Parallel Binary Code Analysis

Xiaozhu Meng
Department of Computer Science
Rice University
Houston, TX, USA
Xiaozhu.Meng@rice.edu

Jonathon M. Anderson
Department of Computer Science
Rice University
Houston, TX, USA
jma14@rice.edu

John Mellor-Crummey
Department of Computer Science
Rice University
Houston, TX, USA
johnmc@rice.edu

Mark W. Krentel
Department of Computer Science
Rice University
Houston, TX, USA
krentel@rice.edu

Barton P. Miller
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI, USA
bart@cs.wisc.edu

Srđan Milaković
Department of Computer Science
Rice University
Houston, TX, USA
sm108@rice.edu

Abstract

Binary code analysis is widely used to help assess a program's correctness, performance, and provenance. Binary analysis applications often construct control flow graphs, analyze data flow, and use debugging information to understand how machine code relates to source lines, inlined functions, and data types. To date, binary analysis has been single-threaded, which is too slow for convenient use in performance tuning workflows where it is used to help attribute performance to complex applications with large binaries.

This paper describes our design and implementation for accelerating the task of constructing control flow graphs (CFGs) from binaries by using multithreading. Prior research focuses on algorithms for analysis of challenging code constructs encountered while constructing CFGs, including functions sharing code, jump tables, non-returning functions, and tail calls. These algorithms are described from a program analysis perspective and are not suitable for direct parallel implementation. We abstract the task of constructing CFGs as repeated applications of several core CFG operations that include creating functions, basic blocks, and edges. We then derive CFG operation dependency, commutativity, and monotonicity. These operation properties guide our design of a new parallel analysis for constructing CFGs. Using 64 threads, we achieved as much as 25× speedup for constructing CFGs and 8× for a performance analysis tool that leverages our new analysis to recover program structure.

PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8294-6/21/02...$15.00
https://doi.org/10.1145/3437801.3441604

CCS Concepts: • **Software and its engineering** → *Software performance*; **Automated static analysis**; • **Theory of computation** → Program analysis.

Keywords: control flow graph construction, performance analysis, operation dependencies and properties

ACM Reference Format:
Xiaozhu Meng, Jonathon M. Anderson, John Mellor-Crummey, Mark W. Krentel, Barton P. Miller, and Srđan Milaković. 2021. Parallel Binary Code Analysis. In *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21), February 27-March 3, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3437801.3441604

1 Introduction

Binary code analysis is a foundational technique for a variety of applications, including performance analysis [2, 8, 22], software correctness [3, 11], software security [14, 31, 32], and software forensics [21, 26]. Important binary code analysis capabilities include constructing control flow graphs (CFGs), analyzing control flow and data flow, and extracting source line mappings and data types from debugging information, when it is available. Traditionally, binary analysis applications are single-threaded. However, use of binary analysis as part of iterative workflows would substantially benefit from accelerating its performance.

Increasingly, compilation of sophisticated applications and frameworks yields multi-gigabyte binaries. We have witnessed this trend in both multi-physics applications developed by national laboratories as well as popular machine learning frameworks such as TensorFlow [1]. The developers of such large software systems use the following performance analysis workflow to optimize their codes: (1) compile the source code to generate the binary program, (2) measure the performance of the binary during execution, (3) attribute measurements to the corresponding source code constructs using binary analysis, and (4) optimize the source guided by performance metrics. These four steps are repeated until developers are satisfied with their software's performance.

In this performance analysis cycle, using binary analysis to construct CFGs to understand loop nests in optimized code and report performance at the loop nest level is extremely useful for scientific codes [29]. However, binary analysis must be repeated after any source code change because even small code changes can lead to dramatically different binaries, especially with C++ template instantiation and aggressive compiler optimizations. Single-threaded binary analysis takes too long to analyze large binaries: it takes more than 20 minutes to analyze a 7.7GiB shared library from TensorFlow. Such slow analysis would interfere with the workflow of developers tuning code for production. This observation holds even for long-running, iterative scientific computations as they can be tuned based on analysis of measurements of a few iterations.

To address speed requirements imposed by analyzing large binaries, we applied data parallelism to several binary analysis tasks needed for performance analysis, including parsing DWARF debugging information, identifying loops, and iterating over functions. However, we quickly encountered a challenging bottleneck: constructing Control Flow Graphs (CFGs) for large binaries, which involves identifying functions, constructing basic blocks,and connecting edges between basic blocks. We observed that serial CFG construction can take over 80% of the total runtime.

In this paper, we present our design and implementation of parallel CFG construction using function level parallelism. We observe that existing serial algorithms for CFG construction are described from a program analysis perspective, focusing on understanding complex machine code generated by compilers [10, 20, 28], including non-returning functions, tail calls and jump tables. They do not explicitly identify data or analysis dependencies, which are crucial concepts for parallel algorithms. This gap leads to two major challenges.

First, because binary functions may share code, threads analyzing different functions may end up concurrently analyzing shared code and require synchronization. Without a clear specification of analysis dependencies and interaction, adding synchronization in an ad-hoc fashion is likely to cause either incorrect algorithms (missing synchronization) or poor performance (adding unnecessary synchronization).

Second, in contrast to serial CFG construction algorithms, which assume a static CFG before and while performing a specific analysis, a parallel algorithm for CFG construction needs to consider concurrent graph changes by other threads. We identify flaws of existing serial analysis for jump tables and tail call identification, where the final results of these analyses may change if they are applied in a different order.

To address these two challenges, we abstract CFG construction as repeated applications of primitive CFG construction operations. These operations include creation of functions, basic blocks and edges, modification of basic block ranges, and removing blocks and edges. We derive operation properties, including operation dependencies, commutativity, and monotonicity, and use this theoretical framework to reason about the correctness and performance of CFG construction algorithms. Based on this abstraction, we design a new parallel algorithm for CFG construction, which expresses parallelism as commutative operations, addresses correctness issues in existing serial algorithms, and improves performance by addressing operation dependencies.

We implemented our new parallel CFG construction in the Dyninst binary analysis and instrumentation toolkit [23], and evaluated the performance characteristics of our parallel binary analysis with a number of large binaries. We achieve as much as 25× speedup when constructing control flow graphs using 64 hardware threads. We then showcase the benefits of parallel binary analysis with `hpcstruct`, a utility in HPCToolkit [2], which relates performance measurements back to source code; we achieved 8× speedup for `hpcstruct`.

In summary, this work makes the following contributions:

1. An abstraction of CFG construction that is suitable for parallel implementation.
2. A new algorithm for parallel CFG construction that is derived from the properties of CFG operations.
3. An implementation of the new algorithm in Dyninst that can be used by other binary analysis application developers.
4. Demonstrating the effectiveness of our parallel analysis with `hpcstruct`, which significantly accelerates program structure recovery for performance analysis.

2 Related Work

There is rich literature about constructing CFGs from binaries [4, 10, 15, 16, 27]. A commonly used approach is control flow traversal [27, 30]. Starting from known function entry points such as the ones found in the symbol table, it follows control flow transfers in the program to discover code and identify additional function entry points for further analysis. We discuss several challenging code constructs that must be addressed during control flow traversal and representative binary analysis tools that implement control flow traversal.

2.1 Challenging Code Constructs

Functions sharing code: A common compiler optimization is to share binary code between functions with common functionality, such as error handling code and stack tear-down code. We have observed such code sharing in several versions of glibc and code compiled by the Intel Compiler Suite (ICC). In addition, binary analysis tools typically represent a function with multiple entry points as multiple single-entry functions that share code. Thus, Fortran functions with multiple entry points specified with the `entry` keyword, and binaries on Power 8 or newer (the ABI specifies that each function has at least two entry points) lead to functions sharing code. This issue can be addressed by defining a function as the basic blocks that are reachable from the function entry by traversing only intra-procedural edges [5, 20].

The actual fraction of shared basic blocks varies significantly. On Power 8 or newer, almost all of the basic blocks are shared due to its new ABI specification. On x86-64, 86% of the binaries in `/lib64` on a Redhat 7.9 system have no shared blocks; however, we have seen sharing as high as 70% in `/lib64/libisc.so`. The TensorFlow binary studied in Section 6 has 3.5% shared blocks, including blocks shared by 186 functions. Correctness requires special handling for shared blocks, which is an integral part of our strategy.

Non-returning functions: Binary analysis tools often define a call fall-through edge, which is a summary edge representing that the control flow at a function call will return to the call site. However, a function call to a non-returning function will never return to its call site, so there should be no call fall-through edge at such call sites. A wrongly created call fall-through edge can lead to confusing control flow and cascading impacts on binary analysis applications. A general approach for identifying non-returning functions is to match function names against known non-returning functions such as `exit` and `abort` and use an iterative analysis to identify functions that always end in calls to non-returning functions. One example is a fixed point analysis, which defers the parsing of a call fall-through edge until the callee's return status is determined and marks every function in a cyclic dependency as non-returning [20].

Jump tables: Compilers often emit indirect jumps for switch statements. The targets of these indirect jumps are calculated based on jump table data in the binary. It is critical for complete control flow traversal to resolve jump targets calculated through jump tables. A common approach is to use backward slicing to identify the instructions that are involved in the target calculation and construct a symbolic expression of the jump target to identify actual jump targets [10, 20, 28, 33].

Tail calls: A tail call [7] is a compiler optimization that uses a jump instruction at the end of a function to target the entry point of another function, thus not every branch should be labeled as intra-procedural. Tail calls are often recognized through heuristics [10, 20], including (1) a branch to a known function entry is a tail call; (2) a branch to a basic block that is reachable through only intra-procedural edges of the current function is not a tail call; (3) if there is stack frame tear down before a branch, it is a tail call.

2.2 Binary Analysis Tools

Recent binary analysis tools address these challenging code constructs in a similar way, including angr [28], Dyninst [20], and rev.ng [10]. However, their software infrastructures have distinct characteristics regarding analysis speed.

Both angr and rev.ng first lift machine instructions to an Intermediate Representation (IR) and then perform analysis on the resulting IR. angr uses Valgrind's VEX IR and rev.ng uses QEMU to lift a binary to LLVM IR. While this approach has the advantages of not being architecture specific and facilitating the development of complex data flow analysis such as points-to analysis and value set analysis, it leads to a significant performance slowdown for two reasons. First, the lifting process itself is slow. Second, the number of assignments in the IR is significantly larger than the number of machine instructions as one instruction may be lifted to multiple IR assignments, especially on CISC architectures such as x86-64.

In contrast, Dyninst directly operates with the binary. Dyninst's instructionAPI provides an architecture independent interface for querying instruction opcodes, instruction operands, registers, and memory addressing modes. The CFG construction code inside Dyninst works with this "bare-metal" instruction interface. The only exception is that when Dyninst resolves jump tables, Dyninst lifts machine instructions to ROSE IR [24]. However, since lifting is applied to instructions that are involved in the jump table calculation found by backward slicing, typically only a small portion of the binary is lifted.

3 Notation

To provide a foundation for discussing parallel CFG Construction, we first present an abstraction of control flow graphs and a series of core operations on them. We build upon the abstraction designed for binary modification [5], which works with fully constructed CFGs, and extend it to abstract the process of constructing CFGs.

Definitions: We define a CFG $G = \langle B, C, E, F \rangle$ to be a tuple of the following:

- B is a set of address ranges $[s, e)$, representing basic blocks within the binary. Each of these contains at most one control flow instruction, which if present is the final instruction within the range, and has incoming control flow at only address s.
- C is a set of candidate blocks $[t]$, representing addresses which are known to start basic blocks but do not have known ending addresses yet.
- $E \subseteq \{(a \longrightarrow b) : a \in B, b \in B \cup C\}$ is a set of directed edges between basic blocks, representing possible control flow executions between blocks.
- $F \subseteq B \cup C$ is the set of function entry blocks.

Partial order: We define a partial order between CFGs, such that a larger graph includes more control flow elements. We define $G_1 \preccurlyeq G_2$ if all of the following are true:

- The address ranges contained in G_1 are contained by G_2. Formally, let A_1 and A_2 be the addresses contained by the blocks in B_1 and B_2 respectively. Then we require $A_1 \subseteq A_2$.
- The explicit control flow in G_1 is also in G_2, regardless of adjustments to block ranges. Formally, for every edge $([s_a, e_a) \longrightarrow [s_b, e_b))$ or $([s_a, e_a) \longrightarrow [s_b])$ in E_1,

E_2 must contain one of the edges $([s'_a, e_a) \rightarrow [s_b, e'_b))$ or $([s'_a, e_a] \rightarrow [s_b])$. Intuitively, G_2 may contain additional edges that target addresses inside a or b, causing them to be split, but the end address of the source block e_a and the start address of the target block s_b are preserved under the partial order.

- The implicit control flow through a basic block in G_1 is preserved in G_2. Formally, for every block $b = [s_0, e) \in B_1$ there is a sequence of blocks $[s_0, s_1), \ldots, [s_n, e) \in B_2$ such that $([s_i, s_{i+1}) \rightarrow [s_{i+1}, s_{i+2})) \in E_2$ for $i = 0, \ldots, n-2$. This means that a block b in G_1 can be split into multiple smaller blocks in G_2 to incorporate other incoming control flow.
- Function entry labels in G_1 are preserved in G_2, regardless of range adjustments. Formally, for every block $[s, e)$ or $[s] \in F_1$, there is a block starting at the same address, i.e., $[s, e')$ or $[s] \in F_2$.

CFG operations: To construct a CFG based on an underlying binary, we define several core operations:

Block End Resolution (BER): Given G containing a candidate block $[t] \in C$, we define $O_{BER}(G, [t])$ as G with the candidate block $[t]$ replaced by a basic block starting at t with a determined end address. There are three possible cases:

- *Block splitting.* If there is an existing block $b = [s, e)$ in B such that $s < t < e$, then we have to split b into the basic blocks $[s, t)$ and $[t, e)$. Any incoming edges on b are redirected to $[s, t)$, while outgoing edges on b and incoming edges on $[t]$ are moved to $[t, e)$.
- *Early block ending.* If there is an existing block $b = [s, e)$ in B such that $t < s$ and the range $[t, s)$ contains no control flow instructions, we replace $[t]$ with $[t, s)$ as before and add the edge $([t, s) \rightarrow [s, e))$.
- *Linear parsing.* If neither of the previous cases apply, let e be the address directly after the first control flow instruction following t. We replace $[t]$ with $[t, e)$ as in the first case.

Direct Edge Creation (DEC): Given a block a in a graph G ending in a direct control flow instruction, we define $O_{DEC}(G, a)$ as G with outgoing edges appended to a, based on the control flow instruction within a (if one exists). There are three cases:

- If a terminates with an unconditional jump to address t, we add the edge $(a \rightarrow [t])$.
- If $a = [s, e)$ terminates with a conditional jump to address t, we add edges for the cases where the condition is true $(a \rightarrow [t])$ and false $(a \rightarrow [e])$.
- If a terminates with a function call instruction to address t, we add the edge $(a \rightarrow [t])$.

Call Fall-Through Edge Creation (CFEC): Given an edge $e = ([s, e) \rightarrow f)$ in a graph G where $[s, e)$ contains a function call instruction and $f \in F$, we define $O_{CFEC}(G, e)$ as G potentially with the additional edge $([s, e) \rightarrow [e])$ summarizing the execution of the callee function. Correct application of this operation depends on non-returning function analysis to identify whether the target function can return or not.

Indirect Edge Creation (IEC): Given a block a in a graph G which contains a jump to a dynamic address, we define $O_{IEC}(G, a)$ as G with the additional edges $(a \rightarrow [t_1]), \ldots, (a \rightarrow [t_n])$, where $t_1, \ldots, t_n$ are target addresses determined statically by inspecting potential code paths leading to the final instruction of a. If this analysis is unable to determine a constant target for a code path (e.g. indirect call through a function pointer), no edges are added and the inspection of other possible paths continues unmodified.

Function Entry Identification (FEI): Given an edge $e = (a \rightarrow b)$ in a graph G, we define $O_{FEI}(G, e)$ as G with the block b labeled as a function entry. This operation is trivial if e was created by an explicit call instruction, but further heuristics are required to identify functions that are reached only through optimized tail calls.

Edge Removal (ER): Given an edge $e = (a \rightarrow b)$ within a graph G, we define $O_{ER}(G, e)$ as G with the edge e removed along with any blocks and edges that are no longer reachable from any function entry point. Formally, let $B' \subseteq B$, $C' \subseteq C$, and $E' \subseteq E$ be the sets of blocks, candidate blocks and edges in G reachable from any block in F without traversing e. We can then define: $O_{ER}(G, e) = \langle B', C', E', F \rangle$. An edge removal operation may lead to removing multiple blocks and edges from the graph.

Starting with the initial graph $G_0 = \langle \emptyset, F_0, \emptyset, F_0 \rangle$, where F_0 is the set of candidate function entry blocks discovered via the binary's symbol table and unwind information, the task of CFG construction can be abstracted as repeated application of these operations. We denote $G_1, G_2, \cdots, G_{n-1}$ as the intermediate results and G_n as the final CFG.

4 CFG Operation Properties

We present several important properties of the defined operations, assess existing serial algorithms with these properties, and use these properties to identify critical correctness and performance issues for parallel CFG construction.

4.1 Properties

Operation dependencies: To correctly build the CFG, operations should be applied with an order that satisfies the dependencies among them. We identify two types of dependencies:

Applicability Dependency. We cannot apply operations to a graph element that has not been discovered. For example, we must create an edge before we can resolve the target block candidate of this edge.

Non-returning Function Dependency. The correctness of O_{CFEC} for creating call fall-through edges depends on the operations applied to the callee functions to determine whether the callee can return or not. If O_{CFEC} is applied when the callee cannot return, an erroneous call fall-through edge would be added.

Operations that satisfy either type of the above dependency must be applied in order. We classify operations that are not constrained by any dependency into three categories:

Commutative operations: Operations O_{BER} and O_{DEC} commute with themselves and with each other, allowing us to choose an order convenient for processing. To establish this, we discuss the following three cases:

(1) Given two candidate blocks $[a]$ and $[b]$ where $a < b$, $O_{BER}(O_{BER}(G, [a]), [b]) = O_{BER}(O_{BER}(G, [b]), [a])$. First, if there is a control flow instruction ending at address c where $a < c < b$, candidate block $[a]$ will end before c while candidate block $[b]$ will end after c. These two operations will act on non-overlapping address ranges and be independent, which gives us commutativity. Second, if a control flow instruction ends at c where $a < b < c$ and c is first control flow instruction following b, we have

$$\begin{aligned} & O_{BER}(O_{BER}(G, [a]), [b]) \\ &= O_{BER}(G \cup \{[a, c)\}, [b]) && \text{(Linear parsing)} \\ &= G \cup \{[a, b), [b, c)\} && \text{(Block splitting)} \\ &= O_{BER}(G \cup \{[b, c)\}, [a]) && \text{(Early block ending)} \\ &= O_{BER}(O_{BER}(G, [b]), [a]). && \text{(Linear parsing)} \end{aligned}$$

Thus we also have commutativity in this case.

(2) Given two blocks a and b, $O_{DEC}(O_{DEC}(G, a), b) = O_{DEC}(O_{DEC}(G, b), a)$. This is because $O_{DEC}(G, a)$ only considers the terminating control flow instructions within the block a.

(3) Given a candidate block $[t]$ and a block $[s, e)$, we have $O_{BER}(O_{DEC}(G, [s, e)), [t]) = O_{DEC}(O_{BER}(G, [t]), [s, e))$. We observe that $O_{DEC}(G, [s, e))$ depends on only the terminating control flow instruction ending at e and will generate only new candidate blocks while $O_{BER}(G, [t])$ does not depend on candidate blocks. Therefore these two operations are independent and thus commutative.

The operation O_{ER} also commutes with itself, allowing us to choose an order convenient for processing. The graph $O_{ER}(O_{ER}(G, e_1), e_2) = O_{ER}(O_{ER}(G, e_2), e_1)$ will contain no blocks reachable only through e_1 and e_2, which gives us the commutativity property.

Monotonic ordering property: While O_{IEC} does not commute trivially with any other operation, we can still establish a weaker property. Let $O_{IEC}(G, a)$ be an indirect edge creation operation and $O_x(G, b)$ be an O_{BER} or O_{DEC} operation. If $O_x(G, b)$ contains an additional edge allowing b to reach a, a subsequent $O_{IEC}(G, a)$ operation may add more edges than one applied prior due to the additional possible code paths. Since O_{IEC} considers code paths separately, it will never elide edges due to additional control flow, thus we have the property $O_x(O_{IEC}(G, a), b) \preccurlyeq O_{IEC}(O_x(G, b), a)$. As our goal is to achieve a maximal CFG, this allows us to reorder O_{IEC} after any O_{BER} and O_{DEC} operations without decreasing the final result.

```
A:              B:
...             ...
leaveq          mov %rsi, 1
jmp 0x400       jmp 0x400
```

Listing 1. An example that leads to inconsistent results with the tail call heuristics used by Dyninst.

Non-reorderable operations: Unlike the others, the operations O_{CFEC} and O_{FEI} do not always commute, nor do they satisfy the ordering property above in all cases. Both of these operations use implementation-specific analyses: non-returning function analysis for O_{CFEC} and tail call identification heuristics for O_{FEI}, both of which at times require inspection of large portions of the graph.

4.2 Serial Algorithm Assessment

We compare the serial algorithms implemented by angr [28], Dyninst [20], and rev.ng [10] using the notation and operations defined above.

First, Dyninst and angr's CFG construction can be characterized with an increasing expression: $G_0 \preccurlyeq G_1 \preccurlyeq G_2 \preccurlyeq \cdots \preccurlyeq G_n$. While this increasing construction strategy does not guarantee best performance, it has the advantage of not performing redundant work of adding and then removing graph elements.

In contrast, rev.ng has an additional step to clean candidate function entries after this increasing phase. This cleaning step can address issues caused by operations that do not commute such as tail call identification. Listing 1 is an example where Dyninst will give inconsistent results depending on analysis order. In this example, functions A and B branch to the same address. If A is analyzed first, because `leaveq` tears down the stack frame, Dyninst will treat the branch in A as a tail call and create a new function at the branch target; later when Dyninst analyzes B, we find that B branches to a known function entry, so the branch in B is also a tail call. In this case, function B will not include the block at 0x400. If B is analyzed first, because there is no stack frame tear down before the branch in B, Dyninst will not treat the branch as a tail call, and the block at 0x400 will be part of B. Therefore, the function boundary of B is determined by the order of analysis. Without other context, it is equally valid to conclude either "A and B both tail call to 0x400" or "A and B share block at 0x400". The cleaning step can ensure a consistent answer.

Second, jump table analysis implemented in tools does not necessarily satisfy the monotonicity property we defined for O_{IEC}. The root cause is imprecise jump table analysis where jump table targets can be over-approximated. Suppose we have $O_{IEC}(G, b_1)$ and $O_{IEC}(G, b_2)$. Due to imprecise jump table analysis, $O_{IEC}(G, b_1)$ generates an over-approximated set of jump targets, resulting in invalid outgoing edges. Such additional but confusing control flow may cause $O_{IEC}(G, b_2)$ to fail, leading to an empty set of targets. However, if $O_{IEC}(G, b_2)$ is performed first, we may get the correct non-empty set of jump targets for b_2. We have observed this problem in Dyninst's jump table analysis. While rev.ng and angr both provide detailed descriptions about how they resolve jump tables, neither is able to guarantee no over-approximation of jump targets.

4.3 Challenges For Parallelism

We identify three issues that must be addressed to achieve effective parallel analysis.

First, commutative operations still need careful synchronization. Suppose we have two blocks $a \neq b$ containing direct control flow to a target address t and consider the case where we perform $O_{DEC}(G, a)$ and $O_{DEC}(G, b)$ concurrently. These two operations commute trivially adding the edges $(a \rightarrow [t])$ and $(b \rightarrow [t])$ respectively, however only the operation performed first will create the candidate block $[t]$, which must be referenced by the second to create its edge. This is trivial to maintain for serial algorithms, but synchronization is necessary to maintain this uniqueness property in a parallel setting.

Second, non-returning function dependencies between operations can lead to ineffective parallelism. In a call chain where F_1 calls F_2, F_2 calls F_3, $\cdots$, and F_{n-1} calls F_n, an O_{CFEC} operation in F_1 may need to wait for operations in F_2 to complete, which may need to wait for operations in F_3, and so on. This effect causes undesirable serialization during analysis.

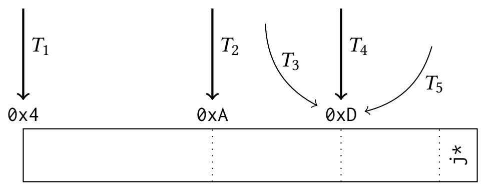
Third, the monotonic ordering property for operation $O_{IEC}(G, a)$ indicates that we might be able to find more control flow targets if it is reordered after other edge creation operations (namely O_{BER} and O_{DEC}). However, deferring $O_{IEC}(G, a)$ can exacerbate the issue of non-returning function dependencies, as this will delay the discovery of returns that are only reachable through the indirect jump.

5 Parallel CFG Construction

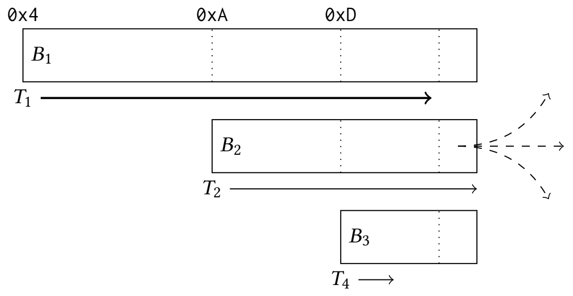
We designed a new parallel algorithm to produce consistent results for tail call identification and jump table target resolution, support efficient concurrent CFG operations, and carefully order CFG operations for better performance. We define five invariants to support concurrent CFG operations (Section 5.1) and divide the algorithm into two stages:

Control flow expanding: We perform control flow traversal for initialized functions in parallel, during which we may discover more functions and repeatedly apply control flow traversal until there are no more functions to analyze (Section 5.2).

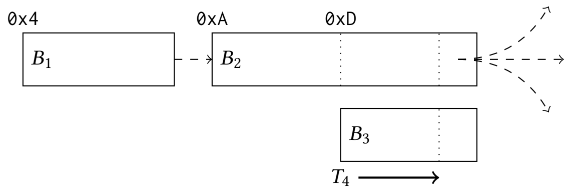
Control flow finalization: This stage includes cleaning control flow edges and blocks created by over-approximated jump tables, cleaning inconsistent tail call identification results, and determining which basic blocks belong to which function by traversing intra-procedural edges from function entry blocks (Section 5.3).



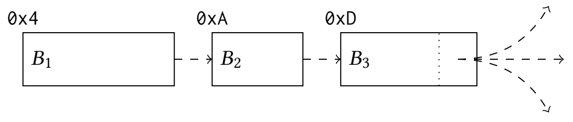
(a) T_1 and T_2 branch to two different addresses. T_3, T_4, and T_5 branch to the same target. j* represents a conditional or unconditional control flow transfer.



(b) T_1, T_2, and T_4 create new basic blocks. T_2 first reaches the block end and creates new control flow edges. T_1 then reaches the block end and needs to split blocks.



(c) T_1 finishes splitting blocks. T_4 reaches the block end and needs to split blocks.



(d) T_4 finished splitting blocks, which involves moving edges.

Figure 1. An example of five threads working with a common area of code. Solid edges represent the progress of threads. Bold solid edges represent actions to take place. Dashed edges represent control flow edges in the CFG.

5.1 Control Flow Traversal Invariants

We use Figure 1 to illustrate how our five invariants ensure that threads correctly perform concurrent CFG operations.

Invariant 1: Block Creation. There is at most one basic block starting at any given address. This invariant means that if threads branch to the same target concurrently, one and only one thread should create the block and make the block visible to other threads. This invariant applies to direct edge creation operations, call fall-through edge creation operations, and indirect edge creation operations.

In Figure 1a, threads T_3, T_4 and T_5 branch to the same address. According to this invariant, only one thread should create a new basic block. As shown in Figure 1b, T_4 creates a new basic block B_3. T_3 and T_5 do not create any new basic blocks and leave the common code area to work with other code. Independently, T_1 creates basic block B_1 and T_2 creates basic block B_2.

Maintaining this invariant requires an efficient concurrent data structure that synchronizes threads branching to the same target, while allowing threads branching to different targets to proceed independently. We use the concurrent hash map provided by Intel's Threaded Building Blocks library [13] to fulfill these two requirements, which provides entry-level reader-writer locks. The `insert` method of `concurrent_hash_map` ensures that only one concurrent insertion with the same key will succeed. Therefore, we can use the return value of `insert` to determine whether the current thread has successfully created a block and should continue analysis of the block. Threads that see a `false` return value know that another thread has created the block and can move on to other work.

Invariant 2: Block End. There is at most one basic block ending at any given address. This invariant applies to block end resolution operations. In Figure 1b, thread T_1, T_2 and T_4 will independently parse their blocks until they reach the indirect jump instruction. Based on this invariant, only one thread should register the block end address, which is T_2 in this example.

A naïve implementaion of this invariant is to let a thread check whether a block exists at its current working address. If there exists one, the working thread can end its block. However, this implementation means that there will be a block start lookup after decoding each instruction. This would create a performance hotspot on the concurrent data structure used for Invariant 1.

We defer this check until the working thread reaches a control flow instruction. In this way, we reduce the frequency of global concurrent data structure lookups from once per instruction to once per control flow instruction. This design causes redundant instruction decoding between overlapping blocks analyzed by different threads. However, while functions sharing code is commonly seen in binaries, most of the code blocks in a binary are still not shared. This means that most of the time, a thread is going to branch into a block that was created by itself, not created by other threads. Therefore, we implemented a thread local cache that maintains addresses that have been analyzed by the thread and use this cache to reduce redundant decoding.

Invariant 3: Edge Creation. The thread that registers a block's end is responsible for creating out-going control flow edges from that block. This invariant applies to all edge creation operations and ensures that no redundant control flow edges are created and jump table analysis for a particular indirect jump is always performed by one thread. This also reduces unnecessary block start lookups by avoiding redundant edges. As shown in Figure 1b, because thread T_2 registers the block end, T_2 continues to perform control flow analysis to resolve the indirect jump targets and create control flow edges. T_2 then leaves the common code and continues to work with other code.

Invariant 4: Block Split. The threads that reach a block end but do not register the block end will need to split blocks. This invariant applies to the block splitting case in block end resolution operations. Suppose we have block $B_1[x_1, y), B_2[x_2, y), \ldots B_n[x_n, y)$ created by n threads, where $x_1 < x_2 < \ldots < x_n < y$. The results of block split should be $B_1[x_1, x_2), B_2[x_2, x_3), \ldots, B_n[x_n, y)$, with a fall-through edge between each pair of adjacent basic blocks. It is inefficient to wait for all relevant blocks before performing splitting, so we present the following eager block split algorithm.

Based on Invariant 2 (block end), only one block $B_i[x_i, y)$ will register its end at y. When some other block $B_j[x_j, y)$ reaches y, the working thread can look up B_i as the registered block. Depending on the relationship between x_i and x_j, we have two cases:

- If $x_i > x_j$, B_j is split into $[x_j, x_i)$ while B_i is untouched. We then register B_j at block end address x_i, which will trigger a new iteration of block split when another block has already registered block end at x_i. As shown in Figure 1c, T_1 splits blocks B_1, registers B_1 ending at $0xA$ and then leaves the common code.
- If $x_i < x_j$, B_i is split into $[x_i, x_j)$ while B_j is untouched. We then replace B_i with B_j for block end address y, register B_i for block end address x_j, and move outgoing edges from B_i to B_j. Similar to the first case, registering B_i at x_j may recursively require another block split. As shown in Figure 1d, T_4 splits B_2 and moves control flow edges from B_2 to B_3.

For both cases, each iteration of the block split algorithm ends with a smaller block end address. Therefore, our block split algorithm is guaranteed to converge.

Listing 2 shows how we implement Invariant 2 (block end), Invariant 3 (edge creation), and Invariant 4 (block split). `concurrent_hash_map` can be instantiated with any mainstream implementation of concurrent hash tables, including Intel TBB's concurrent hash map, and provides the following two properties [19]. First, `concurrent_hash_map` should guarantee uniqueness of keys, which is necessary for Invariant 2. Second, concurrent calls to `concurrent_hash_map`'s `insert` method with the same key should have one and only one call returning `true`; we use this property for Invariant 3 and 4.

Structure `MapEntry` contains a pointer to a `Block` and a `Mutex`. `Block` has several auxiliary methods: `start()` returns its starting address; `end()` returns its ending address; `setEnd()` updates its ending; `appendOutEdge(b2)` creates a new outgoing edges to `b2`. `Mutex` supports `lock` and `unlock` methods for mutual exclusion and serves as an entry level

```
1   struct MapEntry {
2     Block* b;
3     Mutex* m;
4   };
5   concurrent_hash_map<Address, MapEntry> blockEnds;
6   EdgeSet registerBlockEnd(Block* b) {
7     bool isNewEntry = false;
8     MapEntry e;
9     // e references the newly inserted or the existing entry
10    std::tie(isNewEntry, e)
11      = blockEnds.insert(b->end(), b, new Mutex());
12    e.m->lock();
13    if(newEntry) {
14      // Block end registered, create outgoing edges for b
15      EdgeSet edges = createEdgesFor(b);
16      e.m->unlock();
17      return edges;
18    } else {
19      // Block end not registered, split blocks
20      Block* b2 = e.b;
21      if(b->start() < b2->start()) {
22        b->setEnd(b2->start());
23        b->appendOutEdge(b2);
24        e.m->unlock();
25        // b's ending address has changed and re-register
26        registerBlockEnd(b);
27      } else {
28        e.b = b;
29        b2->setEnd(b->start());
30        // Move b2's out edges to b
31        MoveOutEdges(b2, b);
32        b2->appendOutEdge(b);
33        e.m->unlock();
34        // b2's ending address has changed and re-register
35        registerBlockEnd(b2);
36      }
37      return ∅;
38    }
39  }
```

Listing 2. Implementation of Invariant 2 (block end), Invariant 3 (edge creation), and Invariant 4 (block split).

lock for the concurrent hash map. Some implementation of concurrent hash tables provides an entry-level lock natively. For example, Intel TBB's concurrent hash map exposes entry-level reader-writer locks via an "accessor" semantic. We can obtain an "accessor" for the existing entry in the table (inserting one if requested and not already present). The accessor acts as a read or write lock on the entry, and other threads that are trying to obtain a conflicting accessor will wait until the holding thread releases its own accessor. In such case, a `Mutex` in `MapEntry` is not necessary and `MapEntry` can be simplified to be just a `Block` pointer.

Lines 14 to 17 create new edges (Invariant 3) and only the thread who inserted the entry will enter this case. Routine `createEdgesFor` creates control flow edges according to the encountered control flow instruction. Lines 19 to 37 show an implementation of our block split algorithm (Invariant 4). The recursion in line 26 and 35 is guaranteed to finish as each recursion will work with a basic block having a decreasing ending address. Entry-level locks ensure the addition of edges in line 15 is mutually exclusive with the movement of edges on line 31, allowing us to use simpler serial structures to represent the outgoing edges for each block. While it may be possible to use finer grained synchronization here, our performance profiling has not shown this mutual exclusion to be a performance bottleneck.

```
1   void ParallelCFT(q) {
2     delayed_func = new HashMap();
3     while(!q.empty()) {
4       finish { LaunchTasks(q, delayed_func); }
5       MarkNonReturning(q, delayed_func);
6     }
7   }
8   void LaunchTasks(q, delayed_func) {
9     while(!q.empty()) {
10      func = q.pop();
11      async {  // execute the following code block as a task
12        newq = processAFunc(func, delayed_func);
13        LaunchTasks(newq, delayed_func);
14      }
15    }
16  }
17  Queue processAFunc(f, delayed_func) {
18    newq = new Queue()
19    while(f.hasMoreBlocks()) {
20      b = f.nextBlock();
21      b->setEnd(findNextControlFlow(b));
22      edges = registerBlockEnd(b);
23      for(e : edges) {
24        switch(e.type()) {
25        case "call": processCall(f, newq, delayed_func); break;
26        case "ret": processReturn(f, newq, delayed_func); break;
27        default: createNewBasicBlock(f); break;
28        }
29      }
30    }
31    return newq;
32  }
```

Listing 3. The algorithm for parallel control flow traversal.

Invariant 5: Function Creation. There is at most one function starting at any given address. This invariant applies to function entry identification operations and has similar properties and requirements to Invariant 1 for creating blocks, can be implemented in a similar way.

These five invariants ensure that commutative operations can be safely reordered and performed concurrently, and the relative speed of threads will not impact the final results.

5.2 Parallel Control Flow Traversal

Listing 3 presents the algorithm for parallel control flow traversal, which is based on an `async-finish` model of task parallelism [12] and implemented with OpenMP. The function `ParallelCFT` takes q as input, which is initialized with function starting addresses found in the symbol table. We launch tasks in parallel at line 4 with a `finish` code block and the master thread will block until the call to `LaunchTasks` returns.

`delayed_func` is a `concurrent_hash_map` that maintains non-returning function dependencies. When we reach a call site in function f that calls g, if g's non-returning status is unknown, we add an entry to `delayed_func` to note that f has deferred parsing work that waits for g. When we determine the return status of f (either returning or non-returning), we look up `delayed_func` to mark any functions that have deferred work due to f ready for processing. At line 5, after all tasks are finished, `delayed_func` can be non-empty. In this case, functions in `delayed_func` form cyclic dependencies. We mark all functions in `delayed_func` as non-returning and start a new iteration of traversal.

Routine `LaunchTasks` launches tasks in parallel. At line 11, `async` creates a new child task to execute the succeeding code block while the parent task proceeds to iterate over task queue q. The child task process a function, generates a new task queue that includes more functions to process, and launch tasks to process the new queue. This parallel task scheme has the advantage of making newly discovered functions immediately available for processing, and avoid maintaining a global task queue, which would require synchronization across all threads.

Routine `processAFunc` processes a function f. The traversal in f is repeated until there are no more unanalyzed basic blocks (Line 19). For each block b, `findNextControlFlow` (Line 21) decodes instructions until a control flow transfer instruction is encountered and returns the end address of the encountered control flow instruction. A thread-safe instruction decoder is necessary for this step.

Routine `registerBlockEnd` was shown in Listing 2. Only the thread that successfully registers the block end will see a non-empty set of control flow edges returned, following Invariant 3 (edge creation). All other threads reaching the same block end will see an empty set of edges and will follow Invariant 4 (block split) to split the blocks.

The thread that creates the control flow edges will proceed to traverse the edges (Line 23 - 28). If we encounter a function call to g, routine `processCall` may need to create a new function, following Invariant 5 (function creation), and add the new function to the new task queue. We then acquire an accessor for g in `delayed_func`. If g's return status is unknown, we add "f waiting for g" to `delayed_func`.

If we encounter a return edge, routine `processReturn` acquires an accessor for f in `delayed_func`, sets f as returning, adds all functions that are waiting for f in the new task queue, and removes f from `delayed_func`. We then also add f to the new task queue and terminate the current task. This strategy enables us to immediately launch new tasks to process the newly available functions, and is effective for addressing non-returning function dependencies because large functions often contain multiple return instructions.

If we encounter other types of edges, such as indirect, direct or conditional branches, we create new basic blocks based on invariant 1 (block creation) at line 27. Finally, we return newly discovered function entries to initialize new control flow traversal (Line 31).

Jump table analysis. We address two issues raised in Section 4 about jump table analysis. First, jump table analysis (O_{IEC}) in Dyninst does not satisfy the monotonic ordering property. We identify that when Dyninst encounters instructions or path conditions that it cannot analyze, Dyninst will fail to analyze the jump table and generate an empty set of control flow targets. This issue can be addressed by taking the union of the targets discovered along different paths, essentially ignoring instructions or path conditions that fail analysis. In this way, jump table targets identified along valid control flow paths can still be propagated to the indirect jump, and the analysis can generate non-empty set of control flow targets. While this strategy makes the jump table analysis in Dyninst satisfies the monotonic ordering property, it can over-approximate jump table sizes and lead to bogus control flow edges. We will introduce a cleaning strategy in the CFG finalization stage to remove bogus control flow edges.

Second, the monotonic ordering property specifies that we can get a larger graph if we delay jump table analysis as much as possible, but this might delay the finding of return instructions and hurt parallelism due to non-returning function dependencies. We balance these two factors by ordering jump table analysis after the analysis of direct control flow edges in this function, but before call fall-through edges when the callee does not have a known return status. In addition, we repeat the analysis of a jump table after more control flow paths are created within the same function. This fixed-point analysis of jump tables allows us to find most of the targets early in the analysis and gradually converge to a complete set of targets.

5.3 CFG Finalization

The goal of CFG finalization is to remove wrong CFG elements and determine function boundaries. No new CFG elements will be added. CFG finalization includes jump table finalization and tail call finalization. Both steps are done with function level parallelism.

The first step is jump table finalization, where we remove wrong indirect control flow edges. We find that over-approximation of jump targets is caused primarily by over-approximation of jump table sizes. We can mitigate this problem by leveraging an observation that compilers do not emit overlapping jump tables [33]. Therefore, if the analysis of a jump table overflows into another jump table, we can detect over-approximation and apply edge removal operations O_{ER} to remove wrong edges and cascading dangling blocks. We make two observations about this strategy. First, we have established in Section 4 that edge removal operations commutate. Therefore, it is safe to perform this mitigation strategy in parallel. Second, this strategy cannot be used during the parallel control flow traversal step. This is because when we analyze a jump table, we do not know the exact locations of all jump tables in the binary. For this reason, we delay this mitigation of over-approximation until the CFG finalization phase.

We then perform tail call finalization, where we address wrong tail calls edges and determine function boundaries. We handle this with an iterative parallel control flow graph search. Starting from function entries, we add blocks to the boundary of a function by traversing intra-procedural edges. After getting the temporary function boundaries, we use three rules in order to correct tail call results. First, if a branch is marked as not a tail call, but the edge target has a CALL incoming edge, we correct this edge to be a tail call. Second, if a branch is marked as a tail call, but the branch target is within the current function boundary, we correct this edge to be not a tail call. Third, if a branch is currently a tail call, but the edge target has only the current edge as incoming edges, we treat this as not a tail call. This is generally caused by outlined code blocks. After correcting tail calls, we re-perform the function boundary graph search and the tail call correction procedure. We flip the determination of tail call at most once for each edge, ensuring convergence. Finally, we remove functions that do not have incoming inter-procedural edges.

6 Evaluation

We evaluate the correctness of our parallel CFG construction algorithm and implementation and the performance of our work using `hpcstruct` from the HPCToolkit performance analysis tools [2].

6.1 Correctness

Our data set contains 113 binaries obtained by compiling the coreutils and tar projects. These binaries are compiled with GCC 9.3.0 for x86-64, with link-time optimization disabled and other optimizations enabled as specified by the package. Since it is difficult to obtain accurate ground truth for a binary's CFG [10, 20, 28], we approximate the ground truth by compiling these binaries with debug information and injected the flag `-fdump-rtl-dfinish` to generate RTL intermediates. The debug information and RTL are used only for generating the ground truth.

The ground truth of this data set consists of three parts. First, we represent the boundaries of each function with a set of address ranges, essentially projecting the CFG of a function to the virtual address space. The DWARF `.debug_info` section encodes function ranges. In particular, it supports multiple non-contiguous ranges for one function and supports one range corresponding to multiple functions. Therefore, we can evaluate the handling of functions sharing code and non-contiguous functions. Second, we include the size of a jump table as part of the ground truth, which can be extracted by scanning the RTL files. Unfortunately, we cannot derive jump table locations or the actual targets from the RTL files. As existing jump table analysis has focused on bounding the size of jump tables, we believe jump table sizes provide significant evaluation value. Third, RTL encodes the ground truth for calls to non-returning functions, where a non-returning call has `REG_NORETURN` as one of its arguments.

We identified four distinct differences between our implementation and the ground truth: (1) Failing to identify non-returning calls to '`error`'. '`error`' is non-returning when the first argument is non-zero, but returning when the first argument is zero. Existing non-returning function analysis performs name matches for external functions. This approach does not work for '`error`'. (2) For a function `foo`, the compiler may emit another function symbol ("`foo.cold`") for outlined cold blocks from `foo`. In our results, `foo` does not include addresses from `foo.cold`. However, the debugging information lists the address ranges of "`foo.cold`" as part of `foo`. (3) Failing to resolve a jump table whose calculation uses the stack to store intermediate values. (4) An extra indirect jump target caused by failing to identify a non-returning call to '`error`', which causes a wrong control flow edge and jump table approximation.

In all cases above, the differences are caused by either incorrectness in the individual CFG operations (O_{CFEG} and O_{IEC}) or mismatches between the symbol table and DWARF information. In other words, the errors are not caused incorrect parallelism and can be fixed by improving the implementations of O_{CFEG} and O_{IEC}.

6.2 HPCToolkit's `hpcstruct`

HPCToolkit is an integrated suite of tools for measurement and analysis of application performance on computers ranging from desktops to supercomputers. To aid in the attribution of performance measurements to an application's source code, the `hpcstruct` utility in HPCToolkit relates each machine instruction address to the static calling context in which it occurs. In particular, `hpcstruct` is able to relate instructions to their original function or loop construct by inspection of the binary's final CFG, and to an inlined function or template and source lines if DWARF debugging information is available. `hpcstruct` is built upon Dyninst and directly benefits from the new parallel CFG construction algorithms described in this paper.

We use four large binaries to illustrate the effectiveness of our parallelization for speeding up performance analysis, including two binaries from Lawrence Livermore National Laboratory (Ares and Kull), one large binary from Argonne National Laboratory (Camellia), and one shared library from TensorFlow [1]. Statistics about these binaries are given in Table 1. Experiments with Ares and Kull were done on a machine with 16 hardware threads at LLNL. Experiments with Camellia and TensorFlow were done on a system with two 18-core Xeon E5-2695v4 processors.

The results are presented in Table 2 and in Figure 2. The execution of `hpcstruct` consists of (1) parsing DWARF, (2) CFG construction, (3) constructing loops, (4) CFG traversal to query program structure, and additional serial stages unrelated to binary analysis. "Prior" consists of parallel (1), (3), (4) with serial (2). "Current" uses parallel implementations

Table 1. Statistics of the large binaries.

	Size (MiB)			
Binary	Total	`.text`	Arch.	Endian (Width)
Ares	363.40	77.01	Power	Little (64)
Kull	1913.50	149.13	x86	Little (64)
Camellia	299.08	40.81	Power	Big (32)
TensorFlow	7844.81	112.21	x86	Little (64)

Table 2. Performance results, averages of 10 runs. Times for Prior and Current `hpcstruct` represent performance before and after applying our parallel CFG construction, isolated in `CFG`. `%SCFG` lists the percentage of Prior spent in serial CFG construction.

Binary		Time Taken in `hpcstruct` (s)			
	Cores	Prior	`%SCFG`	`CFG`	Current
	1	237.97	42.68%	101.57	237.97 ± 3.79
Ares	16	120.80	84.08%	11.21	30.44 ± 0.28
	Speedup	1.97×		9.06×	7.82×
	1	690.86	25.59%	176.79	690.86
Kull[1]	16	269.68	65.56%	19.66	112.55
	Speedup	2.56×		8.99×	6.14×
	1	118.39	38.93%	46.10	118.39 ± 2.24
Camellia [25]	16	60.93	75.66%	5.38	20.21 ± 0.17
	Speedup	1.94×		8.57×	5.86×
	1	1252.88	8.89%	112.55	1252.88 ± 19.70
TensorFlow [1]	16	263.81	42.66%	9.56	160.82 ± 3.08
	Speedup	4.75×		11.77×	7.79×
	32	253.18	44.45%	5.49	146.12 ± 1.70
	64	262.70	42.84%	4.46	154.61 ± 2.86
	Speedup	4.77×		25.22×	8.10×

for all four stages. The "CFG" column is the time taken by the parallel (2) in "Current" `hpcstruct`.

The "Prior" column shows that a serial CFG construction significantly limits the parallelization of `hpcstruct`, achieving a speedup of 2× to 5× with 16 hardware threads; using more than 16 hardware threads hardly improve performance. The "%SCFG" column shows that serial CFG construction is clearly a bottleneck, taking about 40% to 80% of the total runtime when `hpcstruct` runs in parallel.

The "Current" column shows that `hpcstruct` has an end-to-end speedup of 6× to 8×, which is much higher compared to `hpcstruct` with serial CFG construction. We achieve a speedup of 9× to 12× for parallel constructing CFGs with 16 hardware threads, and up to 25× with 64 hardware threads, showing that our parallel CFG construction scales well. After our changes `hpcstruct` is now limited by other serial phases

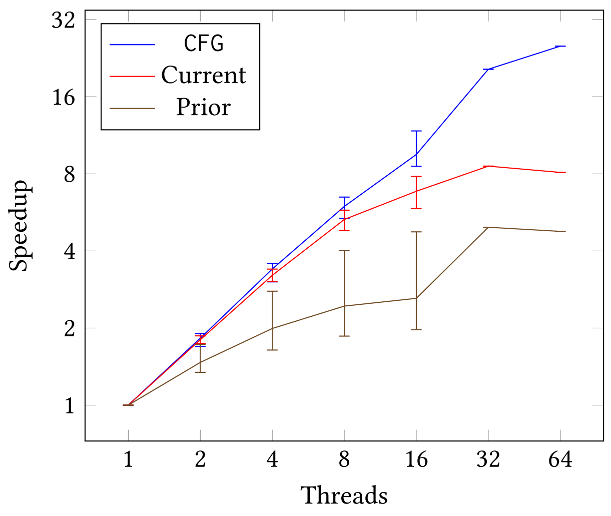


Figure 2. Average speedup (geometric mean) of `hpcstruct` on the four binaries, as described in Section 6.2.

and the total runtime of `hpcstruct` has been significantly reduced.

7 Discussion

Comparing with other tools: We ran both angr [28] and rev.ng [10] on the 7.7GiB shared library from TensorFlow. Neither tool finished CFG construction within 1 hour. We believe their software architecture (angr is written in python) and binary analysis approach (lifting binaries to IR) are not suitable for analyzing large binaries.

Benefiting other applications: CFG construction is also used by other binary analysis applications. For example, software vulnerability searching calculates binary code similarity [6, 9] to match known vulnerable code.

Compiler assisted analysis: Our work opportunistically uses information from the compiler (such as symbol tables and DWARF). However, this is not a complete solution and we cannot rely on sufficient or even accurate compiler support. Surprisingly often for even the most widely-used compilers, the compiler-provided information is incomplete or inaccurate [17]. For performance analysis, software developers often use the compiler and optimization flags that lead to greatest performance, which often leads to less accurate debugging information.

Source code CFG construction: The challenges of binary code CFG construction are largely distinct from those of source code CFG construction. Binary code functions can share code, which is the main reason that we must derive operation properties to guide our design invariants to support analysis of multiple functions in a binary in parallel. In

[1]Results for Kull are based on one run for each thread count. We have limited access to the binary and cannot repeat the experiment.

contrast, source code functions cannot overlap unless functions are nested. In this case, CFG construction for source code does not require rigorous synchronization.

Task load balancing: We apply a heuristic that begins analysis of large functions first. This minimizes the potential for extreme imbalance that would occur if a thread began analysis of a large function when analysis of all other functions was nearly complete. With our approach, the analysis of large functions by one or more threads is typically balanced by analysis of small functions by others. We acknowledge that if there is a large function whose size is larger than all small functions combined, our approach will suffer from load imbalance. In practice, we rarely observe such extreme imbalance. Avoiding imbalances caused by a small number of extremely large functions would require intra-function parallelism.

Function Pointers: We do not handle function pointers in this work and only track through direct calls in our interprocedural analysis. We handle indirect calls conservatively, assuming they will return. Two main approaches for function pointer analysis are signature matching based on type information [18] and point-to analysis [34]. These approaches are mostly studied with source code. Extending them to binary code analysis is an interesting research direction for future work.

8 Conclusion

With the increasing size of software, it is critical to add multithreaded parallelism to speed up binary analysis. Our work centers on a theoretical abstraction that expresses CFG construction as applications of individual CFG operations. We derived operation dependencies, commutativity, and monotonic ordering properties, which guided us towards a new parallel CFG construction algorithm. We evaluated our parallel binary analysis with a performance analysis tool `hpcstruct`, achieving as much as 25× speedup for parallel CFG construction and 8× overall for `hpcstruct` using 64 hardware threads, significantly cutting the wait times for their users and developers.

Acknowledgments

We thank Matthew Legendre (LLNL) for running our experiments on Kull and Ares. This research was supported in part by the Exascale Computing Project (17-SC-20-SC)—a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, by National Science Foundation (Awards 1450273 and 1449918), and by DOE grant DE-AC05-00OR22725 under ORNL contract, 4000164398.

References

[1] Martín Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience* 22, 6 (April 2010), 685–701.

[3] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. 2007. Stack Trace Analysis for Large Scale Debugging. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Long Beach, California, USA, 1–10.

[4] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. 2011. Refinement-based CFG Reconstruction from Unstructured Programs. In *12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Austin, TX, USA, 54–69.

[5] Andrew R. Bernat and Barton P. Miller. 2012. Structured Binary Editing with a CFG Transformation Algebra. In *2012 19th Working Conference on Reverse Engineering (WCRE)*. Kingston, Ontario, Canada, 10 pages.

[6] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-Architecture Cross-OS Binary Search. In *2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. Seattle, WA, USA.

[7] William D. Clinger. 1998. Proper Tail Recursion and Space Efficiency. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, Montreal, Canada, 174–185.

[8] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. 2002. Active Harmony: Towards Automated Performance Tuning. In *2002 ACM/IEEE Conference on Supercomputing (SC)*. Baltimore, Maryland, 1–11.

[9] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, California, USA, 266–280.

[10] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *26th International Conference on Compiler Construction (CC)*. Austin, TX, USA.

[11] Yizi Gu and John Mellor-Crummey. 2018. Dynamic Data Race Detection for OpenMP Programs. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. Dallas, Texas.

[12] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. 2009. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*.

[13] Intel. [n.d.]. Threaded Building Blocks, https://www.threadingbuildingblocks.org/.

[14] Emily R. Jacobson, Andrew R. Bernat, William R. Williams, and Barton P. Miller. 2014. Detecting Code Reuse Attacks with a Model of Conformant Program Execution. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*. Munich, Germany, 18 pages.

[15] Johannes Kinder and Dmitry Kravchenko. 2012. Alternating Control Flow Reconstruction. In *13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Philadelphia, PA.

[16] Johannes Kinder and Helmut Veith. 2008. Jakstab: A Static Analysis Platform for Binaries. In *20th International Conference on Computer Aided Verification (CAV)*. Princeton, NJ, USA, 423–427.

[17] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug Information Validation for Optimized Code. In *Proceedings of*

the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). London, UK, 1052–1065. https://doi.org/10.1145/3385412.3386020

[18] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. London, United Kingdom.

[19] Tobias Maier, Peter Sanders, and Roman Dementiev. 2019. Concurrent Hash Tables: Fast and General(?)! *ACM Trans. Parallel Comput.* 5, 4, Article 16 (Feb. 2019), 32 pages. https://doi.org/10.1145/3309206

[20] Xiaozhu Meng and Barton P. Miller. 2016. Binary Code Is Not Easy. In *The International Symposium on Software Testing and Analysis (ISSTA)*. Saarbrücken, Germany.

[21] Xiaozhu Meng, Barton P. Miller, and Kwang-Sung Jun. 2017. Identifying Multiple Authors in a Binary Program. In *22nd European Conference on Research in Computer Security (ESORICS)*. Oslo, Norway.

[22] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. 1995. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer* 28, 11 (Nov. 1995), 37–46.

[23] Paradyn Project. [n.d.]. Dyninst: Putting the Performance in High Performance Computing, http://www.dyninst.org.

[24] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.

[25] Nathan V. Roberts. 2014. Camellia: A software framework for discontinuous Petrov-Galerkin methods. *Computers & Mathematics with Applications* 68, 11 (2014), 1581 – 1604. https://doi.org/10.1016/j.camwa.2014.08.010 Minimum Residual and Least Squares Finite Element Methods.

[26] Nathan Rosenblum, Xiaojin Zhu, and Barton P. Miller. 2011. Who wrote this code? identifying the authors of program binaries. In *16th European Conference on Research in Computer Security (ESORICS)*. Leuven, Belgium, 18 pages.

[27] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. 2002. Disassembly of Executable Code Revisited. In *Ninth Working Conference on Reverse Engineering (WCRE)*. Richmond, VA, USA.

[28] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA.

[29] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. 2009. Binary Analysis for Measurement and Attribution of Program Performance. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Dublin, Ireland). 441–452.

[30] H. Theiling. 2000. Extracting Safe and Precise Control Flow from Binaries. In *the Seventh International Conference on Real-Time Systems and Applications (RTCSA)*. Cheju Island, South Korea, 23–30.

[31] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Denver, Colorado, USA.

[32] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA.

[33] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland.

[34] Jisheng Zhao, Michael G. Burke, and Vivek Sarkar. 2018. Parallel Sparse Flow-Sensitive Points-to Analysis. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. Vienna, Austria, 59–70.

A Artifact Appendix

A.1 Abstract

The artifact contains code and data needed to reproduce the correctness and performance experiments used in Section 6 of the paper.

The correctness experiments compare the output from our analysis with the data extracted from DWARF debugging information. To our best effort, the output from our analysis is normalized to match the output from DWARF. Still, the comparison requires manual analysis and reproducing the analysis results require basic understanding of x86 assembly.

The performance experiments run our analysis with large binaries using various numbers of threads to compute the speedup of our analysis. In the paper, we used four large binaries in the experiments. We provide two of the four binaries in this artifact. Running with Camellia and TensorFlow take around 23GB and 75GB of memory respectively. We did the experiments on a machine with 128GB memory and 72 hardware threads. Our artifact also supports running performance experiments with your own binaries.

Below we provide a brief description for the artifact. We provide a README file that includes detailed steps for artifact evaluation, which is available in the tar ball downloaded at https://zenodo.org/record/4295514#.X8LXuqpKhTY.

A.2 Getting Started Guide

The artifact is available as a docker container. You can use the pre-built image:

```
docker run -it registry.gitlab.com/blue42u/ppopp-docker
```

You can also build the docker image yourself, which will take about 2 hours for installing necessary software dependencies:

```
git clone https://gitlab.com/blue42u/ppopp-docker
cd ppopp-docker
docker build .
```

A.3 Reproducing the correctness results

In the container, directory `/correctness` contains all the files needed for the correctness experiments. The output for this experiment is already computed in the process of building the docker image. Please refer to Section C in the README file for more details about how we manually compare the difference between our analysis output and ground truth derived from DWARF debugging information.

A.4 Reproducing the performance results

Directory `/performance` contains all the files needed for this experiment. We provide Camellia and TensorFlow as two sample input binaries. To reproduce the results shown in Table 2, you can run:

```
cd /performance; make THREADS='16 64' REPS=10
```

The `THREADS` argument means that to run the experiments with 16 and 64 threads (single thread is always included for computing speed up). The `REPS` argument specifies the number of iterations for running with each binary. The final results include per-binary results and their concatenation in `results.txt` (which is the default make target if none is specified).

Running with Camellia and TensorFlow take around 23GB and 75GB of memory respectively. We did the experiments on a machine with 128GB memory and 72 hardware threads. The whole experiments may take a couple of hours depending on the computing power of your machine. You can reduce the number of repetition to reduce the total runtime. When `REPS` is set to 1, the output will include "nan" in the last column because we cannot calculate the variance with just 1 run.

Run `make` in `/performance` will trigger analyzing all binaries in the `/performance/input` directory. By default, `/performance/input` has the camellia and tensorflow binaries. You can exclude binaries from the experiment by moving binaries to other places or include binaries to the experiment by adding binaries to the directory.