Cyclebite: Extracting Task Graphs From Unstructured Compute-Programs

Benjamin R. Willis, Aviral Shrivastava, *Senior Member, IEEE*, Joshua Mack, *Graduate Student Member, IEEE*, Shail Dave, *Member, IEEE*, Chaitali Chakrabarti, *Fellow, IEEE*, and John Brunhaver

***Abstract*—Extracting portable performance in an application requires structuring that program into a data-flow graph of coarse-grained tasks (CGTs). Structuring applications that interconnect multiple external libraries and custom code (i.e., "Code From The Wild" (CFTW)) is challenging. When experts manually restructure a program, they trivialize the extraction of structure; however, this expertise is not broadly available. Automatic structuring approaches focus on the intersection of hot code and static loops, ignoring the data dependencies between tasks and significantly reducing the scope of analyzeable programs. This work addresses the problem of extracting the data-flow graph of CGTs from CFTW. To that end, we present Cyclebite. Our approach extracts CGTs from unstructured compute-programs by detecting CGT candidates in the simplified Markov Control Graph (MCG), and localizing CGTs in an epoch profile. Additionally, the epoch profile extracts the data dependence between CGTs required to build the data-flow graph of CGTs. Cyclebite demonstrates a robust selectivity for critical CGTs relative to the state-of-the-art (SoA), leading to a potential speedup of 12x on average and thread-scaling of 24x on average compared to modern compiler optimizers. We validate the results of Cyclebite and compare them to two SoA techniques using an input corpus of 25 open-source C/C++ libraries with 2,019 unique execution profiles.**

***Index Terms*—Produce-consume task graph, memory dependency analysis, task partitioning, dynamic control flow graph, epoch.**

I. Introduction

STRUCTURED representations of applications support high-performance execution [1] and system portability [2]. Domain-specific languages (DSLs) limit their expressibility to directly infer a program's organization and reduce the cognitive load of programmers [3]. A structure that defines tasks and their data dependency drives schedule optimization, increasing task-to-task data locality and concurrence through pipeline parallelism [4], [5]. Further, the application's overall structure is required for optimization when its tasks have data-flow interdependence [6]. Prior works in scheduling [7], high-level synthesis [8], and performance prediction [9] often assume an apriori structured representation.

Unfortunately, modern software engineering methodology obfuscates the required program structure: tasks interconnected by data-flows [4]. The virtues of abstraction and modularity, typified by templated application programming interfaces (APIs), increase productivity and accessibility. However, the obfuscation and blocking required for these interfaces prevent holistic or structural optimization [3]. Further, API calls do not map one-to-one to tasks; therefore, we require more than a static examination of these calls. Thus to support the optimization and parallelization of CFTW, we require a method to extract their task dependency graph.

Existing methodologies miss the code required for complete program optimization and include code that bloats the space of optimizable code. Static analysis (SA) techniques rely on the information included manually by the programmer to point to optimization opportunities. When applied to CFTW, no explicit structure is defined. Thus, SA cannot extract structure (Fig. 1a). HotCode (HC) points to disparate control structures like those in Fig. 1(b), requiring manual intervention to resolve into complete tasks. HotLoop (HL) finds complete loops in the structure but misses loops critical for optimization and may also include information about the task that is not useful, as seen in Fig. 1(c). Finally, all structuring techniques cannot resolve indirect function calls and lack communication information between tasks, though memory profiling techniques exist [10]. An ideal solution, like Fig. 1(d), captures each complete task and observes its communication patterns with others.

The Cyclebite toolchain [11] extracts produce-consume task graphs from programs. Cyclebite observes the frequency of basic-block transitions to construct a Markov Control Graph (MCG). Cyclebite identifies CGTs by searching for the structures that give rise to frequent execution, i.e., a graph cycle. To facilitate that cycle finding, Cyclebite inlines branches and functions such that strongly connected components become cycles. With the CGTs identified, Cyclebite performs a second pass of the program's execution to localize the *epochs*

Manuscript received 14 July 2023; revised 15 October 2023; accepted 16 October 2023. Date of publication 30 October 2023; date of current version 22 December 2023. This work was supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under Agreement FA8650-18-2-7860. Recommended for acceptance by J. Shalf. *(Corresponding author: Benjamin R. Willis.)*

Benjamin R. Willis, Chaitali Chakrabarti, and John Brunhaver are with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: benroywillis@gmail.com; chaitali@asu.edu; jbrunhaver@gmail.com).

Aviral Shrivastava and Shail Dave are with the School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ 85281 USA (e-mail: ashriva6@asu.edu; sjdave@asu.edu).

Joshua Mack is with the Electrical and Computer Engineering Department, University of Arizona, Tuscon, AZ 85719 USA (e-mail: jmack2545@arizona.edu).

Digital Object Identifier 10.1109/TC.2023.3327504

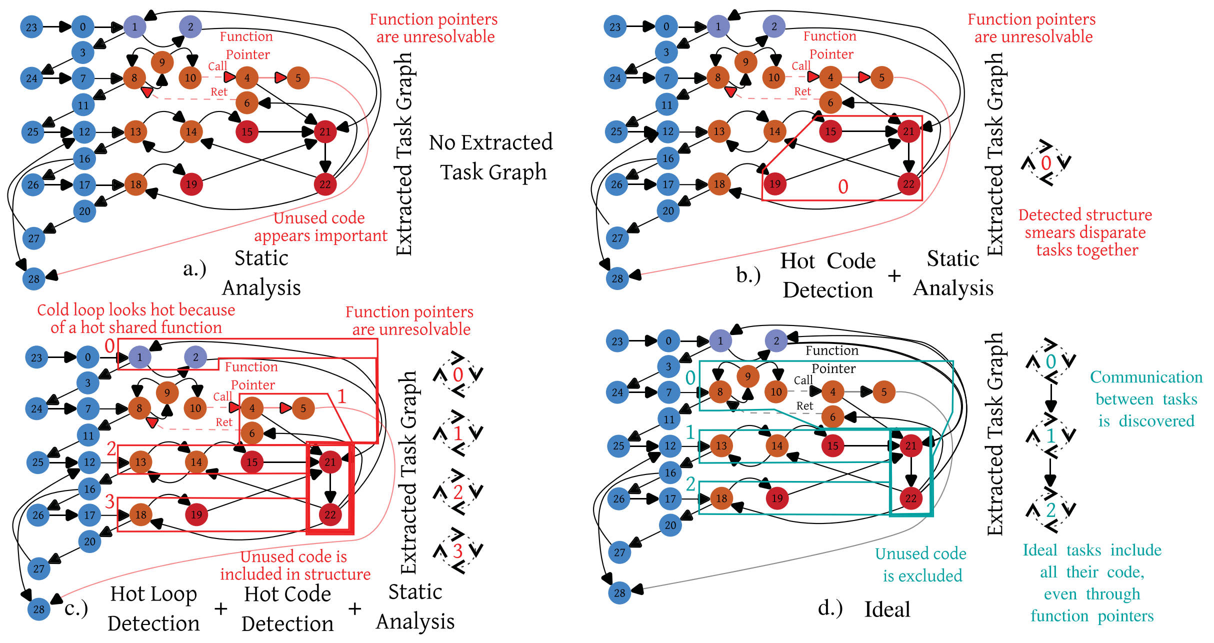

Function pointers are unresolvable
Function Pointer
Call
Ret
Unused code appears important
Extracted Task Graph
No Extracted Task Graph
a.)
Static Analysis
b.)
Hot Code Detection + Static Analysis
Detected structure smears disparate tasks together
Cold loop looks hot because of a hot shared function
Unused code is included in structure
c.)
Hot Loop Detection + Hot Code Detection + Static Analysis
Communication between tasks is discovered
Unused code is excluded
Ideal tasks include all their code, even through function pointers
d.) Ideal

Fig. 1. State-of-the-art structuring techniques may find sub-optimal structures in Code From The Wild (CFTW). All SoA techniques cannot resolve function pointers. Static Analysis (SA) (a) does not contain any information on the relative importance of different codes. HotCode (HC) (b) detection captures hot code that may not include critical sections of the task. HotLoop (HL) (c) detection incorporates statically-defined loop information to HC but captures *all* static code - including unused code. An ideal task structure (d) resolves the function pointer, captures each task in its entirety (including orange code), rejects all unimportant code like unused error checks, and finds communication patterns between the tasks it finds (like read-after-write (RAW) dependencies).

(a section of time in which the target application is within a distinct section of its code) [12], [13] observe the data-flow between CGTs. We demonstrate the robustness of this approach using a corpus of 2,019 C/C++ applications spanning image and signal processing, linear algebra, cryptography, and software-defined radio.

The contributions of the paper are the methods to

1) generate a simplified Markov Control Graph (MCG)
2) extract coarse-grained tasks from that MCG
3) use coarse-grained tasks to localize epochs and attribute data-dependencies to those epochs.

II. Background

Portable high performance requires a high-level program structure: a produce-consume interconnection of CGTs [14]. Halide [3] and DeLite [2] leverage the explicit structure of an application to optimize the schedule and communication of concurrent interdependent tasks - including a trade-off between locality, parallelism, and re-computation. This structure parameterizes computation and memory access patterns to accommodate architecture specifics (e.g., cache hierarchy and compute-lane width). Further, that parametric high-level representation extends the compilation support beyond general-purpose processors to include graphics processors [15] and programmable gate arrays [8]. Thus, our work seeks to extract the *produce-consume task graph* (a directed acyclic graph of nodes and edges; a node is a task and each edge points from producer to consumer) from CFTW.

CGTs encapsulate fine-grained tasks called atoms (e.g., matrix multiplication encapsulates multiply-accumulate) in a high-level abstraction describing the interrelation of those atoms, which facilitates optimization. Polyhedral optimization tools [16] utilize the memory access pattern relationship between atoms to re-order and fuse their execution to improve data locality and parallelism. Domain-specific languages (DSLs) [3], [6], [17] relax the constraints required to detect polyhedral parallel patterns in static code by bringing those patterns into the language semantic, resulting in greater freedom of expression. Cyclebite provides the flexibility and performance of DSL semantic templates without requiring that programs (e.g., CFTW) be re-written into DSLs.

State-of-the-art (SoA) structuring techniques match tasks to templates. A *HotCode* (HC) based approach seeks contiguous sections of code with a high frequency of execution [18], [19]. Unfortunately, the complete description of a task will likely include low-frequency code, as with low probability control flow. The *HotLoop* (HL) template identifies static affine loops containing HC to incorporate those low-frequency components [20], which is critical for polyhedral analysis [16]. However, static loops often contain additional code unsuitable for optimization and fail to capture recursive tasks. Another templated approach, called *HotFunction* (HF), assumes functions that contain HC indicate the CGTs of the program [21], [22]. HL and HF require continuous static code, meaning that *function indirection* (function pointers) may result in task fragmentation. Additionally, code reuse causes false positives and fuses unrelated tasks when structuring CFTW. The gaps between these template models and the required code are significant, and we quantify this difference in Section V-C. We require a control pattern sufficiently general to accommodate all existing patterns while excluding the false-positive and false-negative tasks SoA techniques find in CFTW.

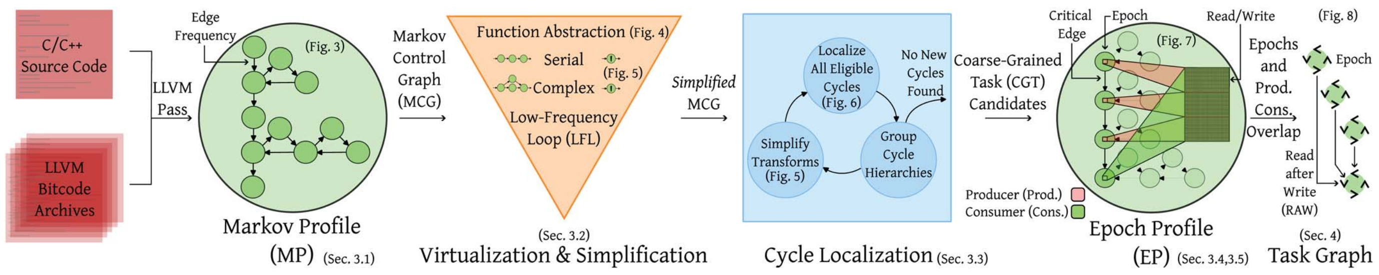

C/C++ Source Code
LLVM Bitcode Archives
LLVM Pass
Edge Frequency
(Fig. 3)
Markov Profile (MP) (Sec. 3.1)
Markov Control Graph (MCG)
Function Abstraction (Fig. 4)
Serial
Complex
(Fig. 5)
Low-Frequency Loop (LFL)
(Sec. 3.2) Virtualization & Simplification
Simplified MCG
Localize All Eligible Cycles (Fig. 6)
No New Cycles Found
Group Cycle Hierarchies
Simplify Transforms (Fig. 5)
Cycle Localization (Sec. 3.3)
Coarse-Grained Task (CGT) Candidates
Critical Edge
Epoch
(Fig. 7)
Producer (Prod.)
Consumer (Cons.)
Epoch Profile (EP) (Sec. 3.4,3.5)
Read/Write
(Fig. 8)
Epochs and Prod. Cons. Overlap
Epoch
Read after Write (RAW)
(Sec. 4) Task Graph

Fig. 2. Overview of the Cyclebite structuring pipeline. A C/C++ program and its dependencies are compiled into a static LLVM IR bitcode file and instrumented with a dynamic profile pass - the Markov Profile (MP). The instrumented program is compiled and run, allowing MP to generate a state-transition table, which is exported as a Markov Control Graph (MCG). Four transformations designed to simplify the localization of cycles are applied to the MCG - this results in the *simplified* MCG. Cycles are localized and grouped into hierarchies, forming coarse-grained task (CGT) candidates. CGT candidates are exported into a second dynamic profile pass, the Epoch Profile (EP), that localizes program epochs and observes their memory transactions. After all epochs have been localized, memory transactions are used to form producer-consumer relationships between epochs. This results in a task graph - nodes are epochs and edges are read-after-write (RAW) dependencies.

Cyclebite contributes a widely applicable CGT template: cyclic subgraphs in the observed *dynamic control flow graph* (DCFG). This generalized template captures the parallel programming patterns described by Asanovic [23], as their naive and optimized expressions utilize loops (not necessarily affine) which result in DCFG cycles. When a CGT comes from a hot loop, hot function, or strongly connected component (SSC) [24], Cyclebite finds it while rejecting the code that appears hot but is not part of a CGT. In addition to their cyclical structure, Cyclebite requires that each CGT have at least one instance with significant frequency. Prior structuring techniques focus on the total frequency count of each task (e.g., HC) and rely on static boundaries of program primitives (e.g., static loops (HL) or functions (HF)) - they do not take into account the frequency of task instances. Significantly, Cyclebite's generalized template captures parallel patterns (e.g., sparse matrix multiply, breadth-first search) that are not captured by HL as their implementations are rarely affine loops.

When SoA techniques construct control flow graphs on CFTW, the CGT structures are obfuscated by redundant function calls, high-branching sequences, and function indirection. CFTW often contains software engineering practices such as code reuse (i.e., modular functions), flexible code (i.e., high-branching sequences), and interposable functionality (i.e., function pointers); these techniques appear often in open-source libraries. *Modular functions* are implementations of a mundane task (e.g., logging, exception handling, error-checking); they are difficult to structure because they may be truly important code (and thus should be accepted) or just incidentally hot from many singleton calls, or its alias (and should be rejected). High-flexibility code will accommodate many different cases within a single CGT, making CGTs appear bloated when structured by HL even though they only use a small portion of their static code. Static information alone will have missing edges in both the control flow graph and the call graph, possibly fracturing a CGT into many disconnected tasks. To obtain the information to resolve these issues, dynamically-observed frequency of each *state transition* (a change in program state from the previous state to the current) is necessary.

Cyclebite resolves function indirection and dynamically-determined control code by constructing a DCFG where the nodes are basic blocks [25] and the weighted directed edges are the observed frequency of transitioning from one basic block to the next. Frameworks relying on HC [26], [27] annotate the static control flow graph with the frequency of basic block executions, which encounter the same state-transition ambiguities of a purely static analysis. Cyclebite records a state transition history of 1 in its dynamic profiles, meaning its memory overhead and execution-time dilation are manageable, allowing it to scale to large applications. A complete history of the execution order of basic blocks, while capturing significant information, requires arbitrarily large log files, limiting the scope of this technique to short-running programs [28]. A complete, simplified DCFG and call graph have all the necessary information for Cyclebite to localize its cycles.

Cyclebite localizes CGT instances and their data-flow patterns by dynamically observing the epochs of the program and the memory transactions each epoch executes, combining prior work on instruction-level memory dependency analysis [10], [26], [27] and epoch-based simulation abstractions [13]. Cyclebite localizes the epochs of an application by incrementing the epoch each time a *critical edge* (a basic block transition that denotes the boundary of a CGT) is observed (further explained in Section III-D) - when an epoch maps to a cycle localized by Cyclebite, that epoch is a *CGT instance*. SoA structuring techniques do not account for the frequency of the instances of the tasks they select - this gives rise to false-positive tasks that were used many times over the execution of a program but represent common functionality rather than a task's atom. In order for programmers to hide the communication and memory overheads of accelerators [29], its utilization must be significant - this makes CGTs with low per-instance utilization poor accelerator candidates. Cyclebite rejects false-positive CGTs by rejecting CGTs whose per-instance utilization is low. Modern memory profiling frameworks [30] attribute memory information to simple static structures like individual instructions and static loops. SD3 [10] and Cyclebite use the temporal ordering of CGT instances to find last-writer dependencies [31]. However, while contributing methods to reduce memory usage and runtime overhead, SD3 only supports localizing the instances of static loops, limiting the scope of its analysis to dependencies only between tasks (which may miss important memory operations in serial code like memmov and memset) and only supports tasks from HL. Cyclebite's memory pass supports CFTW - serial code operations are also captured in their own epochs, and all CGTs captured by Cyclebite can be profiled. Cyclebite exports a task graph - the final structure of the target program - that represents program epochs as nodes and communication patterns between epochs as edges.

III. Cyclebite

Cyclebite structures CFTW by generating a model of the target application's control structure, called the Markov Control Graph (MCG), and using the MCG to create a produce-consume task graph. Cyclebite observes the DCFG of a target application with the Markov Profile (MP) - a dynamic profiling backend that observes the frequency of state transitions in the program - which exports a *state transition table* (a sparse matrix that contains the observed frequencies of each state transition in the program - row IDs are source node IDs, and column IDs are sink node IDs) as seen in Fig. 3(b). Then Cyclebite constructs a *simplified* MCG from the state transition table by building an MCG and applying a set of transforms that simplify cycle localization. Cyclebite localizes the cycles within the simplified MCG, forming *CGT candidates* until no more localizable cycles are left. Finally, Cyclebite executes a second dynamic profile, the Epoch Profile (EP), that localizes the epochs of the target program and discovers communication patterns between them to form a task graph. CGT candidates whose per-instance utilization are found to be sufficient by EP are designated as *CGTs* (coarse-grained tasks that have at least one instance whose local frequency is 32 or greater).

A. Markov Profile

MP dynamically observes each state transition during the target application's execution, collecting all information necessary to resolve ambiguous control flow like function pointers and unused code. SA, HC, and HL cannot resolve function pointers, as shown in Fig. 1(a)–(c). Further, HC and HL will not be able to recognize the hot modular function (nodes 21 & 22), which both fuses disparate cycles together and upgrades the cold cycle (nodes 1 & 2) to a hot one. Unlike a dynamic trace that records each state in chronological order, as shown in Fig. 3(a), MP observes the frequency of each *first-order state transition* (the transition that occurs between the previous basic block and the current basic block) which captures all information necessary to resolve function pointers and unused code. A target program is instrumented with MP using an LLVM pass, which injects a function call to MP at the entry point of each basic block in the program. To simplify the control sequence of context switches, MP transforms basic blocks that contain context-switching (like function calls) and context-ending (like return instructions) instructions to only contain that instruction. At each call, MP records each state transition by hashing the pair of basic blocks involved in the state transition and incrementing the frequency of that entry. MP stores the frequency of each state transition in a state transition table, shown in Fig. 3(b).

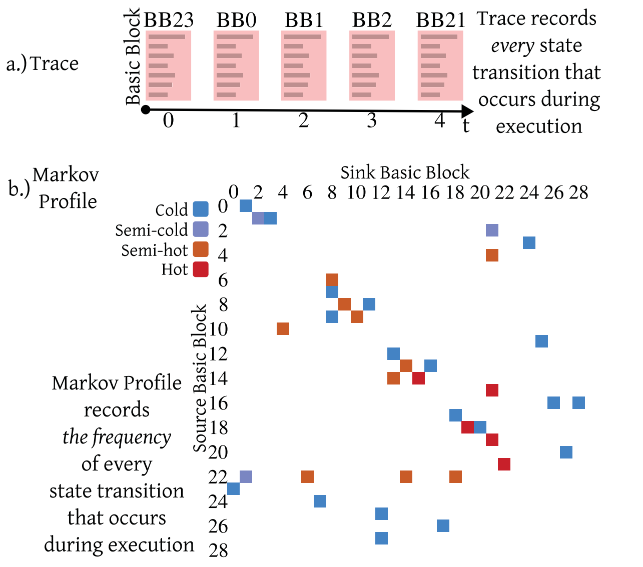


Fig. 3. A dynamic trace records *both the state transition and its timestamp* during the execution of a target program. Dynamic profiles like the Markov Profile (MP) record *the frequency* of each state transition, which omits the timestamp of each state transition. While MP is far more efficient in time and memory than a dynamic trace, functions shared among many tasks can cause these tasks to fuse together, making tasks challenging to separate during localization.

The state transition table is then exported and used to generate the MCG of the program.

The MCG structures the state transition table into a graph representation of the program's control flow, free of control ambiguities like function pointers and unused code. MCG is a directed graph with nodes as LLVM basic blocks and directed edges as probability-weighted state transitions. To calculate each edge weight, the total frequency of all outgoing edges of a given node normalizes each edge frequency, forming probabilities; this forms a Markov chain. While the basic block groups with cyclical structure can be localized in this abstraction, the MCG likely contains control sequences that make cycle localization difficult (off-path control inside cycle bodies, functional reuse that makes disparate cycles appear in the same cycle). To address this problem, cycle-preserving transforms are applied to the MCG.

B. Virtualization and Simplification

Cyclebite simplifies cycle localization by applying a series of transforms to the MCG that reduce the complexity of subgraphs and preserve all CGT-eligible cycles. Cycles that contain off-path control, nested cycles, and shared functions are difficult to localize because of many possible traversals through their execution. Fig. 1(a) demonstrates a few cycles whose executions contain many possible paths, including through other cycles. Thus, transforming this cycle to a form that contains as few paths of execution as possible and with as little overlap with other cycles as possible simplifies its localization.

Cyclebite eliminates the overlap of independent cycles that share modular functions by inlining their subgraphs in the MCG. The bodies of these shared functions merge otherwise unrelated cycles, making separating fused tasks impossible.

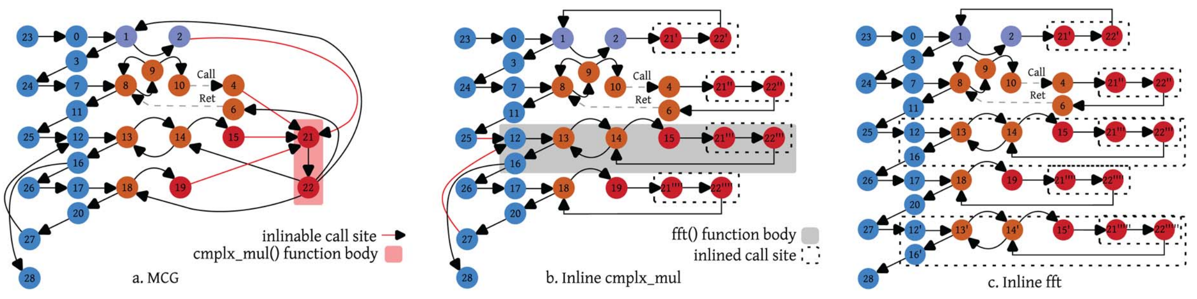


Fig. 4. Shared functions can fuse unrelated cycles together. To unfuse their control flow in subfigure a), Cyclebite inlines shared function bodies at all their call sites, seen in green. Inlining is done in reverse-hierarchical order, meaning Cyclebite transforms the child-most function calls first and the parent-most function calls last. In b), a function body used by all four cycles is inlined, and then in c), a function called twice is inlined at each call site.

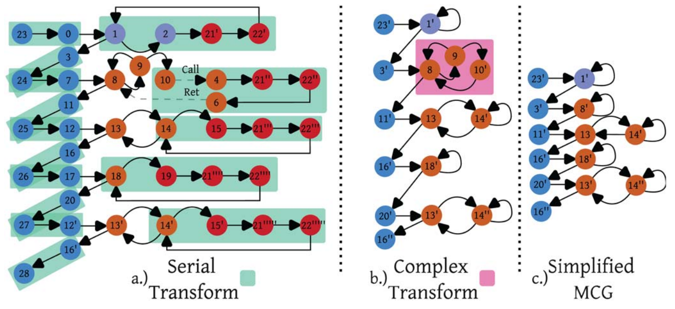


Fig. 5. Cyclebite reduces the complexity of cycle localization by simplifying the MCG. Nodes with apostrophes succeeding their ID are *virtual nodes* (nodes that represent a transformed subgraph). Serial Transform is applied to the MCG first (a). Next, Complex Transform (b) reduces subgraphs with multiple execution paths and no cycles to single nodes. Once no transform opportunities are present (c), the *simplified* MCG is ready for cycle localization.

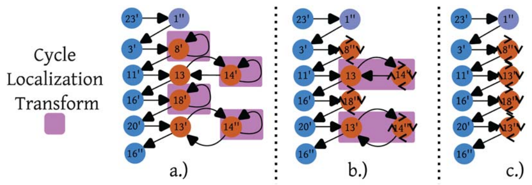


Fig. 6. Cyclebite localizes cycles in hierarchical order. Starting with the child-most cycles (a) Cyclebite transforms the subgraphs highlighted in purple into single nodes found in b). The nodes with arrows on their border represent virtualized cycles. Then, the parent cycles 13–14” and 13’–14”’ are transformed in subfigure b) until no more cycles are available for localization (c).

Fig. 4(a) shows the MCG with shared function bodies highlighted. This transform inlines “child-most” functions first, then its parent functions, and so on, until no inlinable function bodies are left. Fig. 4(b) shows the MCG after “cmplx_mul” has been inlined at all its call sites, resulting in 4 instances of its function body. Next, Cyclebite inlines the “fft” function body. Fig. 4(c) shows the MCG after “fft” has been fully inlined, resulting in an MCG that contains no overlap between disparate cycles. After function virtualization is complete, simplification of the MCG can begin.

Cyclebite reduces complex subgraphs in the MCG to a single node. Serial chains like the subgraphs highlighted in Fig. 5(a) increase the size of the MCG without providing information about cyclical subgraphs. Fig. 5(b) shows these subgraphs after being reduced to a single node. Cycles whose body contains predication, like the highlighted subgraph in Fig. 5(b), have multiple execution paths that must enter through a single node and exit through a single node, making cycle localization difficult while providing no useful information about cycles. Fig. 5(c) shows this subgraph transformed into a single node. Additionally, when low-frequency cycles (whose iteration count is insignificant, below a threshold $= 16$) exist in the program, Cyclebite transforms their control flow to a single node. We believe that any cycle that recurs less than 16 times is executing a mundane task - allocating memory, initializing a static filter, or performing error checking. To preserve the semantics of the start and end of the MCG, each transform is not allowed to eliminate the beginning and ending nodes (the beginning node has zero predecessors, and the ending node has zero successors). When no transform opportunities are left, the MCG is called a *simplified* MCG and is ready for its CGT candidates to be localized.

C. Cycle Localization

Cyclebite uses a modified version of Dijkstra’s shortest path algorithm (which projects edge weights onto a negative log space to transform maximum probabilities to minimum probabilities) to localize cycles in an order that is determined by two cycle characteristics. First, cycles are evaluated for their entrance and exit edges: the cycles with the smallest sum of entrance and exit edges are localized first. Second, if a tie exists in the entrance/exit sum of two or more cycles, Cyclebite selects the cycle with the highest path probability. *Path probability* is the permutation of all edge weights that must be traversed to complete one revolution of a cycle - thus, cycles with the highest probability of recurring are prioritized. Cyclebite uses these rules to order the localization of cycles when cycle overlap exists.

Cycles overlap in the MCG as a consequence of nesting. The ordering of cycles in a cycle nest matters because each cycle serves a unique purpose - indexing dimensions in the memory working set of the cycle nest, carrying out reductions, or mapping orthogonal dimensions onto a transformed space. Cyclebite prioritizes the inner-most (child) cycles first, as highlighted in Fig. 6(a) - these are the cycles commonly found with one entrance and one exit to its parent cycle only. Fig. 6(b) shows the result of Cyclebite’s first iteration of localization: each child-most cycle has been transformed. The hierarchical relationship of each cycle is used to group them together as they are localized. Fig. 6(c) shows the final iteration of localization. Though not explicitly shown in Fig. 6, Cyclebite applies simplifying transforms to the MCG between each iteration. Cycle localization executes iteratively until no eligible cycles are left. Each localized cycle hierarchy becomes a *CGT candidate* - a group of basic blocks that forms at least one cycle, has an iteration count of at least 16, and boundaries (called critical edges) unique to all other CGT candidates. The Epoch Profile (EP) uses the critical edges of each CGT candidate to localize the epochs of the program and discover communication between them.

D. Epoch Profile

EP evaluates the utilization of CGT candidates and discovers memory dependencies between code sections of the target application by localizing each *epoch* (a period of time in which the program executes a distinct section of its code) with a dynamic profile. When discovering memory dependencies between the tasks of an application, the information that maps a memory transaction to its task at the time of its observation is not readily available. While the target program is executing, **EP increments the current epoch every time the executing program encounters a *critical edge*** (a basic block transition that either enters or exits a CGT candidate). Epochs may denote the instance of a CGT or non-CGT code; a CGT candidate instance is an epoch, but an epoch is not necessarily a CGT candidate instance. EP observes the utilization of each CGT candidate instance by recording its *local frequency* (the frequency of the member basic block with the highest frequency in that epoch). CGT candidates also have a *global frequency* (the sum of the local frequencies of all instances of a CGT candidate). A CGT candidate with high local frequency has significant utilization when it executes - Cyclebite upgrades this CGT candidate to a CGT. A CGT candidate with high global frequency but low local frequency indicates low utilization each time it executes - Cyclebite rejects it.

EP injects backend calls at the start of each basic block to profile state transitions during the program's execution. Like MP, EP transforms the target program before instrumentation - basic blocks with context-switching instructions (like function call and return) only have that instruction. EP increments the current epoch when it encounters a critical edge. During each epoch, like those shown in Fig. 7(a), EP records each member basic block's frequency count, which determines the local frequency of the epoch. If the epoch refers to a CGT candidate and its local frequency is greater than a parameter (parameter = 32), EP designates that CGT candidate as a CGT. We chose to set the parameter to 32 because a task with fewer than 32 operations likely does not justify the overhead cost of running it on an accelerator. The resulting structure is a directed graph: epochs (which may or may not refer to a CGT) are nodes, and temporal relationships are edges pointing toward increasing time. EP uses the temporal ordering of epochs to evaluate the communication patterns between CGT and non-CGT epochs.

E. Communication Extraction

EP discovers communication between epochs by recording the memory transactions of each epoch as they execute and

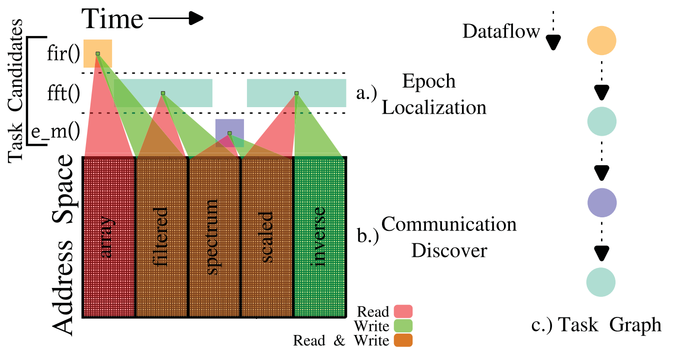


Fig. 7. The Epoch Profile (EP) localizes the *epochs* of the target application by observing the unique entrances and exits (called *critical edges*) of the CGT candidates extracted from the simplified Markov Control Graph (MCG). An epoch is a period of time in which the program executes a distinct section of its code, and may or may not be a CGT candidate instance. At each critical edge, EP increments the current epoch of the application (a). As the epoch executes, EP observes its *local frequency* (the frequency of the member basic block with the highest frequency) and its memory transactions (load, store, memset, memmov, etc.). If the epoch is a CGT candidate instance, and its local frequency is sufficient, that CGT candidate becomes a CGT. CGT candidates whose instances have high local frequency require significant work when called, making them good accelerator candidates [29]. Once all epochs have been localized, their memory transactions are used to discover producer-consumer relationships (b). The resulting structure is a task graph (c) - each node is an epoch (which may or may not be a CGT), and each edge is a read-after-write (RAW) dependency.

finding intersections between the input working set of the current epoch and the output working sets of its predecessor epochs in reverse-temporal order. Localized CGTs are only part of a program's structure - communication between these CGTs constrains concurrent execution, scheduling, and memory locality. EP stores the addresses from both load and store instructions, and memory operations (like memset and memov), executed during each epoch. To better manage the memory usage of EP, it implements a lossless compression scheme (called *memory tuples*) where contiguous memory accesses are combined into one entry that represents the memory footprint of the combined memory accesses. Once execution is complete, EP discovers communication relationships between epochs by overlapping touched memory addresses of a given epoch with the touched memory addresses of its temporal predecessors (in reverse-chronological order, i.e., most-recent predecessor first). Using the temporal ordering of epochs, EP ensures that each intersection is the true dependency of that intersection, called the "last writer" [32] of that memory. Thus, EP forms a *task graph*: program epochs are nodes, and directed edges (that point from producer to consumer) are the communication between epochs.

During execution, EP records read-from and written-to addresses inside each epoch, shown in Fig. 7(b). After execution, EP discovers communication between epochs by walking the epoch history in reverse-chronological order. For each predecessor of the epoch under evaluation (the current epoch), EP overlaps the written-to addresses of the predecessor with the read-from addresses of the current. Any overlap "explains" that portion of the read-from addresses of the current epoch - this forms a RAW relationship. Before evaluating the next predecessor, each explained read-from address is removed from the current epoch. This process is repeated until one of two conditions is reached: all read-from memory addresses are explained, or no predecessors remain. Once all RAW relationships between tasks are discovered, the task graph of the program is complete, shown in Fig. 7(c). In the next section, we demonstrate Cyclebite's structure extraction capability in a real-world example.

IV. Extracted Structure Study

We demonstrate Cyclebite's ability to extract a comprehensive pipeline of parallel tasks from CFTW with a case study. The input program is a working example of CFTW: a Harris corner detector from an open-source C++ computer vision library that utilizes modular function reuse, templated objects, function pointers, and loop body predication. Our case study shows Cyclebite automatically extracts parallelism from CFTW despite these challenges. Fig. 8 presents the results of the program structure extracted by Cyclebite and is discussed below.

A. Methodology

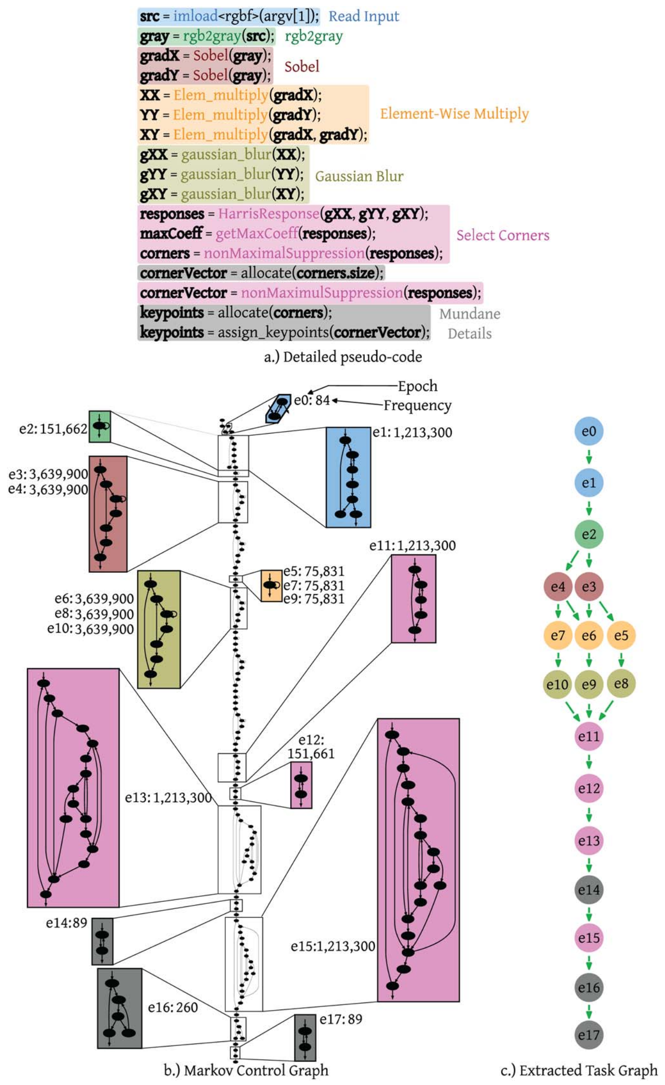
We selected the Harris corner detection algorithm driven by NVision [33], an open-source C++ computer vision API, as the application for our structuring case study. The NVision library is a comprehensive example of the programming style of CFTW. The author utilized C++ primitives to modularize the code for many use cases (including object templating, exception handling, and function overloading, which bloat the static code with unused templated objects; and function indirection, which creates holes in the static structure of the program). Additionally, Harris utilizes a few general operations (convolution, Sobel, matrix searching) found widely throughout signal and image processing, making it a good example of applications from these domains.

LLVM9.0.1 was used with compiler flags with -O3 -g3 to compile the target application. Full debug symbols mapped the extracted tasks back to the source code. Cyclebite structured the program on an idle machine with dual Intel Xeon E5-2650 processors and 256GB of memory.

B. Structure

Cyclebite extracts a comprehensive structure of parallel CGTs from the Harris corner detector. Fig. 8 shows Cyclebite's results. Fig. 8(a) is a pseudo-code program describing each CGT extracted by Cyclebite. Each line has a function name (which describes what the CGT does), arguments (which indicate what the dependencies of that CGT are), and a returned variable (which shows the data it produces). Fig. 8(b) is the simplified MCG constructed by Cyclebite with CGT candidates highlighted. Fig. 8(c) is the task graph extracted from the target application by Cyclebite with non-CGT epochs omitted for simplicity. Nodes are CGT instances and edges are communication patterns between them.

Subfigures in Fig. 8 relate to each other in three different ways. First, each pseudo-code line in a), CGT candidate in b) and CGT instance in c) is color-matched (by an expert inspection of the code underneath each CGT) to its general



```
src = imload<rgbf>(argv[1]);
gray = rgb2gray(src);
gradX = Sobel(gray);
gradY = Sobel(gray);
XX = Elem_multiply(gradX);
YY = Elem_multiply(gradY);
XY = Elem_multiply(gradX, gradY);
gXX = gaussian_blur(XX);
gYY = gaussian_blur(YY);
gXY = gaussian_blur(XY);
responses = HarrisResponse(gXX, gYY, gXY);
maxCoeff = getMaxCoeff(responses);
corners = nonMaximalSuppression(responses);
cornerVector = allocate(corners.size);
cornerVector = nonMaximulSuppression(responses);
keypoints = allocate(corners);
keypoints = assign_keypoints(cornerVector);
```

Fig. 8. Cyclebite extracts a comprehensive structure of parallel CGTs in code-from-the-wild (CFTW). Cyclebite structured a 3-line application implementing the Harris corner detection algorithm using NVision [33]. Subfigure a) uses pseudo-code to describe the CGTs extracted from the target application, each color-matched to its general operation (e.g., "Sobel"). Cyclebite found all five general operations in the target application. Subfigure b) Shows the simplified Markov Control Graph (MCG) with each CGT candidate highlighted and color-matched to its general operation and annotated with its epochs and their local frequencies (e.g., "e0: 84" is the only epoch of the first task candidate, which reads an input image, and has a local frequency of 84). Subfigure c) is the task graph extracted by Cyclebite with non-CGT epochs omitted for simplicity. Each node is a CGT instance labeled with its epoch name (e.g., "e0" maps to the first CGT candidate in subfigure b) and is color-matched to its general operation. Cyclebite discovered parallelism in the calls to Sobel, Elem_multiply and gaussian_blur.

operation (e.g., "Sobel", "gaussian_blur") to demonstrate the components of the Harris processing pipeline we would expect to capture. Second, each pseudo-code line in a) maps 1:1 with a node in c) and describes the work done in its corresponding CGT instance. Third, each epoch ID and its local frequency (e.g., "e0: 84") that annotates a highlighted CGT candidate in b) maps to a node (with that label) in c) to demonstrate why that CGT candidate was selected as a CGT (its frequency was greater than 32).

Cyclebite detects parallel tasks in CFTW. Each CGT candidate in Fig. 8(b) with multiple annotated epochs ran that many times, and in each of these cases (Sobel, Elem_multiply, gaussian_blur) Cyclebite found parallelism in those instances. Fig. 8(a) shows the parallelism between these CGT instances with the variables produced and consumed by each instance. Fig. 8(c) is the task graph that results from this detected parallelism - the task graph shows which epochs depend on which, how many stages are in the extracted task graph, and which stages have parallel tasks.

Cyclebite extracts the *executed* structure of its target applications, not the *intended* structure. The *executed* structure is the structure implemented by the programmer (and may include discretionary or unnecessary steps), whereas the *intended* structure is the structure that contains all operations required to execute the desired algorithm. CGTs labeled “mundane details” and colored dark in Fig. 8(a) were not considered essential to the Harris algorithm after expert inspection of their underlying source code. The three dark-colored epochs (e14, e16, e17) pre-allocated, sorted and structured the corners found by the algorithm (which is not required by Harris).

In the next section, we demonstrate the potential speedups and scalability that Cyclebite’s extracted structure makes available, and compare its extracted structure to that of two SoA techniques: HC and HL.

V. Results

Cyclebite extracts a parsimonious structure from CFTW compared to SoA techniques that leads to profitable speedups. We demonstrate the potential performance benefits that Cyclebite’s automatic task graph extraction makes possible with two experiments. First, we show the performance improvements task graphs make possible when mapping a small corpus of SDR applications to a domain-specific system-on-chip (DSSoC) using the DS3 [34] simulation platform. Second, we use Halide to “hand-compile” extracted task graphs from a corpus of image processing algorithms and demonstrate their performance scalability with thread count. Each experiment generates a proxy result for the optimizations one could expect from Cyclebite-extracted task graphs if an automated downstream optimization and compilation pipeline existed, and the results are a guide for expectations of future results. We compare the extracted structure of Cyclebite with two SoA structuring techniques: HotCode (HC) and HotLoop (HL). Our results show that Cyclebite captures the important codes from CFTW, while HC and HL miss important codes and capture the wrong ones. Finally, we show that Cyclebite is robust to a broad cross-section of CFTW, and the overheads incurred by its toolchain are manageable.

A. DSSoC Speedup

1) Methodology: We use Cyclebite to structure a basket of SDR applications and map the extracted task graphs to a DSSoC. The task graph extracted by Cyclebite provides information for accelerator candidates and parallelism. Using the DS3 architecture simulation platform, we generate performance estimates for 5 SDR applications listed in Table I. We use two versions of each application: a “baseline” version and a “performance” version. The difference between the simulated runtimes of each application type yields a speedup value made possible by the extracted application structure.

TABLE I
Application Corpus Structured by Cyclebite and Mapped to the DS3 [34] Simulation Framework

Application	APIs
Temporal Mitigation (TempMit)	-
Radar Correlator (Corr)	GSL
Single-Carrier Receive (SCR)	GSL & FFTW
Single-Carrier Transmit (SCT)	GSL & FFTW
Wifi Transmit (Wifi_Tx)	GSL & FFTW
Wifi Receive (Wifi_Rx)	GSL & FFTW

TABLE II
Performance Scaling of Applications Simulated With DS3. Each Application Is Structured into Tasks With Communication Patterns

Application	Baseline [μs]	Performance [μs]	Speedup
TempMit	8,300	876	9.5
Corr	6,360	961	6.6
SCR	21,700	924	23.5
SCT	1,570	827	1.9
Wifi_Tx	16,779	881	19.0
Average			12.1

DS3 simulates the Dash DSSoC, a domain-specific architecture built for SDR applications like communication and radar. To simulate a Cyclebite task graph in DS3, an expert selects the CGTs (extracted by Cyclebite) that describe the application and annotates each one with its communication patterns with other CGTs (from the Cyclebite task graph) - precisely what Cyclebite automates. The CGTs DS3 supports are a significant fraction of the runtime of each application, though they are not comprehensive in the tasks that are required to make real-world applications work. The simulation results are a reasonable estimate of the optimized performance of each application.

DS3 simulates each application in two ways. The baseline version implements the application on a single-core scalar processor, executing each task in the application serially. The performance version implements the application on a 4-core processor with two GEMM, two FFT and two Viterbi accelerators using an integer linear programming (ILP) schedule [35]. Both versions are run 10 times back-to-back - each time listed in Table II is the time it took to run all 10. The speedup obtained by the performance version over the baseline version represents the speedup possible when structuring CFTW and optimizing it based on that structure.

2) Applications: We used a corpus of applications found in software-defined radio (SDR). Since DS3 simulates an architecture built for this domain, we naturally used a corpus of target programs for that architecture. Table I shows these applications. The open-source libraries used in the corpus are ubiquitous among SDR applications from the wild. FFTW implements the fast Fourier transform (FFT) and GSL implements general matrix-matrix (GEMM).

3) Speedup: Cyclebite’s extracted task graphs enable significant speedups on the simulated DSSoC. Table II shows speedups of 12.1 on average (arithmetic mean) of the performance applications versus their baseline counterparts. All applications performed better in their performance implementations than the serial ones, indicating that Cyclebite extracted an effective structure from each application. The larger the application, the more speedup it had due to the application-wide optimization opportunities task graphs make available. While some applications experienced significant speedups (like Single-Carrier Receive (SCR)) because of their high degree of parallelism and mappability to dedicated hardware, other applications saw low speedup (Single-Carrier Transmit (SCT)) because of a lack of parallelism in their task graph.

TABLE III
Algorithms for Thread Scalability

Algorithm	Description
SGEMM	Single-precision matrix-matrix multiply.
IIRBlur	Two-tap linear infinite-impulse response image filter implemented column-wise in two passes. The first pass is along the columns, both down and up. The second pass is along the columns of the transposed result of the first pass, both down and up.
Harris	Harris corner detection.
Stencil Chain	A 5-length chain of 5x5 linear image filters with discrete approximations of a Gaussian filter with mean 0 and variance 1, implemented with floating-point precision.
SIFT	Scale-invariant feature-transform

B. Thread-Scalability

We demonstrate the thread scalability of task graphs by translating applications structured by Cyclebite to Halide. The Halide DSL relies on a program abstraction that closely resembles the structure extracted by Cyclebite. Once this structure is extracted from CFTW and made available to Halide, it compiles and exports an optimized version of the application whose performance scales with thread count. We selected linear algebra and image processing algorithms as the input corpus for their task-oriented, communication-driven structures. Using Halide to implement extracted task graphs from Cyclebite leads to strong thread-scaling of program performance, improving application runtimes by 24x on average with 16 threads compared to SoA auto-parallelization tools.

1) Methodology: We used the Halide v10.0.0 domain-specific language [3] as our proxy for a future Cyclebite optimization framework. It specializes in image-processing algorithms, which present various parallel optimization opportunities. Halide's framework requires a programmer to structure the input program statically and explicitly with a 1-to-1 mapping between functions and tasks and using polyhedral iterator spaces to define communication patterns clearly - the same structure exported by Cyclebite. Thus, Halide's optimization framework receives the same information from Cyclebite as it would from the programmer.

Static LLVM IR bitcode archives give Cyclebite and LLVM-Polly all the necessary visibility into the function symbols of the program to extract structure. We compiled test programs statically to maximize visible function symbols at compile time. Naive programs are compiled statically into LLVM IR bitcode. We compiled library-driven programs with static LLVM IR bitcode archives, making all their function symbols visible to Cyclebite. For Halide programs, we compiled them into LLVM IR bitcode first, then to the target architecture.

The scalability study has the following design. Each program has five versions: Naive, Naive + Polly, Library, Library + Polly, and Cyclebite + Halide. For each application in Table III, we constructed three versions: a naive version, a library version, and a Cyclebite + Halide version. Novice programmers wrote the Naive version and used no external dependencies. Each library program used an external dependency (GSL for SGEMM and OpenCV for all other algorithms) and consisted entirely of API calls. Each Cyclebite + Halide program was either written by an expert or was taken from the Halide GitHub repository [36]. Naive and Library programs have two versions: a non-LLVM-Polly configuration, which compiled the program with -O3 -g0 compiler flags (using LLVM9.0.1), and an LLVM-Polly optimized version ("+ Polly"), which enabled LLVM-Polly to optimize each program in addition to the regular optimizer (-O3 -g0). LLVM-Polly can make all speculative code optimizations and OpenMP code generations where it sees fit.

Each scalability study ran on an idle machine with dual Intel Xeon E5-2650 processors and 256GB of memory. Each application data point had 225 runtime samples. We conducted 15 trials of 15 samples; each trial is the arithmetic mean of 15 samples. The final data point is the median of the trials.

2) Applications: We focus on applications from linear algebra and image processing. Halide works best with these applications. We chose GSL v2.5 and OpenCV v4.6.0 for library applications for their large open-source communities and infrastructures. Each library has hand-tuned API calls that achieve state-of-the-art performance for linear algebra and image processing algorithms. Each algorithm is listed with a brief description in Table III.

Cyclebite automatically scales to new applications - hand-tuned APIs require manual hand-tuned scaling. For design efforts that require optimization, Cyclebite speeds up programs with little overhead. OpenCV and GSL have to manually hand-tune high-performance implementations when introduced to new algorithms, incurring significant time and effort.

3) Scalability: The scalability of each extracted task graph shows Cyclebite extracts parallel CGTs in its exported structure. Fig. 9 shows the thread scalability achieved by the Halide implementation of each application. Except for SGEMM, each Halide speedup curve is roughly linear for most thread counts - indeed, the CGTs found were parallel kernels.

Speedups achieved from hand-compiled Cyclebite task graphs come from both parallelism and memory locality. The task graph extracted by Cyclebite provides delineated tasks that are blocked and vectorized when possible, and parallelism between tasks that run concurrently. Additionally, the working set size, computation order and re-use, and task fusion all made available by the task graph help optimize the memory usage of the algorithm, increasing performance beyond the thread-scaling limit in some cases.

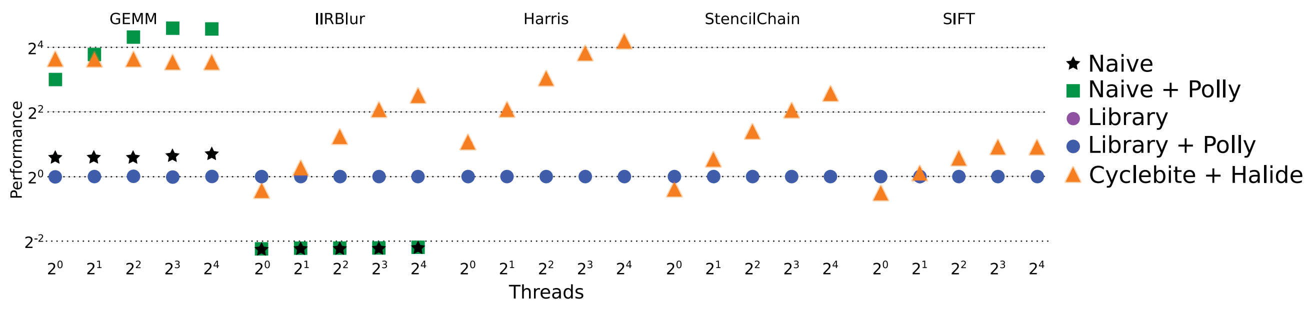


Fig. 9. Thread-scalability for each application in Table III: single-precision general matrix-matrix (SGEMM), linear infinite impulse response image filter (IIR Blur), Harris corner detection algorithm (Harris), a chain of stencil kernels (StencilChain) and scale-invariant feature transform (SIFT). Each application has five types: naive, which contains no external dependencies and is written by a non-expert; library, which uses GSL (for SGEMM) or OpenCV with threading enabled (for IIR Blur, Harris, StencilChain and SIFT); Cyclebite + Halide, that models the scalability of the task graph extracted by Cyclebite from the Naive version of the application; Naive + Polly, which is the naive implementation optimized by LLVM-Polly; and Library + Polly, which is the library implementation optimized by LLVM-Polly. Each data point is normalized by the performance of Library + Polly with one thread.

As CGT counts and communication complexity increase, the speedup results improved in favor of Cyclebite + Halide. When applications contain few CGTs (with SGEMM and IIRBlur), the API implementation is competitive with the Halide proxy for small thread counts. Intuitively, the API calls are *hand-tuned models* for the input; they used specific primitives for vector and parallelization opportunities for that application. The Halide implementation has a *generated model* for the application, which lacks implementation details. In other words, the API call was hand-tuned by experts, but Halide is just a compiler. Thus, for simple workloads, we expect the API to do well, and in this case, better than Cyclebite + Halide. More complex applications (Harris, StencilChain, SIFT) saw Cyclebite + Halide trounce the competition as thread count increased. Cyclebite + Halide finds optimizations that exploit parallelism and locality *between* CGTs, leading to more profitable speedups than programs that do not. The API implementation does not have information about the broader application; it is stuck optimizing *within* tasks.

LLVM-Polly could not recognize opportunities for speedup in an application beyond the SGEMM example, even with all speculative flags turned on. While it achieved the top performance mark for SGEMM, likely because of its superior vectorization capability, its programs were no different from the regular-optimization version. After analyzing the static control parts, or statically-defined polyhedral loops eligible for optimization within LLVM-Polly's framework, the requirements appeared to be more stringent than just static loop bounds; it required clearly-defined affine loop iterators and determinable functions within loops.

Mapping task graphs extracted from Cyclebite to DSSoC architectures and multi-threaded implementations leads to impressive performance improvements and thread scaling. First, the task graphs extracted by Cyclebite are comprehensive in their structure, as shown by 12.1x speedups of task graphs simulated on a DSSoC compared to their serial equivalent. Second, Cyclebite extracts parallel task graphs, as shown by an average of 24x speedup for a 16-thread implementation of a hand-compiled task graph compared to SoA auto-parallelization tools. Next, we evaluate the code Cyclebite structures from CFTW, and the code Cyclebite rejects as unimportant.

C. *Correspondence to SoA*

Extracting a structured task graph representation requires a precise selection of basic blocks. Including too many basic blocks will result in false or bloated tasks. Including too few basic blocks will result in missing or incomplete tasks. We examine when three techniques (HotCode, HotLoop, and Cyclebite) agree on basic blocks selected for inclusion in the extracted structure of CFTW across the applications in Dash-Corpus.

1) SoA Techniques: SoA structuring techniques structure the target program around the code that runs most frequently, known as HotCode (HC). HC represents the parts of the program that require the most time to complete. Naturally, program optimization techniques focus on reducing the time to complete these tasks first. To construct a proxy for this strategy, we built a structuring technique that measures the frequency of basic blocks and designates as tasks all basic blocks whose frequency explains 95% of the total basic block frequency in the program. However, this strategy can lead to results that are not contiguous: hot blocks that exist by themselves spread throughout the program that, when standing alone, are difficult to optimize.

An extension to HC is HotLoop(HL), which incorporates the static loops of the program into HC. HL designates as a task any statically-defined loop with at least one hot basic block. HL is a generous approximation of the static control parts (SCoPs) LLVM-Polly [16] uses to structure programs - LLVM-Polly's performance in Section V.B.3 provides insight into the performance benefits one can expect by using HL as their structuring method. It structures hot basic blocks with their cold constituents (e.g., outer loops that may be important for optimization). Using these two structuring techniques, we compare their results across all applications in Dash-Corpus to Cyclebite and discuss their similarities and differences.

2) Methodology: We measure the basic blocks Cyclebite (Cb) accepts and rejects as members of a CGT and compare them to tasks selected and rejected by HC and HL. HC structures all hot basic blocks in an application (all basic blocks whose frequencies, sorted from greatest to least, explain at least 95% of all basic block frequencies). HL structures all basic blocks that belong to a statically-defined loop with at least one hot member basic block. Simply put, HC represents all hot basic blocks in Dash-Corpus, and HL represents all basic blocks that belong to a statically-defined loop in Dash-Corpus.

We ran Cb, HC, and HL on 2,019 applications in Dash-Corpus. Comparing these three techniques - HC, HL, Cb - yields seven categories: HC only, HL only, Cb only, HC and HL, HC and Cb, HL and Cb, HC and HL and Cb. Each category is mutually exclusive of all others (e.g., "HC" contains only the basic blocks exclusively structured by HC; "HC & HL & Cb" contains only the basic blocks structured by all three) and collectively exhaustive of the basic blocks selected as members of tasks.

Structuring CFTW is a balancing act - structure too much code and the programmer has to parse through non-tasks; structure too little code and the programmer has to find tasks manually. Either of these scenarios will lead to manual intervention to make speedups possible. Indeed, SoA techniques like HC and HL find meaningful tasks, but they miss code that should be tasks and capture code that shouldn't be tasks. Thus, Cyclebite both accepts and rejects code that HC and HL structured.

We used a cluster of 11 servers. To batch jobs to the entire server, the SLURM workload manager distributed bash scripts to all machines. Each server runs Ubuntu-server 18.04, two Xeon E5-2650 processors with ten physical cores each and 256GB of RAM available. We used LLVM9.0.1 to compile each program. All debug symbols are on, and the compiler optimizer is at its lowest setting (-g3 -O0).

3) Correspondence: Cyclebite automates significant work required to structure CFTW. Fig. 10 shows each of the correspondence categories between Cb, HC and HL. The differences between categories in Fig. 10 are large, indicating that SoA techniques both miss and include a significant amount of task and non-task code. Only the colored regions are categories in the Venn - the white region is not a category. Each magnitude represents the number of basic blocks belonging to that category.

brown: cold code within static loops rejected by Cyclebite. Unused basic blocks dominate this category, and Cyclebite can trivially reject code that never ran with the dynamic information from MP. Further, statically-defined loops may contain modular operations that themselves are hot, upgrading the entire (cold) loop to a hot loop. This cold loop is not a task in the application. Cyclebite rejects this code through its function inline transform.

blue: hot code not in a static loop and rejected by Cyclebite. Highly reused modular functions contain serial code that becomes hot when tasks reuse it throughout the application. Further, EP finds cycles within these functions with a low frequency of recurrence. Cyclebite rejects these loops.

red: hot code within static loops that Cyclebite rejects. HL structures statically defined loops that recur a few times if they contain hot basic blocks. HL often structures low-frequency loops when highly-reused modular functions are members of a loop. Cyclebite rejects this code through its function inline transform.

olive green: code exclusively structured by Cyclebite. This category contains code missed by SoA techniques. Any task

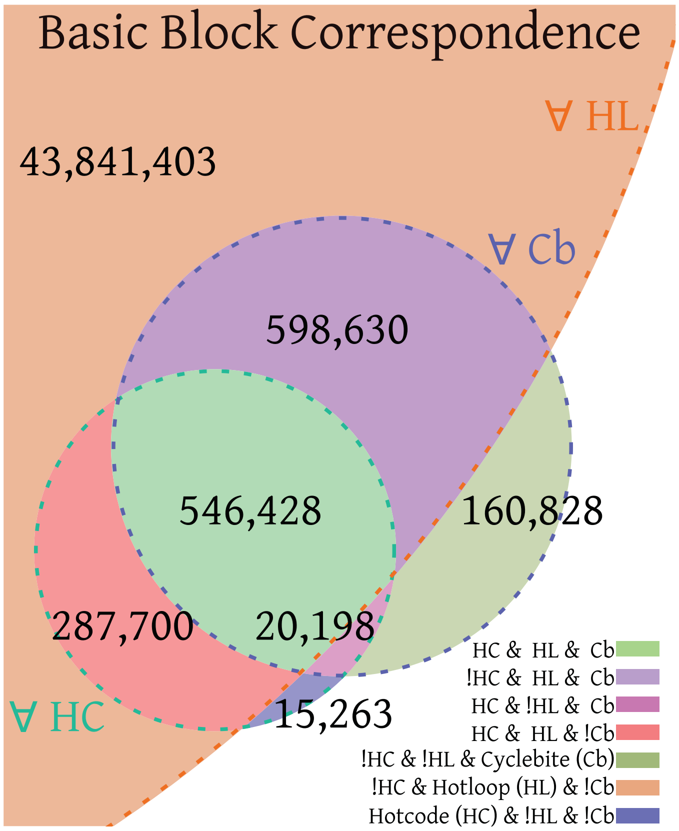


Fig. 10. A demonstration of which basic blocks are accepted as task members and rejected from being task members, according to Cyclebite (Cb), HotCode (HC), and HotLoop (HL). HC and HL are proxies of two state-of-the-art structuring techniques. All basic blocks structured by HC are hot, and all basic blocks structured by HL belong to a statically-defined loop with at least one hot member basic block. Each magnitude is the basic block count of that region. Each region is exclusive to all others and collectively exhaustive of all basic blocks structured in Dash-Corpus. For example, "HCHL" comprises the basic blocks only structured by HC and HL. For every 100 basic blocks executed in a Markov Profile in Dash-Corpus, HL structured 340 blocks (because of unused blocks), Cb structured 10, and HC structured 6.

structured by SoA techniques that uses function pointers will not contain the code called by that function pointer. But MP resolves that function, and Cyclebite captures its code. Additionally, low-frequency tasks outside of a hot task, like the outer loops of a matrix multiply, may be rejected by SoA techniques for lacking hot code.

pink: hot code outside a static loop accepted by Cyclebite. Recursive algorithms with highly-recurrent structures give rise to hot basic blocks without being contained within a traditional loop structure. Furthermore, HC and HL miss function pointers within highly-recurrent loops that Cyclebite captures. Thus Cyclebite finds hidden, complete tasks compared to SoA methods.

violet: cold code within a static loop that Cyclebite accepts. Loop bodies often contain predication - branches that occur during the execution of a loop iteration - that biases the execution of the loops toward one side of the fork. Thus, the often-used code becomes hot, and the seldom-used code becomes cold. HL and Cb agree on these blocks because they capture the entire loop.

green: hot basic blocks within static loops accepted by Cyclebite. All techniques agree this code should be a task. Code

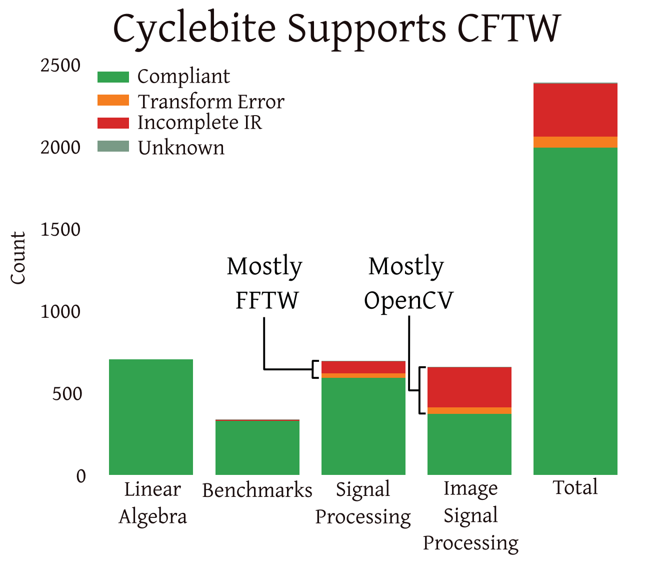


Fig. 11. Cyclebite is compliant with 84.5% (2,019) of the 2,390 applications in Dash-Corpus. An application is *compliant* with Cyclebite if Cyclebite extracts the task graph from that application without error, both known or unknown. Green ("Compliant") represents compliant applications. Orange ("Transform Error") is a known issue within the transforms applied to the MCG. Red ("Static Structure Ambiguity") represents applications with function symbols not compatible with LLVM IR bitcode, creating blind spots in the Markov Control Graph (MCG). Gray ("Unknown") comes from applications whose MCG had a known problem (like having more than one start node), but the cause of this problem could not be determined.

commonly found in this category are inner loops of matrix multiply, convolution, correlation, FFT, and others.

D. Robust

We evaluated Cyclebite's robustness with applications from Dash-Corpus. Robustness is Cyclebite's ability to extract a structure that both makes sense and is error-free, both known and unknown. Input applications that make Cyclebite segfault are not compliant. Further, if Cyclebite hits a known condition in its flow of transforms that leads to a bad result, then Cyclebite is non-compliant with the program. Once Cyclebite performs all stages of its pipeline and produces a result, Cyclebite is compliant with that application.

The Cyclebite toolchain was compliant with a large cross-section of CFTW. Fig. 11 shows Cyclebite's compliance with all Dash-Corpus programs. Cyclebite structured 84.5% (2,019) of 2,390 applications in Dash-Corpus. For each of these compliant programs, Cyclebite satisfied three requirements for compliance. First, it completed its execution without error (such as segfaults and other implementation-related issues). Second, the simplified MCG had exactly one starting node and explainable exit nodes (more than one exit node is possible when an inlined function contains the basic block that terminated the program). Third, the dynamic call graph of the program was a whole piece, and all nodes were reachable.

For non-compliant programs, holes in the static structure were the most prevalent cause. Holes in an application's profile come from function symbols not compilable into static LLVM IR bitcode (like the standard C library or special vector libraries produced by hardware vendors like Intel's Integrated Performance Primitives library). Because MP cannot instrument these parts of the program, MP is not able to observe the entire execution of the program, resulting in an incomplete profile. Non-compliant function symbols are abundant in FFTW and OpenCV applications.

E. Profiling Overhead

The median time dilations for MP and EP were 11.5 and 253, respectively, across 2,008 applications in Dash-Corpus. In prior work, SD3 [10], for single-threaded profiling like that of Cyclebite, suffered dilations of anywhere between 250x–850x. Thus, the dilation from Cyclebite's dynamic profiles is manageable compared to SoA memory profilers. If the dilation of the program resulted in an extension of the program by hours or days, shorter versions of the program solve this problem.

For EP, memory utilization is very low for a naive SGEMM example (2,776 bytes) - this example has contiguous memory accesses which makes the compression scheme (memory tuples) in EP efficient. In examples where memory accesses are not contiguous (for example, accessing a single member of a user-defined structure in an array of that structure) the compression scheme will not be effective, and memory usage will be much higher.

VI. Discussion

A. Limitations

The Cyclebite toolchain only supports C and C++ programs, limiting Cyclebite's scope and the number of function symbols visible to its profilers (e.g., libraries like Armadillo that utilize Fortran and Libc cannot be profiled completely). Assembly-only vector libraries like Intel's IPP library, used by OpenCV, are not compatible. While Cyclebite's analysis is blind to these functions, non-compliant function symbols can be manageable by leveraging their nomenclature (e.g., libc::qsort() implements qsort).

Cyclebite only sees code that executes. However, MP can trivially give each statically-defined edge a minimum frequency of 1, guaranteeing each edge in the program is seen by the analysis.

Basic block recurrence implies functional recurrence, but functional recurrence does not necessarily imply basic block recurrence. Libraries like FFTW and Spiral generate algorithm code by partially or entirely unrolling the execution. Thus, cyclical behavior in the computation may not be represented by cyclical behavior in the basic blocks of the program.

Cyclebite ignores all user-space codes executed before and after main(). Cyclebite's profilers start right before main() begins and end right after main() returns. These constraints on the boundaries of dynamic profiles ensure the DCFG is a complete piece.

B. Wider Impacts

The performance numbers Cyclebite showed in its evaluations are scalable to highly parallel applications from linear algebra, graphics, software-defined radio, cryptography, and graph applications.

Cyclebite can support languages and platforms beyond C/C++ (like Python, Fortran and Node.js) with significant engineering effort.

Downstream optimization tools will use the exported task graph to automatically optimize the structure, schedule, and communication of the program for target platforms, parallelism, and memory locality.

VII. Conclusion

Cyclebite is an open-source toolchain [11] that structures wild programs into its CGTs with communication patterns. It is compliant with a large cross-section of open-source C and C++ libraries. Using a dynamic execution profile of a program, it finds CGT candidates within the program by simplifying its dynamic execution graph and localizing the highest-probability cycles in the program. Finally, it localizes CGTs in the target application by localizing the epochs of the program, and observing communication patterns between those epochs, resulting in a task graph. Its exported structure delivers speedups over state-of-the-art optimization frameworks. Future work of Cyclebite will increase its compliance with other programming languages (like Python) and algorithms within and outside Dash-Corpus, and downstream optimization tools that will use the structure exported by Cyclebite to optimize target applications.

Acknowledgment

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusion contained herein are those of the authors and should not be interpreted as necessarily representing the social policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

The authors thank Liangliang Chang, Mukul Gupta, Vamsi Lanka, and Sriharsha Uppu for writing input applications, and Richard Uhrie for building TraceAtlas [28], the predecessor to Cyclebite.

References

[1] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Proc. Int. Conf. Compiler Constr.*, Berlin, Germany: Springer, 2002, pp. 179–196.

[2] K. J. Brown et al., "A heterogeneous parallel framework for domain-specific languages," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, Piscataway, NJ, USA: IEEE, 2011, pp. 89–100.

[3] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, New York, NY, USA: ACM, 2013, pp. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[4] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic fine-grain scheduling of pipeline parallelism," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2011, pp. 22–32.

[5] A. Adams et al., "Learning to optimize halide with tree search and random programs," *ACM Trans. Graph. (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.

[6] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "TACO: A tool to generate tensor algebra kernels," in *Proc. 32nd IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, 2017, pp. 943–948.

[7] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 1–11, Jul. 2016. Accessed: Jan. 14, 2021. [Online]. Available: https://doi-org.ezproxy1.lib.asu.edu/10.1145/2897824.2925952

[8] J. Li, Y. Chi, and J. Cong, "Heterohalide: From image processing DSL to efficient FPGA acceleration," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, New York, NY, USA: ACM, 2020, pp. 51–57. [Online]. Available: https://doi.org/10.1145/3373087.3375320

[9] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *Proc. Int. Conf. High Perform. Comput., Netw. Storage Anal. (SC)*, Washington, DC, USA: IEEE Comput. Soc. Press, 2012, pp. 1–11.

[10] M. Kim, N. B. Lakshminarayana, H. Kim, and C.-K. Luk, "SD3: An efficient dynamic data-dependence profiling mechanism," *IEEE Trans. Comput.*, vol. 62, no. 12, pp. 2516–2530, Dec. 2013.

[11] B. Willis, "Cyclebite," 2023. Accessed: Jul. 13, 2023. [Online]. Available: https://github.com/benroywillis/Cyclebite

[12] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill, "Lamport clocks: Verifying a directory cache-coherence protocol," in *Proc. 10th Annu. ACM Symp. Parallel Algorithms Archit.*, 1998, pp. 67–76.

[13] S. L. Min and J.-L. Baer, "Design and analysis of a scalable cache coherence scheme based on clocks and timestamps," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 01, pp. 25–44, Jan. 1992.

[14] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "GRAMPS: A programming model for graphics pipelines," *ACM Trans. Graph.*, vol. 28, no. 1, pp. 1–11, Feb. 2009. [Online]. Available: https://doi.org/10.1145/1477926.1477930

[15] "Halide tutorial lesson 12: Using the GPU," Dec. 2022. Accessed: Nov. 20, 2022. [Online]. Available: https://halide-lang.org/tutorials/tutorial_lesson_12_using_the_gpu.html

[16] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly—Polyhedral optimization in LLVM," in *Proc. 1st Int. Workshop Polyhedral Compilation Techn. (IMPACT)*, vol. 2011, 2011, p. 1.

[17] C. Lattner et al., "MLIR: Scaling compiler infrastructure for domain specific computation," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, 2021, pp. 2–14.

[18] V. A. Ying, M. C. Jeffrey, and D. Sanchez, "T4: Compiling sequential code for effective speculative parallelization in hardware," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Compu. Archit. (ISCA)*, Piscataway, NJ, USA: IEEE, 2020, pp. 159–172.

[19] G. Venkatesh et al., "Conservation cores: Reducing the energy of mature computations," *ACM SIGPLAN Not.*, vol. 45, no. 3, pp. 205–218, 2010.

[20] W. Zuo et al., "Accurate high-level modeling and automated hardware/software co-design for effective SoC design space exploration," in *Proc. 54th Annu. Des. Automat. Conf. (DAC)*, New York, NY, USA: ACM, 2017. [Online]. Available: https://doi.org/10.1145/3061639.3062195

[21] M. Minutoli et al., "Svelto: High-level synthesis of multi-threaded accelerators for graph analytics," *IEEE Trans. Comput.*, vol. 71, no. 3, pp. 520–533, Mar. 2022.

[22] "Getting started with Xilinx OpenCV in Vivado HLS," Jul. 2022. [Online]. Available: https://developer.ridgerun.com/wiki/index.php?title=Getting_Started_with_Xilinx_OpenCV_in_Vivado_HLS

[23] K. Asanovic et al., "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[24] R. Uhrie, "Automatic computational domain detection," Ph.D. dissertation, Ann Arbor, MI, USA: Arizona State Univ., 2021.

[25] "llvm::basicblock," Jul. 2022. Accessed: Nov. 20, 2022. [Online]. Available: https://llvm.org/doxygen/classllvm_1_1BasicBlock.html

[26] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 39, no. 4, pp. 49–57, Apr. 2004. [Online]. Available: https://doi.org/10.1145/989393.989401

[27] "llvm-cov—Emit coverage information," Jul. 2022. Accessed: Nov. 20, 2022. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-cov.html

[28] J. B. Richard Uhrie and C. Chakrabarti, "Automated parallel kernel extraction from dynamic application traces," 2020, *arxiv:abs/2001.09995*.

[29] J. Brunhaver, "A rose by any other name would run just as long," *Fosdem,* 2020. Accessed: Jun. 2, 2023. [Online]. Available: https://archive.fosdem.org/2020/schedule/event/a_rose_by_any_other_name/

[30] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” *ACM SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, 2007.

[31] S. P. Amarasinghe and M. S. Lam, “Communication optimization and code generation for distributed memory machines,” *ACM SIGPLAN Not.*, vol. 28, no. 6, pp. 126–138, Jun. 1993. [Online]. Available: https://doi.org/10.1145/173262.155102

[32] S. P. Amarasinghe and M. S. Lam, “Communication optimization and code generation for distributed memory machines,” in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1993, pp. 126–138.

[33] F. Meyer, “NVision,” 2023. Accessed: Feb. 10, 2023. [Online]. Available: https://github.com/Rookfighter/nvision

[34] S. E. Arda et al., “DS3: A system-level domain-specific system-on-chip simulation framework,” *IEEE Trans. Comput.*, vol. 69, no. 8, pp. 1248–1262, Aug. 2020.

[35] J.-F. Puget, “Constraint programming next challenge: Simplicity of use,” in *Proc. 10th Int. Conf. Princ. Pract. Constraint Program. (CP)*, Toronto, ON, Canada, Sep. 27–Oct. 1, 2004, Berlin, Germany: Springer, 2004, pp. 5–8.

[36] “Halide,” 2023. Accessed: Oct. 16, 2022. [Online]. Available: https://github.com/halide/Halide

Benjamin R. Willis received the B.S.E. degree in electrical engineering and the M.S. degree in computer engineering from Arizona State University, in 2019 and 2020, respectively. His thesis was on a dataset of kernels and machine learning of computation. He is currently working toward the Ph.D. degree in computer engineering with Arizona State University. His research interests include software ontology and a machine understanding of computation.

Aviral Shrivastava (Senior Member, IEEE) received the bachelor’s degree in computer science and engineering from IIT Delhi, India, in 1999, and the master’s and Ph.D. degrees in information and computer science from the University of California at Irvine, USA, in 2002 and 2006, respectively. He is currently a Professor and the Graduate Program Chair of the School of Computing and Augmented Intelligence, Arizona State University (ASU), Tempe, AZ, USA, where he leads Make Programming Simple Lab. His research interests include compilers and microarchitectures for many-core, heterogeneous, and accelerated computing, error-resilient computing, and cyber–physical systems.

Joshua Mack (Graduate Student Member, IEEE) is working toward the Ph.D. degree in electrical and computer engineering program with the University of Arizona. His research interests include reconfigurable systems, emerging architectures, and intelligent workload partitioning across heterogeneous systems.

Shail Dave (Member, IEEE) is working toward the Ph.D. degree with the School of Computing and Augmented Intelligence, Arizona State University. His research techniques and infrastructures enable efficient processing of critical applications like machine learning on hardware accelerators in agile and sustainable manner. His research is published and referenced in flagship conferences and journals in computing design automation, embedded systems, and computer architecture and invited to premier industrial forums.

Chaitali Chakrabarti (Fellow, IEEE) received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India, in 1984, and the Ph.D. degree in electrical engineering from the University of Maryland, College Park, in 1990. She is a Professor with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe. Her research interests include VLSI algorithm-architecture codesign of signal processing and communication systems and all aspects of low-power embedded systems design.

John Brunhaver is a Principal Computer Architect with Lemurian Labs. He was an Assistant Professor with Arizona State University, where he researched an ontological approach to hardware and software codesign for domain specific systems on chip and radiation hardening of safety-critical microprocessors through procedural hardware generation. His Stanford University Ph.D. thesis, “The design and optimization of a stencil engine,” examines the virtual machine model for image processing and understanding domain-specific processors.