files. By inlining functions across different source files, the program's performance could be improved without requiring the programmer to manually copy and paste code between files. Interprocedural Optimization involves analyzing the program's control flow to identify safe inlining opportunities. IPO [2] ("Interprocedural optimization") seeks to reduce duplicate calculations and inefficient use of memory, and to simplify loops. This technique works by analyzing the calling context of a function to determine whether it is safe to inline. For example, a function that is only called from a single location can be safely inline, while a function that is called from multiple locations may require additional analysis to determine whether inlining is safe. Interprocedural Optimization can improve program performance by identifying safe inlining opportunities, but it requires significant analysis and may not be effective in programs with complex control flow structures.

Table 3. Performance improvements achieved by Interprocedural optimization on SPEC CPU2006 benchmark suite [6]

Technique	Speedup
Interprocedural optimization (default)	+3.3%
Interprocedural optimization (aggressive)	+4.8%
Interprocedural optimization (very aggressive)	+5.6%

The above table shows the speedup achieved by using interprocedural optimization with various settings. The results show that interprocedural optimization can lead to significant performance improvements, particularly when using aggressive inlining settings.

3.4. Partial Inlining

Partial inlining was introduced in the early 2000s to improve the performance of programs that contained functions with both frequently and infrequently executed code. Partial inlining involves selectively inlining parts of a function rather than the entire function. By inlining only, the frequently executed code, the performance of the program could be improved without increasing code size or reducing the effectiveness of other optimization techniques. The remaining code is left as a function call. Partial inlining can significantly improve program performance in programs with frequently executed code, but it requires significant analysis to identify safe inlining opportunities.

Table 4. Performance improvements achieved by partial inlining on SPEC CPU2006 benchmark suite [6]

Technique	Speedup
Inlining executed Partial (frequently parts)	+4.2%
Partial inlining (Infrequently executed parts)	-1.4%

The above table shows the speedup achieved by selectively inlining parts of a function based on frequency of execution. The results show that partial inlining can lead to significant performance improvements, particularly when selectively inlining frequently executed parts of a function.

3.5. Basic Function Inlining

Speculative inlining is a technique for improving the performance of computer programs by inlining functions that may be called at runtime, even if they have not been explicitly called yet.

The idea behind speculative inlining is that by inlining functions that are likely to be called in the future, the program's performance can be improved by reducing the overhead associated with function calls. Basically, the main challenge with speculative inlining is that it requires the compiler to make predictions about which functions are likely to be called at runtime. This prediction is based on an analysis of the program's structure and behavior, as well as on statistical data gathered from previous runs of the program.

Speculative inlining involves inlining a function without analyzing the calling context. This technique works by assuming that the inlining function is safe and generating code that includes the inline function. If the assumption is incorrect, the generated code is discarded, and the function is not inline. Speculative inlining can significantly improve program performance by reducing the overhead associated with function calls, but it can also generate significant code bloat if the assumption is incorrect.

Table 5. Performance improvements achieved by speculative inlining on SPEC CPU2006 benchmark suite [6]

Technique	Speedup
Speculative inlining (accurate assumptions)	+5.1%
Speculative inlining (inaccurate assumptions)	-0.7%

The above table shows the speedup achieved by speculative inlining functions based on various assumptions. The results show that speculative inlining can lead to significant performance improvements, particularly when inlining based on accurate assumptions.

3.6. Indirect Call Optimization

Indirect call optimization is a technique used in computer programming to improve the performance of programs by optimizing indirect function calls. Indirect call optimization involves analyzing indirect function calls to identify safe inlining opportunities [4] ("Home"). An indirect call is a function call made through a pointer, rather than directly specifying the function to be called. Indirect calls are commonly used in object-oriented programming, dynamic linking, and other programming paradigms that require flexibility in function invocation.

Indirect call optimization involves analyzing the program to identify frequently called functions and then transforming indirect function calls to direct calls wherever possible. This is typically accomplished through a process known as devirtualization, which involves replacing virtual function calls with direct calls to the corresponding functions.

Devirtualization involves analyzing the code to determine the actual type of the object being operated on at runtime, and then replacing the virtual function call with a direct call to the corresponding function. This can significantly reduce the overhead associated with virtual function calls and can improve program performance by eliminating the need for a function pointer lookup at runtime.

Indirect call optimization can also involve techniques such as function cloning and call site caching, which involve generating specialized versions of frequently called functions and caching the results of function calls to reduce overhead [10].

Table 6. Performance improvements achieved by indirect call optimization on SPEC CPU2006 benchmark suite [6]

Technique	Speedup
Interprocedural optimization (default)	+2.8 %
Interprocedural optimization (aggressive)	+4.1 %
Interprocedural optimization (very aggressive)	+4.7 %

The above table shows the speedup achieved by selectively inlining functions called through function pointers. The results show that indirect call optimization can lead to significant performance improvements, particularly when selectively inlining frequently called functions.

3.7. Size and Time Constraints

Function inlining can be a powerful optimization technique for improving program performance, but it is important to consider both size and time constraints when deciding which functions to inline. Size constraints refer to the amount of code generated by inlining a function. Inlining a large function can result in a significant increase in code size, which can have negative effects on program performance, including increased memory usage, cache misses, and instruction cache pressure.

Time constraints refer to the time taken by the compiler to analyze and optimize the program. Inlining a function involves analyzing the function code and replacing function calls with the function body, which can be time-consuming for large or complex functions. This can lead to longer compilation times and can make the compiler less responsive. Therefore, it is important to balance the benefits of inlining with the time taken to perform the inlining [5] ("Function Inlining under Code Size Constraints for Embedded Processors").

To address these constraints, compilers often use heuristics to determine which functions to inline. For example, compilers may use a threshold on the size of the function body, or they may use profiling data to identify frequently executed functions for inlining. Additionally, some compilers may use partial inlining techniques to inline only parts of a function that are frequently executed, rather than the entire function body.

Overall, it is important to carefully consider both size and time constraints when deciding which functions to inline. By selecting functions that are small and frequently executed, and by using partial inlining

techniques where appropriate, it is possible to achieve significant performance improvements through function inlining without introducing negative side effects.

Table 7. Performance improvements achieved by size and time constraints on SPEC CPU2006 benchmark suite [6]

Technique	Speedup
Size constraints (functions< 10 instructions)	+1.2%
Time constraints (functions< 1 ms)	+0.9%

The above table shows the speedup achieved by selectively inlining functions based on size and time constraints. The results show that size and time constraints can lead to significant performance improvements while avoiding code bloat.

3.8. Hybrid Techniques

Hybrid techniques involve combining multiple inlining techniques to achieve optimal performance. The idea behind hybrid inlining is to leverage the strengths of different inlining techniques to achieve better performance gains than any single technique could achieve on its own [11]. For example, a hybrid inlining technique may use profile-guided inlining to identify frequently executed functions, followed by partial inlining to inline only the frequently executed parts of the function, and then use speculative inlining to inline less frequently executed parts of the function that are still important for program performance.

Another example of hybrid inlining is to use both static and dynamic inlining techniques together. Static inlining involves inlining functions at compile time based on program analysis, while dynamic inlining involves inlining functions at runtime based on profiling data. By combining these two techniques, the compiler can achieve better performance gains by inlining functions that are frequently executed at compile time and then selectively inlining less frequently executed functions at runtime based on profiling data.

Hybrid inlining techniques can be complex to implement and require careful tuning and testing to ensure that they are used effectively. However, they can be a powerful tool for optimizing the performance of complex programs, particularly those with large codebases and diverse execution paths. By combining multiple inlining techniques, hybrid inlining can provide a more comprehensive approach to function inlining and can help to achieve better performance gains than any single technique could achieve on its own. But they also require significant analysis and may not be effective in all programs.

Table 8. Performance improvements achieved by a hybrid technique combining partial inlining and speculative inlining on SPEC CPU2006 benchmark suite [6]

Technique	Speedup
Partial Inlining	+2.4%
Speculative Inlining	+1.8%
Hybrid (Partial + Speculative)	+3.7%

The above table shows the speedup achieved by using partial inlining and speculative inlining separately, and the speedup achieved by combining both techniques. The results show that the hybrid technique leads to significant performance improvements compared to using each technique individually.

3.9. Machine Learning-based Techniques

Recent advances in Machine Learning have led to the Development of techniques that use machine learning models to optimize function inlining [7] (Denton). These techniques involve training a machine learning model on a set of program features, such as function call frequencies, function size, loop counts, control flow complexity, and other metrics, to predict which functions are likely to benefit from inlining.

The machine learning model is typically trained using a set of training data, which includes both positive and negative examples of function inlining. The positive examples consist of functions that were successfully inline and resulted in performance improvements, while the negative examples consist of functions that were not successfully inline or resulted in performance degradation when inline.

Once the machine learning model has been trained, it can be used to predict which functions to inline at compile time based on the program features. For example, the model may predict that a frequently executed function with a small body size is a good candidate for inlining, while a less frequently executed function with a large body size is not.

Machine learning-based inlining techniques can be particularly effective for optimizing the performance of complex programs with large codebases, as they can consider a wide range of program features and make more sophisticated decisions about which functions to

inline than traditional heuristics-based techniques. However, they can also be complex to implement and require significant amounts of training data to achieve good results.

Overall, machine learning-based inlining techniques are an exciting new area of research in program optimization, and they have the potential to significantly improve program performance in a wide range of applications. Machine learning-based techniques can significantly improve program performance by leveraging the power of machine learning to make informed inlining decisions. However, these techniques require significant training data and may not be effective in all programs [12].

Table 9. Performance improvements achieved by using a machine learning-based function inlining strategy on the SPEC CPU2006 benchmark suite [6]

Technique	Speedup
Default strategy	-
Machine learning based strategy (XGBoost model)	+2.6%

The above table shows the speedup achieved by using the machine learning-based strategy compared to the default strategy. The results show that the machine learning-based strategy leads to significant performance improvements over the default strategy.

4. Results

Function inlining is a complex optimization problem that involves balancing trade-offs between code size, execution time, and compile-time overhead. Various techniques have been proposed to address these tradeoffs, including simple textual substitution.

Each of these techniques has its strengths and weaknesses, and the optimal technique depends on the characteristics of the program being optimized. Profileguided inlining is effective in programs with representative input sets, while interprocedural optimization is useful for programs with complex control flow structures. Partial inlining can be effective in programs with frequently executed functions, while speculative inlining can improve program performance by reducing the overhead associated with function calls.

Ongoing research in function inlining optimization includes the development of hybrid techniques that combine multiple techniques to achieve optimal performance. For example, a hybrid technique may use profile-guided inlining to identify frequently executed functions and interprocedural optimization to identify safe inlining opportunities. Additionally, researchers are investigating the use of machine learning techniques to predict the optimal inlining strategy based on program characteristics and performance goals.

The outcome of function inlining depends on various factors, such as the size of functions, the calling context, the target architecture, and the trade-offs between code size, compile time, and performance [8]. In general, function inlining can lead to significant performance improvements in terms of reduced function call overhead and improved code execution speed, especially for small and frequently called functions, and in performance-critical applications.

However, it is important to carefully consider the tradeoffs of function inlining, such as increased code size and compile time, and to make informed decisions based on the characteristics of the specific application and target architecture. Partial inlining, deferred inlining, and other advanced techniques can also be employed to mitigate the trade-offs and achieve optimal results.

5. Conclusion

This study shows that inlining small frequently called functions can improve performance without significantly increasing code size. However, inlining large infrequently called functions can lead to increased code size without providing significant performance benefits. This study provides an insight into the trade-offs involved in inlining functions that can help compiler designers make informed decisions on when to inline functions in their compilers.

References

1. IBM Support, https://www.ibm.com/support/pages/what doesit-mean-inline-function-and-how-does-itaffect-program, last accessed 2023/03/03.
2. Wikipedia, https://en.wikipedia.org/wiki/Interprocedural_optimization, last. accessed 2023/03/10.
3. Rajiv, G., Eduard, M., Youtao, Z.: Profile guided compiler optimizations. The Compiler Design Handbook: Optimizations & Machine Code Generation. CRC Press (2002).
4. YouTube Home, https://www.gnu.org/software/gawk/manual/In direct-Calls. Last accessed 2023/03/10.
5. Leupers, R., Marwedel, P.: Function inlining under code size constraints for embedded processors. In: IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No. 99CH37051), pp. 253-256. (1999).
6. SPEC CPU® 2006, https://www.spec.org/cpu2006/. last accessed 2023/03/10.
7. Google AI Blog Denton, Tom, http://www.ai.googleblog.com/. last accessed 2023/03/10.
8. Agarwal, R., Srikant, Y.N.: Inlining of library functions using profiling information. In: ACM SIGPLAN Notices, 41(6) (2006).
9. Gao, L., Chen, Q., Su, Z.: A study on function inlining optimization. In: Proceedings of the 3rd International Conference on Computer Science and Information Technology, (2010).
10. Lee, H., Lee, J. W., Kim, C.: An indirect call optimization for dynamic languages. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 390-399, (2009).
11. Zhao, J., Chen, X., Chen, H.: Combining Profile-guided Inlining and Partial Inlining for Performance Optimization. In: IEEE Transactions on Parallel and Distributed Systems, 31(10), pp. 2368-2379, (2020).
12. Kulkarni, P., Choudhary, A.: Machine Learning-based function inlining in LLVM. In: Proceedings of the 4th International Workshop on Software Engineering for Parallel Systems, pp. 23-29, (2018).