SIGBOVIK'75 Technical Note: Conditional Move For Shell Script Acceleration

Dr. James McCann
IX@TCHOW-ARPA

Dr. Tom Murphy VII
TOM7@T7-ARPA

Abstract

One of the most effective programming interfaces for modern microprocessing computers is the command-line interpreter, or shell. Shell scripts provide a high-level abstraction of the operations of a microprocessor, making them an appealing alternative to hand-translated machine code or so-called ''macro assembly languages''. Unfortunately, shell programs are also significantly slower than their assembly counterparts. One potential source of this slowdown is branch misprediction. In this paper we show how to address this drawback by adding predicated execution to the shell.

1 Introduction

One common trend in computing as of late has been the migration of features from CISC instruction sets developed by DEC, Intel, and other captains of industry to the relatively underserved ''high level'' languages community, who generally focus on simpler and less modern operations. We continue this tradition by showing how to bring the advantages of predicated execution to shell scripting.

Predicated execution allows processors to avoid the root cause of branch prediction stalls: branches. Instead, instructions are provided which can check one or more predicate registers and effectively become no-ops if the registers are not set. These so-called predicated instructions are always executed, so no pipeline stalls need to be included while the processor decides which instruction needs to be fetched next.

Predicated execution for the shell provides a similar benefit -- expensive branch predictions can be avoided, resulting in tremendous speed-ups (up to 100x in our tests).

2 Implementation

The minimal instruction needed for conditional execution on a modern five-stage pipelined cpu is conditional move. This instruction moves a result from one register to another if a tertiary condition register is set.

In shell scripting, where files are the obvious equivalent of registers, the semantics are clear:

Usage:

cmv <file1> <file2>[1]

Rename file1 to file2, if the condition register is set.

But what is the condition register? We explored two choices of condition register -- the exit value of the previous command, which leaves us with a bit of a problem, since this value is not readily available to a process; and the processor's cf[2] register, which is also not readily available to shell scripts.

2-1 The 'csh' shell

The first method we propose to allow easy access to the return value of the previous command is to run a modified shell -- which we dub the 'csh' shell -- that stores the return value of the previous command into an environment variable when invoking a subprocess.

In our implementation (Figure A), the variable is called 'DOLLARSIGN_QUESTIONMARK_ALL_SPELLED_OUT' for obvious reasons.

2-2 The 'c' utility

While it would be straightforward to implement, exclusively, a conditional move

1. Note the use of AT&T assembly syntax, where the source operand comes before the destination operand.
2. ''condition flag''

```
diff --git a/src/exec.c b/src/exec.c
index 87354d4..7552f8d 100644
--- a/src/exec.c
+++ b/src/exec.c
@@ -112,6 +112,8 @@ shellexec(char **argv, const char *path, int idx)
         char **envp;
         int exerrno;

+        /* pass exit status of last process to executed program */
+        setvarint("DOLLARSIGN_QUESTIONMARK_ALL_SPELLED_OUT", exitstatus, VEXPORT);
         envp = environment();
         if (strchr(argv[0], '/') != NULL) {
                 tryexec(argv[0], argv, envp);
```

Figure A: This patch for https://git.kernel.org/pub/scm/utils/dash/dash.git/ creates a shell that stuffs the return value of the previous command into an appropriately-named environment variable.

utility; we instead embraced the high-level nature of shell scripting by creating a general purpose predication utility, 'c', which (when called as 'cNNN ...') will run 'NNN ...' when the proper predicate values are set. Since this utility's behavior is based on its name, one can create a new instance of it by simply creating an inode link. For example, to make a conditional version of /bin/sh:

```
ln -s /bin/c /bin/csh
```[3]

Our implementation of 'c' (see https://github.com/ixchow/c/blob/master/c.c) is written, naturally, in C, and is built to support both the shell-level approach discussed above and the kernel-level approach discussed below.

3 Evaluation

In order to evaluate the performance gains of conditional evaluation, we compared conditional and traditional versions of several simple shell scripts (see Appendix). We timed the scripts by first clearing the page cache, then running the traditional version of the script, then running the conditional version of the script.

The tested tasks were:

* 'echo' which makes two static checks and echos a string depending on the result;
* 'copy' which copies the smaller of two files to a destination;
* and 'compile' which compiles an output file depending on the timestamp of a source file.

Results are given in Table I. In all cases the predicated execution version of the task does better. Indeed, for the compile task, the overall execution time is reduced to 1894299ns -- that's 139426014ns faster than simply running the compiler!

4 Pushing performance

While a 2-100x cyclefold improvement is nothing to shake a luggable microcomputer's vacuum fluorescent display at, these results fall short of what we could hope for. One possible explanation for the lukemoist performance is that the condition itself is stored in a high-level way (see Figure B) using environment variables. Using high-level parts of the computer is a well-known cause of cycle overslows.

The fastest place to store the condition is in the CPU itself, using electrons. The CPU can only be accessed

3. Note the use of INTEL assembly syntax, where the source operand comes after the destination operand.

task	copy	echo	compile
ours	457516888ns	14104407ns	1894299ns
old	5753853493ns	29336387ns	182916245ns

Table I: Benchmark results. Our approach is dramatically faster in all cases.

through the Operating Kernel. As a proof of concept, the authors created a Kernel module[4] that directly accesses the CPU's FLAGS register. It presents the flags as files in the /proc filesystem where they can be accessed by any process:

```
$ ls -al /proc/flags
-rw-rw-rw-   1 root root 0  af
-rw-rw-rw-   1 root root 0  cf
-rw-rw-rw-   1 root root 0  df
-rw-rw-rw-   1 root root 0  if
-rw-rw-rw-   1 root root 0  iopl0
-rw-rw-rw-   1 root root 0  iopl1
-rw-rw-rw-   1 root root 0  nt
-rw-rw-rw-   1 root root 0  of
-rw-rw-rw-   1 root root 0  pf
-rw-rw-rw-   1 root root 0  sf
-rw-rw-rw-   1 root root 0  tf
-rw-rw-rw-   1 root root 0  zf
```

Each file contains -- at the moment that it is read -- either are ‘‘1’’ or ‘‘0’’ if the corresponding bit is set in the FLAGS register. Writing a ‘‘1’’ or ‘‘0’’ to a file will modify the corresponding bit. The ‘c’ utility described in Section 2-2 has experimental support for storing the condition result in the cf flag (formally ‘‘carry flag’’ but the mnemonic can also be used for ‘‘condition flag’’) via this kernel extension.

Alas, with great power comes great instability. There is some risk that the FLAGS register[5] is modified by other applications running in time-share with the ‘‘main’’ shell script. In this case, the FLAGS register may not correctly reflect the indicated status, and conditional operations may occur or not occur contrary to the shell program's coding. On the other hand, some uses of /proc/flags are very robust. For example, setting /proc/flags/tf, the trap flag, reliably terminates the current process with a fatal error.

We installed this kernel module on several shared workservers that we administer. Preliminary user reports include indications that the behavior is ‘‘very unstable’’ or ‘‘does not work at all.’’ Clearly, a wider-scale test deployment is needed.

5 Future Work

Given that predicated execution leads to a CISC-ridiculous improvement in the speed of shell scripts, it is natural to ask what other CISC-onesquential results can be obtained by bringing other micro-architectural features to high-level languages.

Branch delay slots -- instructions after a branch that are always executed -- can already be trivially supported in shell by writing the delayed commands as a background task in front of the branch in question, then foregrounding them afterward, as per:

```
delay-command &
if [ -x "something" ]
then
    fg
    #...
else
    fg
    #...
fi
```

Notice that this is actually much more flexible than current (MIPS) microprocessor implementations, since multiple commands may be queued in the delay slot

4. It can be downloaded via hypertex at sf.net/p/tom7misc/svn/HEAD/tree/trunk/csh/
5. As an additional technical matter, this approach does not work for multiprocessor systems, where there is one FLAGS register *per CPU*. Fortunately, such systems are a mere theoretical curiosity.

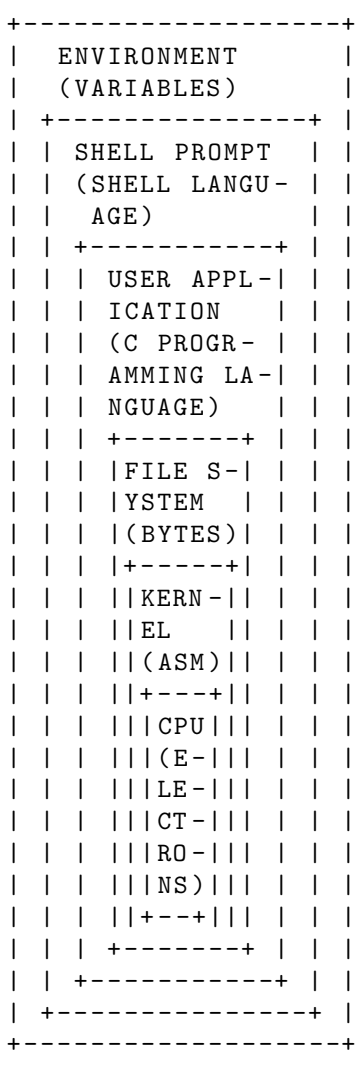

Figure B: The ‘‘levels’’ of a computer-system, from ‘‘high’’ (outer boxes) to ‘‘low’’ (inner boxes). As we descend to lower levels of the computer, the programming tools used (in parentheses) become more difficult, but more powerful, and more fast. Only master wizards are permitted at the lowest levels, such as ‘CPU’.

position and overlapped with the test execution.

A similar approach works to enable speculative execution, wherein code in a conditional is executed before the condition is checked:

```
true-command &
false-command &
if [ -x "something" ]
then
    kill %%
    fg %-
else
    kill %-
    fg %%
fi
```

Mind you, if either true-command or false-command have any side-effects before the test completes, this approach may lead to undesirable output; but a fast enough CPU will certainly turn this ‘‘race condition’’ into a ‘‘victory condition’’. As a compromise, on slower CPUs, each branch of the if statement could be run in a separate chrooted union-mount, with a snapshot of the result written back after the test has resolved.

Such a technique may be vulnerable to the Shpectre vulnerability, leaking information to other processes on the time-share via side-channels like the cache. Thus it is recommended to flush the cache before and after using this technique:

```
sync
echo 3 > /proc/sys/vm/drop_caches
swapoff -a
true-command &
false-command &
if [ -x "something" ]
then
    kill %%
    fg %-
else
    kill %-
    fg %%
fi
swapon -a
echo 3 > /proc/sys/vm/drop_caches
sync
```

6 Conclusions

We have demonstrated that shell scripts can benefit from conditional execution.

Q	E
E	D

Appendix: Test Code

This appendix contains source listings for the the shell programs used in the benchmarking process described above. The utilities ‘cecho’, ‘ccp’, and ‘ccc’ are all links to the ‘c’ program described in the main body of the paper. Notice how the predicated execution versions are also generally shorter than their tranditional counterparts.

```
# Echo; traditional
if [ "A" = "A" ]
then
    /bin/echo "Hello"
fi
if [ "A" = "B" ]
then
    /bin/echo "World"
fi
```

```
# Echo; predicated execution

[ "A" = "A" ]
cecho "Hello"
[ "A" = "B" ]
cecho "World"
```

```
# Copy; traditional
sizeA=‘stat -c %s fileA‘
sizeB=‘stat -c %s fileB‘
if [ $sizeA -le $sizeB ]
then
    echo "fileA is smaller"
    cp fileA fileS
else
    echo "fileB is smaller"
    cp fileB fileS
fi
```

```
# Copy; predicated execution

sizeA=‘stat -c %s fileA‘
sizeB=‘stat -c %s fileB‘
[ $sizeA -le $sizeB ]
cecho "fileA is smaller"
ccp fileA fileS
[ $sizeA -gt $sizeB ]
cecho "fileB is smaller"
ccp fileB fileS
```

```
# Compile; traditional
if [ "prog.cpp" -nt "prog" ]
then
    cecho "Compiling program..."
    cc prog.cpp -lstdc++ -o prog
fi
./prog
```

```
# Compile; predicated execution

[ "prog.cpp" -nt "prog" ]
cecho "Compiling program..."
ccc prog.cpp -lstdc++ -o prog
./prog
```