SYZYGY – A Framework for Scalable Cross-Module IPO

Sungdo Moon, Xinliang D. Li, Robert Hundt, Dhruva R. Chakrabarti,
Luis A. Lozano, Uma Srinivasan, and Shin-Ming Liu

Java, Compilers, and Tools Lab, Hewlett-Packard Company
11000 Wolfe Rd, Cupertino, CA 95014
{sungdomo,davidxl,rhundt,dhruva,lozano,uma,shin}@cup.hp.com

Abstract

Performing analysis across module boundaries for an entire program is important for exploiting several runtime performance opportunities. However, due to scalability problems in existing full-program analysis frameworks, such performance opportunities are only realized by paying tremendous compile-time costs. Alternative solutions, such as partial compilations or user assertions, are complicated or unsafe and as a result, not many commercial applications are compiled today with cross-module optimizations.

This paper presents SYZYGY, a practical framework for performing efficient, scalable, interprocedural optimizations. The framework is implemented in the HP-UX Itanium® compilers and we have successfully compiled many very large applications consisting of millions of lines of code. We achieved performance improvements of up to 40% over optimization level two and compilation time improvements in the order of 100% and more compared to a previous approach.

1. Introduction

Interprocedural optimizations (IPO) have been known to provide substantial runtime performance benefits. To name a few, cross-module procedure inlining, indirect call promotion, dead global variable elimination, and code and data layout optimizations are proven methods for improving performance of an application. For many of such optimizations, like dead function and global variable elimination, whole-program analysis is a must for ensuring the correctness of program execution. Others, such as those benefiting from interprocedural pointer alias analysis information, are more effective when performed in whole-program mode. Yet other transformations, such as procedure inlining, are able to yield maximal potential performance as the scope is enlarged to include the entire application.

Nevertheless, IPO has not seen wide-scale uses amongst application developers and vendors, as long compilation time and huge memory consumption present significant usability hurdles to the deployment of whole program optimization. In a naive implementation, the working set for interprocedural optimizations grows in a super-linear fashion with the size of application. The sheer size and the amount of swapping it induces put insurmountable pressure on the virtual memory system. We have also observed that such implementation very quickly reaches its limits in file set handling when thrown at large applications.

One of the existing techniques for addressing these problems is partial IPO, i.e., performing interprocedural optimizations on a subset of the files constituting an application. This technique is more of a workaround than a viable solution because it cannot apply transformations that rely on whole-program analysis. In addition, it requires guidance either directly from the application developer or from an extra profiling pass to determine the subset of the application files that are important. Another common technique is to use user assertion options, such as telling the compiler that the addresses of global variables are not taken anywhere in entire application. In General, the use of such assertions is unsafe and requires intimate knowledge of the details of the application. They also do not prove to be adaptive to changes in the application over time. Other assertions like linker options for binding or specifying the link order require significant user intervention to exploit optimization opportunities and are once again not adaptive to changes in the application. Some compilers use elaborate memory management schemes to manage large working sets such as offloading information when a certain memory usage threshold is reached [3]. However, they typically run into a problem because of not knowing how large a working set should be allowed to grow. By the time the threshold is reached it is usually too late and the offloading itself becomes a bottleneck.

In this paper, we describe *SYZYGY*, a scalable framework for performing IPO that provides a viable solution to developers and vendors in terms of usability and runtime performance of applications. We informally define *scalability* of an IPO compilation process in terms of achieved compilation time at a lower optimization level (O2) and in terms of utilized disk space compared to a compilation producing debug information (`-g`). In particular, for our purpose, an IPO compilation is considered to scale if the total compilation time overhead compared to an O2 compilation and the required disk space compared to a debug compilation do not exceed the a factor of 3.

We achieve scalability through various means, which will be explained in greater detail in the remainder of this paper. Firstly, we push as much functionality as possible into the parallelizable front-end and back-end parts of the compiler. Secondly, we generally try to split optimization phases into analysis and transformation phases, minimizing the size of additional in-memory or persistent data necessary for communication between the phases. Thirdly, the number of read and write operations on intermediate files is kept to a minimum during the compilation process and lastly, the intermediate file format has been designed carefully to minimize disk size and to optimize access times.

The rest of this paper is organized as follows. Section 2 introduces our view of an ideal IPO compilation model. Section 3 provides a description of the SYZYGY IPO framework and several implementation issues, as well as IELF, our persistent intermediate representation (IR). In Section 4, compilation time and performance measurements are presented. Section 5 addresses related work, followed by the conclusions.

2. IPO compilation model

There are many, often conflicting, objectives to accomplish during the compilation process, such as correctness, high performance, reasonable compilation time, debuggability, and so on. Putting aside the correctness, the indispensable goal of the compilation, optimizing compilers usually put more emphasis on achieving higher performance with the expense of longer compilation time. In case of IPO compilation, however, especially for large applications with

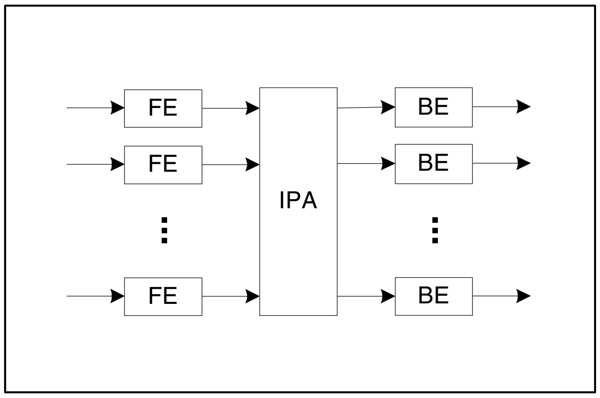

Figure 1: Overview of IPO compilation model.

thousand of files and millions of lines of codes, an excessive compilation time can be a major obstacle for its wider acceptance.

In designing the SYZYGY IPO framework, the major emphasis has been put on achieving scalable compilation time for large applications, while preserving analysis precision and optimization opportunities as much as possible. The incremental IPO recompilation model [5], where the compilation is confined to the portions of the previous compilation that are invalidated by the source code changes, is one solution during recompilation of large applications in the event of small source code changes. While it is an attractive partial solution and possibly a future extension of our framework, in this paper we focus on the compilation of whole programs.

The proposed IPO compilation model splits a standard compilation process into three distinct phases as illustrated in Figure 1: front-end (FE), interprocedural analysis engine (IPA), and back-end (BE). Both front-end and back-end operate on a single module at a time. The interprocedural analysis engine is the only place where whole program is seen and the interprocedural information is computed. While FE and BE can be executed in parallel at the module granularity taking advantage of multiple CPUs available within a machine or across a pool of networked machines, IPA is sequential in nature and thus become a major bottleneck in overall compilation time. As a result, it is very important to minimize the time taken during IPA to achieve scalable overall IPO compilation.

In par with the compilation model, all interprocedural optimizations are also divided into three phases: summary collection, interprocedural analysis, and interprocedural optimization.

2.1 Summary collection (FE)

FE collects the module-level and procedure-level summaries that are needed for interprocedural analysis. The *summary* for a given optimization is a minimal subset of program representation that is essential in computing interprocedural information to determine the legality and the profitability of an optimization. Certain summaries are shared amongst several independent interprocedural analyses. Before collecting summaries, FE first applies a series of simple transformations, such as constant folding, IR canonicalization, algebraic expression simplification, etc., to present the IR to subsequent phases in a simplified and canonical form. Then standard scalar optimizations are performed to assist the collection of much more compact and precise summary information.

The use of summaries for interprocedural analyses has several benefits. Firstly, it significantly reduces the working set. Secondly, summaries for an individual module can be computed independently in parallel with the summary computation for other modules. Also in case of recompilation, the summaries for unmodified modules can be reused without recomputing them. Lastly, the effect of certain interprocedural optimizations can be easily reflected on the summaries without modifying the IR. Therefore, well-designed summaries are crucial for IPA to efficiently compute effective interprocedural information.

2.2 Interprocedural analysis (IPA)

IPA processes a set of modules that constitute the whole compilation and builds the global symbol table and the call graph after performing name resolution and type unification. Summaries gathered by FE are associated with the corresponding call graph nodes or edges. Then a series of analyses are performed interprocedurally based on the summaries on top of the call graph and the local/global symbol tables. Interprocedural analysis results are gathered in an appropriate form and passed to BE either to direct interprocedural transformations or to allow further local optimizations with refined information.

In non-IPO compilation, optimizations have to conservatively assume the behavior of non-visible parts of a program. The purpose of most interprocedural analyses is to collect the behavior of whole program and assist in improving the quality of local optimizations. For instance, the results of interprocedural constant propagation can be used by local constant propagation or other local optimizations to make safe assumption on the possible values of a formal argument.

There are several interprocedural optimizations such as procedure inlining, procedure cloning, and indirect call promotion that need contexts from other modules in order to perform the required transformations. For example, inlining may need the callee procedure body from another module and indirect call promotion may need the target procedure symbol. Usually these kinds of transformations are performed during IPA, but can be deferred to BE to fully take advantage of BE's parallelism. This can be accomplished either by copying the required contexts or through the central program database shared by multiple BEs.

In this ideal IPO model, IPA does not directly reference or modify the code representation part of the IR; only the summaries are accessed. Although the actual code transformations are deferred, it is important for an individual analysis phase to update summaries reflecting what will happen later so that the following analyses can be performed with valid summaries. When transformations are applied, existing summaries become either invalid or less precise (conservative). The invalidated summaries should be appropriately updated for the correctness of subsequent analyses, while less precise (conservative) summaries can be safely used without an update.

A cross-module interprocedural compilation uncovers unique compilation time reduction opportunities over a normal compilation, that is, to remove redundant or unused compilation units, such as duplicate COMDAT[1] sections and dead functions. Interestingly, it is our observation that larger applications tend to have many of such opportunities, helping IPO to become a more affordable option.

2.3 Interprocedural optimization (BE)

BE consumes an individual module annotated with interprocedural analysis results computed by IPA and performs optimizations and code generation. The scope of optimizations performed within BE is either intra-module/inter-procedural or intra-procedural level. In case of cross-module/inter-procedural transformations, IPA already decided what to do and prepared IR with enough contexts from other modules such that the original cross-module transformations can be performed as intra-module transformations.

[1] COMDAT is an ELF section that can be defined by more than one object file. It is typically used for the compile-time template instantiation.

3. SYZYGY IPO framework

This section presents a detailed description of the SYZYGY IPO framework which is implemented in the HP C/C++/Fortran compilers for HP-UX Itanium® systems. First we describe IELF files, a persistent binary intermediate representation used to communicate between FE, IPA, and BE.

3.1 Intermediate representation (IELF)

An IELF file contains a language independent intermediate representation (IR) of a compilation unit. IELF files are regular 32/64-bit ELF [12] files containing an ELF header, a symbol table, a string table, a section table, as well as several sections for binary representation of the internal IR representing the compiled program.

In the memory representation, the IR is maintained in memory arenas (blocks of memory whose allocation and de-allocation are controlled by the compiler; similar to [11]). All the references to arena objects are done through unique identifiers (ID) instead of pointers. Each arena maintains an associated ID array which contains pointers to owned objects; the object's ID is the position of the pointer to the object in the ID array associated with its owning arena. In IELF files, ID arrays are stored in a separate ELF section as arrays of arena offsets. Another ELF section contains mappings for fast finding of an arena's ID array.

The usage of IDs for cross-referencing between arena objects enables a consistent in-memory and out-of-memory representation of the IR. As a result, when an IELF file is read in, persistent arenas can be directly `mmap`'ed into the core memory. All references to objects in persistent arenas remain valid when they are mapped from or unmapped to the IELF file. Only ID arrays need to be handled appropriately to achieve persistency. The conversion between pointers and arena offsets in ID arrays is performed when IELF files are read or written. The conversion of ID arrays is done in a lazy fashion whenever possible. For frequently-used arenas, such as the global symbol table, their ID arrays are always converted during IELF read/write. The ID arrays of other arenas are converted to array of pointers when first referenced and converted back to array of arena offsets on write. The conversion of an ID array is extremely fast and only requires one pass over it.

During various compilation phases, a large number of IR objects can be created and deleted resulting in fragmentation in the arenas, which in turn can result in poor locality. This can be severe particularly for the arenas of variable length objects where reusing of deleted space is difficult. Therefore, during IELF writing, the live objects within an arena are compacted together.

Typically, the size of an IELF file is on average five times larger than its real object file, but for certain pathological cases we observed overheads of 700 times compared to the real object file size. This can be easily understood. For example, given a source file containing a very short function with a reference to a field of a very large structure, its real object file would contain only a few instructions, such as a load of the structure address, a load of an field offset, and then the access itself. However, the IR has to represent the whole structure with field and type information. Therefore, it seems fairer to compare the size of an IELF file with that of a corresponding real object file containing debug information, where we experienced average overhead of slightly below 3 times.

Since IELF files contain binary data, care must be taken to guarantee the compatibility during the evolution of the compiler. We designed several ways to extend the content of IELF files in an upward compatible fashion so that newer compilers can work with IELF files generated from an older compiler. For example, an IELF file can contain arbitrary number of annotations, blocks of arbitrary binary data, and an object's ID can be used as an index into them. The compiler has to gracefully react depending on the existence of a certain annotation.

In summary, in designing the IELF representation, we made the classic engineering tradeoff and favored speed over size. Due to this design choice, reading of IELF files became extremely fast. Writing of IELF files turned out to be slower, and depending on machine characteristics, we experienced factors for writing over reading in the range from 2 to 10. Therefore, avoiding writing of IELF files during compilation is very important.

3.2. IPO framework

An overview of the SYZYGY compilation framework for cross-module IPO is illustrated in Figure 2. It divides a standard build process in four distinct phases: front-end, linker, ipa, and back-end.

The front-end (FE) takes input source files and generates output IELF files after a series of simple optimizations and summary gathering phases. The summary gathering and the generation of IELF files are controlled by either specifying "`+O4`", or combining a lower optimization level with the flag "`-ipo`". In cross-module IPO, IELF files are faked as real object files so that an existing build process can remain intact.

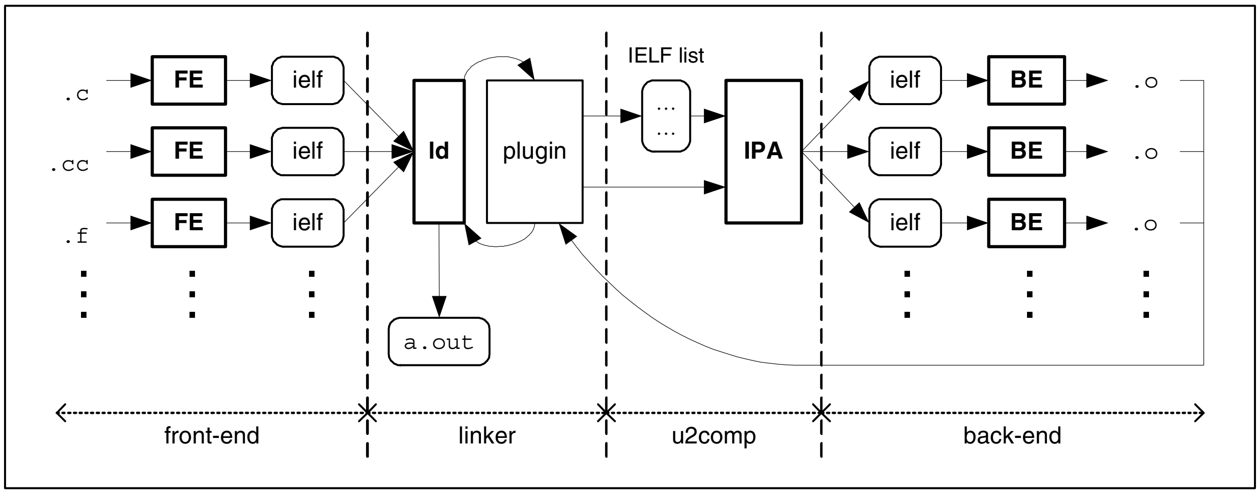

Figure 2: SYZYGY IPO framework.

At link time, the linker (`ld`) resolves all dependencies and determines the list of object files required for the final build. The HP-UX linker has a mechanism for automatic dynamic extension and allows loading of so-called *plugins*, shared libraries that provides certain functions that follow a specific naming convention. These functions are searched and called by the linker, passing arrays of function pointers back and forth to allow effective data exchange between the linker and the plugins. The IPO specific plugin claims front-end generated IELF files, declaring to the linker its intention to work with them. It assembles a list of the IELF files, extracts individual IELF files out of IELF archives if needed, and gathers linkage information provided by the linker. After that, it finally forks and executes `u2comp`[2].

The IPA process `u2comp` accepts as input the IELF files and the linker-feedback files, performs cross-module interprocedural analyses and transformations, and generates another set of IELF files that will be consumed by back-ends. In addition, `u2comp` generates a "`makefile`" that builds final targets (real objects files) for all the IELF files by invoking the back-end (BE). Before `u2comp` finishes, it invokes `make` in parallel mode on the generated makefile. The `make` process replaces `u2comp`, which is no longer needed.

The default number of back-end processes that `make` creates is set to the number of processors on the machine and can be user specified. Multiple flavors of make are supported, such as HP-UX `make`, GNU `gmake` [8], and IBM Rational `clearmake` [13]. The `clearmake` offers the capability of distributing builds over multiple machines. This "farming out" of back-ends together with a proper build environment infrastructure can lead to a dramatic reduction of the time spent in back-end optimizations and code generation.

After all back-ends finish, the control goes back to the linker, which then links the back-end generated real objects files with needed libraries to create the final executable.

3.3. Interprocedural optimizations

The ideal model presented in Section 2 advocates an IPO model where the interprocedural transformations are performed by BE using the interprocedural analysis results produced by IPA. With separate analysis and transformation, this can be easily accomplished for most interprocedural optimizations. As explained in the previous section, for some interprocedural transformations such as procedure inlining, IPA needs extra preparation work in order to defer them to BE. In the current implementation, we chose to perform procedure inlining in the `u2comp` process.

Procedure inlining can dramatically change the structure of a program, so it is challenging to appropriately update pre-inline summaries. For this reason, procedure inlining is performed last in the IPA phase ordering. Due to this decision, all the other interprocedural transformations are also performed during IPA just before procedure inlining. The program structure changes by inline transformation are too severe to apply transformations determined by the analyses performed before procedure inlining.

The `u2comp` phase starts with building global data structures such as global symbol table. We first remove redundant COMDAT sections and dead

[2] "`u2comp`" is a historical name for the HP-UX IPA executable.

functions to avoid unnecessary compilations spent on them during `u2comp` and back-end. After the duplicate COMDAT elimination, impacted modules are written out. Here, we could avoid writing of IELF files by using the same techniques described below. However, we chose to do that as the size of IELF files for C++ programs are greatly reduced after COMDAT elimination which helps reducing file I/O overhead for subsequent phases. Dead functions are identified right after the call graph construction and marked in the global symbol table, but the actual elimination is performed later, together with other transformations, to avoid IELF file writing. No summaries are impacted by these optimizations, except for the ones associated with removed objects which will be removed or ignored. Note that subsequent optimizations such as procedure inlining can result in more dead functions.

After that, several well-known interprocedural analyses follow. The analysis results can either be kept in memory until the next IELF file write or be written out to separate files depending on the amount of information. For the analyses that derive transformations, we keep lists of transformations to be performed and apply them together just before the procedure inlining phase. The transformations are applied module by module and the resulting IELF files are written out. Before the transformation phase, the effects of a certain transformation need to be visible to the following analyses by updating either summaries or local symbol tables, which demands writing of IELF files. In order to avoid continuous rewriting, we maintain patch lists and apply them to the summaries and the IR whenever the corresponding IELF file is read-in. Although patching is done multiple times, this process is extremely fast and introduces negligible overhead.

In summary, in our framework, all interprocedural optimizations excluding procedure inlining are performed with a single rewriting of IELF files (twice if duplicate COMDAT sections exist). This is accomplished by separating analysis and transformation phases, performing analyses on top of summaries, patching the IR on reads instead of rewrites, and applying a set of transformations in group at the end.

3.4 Cross-module procedure inlining

Procedure inlining [7] is a common optimization employed in compilers in order to reduce call overhead and enable optimizations that are otherwise not performed. Given a call-site from a caller routine to a callee routine, this technique replaces the call-site with a copy of the code for the callee. The elimination of the call-overhead directly improves performance (barring any potential I-cache penalties due to increased routine size). Another key benefit of procedure inlining is to increase the scope seen by the scalar/loop optimizer and instruction scheduler. While run-time performance improves in general, there are compile-time challenges when procedure inlining is applied to thousands of files.

A traditional approach such as [2] works on the raw program representation during the inlining phase. While this works pretty well for small programs, the memory consumption of the compiler easily hits the system limit for larger applications. The method presented in [3] solves this problem by offloading data structures onto disk to free up process memory. While this scheme allows the memory consumption of a compiler to stay within system limits, the compilation time greatly suffers due to loading, offloading, and thrashing.

Our framework solves this problem by having the analysis phase operate solely on program summary information. The IR is not opened at all during the analysis phase. A minimum set of summary information is maintained in the persistent representation of the program. At the start of the inlining phase, a top-level driver collects these persistent summaries and creates consolidated in-core summaries suitable for the analysis. The inlining heuristics employed in the analysis phase operate on these in-core summaries.

In our framework, the inlining transformation phase attempts to keep a limit on the number of open files. The main motivation is to minimize the number of reads and writes of the IELF files. An ordering among the inlinable call-sites is dynamically generated during the transformation process in order to achieve this goal [6]. A limit is imposed on the number of open IELF files and the amount of memory they can consume. Once this limit is reached, IELF files are closed as necessary to accommodate opens of other IELF files.

The approach presented in [2] performs several iterations of procedure inlining and scalar optimizations in order to accurately capture the impact of post-inline optimizations to assist subsequent inlining decision. While this works pretty well for achieving run-time performance, it is usually disastrous for compilation time while compiling large applications. Our framework uses a single analysis and a single transformation phase. In order to anticipate the effects of a certain inline instance, the summaries are updated after every inline decision. This leads to a significant improvement in compilation time as shown in the next section. We believe that careful update of the summaries can capture much of the effects of

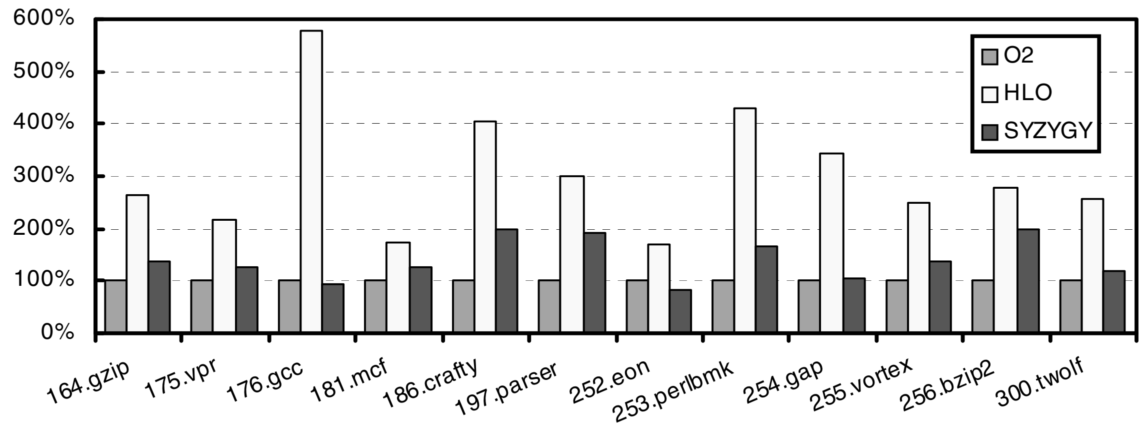

Figure 3: Compilation time comparison.

subsequent optimizations and it is not necessary to have a multi-pass inlining scheme.

4. Experimental results

Our experimental results are obtained on an HP server rx5670 with four 900 MHz Intel® Itanium® 2 processors, 8GB main memory, 1.5 MB L3 cache. We use the SPEC2000 integer benchmark suite for evaluating and comparing the compile-time and run-time behavior of our compiler (Figure 3 and Figure 4). We use a very large commercial application to analyze scalability parameters, such as memory footprint and CPU load (Figure 5).

The following three configurations are compared:

- **O2**: This is the default optimization level without any interprocedural optimization. All optimizations at this level are performed at the routine-level.
- **SYZYGY**: This setup corresponds to the highest optimization level, O4, which invokes interprocedural optimizations based on the SYZYGY IPO framework in addition to O2-level optimizations. The back-end phase is parallelized to 4 processes, while no front-end parallelization is used matching with O2 behavior. This setup also invokes the high-level loop optimizer which has minimal impact on the performance of our benchmark applications.
- **HLO**: This setup uses the previous interprocedural optimizer in the HP-UX compilers based on the framework presented in [2,3]. The set of optimizations performed in this setup is equivalent to the SYZYGY setup.

The compilation time comparisons illustrated in Figure 3 highlight the followings:

- **SYZYGY vs. O2**: The compilation time overhead of O4 over O2 is roughly less than two, which is a very positive result. We find this overhead factor confirmed in many large commercial and technical applications on machines with two or more processors. Note that large portions of O4 compilation time overhead are due to the compilation time increase in back-end optimizations due to increased routine sizes after aggressive procedure inlining. Also note that for some applications having a serial build process, the effects of the back-end parallelization were significant enough to make the O4 compilation faster than the O2 compilation (for example, 176.gcc and 252.eon). 252.eon is a good example for illustrating the unique advantages of IPO; duplicate COMDAT elimination greatly reduces back-end optimization time.
- **SYZYGY vs. HLO**: The new IPO model achieves a very large compilation time speedup over the previous model. This is a direct benefit of the approaches described in previous sections. It can be noted that compilation time speedup magnifies for larger programs. While for small program, such as 256.gzip which is comprised of 2 source files, the compilation time ratio is around 0.7, for large program, such as 176.gcc, the ratio is already 0.16, and it decreases more for even larger applications.

For commercial compilers, there are compilation time requirements. For example, customers who are adhering to a nightly build and test cycle need their compilation processes to finish within a given time span. The fact that HLO showed very large overheads

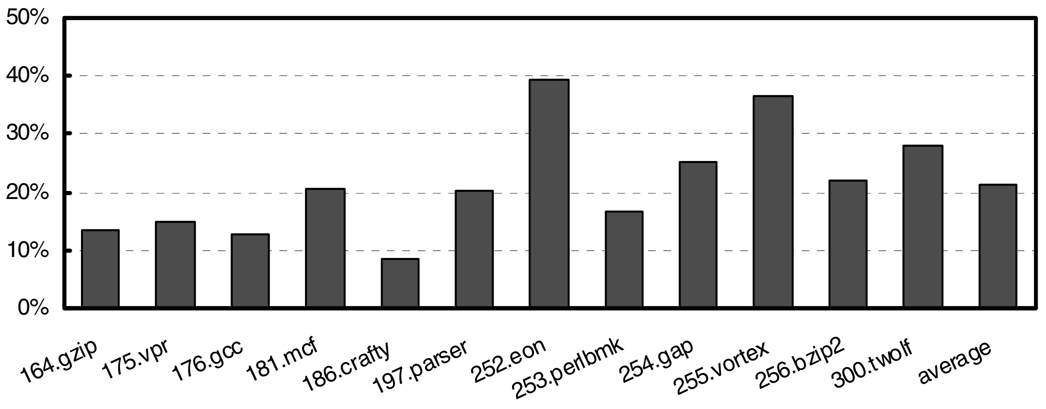

Figure 4: IPO performance improvement over O2.

over O2 caused many customers to abandon this compilation model, forgoing potential performance opportunities.

The performance improvement with O4 over O2 is illustrated in Figure 4. Run-time performance improves significantly when interprocedural optimizations are performed, which is an expected result. This is not only true for the SPEC2000 integer benchmark programs, where we observed an average improvement of over 20%, but we have also observed significant speedups on many large commercial and technical applications. The performance benefits obtained using SYZYGY versus using HLO are comparable, as the same set of optimizations is performed (there is a slight advantage to SYZYGY due to better inlining decisions).

In order to obtain a deeper understanding of the behavior of IPO, we monitored both CPU and memory usage during compilation of a very large commercial application and correlated this data with individual phases in `u2comp`. This application consists of about 5000 source files mainly written in C and has about 4 million lines of code. The call graph for this application has around 70000 routines and around 700000 call sites.

The `u2comp` binary runs as a 32-bit process and such a process has a 1GB address room available on HP-UX for text (code) and another 1GB for data (static data, stack, and dynamic data). The `u2comp` binary however is built as an EXEC_MAGIC binary, which is a HP-UX term indicating that text and data segments can be shared, resulting in an available data space of approximately 1.9 GB (there are some OS specific memory blocks allocated in these segments as well).

The graph in Figure 5 shows memory consumption and CPU load during the `u2comp` phase. The x-axis represents a compilation time of about 3.5 hours. The left y-axis represents memory consumption in Gigabytes, and the right y-axis represents CPU load in percent. The black line in the graph corresponds to the changes in CPU load over time during IPO. The filled areas of the graph represent as ordered from the bottom: text (code) size, which is only a very thin bar at the very bottom, stack size, which is set to a default of 256 MB by the kernel, static and dynamic data, which makes the bulk of the memory consumption, and the size of `mmap`'ed files, which are mapped into the current address space.

One can clearly distinguish I/O intensive phases, as well as CPU bound phases. The graph illustrates the following:

- Memory consumption is very low (about 200MB for dynamic data) and the CPU load is average until the call graph is constructed, which corresponds to the first little bump in the memory consumption after about one fifth of the IPO time.
- Memory consumption for the following analyses only increases moderately over the second fifth of the IPO time. The CPU load increases to high, indicating that analysis is CPU bound and performs more IELF file reading than writing.
- The third fifth of the IPO time also shows only moderate increase in memory consumption. However, the CPU load goes down drastically, which is caused by the IPO phase writing back the transformation results to the IELF files.
- The spike in the middle of the graph marks the start of the procedure inlining phase. Since we keep a

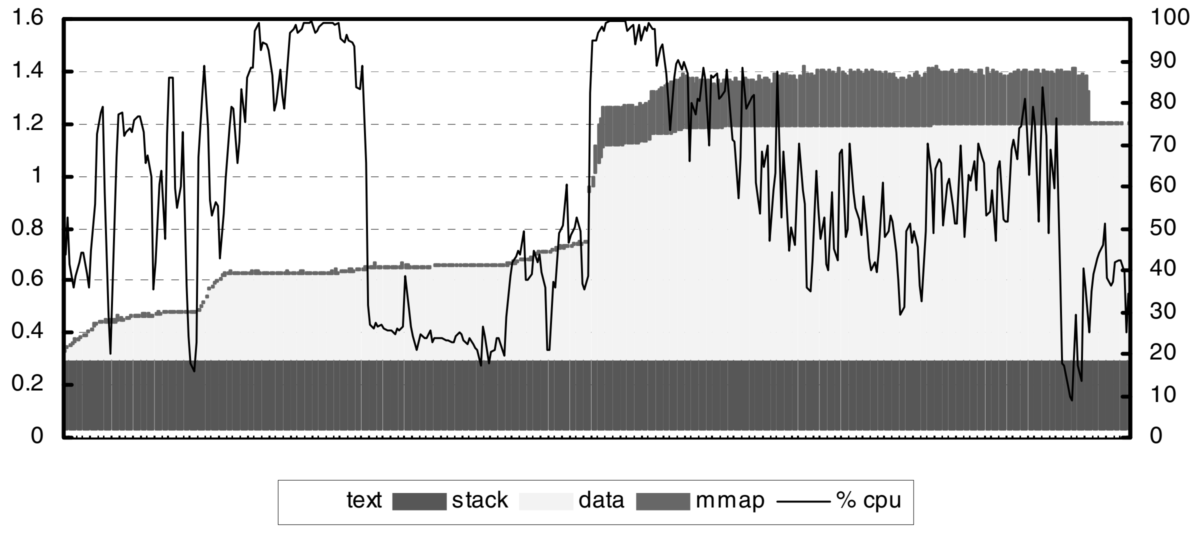

Figure 5: Memory usage and CPU load during IPA.

pool of IELF files open, the size of `mmap`'ed files increases as well as core memory usage. Note that the inlining heuristics try to open up as much as 500 MB in core. The inlining phase takes approximately 50% of the IPO time.

The total memory requirement for this compilation was about 1.4 GB, which is well below the ~1.9 GB available for a 32-bit process. The memory consumption can be scaled down even more by making the inliner file cache smaller. This however may increase compile time, as more files need to be rewritten.

5. Related work

Hall's dissertation addresses interprocedural optimizations and related issues [10]. This work mainly focuses on an algorithm for a call graph construction, a program representation for IPO, and some in-depth studies on procedure cloning and inlining. While this work is related to certain aspects of our IPO framework, in particular the inlining component, it does not cover as many issues or addresses as large scale a problem as our infrastructure does.

The approach described in [3] is a full-fledged implementation in a production compiler that has been deployed for some applications, and which now has been replaced with the SYZYGY IPO framework. From our experience, the inherent framework of the optimizer in [3] had serious limitations when it came to operation in whole program mode for large applications. While the implementation used a threshold based memory off-loading scheme, it did not pay any attention to the size of the working set a priori like our approach. It also did not propose any solutions for minimizing overhead due to intermediate file set handling.

The promising new infrastructure LLVM [16,17] for the `gcc` compiler tries to solve a similar problem to what we discussed in this paper and their front-end architecture is similar to ours. However, their post link optimization phase is monolithic and operates on a single merged "object" file. The scalability of this approach is not clear, as it is exposed to the same problems as the model in [3]. The Open Research compiler (ORC) [15] is a continuation of the SGI Pro64 compiler [9] and has a similar overall IPO architecture as our current implementation. We believe that the ideal IPO model, once implemented, would further improve overall scalability of IPO.

Some approaches such as [18] work on a much lower form of the IR, one that is closer to the target machine. The advantage for such link time optimizers is that the memory management issue is greatly simplified without the high-level symbol table representation. However, they are not as retargetable as high-level optimizers and working on a machine-level IR has its own set of unique problems. The gcc compiler server [4] offers a model where the compiler is loaded as a daemon process (or memory resident process) accepting compilation requests. The compiler maintains the IR for all input files before starting optimizations and code generation. This model seems to be more suited for compile time reduction and its scalability to very large applications is unclear.

6. Conclusion

In this paper we described our experience with the new SYZYGY framework for performing cross-module IPO. This framework has been implemented and deployed on HP-UX Itanium® platforms. Using this framework, we have successfully compiled dozens of large applications consisting of thousands of files with millions of lines of code. We have demonstrated significant performance improvements using IPO with reasonable compilation times. The IPO compilation model presented in this paper is significantly more usable than other approaches described in the past.

7. Acknowledgements

This work is the result of a strong team effort within the HP compiler groups. We thank Suneel Jain for his support and contributions and the anonymous reviewers for their invaluable feedback.

8. References

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[2] A. Ayers, R. Schooler, and R. Gottlieb, "Aggressive Inlining", *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997, pp. 134-145.

[3] A. Ayers, S. de Jong, J. Peyton, and R. Shooler, "Scalable Cross-Module Optimization Framework", *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, June 1998, pp. 301-312.

[4] P. Bothner, "A Compiler Server for GCC", *Proceedings of the First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[5] M. Burke and L. Torczon, "Interprocedural Optimization: Eliminating Unnecessary Recompilation", *ACM Transactions on Programming Languages and Systems*, 15(3), July 1993, pp. 367-399.

[6] D.R. Chakrabrati, L.A. Lozano, X.D. Li, and S. Liu, "Dynamic, Context-Sensitive Cross-File Inline Ordering", *Submitted for publication*.

[7] K.D. Cooper, M.W. Hall, and L. Torczon, "An Experiment with Inline Substitution", *Software - Practice and Experience*, 21(6), June 1991, pp. 581-601.

[8] Free Software Foundation, "GNU Make", http://www.gnu.org/software/make.

[9] G. Gao, J.N. Amaral, J.C. Dehnert, and R.A. Towle, "Tutorial on The SGI Pro64 Compiler Infrastructure", *Tutorial presented at the 9th International Conference on Parallel Architectures and Compilation Techniques*, Philadelphia, Pennsylvania. October 2000.

[10] M.W. Hall, "Managing Interprocedural Optimization", *PhD Dissertation*, Rice University, April 1991.

[11] D.R. Hanson, "Fast Allocation and Deallocation of Memory based on Object Lifetimes", *Software – Practice and Experience*, 20(1), January 1990, pp. 5-12.

[12] Hewlett-Packard Company, "ELF Object File Format", http://devrsrc1.external.hp.com/STKT/partner/elf-64-hp.pdf.

[13] IBM, "IBM Rational ClearCase Command Reference", ftp://ftp.software.ibm.com/software/rational/docs/v2002/cc/ccase_all/ccref/clearmake.html.

[14] Intel Cooperation, "Intel® Itanium® Architecture Software Developer's Manual".

[15] R. Ju, S. Chan, F. Chow, X. Feng, and W. Chen, "Open Research Compiler (ORC): Beyond Version 1.0", *Tutorial presented at the 11th International Conference on Parallel Architecture and Compilation Techniques*, Charlottesville, Virginia, September, 2002.

[16] C. Lattner and V. Adve, "Architecture for a Next-Generation GCC", *Proceedings of the First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[17] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization*, Palo Alto, California, March 2004.

[18] A. Srivastava and D.W. Wall, "A Practical System for Intermodule Code Optimization at Link-Time", *Journal of Programming Language*, 1(1), December 1992, pp. 1-18.