OPTION PROFILING and OPTION TRACECACHE / OPTION CACHE

PicoMite MMBasic — Performance Optimisation Guide

Overview

MMBasic provides two complementary tools for understanding and improving program performance:

Tool	Purpose
`OPTION PROFILING`	Measures *where* time is spent — identifies slow SUBs, busy variables, and hot code paths
`OPTION TRACECACHE`	Speeds up *simple assignment statements* and `IF` conditions by compiling them once and replaying without re-parsing
`OPTION CACHE SUB`	Restricts the trace cache to specific SUBs/FUNCTIONs
`OPTION CACHE DEBUG`	Reveals why particular lines are not being cached

The two main tools work best together: use profiling first to identify *what* to optimise, then enable the trace cache to accelerate the hottest code, and profile again to confirm the improvement.

OPTION PROFILING

What It Does

When profiling is enabled, MMBasic counts and times every significant operation during program execution. When the program ends (via `END`, not `CTRL-C`) a performance report is printed to the console.

Syntax

```
OPTION PROFILING ON    ' enable before running the program
' ... run program ...
END                    ' report printed here
OPTION PROFILING OFF   ' release memory
```

Profiling can also be turned on at the BASIC prompt before typing `RUN`.

Memory Cost

Enabling profiling allocates approximately **9–14 KB** from the heap:

Array	Size
Per-token command counts	4 KB

Per-SUB call counts	~1–2 KB
Per-SUB inclusive time	~2–4 KB
Per-SUB exclusive time	~2–4 KB

This memory is held for the life of the profiling session and released by `OPTION PROFILING OFF`. On a Pico with limited heap it may affect programs that use large arrays or string buffers — disable profiling for final production use.

Runtime Overhead

Each executed statement incurs one extra integer increment. Each SUB or FUNCTION call captures a timestamp on entry and exit. Each variable lookup (`findvar`) increments two counters. In practice this adds roughly **5–10 %** to execution time — enough to skew absolute timings slightly, but the *relative* rankings of hot SUBs remain reliable.

Reading the Profiling Report

When `END` is reached the interpreter prints a block like this:

```
[PERF] elapsed=1482315 us  statements=924160  findvar=382000 (locals=301200 [78%]
globals=80800 [21%])  user_subs=184000
[PERF] tracecache: enabled=1 size=64 replays=301120 compiles_ok=47 compiles_bad=3
[PERF] tracecache: lookup_null=0 alloc_fail=0 optin_skip=0
[PERF] tracecache hits by SUB (top 20):
        let_hits   if_hits     total  name
          180000     30000    210000  Integrate
           90000     20000    110000  UpdatePos

[PERF] top commands by dispatch count:
          924160  LET
          184000  NEXT
           92000  FOR
           46000  IF

[PERF] top SUBs by exclusive (self) time:
       self_us    incl_us     calls   self_us/call  name
        820000    1200000     46000          17  Integrate
        310000     310000     92000           3  UpdatePos
         90000      90000      4000          22  DrawScreen

[PERF] top SUBs by call count:
          92000  UpdatePos
          46000  Integrate
```

Elapsed Time and Statement Count

`elapsed` is the wall-clock time in microseconds from program start to `END`. `statements` is the raw count of statement dispatches. Dividing gives average time per statement.

findvar Breakdown

Every variable access goes through `findvar()`. The split between locals and globals tells you about variable lookup cost:

- **High local %** is good — local variable lookups are faster than globals.
- **Low local % in a hot SUB** suggests you could declare more variables as `LOCAL` inside that SUB.

Top Commands by Dispatch Count

This shows which BASIC keywords are executed most. `LET` (assignment) usually dominates in computation loops; `FOR` / `NEXT` indicate loop overhead. Use this list to confirm that the trace cache (see below) is targeting the right statements.

Top SUBs by Exclusive (Self) Time

This is the most important section for optimisation:

- **`self_us`** — time spent *inside* this SUB, not counting time in SUBs it called. This is your primary ranking metric.
- **`incl_us`** — total time including all callees. If `incl_us` >> `self_us`, the SUB itself is fast but calls something slow.
- **`calls`** — how many times the SUB was invoked.
- **`self_us/call`** — average cost per call. A high value means each call is expensive; a low value with high `self_us` means the SUB is called very frequently.

Identifying "Hot" Subroutines

A subroutine is a candidate for optimisation if:

1. **High `self_us`** — it is at the top of the exclusive-time list.
2. **High call count** — it is called thousands of times per second, making even tiny per-call savings significant.
3. **High `self_us/call`** — each invocation is expensive, suggesting either complex arithmetic, many variable lookups, or nested loops inside it.

The top 1–3 entries by `self_us` typically account for 60–80 % of total runtime. Optimising them gives the greatest return.

OPTION TRACECACHE / OPTION CACHE

What the Trace Cache Does

The trace cache *pre-compiles* simple assignment (`LET`) statements and `IF` conditions into a compact internal bytecode the first time they are seen, then *replays* that bytecode on every subsequent execution without re-parsing the BASIC source. For tight computation loops this eliminates the interpreter's tokeniser, operator-table lookups, and recursive-descent expression evaluator for those lines.

Syntax

```
OPTION TRACECACHE ON [size]   ' enable, optionally set cache size
OPTION TRACECACHE OFF         ' disable and free memory immediately

OPTION CACHE DEBUG ON         ' print lines that fail to compile to cache
OPTION CACHE DEBUG OFF

OPTION CACHE SUB name [, name ...]  ' restrict cache to listed SUBs only
OPTION CACHE SUB OFF                ' cache everywhere (default)
```

`size` is the number of cache slots (16 to 1024, always rounded to the nearest power of 2). The default is 64.

Memory Cost

The cache uses a two-region design:

Region	Formula	Default (64 slots)
Slot headers	slots × ~24 bytes	~1.5 KB
Bytecode arena	headers × 8	~12 KB
Total		**~13.5 KB**

Increasing the slot count grows both regions proportionally:

Slots	Approximate total
16	~3.4 KB
64	~13.5 KB
256	~54 KB
1024	~216 KB

On the Pico's constrained heap, 64 slots (the default) is a good starting point. Increase it only if profiling shows `lookup_null` or `alloc_fail` is non-zero.

What Statements Are Accelerated

The trace cache only handles specific forms. During the first encounter of a `LET` or `IF`, the cache *attempts* to compile it. If compilation succeeds the statement runs from the cache on every future execution of that line. If it fails, the line continues to run through the normal interpreter (with no performance penalty beyond the one-time compilation attempt).

LET (Assignment) — Eligible Forms

Variables must be numeric scalars (`FLOAT` or `INTEGER`) — global or local. Strings and pointer dereferences are not supported. Local variables are cached with per-frame re-resolution (see below).

Form	Example	Cached?
`var = constant`	`x = 3.14`	Yes
`var = otherVar`	`y = x`	Yes
`var = a OP b`	`t = a + b`	Yes
Array element	`a(i) = x + y`	Yes (1D/2D)
Local variable (numeric scalar)	`LOCAL x : x = a + b`	Yes
String	`a$ = b$`	**No**
Expression chain	`x = a + b * c - d`	Depends on depth

Local variable note: Local variable references are compiled using the variable's name rather than its slot index. On the first replay inside a new call frame the cache re-resolves each local name to its current slot; subsequent replays within the same frame execute without lookup. This means `LOCAL` variables in tight loops are fully accelerated. The limit is 8 distinct local variable names per cached statement; a `LET` that references more than 8 different locals is marked uncacheable.

Supported arithmetic operators for `FLOAT` : `+ - * / ^`
Supported operators for `INTEGER` : `+ - * \ MOD`
Supported comparisons: `= <> < > <= >=`
Supported intrinsics: `SIN COS TAN ASIN ACOS ATAN SQR ABS INT()`
Supported logical/bitwise (INTEGER only): `AND OR XOR` `<< >>` `NOT`

IF Conditions — Eligible Forms

The condition expression of every `IF` statement is compiled and cached as a separate `ENTRY_KIND_IF` entry using the same operator set. This applies to both single-line and multiline `IF` forms:

- `IF condition THEN assignment` — condition cached; the assignment is also cached independently as a `ENTRY_KIND_LET` entry.
- `IF condition THEN` / body / `ENDIF` — condition cached; any qualifying `LET` statements in the body are each cached independently as they execute.

Only the condition expression is compiled into the cache — the branch decision and all control flow remain with the interpreter.

Ineligible Statements

Every other BASIC statement runs through the normal interpreter unchanged. The trace cache does not accelerate `FOR` , `NEXT` , `WHILE` , `DO` , `PRINT` , `INPUT` , `SUB` calls, or any statement that modifies program structure.

How to Use the Trace Cache

Step 1 — Profile Without the Cache

```
OPTION PROFILING ON
RUN "myprogram.bas"
```

Look at the `[PERF] top commands by dispatch count` section. If `LET` appears at or near the top the trace cache is likely to help. Note the hot SUBs from the exclusive-time table.

Step 2 — Enable the Cache and Re-run

```
OPTION PROFILING ON
OPTION TRACECACHE ON
RUN "myprogram.bas"
```

Compare `elapsed` times. In the new report, look at the `tracecache` section:

```
[PERF] tracecache: enabled=1 size=64 replays=301120 compiles_ok=47 compiles_bad=3
```

- **`replays`** — how many statement executions were served from the cache. Higher is better; this is the work the interpreter did not have to repeat.

- **`compiles_ok`** — how many distinct lines were successfully compiled.
- **`compiles_bad`** — how many lines were attempted but could not be compiled (e.g., used strings or locals).
- **`lookup_null`** / **`alloc_fail`** — non-zero values mean the cache is full; increase `size`.

The `tracecache hits by SUB` table shows *which* SUBs benefited most:

```
       let_hits   if_hits     total  name
         180000     30000    210000  Integrate
```

Cross-reference this with the profiling exclusive-time table. The best outcome is high `let_hits` in your hottest SUB combined with a meaningful reduction in `self_us`.

Step 3 — Investigate Cache Misses

If `compiles_bad` is high or a hot SUB shows zero hits, turn on cache debug:

```
OPTION CACHE DEBUG ON
OPTION TRACECACHE ON
RUN "myprogram.bas"
```

For every line that fails to compile, the interpreter prints the offending text and the enclosing SUB name. Common causes:

- The variable is a local (`LOCAL x`) — move it to module level if possible, or accept the miss.
- The expression uses a string — strings are not supported.
- The expression uses a user-defined function on the right-hand side — function calls are not cached.
- The right-hand side exceeds 64 intermediate operations — simplify the expression.

Step 4 — Focus the Cache on Hot SUBs

If only a few SUBs matter and the cache size is limited, restrict caching to them:

```
OPTION CACHE SUB Integrate, UpdatePos
OPTION TRACECACHE ON
```

This wastes no cache slots on one-shot initialisation code and ensures the hot SUBs compete only with each other.

Worked Optimisation Example

Suppose profiling reveals:

```
[PERF] top commands by dispatch count:
          900000  LET
           90000  NEXT

[PERF] top SUBs by exclusive (self) time:
       self_us    incl_us     calls   self_us/call
        820000    820000      90000           9  Integrate
```

`Integrate` is called 90 000 times and its body is dominated by `LET` statements. Each call takes ~9 µs.

After `OPTION TRACECACHE ON`:

```
[PERF] tracecache: replays=720000 compiles_ok=8 compiles_bad=0

[PERF] top SUBs by exclusive (self) time:
       self_us    incl_us     calls   self_us/call
        490000    490000      90000           5  Integrate
```

`replays=720000` means 8 cache entries were each hit 90 000 times. `Integrate` dropped from 9 µs/call to 5 µs/call — a 44 % speedup for that SUB and roughly 22 % off total elapsed time.

Limitations

OPTION PROFILING

- The report is only printed when the program ends via `END`. A `CTRL-C` interrupt discards it.
- Profiling overhead (5–10 %) slightly inflates absolute timings. Use `self_us/call` rankings, not raw microsecond values, for decisions.
- The SUB timing uses inclusive/exclusive accounting based on timestamp differences. Very short SUBs (under ~5 µs per call) have timing noise comparable to the call overhead itself.
- Only the top 20 entries are shown in each table.
- Top-level code (outside any SUB) is tracked internally but labelled as index `MAXSUBFUN` in arrays; it may not appear labelled in the report.

OPTION TRACECACHE

- Only `LET` and `IF` conditions are eligible — no other statement types.
- **Local variables are cached**, but require re-resolution on each new call frame. When a SUB is entered the cache detects the frame-generation change and re-resolves local variable slots before replaying. This is fast but adds a small one-time cost per call — negligible unless the SUB is called billions of times with almost no body. Up to 8 distinct local variable names per cached `LET` are supported; lines with more than 8 distinct locals fall back to the interpreter.
- **Strings are never cached.**
- The cache uses the *address* of the source token in program memory as its key. If the program is edited (`NEW`, `EDIT`, `DIM` changes, `OPTION BASE` or `OPTION EXPLICIT` changes), all cache entries are invalidated and must be re-compiled on the next run. This invalidation is automatic and safe.
- Each cache slot holds up to 64 intermediate operations. A very long expression may exceed this and fall back to the interpreter.
- The hash table uses open addressing with a probe limit of 8. With a small cache and many distinct cacheable lines, collisions can prevent some lines from ever being cached (`lookup_null` will be non-zero). Increase `size` or use `OPTION CACHE SUB` to focus the cache.
- The arena (bytecode storage) is fixed at allocation time. If it fills up, no new lines can be compiled (`alloc_fail` becomes non-zero). Increasing `size` increases arena capacity proportionally.
- The trace cache does not overlap with the profiling overhead — running both simultaneously is safe but adds ~13.5 KB + ~9–14 KB to heap usage.
- The cache cannot be resized while enabled. Issue `OPTION TRACECACHE OFF` first, then `OPTION TRACECACHE ON newsize`.

Quick Reference

```
' -- Profile only --
OPTION PROFILING ON
```

```
RUN "prog.bas"           ' prints report on END

' -- Cache only --
OPTION TRACECACHE ON     ' 64 slots, ~13.5 KB
RUN "prog.bas"

' -- Both together (recommended workflow) --
OPTION PROFILING ON
OPTION TRACECACHE ON
RUN "prog.bas"

' -- Debug cache misses --
OPTION CACHE DEBUG ON
OPTION TRACECACHE ON
RUN "prog.bas"

' -- Restrict cache to hot SUBs --
OPTION CACHE SUB Integrate, UpdatePos
OPTION TRACECACHE ON
RUN "prog.bas"

' -- Cleanup --
OPTION PROFILING OFF
OPTION TRACECACHE OFF
```

Memory Budget Summary

Option	Heap Used
`OPTION PROFILING ON`	~9–14 KB
`OPTION TRACECACHE ON` (64 slots)	~13.5 KB
`OPTION TRACECACHE ON 256`	~54 KB
Both enabled (default sizes)	~23–28 KB

Interpreting Results: Decision Flowchart

```
Run with OPTION PROFILING ON
         |
         v
Is LET top command?  --No--> Cache unlikely to help. Look at
         |                   SUB call overhead or algorithm.
        Yes
         |
         v
Enable OPTION TRACECACHE ON, re-run
         |
         v
Check replays / (replays + statements): is hit rate > 50%?
```

```
         |                            |
        Yes                          No
         |                            |
    Good. Check elapsed           Check CACHE DEBUG ON
    time improvement.             for miss reasons.
         |                            |
    Improvement < expected?      >8 locals in one LET? -> Split expression
         |                       Strings? -> Accept
        Yes                      Cache full? -> Increase size
         |
         v
Use OPTION CACHE SUB to focus
cache on hottest 2-3 SUBs.
Re-run and compare.
```