems) method

Asked 11 years, 6 months ago Modified 3 years, 11 months ago Viewed 539 times

4

In implementing most algorithms (sort, search, graph traversal, etc.), there is frequently a trade-off that can be made in reducing memory accesses at the cost of additional ordinary operations.

Knuth has a useful method for comparing the complexity of various algorithm implementations by abstracting it from particular processors and only distinguishing between ordinary operations (oops) and memory operations (mems).

In compiled programs, one typically lets the compiler organise the low level operations, and hopes that the operating system will handle the question of whether data is held in cache memory (faster) or in virtual memory (slower). Furthermore, the exact number / cost of instructions is encapsulated by the compiler.

With Forth, there is no longer such encapsulation, and one is much closer to the machine, albeit perhaps to a stack machine running on top of a register processor.

Ignoring the effect of an operating system (so no memory stalls, etc.), and assuming for the moment a simple processor,

(1) Can anyone advise on how the ordinary *stack* operations in Forth (e.g. dup, rot, over, swap, etc.) compare with the cost of Forth's memory access fetch (@) or store (!) ?

(2) Is there a rule of thumb I can use to decide how many ordinary operations to trade-off against saving a memory access?

What I'm looking for is something like 'memory access costs as much as 50 ordinary ops, or 500 ordinary ops, or 5 ordinary ops' Ballpark is absolutely fine.

I'm trying to get a sense of the relative expense of fetch and store vs. rot, swap, dup, drop, over, correct to an order of magnitude.

performance algorithm theory computation-theory forth

Path sensitive mems measurement

- **Reg vs Cache vs RAM—— mems ≈ RAM access**
  - Array access ≈ RAM access => mems ≈ array access
- **Example**:
  - Counting memory read/write operations (e.g., array accesses)

o, arr[i] = i+2;

oo, arr[i+1] = arr[i];

- **Path-sensitive analysis**:

$$\text{Performance} = \frac{\sum_i (\delta_i * \text{pind}_i)}{\sum_i \delta_i}$$

  - Computing mems per execution path and aggregate via weighted averages

Background

- Most methods require actually running the program to measure time
- Traditional runtime profiling is platform-dependent and expensive.
- Theoretical complexity ≠ practical performance:
  - hidden constants and hardware variations.

Background

- Most methods require actually running the program to measure time
- Traditional runtime profiling is platform-dependent and expensive.
- Theoretical complexity ≠ practical performance:
  - hidden constants and hardware variations.

Motivating Example

- Same size(n), same path lengths, but different mems

```
void test(int n, int mode) {
    int arr[n], a = 0, b = 0;
    for(int i = 0; i < n; i++) {
        if(mode > 0) {
            arr[i] = i * 2 + arr[i];
            mode = mode - 1;
        } else {
            b = i * 3 + b;
            mode = mode - 1;
        }
    }
}
```

Motivating Example

- Same size(n), same path lengths, but different mems

```
void test(int n, int mode) {
    int arr[n], a = 0, b = 0;
    for(int i = 0; i < n; i++) {
        if(mode > 0) {
            arr[i] = i * 2 + arr[i];
            mode = mode - 1;
        } else {
            b = i * 3 + b;
            mode = mode - 1;
        }
    }
}
```

True Branch: mems+=2

False Branch: mems+=0

Motivating Example

- Same size, same path lengths, but different mems

```
void test(int n, int mode) {
    int arr[n], a = 0, b = 0;
    for(int i = 0; i < n; i++) {
        if(mode > 0) {
            arr[i] = i * 2 + arr[i];
            mode = mode - 1;
        } else {
            b = i * 3 + b;
            mode = mode - 1;
        }
    }
}
```

TABLE I
REFINED RESULTS: EXECUTION AND VALGRIND TIME ON UNINSTRUMENTED PROGRAMS

p_1	p_2	file	len	mems	t_0(ms)	vg_0(ms)
100	0	path_6	200	0	0.068	8.488
100	50	path_7	200	100	0.061	8.189
100	100	path_8	200	200	0.052	7.762
500	0	path_9	1,000	0	0.241	12.818
500	250	path_10	1,000	500	0.178	12.289
500	500	path_11	1,000	1,000	0.187	12.399
10,000	0	path_30	20,000	0	0.079	5.729
10,000	2,500	path_31	20,000	2,500	0.092	5.555
10,000	5,000	path_32	20,000	5,000	0.089	5.838
10,000	7,500	path_33	20,000	7,500	0.116	5.718
10,000	10,000	path_34	20,000	10,000	0.128	5.538
50,000	0	path_35	100,000	0	0.362	8.015
50,000	12,500	path_36	100,000	12,500	0.418	9.257
50,000	25,000	path_37	100,000	25,000	0.546	8.718
50,000	37,500	path_38	100,000	37,500	0.630	9.018
50,000	50,000	path_39	100,000	50,000	0.639	8.824
100,000	0	path_40	200,000	0	0.720	9.824
100,000	25,000	path_41	200,000	25,000	0.857	10.585
100,000	50,000	path_42	200,000	50,000	1.051	11.500
100,000	75,000	path_43	200,000	75,000	1.201	12.427
100,000	100,000	path_44	200,000	100,000	1.361	14.158
200,000	0	path_45	400,000	0	1.319	15.259
200,000	50,000	path_46	400,000	50,000	1.662	16.742
200,000	100,000	path_47	400,000	100,000	2.077	18.213
200,000	150,000	path_48	400,000	150,000	2.467	19.587
200,000	200,000	path_49	400,000	200,000	2.544	21.368

Figure 1. Graphical representation of memory access frequency and execution time correlation.

Motivating Example

- Same size, same path lengths, but different mems

```
void test(int n, int mode) {
    int arr[n], a = 0, b = 0;
    for(int i = 0; i < n; i++) {
        if(mode > 0) {
            arr[i] = i * 2 + arr[i];
            mode = mode - 1;
        } else {
            b = i * 3 + b;
            mode = mode - 1;
        }
    }
}
```

same path lengths
large mems value
longer time

TABLE I
REFINED RESULTS: EXECUTION AND VALGRIND TIME ON UNINSTRUMENTED PROGRAMS

p_1	p_2	file	len	mems	t_0(ms)	vg_0(ms)
100	0	path_6	200	0	0.068	8.488
100	50	path_7	200	100	0.061	8.189
100	100	path_8	200	200	0.052	7.762
500	0	path_9	1,000	0	0.241	12.818
500	250	path_10	1,000	500	0.178	12.289
500	500	path_11	1,000	1,000	0.187	12.399
10,000	0	path_30	20,000	0	0.079	5.729
10,000	2,500	path_31	20,000	2,500	0.092	5.555
10,000	5,000	path_32	20,000	5,000	0.089	5.838
10,000	7,500	path_33	20,000	7,500	0.116	5.718
10,000	10,000	path_34	20,000	10,000	0.128	5.538
50,000	0	path_35	100,000	0	0.362	8.015
50,000	12,500	path_36	100,000	12,500	0.418	9.257
50,000	25,000	path_37	100,000	25,000	0.546	8.718
50,000	37,500	path_38	100,000	37,500	0.630	9.018
50,000	50,000	path_39	100,000	50,000	0.639	8.824
100,000	0	path_40	200,000	0	0.720	9.824
100,000	25,000	path_41	200,000	25,000	0.857	10.585
100,000	50,000	path_42	200,000	50,000	1.051	11.500
100,000	75,000	path_43	200,000	75,000	1.201	12.427
100,000	100,000	path_44	200,000	100,000	1.361	14.158
200,000	0	path_45	400,000	0	1.319	15.259
200,000	50,000	path_46	400,000	50,000	1.662	16.742
200,000	100,000	path_47	400,000	100,000	2.077	18.213
200,000	150,000	path_48	400,000	150,000	2.467	19.587
200,000	200,000	path_49	400,000	200,000	2.544	21.368

Figure 1. Graphical representation of memory access frequency and execution time correlation.

Experiments

- **RQ1**:
  - For different paths in the same program, does mems always correlate with execution time?
- **RQ2**:
  - Across different programs and different paths, does mems still correlate with execution time?

Approach

- AST Instrumentation:
  - counting mems
  - printing paths and execution time

red: original program
black:inserted code

Listing 1. Simplified instrumented version with original code highlighted

```
1  LARGE_INTEGER freq, start, end;
2  QueryPerformanceFrequency(&freq);
3  QueryPerformanceCounter(&start);
4  int a = 0, b = 0, i;
5  for (i = 0; i < n; i++) {
6     path_len = path_len + 1;
7     if (mode > 0) {
8        path_len = path_len + 1;
9        arr[i] = i * 2 + arr[i];
10       mems = mems + 2;
11       mode = mode - 1;
12    } else {
13       path_len = path_len + 1;
14       b = i * 3 + b;
15       mode = mode - 1;
16    }
17 }
18 printf("Total path length: %d\n", path_len);
19 printf("Total memory accesses: %d\n", mems);
20 QueryPerformanceCounter(&end);
21 double time_taken = (double)(end.QuadPart -
       start.QuadPart) / freq.QuadPart;
22 printf("Execution time: %f\n", time_taken);
```

Results

- **RQ1**:
  - In the same program, does mems correlate with execution time?

TABLE II
CORRELATION COEFFICIENTS BETWEEN MEMORY USAGE AND EXECUTION TIME

Program	Correlation Coefficient	Interpretation
Array	0.99980	Very strong
Bubble	0.99980	Very strong
Insertsort	0.99996	Very strong
Sieve	0.99986	Very strong
Topo	0.99900	Very strong

Results

- **RQ2**:
  - Across different programs, does mems correlate with execution time ?

large mems value
longer time

TABLE III
SELECTED EXPERIMENTAL RESULTS ON SERVER

Program	n	Path Len.	mems	Time (ms)
bubble	100	9,999	19,800	1.857
change	100	10,106	14	1.473
shell	1,000	12,715	18,800	2.743
sieve	5,000	13,175	13,089	1.267
array	5,000	15,002	20,000	1.479
FFT	2,048	18,397	118,592	7.134

Results

- **RQ2**:
  - Across different programs, does mems correlate with execution time ?

large mems value
longer time

TABLE III
SELECTED EXPERIMENTAL RESULTS ON SERVER

Program	n	Path Len.	mems	Time (ms)
bubble	100	9,999	19,800	1.857
change	100	10,106	14	1.473
shell	1,000	12,715	18,800	2.743
sieve	5,000	13,175	13,089	1.267
array	5,000	15,002	20,000	1.479
FFT	2,048	18,397	118,592	7.134

large mems value
shorter time

Results

- **RQ2**:
  - Across different programs, does mems correlate with execution time ?
    - ——> Different programs, different structures (n, path lengths, number of conditional branches...)
    - ——> n also correlate with execution time

Enhanced Validation

- Regression Models:
  - Log-log regression shows mems explains 41% of execution time variance globally.
  - Intra-program regression coefficients: β ≈ 1.0 (e.g., bubble, insertsort)

TABLE VII
PER-PROGRAM LOG-LOG REGRESSION OF MEMS VS. EXECUTION TIME.

Program	Coef	95% CI Low	95% CI High	R^2	N
bubble	0.98	0.96	1.00	0.9996	8
insertsort	0.96	0.93	0.99	0.9988	9
selectsort	1.87	1.77	1.97	0.9970	8
shellsort	0.94	0.87	1.00	0.9949	8
array	0.95	0.91	0.99	0.9946	18

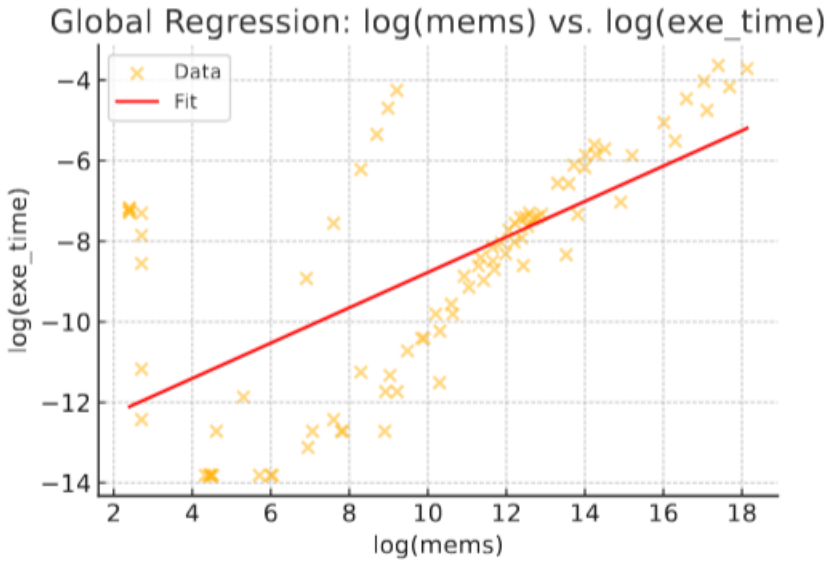

Figure 7. Global log-log regression between mems and execution time. The shaded region represents the 95% confidence interval for the fitted model.

Conclusion & Future Work

- **Conclusion**:
  - mems is reliable for intra-program path comparison but insufficient for cross-program prediction.
  - Comparing different programs needs more issues (or in some special cases)
- **Future Directions**:
  - Combining mems with path length, arithmetic intensity, or cache models.
  - Extending to larger codebases and real-world applications.
  - Our tool: Eppather
    - A static testcase generation tool about mems

Background

- mems: memory access metric (Knuth)
- Existing metrics rely on runtime/hardware
- Can mems predict computational cost stically?

Method

- Count mems statically in code
- Compare with execution time
- Analyze correlation with examples & regression

Experiment

- Single program: mems ↑ → time ↑ (linear)
- Across programs: correlation strong, some exceptions

Conclusion

- mems is effective for path-level analysis
- Cross-program use needs more research

Q&A

Thanks for listening

zhanglw@ios.ac.cn