sdcc: Simple distributed compilation

Michael Chang

December 4, 2014

1 Introduction

Compiling large software systems can be very time consuming. Compilation speed is often limited more by the CPU than I/O or memory. Thus, although parallelism on a single machine can improve compile times substantially, there is an opportunity to further speed up compilation by distributing the task across multiple machines. Although software exists for this purpose, most notably distcc, this software is very complicated and does not take advantage of the full potential for optimization.

The goal of this project is to design a simple and lightweight application for distributed compilation. More specifically, our objectives are as follows:

- The system should be easy for clients to use. For example, distcc can be run by replacing the CC environment variable when running make. We will use the same client interface.
- The system should not depend on knowledge of the language, and should rely very little on knowledge of the compiler. This is unlike distcc, which implements its own analyzer of #include directives in order to handle header file dependencies.
- The system should work correctly without requiring any meta information (such as dependencies), but its performance may be improved significantly with this information.

The first two objectives clearly relate to simplicity of the system and would allow it to be widely usable. The final objective arises from the hypothesis that much of the communication latency in distributed compilation will come from requesting and transferring source and header files. As such, using dependency and other meta information to reduce the number of file transfers should lead to a substantial performance benefit.

In the rest of this paper, we describe the design and implementation of sdcc. In section 2, we discuss the system's components and how they interact. In section 3, we discuss a number of implementation details and challenges we faced, and some which we need to address further. In section 4, we discuss preliminary evaluations and summarize our progress and future direction.

2 Overview of sdcc

sdcc consists of three components: the client program (sdcc), the server daemon (sdccd), and the system call wrapper library (libsdcc).

The workflow of an distributed compilation using sdcc is as follows: First, the user runs a copy of sdccd on each machine which will take part in the compilation. These machines should have a working compiler and all relevant system libraries installed, but need not have any of the files to be compiled. Then, the client invokes "make" from the master machine, setting the CC variable to sdcc. For each file to be compiled, a newly spawned sdcc process opens a connection to one of the sdccd processes, passing information about what file(s) to compile. sdccd then spawns the compiler, using libsdcc to monitor its system calls an automatically detect dependencies. Required files are sent by sdcc to sdccd. Finally, the compiler output is returned by sdcc.

2.1 libsdcc

libsdcc is a simple library that provides wrapper functions around a handful of system calls. It is dynamically loaded into the compiler using LD_PRELOAD. Currently, it overrides access() and open().

When the compiler attempts to access() or open() a non-existent file relative to the current directory (i.e. not a system header or library) for reading, libsdcc intercepts the system call and checks with the client machine (through sdccd). If the file indeed does not exist, execution continues normally. Otherwise, execution stalls until the file can be transferred from the client machine, and then the system call is retried.

2.2 sdcc and sdccd

The client and server programs communicate using a TCP connection. The server listens on a designated port, and the cleint keeps a list of available remote servers. When the connection is first established, the client sends the full compilation command to the server. The server spawns the compiler process, setting LD_PRELOAD to libsdcc. The server passes libsdcc a pair of file descriptors by setting an environment variable (the file descriptors are inheited by the compiler process), which are used to communicate between sdccd and libsdcc.

When the library encounters a file that does not exist, it writes the path to sdccd via the pipe. sdccd sends this path to sdcc over TCP, requesting the file if it exists. If it does not, this response is propagated to the library. If the file does in fact exist on the client machine (for example, because it was the output of a previous compilation), sdcc transfers this file to sdccd, which stores it in the correct filesystem location and informs the library.

When the compiler process has terminated, sdccd informs sdcc by sending a blank line. sdcc then requests the output file, which it extracted from the compiler command (e.g. the -o option of gcc). If this file was generated, sdccd transfers it back; otherwise (for instance, if the compilation had an error), sdcc returns failure.

3 Implementation Details

There are a number of implementation details and challenges we encountered on this project. Many of these details manifested themselves when trying to compile the Linux kernel using sdcc. Some have been partially or mostly addressed, while others required additional work to resolve.

3.1 Identifying Outputs

In order to accurately identify the output of the compiler, sdcc needs to understand enough compiler flags to identify all flags that could define an output file. In addition to the standard -o flag for compiled output, there are also a number of flags to generate Makefile dependencies and output those to a file.

In order to avoid parsing the command line arguments of hte compiler, which is required to be fully compiler agnostic, we could instead attempt to identify attempts to open() files for writing. But as discussed in the next section, this is also somewhat challenging and was not reliable with GCC.

The current implementation of sdcc scans the command line for the -o option. We expect to improve our handling of this issue in the next week.

3.2 Wrapping System Calls

If all attempts to open a file eventually called libc's open() (or open64()) function, finding them all would be easy. However, this is not the case. In particular, glibc's fopen() does not call open() directly. Because the GNU assembler and linker use fopen, libsdcc needs to separely wrap this function.

There appear to be some other files libsdcc is not identifying, such as the final executable output by the linker. We need to explore other ways that files are getting opened and add additional handlers to libsdcc. (This is not entirely straightforward because of glibc's complexity.)

Another system call that was an issue was access(). It appears that GCC is inconsistent about whether it first tries to access() a file before open()ing it, or whether it opens it directly. For example, GCC calls access() on each of its input files, and immediately errors out if they do not exist. But in its search for header files, it directly open()s each path until one succeeds. This means that libsdcc needs to also wrap access() and check with the client for these calls, which appears to add a significant number of checks.

We considreed alternate strategies for system call interposition, but none seemed as clean as LD_PRELOAD. ptrace, for example, incurs a massive performance penalty. (On the other hand, we did not measure a performance difference caused by the wrappers themselves.)

3.3 Other and Unusual Commands

Not all invocations of GCC are as straightforward as "gcc -c foo.c -o foo.o." There are a number of cases in Linux where GCC is invoked with /dev/null as input, or with - (stdout) as output. We couldn't tell what exactly was happening with these calls. Currently, sdcc special cases /dev/null and - by invoking gcc directly on these commands. We will further investigate these commands in the next week.

Another issue is that calls to the compiler aren't necessarily the only important parts of a build process. For example, it is very common for "make" to mkdir many directories. By overriding CC, sdcc is unable to know about these other commands. Currently, sdccd handles missing directories by creating them as needed, but this only works for input files, not output files (which can fail to be created without the proper directory hierarchy). One solution to this is to inform sdccd of the output file upfront, so it can create those directories too. It is not clear if we need to handle additional commands besides mkdir.

3.4 Security Implications

A major security concern of the system is that sdccd needs to accept an arbitrary invocation of the compiler and execute it. (In fact, the current implementation is worse in that it accepts arbitrary commands, but this can be addressed by restricting the program(s) which can be exec()ed by sdccd.) This could be an issue, if, for example, the client executes a command like "gcc -E foo -o /etc/passwd."

Although we might assume that the user controls the server machines, this seems like a very risky assumption, especially if the servers do not properly restrict client connections. It is not clear how much we can really protect the user from malicious commands.

4 Evaluation and Next Steps

Our goal is for sdcc to successfully compile the Linux kernel, which, on the author's machine with a single make thread, takes about 7 minutes. Due to the limitations described above, the current implementation is unable to proceed past the first few GCC invocations (make dependencies appear to be the current issue).

We tested sdcc on a smaller project to obtain preliminary results. The project contains 15 source files totaling about 2,000 lines. It takes 0.6s to compile using one thread on the author's machine, and 0.2s to compile using 8 threads (make -j8).

With only one sdccd server and one make thread, compilation takes 2.4s. The 4x slowdown seems largely due to the number of access/open calls that actually refer to nonexistent files. Using four sdccd servers on the same machine, and make -j8, compilation time reduced to 0.6s, on par with single-threaded performance. This number does not change substantially when using four remote machines on Stanford campus (which can be explained by the local machine having 8 virtual cores and network latency being very low).

These results, though disappointing at first, suggest that sdcc has the potential to do well if we are able to reduce the number of RTTs between the client and server. This, and the discussion above, suggest the following next steps:

- Fix issues with directories and output files (3.1 and 3.3). We hope this is the primary roadblock for compiling Linux.
- Explore whether other library functions are being used to open files (3.2). This may help with the previous issue.
- Use the directory hierarchy after a build to cache information about known-nonexistent files. This should substantially improve the performance of the system.

- Explore other ways to reduce the amount of client-server communication.

We hope to address each of these points more substantially by next week.