Bachelor Informatica

Universiteit van Amsterdam

Concurrent Linking with the GNU Gold Linker

Sander Mathijs van Veen

August 14, 2013

Supervisor(s): Dr. Clemens Grelck

Signed:

Abstract

Traditional build systems for large-scale software projects that produce a binary out of many source files can benefit from the ubiquity of multi-core systems in the compilation phase but not in the linking phase, which thus acts as a barrier synchronisation. This barrier makes it impossible to scale the jobs of the build system linearly with the available CPU cores. The linking phase is often not negligible because linking previously created object files reduces the time the compilation will take if only a few files are changed. Concurrent linking can remove this barrier by interleaving parts of the linker work with the compilation jobs. The biggest reductions in build time are achieved by changing the build system to divide the jobs by the available CPU cores. Concurrent linking can reduce the build time even further, because theoretically 50% of the time spent during link-time can be done concurrently to the compilation jobs. On top of that, it is possible to combine incremental and parallel linking with concurrent linking to build incremental builds faster on CPUs with many cores.

Contents

CHAPTER 1

Introduction

In the early days of computing, the computer executed programs in a sequential, non-preemptive order. The computer loaded a program into memory, followed by executing the program. After execution, the next program was loaded into memory and the cycle continued. These programs were stored in a single file. Loading the program was as easy as copying the binary code into memory – given that the code fits into memory – and setting the program counter to the memory offset of entry point's first code instruction.

One of the first steps to improve the reusability of code was separating functionality by moving code into different files. Separating the functionality allowed the developers to reuse parts of the code. Reusing code became important because it saves development and maintenance time in the long term. Instead of maintaining every copy of the code, the developer has to maintain one file containing the code. The other files can use the code stored in that file by defining that the used code is stored elsewhere.

Having the ability to reuse parts of code saves development time, but it will also increase the time before the computer can run the binary program. Instead of just copying a single file to memory, the computer has to load the dependencies of the file as well. It is also necessary to check if the reused code does actually exist in the specified file. If the code does not exist, the computer must abort loading and running the program. If the computer does not abort the program, it results in undefined behaviour. Loading code dependencies into memory and checking if the dependent code exists became more sophisticated over time, and are these days known as *relocation* and *symbol resolution*.

Once the multi-core CPU became mainstream, the time to build and start a program was not reduced by using multiple cores. The compilation, linking and loading phases were simply not designed for multi-core CPUs, since the phases were only run in sequential order at that time. Running the compilation of files in parallel was the easiest step to improve the build time. Because a compiler can compile each file individually, the build system can run a compilation job on each available core.

The raise of dynamic shared libraries caused more work for the linker to perform during link-time. The use of dynamic shared libraries requires time consuming relocations. Those have to be performed every time before the program runs, or once at link-time. Performing those relocations during link-time reduced the delay to run a program that uses a dynamic shared library, but increased the work for the linker.

Before dynamic shared libraries, the linker did not consume that much time during the build. Since the link-time increased, the edit-compile-run cycle became much longer. It is possible to improve compilation speed by reducing the optimization level, but this is not possible for linking. Linking scales linearly with the symbols in the binary regardless of the optimization level. The linking phase causes a delay for building large binaries because binaries grow larger while the linker has to perform its work sequentially.

There are two main concepts to let the linker benefit from multi-core CPUs: performing the work that the linker has to do in parallel, and running the linker concurrently with the compilation jobs. The former will speed up the linking phase by dividing the object files between the available CPU cores. The later will speed up the overall build by running parts of the linking phase when an object file is ready.

Parallel linking will reduce the time spent during the linking phase. It is important to note that parallel linking does not remove the synchronization barrier introduced by the linking phase at all. A parallel linker can only start when all object files are ready to be processed by the linker.

Concurrent linking on the other hand can be seen as preemptive scheduling of the linking job. The linking job is performed in chunks and interleaved with the compilation jobs. When an object file is ready, the linker is notified and performs (a portion of) its work. The synchronization barrier is still there, but the blocking part of the linking phase is reduced. Instead of processing all object files, the concurrent linker is basically done after combining the earlier processed output and writing the output file.

This thesis investigates how, why and when the link-time is reduced by using *concurrent linking* with the GNU Gold linker. A concurrent linker cannot perform well without a multi-core-aware build system. Therefore, (non-)recursive build system are also investigated to illustrate the importance of a multi-core-aware build system.

CHAPTER 2

Linkers

2.1 Build phases used for compiling source files

The basic phases of the build process – preprocessing, compiling, assembling and linking – are amongst the first things to be covered in an introduction to compiled programming languages. Typically, the build process for a single source file is straightforward, since there is a one-to-one relationship between phases: each phase of the build process takes a single input file and produces a single output file.

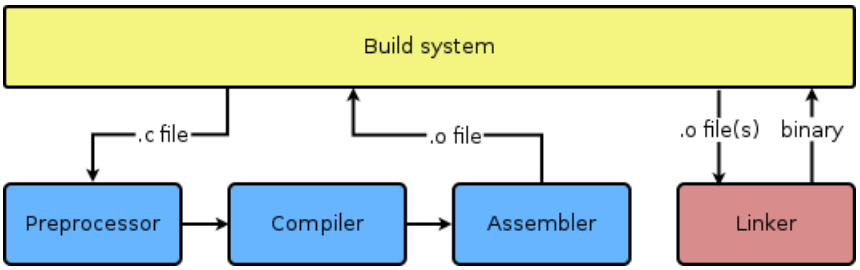

Figure 2.1: Overview of the interaction between the build system and a C compiler.

When a project consists of multiple source files that will be used to produce a compiled binary, there is an n-to-one relationship between the compiling and linking phases. It is possible to use one object file in multiple binaries, but that does not result in a n-to-m relationship, since the linker only outputs a single file.

Intermediate object files produced by the compilation phase are commonly saved to disk during a build process. Instead of saving to disk, the compiler can send intermediate data through pipes (e.g. C compilers will do this when `-pipe` is added to the `CFLAGS`). This can result in faster compilation, and more memory usage. Rebuilding a binary after a source file is modified takes less time if previously-saved intermediate files are reused. More specifically, reusing these intermediate files reduces the time spent on the preprocessing and compilation phases.

2.2 Build system and linking phase

GNU Make[1], one of the Linux ecosystem's most popular build systems, tracks the status of a project's intermediate files and final targets, i.e., it tracks whether the files are up-to-date or require a rebuild. Instead of manually invoking the commands for each build phase, GNU Make

[1]GNU Make: `http://www.gnu.org/software/make/`

invokes the requested recipes based on the status of the recipes' dependencies. GNU Make also offers a job server which makes it possible to run the recipes in parallel, where dependencies permit this (if job `a` depends on job `b`, job `a` cannot run in parallel with job `b`).

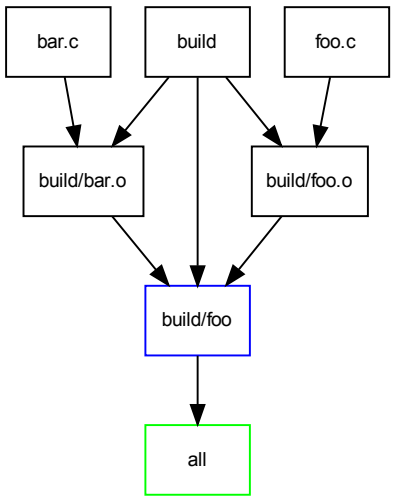

Figure 2.2: An illustrated example of a directed acyclic graph used for calculating dependencies in GNU Make. In order to build the target `all`, it is necessary to satisfy its dependencies: the target `build/foo`.

In figure 2.2, the target `build/foo` has three dependencies:

- the object file `build/bar.o`.
- the object file `build/foo.o`.
- the directory `build/`.

Given the fact that the directory `build/` is also a dependency of the object files `build/bar.o` and `build/foo.o`, GNU Make is forced to resolve the directory `build/` before the object files.

The targets `build/bar.o` and `build/foo.o` can be built in parallel, once the dependent directory `build/` is created. When both object files are created, the linker can be invoked to produce the target `build/foo`.

Important to note is that the linking phase cannot succeed if any of its dependencies are not built. In such cases, the set of symbols available to the linker is incomplete, and the linker fails whilst trying to resolve all symbols. Therefore the linking phase cannot start until all dependencies are built (given that the linker has no information about which input files are ready for processing and which are not).

2.3 What does a program linker do?

The program linker, also known as the *linker* and `ld`, takes one ore more object files generated by a compiler, and creates one composite executable or shared library. The program linker resolves symbolic references (also known as *symbols*), arranges the objects in a program's address space (when linking statically) and merges code and data sections. [4]

Depending on whether an executable or shared library is requested as output type, a missing symbol can abort the linking process. A shared library can have unknown symbols – otherwise it would be difficult to link separated dependent libraries – while an executable cannot.

It is also possible for a shared library to declare symbols as *weak*. This means that the symbol can be redefined in another shared library or the executable. A symbol is by default *strong*,

redefining the symbol is considered a link-time error and aborts the linker. A weak symbol (thus explicitly marked as *weak*) can be redefined multiple times, as long as the redefining symbol is either weak or the last definition of the symbol. The last definition is used as the final result in the shared library or executable being built. [1]

2.4 What does a dynamic linker do?

The dynamic linker, also known as a *loader* and `ld.so`, will find and load the shared libraries needed by a program, prepare the program to run, and then run it. It decides where the code and data sections of a shared library or executable should be placed in memory.

The memory addresses of the code and data sections are calculated at runtime, due to attack migration techniques like Address Space Layout Randomization (ASLR). ASLR is a security method which randomly arranges the position of code and data sections of an executable. This makes it more difficult for an attacker to guess memory addresses.

However, it is possible to *prelink* a shared library. This will apply randomization at prelink time, which will calculate the random addresses of the code and data sections once. Although prelinking will speed up program startup time, it also places the libraries at fixed addresses, which could be seen as defeating ASLR.[2]

[2]There is a discussion about if prelinking is really worth it to apply: `http://lwn.net/Articles/341244/`

CHAPTER 3

Relocations

During compilation, the program linker does not know which virtual address the shared library will use. It is also possible that the shared library will run at multiple virtual addresses, since the run-time linker chooses the virtual memory address of the libraries.

Using position dependent code in a shared library causes a lot of relocations for the dynamic linker. Therefore, the shared library code must be position independent at run time, in order to start a program with shared libraries quickly. Also note that programs can start quicker because there is a good chance that commonly used shared libraries are already loaded in memory. [3]

The memory pages containing position dependent code cannot be shared between processes, otherwise processes would interfere with each others' state. Think of memory corruption by overwriting a data structure, or reading from the wrong memory address which leads to undefined values.

3.1 The ELF binary format

There are many binary formats used for storing machine instructions and related metadata. In the Unix world, the Executable and Linkable Format (ELF) is the most commonly used format for executables, object code, shared libraries and core dumps.

In an ELF dynamically linked file, code and data segments are separated. The separation of segments is necessary to perform memory mapping efficiently when loading the program. Code and data segments also have different access privileges by default:

- code segments are executable, but not writable.
- data segments are writable, but not executable.

This separation of access privileges is also known as W^X (or DEP on Windows). W^X implies that something is writable or executable, but not both. This defeats some buffer overflow attacks, including the most common stack-based attack. For example, W^X ensures that the stack is not executable, thus malicious code injected into stack will not execute but cause the program to terminate.

Besides the code and data segments, ELF files can also contain metadata sections, e.g.:

- `.rela.dyn` and `.rela.plt` for run-time relocations.
- `.symtab` and `.symdyn` for link-time and run-time symbol tables respectively.

Notice that two symbol tables are necessary. The symbol table used at run-time can be reduced to just the dynamically loaded symbols while the statically known symbols are only useful to the linker at link-time for symbol resolution.

3.2 Relocations at runtime

ELF binaries contain a relocation list – called `.rela.dyn` for variables, and `.rela.plt` for functions – which describes the fixes that are necessary to run the executable code.[2] A relocation consists of:

- where in memory the fix is to be made (offset).
- the algorithm to calculate the fix (type).
- the symbol to fix (string and object length).

The dynamic linker will run the specified algorithm on the offset and symbol to calculate the actual address. Once calculated, the address is stored into memory. Note that the algorithm to calculate the fix is dependent on the instruction set in use. However, the overall concept of calculating the real memory address of the to-be-relocated symbol is similar.

For example, ARM does not have a global offset table, but uses address pools instead. This is caused by ARM's immediate instruction operands, which consist of an 8-bit constant value. Using only 8 bits, there cannot be more than 256 memory addresses in a single global offset table. Due to the use of multiple address pools, calculating the real memory address involves more work. [8]

3.3 Indirect memory accesses

The disadvantage of position independent code is slightly decreased performance for non- static function calls and for memory accesses of global or static variables. Position independent code has to look for the proper function call in the Procedure Linkage Table (PLT), while the real addresses of global and static variables are stored in the Global Offset Table (GOT). The decrease in performance is caused by these indirect memory accesses.

For the i386 instruction set, every call to a global function requires one extra instruction after the first time the function is called. Every reference to a global or static function uses four extra instructions the first time that execution reaches it. The first time a global function is called, four extra instructions are necessary as well.[6]

Note that it is not feasible to fix these indirect memory accesses for global function calls, due to the fact that the position independent code is shared with processes. Position dependent code does not have these indirect memory accesses, but also implies that all code has to be duplicated in memory. This is a noticeable waste of memory for commonly shared libraries like libc, glibc, and user interface toolkits.

CHAPTER 4

Concurrent linking

4.1 Introduction to concurrent linking

Generally, linking is the last phase in the build process of C/C++ programs. Most build systems, for example GNU Make, invoke programs defined in the build system, when a given set of prerequisites is completed. That way, linking requires that all linker input files (object files, shared objects, archives, etc.) are built, before linking can start.

While source compilation can mostly be parallelized well, the linking jobs cause CPU under-utilization, because it has to wait for all its input files to finish. The result is that the linking phase adds a significant amount of time to the build process, when a program is rebuilt (the *edit-compile-run* cycle).

Concurrent linking means that the linking phase happens simultaneously with the compilation phase. This subsection explains why this technique is called *concurrent linking*, instead of *pipelined linking* or *interactive linking*.

Pipelined linking Pipelined linking implies that the linking phase is part of a larger pipeline. This is not false per se, but it is somewhat strange because the linker can merge multiple object files into a single output file. Conceptually, this would mean that the linker is more *sink* than *pipe*.

It's also possible that the linker is run completely in isolation of the other remaining compilation jobs. In that case too there is no pipeline, while concurrent linking still fits the description of what's happening.

Interactive linking To some degree, one can argue that concurrent linking is an interactive process: once an object file is ready, the linker is notified and the linker starts processing the marked object file. The linker interacts with the build system. This is a sound explanation, but interactive linking could also mean that the end user has the ability to interact with the linker (using scripting, text-based questioning, etc.). The word *interactive*, then, is too ambiguous in this context, so sticking with concurrent linking would be better.

Parallel linking Concurrent linking is different from parallel linking because the work unit is the job and the object file. Parallel linking speeds up the linking process by using multiple threads (by dividing the object files across multiple cores), while concurrent linking will speed up the overall build process by increasing core utilization and thus improving overall parallelism. See also *Related work* for more details about parallel linking.

4.2 Reasons to choose the GNU Gold linker

GNU Gold is a relatively new ELF linker and written from scratch in C++ by Ian Lance Taylor, Google (hence the name *go-ld*). Given the fact that GNU Gold is written from scratch, and designed to speed up the linking process with multiple threads, it is a great candidate to implement concurrent linking.

Once concurrent linking is implemented in GNU Gold, it will speed up the overall (re)build process compared to GNU ld, since it also features incremental linking and parallel linking. The older GNU ld does not support these features (and it probably will never support them). The disadvantage of Gold is that it only targets ELF, while GNU ld support various executable and linking formats.

It should also be noted that GNU ld has been build for COFF and a.out binary formats. When ELF finally became a standard, ELF support was added to GNU ld. There was no clean way of adding ELF support – besides rewriting the linker from scratch – which made GNU ld inefficient for linking ELF binaries by design.

One of the drawbacks of the ELF implementation in GNU ld was the amount of scanning the symbol table. Instead of three scan operations necessary in GNU Gold, it is necessary to perform thirteen scan operations in GNU ld.[7] GNU ld is also build on top of libbfd, which makes symbol table entries 80 bytes larger per table entry. This number adds up quickly for the hundreds of thousands of symbols found in libraries of large code bases, like Firefox and LibreOffice.

4.3 Provide dependency information to the linker

Besides the performance impact, there is also a more practical problem to solve: how should dependency information be shared with the linker process?

There are three main ways to solve the information sharing problem:

- Spawn linker processes and use `inotifyd` in each linker process.
- Two linking states and additional command line options.
- Modify the build system environment (GNU Make).

Each way has its own advantages and disadvantages, and the comparison between these ways are mentioned below in more detail. To summarise the comparison, modifying the build system environment (e.g. GNU Make) is chosen, because that's the only way to avoid large modifications to the Makefiles of existing software.

Use `inotifyd` for file tracking

In the Linux world, `inotifyd` is a daemon for the inotify API. The inotify API provides a mechanism for monitoring file system events. Inotify can be used to monitor individual files, or to monitor directories.[1] File system events are creation, modification, and deletion of a file or directory.

Using `inotifyd`, the linker process can be informed about the creation or modification of an object file. The disadvantage is that the linker process should be started in advance, otherwise it is unable to detect the file system event. This implies that all linker processes must be spawned when the build system starts. For large code bases, starting all linker processes is wasting resources.

Using the inotify API requires two linker processes: one background process used to set up the inotify watches, and one foreground process to let the build system know the inotify watches are

[1] `http://linux.die.net/man/7/inotify`

all set. The foreground process will exit with an exit code indicating success or failure, while the background process wakes up for each created or modified object file it watches.

This has also some implementation disadvantages, for example, there is no guarantee that the inotify API is available on the platform (e.g. it's not enabled in the kernel by default on all Linux distributions, it's Linux-specific).

Two linking stages

Similar to incremental linking, two-stage linking is a way of letting the linker know that a specific object file is ready for processing. This is done by instrumenting the linker in the Makefile with additional command line flags. The linker process exits after the marked object file has been processed.

In order to maintain the file input order given on the command line, the linker can skip processing an object file and make a note of the skipped object file in the resulting output file. If there are any skipped object files, the linker will first process that file and after that, it continues with processing the object file marked as ready. Once all dependencies are resolved, the linker is invoked with a command line flag which indicates that the linker should complete the linking phase.

The advantage of this approach is that there are no modification required for the build system itself. This means that old versions of Make can use concurrent linking, however the Makefile should be altered in order to achieve this. Basically, this is moving the work to all end users: instead of fixing the problem of letting the linker know which files are ready, every Makefile should be modified.

Connect build system with linker

The third idea is based on sharing information directly: send the filename of an object file to the linker, once an object is ready to be processed by the linker. When GNU Make has constructed its directed acyclic graph of targets and prerequisites, it knows all recipes to invoke with the status of targets and dependencies.

A Unix pipe could be used for sharing the information: GNU Make sends, Gold receives. The reason to choose a Unix pipe is simplicity. Similar to the design choice of using the Unix pipe to connect GNU Make processes to the GNU Make's jobserver, a simple Unix pipe is cross-platform, maintainable and easy to debug.[5]

For concurrent linking, if Make sees that linking using Gold is necessary for a target and the `--concurrent-linking` flag is set for Gold, Make starts Gold when the first object file is ready. Make will create a pipe and sends each filename to Gold through the pipe.

Reading from the pipe (Gold's side) will block[2] until:

1. there is no more data to read from the pipe. Filenames are separated using a `\0` char (since `\n` is considered valid as a character for filenames).
2. all objects are sent through the pipe (e.g. `close()` from Make).

In the first case, Gold can produce parts of the final output file with the input files sent thus far. In the second case, Gold can safely finish the output file (since all files are received through the pipe).

In case of a failed dependency, Make knows what linking processes are running, and Make can terminate those. Once the linking process is killed, Make should delete the linker's output file. The reason is that the killed process could have produced an incomplete output file, and the

[2]The Gold linker process will be put into the `TASK_INTERRUPTIBLE` state. This is a desired state, because the process does not consume any CPU time until new data is arrived, or until the process got a signal.

next invocation of Make is not aware of the incompleteness. The next invocation of Make only sees a more recent output file (compared to the input files), and assumes that the output file is produced successfully.

CHAPTER 5

Experimental evaluation

5.1 Increasing efficiency

Due to the dependency constraint explained in the introduction, the linking phase is a blocking phase. In multi-core CPU systems, blocking impacts overall throughput: only one CPU can do the linking job at a time, leaving the other cores under-utilised. The result is that building the source tree with more idle cores available does not reduce the build time.

In order to measure CPU core under-utilisation, a custom-made build system analysis tool (hereafter called BSA) is used. The tool is based on `strace` and produces a detailed report about the spawned processes during the build. The integrated report viewer displays a waterfall graph that will visualize the spawned processes.

In figure 5.1, a portion of a BSA waterfall graph is shown. The waterfall graph is a build system visualization of the Mozilla's JavaScript engine. The blue bars are shell scripts and used for bootstrapping the build system (e.g. a cross-platform installation tool). In the original build system, bootstrapping is a blocking phase because the rest of the build system depends on the installation tool. The green bars are compilation jobs and can only be spawned when the blocking bootstrapping phase is completed.

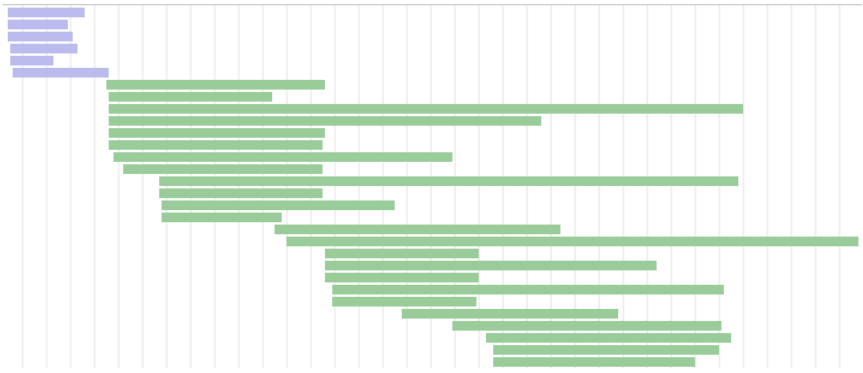

Figure 5.1: Example of BSA's waterfall graph. Shell scripts (blue bars, in the top left) cause CPU under-utilisation. Since the build system is running with twelve available job slots, the compiler jobs (green bars) cause almost an optimal CPU utilization: once a compiler job is done, the next job is started immediately.

BSA can easily visualize where CPU core under-utilisation occurs during the build. When a vertical line is drawn in the waterfall graph, the intersection of the line with the bars indicate the amount of job slots in use at that time. The tool can be extended to calculate the core

under-utilisation automatically and mark the parts where under-utilisation occurs. Calculation of under-utilisation can be done by counting the running jobs at that time and dividing that by the number of available job slots.

Deferring the linking job can render the disk cache utilisation less than optimal, since other data is read from disk. Object files on disk are no longer in the disk cache, making it inefficient to re-read them. If the object files were processed while their content was in memory, the overhead from disk I/O would be spared.

5.2 Relocations and concurrent linking

In order to implement concurrent linking in Gold, it is important to know which parts in Gold may benefit from concurrent linking. Therefore, a high level overview of the linker tasks and an overview of the time spent performing those tasks is necessary. It's also worth mentioning that linking is primarily an I/O-bound process. This means that the linker will perform different for files cached in memory and files read from disk.

The fact that there is a dependency between tasks that can be performed only with all object files is illustrated in figure 5.2. The state `middle` cannot be reached until the `Middle_runner` is done. In order to be able to perform the `Read_relocs` and `Scan_relocs` tasks concurrently with the compilation jobs, it is necessary to modify the `Middle_runner` to enqueue the `Read_relocs` and `Scan_relocs` tasks before the *middle* state is reached.

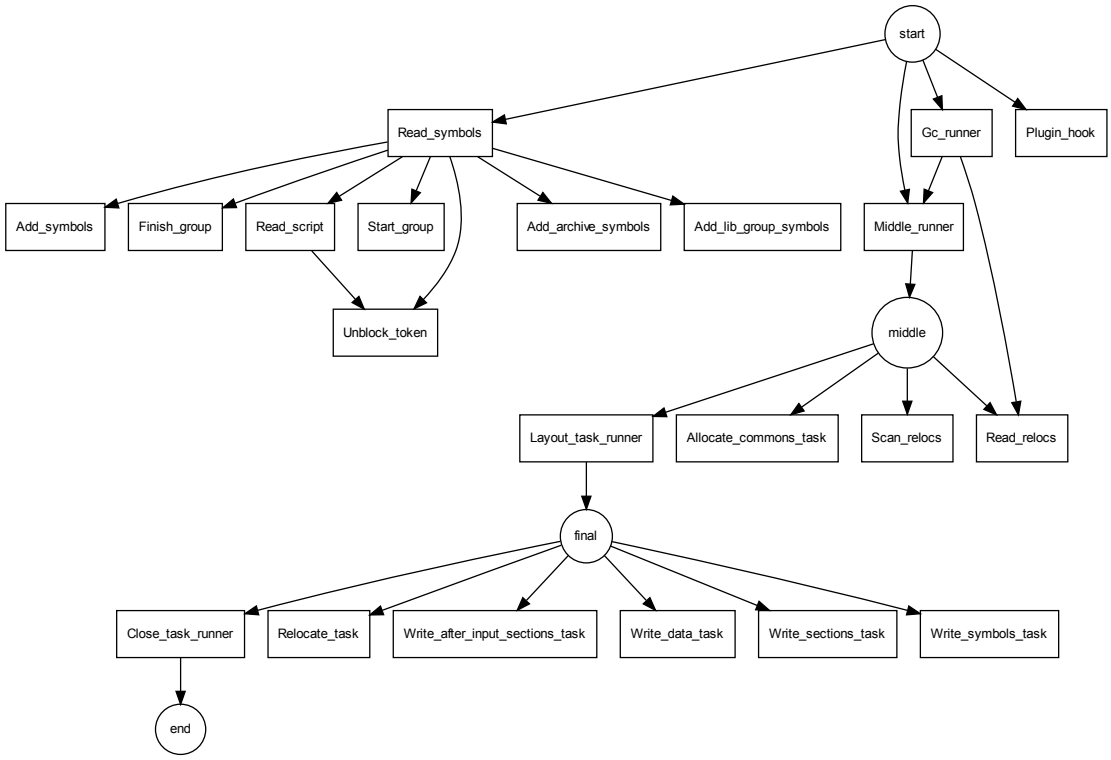

Figure 5.2: Graph displaying the relationships between the linker tasks and states. The circles indicate a linker state and the boxes a linker task. The arrows indicate that the task or state can spawn the task it is pointing to, or enter a new state.

In figure 5.3, the most time consuming Gold linking tasks are displayed. Mozilla Firefox's libxul library is used for the benchmark because libxul requires linking a lot of object files and takes almost a minute to link on a fast CPU. When libxul's object files are cached, 80% of the linking time is spent in `Task_function` (36%) and `Relocate_task` (44%). The `Task_function` task

maps the input sections to output sections and lays those out in memory. This is done after all input files are read. And `Relocate_task` – as the name suggests – performs the relocations.

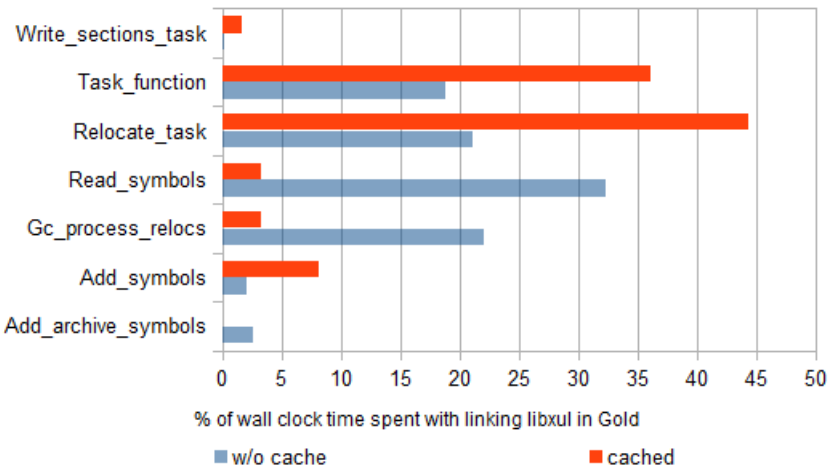

Figure 5.3: Overview of the main tasks performed during linking with Gold. Cached means that the object files used for linking are in the file system cache. Gold takes 55 seconds to produce libxul without the file system cache, and 6.1 seconds with.

The `Task_function` task cannot benefit from concurrent linking (because it needs every input file to be performed), while the `Relocate_task` can to a certain point. Since relocation is an incremental process, some of the object files will depend on another. Therefore, processing some of the object files is a blocking operation. This will reduce the efficiency, because the linker has to stop and wait for the dependent object files to be ready. The less time consuming tasks `Add_symbols` and `Read_symbols` can be performed during concurrent linking as well.

Optimally, this means that the linking time of libxul can be reduced by 55% (= 44% for `Relocate_task` + 8% for `Add_symbols` + 3% for `Read_symbols`). This reduces the linking time from 6.1 seconds to 2.7 seconds in the optimal case, when the object files are cached.

When the object files are not cached, Gold spent most of its time in `Read_symbols` (32%), `Gc_process_relocs` (22%), `Relocate_task` (21%) and `Task_function` (18%). Optimally, this means that the linking time of libxul can be reduced by 55% (= 32% for `Read_symbols` + 21% for `Relocate_task` + 2% for `Add_symbols`). This reduces the linking time from 55 seconds to 25 seconds in the optimal case, when the object files are not cached.

In the current design of the GNU Gold linker the task `Relocate_task` can only start when the *final* state is reached. Relocation is a sequential, incremental process, which makes it possible to move the task `Relocate_task` to the `start` state. In order to do so, Gold has to make sure that the relocation task can only start for object files where all of its dependent object files have been processed. Otherwise, Gold has to add the object file to a to-do list of object files, which will be processed once their dependencies are processed.

5.3 Recursive Make considered harmful

In GNU Make, each Make instance has its own list of targets to build. There is no communication between two Make instances about the targets to be made. If there are less jobs to be done than concurrent job slots available, Make will wait until the last recipe is executed. This results in waiting for the last recipe to finish in each Make instance, which will cause underutilization of the other cores during the last recipes. This prevents multi-core systems from speeding up builds linearly.

Experiments with Mozilla's Firefox

The source tree of Mozilla Firefox is large. As of November 2012, there are approximately 50,000 files, of which 10,000 files are C/C++ source and header files. The tree consists of 48 top level directories, approx. 3000 directories in total, and 1148 Makefile.in files spread across those directories. Last but not least, it uses a recursive build system. The build system is complex due to the size of the project, which makes it interesting to look into it in more detail.

The first noticeable thing about the Mozilla build system is slow rebuilds. This has a couple of reasons:

Multiple GNU Make instances Mozilla's build system spawns new Make processes for each browser component. The new Make process will enter the sub directory, execute the tasks it was asked to do, and leave the sub directory (this is done using the `-C dirname` flag). This behaviour is known as *Recursive Make*.

Spawning the sub process will cause the parent process to halt, since the recipe of the sub process does not finish. A single Make instance implies that it knows all targets to build, while multiple instances only know their own targets. In order to improve parallelism, a single Make process should be spawned during the entire build, because it knows which and when targets are done across components.

Dependencies are not properly stated Since the build system uses multiple Make instances, one component does not know what has changed in another component. However, it is able to detect if the other component as a whole is changed, and when that's the case, it will re-build the dependent targets as needed. To fix this, more fine-grained dependencies between components should be stated in the build system.

More dependencies than necessary Many C++ files are including header files that are not strictly needed by the C++ file. Over time, people added include statements for new code, however once the code was rewritten or removed, some of those include statements became useless. Since there is no good way of warning the user about unused includes, developers did not know that these includes became useless and slowing the build system down. As of May 2013, Mozilla is aware of this, and is fixing this by using a tool called *include-what-you-use*.[1]

Fixing the problems mentioned above takes a lot of time, because the source tree is large and a less efficient build system means stalled development. At the moment, Mozilla is looking for a more automated way to re-architect its build system.

A synthetic project

To illustrate the difference in build time between recursive and non-recursive build systems, consider the following synthetic project:

- There are three modules: `a`, `b` and `c` and each module has one sub module: `a` has `a1`; `b` has `b1`; `c` has `c1`.
- Each sub module contains three source files: `a1` has `a2.c`, `a3.c`, and `a4.c`. Similar for sub modules `b1` and `c1`.
- The final targets are the binary `c` and the shared libraries `a.so` and `b.so`.
- The shared library `a.so` is a result of `a{2,3,4}.o`. The shared library `b.so` is a result of `b{2,3,4}.o` and `a.so`. The binary `c` is a result of `c{2,3,4}.o` and `b.so`.

[1] `http://code.google.com/p/include-what-you-use/`

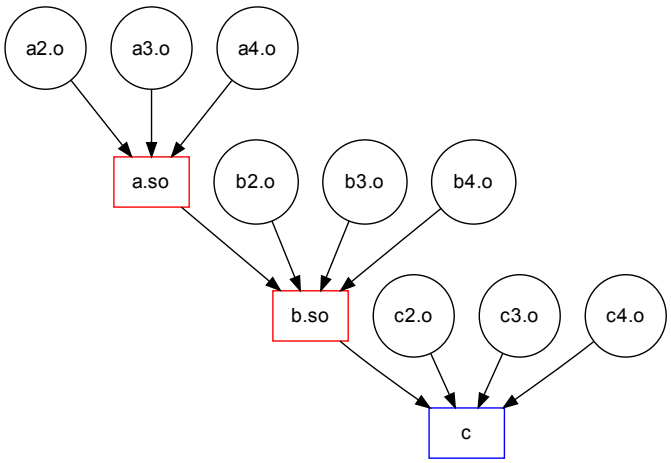

Figure 5.4: In order to resolve the dependencies of the targets, GNU Make will construct this directed acyclic graph (DAG). Source files to object files dependency edges are omitted in the DAG for clarity.

The build system for the synthetic project

For the synthetic project mentioned above, both a recursive build system and a non-recursive one are given. In order to avoid the creation of dummy source files, the command `sleep 2` is used for source file compilation and `sleep 3` for linking the object files. The values 2 and 3 are artificial; they are used to show the difference in build time between the two build phases in a clear way.

```
MODULES := a0 b0 c0

all: $(MODULES)

$(MODULES):
        make -C $@

b0: a0
c0: b0
```

Code example 1: Example of a root Makefile of the synthetic recursive build system. Notice the recursive calls to Make using `make -C`, which will cause the blocking behaviour. Since the module `a0` is responsible for producing `a.so`, the module `b0` depends on the module `a0`. The module `c0` depends on `b0`, because the module `b0` will produce `b.so`.

```
all: a.so

a_deps: ; $(MAKE) -C ${@:%_deps=%}

a.so: a_deps
        @echo "$@ start"; sleep 3; echo "$@ done"
```

Code example 2: Example of the `a0` module Makefile of the synthetic recursive build system. The command `${@:%_deps=%}` removes `_deps` from the target name to enter directory `a`. Substitute `a` with `b` or `c` accordingly.

```
OBJ_FILES := a2.o a3.o a4.o

all: $(OBJ_FILES)

$(OBJ_FILES):
	@echo "$@ start"; sleep 2; echo "$@ done"
```

Code example 3: Example of the `a0/a` submodule Makefile of the synthetic recursive build system. The rule for compiling C source files to object files is an implicit rule in Make and therefore omitted in the Makefile. Substitute `a` with `b` or `c` accordingly.

While the root Makefile of the recursive build system is not that large, the module and submodule are of a similar size. This is not a problem at first. Once the build system evolves, a recursive build system can become a burden to maintain.

The non-recursive build system on the other hand has a larger root Makefile, where it includes the smaller module and submodule Makefiles. The build recipes are stored in one central place. The dependencies and targets are stored in the child Makefiles, based on the source directory layout. This makes non-recursive build system easy to maintain over time, but also harder to do right at the beginning.

```
TARGETS := a.so b.so c

include a0/Makefile
include b0/Makefile
include c0/Makefile

OBJ_FILES := \
	$(OBJ_FILES_a0) \
	$(OBJ_FILES_b0) \
	$(OBJ_FILES_c0) \

all: $(TARGETS)

$(OBJ_FILES): ; @echo "$@ start"; sleep 2; echo "$@ done"
$(TARGETS): ; @echo "$@ start"; sleep 3; echo "$@ done"

a.so: $(OBJ_FILES_a0)
b.so: $(OBJ_FILES_b0) a.so
c:    $(OBJ_FILES_c0) b.so
```

Code example 4: Example of a root Makefile of the synthetic non-recursive build system.

```
OBJ_FILES_a0 :=
include a0/a1/Makefile
```

Code example 5: Example of a module Makefile of the synthetic non-recursive build system.

```
OBJ_FILES_a0 += a2.o a3.o a4.o
```

Code example 6: Example of a submodule Makefile of the synthetic non-recursive build system.

Note that both implementations are overly simplified, since there are, for example, no `CFLAGS` set, no recipe to create the executable `c` and the command `sleep` is used for the compilation and linking phase. However, both are sufficient for this experiment to show the problems involved with recursive build systems.

If all jobs are executed in sequential order, benchmarking the results of both implementations will result in the same build time (26 seconds). There are no underutilized cores, since Make can only use one core. Therefore, both implementations are benchmarked with two concurrent job slots (`make -j2`) using `time`. The `time` tool is part of the GNU Bash shell, which gives timing results in the order of milliseconds. There was no need for more precise timing results in this benchmark, because `sleep` is used.

Measurement	Recursive	Non-recursive	Optimized
Build time `-j2`	21 sec.	17 sec.	16 sec.
% of total time	100%	81%	76%

Table 5.1: Build system comparison. The recursive build system will spawn a new Make instance for every sub directory, the non-recursive build system uses a single Make instance, and the optimized build system is non-recursive and will perform the tasks in the optimal order.

Although this is a synthetic benchmark, figure 5.5 clearly illustrates the bottleneck of recursive build systems: once a new Make process is spawned, the parent Make process has no idea when the child Make process will be finished, or what the child Make process is doing.

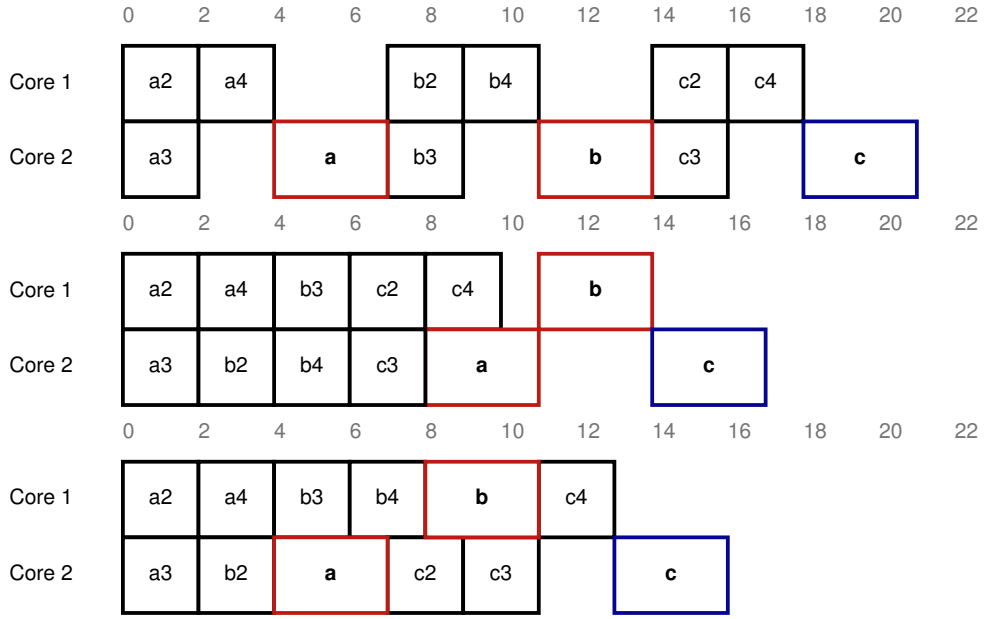

Figure 5.5: Visual comparison of the recursive, non-recursive and optimized build system with two processor cores. The blank areas between the boxes in a row indicate underutilization of the processor core.

At this point, there is no systematic way for translating a recursive build system into a non-recursive one. Since most large code bases have a rather inefficient build system for multi-core CPUs (e.g. Firefox), or large code bases that are not supported by GNU Gold yet (e.g. the Linux kernel)[2], it is hard to experiment with concurrent linking in a real world scenario.

[2]Linking the Linux kernel does not work at the moment, because the kernel is using undocumented features of GNU ld's version scripts.

Firefox's JavaScript engine

In order to prove that recursive build systems are also inefficient in real world scenarios, this experiment targets the build system of Firefox's JavaScript engine (located in `js/src`). Because the build system had to be rewritten manually from scratch, a single but complex component of the Firefox JavaScript engine is used for the experiment.

In this experiment, the source code is compiled for development/debugging purposes (thus less optimized than release builds). The reason for this setting is that concurrent linking is a useful technique during development. Development builds are built often, and a fast build implies a shorter *edit-compile-run* cycle.

The following results are obtained from a machine running Gentoo amd64 with an Intel i7, 8 GB ram, and using GCC version 4.5.3. The used GCC code optimization are `-Os` and GNU Make was running with `-j12`.

Task	Recursive	Non-recursive	Non-recursive + `ar` fix
Complete rebuild	30 sec	26 sec (87%)	25 sec (83%)
`touch` jsbool.h	25 sec	22 sec (88%)	21 sec (84%)
`touch` jsreflect.h	6.7 sec	4.3 sec (64%)	2.3 sec (34%)
avg. % of build time	100%	80%	67%

The *complete rebuild* task is a build of `js/src` from scratch. The `touch` *jsbool.h* and `touch` *jsreflect.h* task will only rebuild the files that depend on the header `jsbool.h` resp. `jsreflect.h`. The `jsbool.h` file is included (directly or indirectly) in most files, while `jsreflect.h` is only included in two files.

This experiment clearly demonstrates the importance of a non-recursive build system: switching from a recursive build system to a non-recursive one reduces the build time with 20% on average. If you optimize the build system (like the `ar` fix[3]), the build time for `js/src` is reduced with 33% on average.

Concurrent linking in practice

The build analysis tool described in section 5.1 can visualize the duration of spawned processes in a build. Combined with a non-recursive build system and concurrent linking, it is possible to measure the effect of using concurrent linking. At this moment, reading the symbols and adding the symbols to the symbol table can benefit from concurrent linking. Relocating the symbols – the most beneficial task – is not altered, because the modifications made the linker produce invalid output and/or let the linker crash. Therefore, only the improvements from reading the symbols and adding the symbols in a concurrent way were measured.

One of the first noticeable things is the creation of the linker process earlier on. This is a good sign because it means that Make spawns the linker process right after the first dependency is ready. When the object files were in the file system cache, concurrent linking reduced the linking time by 8.1% on average. This means that 0.48 seconds are saved of a total linking time of 6.1 seconds for a cached build.

When the object files are not in the file system cache, concurrent linking reduces the linking time by 22% on average. This means that 12 seconds is saved of a total linking time of 55 seconds for a non-cached build. This may seems like a large improvement, but it is important to note that this is an one-time saving: the next rebuild will read the object files from the file system cache.

The time saving could be explained by improved I/O operations caused by mixing the I/O operations with CPU-bound processing. Prior to concurrent linking, compilation utilises the CPU while the disk was not used that much. Once the compilation was done, the linker kicks in and starts doing I/O-heavy work.

[3]The build system was invoking `ar` unnecessarily on Linux. It was originally added because archive files are not indexed on Mac OS X.

At the moment, the linker reads the object files one-by-one in the order of the command line arguments. If an object file is sent to the linker but the preceding object file is not yet processed, the linker will wait until the filename of the preceding one is received through the pipe. Eventually, this could be solved by marking the symbols that are inserted out-of-order into the symbol table, and once the preceding object file is ready, fix the out-of-order inserted symbols.

There were also two more practical issues with concurrent linking: How to detect if Gold is used as linker, and how to detect if Make supports concurrent linking? This can be solved using a `configure` script, which will set `USE_GOLD_LINKER` when the testcase detects that Gold is used as linker. The `USE_GOLD_LINKER` variable can be used in the Makefile to add the flag `--concurrent-linking` to the `LDFLAGS` when Gold is in use.

Make supports a compile-time variable `.FEATURES`, which is accessible in Makefiles. This variable can be used to verify that concurrent linking is supported by the Make instance if the string `concurrent-linking` is a substring of `.FEATURES`. This allows developers to create backwards-compatible Makefiles: recent versions of Make can benefit from concurrent linking, while old ones continue to work as expected.

CHAPTER 6

Related work

Incremental linking Instead of always building a binary from scratch, incremental linking tries to *patch* an existing binary with the new code sections. The result is a binary that differs from the binary produced when a full rebuild is carried out. This is unlike concurrent linking, which produces a binary output identical to that produced during a full rebuild.

Parallel linking Apply the linking process to two or more input files in parallel and merge the result. Parallel linking will produce the same output as with concurrent linking. The difference is the work unit: parallel linking works on a object file level, while concurrent linking works on a process level.

Link time optimization In a multiple C-file based program, it is hard to optimize a C-file because the compiler cannot know for sure how a function is used across the file boundary. Link time optimization is a technique used to optimize a function which is called in a different file, by combining the static analysis information from both files. It basically delays the optimization phase until the linker knows which object files are used in the final binary.

Besides the difference of linking techniques mentioned above, concurrent linking can be combined with one or more of these techniques to improve the build time even more. It is possible to combine incremental linking with concurrent linking because the linker can still substitute the old data and code sections with the content of the new object files in a concurrent way.

Parallel linking is more difficult to combine with concurrent linking, but still possible to do. The combination with parallel linking has the disadvantage that the build system does not know what the linker is doing precisely. The build system knows if the linker does either something or nothing at all (it is blocked on I/O and waiting for input from the build system). This makes it harder to determine what amount of CPU cores the linker can use. It depends on the compiler-linker-job-ratio, because compilation takes generally more time than linking. If there are more linker threads waiting for object files, those resources can also be used for compilation. However, if there are a lot of available CPU cores, parallel linking can improve the build time.

CHAPTER 7

Conclusion

Once the multi-core CPU became mainstream, the time to build a program was not reduced by using multiple cores. On top of that, the raise of dynamic shared libraries caused more work for the linker to perform during link-time. It is possible to improve compilation speed by reducing the optimization level, but this is not possible for linking. Linking scales linearly with the symbols in the binary regardless of the optimization level. Since the link-time increased, the edit-compile-run cycle became much longer.

The delay caused by the linker has become a real problem for large code bases. This thesis investigated what concurrent linking and build systems can do to reduce this delay.

Although build systems are not the main topic of this thesis, build systems have a high influence on build time. As demonstrated with the Mozilla build system experiments, switching from a recursive build system to a non-recursive one reduces the build time with 33% on average.

For incremental, non-recursive builds where only one file is modified, on average 0.10 seconds are saved during linking time from the total build time of 2.1 seconds. This is not a large improvement. However, once `Relocate_task` is properly adjusted to concurrent linking, reductions in the order of 50% can be expected during linking time.

On top of the 33% time saving, concurrent linking reduces the total build time with another 2.4% on average. Thus combining concurrent linking with a non-recursive build can reduce the build time with 35% on average.

Generally, linking is only a small part of a rebuild from scratch. However, when a few files are modified, linking claims a larger percentage of the total build time. For these incremental builds, concurrent linking offers a great way to reduce the build time while the output remains the same.

To conclude: In order to use concurrent linking effectively, it is important to have a multi-core-aware build system. Otherwise, the improvements gained from using the linker in a concurrent way are not noticeable.

CHAPTER 8

Future work

There are still some missing pieces in the current implementation, and are considered future work.

Gold supports linking with multiple threads, but how many threads should be used for linking? This depends on the available job slots, compilation-linking ratio, and compiler optimization flags. In order to keep the linker running at full speed, there is a certain job ratio between the compiler and linker. If the compiler is producing less work than the linker can handle, the linker is waiting for input files. On the other side, if there is more work produced by the compiler, the linker can use more than one thread to speed up the linking phase.

Can Make use the job slot of a suspended Gold linker? If the Gold linker is blocked (waiting for filenames from the pipe), the job slot is in use for a suspended task. If Make can temporarily use the token for another job, the amount of concurrently running jobs can be optimal.

Adjusting `Relocate_task` to benefit from concurrent linking was much harder than expected. Besides modifying the data structures, the current design assumes that every object file is available. In the end, this meant that `Relocate_task` had to be rewritten in order to work properly together with concurrent linking. This was way more work than expected, and therefore considered future work.

An analysis of concurrent linking in combination with incremental and/or parallel linking would be interesting as well.

CHAPTER 9

Appendix A: GNU Gold linker tasks

This document lists the available internal GNU Gold linker tasks with description, grouped by task subject, and alphabetically ordered. It also contains a graph of the relationships between the various tasks in GNU Gold.

The linker task subjects are:

1. Global tasks
2. Input tasks
3. Relocation tasks
4. Runner tasks
5. Output tasks
6. Plugin tasks

Overview of the GNU Gold linker tasks

Global tasks

`Task_function` Waits for a blocker and then runs a function. It is just a closure.

Input tasks

`Add_archive_symbols` Read an archive and pick out the desired elements and add them to the link.

`Add_lib_group_symbols` Pick out the desired elements from the library group indicated by a `--start-group` and `--end-group` linker flag and add them to the link.

`Add_symbols` Add the symbols to the symbol table. These tasks must be run in the same order as the arguments appear on the command line.

`Dir_cache_task` Read the directory content, to fill the cache.

`Finish_group` Finish up handling a group. It is just a closure.

`Read_relocs` Read the relocations for an object file, and then queue up a task to see if they require any GOT/PLT/COPY relocations in the symbol table. If garbage collection or identical comdat folding is desired, we process the relocations first (`Gc_process_relocs`) before scanning them (`Scan_relocs`). Scanning of relocs is done only after garbage or identical sections are identified.

`Read_script` Read a file which was not recognized as an object or archive. It tries to read it as a linker script, using the tokens to serialize with the calls to `Add_symbols`.

`Read_symbols` Reading the symbols from an input file. This also includes reading the relocations so that we can check for any that require a PLT and/or a GOT. After the data has been read, this queues up another task (`Add_symbols` or `Add_archive_symbols`) to actually add the symbols to the symbol table. The tasks are separated because the file reading can occur in parallel but adding the symbols must be done in the order of the input files.

`Start_group` Starts the handling of a group. It exists only to pick up the number of undefined symbols at that point, so that we only run back through the group if we saw a new undefined symbol.

`Unblock_token` If we fail to open the object, then we won't create an `Add_symbols` task. However, we still need to unblock the token, or else the link won't proceed to generate more error messages. We can only unblock tokens when the workqueue lock is held, so we need a dummy task to do that. The dummy task has to maintain the right sequence of blocks, so we need both `this_blocker` and `next_blocker`.

Relocation tasks

`Allocate_commons_task` Allocate the common symbols. We use a blocker to run this before `Scan_relocs`, because it writes to the symbol table just as they do.

`Gc_process_relocs` Process the relocations to figure out which sections are garbage. Very similar to scan relocations.

`Relocate_task` Perform all the relocations for an object file.

`Scan_relocs` Scan the relocations for an object to see if they require any GOT/PLT/COPY relocations.

Runner tasks

`Close_task_runner` Closing the file.

`Gc_runner` Arranges the tasks to process the relocations for garbage collection.

`Layout_task_runner` Mapping the input sections to output sections and laying them out in memory. This task is queued when all input object files have been read.

`Middle_runner` Arranges to run the functions done in the middle of the link. It is just a closure.

Output tasks

`Write_sections_task` Write out data in output sections which is not part of an input section, or which requires special handling. When this is done, it unblocks both `output_sections_blocker` and `final_blocker`.

`Write_data_task` Write out data which is not part of a section or segment.

`Write_symbols_task` Write out the global symbols.

`Write_after_input_sections_task` Write out data in output sections which can't be written out until all the input sections have been handled. This is for sections whose contents is based on the contents of other output sections.

Plugin tasks

`Plugin_finish` Runs after all replacement files have been added. For now, it's a placeholder for a possible plugin API to allow the plugin to release most of its resources. The cleanup handlers must be called later, because they can remove the temporary object files that are needed until the end of the link.

`Plugin_hook` Handles the "all symbols read" event hook. The plugin may add additional input files at this time, which must be queued for reading.

`Plugin_rescan` Rescan archives as needed.

GNU Gold linker tasks handling

This document describes the implementation of the work queue, locking and handling of tasks inside the GNU Gold linker. It also contains an interaction graph of various components inside the GNU Gold linker.

Main initialisation

The function `main()` (defined in `main.cc`) will:

- **Initialise the work queue** based on the given number of threads. See `Workqueue::Workqueue()`.
- **Enqueue initial tasks** using the given command line options. See `queue_initial_tasks()`.
- **Bootstrap the work queue** and spawn the needed threads. See `Workqueue::process()`.

Once the work queue is empty, `main()` continues with:

- Generating statistics (optionally).
- Outputting the cross reference table.
- Handling of warnings and errors (if any).

Note that only the task handling relevant actions are listed above.

Initial tasks

GNU Gold will start with initialising a `Read_symbols` task for each input file given at the command line. The `Read_symbols` tasks are chained together using `Task_token` tokens, which act as a dependency constraint. Theses dependencies are blocking and used to add the symbols to the symbol table in order.

After creating the `Read_symbols` tasks, a `Task_function` will be enqueued, which will have a dependency on the last `Task_token`. The `Task_function` will trigger either the `Gc_runner` or the `Middle_runner`, depending on whether Garbage Collection or Identical Code Folding is enabled. The `Gc_runner` will enqueue `Read_relocs` tasks to identify unused code sections. After the `Gc_runner` finishes, the `Middle_runner` task function will be enqueued.

Middle tasks

The `Middle_runner` task function invokes `queue_middle_tasks()`. This will queue up the middle set of tasks. These are the tasks which run after all the **input objects have been found** and all the **symbols have been read**, but **before output file is layed out**.

First, allocate common symbols using `Allocate_commons_task`. This task uses a blocker to run the task before the `Scan_relocs` tasks, because it writes to the symbol table just as they do.

Second, read the relocations of the input files. This is done to find which symbols are used by relocations which require a GOT and/or a PLT entry, or a COPY relocation.

Finally, when all those tasks are complete, laying out the output file can start. Laying out the output file is done in the task function `Layout_task_runner`.

Final tasks

The task function `Layout_task_runner` will call `queue_final_tasks()`, which will enqueue the final task set.

First, create the `output_sections_blocker` blocker to block any objects which have to wait for the output sections to complete before they can apply relocations. This blocker is unblocked when the `Write_sections_task` task is finish.

Second, create the `input_sections_blocker` blocker to wait until all the input sections have been written out. This blocker is unblocked when all `Relocate_task` tasks are finish.

Third, create a blocker to block the final cleanup task. This blocker will be unblocked when `Write_symbols_task`, `Write_sections_task`, `Write_data_task` and all `Relocate_task` tasks are done.

Last but not least, the task `Close_task_runner` is enqueued to close the output file. This task blocks `Write_after_input_sections_task`, depending on which is the latest enqueued task.

Work queue overview

During the compilation phase of GNU Gold is determined if the work queue has support for multi-threading. By default, multi-threading support is disabled. It can be enabled using the configure flag `--enable-threads`. If threads are enabled at compile time, `Workqueue` class will use the `Workqueue_threader_threadpool` class as its thread handler. Otherwise, the `Workqueue_threader_single` class is used as its thread handler.

Bibliography

[1] David William Barron. *Assemblers and Loaders*. Elsevier Science Inc., New York, NY, USA, 3rd edition, 1978.

[2] Pat Beirne. Study of ELF loading and relocs. `http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html`, 1999. [Online; accessed August 2013].

[3] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[4] Leon Presser and John R. White. Linkers and loaders. *ACM Comput. Surv.*, 4(3):149–167, September 1972.

[5] Paul D. Smith. GNU Make "jobserver" implementation. `http://mad-scientist.net/make/jobserver.html`, 2003. [Online; accessed August 2013].

[6] Ian Lance Taylor. Linkers part 4: Shared Libraries. `http://www.airs.com/blog/archives/41`, 2007. [Online; accessed August 2013].

[7] Ian Lance Taylor. GNU Gold Linker. `https://events.linuxfoundation.org/slides/lfcs2010_taylor.pdf`, 2010. [Online; accessed August 2013].

[8] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutler, and Koen De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on*, pages 7–12. IEEE, 2005.