Incremental Linking on HP-UX

Dmitry Mikulin
Hewlett-Packard Company
mikulin@cup.hp.com

Murali Vijayasundaram
Hewlett-Packard Company
vm@cup.hp.com

Loreena Wong
Hewlett-Packard Company
loreena@cup.hp.com

Abstract

The linker is often a time bottleneck in the development of large applications. Traditional linkers process all input files, even if only one or two objects have changed since the previous link. To shorten link time, we have developed an incremental linker for HP-UX which only processes modified files. Users can take advantage of the performance gains without modifying their usage patterns of the existing HP-UX linker since the incremental linker is implemented on top of the regular 64-bit linker. In addition to the tasks of the normal linker, the incremental linker must save extra information about input files, symbols and relocations, allow for the expansion of existing files and addition of new ones by allocating padding spaces in the output file and use this information to perform in-place updates. The results of several different design considerations and tradeoffs are materialized in link-time performance gains of up to thirteen times that of a normal link for large applications.

1. Introduction

1.1 Motivation behind incremental linking

In recent years application sizes have grown dramatically. This increase has been enabled by significant advances in software development tools such as object-oriented languages which shorten development cycles and allow smaller groups of developers to produce very complex applications with hundreds of thousands or even millions of lines of code. Understandably, these applications take longer to compile and link. This increase becomes most obvious during application debugging when multiple edit-compile-link-debug cycles must be performed in a short period of time. Most changes involve only a small portion of an application source base, typically one or two modules. The problem of long compile times is partially solved by 'make' utilities that recompile only modified source files. Unfortunately there is no easy solution for traditional linkers—they still need to process all object files and libraries to resolve cross-module references and assign addresses to all symbols. The linker becomes a bottleneck that significantly affects a developer's productivity.

The only way to shorten link time, apart from making algorithmic improvements in the linker itself, is to reduce the size of input processed by the linker. Since only a few object files are typically modified before a link is performed, it is possible to reduce the input size by only processing modified objects and replacing their contribution to the output files in place. This fact is the motivation behind the incremental linker.

An incremental linker should process only those input files that have been modified since the last time a link was performed. With meticulous planning and saving of necessary data, all other input files that have not changed should not need to be reprocessed.

1.2 Related work

Despite the great importance of fast link times and the fact that several production systems have implemented incremental linkers (IBM mainframes had incremental linking capabilities since the 1960s; Sun Microsystems offers this functionality which can be optionally enabled on their Solaris systems; Microsoft C/C++ Development Studio has it enabled by default in debug links), very little has been written on the subject.

An academic paper by Quong and Linton [1] provided us with valuable analysis of padding space allocation and reuse. It also contained important performance data for programs of various sizes. Even though the authors chose a different approach in their implementation (a memory resident component which maintains state information about the incremental linker), the paper gave us insight into usage patterns as well as performance and implementation trade-offs.

A paper by Hoffman and Curwen [2] described an alternative way of solving the problem of long link times based on dynamic linking. Object files comprising the application are grouped into shared libraries which are smaller and therefore can be linked faster. Source changes cause only enclosing libraries to be rebuilt. This approach primarily addresses linking simple applications and will fail in cases with name collisions between shared and archive libraries. Dynamic linking is not always supported (e.g. in embedded systems) and may require significant build process changes which makes the scheme not very practical.

1.3 The HP-UX standard linker

The main job of the linker is to merge several relocatable object files into a single load module (a shared library or an executable program file). In doing this merge, the linker must resolve references across the input objects, layout and assign addresses to the resulting load module. In addition, the linker must be able to handle references to symbols defined outside the current load module. For example, if several object files are linked together with a shared library, the referenced code in the shared library is not copied into the output file. Instead, the linker creates dynamic symbols for these types of references and places information in the output file so that when the proper shared library is brought in at runtime by the dynamic loader, the symbol reference can be bound based on the load address of the program file and shared library. Dynamic symbols are also created for all global symbols which may be referenced outside the load module.

Another key task of the linker is to perform symbol resolution—the process of matching references to the definition of a symbol. If a reference can be bound to a definition within the load module, the linker can simply replace the reference with the address of the symbol. If the reference can not be bound to a definition within the load module, the linker will create a dynamic relocation. The dynamic relocations will be processed by the dynamic loader at runtime.

The linker also supports a –r option to merge multiple object files into a single relocatable object. When the –r option is specified, the linker will retain the symbol and relocation information in the output file, making it suitable for subsequent re-linking.

2. Overview of incremental linking

Incremental linking support on HP-UX is implemented as part of the standard 64-bit linker. Aside from a few exceptions, the vast majority of linker options and functionality is available with incremental linking as well, enabling users to take advantage of the performance gains without sacrificing linker functionality when building shared libraries and executables.

The 64-bit linker now has three operating modes:

- ☐ Normal link mode: Normal linker operation. This is the default mode.
- ☐ Initial incremental link mode: This mode is entered when the +ild option is specified and the output load module (executable or shared library) does not exist or the output load module is not an incrementally linked executable. In this mode, the linker will create an output load module that is suitable for incremental linking.
- ☐ Incremental link mode: This mode is entered when the +ild options is specified and the incrementally linked output load module exists. In this mode, the linker will make incremental updates to the output load module. It is in this mode that the great performance gains can be realized.

This paper focuses only on the latter two modes.

In initial incremental links, the linker processes all input object files and libraries the same as it does for normal links. In addition to this basic functionality, the incremental linker must do additional work to enable subsequent incremental links. The linker must store extra information about the input files processed, all global symbols, as well as relocations. Also, the incremental linker must allocate proper padding space for text, data, bss, and other sections in the output file in order to allow room for future expansion from additional input files, definitions, references, etc. Because of this added functionality required for initial incremental links, the time spent in initial links is slightly higher than that of a normal link.

Once all the proper information has been stored in an initial incremental link, subsequent incremental links can be performed. The linker uses timestamps on individual input files to determine which files have changed and only reprocesses modified files during incremental links. The linker uses saved relocation information to patch the symbolic references in the rest of the output file. These tasks are described in greater detail in the rest of this paper.

The incremental linker is intended for use by programmers during developmental stages only, and should not be used for release builds of products. Because the incremental linker pads sections for future expansion, programs are bloated in size and contain information not necessary for execution of the program. In addition, the incremental linker is incompatible with most compile- and link-time optimization techniques and thus cannot produce optimized executables. The incremental linker, however, is an excellent timesaving tool for use during development when programmers are constantly adding and modifying small portions of code and rebuilding programs for testing. In addition, incrementally linked programs are still source-level debuggable in the same manner as normally linked executables.

The remainder of this document describes design considerations and implementation details, and performance results of the incremental linker for HP-UX.

3. Command line, file and library processing

Because our incremental linker is implemented as an extension of the regular system linker, practically all options are allowed when building incrementally linked executables and shared libraries. The use of the mapfile option, which allow users to change the layout of the output load module through a mapping file, is also supported, as long as the specified mapfile doesn't change between incremental links; if it does, we fall back to an initial incremental link. The only options incompatible with incremental linking are link time performance and optimization options. Some linker options significantly influence the resulting output module and if added to or removed from the command line, may make the executable difficult or impossible to incrementally update. For this reason, we decided not to allow any command line option changes in incremental links with the exception of tracing and verbose options; these can be freely added, removed or changed since they have no effect on the resulting output file.

In an initial incremental link, file names, types and time stamps for all object files, shared libraries, archive libraries and their processed members are saved in the output file. In subsequent incremental links, this information is used to detect which files need to be reprocessed. For object files, the logic is simple: if a file's time stamp has changed, that file needs to be reprocessed.

If an archive library is modified, we need to check each individual member processed in previous links for time stamp changes and reprocess only the ones that were actually modified. In some cases, we may need to reprocess an archive library even if the archive itself was not modified. If any of the modified object files introduces a new unresolved symbol reference, we need to scan archive library symbol tables and extract members that define that symbol. However, contributions of archive members are never removed from the output file, even if there are no more references to any of the symbols defined in those members.

Since symbols from shared libraries are not propagated into the output file, we do not reprocess shared libraries during incremental links. Instead, it is the dynamic loader's responsibility to catch unresolved symbols at runtime.

4. Padding and reuse of space

The incremental linker re-links programs by inserting modified object code into the existing output file. During the initial incremental link various output file sections such as text, data, bss etc., are padded with additional space for future expansion. The output file data structures like symbol table, section table and linkage tables are also padded with additional space.

During incremental links, the linker will vacate the space occupied by modified object files. The vacated space in the output file will be reused when the contents of the modified object files are copied over to the output file. The incremental linker always tries to fit the modified object file's contributions into their previous location. After several incremental links, the padding space may become exhausted. When this occurs, the incremental linker will fall back to performing a full initial incremental link during which additional padding space will be allocated.

4.1 Padding space allocation

The incremental linker allocates two kinds of padding spaces:

- File specific padding space. Each section in the output file consists of contributions from all object files. The Figure 1 shows the layout of an output section created during normal link. The incremental linker allocates a padding space after contributions form each object file. These file specific padding spaces are allocated to allow for possible growth in the object file's contributions. Figure 2 shows the layout of an output section with file specific padding.
- Generic padding space. During incremental links new object files may be added to the link. To accommodate contributions from new object files, a large padding area (as shown in Figure 2) is allocated at the end of each output section.

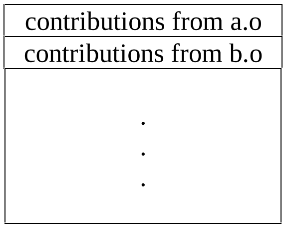

Figure 1. Layout of an output section by the normal linker

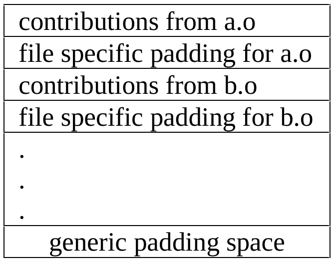

Figure 2. Layout of an output section by the incremental linker

4.2 Keeping track of section layout

To enable the incremental linker to efficiently locate and replace a modified object file's old contributions, we maintain a data structure called a linkmap in the output file. The linkmap consists of two sections: a file table and a section map.

- File table. The file table contains a mapping of file identifiers to path names of object files. The file identifier is a unique number assigned to each object file present in the link. The contributions from each object file are laid out in the output sections in sorted order based on the file identifiers of the object files.
- Section map. The section map records the layout of the output section. For each output section we maintain an array of section mappings. Each mapping consists of the following information:
  - File identifier of the contributing object file.
  - Section relative offset of the location where the object file's contributions are copied over.
  - Size of object file's contributions.
  - Size of file specific padding space.

  Just as the output section contents are sorted on file identifiers of contributing object files, the section mapping data is sorted on file identifiers.

4.3 Replacing old contributions

During incremental links we perform the following steps to copy contributions from modified object files to the output file:

- Locate the modified object file's old contributions in the output section by searchin through the section mapping data. Since the section mappings are sorted on file identifiers, a binary search can be performed to locate the object file's contributions.
- Check if the object file's old contributions and file specific padding space is sufficient to fit the new contributions. If the contributions fit, they are copied into the existing space in the output file. The file specific padding increases the chance that the modified file's contributions will fit into their previous location.
- If the contributions no longer fit in their pervious location, fall back to a full initial incremental link.

If there are any new object files in the incremental link, their contributions are copied over to the generic padding area. We also assign increasing unique numeric values as file identifiers to new object files. This keeps the section layout sorted on file identifiers. The file table and section mapping data in the linkmap are consistently updated during incremental links.

5. Symbols

One of the main responsibilities of the linker is to maintain information about symbols, resolve undefined and multiply defined symbols, and assign symbol values. Even though basic rules to maintain both static (.symtab) and dynamic (.dynsym) symbol tables are the same, more information is needed to deal with symbol resolution and assigning symbol values in incremental links. The linker generates various data structures for every symbol—linkage table entries, dynamic relocations, etc. To avoid inefficient use of file space and reuse these structures in incremental links we need a means of keeping track of them. Also, resolving symbols with multiple definitions, some modified and others unchanged, is a non-trivial task. A 100% solution for these challenges would require a substantial amount of additional information to be saved, maintained, and processed on every link, which could negate all advantages of the incremental linker. We had to find a solution acceptable in a vast majority of cases, yet very lightweight both in terms of space and processing time.

5.1 Symbol table management

The existing symbol table structure intermingled local and global symbols without distinction as to which input file defined each symbol. For incremental links, that structure was not sufficient. We needed a structure that allowed us to modify only those entries in the symbol table that pertained to the modified file(s) in an incremental link. In addition, padding areas must be allocated in the correct places to allow for additional symbol definitions, if any, from modified files.

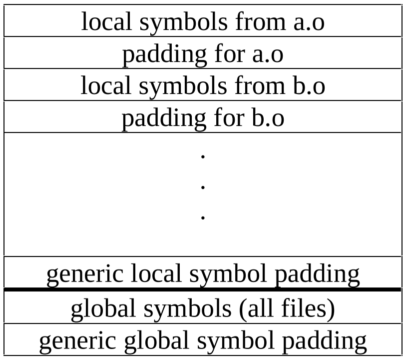

Figure 3. Symbol Table Layout

The symbol table structure is subdivided into two main pieces (Figure 3), one for local symbols and one for global symbols. In addition, there is a parallel secondary symbol table (.ild_symtab) which stores incremental linking-specific information about symbols. Since all local symbols for a given file will be reprocessed in incremental links, information regarding them is not stored in the .ild_symtab. Instead, only information about global symbols is stored there. The additional information is used mainly for symbol resolution in incremental links, which is described in the next section. During incremental links, the symbol tables are reconstructed from the original output file, and all modified global symbols are updated in place. Any new symbols can be added into the appropriate padding area. Once the link is complete, only the modified sections of the symbol tables are rewritten in the output file.

Unlike global symbols, local symbols are not reconstructed in incremental links. Since these symbols are local to the modified file only, the resolution rules are much less complex, and there is no reason to store any additional information about these symbols in the .ild_symtab. The local symbols section of the symbol table is sectioned by file identifier, one section per file. Each file-specific section for local symbols has its own padding space, and there is an additional generic padding section at the end of the locals for any additional files that may be added in incremental links (see Figure 3). The linkmap stores the mapping for the entire symbol table, including each file-specific local symbol section. During incremental links, the linkmap helps determine the symbol indexes of local symbol definitions for modified files and the local symbols are reprocessed, overwriting the old definitions. This is different from the way global symbols are processed in that they are not updated in place, but rather they are completely reprocessed.

5.2 Resolving symbols

The linker resolves symbols based on their definition types. This is a fairly easy task when there is only one definition for a particular symbol in a link—all references are resolved to that definition. However, symbol resolution becomes more complicated when a symbol has multiple definitions such as common (storage request), weak, and strong definitions. In a regular link a symbol with the strongest definition type ‘wins’ over symbols with lower precedence definition types. In incremental links only new and modified files are processed, which means that the linker only scans symbol tables from those files. Situations when only one object file contains a symbol definition are easy to handle. In incremental links we need to update the symbol definition in the output symbol table only if the object containing the definition was modified. If the symbol has multiple definitions it is sometimes difficult to pick the new winner. To illustrate this problem, we will look at a few examples.

Suppose we have two object files, a.o and b.o. One file (a.o) contains a storage request for ‘data1’ integer; the other (b.o) contains an initialized definition for ‘data1’. In a regular link the definition from b.o wins symbol resolution. Suppose the user made changes to remove `data1' from b.o altogether. The regular linker would pick the definition from a.o as a winner. But in an incremental link, if a.o was not modified, the linker does not have enough information to pick the new winner. Ideally, for every symbol we would need to have a list of definitions we have seen in previous links so that we can find a new winning definition and resolve all references to the new symbol. This approach would require us to store quite a bit of additional symbol information in a form of symbol definition lists. These lists, like all other data structures, would need to be padded in case new symbol definitions are added, and updated in incremental links. This solution would add complexity to the implementation, requiring more storage and increasing processing time.

Instead we decided to go with a more lightweight solution which resolves symbols correctly in the majority of cases. In cases where the incremental linker cannot determine the new winner, it will fall back to an initial incremental link.

In an incremental link we maintain two copies of every symbol: the old winner— a winning definition from the previous link restored from the output file; the current winner— a winning definition we have seen so far while processing modified files. At the end of the first pass over modified objects, the linker has to make a decision for every symbol as to what version to pick as the new symbol definition (See Figure 4).

If the object containing the old definition was not modified we just follow regular logic to perform symbol resolution between old and current definitions. If the defining object was modified, there are two possibilities as to which should be the current winner with respect to the old winner. If both symbols come from the same object file, we pick the current winner. If they come from different files, we check which version of the symbol would win regular symbol resolution by the normal linker. If the current version wins, we have no problem; we just take it as the new winner. If the old version wins, we have to do an initial incremental link because we do not have enough information to declare any symbol a winner. If we had a complete list of symbol definitions from the previous link, we could find a symbol which would have won symbol resolution had the old winner not been there at all. But since we decided against maintaining this list, the only available option to recover is to fall back to a full initial incremental link.

```
if (old copy not modified) {
   do regular resolve(old, current)
}
else {
   if (old and current are
       from the same file) {
      resolve to current
   }
   else {
      if (current wins resolve(old,
                              current)){
         keep current as new winner
      }
      else {
         if (old and current are common){
            resolve to current
         }
         else {
            do initial incr. link
         }
      }
   }
}
```

Figure 4. Symbol resolution algorithm.

This may not sound like a good option since it may cause frequent re-links. However, it is not as bad as it seems. There are only a couple of ways to create multiple symbol definitions that do not cause a duplicate symbol error. One is to use common symbols; the other is to use weak definitions. System libraries primarily use weak symbols to prevent user name space pollution. Therefore the only way to get into a situation when the linker has to perform an initial incremental link caused by weak symbols is when a user removes a strong definition overriding a weak symbol from a system library. Under normal circumstances this change will not be performed very frequently. The situation is very similar with common symbols. If a user has multiple commons and an initialized (winning) definition for a symbol, and the winning definition is removed, fall back to the initial incremental link. Even though a change like this is likely to be more frequent, it will only account for a small percentage of code modifications and will not cause an overwhelming number of re-links.

One more corner case involves common symbols and can be handled without re-links. Suppose a user has multiple common definitions with the same name but different sizes. The definition with the biggest size wins symbol resolution. If in an incremental link the winning symbol is removed, we don't have to do a full initial incremental link. Since space has already been allocated, we can always pick the current winner even if it has smaller size.

5.3 Symbol resolution in libraries

The way shared and archive libraries are processed slightly changes the behavior of the linker with respect to reporting unresolved symbols. Since dependent shared libraries are not processed in incremental links, the linker is unable to tell whether a certain symbol is unresolved because there is no definition for it at all, or because its definition is in one of the dependent libraries. In the latter case, there is no problem; the dynamic loader will resolve the symbol at run time. To enable this dynamic symbol resolution, the incremental linker converts all potentially unresolved symbols into dynamic symbols during all dynamic links and lets the loader report errors in case these symbol cannot be found in any of the dependent libraries at runtime.

Because archive members are never removed from the link, run time behavior of some incrementally linked programs may differ from that of programs linked by a normal linker. Suppose you incrementally linked a shared library, liba.sl. One (and only one) of the objects (a.o) in the link referenced a global function func() which was resolved by an archive member func.o from lib.a. Now you remove the call to func() from a.o, so func.o is no longer needed. But since it is not removed in a subsequent incremental link, the symbol 'func' will remain exported by liba.sl and available for look-up. Thus, an application linked with liba.sl and referencing func() will still be able to successfully find it, even though it would have failed if liba.sl had been linked by a regular linker.

In our experience, the cases described above are not very frequent and there is a simple remedy to fix them – perform an initial incremental link.

6. C++ compile-time template instantiation

The HP-UX C++ compiler uses COMDAT to support compile-time template instantiation. COMDAT is a scheme that allows multiple, duplicate copies of code and data to be merged together by the linker into a single copy in the final executable. The comdat allows the compiler to generate multiple copies of a template function in separate object files. The linker must identify sections containing duplicate information and choose one of the copies for inclusion in the output file. The content of a COMDAT section is essentially a directory for the members of the COMDAT set. The section consists of an array of section indexes that point to the member section's entries in the object file section table. When object files containing COMDAT groups are linked, there may be more than one copy of a given COMDAT group. The linker chooses one of these copies to include in the final output file and discards the rest. If the same COMDAT group is defined in multiple files, they are assumed to be functionally equivalent.

In the context of incremental linking, the linker has to handle three different situations: modification of an object file containing a COMDAT group, addition of a new COMDAT group, and removal of a COMDAT group from an object file.

To support full functionality, the incremental linker has to keep track of the list of COMDAT groups contributed by each object file. For each COMDAT group in a modified object file, the incremental linker must check if the COMDAT group was chosen from this object file in the previous link. If so, it should invalidate the COMDAT group in the output file and replace it with the modified COMDAT group. When a new COMDAT group is added to an object file, the linker has to check whether the COMDAT group is already present in the output file. If it is, the COMDAT group should be invalidated and replaced with the new COMDAT group from the modified object file. If the COMDAT group is not already present, it should be added to the output file. Ideally when a COMDAT group is removed from a file, the linker should decide whether to physically remove the group from the output file or replace it from another object file. This scheme would require an extensive amount of bookkeeping. Also the additional information would need to be updated during incremental links. In a big C++ application there may be thousands of COMDAT groups. For example, in one of the test programs we analyzed, there were approximately 130,000 incoming COMDAT groups of which about 40,000 were unique. This scheme would greatly increase the complexity of the implementation, requiring more storage and potentially increasing the incremental link time.

Instead we decided to implement a simplified scheme that does not require maintaining any additional information. In our simplified scheme, the linker chooses a COMDAT group from one of the modified objects and discards the rest. If the COMDAT group is already present in the output file, it will be invalidated and updated with the new version. It is more difficult to handle the case when a COMDAT group is removed from an object that contributed it in the previous link. There are two cases to consider: either the same COMDAT group is defined in another object file or no other object file defines the same COMDAT group. Since we don't keep the list of all COMDAT groups defined by object files, we cannot determine whether the original COMDAT group has been replaced by the next available COMDAT group from a different file. In the second case, if the linker does not remove the COMDAT group, it is possible for the linker to miss a potential unsatisfied symbol error or report duplicate symbol definition that it wouldn't have in a normal link. Instead of implementing a complicated scheme and sacrificing performance, we fall back to a full initial incremental link when a COMDAT group is removed from an object file that contributed it in the previous link.

7. Linkage tables

Linkage tables (LTs) are generated by the linker to enable position independent code and data accesses. Our linker creates three kinds of linkage tables: PLT—procedure linkage table, DLT—data linkage table, and OPD—official procedure label descriptor table.

Each linkage table type is maintained in a similar fashion. As runtime components, LT entries are not allowed to change their position from one incremental link to another. Symbols contain indexes of these entries in order to have access to their corresponding LT entries. These indexes are assigned once for every symbol and after that never change in incremental links. It is very hard to come up with meaningful criteria to group LT entries. DLT and PLT entries are driven by symbol references. With multiple references in multiple objects, it is impossible to attribute an LT entry to any particular file. OPDs are driven by definitions, but in incremental links, definitions may move from object to object. So we decided not to group them at all. Instead we update them in place and rely on our underlying I/O buffering to capture any locality. Linkage tables only use generic padding for all new entries.

8. Import stubs

PLT entries are created in response to direct function calls that are potentially outside the load module we are currently building. These calls are redirected to an import stub that loads a procedure address and the target module's global pointer (GP) from a PLT entry and branches to that address. Calls to local functions can be relocated at link time and do not normally require import stubs. This means that in incremental links, all call sites to modified local functions would have to be relocated. To avoid saving all PC-relative relocations in an output file, we decided to use a different approach. In incremental links all PC-relative calls are directed to go through import stubs. This way we only need to update PLT entries for modified functions to insure that call site and target are connected correctly. One may argue that we added extra overhead for direct calls which slows down the application at run time, but since incremental linker is intended for debugging purposes only, runtime performance is rarely an issue.

Another problem we encountered was how to handle fixing up call sites to import stubs themselves. Normally, a single stub is created to service all PC-relative calls to a particular function. These stubs are attached to text contributions of objects for which they were generated. This makes them impossible to locate in incremental links. Also, if an object for which an import stub was created changes, we need to adjust references to this import stub for all unmodified files as well. This operation can be costly in terms of space (we would need to save information to keep track of import stubs) and, more importantly, link time (we would need to apply relocations for unmodified files). To make the operation simple and fast we decided to generate one import stub per symbol per input object file. The stubs are recreated in incremental links only for modified files and only direct calls from modified files need to be relocated.

9. Static and dynamic relocations

Another key linker functionality is to resolve external symbol references. In an object file these references are expressed as relocation records. Relocations from object files are processed and applied at link time if possible; if not, they are transformed into dynamic relocations applied by the dynamic loader at run time. Correctness of incremental links largely depends on our ability to maintain and process relocations.

Relocations can be broken up into several major categories. The first category is relocations applied to text sections. These relocations must be applied at link time and do not generate dynamic relocations because at run time text is not writable and cannot be relocated. The second category of relocations is those applied to linkage tables. These relocations are generated by the linker itself and may or may not require dynamic relocations depending on symbol types, output module type (executable or shared library), link type (static or dynamic), etc. And finally, there are relocations that are applied to data sections. These mostly deal with static data initializations and can generate dynamic relocations.

9.1 Text segment relocations

All relocations on text sections are in one way or another converted into linkage table relative relocations. Therefore if linkage table updates are done correctly, we do not need to worry about saving static relocations for these sections and reapplying them in incremental links. However, for error reporting purposes, we need to know what symbols were referred in all LT-relative relocations.

Suppose a.o defines a function foo() and b.o calls that function. If in an incremental link the definition is removed from a.o and b.o remains unchanged, we need a way to tell that foo() was referenced from b.o and issue an unresolved symbol message. If the definition for foo() was modified (it's address changed) we just need to update the OPD and the PLT entries for it. Since we don't actually need to apply relocations of this sort for unmodified files, we decided not to save them in their entirety, but rather save symbol indexes: one entry per referenced symbol per object file. In incremental links we scan entries from unmodified files and issue appropriate warnings if corresponding symbols are no longer defined. Unresolved references from modified objects will be detected as part of regular symbol and relocation processing. For locality, all entries are grouped by file and use both file specific and generic padding to accommodate expansion.

9.2 Linkage table relocations

Linkage table entries may require dynamic relocations. Changes of symbol attributes in incremental links may require new dynamic relocations to be generated for entries that did not have them in previous links. Some entries that had dynamic relocations in previous links may not need them any more. This means we must maintain a correspondence between linkage table entries and dynamic relocations for those entries to be able to perform updates. We decided that the easiest and the most efficient way to deal with this problem is to keep linkage tables and LT dynamic relocations in parallel tables. We avoid maintaining any additional information to help us find relocations for LT entries as well as using potentially costly look-up schemes.

9.3 Data segment relocations

Unlike linkage table relocations, we could not avoid saving input relocation records for data segment relocations. We need to reapply these relocations in incremental links for symbols that were modified even though files containing these relocations remained unchanged. We also need to update dynamic relocations if symbol attributes change. For these cases, we extended the standard relocation record to contain an index of the corresponding dynamic relocation. These relocations are grouped by file for better locality of updates and use both file specific and generic padding to accommodate expansion. In incremental links, all records from unmodified objects are scanned. If a symbol is modified, the relocation is applied; a corresponding dynamic relocation is updated if needed.

9.4 Runtime behavior

One of the initial design requirements of the incremental linker was to ensure that the runtime behavior of programs remained the same for incrementally linked programs versus normally linked programs. This meant that there could be no changes to the format, layout, and semantics of any dynamic structures, including dynamic relocations. But as mentioned earlier, we needed to pad dynamic relocation sections for expansion. Also, in incremental links, symbol attribute changes may no longer require dynamic relocations for structures that required them in previous links; these relocations have to be wiped out. We had to use meaningful relocations that would be understood by the dynamic loader without changing the runtime behavior of programs. Consequently, in incremental links we create a dummy common symbol and fill all padding areas with relocations for this symbol. As a result, application start-up time is slightly slower, but since the target use of the incremental linker is for debugging, runtime performance is rarely an issue.

10. Performance

We measured the performance of the incremental linker using the following four programs of varying sizes: bison, the GNU parser generator; gcc, the GNU C compiler, and a large C++ customer application.

Table 1 shows the parameter for each of these sample test programs. All measurements were taken on a HP N4000 server with eight PA-8500 440Mhz CPUs. The system has 16 gigabytes of RAM and runs HP-UX 11.00 operating system.

	# of objects	Source language	#of symbols	Size of output file (Mb)
bison	**23**	**C**	**517**	**0.24**
gcc	**261**	**C**	**7423**	**2.93**
customer program	**1717**	**C++**	**389444**	**118.71**

Table 1. Sample Test Programs

	Normal link	Initial incremental link		Incremental link	
	sec	sec	% of normal link	sec	#times faster than normal link
bison	**0.5**	**0.6**	**120**	**0.15**	**3.3**
gcc	**3.5**	**4.1**	**117**	**0.41**	**8.5**
customer program	**131.0**	**158.0**	**120**	**11.70**	**11.2**

Table 2. Link Time Measurements

Table 2 shows the link time data. We measured and compared the normal and initial incremental link times. The initial incremental link is generally slower than the normal link due to the extra work done in these links. For each of the sample programs we measured, we found that the initial incremental link takes about 17% to 20% more time than a normal link with the same sources.

We next measured the incremental link times after making changes to two object files. For large programs, the incremental link is on average ten times faster than the normal link. The time taken by the incremental linker depends on the amount of code modified. Regardless of the amount of code modified, the time spent on extracting information from the output file will always be the same. We measured the link time on the largest test program (customer program)—after changing up to 30 large object members in an archive library. The link times are shown in Table 3. Even when large number of objects are modified, the incremental link is significantly faster than the normal link. For example, re-linking after changing 30 objects takes about 16.7 seconds which it about 8 times faster than the normal link.

#of objects changed	Incremental link time (sec)
2	**11.3**
4	**11.8**
6	**13.0**
8	**13.1**
10	**13.9**
30	**16.7**

Table 3. Incremental link times

The Table 4 shows the increase in size of the sample programs due to incremental linking. The size increase is due to two reasons: additional incremental linking data stored in the executable and the padding space.

	% increase in size	% increase due to padding
bison	**185**	**152**
gcc	**76**	**65**
customer program	**57**	**29**

Table 4. Program size increase

11. Current status and future work

The implementation was completed only a few months ago and the general HP-UX developer community hasn't had a chance to use the incremental linking capabilities. However, a few internal partners successfully used the incremental linker and provided us with valuable feedback on its correctness, usabily and performance.

Even though a tremendous amount of work has been done to design and implement the incremental linker, there are still a few areas that could use improvement. Currently our linker does not handle removal of symbols and objects very well. Symbols are never removed from either the static or dynamic symbol tables. This may cause undefined behavior of dynamic programs which reference removed symbols. Also, removal of object files from the link line is not handled the way it should be and causes an initial incremental link. Shared libraries are not processed at all in incremental links. Even though this reduces link time, the runtime behavior of programs may be confusing for less advanced users and they might want to have an option for the incremental linker to handle shared libraries the way regular linkers do.

Currently we don't handle padding space very efficiently, particularly in the area of accommodating file expansion. When a file's contribution to an output section uses up all file specific padding, we do a full initial incremental link even though there may be enough space in the generic padding area to store it.

The linker has all the information it needs to deal with these situations, and we will implement these improvements in the near future to reduce the number of limitations and differences between applications and libraries produced by the regular and the incremental linkers. These enhancements will also improve efficiency of file space reuse and shorten incremental link time.

12. Conclusions

We have developed an incremental linker for HP-UX that significantly shortens the edit-compile-link-debug cycle by substantially improving the link time. For large programs, our incremental linker is an order of magnitude faster than the normal linker is, allowing developers faster turnaround after simple bug fixes.

In our design we have chosen to forgo complex schemes that require vast amounts of bookkeeping in an attempt to guarantee incremental linking 100 percent of the time. Instead, we chose lightweight schemes that address the majority of situations that occur in practice. When rare corner cases are encountered, we fall back to performing a full initial incremental link. This choice has significantly reduced the complexity of the incremental linker and maintains the excellent performance gains achieved by the incremental linker. While there are still some enhancements that can be made, the incremental linker has already proven to be an extremely useful developer's tool with significant performance gains.

References

[1] Linking Programs Incrementally. Russel W. Quong, Mark A. Linton. ACM Transactions on Programming Languages and Systems, Vol. 13, No. 1. January 1991.

[2] Pseudo-Incremental Linking for C/C++. William A. Hoffman, Rupert W. Curwen. Dr. Dobb's Journal, October 1999.

[3] 64-Bit Run-Time Architecture for PA-RISC 2.0
http://www.software.hp.com/STK/partner/pa64rt.pdf

[4] ELF-64 Object File Format
http://www.software.hp.com/STK/partner/elf-64-hp.pdf

[5] HP-UX Linker and Libraries User's Guide
http://docs.hp.com/dynaweb/hpux11/dtdcen1a/lnkuen1a