ADBI*
A CS333 Extra Credit Project
Writeup

Chris Frost <chris@frostnet.net>

Description

ADBI is a BF interpreter which uses dynamic code generation to speed program execution, primarily by removing the need to reinterpret BF code when looping. (With some debugging of GNU *lightning*, ADBI could also save the compiled code its DCG produces so that it would not need to reinterpret the BF code when ADBI is ran again.)

The Process to Build Code Dynamically

ADBI begins by reading in the BF code into an array in memory, after which ADBI begins compiling the loaded BF code (compiling is done in a function oddly enough named `compile_bf()`). ADBI uses a set of macros called GNU *lightning*, copylefted by the Free Software Foundation, that take the assembly instructions ADBI generates and turns them into binary instructions for the host cpu and allows for them to be executed (*lightning* is retargetable, currently running on x86 and sparc, with plans for mips and alpha, and so the assembly language used is one unique to *lightning* that allows for quick mapping to the host cpu's instruction set).

`compile_bf()`, where all the code generation occurs, is described now. ADBI will be taking the loaded BF code and will generate a function to be later called that simulates the BF RTN operations [1]. There is an array of bytes in ADBI which serves as the memory for the BF program to store and manipulate its data. Based upon the BF instruction, ADBI generates *lightning* instructions to increment/decrement data, the pointer to data, to do i/o, and to implement the looping constructs. None of these are difficult to express in assembly, though looping takes some thought.

When ADBI encounters a `[`, it pushes the current generated-assembly code location onto a stack (this is a BF compile time operation, it doesn't occur during execution of the BF code), which is used to allow jumping back to the current location, and generates a branch instruction, using a forward branch reference which is patched upon encountering the matching `]`. This forward reference is also pushed onto the stack. ADBI then continues generating code like normal, which is the body of this loop, until it encounters the matching `]`, at which point the forward reference is saved to a local variable, a jump instruction is generated which does a backwards jump to the location on the stack (the beginning of the loop), and then a patch, using the saved forward reference, is made to allow skipping over the loop. The maximum size of the stack used to store these references is a static-sized array, but it is created such that for any loadable program this stack can not be overfilled.

After parsing the loaded BF code, ADBI generates a return instruction, and returns to `main()`, at which point the generated function is called and executed, performing the actions directed by the loaded BF code.

Performance Testing

While I didn't have much time to analyze the performance of ADBI and had a difficult time finding BF programs with long run times, I was able to test two quine programs (all other long-running programs required user input, which is difficult to automate with our reference interpreter). The benchmarking results are shown in Figure 1, showing ADBI to be more than an order of magnitude faster than our given reference interpreter for these tests.

Program	ADBI	Reference
quine.b	0.40s	5.71s
quine-bock.b	0.50s	5.70s

Figure 1: Execution speed comparison (Tests ran on a 900MHz AMD Thunderbird)

References

[1] Frost, Chris, *BF Abstract RTN*. Nov. 2001.

*ADBI is pronounced "Add-Buy" and stands for "**A**DBI **D**ynamic Code Generating **BF** **I**nterpreter."