Tak: Racing Code Through Time

A Detailed Re-Creation of David Johnson-Davies's Cross-Language Tak Function Benchmarks for Acorn BBC Microcomputer Systems

```
DEFFN TAK(X%,Y%,Z%)
  IF X% <= Y% THEN = Z%
  = TAK( TAK(X%-1,Y%,Z%), TAK(Y%-1,Z%,X%), TAK(Z%-1,X%,Y%) )
```

acheton1984, Issue 1.2, 20th March 2025

Table of Contents

Final page: 31

1 Introduction

In June 1986, Acorn User (No. 47, p179) published an article by David Johnson-Davies titled *"Six of The Best Against The Clock"*, which compared six programming languages using the Tak function. The full explanation is in the article, but he suggested that:

> *"it is difficult to imagine a language that performs badly on Tak being much use for anything!"*

This document chronicles my quest to reproduce the benchmarks from the original articles. It covers 2 articles, 10 programming languages, at least 6 spinoff Stardot threads, 4 machine configurations, 2 emulators, and numerous challenges and discoveries along the way.

The original article described the Tak function as a benchmark that tests efficiency in recursion, subroutine calls, and parameter passing. Starting from tak(18,12,6) the result is 7—after 63,609 calls, 47,706 subtractions, and a recursion depth of 18.

Two earlier publications were found which provide more background (links in B.2 Tak Function Background). The 1979 publication introduces Ikuo Takeuchi of NTT, after whom the Tak function is named. He created the function in 1978 to compare the performance of LISP systems.

The publication from 1985 explains that the version used today arose by accident when John McCarthy (designer of LISP) misremembered the original, altering it to return z instead of y. This publication may have been the inspiration for the AU articles, as it shares similarities with the Tak function description and the seed value (tak 18 12 6). It also includes this cautionary 1981 quote from Vaughan Pratt:

> *"Seems to me benchmarking generates more debate than information."*

The original languages tested were **S-Pascal**, **Forth**, **BCPL**, **ISO-Pascal**, **LISP**, and **micro-Prolog**, alongside **BASIC** and **Assembler** (making more than 6 languages!). Some were tested with both integer and floating-point variations.

A follow-up article, *"Testing The Tak"* from November 1986 (AU No. 52, p197) included several user submissions for the Z80 second processor—some of which performed very well, with small-C and Turbo Pascal ranking 2nd and 3rd place overall.

New variations of the original programs included caching results using arrays in BASIC or the ADDCL clause in micro-Prolog (known as memoization, where results of function calls are stored for reuse), which significantly improved performance. A BASIC variant using string lengths instead of numbers was also added.

This re-creation expands on the original tests by adding **COMAL**, with the program derived from BASIC, and **Beebug C**, adapted from Small-C.

Revisiting the original benchmarks began as an effort to explore the languages and match the original tests. Along the way, it has become a way to dig deeper into the results, better understand the computer, and question some of the original conclusions.

The full magazines are available online:

- Acorn User 47, June 1986
- Acorn User 52, November 1986

All project materials, including the setup guide, disc images, results spreadsheet, this analysis document, and feedback, are available in the project repository:
https://github.com/acheton1984/ReTestingTheTak

For questions, issues, or discussion, please open an issue on GitHub:
https://github.com/acheton1984/ReTestingTheTak/issues

2 Approach to Reproducing the Articles

2.1 Languages and Code

The starting point was to gather software and documentation, as well as extract the code examples from the article:

1. For each language, its ROMs and discs were obtained - primarily from Stardot, 8BS, or Flaxcottage (details in the Setup and User Guide).
2. The source code was OCR'd from the article, tidied up, and obvious errors were corrected.
3. The source text file was added to the disc image using Disc Image Manager (DIM), then EDIT was used to correct line endings (Global Replace ¦J/¦M).
4. The resulting file was either EXEC'd into the language and saved, or used as raw source.

Note: The disc images contain no original language code; they only include Tak and project support files. Language sources, instructions for assembling discs, and guidance for running Tak can be found in the "Setup and User" guide in the main repository.

Further corrections were made as the code was run (or compiled). Adjustments to display timings or sizes were added, all without affecting the Tak function:

Assembler	Line 625 was added to report the machine code size.
BASIC	The article had FNT(A,B,C), which are undefined variables, so line 90 from the stored-value version, calling FNT(18,2,6), was substituted. BASIC versions were matched to the OS and second processor. HiBASIC III was used for OS 1.20, HiBASIC IV for MOS 3.20, and the built-in BASIC IV (4r32) relocates automatically for MOS 3.50 The floating-point version was not included in the original article. To create this version, variables X%, Y%, and Z% were replaced with X, Y, and Z.
BCPL	The TIME handling code from p168 of the user manual was added to ensure timings are returned as unsigned (positive) values.
Beebug C	Adapted from Z80 Small-C.
COMAL	Adapted from the BASIC programs.
FORTH	New words added for timing, and additions to print size and run time.
ISO-Pascal	The FP version was adapted from the corresponding integer version, as it wasn't included in the original articles.

micro-Prolog The example using the ADDCL clause could not be made to run, so the ((test)) clause from the original was substituted.

S-Pascal Function key definitions were added to help with timing the program.

2.2 Hardware and Timings

The first article notes:

> *"All of the benchmarks were run on a 6502 second processor, using second-processor versions of the languages where available."*

Not having access to a BBC Micro Model B and an external 6502 second processor, comparative timings were initially obtained from emulators, with additional timings later taken from a real BBC Master 128. The configurations used are detailed below:

- MAME 0.272 and B-Em 2.2 running on a Mac, emulating a Model B with an external 6502 second processor.
- Master 128 with an Acorn 65C102 internal co-processor ('Turbo') running MOS 3.50.
- During the investigation into LISP's performance, this was expanded to include:
    - Master Turbo with Multi OS, running a modified OS 1.20 and MOS 3.20.
    - MAME emulating a Master Turbo running MOS 3.20 and MOS 3.50.

Timings from a real BBC B and second processor were kindly provided by BeebMaster.

All programs are able to report their own run times, some in centiseconds and others in seconds. For consistency, results are given in seconds only.

Run times are based on a single recorded execution, rather than an averaged result. However, repeated tests on both real hardware and emulators consistently produce results within a centisecond or two of the recorded values, suggesting a high degree of consistency.

2.3 Sizing It Up

The second article explains:

> *"the code size is the number of bytes occupied by the actual Tak function, excluding the surrounding program and any interpreter that might be present in the case of interpreted languages. The code [size] was determined by compiling the program both with and without the function definition and taking the difference".*

Calculating code size was a little trickier than implied in the quote, as it's not possible to simply compile code with a key function missing. To follow the spirit of measuring compiled code size with and without tak(), the following approach was adopted—

For compiled languages, two dummy variants of the TAK program were created, each returning 0 if run:

- TAK0 replaces the call to tak(18,12,6) with 0.
- TAK0-, derived from TAK0, removes the entire tak() function.

For interpreted languages, a single non-runnable variant was created:

- TAK-, derived from TAK, removes the entire tak() function

The difference in size between the compiled output generated from TAK0 and TAK0-, or between TAK and TAK- for interpreted languages, therefore, represents the size of the tak() function.

Code size for some languages can be obtained from their built-in features rather than relying solely on file sizes. However, these results were uncovered too late for inclusion here and are instead noted in the respective README files. Sizes using BASIC's TOP, COMAL's SIZE, FORTH's HERE, and LISP's (RECLAIM) with debugging enabled are included. However, micro-Prolog's SPACE primitive, limited to 1 Kb resolution, was unsuitable.

2.4 Results Spreadsheet

As heatmaps and z-scores are new to me, a brief explanation of how they work and how they were used in the analysis follows.

A heatmap is a visual representation of data where values are depicted as colours, making it easy to spot patterns, trends, and outliers across different sets of results. These heatmaps use z-scores calculated from the original timings.

Z-scores (on Wikipedia) measure how far a particular value deviates from a reference value, expressed in standard deviations. They are calculated as:

```
z = (value - reference value) / standard deviation.
```

Although z-scores are often based on a mean or median value, for this data, the reference is typically a specific machine's result. This means:

- A positive z-score means a value above the reference (longer time, slower).
- A negative z-score means a value below the reference (shorter time, faster).

Within the spreadsheet, heatmaps applied to z-scores and percentage differences follow the same colour scheme:

- Negative z-scores are redder (indicating it's running hot/quicker)
- Positive z-scores are bluer (indicating it's running cold/slower).

3 The Results Spreadsheet

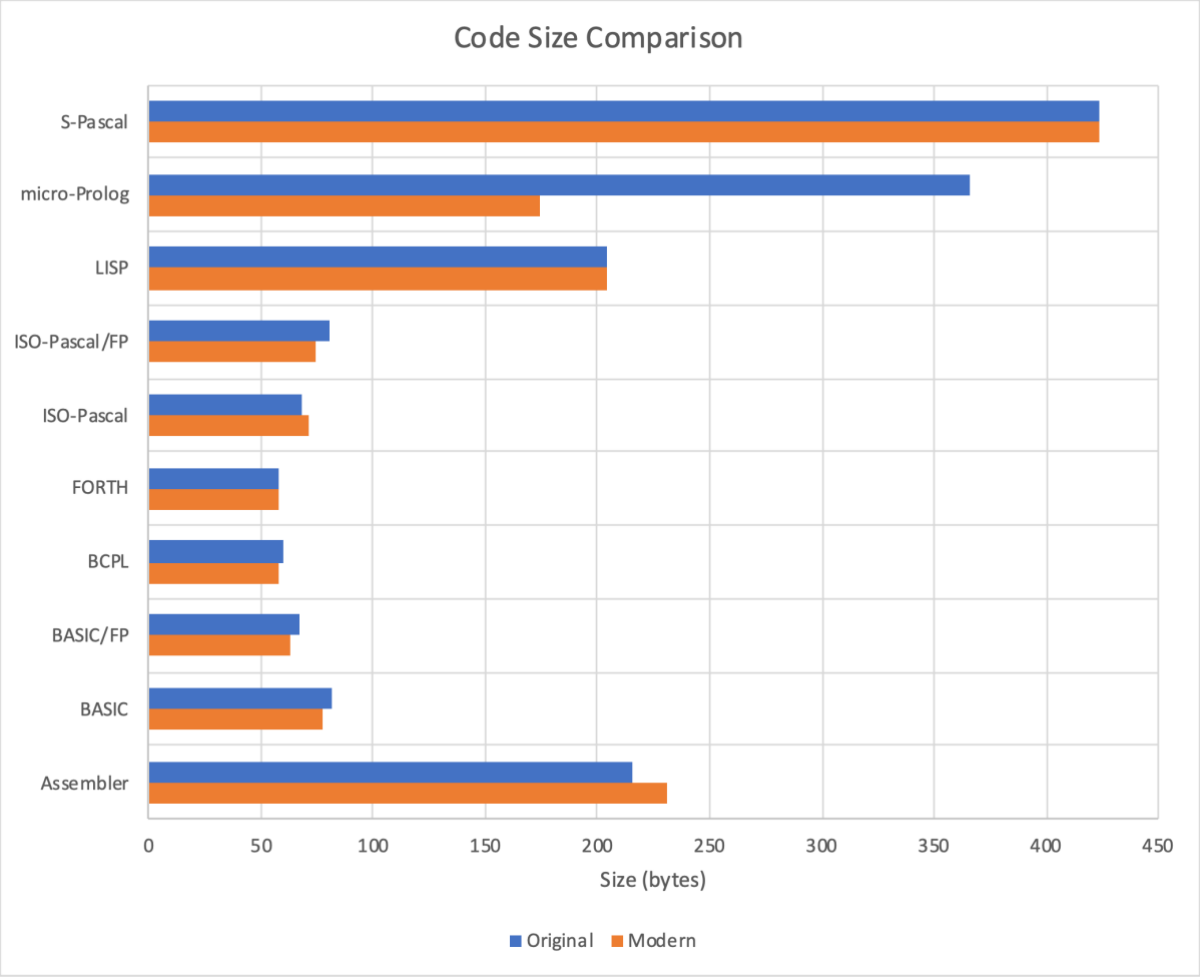
This section explains the master results spreadsheet, which includes code sizes, full timing data, graphs, and heatmaps.

Graphs with timings are presented on a logarithmic scale (base 3), which somewhat mimics the scatter graph from the first article and helps reveal details at the small end of the scale that would otherwise be lost.

3.1 Sizings Sheet

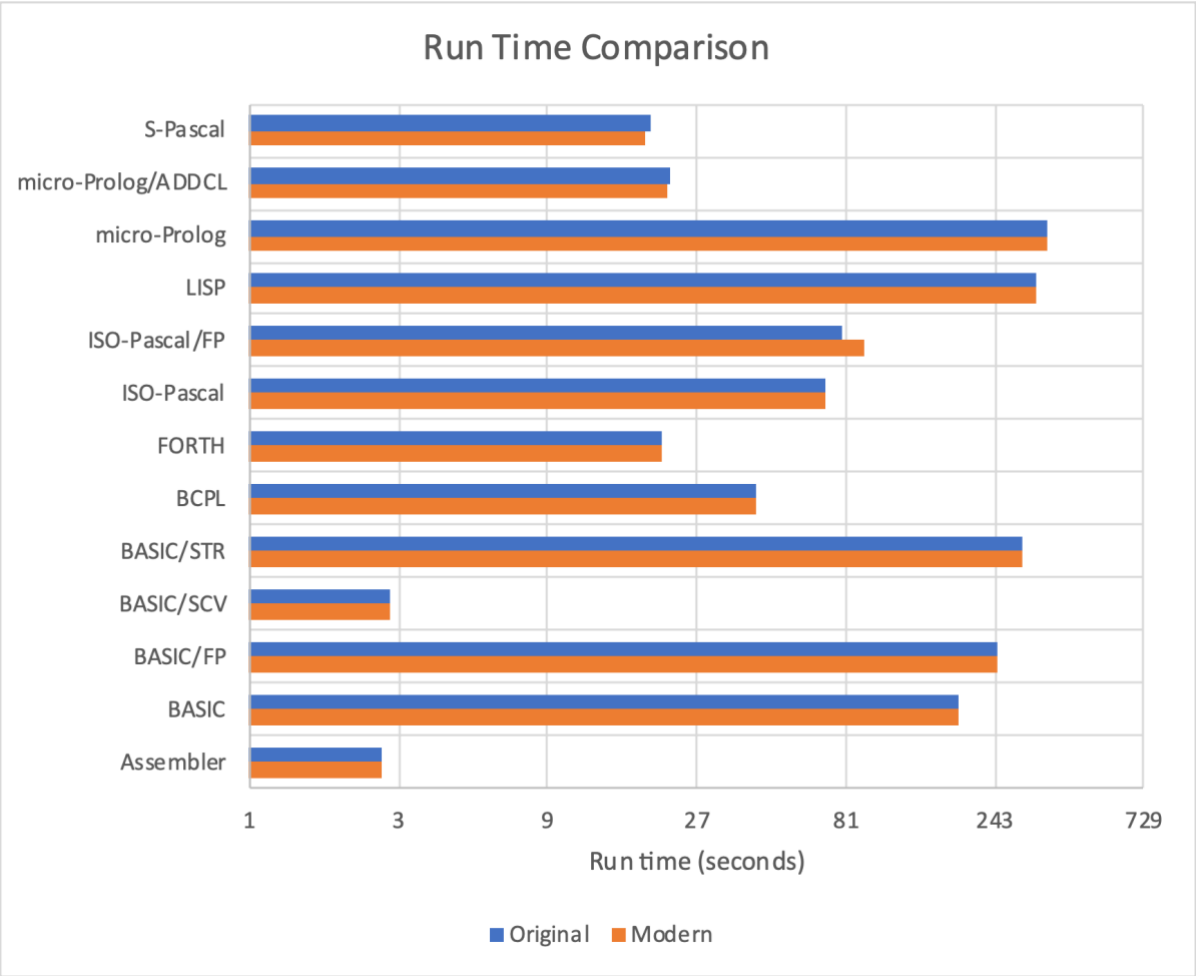
The sheet records the code sizes from the original articles and the modern re-creations. A copy is reproduced in Appendix A.1 Program Sizings.

It also contains a grouped bar chart, reproduced below, comparing the original and modern code sizes where data exists for both.



3.2 Timings Sheet

This sheet includes full timing data, shown in seconds, for all systems and configurations – an extract is included in Appendix A.2 Program Timings. The spreadsheet also features a grouped bar chart comparing original and modern run times, which is reproduced below:



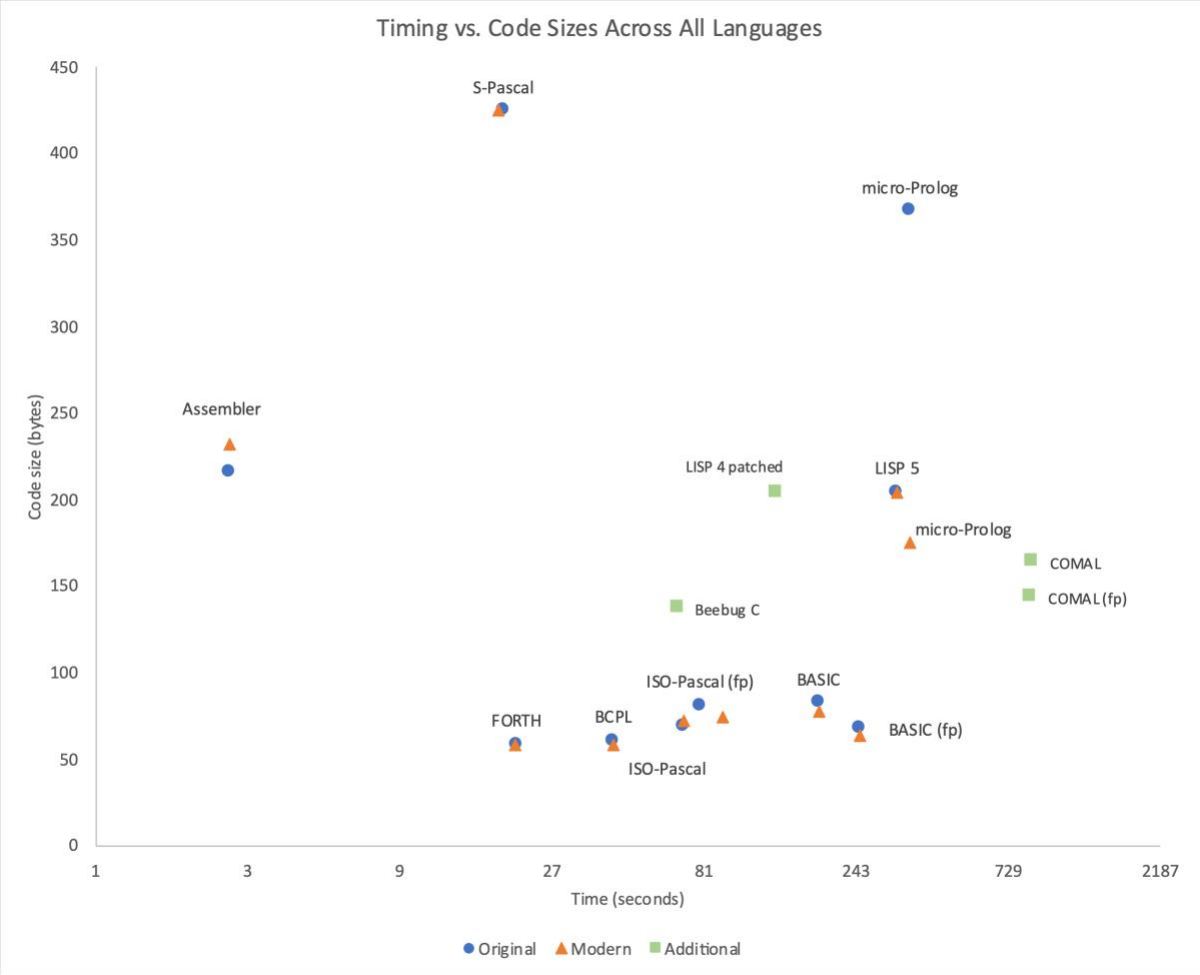
3.3 Scatter Graph Sheet

The graph on this sheet re-creates the original article's scatter graph of timings and code sizes, though the y-axis scale differs slightly from the original.

The graph provides a clear visual representation of the variations between:

- The original results (blue circles)
- The corresponding re-created versions (orange triangles)
- New languages or versions added to the re-creation (green squares)

Significant differences include ISO-Pascal FP's slower modern performance, LISP 4 far outperforming LISP 5, and micro-Prolog's smaller modern size.



3.4 HeatMaps Sheet

This sheet includes several heatmaps, each exploring a different area of the results, as explained below. In addition to the main heatmap, which gives a view across all languages and systems, there are more specific heatmaps that focus on LISP, emulator accuracy, and variations in BASIC across OS versions.

As a reminder, heatmaps use colours to represent performance. Except for LISP, the faster times are shown in red (negative z-scores and percentages), while slower times are shown in blue (positive z-scores and percentages).

3.4.1 Main Heatmap

The main heatmap compares performance across all tested languages and systems. To meaningfully show the diversity of results, the table is split into two parts, each using a different piece of real hardware as its baseline, permitting a clear basis for comparison.

- The left side (columns C to F) uses a Model B (column D) as the baseline.
- The right side (columns G to K) uses a Master 128 with MOS 3.20 (column H).

In both cases, the reference columns (D and H) are zero, as they represent the baseline performance for their respective systems. All other values show deviations relative to these baselines, allowing unusual results to stand out clearly.

Using two baselines helps balance the comparisons across hardware. For instance, if only the BBC B results were used as the baseline, nearly all Master 128-based results (except LISP) would appear as medium to dark reds, making it harder to discern any meaningful trends.

This choice of baselines has a side effect: the original results (column C) are shown, perhaps counterintuitively, relative to the modern baseline. For example, the noticeably red cell at C13 shows that the original was faster than the re-creation, meaning the re-creation is slower than the original.

Language / Variant	Filename	Original	BBC B, OS 1.20 6502 2nd proc.	B-Em BBC B (OS1.20)	MAME BBC B (OS1.20)	Master 128 Mod. OS 1.20	Master 128 MOS 3.20B	Master 128 MOS 3.50B	MAME M128 MOS 3.20	MAME M128 MOS 3.50
Assembler	TAKAsm	0.00	0.00	0.16	-1.94	0.00	0.00	0.00	-1.04	-1.56
BASIC	TAK	0.09	0.00	0.04	-1.96	2.36	0.00	-0.45	-1.52	-1.95
BASIC	TAKfp	0.11	0.00	0.00	-1.96	7.18	0.00	-1.27	-1.53	-2.76
BASIC	TAKscv	-0.33	0.00	-0.17	-2.15	2.93	0.00	-0.50	-1.00	-2.00
BASIC	TAKstr	-0.04	0.00	0.07	-1.99	4.36	0.00	-0.44	-1.52	-1.94
BCPL	TAK	0.22	0.00	-0.90	-1.97	0.00	0.00	-0.03	-1.52	-1.52
Beebug C	TAK		0.00	-0.21	-1.83	0.00	0.00	0.00	-1.53	-1.53
COMAL	TAK		0.00	0.16	-1.65	0.00	0.00	0.00	-1.51	-1.51
COMAL	TAKfp		0.00	0.15	-1.65	0.00	0.00	0.00	-1.51	-1.51
FORTH		0.47	0.00	-0.30	-1.84	0.00	0.00	0.00	-1.52	-1.45
ISO-Pascal	TAK	0.21	0.00	-0.09	-1.95	0.00	0.00	0.00	-1.52	-1.52
ISO-Pascal	TAKfp	-2.15	0.00	-0.05	-0.71	0.02	0.00	0.01	-1.50	-1.50
LISP 5	TAK	-0.12	0.00	1.70	-0.59	-2.14	0.00	7.19	-0.48	6.72
LISP 5 (no tube)	TAK		0.00	-1.92	-1.44	0.14	0.00	8.79	0.16	8.97
LISP 4 patched (tube)	TAK		0.00	0.02	-1.72	0.00	0.00	0.00	-1.51	-1.51
LISP 4 patched (no tube)	TAK		0.00	-1.87	-1.55	2.63	0.00	0.35	0.16	0.50
micro-Prolog	TAK	0.03	0.00	-0.06	-2.01	0.01	0.00	-0.03	-1.51	-1.51
micro-Prolog	TAKacl	0.22	0.00	0.00	-1.91	0.62	0.00	-0.13	-1.64	-1.64
S-Pascal	B.TAK	1.02	0.00	-0.07	-1.42	0.00	0.00	-0.07	-1.50	-1.57

3.4.2 LISP Performance Heatmap

The LISP Performance heatmap compares four test configurations: the original LISP 5 ROM with a second processor (shown in the top row), LISP 5 without a second processor, and the patched LISP 4 ROM both with and without a second processor.

Slower times appear in green (positive z-scores), while faster times are shown in yellow (negative z-scores). The heatmap includes calculations in cells C25-C27, which show the average percentage performance increase relative to the baseline in C24.

Language / Variant	Filename	Original	BBC B, OS 1.20 6502 2nd proc.	B-Em BBC B (OS1.20)	MAME BBC B (OS1.20)	Master 128 Mod. OS 1.20	Master 128 MOS 3.20B	Master 128 MOS 3.50B	MAME M128 MOS 3.20	MAME M128 MOS 3.50
LISP Performance (Original/Patched ROMS, with/out Tube)										
LISP 5 (tube)		333.60	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LISP v5, no tube		17.0%	-0.67	-0.82	-0.60	-0.34	-0.62	-0.55	-0.60	-0.54
LISP patched v4 (tube)		66.5%	-2.31	-2.33	-2.30	-2.26	-2.34	-2.30	-2.33	-2.29
LISP patched v4, no tube		38.8%	-1.47	-1.56	-1.40	-1.09	-1.29	-1.36	-1.27	-1.34

3.4.3 Emulator Performance Heatmap

The Emulator Performance heatmap compares the performance of each emulator configuration to its corresponding real hardware.

Each cell shows the percentage difference in run time from the real hardware for each test (not a z-score), with negative values meaning shorter times (i.e., faster performance). The final row of each column shows the average percentage difference for each system emulated.

Language / Variant	Filename	B-Em BBC B (OS1.20)	MAME BBC B (OS1.20)	MAME M128 MOS 3.20	MAME M128 MOS 3.50
Emulator Relative Performance					
Assembler	TAKAsm	0.38%	-4.55%	-1.04%	-1.56%
BASIC	TAK	0.09%	-4.45%	-1.52%	-1.51%
BASIC	TAKfp	0.00%	-4.44%	-1.53%	-1.51%
BASIC	TAKscv	-0.35%	-4.61%	-1.00%	-1.51%
BASIC	TAKstr	0.15%	-4.19%	-1.52%	-1.51%
BCPL	TAK	-2.23%	-4.86%	-1.52%	-1.49%
Beebug C	TAK	-0.53%	-4.55%	-1.53%	-1.53%
COMAL	TAK	0.39%	-4.11%	-1.51%	-1.51%
COMAL	TAKfp	0.37%	-4.13%	-1.51%	-1.51%
FORTH	F.TAK	-0.67%	-4.19%	-1.52%	-1.45%
ISO-Pascal	TAK	-0.22%	-4.42%	-1.52%	-1.52%
ISO-Pascal	TAKfp	-0.33%	-4.63%	-1.50%	-1.51%
LISP 5	TAK	5.28%	-1.82%	-0.48%	-0.44%
LISP 5 (no tube)	TAK	-0.23%	-0.17%	0.16%	0.17%
LISP 4 patched (tube)	TAK	0.05%	-4.17%	-1.51%	-1.51%
LISP 4 patched (no tube)	TAK	-0.34%	-0.28%	0.16%	0.15%
micro-Prolog	TAK	-0.13%	-4.69%	-1.51%	-1.48%
micro-Prolog	TAKacl	0.00%	-4.71%	-1.64%	-1.51%
S-Pascal	B.TAK	-0.22%	-4.68%	-1.50%	-1.50%
Average		0.08%	-3.88%	-1.24%	-1.28%

3.4.4 BASIC Relative Performance heatmap

The BASIC Relative Performance heatmap illustrates the improving performance of different BASIC versions (BASIC III, BASIC IV, and BASIC IV (4r32)) running on the same hardware.

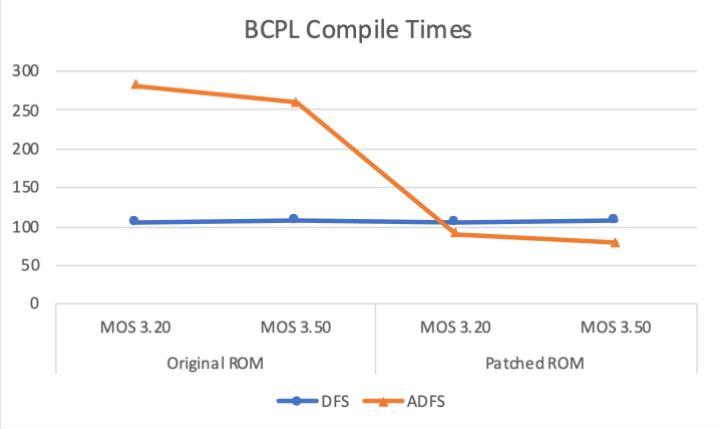
- For each BASIC test, the average run time is calculated across the three OS versions, with each cell showing the percentage difference from the specific test's average.
- The "Version average" row shows the percentage difference from the overall average across all tests for each version of BASIC.
- The "% faster" row highlights the performance improvement of BASIC IV on MOS 3.20 and 4r32 on MOS 3.50 over BASIC III on OS 1.20.

Language / Variant	Filename	Master 128 Mod. OS 1.20	Master 128 MOS 3.20B	Master 128 MOS 3.50B
BASC Relative Performance				
BASIC	TAK	1.57%	-0.56%	-1.01%
BASIC	TAKfp	5.70%	-2.23%	-3.47%
BASIC	TAKscv	1.82%	-0.66%	-1.16%
BASIC	TAKstr	2.75%	-1.16%	-1.59%
Version average		2.96%	-1.15%	-1.81%
% faster than BASIC III			4.1%	4.8%

3.5 BCPL ADFSvDFS Sheet

This sheet records BCPL compile times on DFS and ADFS, both with and without the patched (ADFS-aware) BCPL ROM.

A simple graph ("BCPL Compile Times") illustrates the results.



4 Findings

While the original goal was simply to replicate the benchmarks; unexpected results and issues soon emerged, each prompting further investigation—sometimes with help from Stardot members.

These findings are documented here and organised as follows:

- Inferences: a descriptive summary of notable patterns and observations, along with some personal reflections.
- Subsequent sections: a deeper dive into specific findings, providing explanations and potential impacts. The impact section has been left out when there is little practical impact or no changes to the results ranking. Likewise, references are only included where relevant.

These explanations reflect my attempts to interpret the data and discussions from Stardot and may not be fully accurate. Further analysis, alternative interpretations, and new insights are encouraged to refine or expand the results. Constructive feedback is always welcome.

4.1 Inferences

At first glance, the findings seem to reveal little of real surprise about the languages tested. Those closest to machine code (Assembler, FORTH, S-Pascal) were fastest, followed by the compiled languages (BCPL, Beebug C, ISO-Pascal) with the interpreted languages as the slowest (BASIC, COMAL, LISP, micro-Prolog). Yet with careful planning and judicious use of caching even interpreted languages, such as in BASIC SCV and micro-Prolog ADDCL, can challenge the top-performers.

For an easy performance win, S-Pascal stands out: while limited to small programs and a subset of Pascal, it still delivers one of the fastest run times. BCPL delivered solid results but has a slow compile / link cycle leaving me wondering how the authors of the Domesday System managed their much larger code base.

Code size, however, is no indicator of performance with the largest (Assembler and S-Pascal) also being the fastest. The compiled languages BCPL and ISO-Pascal both strike a good balance of smaller code size and relatively good performance.

The elephant in the heatmap is, of course, LISP. Its colours seem “all over the place” and, counterintuitively, it runs faster *without* the second processor – which makes me wonder why this wasn’t spotted and updated after first being published. A modern patch upends the original findings that BASIC was the fastest interpreted language, with that position now taken by the patched LISP.

It’s easy to speculate whether patches to other languages could have brought similar improvements; however, debugging and code profiling are far easier today with modern tools and emulators than they were at the time.

Although BASIC is compact and convenient, with its heatmap showing a consistent improvement across versions, it still doesn't match the speed of compiled languages, taking around 2.6× as long as ISO-Pascal to calculate Tak. As the patched LISP can now outperform it, the original article's claim that BASIC was "not far behind" BCPL and Pascal seems increasingly tenuous.

In partial mitigation of BASIC, it has the overhead of processing 32-bit integers, while most other languages use only 16-bit. That said, ISO-Pascal also uses 32-bit integers and remains "noticeably far ahead" of BASIC's performance.

The final note on specific languages goes to COMAL which was published two years before the articles were written, so its omission could be seen as curious. One possibility is that it's by far the slowest language and very sensitive to small code changes. More prosaically, it may simply have been left out due to space constraints in the article.

On the whole, the languages show only small differences in size or timings between the re-creations and the originals. However, ISO-Pascal's FP performance and micro-Prolog's size stick out for the wrong reasons. Further improvements in accurately reproducing the originals are unlikely without a clearer understanding of their code, setup, and timing methods—details now lost to history.

It's also possible to infer something about the OS from the language results. For most languages, timing remains consistent across the different OS versions on the Master 128, suggesting that the OS performs its role (handling OS calls or interrupts) with no measurable changes between versions. There are exceptions:

- BASIC: Improvements most likely reflect changes in the language itself.
- LISP: Slower (blue) entries for MOS 3.50 point to increased overhead in the OSWORD handler, an issue made apparent by LISP's frenetic OSWORD calls.
- micro-Prolog: Results imply there are some interactions with the OS that benefit from improvements in later OS versions.

Another non-language issue, though only lightly explored, is that ADFS can be noticeably slower than DFS, particularly for smaller write operations.

It seems likely these languages were originally designed for the Model B with DFS and were not always updated for later systems. More modern features, such as shadow screen modes and ADFS can cause issues. While this doesn't prevent use on a Master 128 or updated Model B, some features may need to be restricted or require workarounds.

Finally, while emulators have been invaluable for developing modern re-creations of these tests, their accuracy varies. On the whole, B-Em runs at a comparable speed to real hardware, while MAME can run up to 4.5% faster.

4.2 Mind the Gaps: Re-Creating the Unknown

The re-created results vary in both code sizes and run times compared to the originals. In total, the modern version is 190 bytes smaller, and the run time is 12 seconds slower than the original. Notable outliers include micro-Prolog's code size and ISO-Pascal FP's run time. Without fuller documentation of the original code, setup, and timing methods—details now lost to history—definitive explanations for these discrepancies remain elusive, and the possible causes below are purely speculative:

- Development versions of the code may have been used during testing, differing from the published listings. There are hints of this in the printed BASIC example, which includes uninitialized variables, and the micro-Prolog ADDCL program, which allows tak() to be run with user parameters instead of the usual (18,12,6) fixed values.
- The exact original hardware configuration and software versions are unknown, so the re-creations may not fully reflect the original setup. Timing-specific factors include:
    - Attached hardware or software might generate extra interrupts, cause bus slowdowns, or intercept and add to OS calls.
    - Differences in hardware speeds, even if within specification, could lead to timing discrepancies.
- Potential misprints in the original articles could affect exact replication. Though only one clear print error has been spotted—the missing last line of the small-C code—no corrections were printed in the follow-up article.
- The methods described in the articles for measuring size and timing leave room for interpretation, e.g., sizes measured using the implausible method of "compiling the program […] without the function," and FORTH's timing method unexplained.
    - The use of a stopwatch for some timings introduces potential inaccuracies.
- Floating-point versions of two programs weren't printed, so assumptions made during the re-creation may not accurately reflect the originals.

Whilst there are clearly differences, it's hard for me to say if they have any real impact. On the one hand, the re-creations are not entirely faithful to the originals as published, reflecting the difficulties of interpreting documentation that could have been more detailed and "unknown unknowns". On the other hand, these differences don't meaningfully alter the rankings or conclusions, so their broader significance is limited. However, this does highlight the importance of including detailed records to make it possible to accurately re-create any computer-based work.

The specific two cases below share much of the explanation and impact just discussed, but stand out because each has additional context that may contribute to their differences.

4.2.1 micro-Prolog's not-so-micro Code Size

micro-Prolog's code size was reported as 366 bytes in the original article's results table, but the printed program is only 236 bytes long, and the modern version has the Tak function as just 169 bytes.

Explanation: The exact cause is unknown, but in addition to the general factors, some micro-Prolog-specific causes may explain the unexpected size:

1. The reported size of 366 bytes may be wrong, as it exceeds the printed code size, or it may belong to an unseen version of the code.
2. The SAVE command in micro-Prolog saves the entire workspace, including all clauses listed under LIST ALL, so it may have included additional unseen code.
3. Extensions such as SIMPLE or MISTI might also affect save sizes, if they were loaded during testing.

4.2.2 ISO-Pascal's Floating-Point Sloth

The floating-point version of ISO-Pascal is 16% slower than the original. This is the largest performance difference among the tests.

Explanation: The exact cause is unknown, but the original article did not include the floating-point (FP) version, so the integer version was adapted by simply changing "integer" to "real" (as the only available FP type). Therefore, this re-created version may differ from the unseen original. Testing with both disc-based and ROM-based versions produced identical results, eliminating that as a possible source of difference.

4.3 Findings While Coding

These "findings while coding" surfaced during the process of trying to use the languages to re-create the tests.

4.3.1 Published Code Errors

Syntax errors or incomplete code were found in the original articles.

Explanation: As outlined in the general code size/timing factors, the errors hint that development code may have been printed instead of a finished version. Some programs lacked code for timing or sizing.

Impact: Not all code, as printed, would run or provide size and timing results without amendment.

Reference: Further details can be found in 2.1 Languages and Code.

4.3.2 BCPL, SAG, ADFS and Execution Addresses

The BCPL Stand-Alone Generator (SAG) utility FIXCIN creates a runnable file with the wrong execution address under ADFS.

Explanation: The FIXCIN utility writes the load and execution addresses separately. The underlying OSFILE call works differently between DFS and ADFS, with ADFS overwriting parts of the command block between calls which results in the

wrong execution address being written out. An ADFS-aware patched version, FIXCIN2, was provided to resolve this issue.

Impact: Although valid code is created by FIXCIN, it will not execute as *RUN will attempt to jump to code in the wrong part of memory.

Reference: BCPL Standalone Generator on ADFS?

4.3.3 COMAL's Unusual Parameter Lifecycle

COMAL's simple conversion from BASIC initially ran 'too quickly' and returned the wrong results.

Explanation: COMAL handles function parameters differently from the other languages. The calling function's parameters are overwritten during each call to tak(), causing them to change as the Tak function is evaluated. As a result, each call to tak() in the same expression starts with different values.

Impact: To ensure parameters remain unchanged throughout an expression's evaluation, temporary variables (tx#, ty#, tz#) were used to hold the original values, and the FUNC was declared as CLOSED to make the temporary variables local. Care must, therefore, be taken when passing parameters with the same name, as changes in one function may inadvertently affect another.

Reference: Comal from Basic, Recursive Variable Scope?

4.3.4 The March of Incompatibility

Not all languages were found to be compatible with all newer features of later systems; for example, problems were encountered with Tube ESCape flag handling, Shadow Modes, MOS 3.50, and ADFS. Some of these issues are discussed in more detail in separate findings.

Explanation: These languages were likely targeted at the early BBC Micro and were not always updated to accommodate later changes in hardware or software.

Impact: Before using a language on a system other than a Model B with DFS, it's worth checking whether a patched version or workaround is available for your setup. If problems arise during use, either stop using the conflicting feature or ask for help on Stardot.

Reference: Suggestions that the Tube protocol, for example, wasn't fully evolved can be found in "LISP: Slower with Co-Pros and MOS 3.50"

4.4 Findings During Timing Review

Once the re-creations were completed and working, studying the results helped uncover many possibly unknown facets of the languages.

4.4.1 Performance Improvements Across BASIC Versions

BASIC shows a steady performance improvement across newer versions, as shown in cells G3-I6 of the main heatmap and in the separate BASIC performance heatmap. Compared to BASIC III, BASIC IV is 4.1% faster, and 4r32 is 4.8% faster.

Explanation: By using the "native" BASIC for each system, the results highlight changes in the BASIC interpreter itself rather than solely reflecting the underlying system. BASIC IV benefits from the 65C02's more compact and faster instruction set, while 4r32 has further improvements, such as updated trigonometric functions (though unused in these tests).

Impact: Although the BASIC versions do not affect the overall results, it is important to know which version is used in any tests involving BASIC.

4.4.2 BCPL: Compiles Slower on ADFS than DFS

BCPL compile times on ADFS, at 281 seconds, were noticeably slower than on DFS, at 105 seconds.

Explanation: A pair of test discs (one DFS, the other ADFS—available from the project GitHub repository) were created to measure and confirm the finding. A Stardot user suggested that, as BCPL predates ADFS, it might fall back on slower file system calls (OSBGET/OSBPUT) instead of using the more efficient OSGBPB. A patched version of the ROM was already available elsewhere to address this issue.

Impact: With the patched ROM, compile time on ADFS was reduced to 90 seconds, making it over 3x faster.

References: The initial discussion - Is ADFS slower than DFS?
The patched rom - Sideways ROM Software Development

4.4.3 COMAL: Floating-Point Faster Than Integer

COMAL, unlike all other languages tested, was quicker at floating-point than at integer.

Explanation: The performance difference is likely due to the length of variable names, with integer variables using three characters (e.g., xx#) and floating-point variables using two (e.g., xx).
Profiling from Stardot showed that only 0.39% of the runtime was spent on arithmetic, prompting the creation of two test programs to explore other factors.

- VarSpd showed little difference in performance between integer and floating-point operations, but function name length had a clear impact: "int_very_long_func_name" took 46 seconds, while one named "a" completed in 31.
- TAKffp was created by doubling the first letter of each variable name in TAKfp. This increased the execution time from 615 seconds for TAKfp to 630 seconds for TAKffp.

The original article noted a similar effect in BASIC: *each additional character in a function name adds 18 microseconds per call*.

Impact: Variable and function name length appear to have a major impact on execution time, possibly more so than in any other language tested, showing the need to pick shorter names where possible for best performance.

Reference: COMAL FP Faster than Integer?

4.4.4 LISP: Faster Without the Second Processor

LISP was about 20% faster without the second processor.

Explanation: LISP uses the OSWORD 5 call (which is documented as slow) around 3,000 times per second to check the status of the ESC flag on the host, creating a significant bottleneck across the Tube. Ideally it should just check the local copy at &FF, but a former Acorn employee thought LISP may have predated this 'standard' tube functionality A patched ROM, LISP407esc2, vastly improves performance by reading the local copy of the flag, making it 67% faster across the tube.

Impact: Had LISP been patched by the time of the original tests, LISP would have overtaken BASIC, and overturned the original article's findings that BASIC was the fastest interpreted language.

Reference: LISP: Slower with Co-Pros and MOS 3.50

4.4.5 micro-Prolog: Faster on Newer OS Versions

micro-Prolog's performance improves slightly across the three versions of the OS running on the Master 128, shown as a shift from pale blue to pale red in the main heatmap.

Explanation: While most languages tested showed consistent run times across the OS versions on the Master 128, micro-Prolog's performance variation suggest that:

1. Its timings are not purely computational and must involve interactions with the OS.
2. Whatever (currently unknown) OS call it relies on has become more efficient in the later OS versions.

Impact: The exact interactions with the OS are unclear and could be an avenue for further investigation.

4.5 Operating Environment

Alongside the language-specific findings, the results also revealed unexpected results in the OS, filesystem, and emulator results that warranted further investigation.

4.5.1 ADFS: May Feel Slower Than DFS

During language testing, ADFS appeared slower than DFS for some operations.

Explanation: The performance difference between ADFS and DFS depends on the operation. ADFS is capable of faster reads and writes than DFS due to its

higher information density per track. However, for smaller write operations, ADFS incurs a time penalty because it must keep its larger on-disc catalogue and free space maps up-to-date.

Impact: The difference can be more noticeable with compiled languages, where frequent movement of executables, code and data to and from the disk occurs during compile and link steps.

Reference: Is ADFS slower than DFS?

4.5.2 MOS 3.50's Circuitous OSWORD Handling

MOS 3.50 was over 7% slower than MOS 3.20 during LISP testing, as shown by the deep blue in the LISP 5 rows of the heatmap.

Explanation: Two changes in MOS 3.50 each introduce small additional overhead when handling OSWORD calls:

1. The OSWORD routine table was moved to the Terminal ROM, requiring extra time to page it in and out.
2. Refactoring of a part of the OSWORD code has added extra CPU cycles.

Impact: In day-to-day use these changes are unlikely to be noticeable. However, any code making frequent OSWORD calls will experience measurably slower performance in MOS 3.50.

Reference: LISP: Slower with Co-Pros and MOS 3.50

4.5.3 General Impact of OS Versions on Performance

Apart from MOS 3.50's OSWORD handling outlined above, for most languages tested on consistent hardware, MOS version changes generally have no measurable effect on run time.

Explanation: As the Tak function is purely computational, requiring no I/O except printing results at the end, the language should not need to make OS calls. In these cases, the OS is only servicing interrupts, and there is no measured difference in run times across different OS versions.

4.5.4 Emulator Performance

The 3.4.3 Emulator Performance Heatmap shows that the emulators do not perform exactly as the real hardware, with B-Em being the more accurate for a Model B and MAME's performance varying by system, with its Master 128 emulation being the more accurate.

Explanation: Accurate emulation is inherently challenging, requiring the reproduction of not only the timing and nuances of the original hardware (some of which may remain unknown) but also the allocation of resources in modern code to mimic what was originally handled by parallel components.

Impact: Given these imperfections, it's important both to choose the right emulator for your setup and understand any variations in its performance.

4.5.5 Co-Pro Cycle Thief

An indirect finding, sparked by posting MAME's performance results to Stardot, reveals that neither Acorn's "cheese wedge" second processor nor "Turbo" co-processor run quite as many CPU cycles per second as their full published speeds might suggest.

Explanation:	The CPU in any second processor is periodically paused for a cycle to make time for its DRAM to be refreshed. This means that the 3 MHz "cheese wedge" effectively runs at 2.93 MHz and the 4 MHz "Turbo" at 3.94 MHz. The explanation and timing analysis provided on the Stardot thread make this behaviour clear.
Impact:	For users of real hardware, there's no impact at all as the processors are performing exactly as designed. However, it's a key piece of information for anyone maintaining an emulator, or writing timing-critical software.
Reference:	MAME: How to ... ?

5 Future Considerations

5.1 Open Issues

During the re-creation of the tests, several problems or findings were spotted but remain unresolved. While none require immediate attention, addressing them could resolve issues and improve overall reliability

- The BCPL stand-alone version prints “tak(18,12,6)=“ but never the result (7). A debug version could look for problems with stack depth, variable scope, library bugs, etc.
- LISP timings still vary across OS versions, suggesting it continues to make OS calls. Identifying these calls could help further improve LISP’s performance.
- Two different patched versions of LISP exist: one fixes Shadow RAM compatibility, and the other resolves ESCape flag detection across the Tube. A combined patch addressing both issues would be beneficial.
- micro-Prolog’s timings improve across OS versions. Finding why would reveal which OS calls are being used and whether the language could be further optimized.
- Corruption occurred when saving to the S-Pascal disc on MAME/8271. Further investigation may find if it was caused by the 8271 disc interface, a specific DFS version, or MAME.
- Emulator timings differ from real hardware. Understanding and fixing the causes could improve both accuracy and help correct hidden issues or incorrect assumptions in the emulation.

5.2 Further Exploration

There are many potential avenues for further exploration (including none!), but here are some ideas that have emerged during this project:

Further exploration of existing tests:

- Subjectively, HIBASIC III seems to run a few centiseconds slower than BASIC II on the second processor. Was this feeling accurate, and what might be causing it?
- A consistent version of BASIC (e.g., HIBASIC III) run on all systems might reveal more about differences in the underlying system than just the language’s improvements.
- Explore whether compiling BCPL with NONAMES or using PACKCIN from the Stand-Alone Generator improves performance and reduces code size.
- Explore the use of other tools or methods to calculate more precise code sizes.
- Test whether disabling A-D conversions (and suppressing their associated interrupts) has any measurable effect on performance.

Additional systems:

- Run the tests without a second processor to identify any new unexpected results. (Though this might be slow!).
- Does B-Em configured as a Master 128 reveal any additional B-Em anomalies?

- Test with other emulators to compare their accuracy against real hardware.
- Fuller testing of the languages on different configurations (e.g., OS versions, ADFS, Econet, Shadow RAM) may reveal other problems requiring patches or workarounds.

New horizons:

- Re-create the original Z80-based tests from the second article.
- Adapt the tests for BASIC, LISP, Prolog, and C for the ARM Evaluation System.
- Additional 6502-based languages or implementations (e.g. HCCS FORTH, Oxford Pascal, PLASMA, etc.).
- For any of the languages analysis with modern tools, code profiling, and disassembly may yet expose a rich seam of improvements just waiting to be unearthed.

6 Conclusion

How wrong I was to think re-creating the Tak benchmarks from *Acorn User* would be as simple as cutting and pasting some code, downloading the languages, making disk images, and running a few commands. It was supposed to be the work of just a few days.

Although this project set out to simply reproduce the original benchmarks, things didn't go quite as planned. The scope soon spiralled out into more languages, more configurations, and more investigation, because each look at the expanding table of results seemed to point toward yet another line of enquiry.

Finding the right language software and documentation was the first challenge, as software preservation can be fragmented, and different sources can hold differing or mixed versions of the same tools. Getting it all to run, debugging errors, and timing the results was another. But in the end, the tests were all successfully replicated.

Along the way, I uncovered more than I ever expected to, just about the languages: LISP's inefficiencies, COMAL's sensitivity, BASIC's tarnished crown. Also, the environments they run on: how MOS 3.50 and ADFS can be slower than their predecessors, that emulators are imperfect. And even some mistakes in the original articles.

However, spotting an unusual result was only part of the process; understanding it was the real challenge. The Stardot community has been willing to dive into these obscure problems, profile code, and even produce patches with real enthusiasm. A hat tip to all who helped.

Perhaps a project scope at the start might have restrained the sprawl into an ever-broadening series of questions, puzzles, and documentation. But working through that spread has shown me the value of persistence, scepticism, looking sideways at results, and asking for help to understand what's really going on. And more: working on this re-creation, I've found that benchmarks aren't just numbers but, with care, help to reveal the stories of design choices, limitations, and innovations behind the old languages and systems we still enjoy using today.

There is still more to explore and open questions to answer. I hope others will find this as interesting as I have and contribute their own feedback, whether by challenging my findings, following up open questions, or bringing the test to new languages.

Yet even after all that reading, experimenting, analysing, and asking, I still can't work out how to save from FORTH!

6.1 Thanks

As I worked through the different languages, various issues and quirks surfaced, as outlined in the findings section. Although I could spot the problems, I might not know the cause. Asking for help revealed a knowledgeable and helpful community who not only dug into the code but also provided explanations and fixes. Thanks to everyone involved, but especially to:

- **dv8** for the remastered documentation, which provided the starting point for BCPL, COMAL, FORTH, and LISP, and for making the COMAL version of Tak work.
- **Coeus** for patching both BCPL's ROM and FIXCIN to work better with ADFS, and helping with the COMAL FP variable question.
- **tom_seddon** for investigating, identifying and **ebcdic** for fixing LISP's ESCape detection performance issue.
- **BeebMaster** for patiently running all of the tests on a real BBC B and "cheese wedge".
- **geraldholdsworth** for the indispensable Disc Image Manager.
- **BigEd** for uncovering the "origin story" of the Tak function.
- **flaxcottage** for hosting the only copy of micro-Prolog that I could find.
- **8bs** for being home to so many documents.

Appendix A Extract of Results

The code sizing and extract of the timing data from the main spreadsheet are provided here for easy reference.

A "-" in a table indicates that no code size is recorded for that configuration. This may be because the original code size was not printed (e.g., BASIC's SCV) or because the language is a new addition (e.g., COMAL or Beebug C).

A.1 Program Sizings

The table records the code sizes from both the original articles and the modern re-creations, which are fully reproduced below.

Sizes (in bytes)

Language	File	Original	Modern
Assembler	TAKAsm	215	231
BASIC	TAKscv	-	167
BASIC	TAK	82	77
BASIC	TAKfp	67	63
BASIC	TAKstr	-	92
BCPL	TAK	60	58
Beebug C	TAK	-	137
COMAL	TAK	-	164
COMAL	TAKfp	-	143
FORTH	F.TAK	58	58
ISO-Pascal	TAK	68	71
ISO-Pascal	TAKfp	80	74
LISP	TAK	204	204
micro-Prolog	TAK	366	174
micro-Prolog	TAKacl	-	203
S-Pascal	B.TAK	424	424

A.2 Program Timings

The table below presents a subset of the full timing data, showing results across key platforms, but the full data in the spreadsheet adds:

- Master 128 with modified OS 1.20
- MAME with MOS 3.20, MOS 3.50

Timings (in seconds)

Language	File	Original	BBC B	B-Em	MAME	MOS 3.20	MOS 3.50
Assembler	TAKAsm	2.64	2.64	2.65	2.52	1.92	1.92
BASIC	TAK	185	185	185	176	132	131
BASIC	TAKfp	248	247	247	236	167	164
BASIC	TAKscv	2.80	2.82	2.81	2.69	2.00	1.99
BASIC	TAKstr	294	294	295	282	207	206
BCPL	TAK	42	42	41	40	30	30
Beebug C	TAK	-	67	66	64	48	48
COMAL	TAK	-	865	868	829	631	631
COMAL	TAKfp	-	855	858	819	624	624
FORTH	F.TAK	21	20.8	20.6	19.9	15.2	15.2
ISO-Pascal	TAK	70	70	70	67	51	51
ISO-Pascal	TAKfp	79	92	92	88	67	67
LISP 5	TAK	326	327	345	321	332	356
LISP 4 (patched)	TAK	-	136	136	130	99	99
micro-Prolog	TAK	359	359	358	342	260	260
micro-Prolog	TAKacl	22	21.9	21.9	20.9	15.9	15.9
S-Pascal	B.TAK	19	18.4	18.3	17.5	13.4	13.4

Appendix B Sources and Discussions

Links to all the source materials, including patched software, and relevant Stardot discussion threads requesting help or clarification.

Correct at the time of writing, February 2025.

B.1 Original articles

- Acorn User 47, June 1986
  https://archive.org/details/AcornUser047-Jun86/page/n179/mode/2up
- Acorn User 52, November 1986
  https://archive.org/details/AcornUser052-Nov86/page/n197/mode/2up

B.2 Tak Function Background

- Stardot post mentioning background articles
  https://www.stardot.org.uk/forums/viewtopic.php?p=445547#p445547
- ACM Lisp Bulletin, Issue 3, December 1979. “An Interesting LISP Function” by J. McCarthy
  https://dl.acm.org/doi/10.1145/1411829.1411833
- Performance and Evaluation of Lisp Systems by Richard P. Gabriel, May 1985
  https://web.archive.org/web/20160417213353/https://rpgpoet.com/Files/Timrep.pdf

B.3 Discussions Arising

BCPL

- BCPL Standalone Generator on ADFS?
  https://www.stardot.org.uk/forums/viewtopic.php?t=18777
- Is ADFS slower than DFS?
  https://www.stardot.org.uk/forums/viewtopic.php?t=29632

COMAL

- Comal from Basic, Recursive Variable Scope?
  https://www.stardot.org.uk/forums/viewtopic.php?t=29836
- COMAL FP Faster than Integer?
  https://www.stardot.org.uk/forums/viewtopic.php?t=30211

LISP

- LISP: Slower with Co-Pros and MOS 3.50
  https://www.stardot.org.uk/forums/viewtopic.php?t=30075

Emulator performance

- B-Em
  https://www.stardot.org.uk/forums/viewtopic.php?p=440837#p440837

- MAME: How to ... ?
  https://www.stardot.org.uk/forums/viewtopic.php?p=435780&hilit=tak#p435780

General

- publishing long article to stardot?
  https://www.stardot.org.uk/forums/viewtopic.php?t=30192