

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 <i>where</i> time is spent — identifies slow SUBs, busy variables, and hot code paths
OPTION TRACECACHE	Speeds up <i>assignment statements</i> , IF conditions, DO / LOOP conditional clauses, and GOTO / GOSUB / RESTORE jumps 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 ' must be an early statement inside the program
' ... program body ...
```

```
END          ' report printed here
OPTION PROFILING OFF  ' release memory (optional – freed automatically on next RUN)
```

Important: `OPTION PROFILING ON` must be placed as an early statement **inside** the program itself, not typed at the BASIC command prompt before `RUN`. When `RUN` is executed it calls `ClearRuntime`, which resets profiling state — any `OPTION PROFILING ON` entered at the prompt is therefore lost before the program starts. Place `OPTION PROFILING ON` as the first executable line of the program being profiled.

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%] g:
[PERF] tracecache: flags=0x3f size=64 replays=301120 compiles_ok=47 compiles_bad=3
[PERF] tracecache: lookup_null=0 alloc_fail=0 optin_skip=0 jump_hits=18400
[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* assignment (`LET`) statements, `IF` conditions, `DO / LOOP` conditional clauses, and `GOTO / GOSUB / RESTORE` jump targets 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 [, flags]] ' enable; optionally set cache size and featu
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 4096, always rounded to the nearest power of 2). The default is 64.

`flags` is a bitmask that enables individual optimisation categories independently. The default (when omitted) is `63` (all features enabled). To enable only specific categories, OR together the values from this table:

Bit	Value	Category
0	1	Numeric scalar and array <code>LET</code>
1	2	String scalar <code>LET</code>

Bit	Value	Category
2	4	IF condition expressions
3	8	DO / LOOP WHILE and UNTIL condition expressions
4	16	GOTO and GOSUB jump targets
5	32	RESTORE jump targets
—	63	All features (default)

Examples:

```
OPTION TRACECACHE ON           ' all features, default size (63 = 0x3F)
OPTION TRACECACHE ON 128      ' all features, 128 slots
OPTION TRACECACHE ON 64, 5    ' size=64; only numeric LET (1) + IF (4) enabled
OPTION TRACECACHE ON 64, 19   ' size=64; numeric LET (1) + IF (4) + GOTO/GOSUB (16)
OPTION TRACECACHE OFF         ' disable all, free memory
```

The `flags` value is reported as `flags=0x...` in the `[PERF]` trace-cache line, so you can confirm exactly which features were active during a profiling run.

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

Numeric LET (feature bit 0, value 1): Variables must be numeric scalars (`FLOAT` or `INTEGER`) — global or local. Pointer dereferences are not supported. Local variables are cached with per-frame re-resolution (see below).

Form	Example	Cached?
<code>var = constant</code>	<code>x = 3.14</code>	Yes
<code>var = otherVar</code>	<code>y = x</code>	Yes
<code>var = a OP b</code>	<code>t = a + b</code>	Yes
Array element	<code>a(i) = x + y</code>	Yes (1D/2D)
Local variable (numeric scalar)	<code>LOCAL x : x = a + b</code>	Yes

String LET (feature bit 1, value 2): String scalar variables — with the `$` suffix or declared `AS STRING` — support the following assignment forms. Arrays, string functions, and complex concatenation are not cached.

Form	Example	Cached?
<code>s\$ = "literal"</code>	<code>s\$ = "hello"</code>	Yes
<code>s\$ = t\$</code>	<code>name\$ = first\$</code>	Yes
<code>s\$ = s\$ + "literal"</code>	<code>s\$ = s\$ + "px"</code>	Yes (in-place append)
<code>s\$ = s\$ + t\$</code>	<code>buf\$ = buf\$ + sep\$</code>	Yes (in-place append)
<code>s\$ = a\$ + b\$</code>	<code>s\$ = x\$ + y\$</code>	No — left side must equal first operand
String function	<code>s\$ = LEFT\$(t\$, 3)</code>	No
String array	<code>a\$(i) = "x"</code>	No

Note on OPTION ESCAPE : String literal caching is disabled when `OPTION ESCAPE` is set, because escape sequences in string literals would require re-parsing.

Local variable note (numeric): 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. The limit is 8 distinct local variable names per cached statement.

Supported arithmetic operators for `FLOAT`: `+` `-` `*` `/` `^`

Supported operators for `INTEGER`: `+` `-` `*` `\` `MOD`

Supported comparisons: `=` `<>` `<` `>` `<=` `>=`

Supported logical/bitwise (`INTEGER` only): `AND` `OR` `XOR` `<<` `>>` `NOT`

Supported single-argument intrinsics (all take a numeric argument):

Function	Returns	Notes
<code>SIN</code> <code>COS</code> <code>TAN</code> <code>ASIN</code> <code>ACOS</code> <code>ATN</code> <code>SQR</code> <code>ABS</code>	<code>FLOAT</code>	Trig bails in degree mode
<code>INT()</code>	<code>INTEGER</code>	Floor toward $-\infty$
<code>FIX()</code>	<code>INTEGER</code>	Truncate toward zero
<code>CINT()</code>	<code>INTEGER</code>	Round to nearest
<code>SGN()</code>	<code>INTEGER</code>	-1, 0, or +1
<code>EXP()</code>	<code>FLOAT</code>	
<code>LOG()</code>	<code>FLOAT</code>	Bails if $\text{arg} \leq 0$
<code>DEG()</code>	<code>FLOAT</code>	Radians \rightarrow degrees
<code>RAD()</code>	<code>FLOAT</code>	Degrees \rightarrow radians

Supported two-argument intrinsics (both arguments coerced to `FLOAT`):

Function	Returns	Notes
<code>ATAN2(y, x)</code>	<code>FLOAT</code>	Bails in degree mode
<code>MAX(a, b)</code>	<code>FLOAT</code>	Only 2-arg form cached; 3+ args fall back to interpreter
<code>MIN(a, b)</code>	<code>FLOAT</code>	Only 2-arg form cached; 3+ args fall back to interpreter

IF Conditions and DO/LOOP Conditional Clauses — Eligible Forms

The condition expression of every `IF` statement (feature bit 2, value 4) and every `DO / LOOP` `WHILE` or `UNTIL` clause (feature bit 3, value 8) is compiled and cached using the same numeric-expression operator set. String conditions are never cached.

IF statement — both single-line and multiline 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.

DO/LOOP conditional clauses — all four pre-test and post-test forms:

Form	Clause cached
DO WHILE condition ... LOOP	WHILE clause at the DO (checked on every iteration)
DO UNTIL condition ... LOOP	UNTIL clause at the DO (checked on every iteration)
DO ... LOOP WHILE condition	WHILE clause at the LOOP (post-test)
DO ... LOOP UNTIL condition	UNTIL clause at the LOOP (post-test)

The stable tokenised-source address of the condition expression is used as the cache key, so each distinct loop's condition occupies one cache slot and is compiled at most once per run.

Only the condition expression is compiled into the cache — the branch decision and all loop control flow remain with the interpreter. Cache hits for DO/LOOP conditions are counted in the `if_hits` column of the per-SUB breakdown (see the profiling report section above).

Behaviour when a condition cannot be compiled (e.g., `DO UNTIL s$="end"` — a string comparison): the cache attempts compilation on each of the first four executions, then permanently marks the entry as `ST_BAD`. From the fifth execution onwards the cache lookup costs a few integer operations (a hash and an array probe) before returning immediately, adding negligible overhead compared to the normal `getnumber()` call that follows.

GOTO, GOSUB, RESTORE — Jump Target Cache

The destination address of `GOTO` and `GOSUB` statements (feature bit 4, value 16) and `RESTORE` statements (feature bit 5, value 32) with a **literal** line number or **literal** label is resolved once (by scanning program memory) and then cached. Every subsequent execution of the same statement returns the cached pointer directly, skipping `findlabel` / `findline` entirely.

Form	Example	Cached?
GOTO line number	GOTO 100	Yes (bit 4)
GOTO label	GOTO myloop	Yes (bit 4)
GOSUB line number	GOSUB 500	Yes (bit 4)
GOSUB label	GOSUB CalcStep	Yes (bit 4)
RESTORE line number	RESTORE 200	Yes (bit 5)
RESTORE quoted label	RESTORE "DataSection"	Yes (bit 5)
RESTORE variable	RESTORE x	No — target depends on run-time value

Jump hits are reported as `jump_hits` in the `[PERF]` trace-cache line (see below).

Ineligible Statements

Every other BASIC statement runs through the normal interpreter unchanged. The trace cache does not accelerate `FOR / NEXT` stepping, `PRINT`, `INPUT`, `SUB` calls, or any statement that modifies program structure. The *condition expressions* in `DO WHILE`, `DO UNTIL`, `LOOP WHILE`, and `LOOP UNTIL` clauses are compiled and cached (as described above), but the loop control flow itself is not.

How to Use the Trace Cache

Step 1 — Profile Without the Cache

Add `OPTION PROFILING ON` as the first executable line of your program, then run it normally:

```
' First line of myprogram.bas:
OPTION PROFILING ON
' ... rest of program ...
END
```

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

Add both options near the top of your program:

```
' First lines of myprogram.bas:
OPTION PROFILING ON
OPTION TRACECACHE ON
' ... rest of program ...
END
```

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

```
[PERF] tracecache: flags=0x3f size=64 replays=301120 compiles_ok=47 compiles_bad=3
[PERF] tracecache: lookup_null=0 alloc_fail=0 optin_skip=0 jump_hits=18400
```

- `flags` — bitmask of active feature categories (see table above). `0x3f` means all six categories are enabled.
- `replays` — how many `LET`, `IF`, and `DO / LOOP` condition evaluations were served from the expression cache. Higher is better.
- `jump_hits` — how many `GOTO / GOSUB / RESTORE` executions were served from the jump-target cache, skipping `findlabel / findline`.
- `compiles_ok` — how many distinct lines were successfully compiled (expression cache only).

- `compiles_bad` — how many lines were attempted but could not be compiled (unsupported expression or cache full). Use `OPTION CACHE DEBUG ON` to see which, tagged `[TC-BAD]` or `[TC-FULL]`.
- `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

```

The `if_hits` column counts replays of both `IF` conditions and `DO / LOOP` conditional clauses — any expression compiled via the IF-kind cache path. Cross-reference this with the profiling exclusive-time table. The best outcome is high `let_hits` and/or `if_hits` in your hottest SUB combined with a meaningful reduction in `self_us`.

Step 2a — Isolate the Benefit of Individual Categories

If adding the cache made the program *slower*, use the `flags` parameter to test each category independently and identify which is causing overhead:

```

OPTION TRACECACHE ON 64, 1  ' numeric LET only
OPTION TRACECACHE ON 64, 4  ' IF conditions only
OPTION TRACECACHE ON 64, 8  ' DO/LOOP conditions only
OPTION TRACECACHE ON 64, 48 ' GOTO/GOSUB/RESTORE only (16 + 32)
OPTION TRACECACHE ON 64, 3  ' LET only (numeric + string = 1 + 2)

```

Comparing `elapsed` across these runs pinpoints whether the overhead comes from a specific category.

Step 3 — Investigate Cache Misses

If `compiles_bad` is high or a hot SUB shows zero hits, turn on cache debug by adding these lines near the top of your program:

```

OPTION CACHE DEBUG ON
OPTION TRACECACHE ON
' ... rest of program ...

```

For every line that cannot be cached, the interpreter prints a tagged message showing the enclosing SUB name and the statement text. There are two distinct tags:

`[TC-BAD]` — the expression uses features the cache does not support. Increasing the cache size will not help. Common causes: - The expression calls a user-defined function — user function calls are not cached. - More than 8 distinct local variable names appear in a

single `LET` . - The right-hand side exceeds 64 intermediate operations — simplify the expression. - A string condition in `IF` or `DO` — only numeric conditions are compiled.

`[TC-FULL]` — the cache ran out of space for this line. The fix is to increase the cache size with `OPTION TRACECACHE ON <larger>`, or focus it on the hot SUBs with `OPTION CACHE SUB` . Two root causes produce this tag: - *Hash table full* — too many distinct cacheable lines competing for slots; the probe limit was reached (`lookup_null` counter). Only the first occurrence per run is printed; check `lookup_null` in the `[PERF]` report for the full count. - *Arena full* — the bytecode storage area is exhausted (`alloc_fail` counter). Printed once per affected line.

Step 4 — Focus the Cache on Hot SUBs

If only a few SUBs matter and the cache size is limited, restrict caching to them by adding these lines near the top of your program:

```
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
      900000 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: flags=0x3f 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 μ s/call to 5 μ s/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 μ s 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

Important: Like `OPTION PROFILING`, `OPTION TRACECACHE ON` must be placed as an early statement **inside** the program, not at the command prompt before `RUN`. `RUN` calls `ClearRuntime` which resets the cache feature flags to zero before the program starts.

- The `flags` bitmask can be changed at any time while the program runs — changes take effect on the next statement execution. Disabling a category mid-run does not erase previously compiled entries; re-enabling it reuses them.
- `LET`, `IF` conditions, `DO / LOOP` conditional clauses (`WHILE` and `UNTIL`), and `GOTO / GOSUB / RESTORE` with literal targets are eligible. `RESTORE` with a variable argument is never cached because its target depends on the variable's run-time value.
- **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.
- **String LET is cached** for scalar assignments in the forms `s$ = "lit"`, `s$ = t$`, `s$ = s$ + "lit"`, and `s$ = s$ + t$`. String arrays, string functions (e.g. `LEFT$`, `MID$`), and arbitrary concatenation (`a$ = b$ + c$`) are not cached. String *conditions* in `IF` or `DO / LOOP` 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). `OPTION CACHE DEBUG ON` reports these as `[TC-FULL]` . 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). `OPTION CACHE DEBUG ON` reports these as `[TC-FULL]` . 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

Note: `OPTION PROFILING ON` and `OPTION TRACECACHE ON` must appear as early statements **inside the program**, not at the command prompt. `RUN` calls `ClearRuntime` which resets both before the program starts.

```
' -- Profile only --
' (Place at the top of prog.bas)
OPTION PROFILING ON
' ... program body ...
END                ' report printed here

' -- Cache only --
' (Place at the top of prog.bas)
OPTION TRACECACHE ON    ' 64 slots, all features, ~13.5 KB

' -- Both together (recommended workflow) --
' (Place at the top of prog.bas)
OPTION PROFILING ON
OPTION TRACECACHE ON
' ... program body ...

' -- Selective features: numeric LET + IF only --
' (Use while diagnosing whether a category adds overhead)
OPTION TRACECACHE ON 64, 5 ' flags = 1 (LET_NUM) + 4 (IF) = 5
```

```
' -- Debug cache misses --
' (Place at the top of prog.bas)
OPTION CACHE DEBUG ON
OPTION TRACECACHE ON
' ... program body ...

' -- Restrict cache to hot SUBs --
' (Place at the top of prog.bas)
OPTION CACHE SUB Integrate, UpdatePos
OPTION TRACECACHE ON
' ... program body ...

' -- Cleanup (free memory during interactive session) --
OPTION PROFILING OFF
OPTION TRACECACHE OFF
```

Feature Flags Quick Reference

Value	Feature
1	Numeric scalar/array <code>LET</code>
2	String scalar <code>LET</code>
4	<code>IF</code> condition expressions
8	<code>DO / LOOP</code> <code>WHILE/UNTIL</code> condition expressions
16	<code>GOTO</code> and <code>GOSUB</code> jump targets
32	<code>RESTORE</code> jump targets
63	All features (default)

Memory Budget Summary

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

Interpreting Results: Decision Flowchart

Run with `OPTION PROFILING ON`

|

