NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2026-24

June, 2026

Automating Parallel Execution Plan Selection and Tuning

Yian Su

Abstract

Modern hardware is parallel and is increasingly relying on parallel programming to achieve both performance and energy efficiency. However, today's parallel programming places a disproportionate burden on developers, requiring them to design and fine-tune parallel execution strategies across diverse hardware platforms and varying program inputs. I postulate that optimizing compilers can relieve developers from this burden by automatically reasoning about parallel execution strategies while also enforcing correctness properties. Achieving high parallel performance across modern hardware requires selecting an effective parallel execution plan, including how and where parallelism should be expressed and implemented. To automate this selection process, I introduce the Parallel-Semantics Program Dependence Graph (PS-PDG), a compiler abstraction that captures developer-encoded parallel semantics alongside compiler-derived analysis. PS-PDG enables compilers to override suboptimal execution plans with superior alternatives. While static execution-plan selection is effective for regular workloads, irregular and input-sensitive applications require runtime adaptation of parallel granularity. To address this challenge, I introduce the Heartbeat Compiler (HBC), a compilation system that automatically translates high-level fork-join parallelism into binaries capable of dynamically controlling task granularity through heartbeat scheduling to eliminate the need for manual task-size tuning. Beyond performance, C++ applications also face severe memory-safety risks arising from operations on data collections, including iterator invalidation. These bugs are difficult for existing tools to detect precisely and impose undefined behavior and security vulnerabilities in

production environments. To address this challenge, I introduce Ledger, a data collection-oriented static analyzer that provides high-precision detection of invalidation vulnerabilities in programs that operate on complex data collections.

Keywords

Compilers, Compiler Abstractions, Intermediate Representation, Parallelism, Scheduling Techniques, Memory Safety

NORTHWESTERN UNIVERSITY

Automating Parallel Execution Plan Selection and Tuning

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Yian Su

EVANSTON, ILLINOIS

June 2026

ABSTRACT

Modern hardware is parallel and is increasingly relying on parallel programming to achieve both performance and energy efficiency. However, today's parallel programming places a disproportionate burden on developers, requiring them to design and fine-tune parallel execution strategies across diverse hardware platforms and varying program inputs. I postulate that optimizing compilers can relieve developers from this burden by automatically reasoning about parallel execution strategies while also enforcing correctness properties. Achieving high parallel performance across modern hardware requires selecting an effective parallel execution plan, including how and where parallelism should be expressed and implemented. To automate this selection process, I introduce the Parallel-Semantics Program Dependence Graph (PS-PDG), a compiler abstraction that captures developer-encoded parallel semantics alongside compiler-derived analysis. PS-PDG enables compilers to override suboptimal execution plans with superior alternatives. While static execution-plan selection is effective for regular workloads, irregular and input-sensitive applications require runtime adaptation of parallel granularity. To address this challenge, I introduce the Heartbeat Compiler (HBC), a compilation system that automatically translates high-level fork-join parallelism into binaries capable of dynamically controlling task granularity through heartbeat scheduling to eliminate the need for manual task-size tuning. Beyond performance, C++ applications also face severe memory-safety risks arising from operations on data collections, including iterator invalidation. These bugs are difficult for existing tools to detect precisely and impose undefined behavior and security vulnerabilities in production environments. To address this challenge, I introduce Ledger, a data collection-oriented static analyzer that provides high-precision detection of invalidation vulnerabilities in programs that operate on complex data collections.

ACKNOWLEDGEMENTS

My Ph.D. journey has been an extraordinary experience, made possible by the support, mentorship, and encouragement of many people over the past five years.

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Simone Campanoni. He introduced me to academic research, taught me how to think and work as a researcher, and provided unwavering mentorship throughout my time at Northwestern. My decision to pursue a Ph.D. was shaped in large part by his encouragement during my first year at Northwestern, when I began exploring research as a Master's student. This dissertation would not have been possible without his guidance.

I am also grateful to my dissertation committee members—Professor Peter Dinda, Professor Nikos Hardavellas, Professor David I. August, and Dr. Zino Benaissa—for their insightful feedback and thoughtful guidance in helping shape this dissertation into its current form.

Additionally, I would like to express my gratitude to all my colleagues and collaborators both within and outside Northwestern: Enrico Armenio Deiana, Tommy McMichen, Brian Homerding, Federico Sossai, Haocheng Gao, Nick Wanninger, Mike Rainey, Nadharm Dhiantravan, Jasper Liang, Sotiris Apostolakis, Ziyang Xu, Ishita Chaturvedi, Zujun Tan, Yebin Chon, Umut A Acar, Karl Hallsby, Kirill Nagaitsev, David Krasowska, Michael Polinski, Peizhi Liu, Friedrich Doku, Alex Butler, Katie Harrison, Atmn Patel, Yuchen Zhu, Mu-Te Lau, and Akash Deo. I would like to especially thank Tommy McMichen for helping and guidance on the implementation of the Ledger project.

I am deeply grateful to all my friends Henry Nguyen, Prathamesh Korgaonkar, Gourav Kumbhojkar, Narin Dhatwalia, Vijay Kandiah, Xiling Li, Zhen Jia, Jiajia, Zheng Zhang, Wangcheng Xu, Yunqing Sun, Lili Ge, Kyle Wang, Ian Clarke, John Matta, Cindy Graham, and Philip Collora for

the joy, music, and memories we have shared over these years.

Last but most importantly, I would like to sincerely thank my parents, my baby cat Bagel, and my girlfriend for their unconditional love, support, and sacrifice, all of which made this journey possible.

THESIS STATEMENT

Today's parallel programming places a disproportionate burden on developers to select and tune parallel execution strategies across diverse hardware and inputs. I postulate that optimizing compilers can relieve this burden by introducing compiler abstractions and compilation techniques for runtime-adaptive execution that automate the optimization for both static and dynamic parallel execution plans targeting diverse hardware platforms and program inputs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

For decades, Dennard scaling [1] drove software performance forward by allowing each new process generation to deliver faster, more power-efficient single-core processors. This regime ended in the mid-2000s, when voltage scaling became impractical and the resulting power wall foreclosed continued frequency growth [2]. To keep delivering performance, hardware vendors turned to multicore processors, wider single-instruction multiple-data (SIMD) units, and increasingly heterogeneous accelerators. This architectural shift makes modern hardware fundamentally parallel.

To exploit this parallelism, developers write parallel programs using Parallel Programming Models (PPMs) such as OpenMP [3] and Cilk [4]. This process asks developers to identify parallelism opportunities (e.g., loops or independent tasks) and select specific implementation strategies (e.g., thread-level parallelism or vectorization)—together defining a *parallel execution plan* [5]. Ideally, compilers should then take these plans and optimize them to achieve high performance and energy efficiency across diverse hardware platforms and program inputs.

However, in practice, production compilers such as Clang [6] and GCC [7] often fall short of this goal. This is because they treat the developer's plan as an immutable contract and lower it directly into runtime calls early in the compilation pipeline [8], even when the plan is suboptimal for the target hardware or mistuned for the program's input. As a result, the burden of producing efficient parallel code falls disproportionately on developers, who must not only define the parallel execution plan but also make a series of complex decisions—such as the parallel execution model and the granularity of parallel tasks—to fully exploit the underlying hardware. In this thesis, we treat this limitation as an opportunity for the compiler and introduce new compiler abstractions

and compilation techniques that allow optimizing compilers to carry these portions of the parallel-programming burden that today rest on developers.

We first tackle the burden developers face in manually optimizing their parallel execution plans for the target hardware. PPMs allow developers to encode a parallel execution plan in their source code by selecting which loops or tasks to parallelize and declaring variable privatization, reduction, and thread synchronization through high-level language constructs such as OpenMP pragmas [9]. The plan a developer encodes, however, is a single fixed plan, and the best plan for a given program depends on the hardware it runs on. For example, consider the IS benchmark from the NAS [10] benchmark suite: the developer-encoded OpenMP plan uses thread-private arrays guarded by a critical section to update a shared array. The plan is correct, but less performant because the critical section serializes part of the computation, and its synchronization overhead grows with core count. A better plan that scales well on machines with many cores is via a reduction operation by allocating per-thread private arrays, and combining them after the parallel region. However, recognizing this opportunity requires more than the dependence information offered by traditional compiler abstractions such as the Program Dependence Graph [11]. The compiler must also see the parallel semantics the developer expressed through the OpenMP construct, which are discarded by production compilers at the frontend. Prior parallel intermediate representations (IR) [12], [13], [14], [15], [16], [17], [18] address part of this issue by embedding the developer's plan in the IR, but each encodes a single, inflexible execution plan that the compiler must respect; none allows the compiler to *replace* the developer's plan with a better one.

To solve this problem, we propose the *Parallel-Semantics Program Dependence Graph* (PS-PDG), a compiler abstraction that simultaneously represents the parallel semantics expressed by the developer and the dependence information derived from the compiler's own analysis. PS-PDG enables the compiler to reason about the full space of legal parallel execution plans and

select one tailored to the target hardware. Built on PS-PDG, we present *GINO*, an LLVM-based optimizing compiler that automatically explores, selects, and lowers improved parallel execution plans. We demonstrate the effectiveness of PS-PDG and GINO across eight benchmarks from the NAS benchmark suite [19]. On a 56-core machine, GINO improves average speedup over the sequential baseline from 6.8× to 7.8×, and outperforms the developer's original parallel execution plan by up to 46.6%.

While PS-PDG enables the compiler to replace a developer's plan with one better suited to the target hardware, it does not address a second burden: tuning the dynamic aspects of the parallel execution plan that must adapt to varying program inputs. One such aspect is the granularity of parallel tasks. Existing PPMs require developers to explicitly encode granularity decisions in source code, despite the fact that the best granularity is input-dependent. For example, consider a sparse matrix-vector multiplication [20] where the developer picks a static schedule that assigns equal-sized row chunks to threads: this works well on a regular workload such as a matrix with uniformly distributed nonzeros, but suffers severe load imbalance on an irregular workload with a skewed distribution, leaving some threads idle and busy-waiting at the fork-join barrier. Because no single granularity is optimal across inputs, developers must manually implement a different parallel execution plan for each workload they encounter. Prior runtime approaches [21], [22], [23], [24] adapt scheduling decisions at runtime, but still require the developer to decide in source code how much work each task should perform. Heartbeat scheduling [25] removes this requirement in principle by provably bounding parallelism overhead as a fraction of useful work, independent of how much latent parallelism the program exposes. However, its prior implementation [26] required developers to hand-write code largely in assembly and offers no path to automation.

To solve this problem, we propose the *Heartbeat Compiler* (HBC), the first compiler that translates C/C++ programs with high-level fork-join constructs (e.g., OpenMP `parallel for`) into

binaries that automatically tune task granularity at runtime through heartbeat scheduling. HBC transforms each parallel loop into a form that exposes its *potential* parallelism, and lets the program runtime dynamically decide how much of it to instantiate as tasks based on the program input. On irregular workloads—including TACO-generated sparse tensor kernels [20], GraphIt-generated graph algorithms [27], and benchmarks drawn from Rodinia [28]—running on a 64-core Intel-based machine, HBC improves average speedup from 14.2× under OpenMP dynamic scheduling to 21.7×, and matches the performance of binaries that previously required several months of manual implementation [26] efforts from developers.

As a by-product of working on this thesis, we present to the C++ developer community Ledger, a data collection-oriented static analyzer that detects iterator invalidation vulnerabilities and further reduces the burden developers carry in writing correct C++ applications. Iterator invalidation occurs when a program uses an iterator after its underlying container has been modified, producing use-after-free errors that have led to undefined behavior and security vulnerabilities in production software such as Chrome [29] and Firefox [30]. Ledger catches these bugs at compile time, achieving 87.5% recall on a synthetic benchmark suite of C++ programs, compared to 22.8% for prior state-of-the-art static analyzers such as Cppcheck [31] and the Clang Static Analyzer [32].

1.1 Thesis Contribution

The contributions of this thesis are as follows.

- It introduces the Parallel-Semantic Program Dependence Graph (PS-PDG), a compiler abstraction representing both compiler-derived dependence information and developer-expressed parallel semantics. An optimizing compiler GINO, which built upon PS-PDG, is capable of generating better parallel execution plans, improving average speedup from 6.8× to 7.8× over the sequential baseline on the evaluated NAS benchmark suite (§2).

- It details the Heartbeat Compiler (HBC), which compiles high-level fork-join parallelism into binaries that can dynamically control task granularity through heartbeat scheduling. On irregular workloads, HBC improves average speedup from 14.2× with OpenMP dynamic scheduling to 21.7× (§3).

- It presents Ledger, a data collection-aware static analyzer for detecting iterator invalidation vulnerabilities in C++ applications that operate on complex data collections. Ledger achieves 87.5% recall rate on detecting invalidation over 20 synthetic benchmarks, compared with 15.2% and 22.8% from widely adopted industry analysis tools clang static analyzer and cppcheck, respectively (§4).

1.2 Published Work

The PS-PDG work has been published at CGO 2026 [5] with contributions from the thesis author and co-authors Brian Homerding, Haocheng Gao, Federico Sossai, Yebin Chon, David I. August and Simone Campanoni. The HBC work has been published at ASPLOS 2024 [33] with contributions from the thesis author and co-authors Mike Rainey, Nicholas Wanninger, Nadharm Dhiantravan, Jasper Liang, Umut Acar, Peter Dinda and Simone Campanoni.

1.3 Thesis Organization

The rest of this thesis is structured as follows. Chapter 2 describes the Parallel-Semantics Program Dependence Graph. Chapter 3 details Heartbeat Compiler (HBC). Chapter 4 presents Ledger. Finally, Chapter 5 provides a summary of my contributions and discusses directions for future work, and concludes this thesis.

CHAPTER 2

AUTOMATING PARALLEL EXECUTION PLAN SELECTION

2.1 The Parallel-Semantics Program Dependence Graph for Parallel Optimization

Modern shared-memory parallel programming models, such as OpenMP and Cilk, enable developers to encode a parallel execution plan within their code. Existing compilers, including Clang and GCC, directly lower or add additional compatible parallelism on top of the developers' plan. However, when better parallel execution plans exist that are incompatible with the original plan, compilers lack the capability of disregarding it and replacing it with a better one. To address this problem, this work introduces the parallel-semantics program dependence graph (PS-PDG), an extension of the program dependence graph (PDG) abstraction that can simultaneously represent parallel semantics derived from both the developer's original plan and the compiler's own analysis. To demonstrate the power of PS-PDG, this work also introduces GINO, an LLVM-based compiler capable of optimizing parallel execution plans using PS-PDG. Through exploring, reasoning, and implementing better parallel execution plans unlocked by PS-PDG, GINO outperforms the developer's original parallel execution plan by 46.6% at most, and by 15% on average over 56 cores across 8 benchmarks from the NAS benchmark suite.

2.2 Introduction

Modern hardware is parallel. Despite decades of research, many factors still render automatic parallelization of sequential code impractical. Because of this, developers take advantage of multicore processors by explicitly writing parallel code. The community has introduced compiler extensions

and libraries in an attempt to boost the productivity of developers interested in thread-level parallelism (TLP). OpenMP [9] and Cilk [4] shine as the most renowned parallel programming models, where developers can directly express sections of the code that *can* and *will* run in parallel. In these models, the transformation into multithreaded code (e.g., via pthreads) is done by the compiler, which follows developer specifications without objection. This constrained approach suffers from a major drawback. A compiler that by contract follows the developer-encoded *parallel execution plan*[1], i.e., which instructions will run in parallel and how, is severely limited in its ability to tailor optimizations for a target architecture. For example, loops marked for multithreaded execution will execute in that fashion regardless of the profitability of TLP in the target environment. In some cases, SIMD execution may be preferable for tight loops or single-core execution but due to the compilation constraints that OpenMP follows, each annotated loop must execute according to the model specified by the developer. Encouragingly, the same information that lets one reason about multithreading can be reused to reason about vectorization. DOALL loops annotated with `omp for` are candidates for vectorization since they are marked as having no inter-iteration dependences by the developer. Dependences between instructions have in fact been a key abstraction to reason about parallelization of *sequential* code, but the long-established *program dependence graph* (PDG) was conceived to represent sequential programs; its limitations become quickly evident when reasoning about *already parallel* code. Although patterns like reductions on scalars can be amenable for parallelization even when abstracted in the PDG (e.g., by inspecting its SCCs), in general, the semantics that parallel programs carry go beyond data or control dependences. For instance, clauses like `firstprivate` or `lastprivate` encode opportunities that cannot be represented by the sole use of dependences or lack thereof. Similar limitations led research to develop parallel IRs, e.g., TAPIR [12], that can capture what *will* run in parallel while enabling classic

[1] We use the terms *parallel execution plan* and *parallelization plan* interchangeably throughout the paper.

optimizations for sequential IRs. However, like TAPIR, other existing parallel IRs [13], [14], [15], [16], [17], [34] are not suited for *parallel* optimization of parallel code as they encode a single inflexible execution plan that compilers must respect. In other words, **both the PDG and existing parallel IRs fail to abstract the semantics of parallel code while allowing optimization of the parallelism itself**.

This work proposes the *Parallel-Semantics Program Dependence Graph* (PS-PDG) – an abstraction that can simultaneously represent parallel semantics derived from both the developer's original parallel execution plan and the compiler's own analysis. The PS-PDG extends the PDG to allow compilers to access a larger optimization space and restructure parallelism in a way that makes the best use of the target architecture. We introduce GINO, an optimizing compiler that leverages the proposed PS-PDG and is able to craft and lower a parallel execution plan for both multithreaded and vectorized execution.

The main contributions of this work are:

- A definition of the PS-PDG: a hierarchical graph that represents parallel semantics expressed by both the developers' knowledge and compiler analyses (§2.4).
- A detailed analysis of the necessity of each element of the PS-PDG (§2.5).
- An empirical proof of the inadequacy of the PDG as an abstraction for the optimization of parallel programs (§2.6).
- An evaluation of GINO – The first LLVM-based optimizing compiler that uses the PS-PDG for parallel optimization (§2.6).

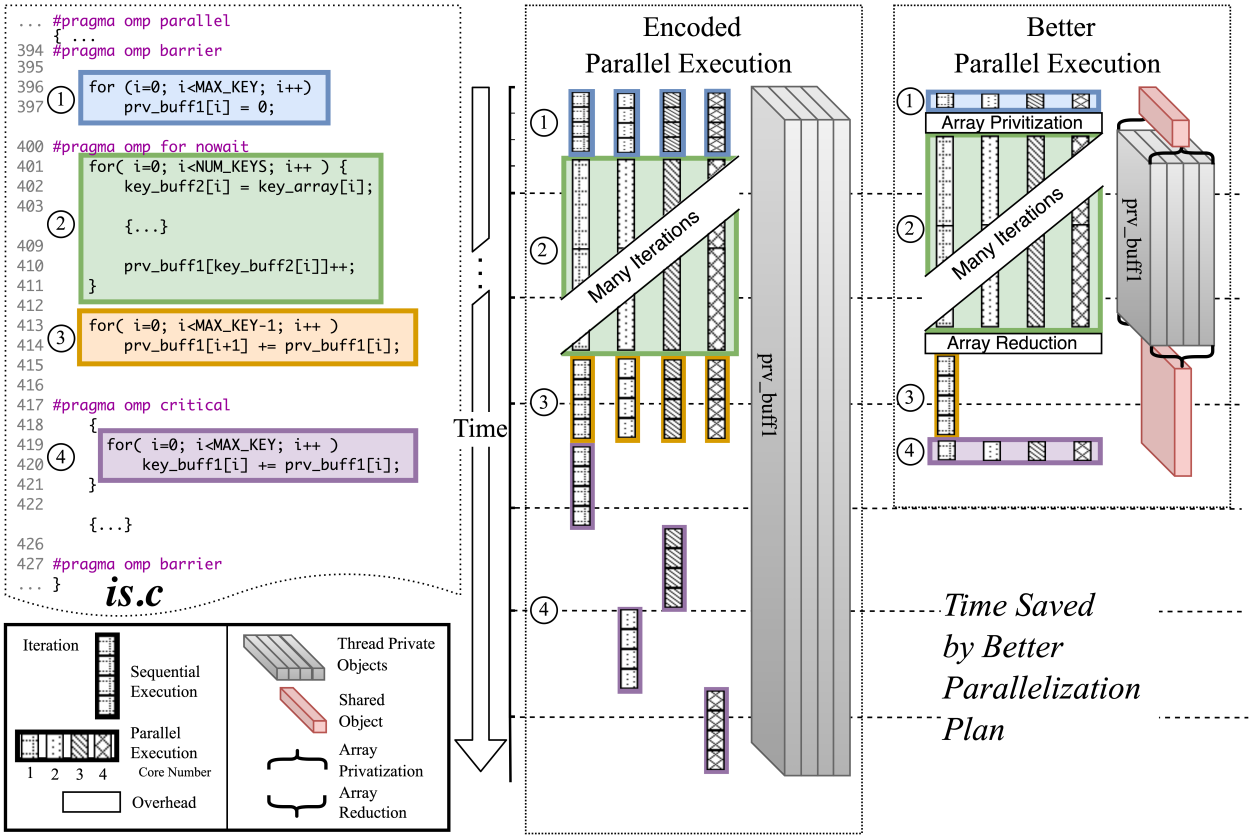


Figure 2.1: The key computational kernel from the IS benchmark with the original and a more performant compiler-selected parallel execution plan.

2.3 Background & Motivation

This section presents an OpenMP program to illustrate how developers explicitly encode a parallel execution plan in the source code. We then demonstrate that an alternative, more efficient plan exists, which cannot be realized within the existing compilation pipeline. This case motivates the need for the PS-PDG abstraction, which captures the precise constraints of a parallel program and enables more effective parallel optimization.

2.3.1 Developers Explicitly Encode Parallelism

Modern shared-memory *parallel programming models* (PPMs) allow developers to encode parallelism directly in their applications to achieve high performance and energy efficiency. A *parallel execution plan* specifies *what* to parallelize (e.g., loops), *how* to parallelize (e.g., variable privatization and initialization), and the *execution model* to use (e.g., tasks, threads, vectorization). Among PPMs, OpenMP [9] is one of the most widely adopted: it enables developers to encode a parallel execution plan in their code using a set of compiler directives (pragmas in C/C++). For instance, `#pragma omp parallel for` distributes loop iterations across multiple threads. Other pragmas offer greater control over the parallel execution, such as `critical`, which indicates that the enclosed code region is protected and should only be executed by a single thread at any given time.

Fig. 3.1 shows the OpenMP source code from the hottest computation in the `IS` benchmark of the NAS benchmark suite [10], together with its parallel execution plan encoded using OpenMP pragmas. The entire kernel is enclosed in a `#pragma omp parallel` region, which spawns multiple threads, each executing the enclosed code region in parallel. Loops ① and ③ do not have worksharing pragmas, so each thread in the parallel region executes the entire loop on its private copy of array `prv_buff1`. Loop ② instead has its iterations distributed across threads. Finally, loop ④ is enclosed in a `critical` section to serialize updates to the shared array `key_buff1` and avoid data races.

2.3.2 Optimizing Parallel Programs

Let us reconsider the parallel region of `IS` shown on the right side in Fig. 3.1, now with a different parallel execution plan. In this new plan, iterations of loop ① are parallelized across threads, each operating on a disjoint slice of the shared array `pre_buff1`. Before loop ②, each thread privatizes `pre_buff1`, creating a local copy. Loop ② is then executed in parallel as before. Once

finished, the private copies are reduced into the shared array. As a result, loop ③ only needs to run on a single thread, avoiding parallel overhead that the original plan incurred. Finally, with only one shared copy of `pre_buff1` remaining, loop ④ can be distributed across threads without requiring a `critical` section.

Transforming the original plan into this optimized one is non-trivial. The compiler must first recognize that `prv_buff1` is privatizable through complex dependence analysis or developer-provided annotations. It must then identify the updates in loop ② as reduction operations, and ensure that aliasing does not occur among the involved array pointers.

2.3.3 Existing Optimizing Compilers Do Not Leverage Parallel Semantics

Unfortunately, existing optimizing compilers for PPMs such as OpenMP [9] and Cilk [4] cannot realize this transformation. As illustrated in the left flow in Fig. 2.2, the compilation pipeline lowers the developer's parallel execution plan directly into runtime calls during the frontend stage. This design simplifies compiler implementation, given that much of the sequential optimization can be reused. However, this approach has a fundamental drawback: the **parallel semantics encoded by the developer are discarded** at the compiler frontend and **unrecoverable from the middle end**. As a result, valuable information – such as parallel loops, which have no loop-carried data dependencies, or variables that can be privatized along with their initialization information – is lost and cannot be recovered for optimization [12].

2.3.4 Optimizing Compilers Need Powerful Abstraction to Represent Parallel Semantics

The motivating example proves that compilers need the capability of optimizing the parallel execution plan expressed by developers. Today's compilers [35], [36], [37], [38], [39] successfully use the PDG abstraction, but PDG does not capture the parallel semantics expressed in the source

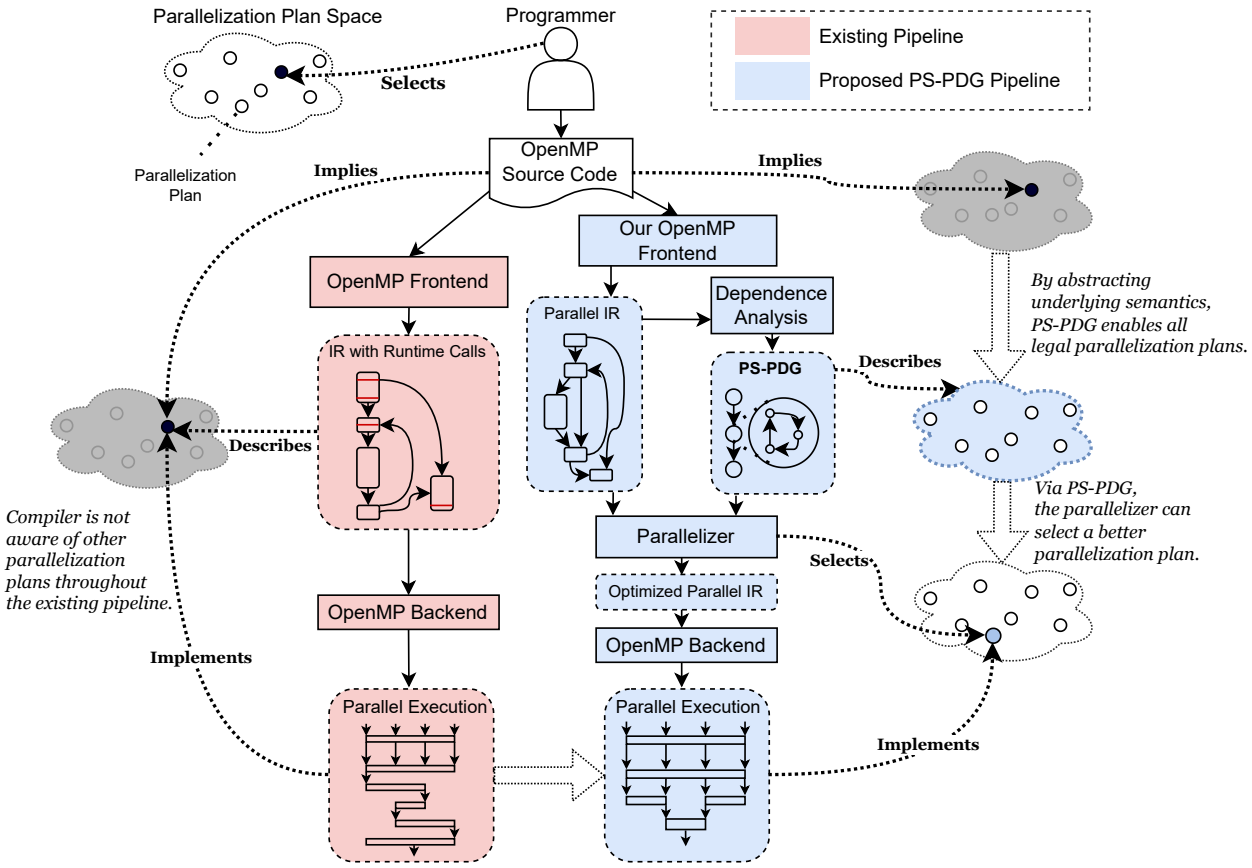


Figure 2.2: Comparison of the existing pipeline with our proposed PS-PDG pipeline. Our proposed pipeline enables an optimizing compiler to reason and implement a more performant parallel execution plan. The Parallel IR refers to MLIR's `omp` dialect in our implementation.

code. To overcome this limitation, this work proposes the PS-PDG, an abstraction that captures the precise parallel constraints implied by the developer's plan. With PS-PDG, compilers can reason about a space of semantically equivalent parallel execution plans. The PS-PDG enables the pipeline shown in the right flow in Fig. 2.2. The proposed pipeline does not rigidly follow the encoded parallel execution plan (like today's compilers do), and instead enables compilers to select the plan that best matches the underlying architecture.

2.4 PS-PDG Definition

PPMs like OpenMP enable developers to make parallelization decisions explicit. A developer can decide where to spawn threads or tasks to distribute the computation of a loop, and how to synchronize their execution (i.e., the parallel execution plan of that program).

Beyond controlling what code can run in parallel and when it can do so, a parallel execution plan also *implies properties of the code* of the original program. For example, a parallel execution plan described using OpenMP can include the declaration that iterations of a loop will run in parallel during their executions. This plan implies the property that the target loop has no loop-carried dependences between its iterations. Another example is an OpenMP `critical` section in a loop, which implies both the need to enforce the atomic property of the target code segment and that any order of invocations of the target segment between loop iterations is valid. We refer to this implied information as *the precise constraints of a parallel program*, which is captured by the PS-PDG.

Table 2.1: Complete PS-PDG Definition

PS-PDG	::= (Node$^+$, Edge*, Variable*, VariableAccess*)
Node	::= (Instruction | HierarchicalNode, Trait*)
HierarchicalNode	::= (Node$^+$, Context?)
Trait	::= (Singuler | Unordered | Atomic, Context)
Edge	::= DirectedEdge | UndirectedEdge
DirectedEdge	::= (Node$_{\text{producer}}$, Node$_{\text{consumer}}$, Data-selector?)
UndirectedEdge	::= (Node, Node, Context)
Data-selector	::= (Any-Producer | Last-Producer | All-Consumers, Context)
Variable	::= (Privatizable | Reducible, Context)
VariableAccess	::= (Variable, Node$^*_{\text{use}}$, Node$^*_{\text{def}}$)
Context	::= *Unique Identifier*

The PS-PDG extends the PDG abstraction to capture the precise parallel constraints of an

OpenMP or Cilk program. Like the PDG, the PS-PDG has nodes to represent computation and edges to represent dependences within the computation, it also includes variables to represent data and use/def edges to represent the relation between data and its computation. As shown in Table 2.1, a PS-PDG consists of one or more nodes with zero or more edges, variables and variable accesses. The rest of this Section describes each extension in detail.

2.4.1 Hierarchical Nodes

Explicit parallel programming enables developers to specify properties of a code region. Often such properties do not hold at finer granularities (e.g., single instruction). For example, an OpenMP `critical` section declares that the code region as a whole has the atomicity property. This atomic property does not hold at a finer granularity like at the single instruction level that composes this critical code region. For this reason, the PS-PDG both adds the ability to have a single node that represents an entire code region and the ability to express properties at their node granularity (§2.4.2).

A node in the PS-PDG represents a non-empty set of instructions organizing the code hierarchically, shown in Fig. 2.3. For example, all instructions of a `critical` section in the PS-PDG is represented by a single node. More generally, a node `N` of the PS-PDG is a non-empty set of one or more instructions or other nodes such that both direct and indirect self-inclusions are not allowed. Having a single node representing a set of instructions is needed to capture the parallel semantics of parallel constructs that target more than a single instruction.

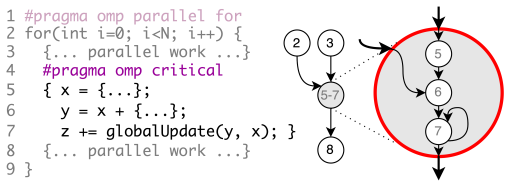


Figure 2.3: Capturing properties of a region into hierarchical nodes with traits

2.4.2 Node Traits

Some properties expressed in a PPM are traits of a code region. These traits can be important for the correctness and/or performance of a parallel application, for instance, atomicity.

A node in the PS-PDG can have various traits. this work implemented the three types of traits that are enough for the target languages OpenMP and Cilk: the atomic, orderless, and singular traits. An atomic node represents a set of computations that must be executed atomically during its parallel execution. An orderless node expresses that different instances of that node can be executed in any order for a given context. A singular node represents a set of computations that must be executed by only a single instance for a given context. An example of a node traits is shown in Fig. 2.4.

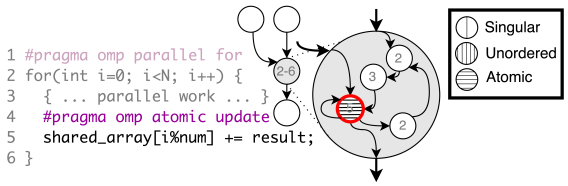


Figure 2.4: How traits can be used to capture atomic updates.

2.4.3 Context

PPMs allow developers to express semantics attached to a code region only when executed within the context of another code region. For example, the code in a `single` OpenMP pragma needs to only be executed for one of the iterations of the innermost parallel loop that contains it. It does not however specify that code should only be executed by a single iteration of an outer loop. In other words, the parallel semantics of a `single` section is valid only in the context of the innermost loop that contains it. Because the contexts in which parallel semantics is valid cannot always be computed, the PS-PDG can specify contexts and their relation with parallel semantics.

A context in the PS-PDG represents a code region to which a parallel semantic applies. A context in the PS-PDG is a labeled hierarchical node, where the label is a unique identifier. Hence, hierarchical nodes of the PS-PDG that do not have a label are not contexts. A parallel semantic explicitly lists the contexts in which it is valid, as shown in Tab. 2.1. For example, the hierarchical node `S` of Fig. 2.5 that captures the `single` code section declares its semantics applies only to context `A`, which is its target loop.

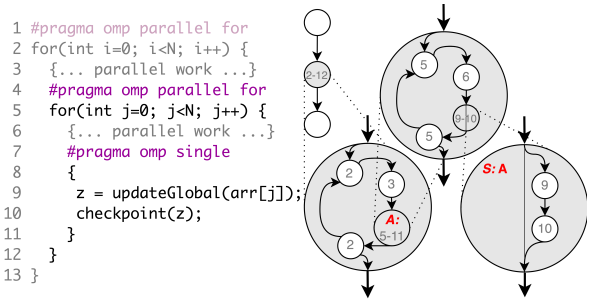


Figure 2.5: Traits applied to context `A` captures the region's semantics.

2.4.4 Directed and Undirected Edges

Parallel programming allows the declaration that two code regions (or two instructions) depend on each other but their relative execution order is not important. This enables efficient parallel executions by avoiding unnecessary synchronizations. PS-PDG includes both directed and undirected dependences (edges) to capture this semantics (Fig. 2.6).

A directed edge in a PS-PDG follows the semantics of the PDG abstraction where the execution of the destination of that edge must wait for the edge's source execution. Instead, an undirected edge expresses a dependence between two computations (e.g., instructions) that cannot run in parallel, but any ordering of their execution is allowed.

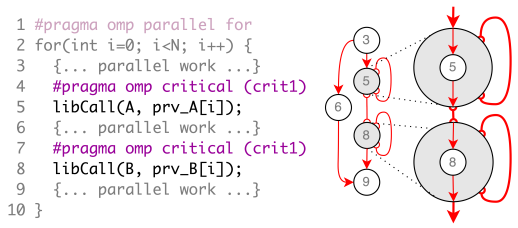


Figure 2.6: Ordering constraints are captured with directed and undirected edges.

2.4.5 Data-Selector Directed Edge

The execution of an application typically includes many instances of a single static instruction (e.g., multiple executions of a single static instruction within a loop). There is always a clear producer-consumer relation between dependent instructions for sequential programs. For example, consider the instructions i and j shown in Fig. 2.7 which has a dependence from i to j. In this sequential program, the last instance of i executed before j will generate the data consumed by j (this is captured by the PDG). However, developers can express richer semantics when developing a parallel program. For example, a developer can express that the data generated by any instance of i can be used by j. This is not expressible in prior abstractions like the PDG. Hence, the PS-PDG introduces data-selectors that can be attached to a direct dependence.

A data-selector defines the set of dynamic instances of a static instruction. A directed edge in the PS-PDG can have up to two data-selectors: one per static instruction attached to the edge. A data-selector of the producer of a dependence defines which dynamic instance(s) of that producer are allowed to generate the data that will unlock the consumer.

this work implements only the data-selectors required to capture the semantics of OpenMP and Cilk, which are the following:

- `Any Producer Selector`: The consumer may use data generated by any instance of the producer.

- `Last Producer Selector`: The consumer must use data generated by the last instance of the producer.
- `All Consumers Selector`: All consumers must use the data generated by the producer.

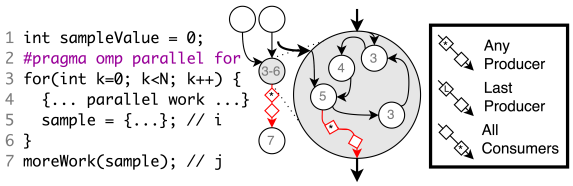


Figure 2.7: Data-selector directed edges can capture non-trivial data relations.

2.4.6 Parallel Semantic Variables

Efficient parallel execution often requires developers to express knowledge about the program's variables that go beyond their reads and writes and their data types. For example, developers can express that a variable can be privatized in threads/tasks and all private instances can be merged (reduced) using application-specific knowledge. This semantics goes beyond what can be expressed in sequential programming and therefore beyond what the PDG can capture (as the PDG was designed for sequential code). To preserve this semantic, the PS-PDG introduces the concept of variables and their parallel semantics (how to clone them, their identity value, and how to reduce them) in its abstraction.

A parallel semantic variable in PS-PDG represents a variable or memory object that can be cloned to create private copies that a thread or task can independently use and modify. This extension includes the code to execute to merge pairs of private copies together. To do so, the variable description includes the reference to a computational node of the PS-PDG that represents a function. This function takes two copies of a variable and it updates the first one with the result of the

merge. This merging operation is what compilers can use to reduce all private copies of a variable into a single one. An example of parallel semantic variable is shown in Fig. 2.8.

Parallel semantic variables are accessed by computation (e.g., an instruction). Because such variables can be stored in memory, their accesses are not captured by the conventional use-def chains [40]. To preserve this relation, the PS-PDG adds the Use/Def edges from a variable to PS-PDG nodes to encode the semantics that a target node uses and/or defines the variable at the source of that edge.

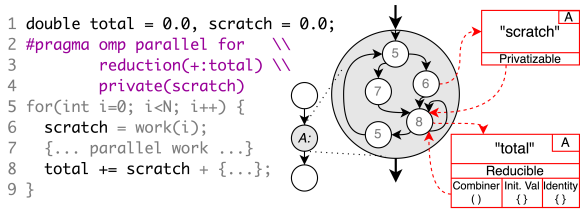


Figure 2.8: Capturing developer knowledge about data through privatizable and reducible parallel semantic variables.

2.5 The Necessity of PS-PDG Components

This section demonstrates that each feature of the PS-PDG is necessary to capture the semantics expressible using the OpenMP programming model. The same result can be obtained similarly for Cilk (not included for space constraints). Each extension is proved to be necessary by removing it from the proposed abstraction and showing that two parallel programs with two different parallel execution plans and semantics are now translated to the same PS-PDG when the extension under evaluation is not available. Additionally, this section provides an example that shows how each feature enables an important optimization. Overall, this section shows the value of each PS-PDG extension.

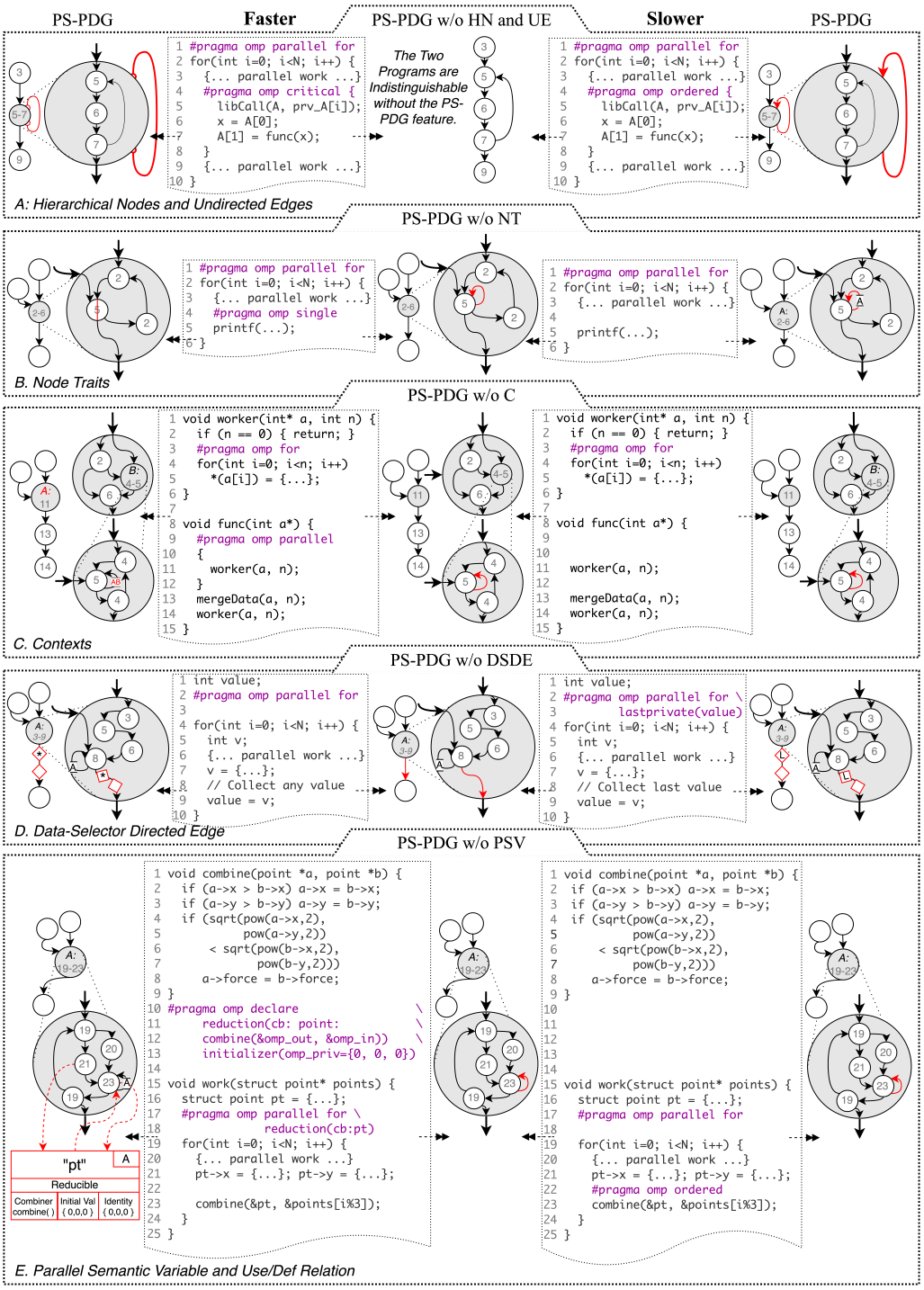


Figure 2.9: Each feature of the PS-PDG is necessary since the removal of any PS-PDG feature would result in a loss of information. Without the given feature, the resulting abstraction is indistinguishable for the faster code (left) and the slower code (right). For readability, we present the program in C. In our implementation, the PS-PDG is built from the MLIR `omp` dialect.

2.5.1 Hierarchical Nodes and Undirected Edges

To understand the value of Hierarchical Nodes (HN) and Undirected Edges (UE), consider the two semantically different programs shown in Fig. 2.9-A. The program on the left requires avoiding overlapping dynamic instances of the critical section but puts no restriction on their order. In contrast, the program on the right requires each dynamic instance of its critical section to be executed in loop-iteration order. The program on the left executes significantly faster than the one on the right because it does not require synchronizations to enforce this additional constraint. A compiler seeking the best parallel execution plan for each program in this way must know whether or not this extra degree of freedom (orderless) exists. In the PS-PDG, the undirected edge and hierarchical node features combined remove the ordering constraint while ensuring that dynamic instances of the connected nodes do not overlap. When this feature is removed, this semantic information is lost. Fig. 2.9-A demonstrates this by showing how these two programs map to the same PS-PDG lacking these features ("PS-PDG w/o HN and UE"). Furthermore, this orderless semantics cannot be represented by the "PS-PDG w/o HN and UE" because the orderless semantics does not hold at the single instruction granularity.

2.5.2 Node Traits

A node in the PS-PDG can hold various traits expressed in a parallel programming language. These traits can be important for the correctness and performance of the parallel application. To understand the value of Node Traits (NT), consider the two semantically different programs shown in Fig. 2.9-B. The program on the left requires the singular execution of the print statement, allowing for quick and simple output from the parallel application. In contrast, the program on the right does not include a `single` annotation for its print statement, meaning multiple calls to `printf`. To maintain correctness, then the compiler must understand how the `printf` fits into the parallel

execution plan. Unfortunately, this is not possible when using the PS-PDG without Node Traits ("PS-PDG w/o NT"). Fig. 2.9-B demonstrates this by showing how these two programs map to the same "PS-PDG w/o NT". In the "PS-PDG w/o NT", the single execution semantic is lost. Further, the single execution trait cannot be determined by compiler analysis from any other aspect of the "PS-PDG w/o NT".

2.5.3 Contexts

To understand the value of Contexts (C), consider the two programs shown in Fig. 2.9-C. The program on the left executes the first call to `worker` in parallel while the program on the right executes it sequentially. By leveraging the parallelism in the hardware, the left program executes significantly faster than the program on the right. To generate the best parallel execution plan for the first `worker` call, the compiler needs to know in which context(s) the independent loop iteration semantic holds. PS-PDG Contexts represent the contexts in which code region parallel semantics hold. Without PS-PDG Contexts, the two programs map to the same "PS-PDG w/o C". Using only the "PS-PDG w/o C" in this example, the compiler cannot know when the loop iterations are independent of each other and must assume they are not.

2.5.4 Data-Selector Directed Edge

Data selectors can be added to the directed edges of the PS-PDG abstraction. Data selectors define which dynamic instance (or instances) of the source node can generate the data that the destination node needs. To understand the value of Data-Selector Directed Edge (DSDE), consider the two parallel programs shown in Fig. 2.9-D. Their semantics are different. The program on the right enforces that the value of the live-out variable `value` that can propagate outside the loop has to be the one generated during the last iteration of that loop. The program on the left of Fig. 2.9-D

allows the propagation of the value generated by any loop iteration, which adds an extra degree of freedom. With this freedom, a consumer of the live-out variable can start before the end of the loop execution, allowing for more overlapping computation. Unfortunately, a PS-PDG without Data-Selector Directed Edges ("PS-PDG w/o DSDE") cannot distinguish these cases. This can be seen as both programs in Fig. 2.9-D map to the same "PS-PDG w/o DSDE". Thus, for correctness, the parallel execution plan generated from the "PS-PDG w/o DSDE" must enforce a stricter semantics of the program, the slower program on the right. Note that the DSDE semantics cannot be inferred by a code analysis on the "PS-PDG w/o DSDE".

2.5.5 Parallel Semantic Variable and Use/Def Relation

The PS-PDG includes Parallel Semantic Variables and Use/Def Relations (PSV) to represent a variable or object upon which the developer has encoded parallel semantics (e.g., how to reduce an object between tasks). These variables are connected to their computation (reads and writes) through Use/Def edges from parallel variables to nodes in the PS-PDG. Consider the two parallel programs shown in Fig. 2.9-E. The program on the left runs all iterations of the loop in parallel without any synchronization between them. Each thread operates on a private copy of the struct `pt`, then all private copies are reduced into a single one to be propagated to the code after the loop. This reduction is performed using application-specific knowledge. In contrast, the program on the right of Fig. 2.9-E has a single copy of the struct `pt` shared among all iterations of a loop. Accesses (reads and writes) of this array are synchronized using an ordered section. The program on the left executes significantly faster than the one on the right because it does not require any synchronization between the loop iterations running in parallel. A compiler that needs to decide the parallel execution plan to apply to the program on the left of Fig. 2.9-E needs to be aware of the ability to privatize and reduce the struct `pt` to generate the best parallel execution plan.

Unfortunately, this is not possible when using a PS-PDG without Parallel Semantic Variables ("PS-PDG w/o PSV"). This becomes clear by observing that both programs in Fig. 2.9-E map to the same "PS-PDG w/o PSV". This means that the parallel execution plan generated from the "PS-PDG w/o PSV" must enforce the stricter semantics of the program on the right of Fig. 2.9-E where all array accesses are ordered. Furthermore, notice that the lost application-specific knowledge about the reduction of the struct `pt` cannot be inferred from the "PS-PDG w/o PSV".

2.6 Evaluation

This section evaluates GINO, the first optimizing compiler that leverages PS-PDG for parallel optimization. After describing GINO's compilation pipeline, parallel execution plan selection mechanism, as well as the experimental settings, we measure GINO's ability to improve performance when it automatically selects and implements a parallel execution plan better than the one specified by the developer, achieving an average **+15%** speedup. Next, we compare GINO against OpenMP when both use the same developer-encoded plan, showing performance parity with the state-of-the-art compiler infrastructure. Finally, we contrast GINO with a state-of-the-art PDG-based optimizing compiler, and demonstrate that the PDG alone is limited in expressiveness: even after supplying parallel semantics, it fails to match the PS-PDG's performance.

2.6.1 GINO Pipeline and Implementation

GINO takes OpenMP C/C++ source code as input and produces parallel binaries that implement an optimized parallel execution plan selected using the PS-PDG abstraction.

Implementation GINO is built by extending NOELLE [37], a PDG-based compilation framework that augments LLVM with dependence-oriented abstractions and advanced alias analysis. In

this work, we modified NOELLE in the following two ways. 1) To construct PS-PDG, we extended NOELLE's PDG abstraction to capture and represent parallel semantics described in §2.4. 2) We extended NOELLE to accept MLIR's `omp` [18] dialect as input (generated using Clang), which preserves explicit parallel semantics. This extended framework is referred to as NOELLE-PSPDG. The original NOELLE authors built an automatic parallelizer (referred to as NOELLE-par) on top of NOELLE, which parallelizes the code by selecting the best parallelization transformation among DOALL [41], HELIX [35], and DSWP [36] on a per-hot-loop basis, solely relying on the PDG. We extended NOELLE-par to operate on PS-PDG and implemented a heuristic-based plan selection mechanism. The above system overall is called GINO. After a plan is selected, GINO lowers it to LLVM IR and relies on the off-the-shelf Clang backend for code generation.

PS-PDG construction The construction algorithm initially allocates a PS-PDG node per loop and uses NOELLE's Forest abstraction [37] to organize them hierarchically per function. Then, for every `omp` operation (e.g., `omp.parallel`, `omp.wsloop`, `omp.critical`), we create a corresponding PS-PDG node. These nodes are nested within each other, respecting their region nesting relations. For each top-level node, we perform a data dependence analysis using both MLIR's AliasAnalysis [42] and MemoryEffects [43] interface. Nodes created from `omp.wsloop` operations don't need loop-carried dependences to be computed since the OpenMP worksharing semantics imply their absence. For all other constructs, memory dependences are computed. Critical sections (`omp.critical`) are conservatively modeled by inserting undirected dependence edges among all operations participating in the region, ensuring serialized execution. Directed edges are node-node pairs and are added between node A and B iff there exists at least one dependence (i_A, i_B) following the definition in PDG, where i_A (resp. i_B) is an operation contained in the region described by A (resp. B). Finally, dependence edges among `omp.critical` nodes

within the same loop tree are marked as undirected. Register (def–use) dependences are not explicitly reconstructed, as they are already represented by MLIR's SSA form [44] and thus implicitly available to the PS-PDG. Parallel-semantics attributes such as `private`, `firstprivate`, and `reduction` are extracted directly from the argument list of the `omp` operations. For each such variable, we create a PSV instance and annotate it with its corresponding attribute kind, allowing the PS-PDG to precisely model the effects of privatization and reductions.

Parallel execution plan selection We implemented a plan selection heuristic guided profile information. Specifically, loops contributing less than 1.2% of the total runtime are vectorized to avoid thread-management overhead, while loops above this threshold are both multithreaded and vectorized. If GINO detects a single core at compile time, it applies vectorization only.

2.6.2 Experimental Settings

We evaluated GINO against NOELLE-par, OpenCilk-2.0 [22], an optimizing compiler for parallel programs backed by TAPIR [12], and OpenMP. All systems use LLVM 14 as the underlying compilation infrastructure. This ensures a fair comparison since all compilers share most of the infrastructure and optimizations. Both OpenCilk and OpenMP binaries are generated using Clang. All benchmarks are compiled using `-O3 -march=native`.

Benchmarks We evaluated GINO over all 8 benchmarks from the NPB 3.0-OMP C [45] version of the NAS [10] benchmark suite. We used input size `B` for benchmarks `BT` and `FT` due to gigabyte-size static variables and `C` for all others.

To evaluate OpenCilk, we ported all 8 NAS benchmarks by refactoring their parallel code regions by replacing each `#pragma omp parallel for` loop with `cilk_for` construct. OpenMP critical sections were rewritten using `cilk::mutex`, and scalar reductions were ex-

pressed using OpenCilk reducers. We restructured OpenMP data attributes such as `private` and `firstprivate` to explicitly allocate all per-worker private data (and `memcpy` initialization where necessary) before entering the parallel region. Each thread-private object inside a parallel code region is accessed via a per-worker array indexed by the worker ID returned by `__cilkrts_get_worker_number()` runtime call. These changes preserve the original program semantics while enabling execution under Cilk's work-stealing [21] model.

Platform All of our results are run and reported using an instance that runs Linux kernel 4.18.0 equipped with two Intel Xeon Gold 6258R processors featuring a total of 56 cores (28 per socket) running at 2.7 GHz, with 32K L1i, 32K L1d, 1024K L2, and 39424K L3 cache. Both hyperthreading and turbo-boost are disabled throughout the evaluation. We report the median result over 30 runs.

2.6.3 GINO Outperforms Developers' Parallel Execution Plans

GINO selects and implements better parallel execution plans that differ from those manually encoded by developers, and consistently delivers higher performance. Fig. 2.10 reports the speedups over sequential for parallel binaries with the original and improved parallel execution plans running on 56 cores. The speedup (on average) increases from **6.8**× to **7.8**× **(+15%)** over sequential baselines after delegating the choice of parallel execution plans to GINO. This demonstrates GINO's ability to generate optimized parallel binaries that leverage both the compiler's analysis and the parallel semantics expressed by developers.

GINO implements more performant parallel execution plans GINO improves `EP` (**+17.2%**) and `IS` (**+13.2%**), by implementing more efficient parallel execution plans. As illustrated in the motivating example (§2.3), both benchmarks originally implement array reduction by privatiz-

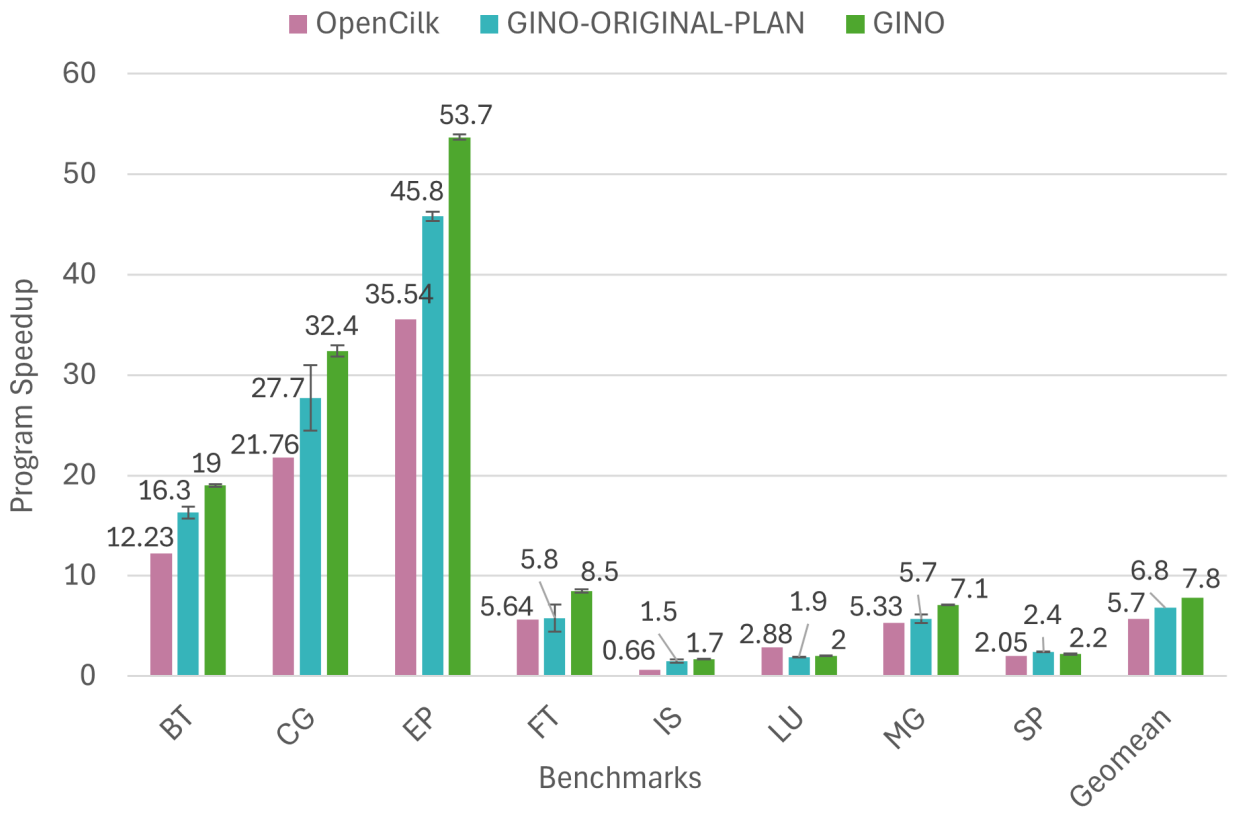


Figure 2.10: The speedup on 56 cores with GINO following the developers' original parallel execution plans versus GINO's PS-PDG-improved parallel execution plan over 8 NAS benchmarks.

ing and initializing a local copy of the array. The reduction is then protected by an OpenMP `critical` section, ensuring that each thread serializes updates to the global array.

GINO recognizes this reduction pattern and chooses a more scalable implementation. Specifically, it allocates a contiguous block of stack memory, partitioned such that each thread owns a cacheline-aligned privatized array. Initialization of the privatized array is vectorized using `memset` instead of a scalar loop. The final reduction is performed by the main thread by aggregating results directly from each thread's local copy.

This plan improves performance in two key ways. 1) It eliminates the runtime overhead of lock/unlock operations protecting the `critical` region, avoiding contention and thread spinning.

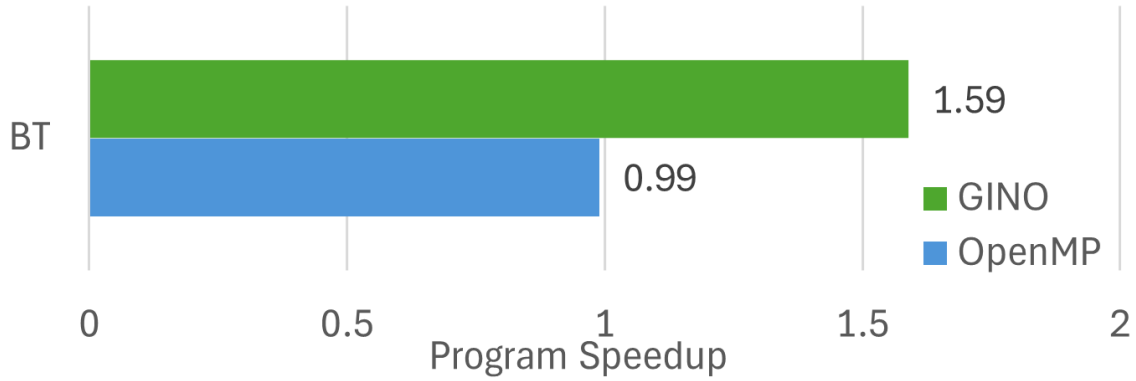


Figure 2.11: Single core speedup for BT resulted from vectorization.

2) It reduces cache coherence traffic: in the original plan, each update to the global array triggers cache invalidations between cores, whereas GINO's plan isolates updates to thread-local arrays and performs a single, cache-friendly aggregation step. The benefits become pronounced when the reduction array becomes large (e.g., 2048 integers for `IS`) and the program runs at scale (56 cores).

GINO vectorizes the code automatically when appropriate GINO gains additional speedup on top of developers' parallel execution plans for `FT` (**+44.1**%), `BT` (**+16.6**%), and `CG` (**+16.5**%) by automatically vectorizing hot loops that are already parallelized. GINO treats parallelization and vectorization as distinct execution plans enabled by the same parallel semantics. Loops marked as DOALL parallel are identified as safe for vectorization since they contain no loop-carried dependences. For such loops, GINO inserts vectorization metadata in the LLVM IR and leverages LLVM's automatic vectorization pass [46]. To improve vectorization efficiency, GINO restructures candidate loops by introducing an inner loop with a fixed step size, determined at compile time from the available SIMD flags (e.g., AVX2, AVX-512) and operand data types. This approach frees developers from the burden of selecting an architecture-specific vector width, which is often error-prone and non-portable.

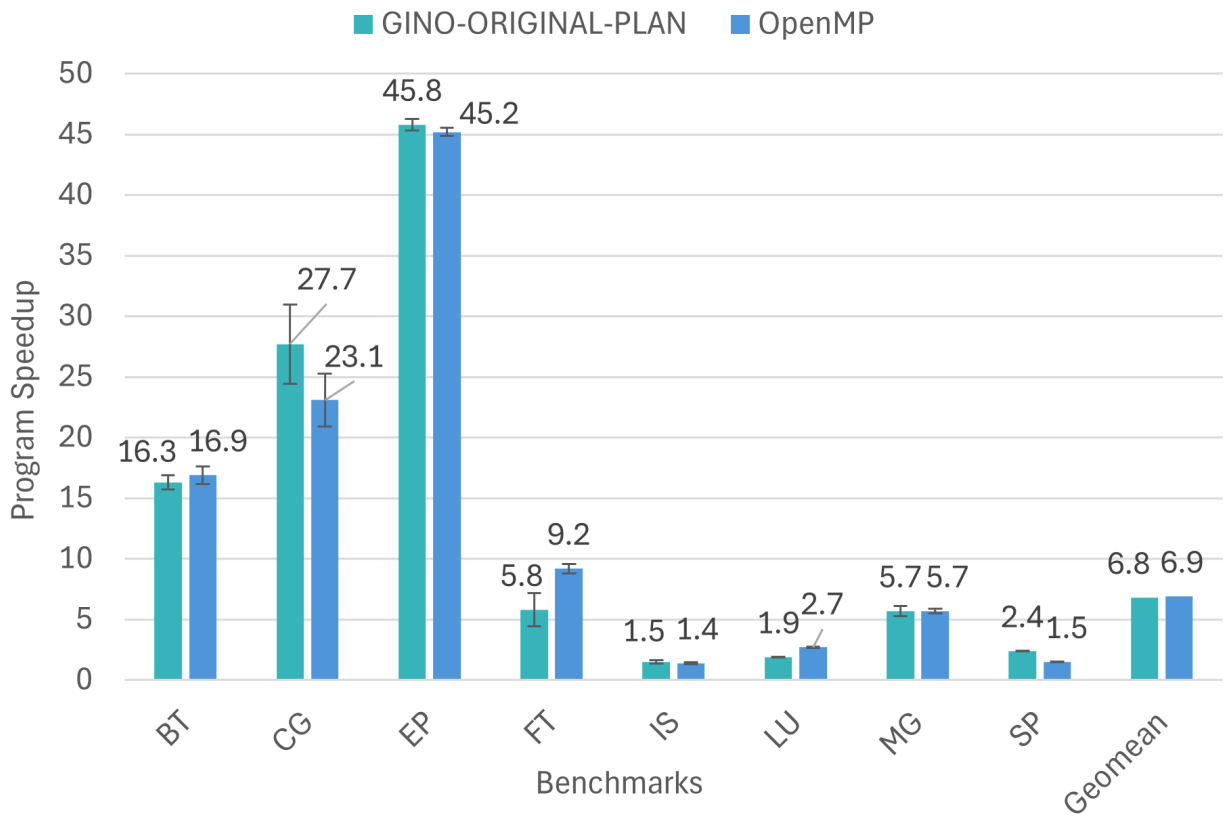


Figure 2.12: The speedup on 56 cores with the developers' original parallel execution plans with GINO versus clang OpenMP over 8 NAS benchmarks.

GINO also applies vectorization to loops where thread-level parallelization is not profitable. `MG` (**+24.6%**) benefits significantly from this strategy: the original plan parallelizes many small loops that are invoked repeatedly, incurring substantial overhead. GINO replaces them with vector execution, reducing overhead and improving performance.

GINO outperforms OpenCilk GINO achieves higher performance than OpenCilk (**+37%**) overall, and performs better (**+20%**) than OpenCilk even when GINO respects the developer-encoded parallel execution plan. The performance difference stems from the fundamentally different execution models used by the two systems. GINO uses the OpenMP runtime for scheduling parallel

loops, which uses large, statically assigned chunks and preserves per-thread data locality, which is well-suited to regular loop nests for NAS benchmarks. In contrast, OpenCilk relies on a divide-and-conquer work-stealing scheduler, which recursively subdivides iteration spaces and creates finer-grained tasks that introduce stealing overhead and reduce per-thread data locality. These effects become detrimental for regular loops such as those in `BT` and `SP`, where static scheduling delivers better performance.

Despite providing strong support for enabling sequential optimizations in parallel programs from TAPIR [12], OpenCilk shares the same limitation with OpenMP compilers that it does not explore or substitute semantically equivalent parallel execution plans, which can be realized using GINO.

GINO generates parallel binaries that adapt to the runtime environment The best parallel execution plan depends on the runtime environment, for instance, the number of available cores. With fewer cores, thread-level parallelism introduces no performance but purely parallel overhead [33], making vectorization a more effective strategy. Fig. 2.11 illustrates this point by running the `BT` benchmark on a single core, compiled with OpenMP and GINO, separately. The OpenMP implementation simply lowers the developer's plan and yields no benefit over the sequential baseline. In contrast, GINO detects the runtime configuration and implements vectorization and achieves a **59%** performance improvement. This example demonstrates that the best parallel execution plan depends on the target environment, and PS-PDG enables compilers to generate binaries that adapt accordingly.

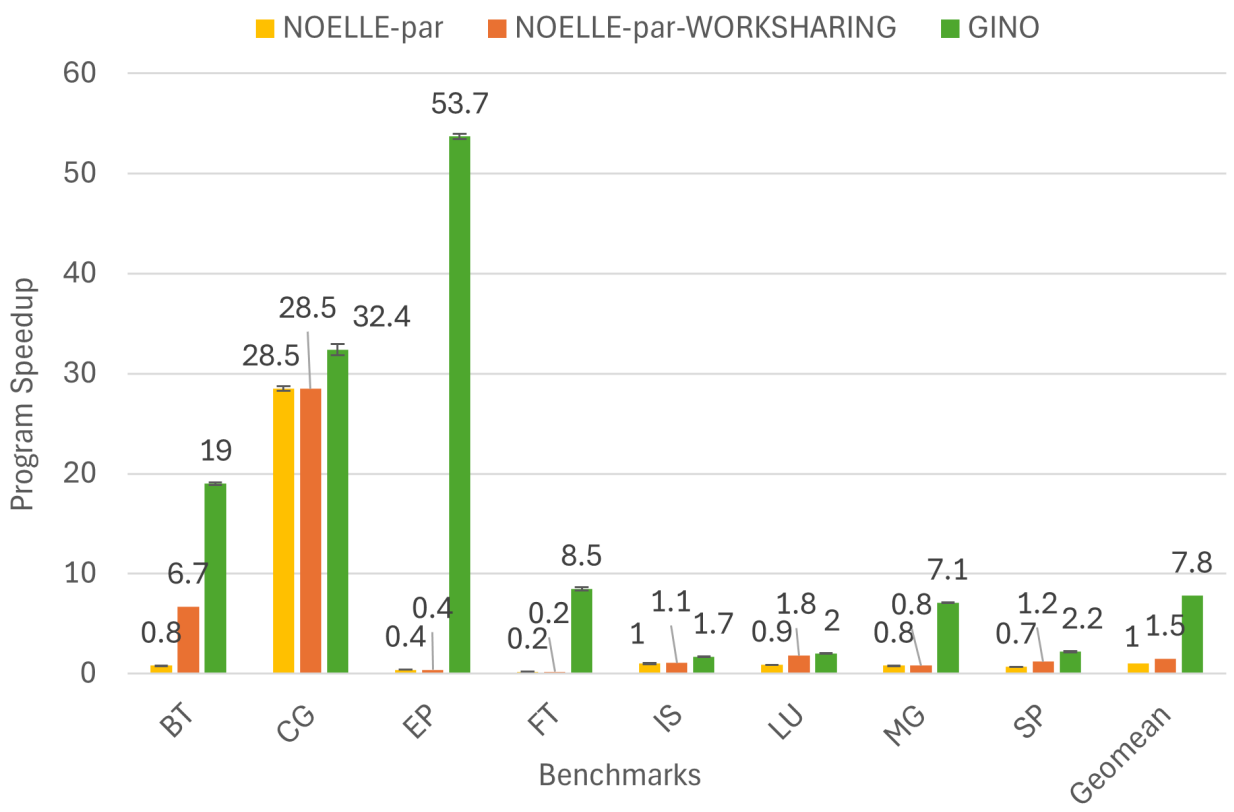


Figure 2.13: Comparing GINO with two versions of NOELLE, one relies on the PDG generated from traditional dependence analysis, the other relies on the PDG improved with work-sharing pragmas.

2.6.4 GINO Delivers On-Par Performance to OpenMP.

Fig. 2.12 compares GINO and OpenMP when both execute the developers' original parallel execution plans. **GINO achieves comparable or superior performance compared to the state-of-the-art OpenMP implementation**.

GINO achieves higher speedup in `CG` (**+17%**), where GINO-generated loop tasks get auto-vectorized by LLVM and outperform OpenMP. This optimization would require developers to explicitly specify `simd` pragmas and vectorization width hints in the OpenMP code.

GINO performs worse on `FT` (-35.8%), because Clang's OpenMP frontend benefits from spe-

cialized knowledge of OpenMP runtime APIs, enabling high-level optimizations such as parallel-region merging [8]. Those optimizations are currently unavailable in GINO.

2.6.5 The PS-PDG Serves as A Practical Abstraction for Parallel Optimization.

For decades, compilers have relied on the PDG for automatic parallelization and optimizations. Our results show that the PDG is insufficient for modern parallel optimization. The first and third bar of Fig. 2.13 compares the speedup of binaries generated by GINO (PS-PDG) and NOELLE-par (PDG) running on 56 cores. On average, GINO achieves a **7.8× speedup**, and consistently outperforms NOELLE-par across all benchmarks. One might think that this gap exists because of the alias analysis imprecision. Fig. 2.13 shows this hypothesis is false; We extended NOELLE to use the worksharing pragmas such as `omp for` to remove all spurious loop-carried dependences of the corresponding loops (as explored by prior work [47]). This is possible thanks to the OpenMP semantics, guaranteeing that each loop of this kind is DOALL. While this reduces the conservatism of the PDG, its performance still lags far behind GINO. The reason is that the PDG cannot capture the parallel semantics of `private`, `firstprivate`, and `lastprivate`. Without these, PDG-based compilers can fail to detect array reductions, allocate thread-private memory and initialize private copies. These results demonstrate that **the PS-PDG is strictly more expressive than the PDG**. By unifying compiler analyses and parallel semantics expressed by developers, the PS-PDG enables optimizations that the PDG alone cannot express or safely apply.

2.7 Related Work

Previous work [13], [14], [15] proposed graph representations of the explicit parallelism encoded in a program to help developers understand their parallelization. These representations directly capture the parallel control flow encoded in the parallel program, in contrast, the PS-PDG captures

the precise constraints of the parallel program decoupled from the encoded parallel execution plan. Other prior work [12], [16], [17], [34], [48] lowered the explicit parallelism into the IR of the compiler, introducing a new IR where the parallel execution plan can be encoded explicitly. Finally, HPVM [49] is designed specifically for heterogeneous hardware to enable optimizations while still maintaining performance portability. The PS-PDG is orthogonal to HPVM as it does not target heterogeneous hardware via a hierarchical dataflow graph or enable optimizations on the graph.

The Galois System [50] and the Kinetic Dependence Graph [51] targeted implicit parallelism in imperative languages. These approaches focus on irregular programs exploiting amorphous data-parallelism to improve performance by dynamically modifying computation task graphs at runtime.

Many functional programming languages represent parallelism either implicitly or explicitly through annotations or parallel constructs in the language itself (e.g., map) [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], [64]. These works directly translated the parallelism into a single or a few predetermined parallel execution plans (usually based on task or fork-join parallelism), where the runtime system is left with few decisions to make (e.g., number of threads).

2.8 Conclusion

This work introduces PS-PDG, a novel abstraction that unifies compiler analyses with developer-expressed parallel semantics. Our PS-PDG-empowered optimizing compiler GINO enables powerful parallel execution plan optimization going beyond those manually encoded by developers. Compared to a state-of-the-art PDG-based optimizing compiler, PS-PDG consistently delivers improved performance, demonstrating that PDG alone is limited for parallel optimization. Together, these results establish PS-PDG as a practical and effective abstraction for future optimizing compilers.

CHAPTER 3

AUTOMATING PARALLEL GRANULARITY CONTROL

3.1 Compiling Loop-Based Nested Parallelism for Irregular Workloads

Modern programming languages offer special syntax and semantics for logical fork-join parallelism in the form of parallel loops, allowing them to be nested, e.g., a parallel loop within another parallel loop. This expressiveness comes at a price, however: on modern multicore systems, realizing logical parallelism results in overheads due to the creation and management of parallel tasks, which can wipe out the benefits of parallelism. Today, we expect application programmers to cope with it by manually tuning and optimizing their code. Such tuning requires programmers to reason about architectural factors hidden behind layers of software abstractions, such as task scheduling and load balancing. Managing these factors is particularly challenging when workloads are irregular because their performance is input-sensitive. This work presents HBC, the first compiler that translates C/C++ programs with high-level, fork-join constructs (e.g., OpenMP) to binaries capable of automatically controlling the cost of parallelism and dealing with irregular, input-sensitive workloads. The basis of our approach is Heartbeat Scheduling, a recent proposal for automatic granularity control, which is backed by formal guarantees on performance. HBC binaries outperform OpenMP binaries for workloads for which even entirely manual solutions struggle to find the right balance between parallelism and its costs.

3.2 Introduction

Parallel programming continues to be hampered by the inability of compilers to translate fork-join nested parallelism in client programs into binaries that perform close to the limits of modern multicore hardware. One reason is that a binary that actually executes all possible parallel tasks described by high-level fork-join constructs (e.g., the `parallel for` of OpenMP) immediately encounters a massive overhead that results in slowdowns measured in orders of magnitude, thereby squashing all benefits of parallelism. Just spawning a parallel task/thread requires a few thousand cycles (even in a customized OS and even for the latest OS solutions), with context-switch costs being similar [65], [66]. Unfortunately, irregular workloads like sparse tensor computation and graph analytics tend to have many independent loop iterations that only take a few tens of clock cycles [67]. Spawning one parallel task per loop iteration in these workloads easily leads to slowing down execution rather than speeding it up, both due to task-spawning overheads and the barrage of context switches due to having many more tasks (e.g., loop iterations) than cores. In a run-time environment without preemption, the context-switch problem may be diminished, but the potential for excessive task-related costs remains.

On the other hand, coarsening loop iterations (known as chunking) to generate fewer tasks can easily degrade performance by starving cores of tasks. This is the old problem of *parallelism granularity control* [68], [69], [70][1]. For example, consider a loop where each iteration processes an element of an input vector, and the amount of computation performed depends on whether the element is non-zero. If the vector's non-zeros are not uniformly distributed, then static chunking leads to an unbalanced computation. Because the fork-join model requires a barrier at the end, the slowest task (the unlucky one with the most non-zeros, say) dictates the latency of the entire loop. The cores running the faster tasks (processing fewer non-zeros) are now idle for a substantial

[1] Our results in Section 3.7.7 corroborate this prior work.

period of time. The appropriate chunking to avoid this is input-dependent. Clearly, we need a dynamic solution.

The status quo places the burden on the application programmer's shoulders. Accordingly, programmers sidestep compilers by expressing a specific parallelism granularity in their code, with the hope of achieving the right balance between fork-join overhead and maximizing the program's performance on their multi-core CPU. But this decision is typically not transferable between platforms, as it depends on architecture-specific aspects related to its out-of-order capabilities (e.g., instruction-issue widths) and inter-core communication costs, as well as OS-specific aspects like thread-creation and thread-switching costs [24]. Such complications are most noticeable and difficult to manage in a certain, increasingly important class of applications that is characterized by its workload being irregular, its performance characteristics being significantly influenced by the input, and its input being drawn from a large range of possible shapes. Recent applications of this kind include those that perform significant amounts of sparse-matrix computation, e.g., in machine-learning algorithms [71], those that use a domain-specific language, e.g., for tensor algebra [20], or those that analyze large graphs [27]. For applications featuring such irregular parallelism, the granularity-control problem threatens to result in either a platform-specific, hard-to-maintain, and costly codebase, or binaries that do not leverage the full potential of the underlying architecture.

Recent research produced an alternative approach to granularity control, known as heartbeat scheduling [25]. It is the first approach that *provably* controls the overheads of parallelism *automatically*, *without embedding platform-specific aspects* into the program's code. At a high level, the properties proven for heartbeat scheduling are twofold: task-related costs are well amortized and the asymptotic amount of parallelism in the source program is preserved. Heartbeat scheduling works by postponing the decision of generating additional parallelism to run-time and makes that decision online, at a regularly occurring event, called the heartbeat. A heartbeat happens at a

fixed rate while the program executes, and enables the program to continuously adapt to changing parallelism. The essential takeaway is that heartbeat scheduling lets programmers express *all* possible fork-join parallelism in their algorithm while allowing the runtime to decide which portion to materialize and when.

Heartbeat scheduling implementations currently require programmers to write their code, at least in part, in assembly. The reason is that the implementations require the code to be structured in an unconventional style, even though the program's code does not make granularity control decisions itself. A mapping to such a bespoke program structure from high-level fork-join programs is not supported by today's compilers. This gap leaves heartbeat scheduling in the hands of a few highly capable programmers with significant time to invest in restructuring their code. We seek to democratize heartbeat scheduling, allowing all programmers to reap its benefits. This requires a compiler that can automatically translate high-level fork-join constructs into a binary that is amenable to heartbeat scheduling at runtime.

This work describes the first compiler capable of automatically generating binaries that are compatible with heartbeat scheduling. The heartbeat compiler (HBC) consumes an ordinary C/C++ program with high-level fork-join constructs, like those available in OpenMP or Cilk. HBC then automatically deconstructs the code into tasks that can be further split when driven by heartbeat events. The heartbeat linker (also introduced by this work) modifies the generated assembly both to link a signaling mechanism to generate heartbeats and to link with the heartbeat runtime. Binaries generated by HBC from programs with irregular workloads have significantly better performance than binaries generated by a more conventional OpenMP compiler. Compared to sequential execution, HBC boosts the performance of irregular workloads from 14.2× (OpenMP) to 21.7× (Heartbeat) on a 64-core Intel-based machine (Fig. 3.4). Finally, the performance of binaries generated with HBC, which compiles them in just a few seconds, is comparable to what was previously

obtained by manually writing heartbeat scheduling binaries over several months [26].

The contributions of this work are:

- We introduce the first compiler capable of realizing heartbeat scheduling in a fully automated manner, generating a binary from given C/C++ programs with loops, which are written using conventional high-level fork-join constructs.
- We introduce the first linker capable of automatically embedding into a binary the rollforwarding mechanism, which prior work proposed for heartbeat scheduling.
- We introduce a new algorithm for automatically generating all possible parallel tasks from nested loops that can be parallelized by heartbeat scheduling. This improvement includes the generation of what we call the *leftover tasks* that were not generated before.
- We design, implement, and evaluate the first compilation pipeline to automatically generate binaries that are capable of performing heartbeat software polling (i.e., continuous checking if an event happened) with little overhead.
- We design, implement, and evaluate the first interrupt-based mechanism to deliver heartbeats on Linux with a custom kernel module to reduce the latency of delivering heartbeats.
- We compare for the first time two signaling mechanisms that are both able to deliver heartbeats. This comparison suggests a counter-intuitive result: software polling is as good as interrupt-based mechanisms.

3.3 Background

To summarize key background concepts, we use the following running example: the sparse-matrix by dense-vector product (*spmv*, shown in Fig. 3.1(a)). We use the implementation of *spmv* that

expects its input matrix to be represented in compressed sparse-row format. The inner and outer loops are parallelized by annotating them as DOALL loops (and treating the reduction variable `result` appropriately). A DOALL loop is a loop where all iterations can run in parallel, in a fork-join execution model, without communication between them. The DOALL decoration can be done using, e.g., the OpenMP `parallel for` construct.

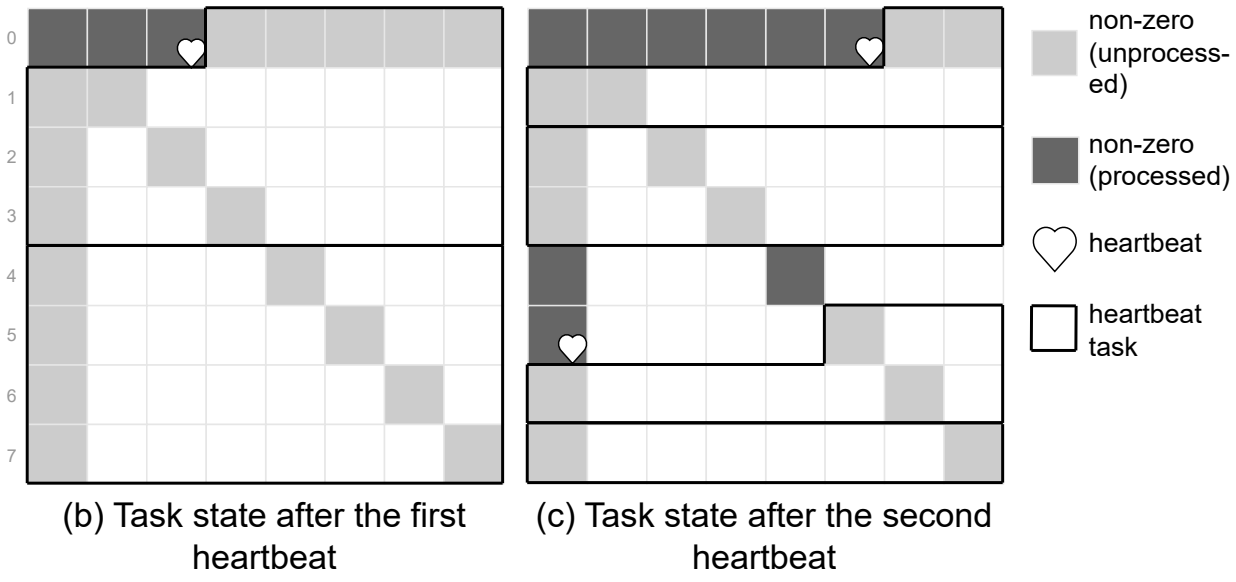
Iterations of a DOALL loop run in parallel by creating and executing tasks. First, tasks are created by partitioning the loop iterations (one task per iteration at the finest granularity). Then, tasks are dispatched at run-time to parallel threads (via thread-specific queues) that run on the parallel cores of the underlying architecture (one thread per core). Notice that the number of tasks can safely (and typically does) exceed the total number of threads or cores.

The parallelism granularity-control problem. If *spmv*'s DOALL loops were to maximize parallelism and spawn a task for each iteration, *spmv* would risk losing its parallel speedup to the overhead costs of creating and managing tasks. Task overheads can be amortized by assigning multiple subsequent loop iterations (called chunk) to a task. This operation is called *chunking* and the number of iterations assigned to a task is called the *chunk size*. OpenMP (like other languages) allows programmers to manually specify the chunk size of a loop. But tuning the chunk size risks pruning away too much parallelism, and even if the program is well-tuned, there is a risk of overfitting [23]. Achieving consistent granularity control is further challenged by irregularity in workloads. For example, our *spmv* is highly input dependent, and consequently, based on the sparsity pattern of the matrix, the concentration of parallelism may fluctuate between its two loops.

Heartbeat scheduling. Heartbeat scheduling achieves granularity control for nested fork-join constructs in an adaptive manner by ensuring that each task is amortized against a specified amount of useful work in the program. That amount is determined by the heartbeat rate, a system-wide

```
void spmv(int n,    double *val,      // number of rows and non-zeros
          int *row_ptr, int *col_ind, // non-zero row and col indices
          double *in, double *out) {  // in and out vectors
  for (int i = 0; i < n; i++) { // row loop
    double result = 0.0;
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++) { // col loop
      result += val[j] * in[col_ind[j]];
    }
    out[i] = result;
  }
}
```

(a) *spmv* serial implementation in C/C++



(b) Task state after the first heartbeat

(c) Task state after the second heartbeat

Figure 3.1: *spmv* and the heartbeat execution model.

parameter that controls the rate at which tasks are spawned. Let us see it in action by stepping through *spmv* when called on a well-known challenge input, the arrowhead matrix, whose diagonal, first row, and first column are filled with non-zero elements. Suppose, for explanatory purposes, the heartbeat rate takes as much time as is needed to process 3 non-zero elements. Now, after (serially) processing up to the 3$^{\text{rd}}$ element of the first row, our initial task has been running for enough time to be interrupted by a heartbeat. When it arrives, the interrupt kickstarts a *promotion*. Promotion is the process used by heartbeat scheduling to activate latent parallelism (in our case,

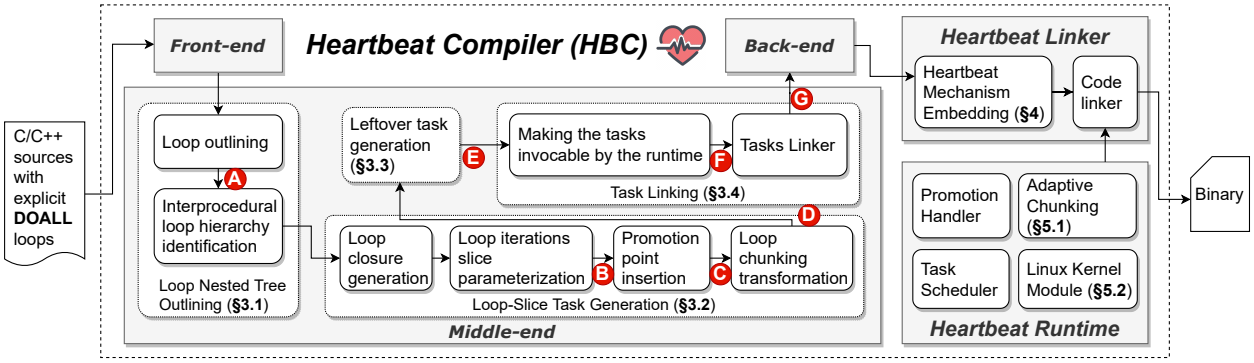


Figure 3.2: Compilation pipeline of HBC including a custom linker and runtime.

the remaining loop iterations) held by a running task.

Under the hood, this two-step process of interrupt, followed by promotion, is implemented by the coordination of two mechanisms. The interrupt is driven by a hardware-based timer interrupt. Promotion happens downstream of an interrupt, where it reifies the loop context, divides up loop iterations, and parcels out the pieces to new tasks. This latter mechanism is called the *promotion-ready program point (PRPPT)* [26]. These two mechanisms are linked together by the application of a classic technique for synchronizing in the presence of interrupts known as *rollforwarding* [72] (see §3.5 for details).

To ensure parallel scalability, heartbeat scheduling assigns the highest priority to outermost parallelism, a policy we call the ***outer-loop-first*** policy. Let us see the policy in action in our running *spmv*, where we left off. As it enters its promotion handler, our running task sees latent parallelism in the outer loop, making that loop the target for promotion. The result is depicted in Fig. 3.1(b), where we see an even division of the remaining iteration space split into two sub-tasks. This process continues in a recursive fashion. The diagram in Fig. 3.1(c) shows a future program state after two more heartbeats, treated by each of the previous two tasks.

Current heartbeat scheduling implementation. Recent work proposed TPAL, the state-of-the-art, low-level model for heartbeat scheduling, but TPAL does not address automation: each TPAL benchmark started as a C++ source program and was manually modified by inserting PRPPTs, as needed. Further manual intervention was needed to mitigate performance issues related to PRPPTs, which required loop chunking by hand. Definitions of the PRPPT handler functions themselves require custom logic, which had to be written by hand. The back-end of TPAL used a semi-automatic rollforward compiler, which required significant human intervention. Finally, although crucial for practical performance, TPAL provided no comprehensive solution for driving interrupts. Its approach was based on special support in a research kernel. A similarly effective solution for a commodity OS is lacking.

3.4 The Heartbeat Compiler (HBC)

The HBC pipeline (Fig. 3.2) takes C/C++ source files, where parallelism is specified by DOALL annotations, and lowers the program via a series of passes, resulting in a binary that implements heartbeat scheduling. The front-end of our HBC pipeline is an extension to the compiler `clang` that identifies all DOALL annotations in the program and emits them in the form of custom LLVM IR metadata terms. Our metadata-enhanced LLVM IR is consumed by a series of passes we implemented as extensions of LLVM's middle end. These lowering passes isolate DOALL loops by outlining them into separate functions and building a representation of their nesting structure (§3.4.1). This nesting structure is used by our middle-end passes to automate the generation of PRPPTs.

To do so, the middle-end computes the closure of each outlined loop such that the generated function can be invoked to execute a specific set of subsequent loop iterations (e.g., from 5^{th} to 9^{th} iteration) while being able to promote parallelism at runtime (§3.4.2). The result is a set of func-

tions (one per DOALL loop) that can be invoked as parallel tasks, called *loop-slice tasks*. After generating the loop-slice tasks, the HBC middle-end generates the *leftover tasks* (§3.4.3). According to the outer-loop-first policy of heartbeat scheduling, the decision of splitting the parallel task T_o related to an outer loop L_o while running its z^{th} iteration can be made during the execution of the j^{th} iteration of one of its inner loops L_i. To maximize parallelism, our work splits T_o into three parallel tasks whose code is generated at compile time. The first one is the execution of the loop-slice task of L_o to execute the first half of the iterations left of L_o. The second one invokes the same loop-slice task but executes the second half of the iterations left of L_o. The last task will execute the remaining computation of the z^{th} iteration of L_o, which includes the remaining computation of the current invocation of L_i (from its $j + 1^{th}$ iteration to the end) as well as the code from the end of L_i to the end of the z^{th} iteration of L_o. We call this last task a *leftover task*. As shown in Fig. 3.1(b), the first two loop-slice tasks generated by HBC cover the remaining iterations of the `row` loop from `1` to `7`. And a third leftover task covers the rest of the iterations of `col` loop from `3` to `7`, and the remaining computation after invoking the `col` loop of row `0`. That is, `out[i] = result`.

After generating both loop-slice tasks and leftover tasks, the HBC middle-end links them into the original code (§3.4.4). This is done by first modifying the above tasks to make them controllable and invocable by the HBC runtime (so the runtime can perform parallelism promotions), and then it replaces the original IR code of the target loops with invocations of the loop-slice tasks where the slice specified is the entire iteration space of the related loop. Hence, if no promotion happens at runtime, the execution stays sequential.

After the passes in the middle-end, we lower the program to binary code, using an off-the-shelf back-end available in the LLVM codebase (e.g., the intel *x86_64* back-end). The output program then reaches our heartbeat linker, the final stage of our pipeline. The heartbeat linker enables the

heartbeat to be seen by the running program by injecting hooks from the runtime into the program's IR.

3.4.1 Loop Nested Tree Outlining

The middle-end starts by outlining all DOALL loops. For each loop of this set, a conventional data-flow analysis identifies its live-in and live-out variables. Then, a function is created to copy the loop in it as done by prior work [73], [74], [75], [76], [77]. This function has live-ins of the loop as parameters and its live-out variables are passed as references. Now the loop can be executed by invoking the function with proper parameters. The resulting functions are then modified to replace any nested DOALL loops with a call to their outlined versions. Live-ins and pointers of the live-outs of a nested DOALL loop are passed via parameters of the injected call. Live-outs are allocated on the caller's stack.

HBC needs to represent the original nesting relation of DOALL loops because it is needed by the heartbeat runtime to implement promotions. Hence, HBC extends the loop-nesting-relation analysis in LLVM to make it inter-procedural. The result of this analysis is a directed graph where nodes are loops and edges represent the nesting relation from a parent loop to one of its children (similarly to [78]). HBC prunes this graph to remove all nodes that do not represent DOALL loops. The result is a tree because the original DOALL loops formed a tree (they came from the same original function).

Our solution follows the general approach of OpenMP compilers, where DOALL loops are outlined first into separate functions before reaching the middle-end. An alternative solution to this problem is to compute the loop-nesting relation of the original code and then implement an ad-hoc outliner transformation that also generates the mapping from the original loops to the new ones that now belong to different functions. This solution would be equivalent to the one we implemented.

We chose our solution to allow us to re-use the general-purpose outliner transformation already available in conventional compilers, which does not provide the loop mapping mentioned above.

The inter-procedural loop nesting tree is used to compute the IDs of each DOALL loop. The ID of a DOALL loop is a pair `(level, index)`, where `level` represents the nesting level of that loop, starting from `0` for the root loop, and `index` represents the position of the loop within its respective nesting level, incrementing from `0` onwards. The root loop has an index value of `0`. In *spmv*, the `row` loop has the identification pair `(0, 0)` and the `col` loop has the identification `(1, 0)`.

3.4.2 Loop-slice task generation

Following Fig. 3.1(b), when the first heartbeat interrupts *spmv*'s `col` loop, a promotion handler is called to spawn parallel tasks. To seed the task, we need the loop's environment. The challenge is that the loop we need to promote (the `row` loop) is not the loop that was in execution at the time a heartbeat was received (the `col` loop). To solve it, we need to pass the environment of the `row` loop to the inner `col` loop and its promotion handler. To do so, HBC generates code to couple each loop with a data structure that captures its closure, its iteration space, and its induction variable. We call this structure the *Loop-Slice Task (LST) context*. LST contexts of a given DOALL loop of a loop nesting tree are allocated before the outermost loop of that tree (the root loop) is invoked. These LST contexts are passed down to all nested loops as a set. When a nested loop invokes the promotion handler, all loops' LST contexts are accessible to the promotion handler to seed any task that runs any parent loop.

Loop closure generation. HBC first modifies the signature of the outlined DOALL loops of each loop-nesting tree such that all parameters are replaced by a pointer to a set of LST contexts, which

includes the LST context of the invoked loop itself. The outlined loop function is now considered a loop-slice task. Next, HBC generates code to load live-ins, live-outs, and iteration space to run from the corresponding LST context and replaces all values previously read from the function parameters with loaded values.

Promotion point insertion. HBC inserts a call to the promotion handler at the latch of a DOALL loop, for all DOALL loops, to enable promotion within a loop-slice task. The latch of a loop L is a basic block within L that is the predecessor of the header of L [79]. DOALL loops only have one latch.

The promotion handler returns whether a promotion happened (`1`) or not (`0`). When a promotion happens, the promotion handler returns only when the execution of all remaining loop iterations has been completed. Therefore, HBC adds a conditional branch to read the value returned by the promotion handler to exit the loop when a promotion happens.

Loop chunking transformation. The promotion handler inserted by HBC can degrade performance because the call breaks up the control flow of the target loop, blocks compiler optimizations, and can impose dynamic costs (depending on which mechanism the heartbeat linker uses to drive heartbeats). To mitigate such costs, the loop chunking transformation modifies the target loop to invoke the promotion handler every S number of iterations (called chunk size). This is obtained by creating within the target loop a sub-loop L_s whose body contains only the original code of the target loop.

The loop chunking transformation needs to guarantee that the promotion handler is invoked every S iteration. This requires extra code for when the number of iterations executed by the original loop is not a multiple of S. In more detail, a chunk can be partially finished within a given invocation of the target loop (e.g., S is bigger than the total number of loop iterations).

Therefore, a task needs to track how many iterations remain to be executed till the full completion of a chunk between (potentially several) loop invocations. To do so, each task maintains a private counter R (initialized to S), and the number of iterations of L_s is set to be the minimum between R and the number of iterations left to finish the current invocation of the target loop. The chunking transformation adds a check after L_s, which will execute after L_s finishes its iterations. It compares the number of iterations C executed by L_s to R, and it invokes the promotion handler only if they match (and reinitializes R to S). Otherwise, it updates R to be $R - C$ (called chunk size transferring).

The loop chunking transformation is applied to every innermost DOALL loop of a loop nesting tree. Finally, the chunk size S is determined by a dynamic technique (§3.5).

3.4.3 Leftover task generation

This compilation step generates all possible leftover tasks of a loop nesting tree of DOALL loops. The generation of such tasks allows HBC to generate additional parallelism compared to prior work, as the leftover task of a promotion can now run in parallel with the other two tasks generated with it. This opportunity was not explored by prior work because tasks were written manually and the number of leftover tasks can grow quadratically with the number of loops in a nesting tree. Hence, it is too much to ask from a programmer to write them all. However, HBC generates them automatically while keeping the code size under control by sharing code between leftover tasks.

Iterating over possible leftover tasks. The possible leftover tasks depend on the shape of the loop nesting tree. They are generated using Algorithms 1 and 2. Algorithm 1 takes as input a loop nesting tree of DOALL loops and all LST contexts. The algorithm identifies the need for generating a leftover task for a pair of loops (L_i, L_j). The loop L_i represents the loop that gets a

Algorithm 1 Generating all leftover tasks between loop pairs.

Input: t: Loop nesting tree
Input: s: Set of all LST contexts of all loops
1: **for all** $l : t.getLeaves()$ **do**
2: $p \leftarrow l.getParent()$
3: **while** p **do**
4: GENERATELEFTOVERTASK(l, p, s)
5: $p \leftarrow p.getParent()$

heartbeat that leads to a promotion; L_j represents the loop that gets split. To identify the above set of pairs, Algorithm 1 iterates over the leaves of the loop-nesting tree (line 1). For each leaf l, it iterates over its ancestors starting from its parent (lines 2-5). For each ancestor p, the pair (l, p) is identified and the associated leftover task is generated by invoking Algorithm 2 (line 4).

Code generation of a leftover task. For each (L_i, L_j) pair found in Algorithm 1, HBC creates a leftover task that will execute in that case. Thus, HBC implements Algorithm 2 taking as input L_i and L_j that represent the case where L_i gets a heartbeat and L_j gets split. It also takes as input the LST contexts of the loops included in the current loop nesting tree. The output is the leftover task t for the (L_i, L_j) case.

The algorithm starts by creating a new empty leftover task t (line 2) followed by increasing the induction variable by 1 of the LST context used by L_i when it gets a heartbeat (lines 3-4). The algorithm then adds the code to invoke the loop-slice task of L_i starting from its next iteration until the end (line 5). At this point, Algorithm 2 has generated the code for t to complete the current invocation of L_i.

What is left for the algorithm is to append the code that composes the work between the end of L_i to the end of the current iteration of L_j, referred as *tail work*. To do so, it iterates over the ancestors of L_i starting from its parent to the ancestor just before reaching L_j (lines 8-14). For each ancestor p, Algorithm 2 adds code to get its LST context, which contains the current induction

Algorithm 2 Generating a leftover task between two loops.

Input: L_i: Loop that gets a heartbeat
Input: L_j: Loop that gets split
Input: s: Set of all LST contexts of all loops

```
 1: function GENERATELEFTOVERTASK(L_i, L_j, s)
 2:     t ← new LeftoverTask()
 3:     t.addCode(c ← s.getLSTContext(L_i))
 4:     t.addCode(c.increaseIV(1))
 5:     t.addCode(call LOOPSLICETASK(L_i, c))
 6:     prev ← L_i
 7:     p ← L_i.getParent()
 8:     while p ≠ L_j do
 9:         t.addCode(c ← s.getLSTContext(p))
10:         t.addCode(TAILWORK(p, prev, c))
11:         t.addCode(c.increaseIV(1))
12:         t.addCode(call LOOPSLICETASK(p, c))
13:         prev ← p
14:         p ← p.getParent()
15:     t.addCode(c ← s.getLSTContext(L_j))
16:     t.addCode(TAILWORK(L_i, prev, c))
```

variable of p (line 9). It then places the tail work of p, which composes of the code after invoking the previous ancestor till the end of the body of loop p using p's LST context (line 10). After that, Algorithm 2 increases p's induction variable by 1 and invokes p's loop-slice task passing its updated LST context (lines 11-12). Finally, the algorithm adds code for the tail work of L_j using its LST context after invoking its previous ancestor (lines 15-16).

3.4.4 Task linking

At this point of the compilation pipeline, the HBC middle-end has generated all possible tasks that could run in parallel. To be performant the heartbeat runtime also needs to quickly access the right triple of tasks that will instantiate when a heartbeat happens. This triple depends both on the innermost loop that has received the heartbeat and the loop that gets split. Because of this, the

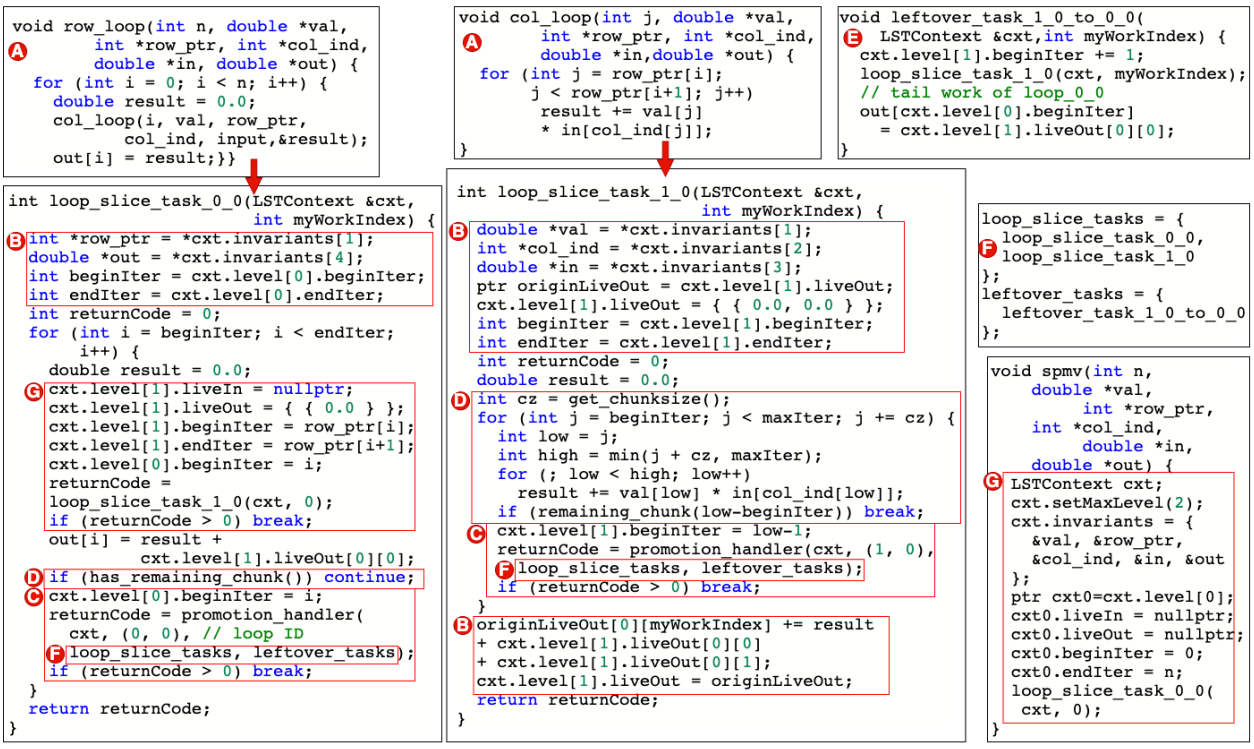


Figure 3.3: Sequence of transformations of *spmv* that are performed by the HBC's middle-end. We used C++ code for readability and illustrative purposes. HBC performs these transformations in LLVM IR. The white letter in a red cyrcle attached to the code surrounded by a rectangular red box indicates where in the pipeline of Fig. 3.2 that code is added.

HBC middle-end ends with the next two steps. First, all tasks are organized to enable an efficient identification of the triple of tasks. Then, the tasks are linked into the program by allocating and initializing the necessary LST contexts of the DOALL loops of a loop nesting tree. This allocation is performed just before jumping to the header of the root of the corresponding loop nesting tree.

Making the tasks invocable by the runtime. To help the heartbeat runtime to quickly find the right task to invoke for a promotion, this step takes advantage of the structure of the loop IDs described in §3.4.1. The middle-end allocates a two-dimensional array for every loop nesting tree with DOALL loops. This array is called the *loop-slice task array* of a loop nesting tree. Each value

of this array is a pointer to the function of a loop-slice task. The array is passed as a parameter to the promotion handler inside all loop-slice tasks that compose this nesting tree. The `level` and `index` that compose the ID of a loop are used as the row and column of the loop-slice task array. The value obtained at that row and column is the pointer of the loop-slice task with that specific loop ID. The loop-slice task array is allocated as globals and are statically initialized.

Leftover tasks also need to be efficiently retrieved by the heartbeat runtime. To this end, the HBC middle-end allocates a hash table called the *leftover task table* where it returns the pointer to the function of a leftover task from a pair of loop IDs. The first ID of the input pair is the loop that has received the heartbeat. The second ID is of the loop that gets split. A perfect hashing function is generated at compile time to perform this mapping. Finally, the leftover task table is allocated as global, statically initialized, and the signatures of all tasks are modified to take a leftover task table.

Tasks linker The last step of the HBC middle-end is to allocate the LST contexts for all loop nesting trees with DOALL loops. For every loop-nesting tree, it replaces the code of the loop at the root of the tree with a call to the equivalent loop-slice task after preparing the initial environment and specifying the whole iteration space for that loop inside its LST context. Any call to a nested loop inside a loop-slice task is replaced by the call to its corresponding loop-slice task, with the environment and the iteration space set by the parent loop. Fig. 3.3 shows the full transformation of *spmv*.

3.5 Heartbeat Linker

HBC implements two mechanisms for handling heartbeats: software polling and hardware interrupts. Software polling proactively reads the timestamp register to decide if a heartbeat has arrived.

Hardware interrupts invoke heartbeat processing on receipt of a timer interrupt. HBC uses software polling by default as it delivers better average performance on an unmodified Linux platform (§3.7.5), but allows the user to select either heartbeat mechanism.

Software polling injection. The linker injects the polling function at PRPPTs (§3.3) and guards the promotion handler call with a conditional branch. The polling function reads the timestamp register (i.e, `TSC` for x86) to tell whether a heartbeat has arrived. If it has, the promotion handler gets called, generating parallelism. Otherwise, it does not.

Hardware interrupt enablement via rollforwarding compilation. In software polling, we always pay the cost of the polls. In contrast, using hardware timer interrupts avoids this, but an interrupt can arrive at any assembly instruction, not just at PRPPTs, and thus rollforwarding [72] is needed.

Conceptually, we implement rollforwarding by having the hardware interrupt trigger an instruction pointer switch from the "source" version of the object code (which contains no polls) to the "destination" version (which contains the polls). A mapping table (the "rollforward table") from source to destination instruction addresses is included in the binary and used by the hardware interrupt mechanism (§3.6.2) to find the appropriate "destination" address to switch to. An inverse table (the "rollback table") is also needed.

Our rollforward compiler (RFC) automates the process of producing the source and destination code (at the assembly level), and tables. RFC is a source-to-source translator that operates over the assembly intermediate file (the "`.s` file"). To generate the source, every input line is prepended with a new label we generate based on its line number (e.g. `__RF_SRC_42`). Lines that involve polling have their instruction elided. To generate the destination, every input line is repeated, but now prepended with a label that corresponds to the source line (e.g. `__RF_DST_42`). In the

destination, the polling instructions are left in place. Finally, we emit the tables, mapping between the newly introduced labels (i.e., we add `__RF_SRC_42` $\leftrightarrow$ `__RF_DST_42`). GNU `ld` resolves all the labels to addresses. Special care is taken to handle numerous edge cases and other issues.

Despite the apparent complexity of this transform, the novelty of RFC is that it operates entirely using regular expressions, instead of requiring compiler backend changes or assembler modifications. RFC comprises 250 lines of dense Perl. It takes < 1 second to translate each of our benchmarks.

3.6 Heartbeat Runtime

This section describes the two runtimes that we designed for the two heartbeat solutions we implemented for HBC: software polling and interrupt-based solutions.

3.6.1 Software Polling using Adaptive Chunking (AC)

AC dynamically updates the chunk size of a leaf loop to reduce the number of wasted polls per heartbeat. The initial chunk size is set to `1`. The runtime runs a sliding window algorithm [80] where the loop chunk size is updated only at the end of the window when a given number of heartbeats (called the window size) have been received. Each worker thread keeps track of how many times the polling function is invoked since the last heartbeat. On each heartbeat, the runtime logs the number of polls made during that heartbeat interval. After the number of heartbeats that compose the window, a thread records the minimum number of polls in the log since the beginning of the window. Then, it computes the ratio of the minimal poll count to a ***target polling count***. This ratio is then multiplied to the current chunk size to form the new chunk size (minimum `1`) for the worker thread.

3.6.2 Hardware Interrupt-based Solutions

HBC supports hardware interrupt-based heartbeats using either portable user-level code or a Linux kernel module.

Interrupt Ping Thread for Purely User-level Operation. This mechanism uses the POSIX `SIGALRM` signal to drive the heartbeat via the *ping thread* model from earlier work [26].

Kernel Module for Accelerated Operation. In earlier work [26], an alternative mechanism was proposed and evaluated that used the x86 APIC hardware timer and inter-processor interrupt (IPI) mechanisms directly. While this dramatically improves heartbeat accuracy, precision, and scalability, it requires the non-trivial inclusion of the application directly into a specialized kernel. As a middle ground, we developed a Linux kernel module that provides many of the same benefits, while requiring no application changes.

Our kernel module configures one core to use the kernel's `hrtimer` interface, a thin veneer over the APIC timer, and allows the runtime to configure heartbeat rates. A timer interrupt invokes the module, which in turn broadcasts an IPI to all current heartbeat-enabled cores. Each of these determines if the current interrupted user thread is a heartbeat application thread. If it is, the handler searches the rollforward table for the interrupted "source" RIP and finds its corresponding "destination". It then edits the return address of its own interrupt frame so that on IRET, control returns to the destination address, switching to the rollforward code.

Unlike a ping thread, this structure operates mostly in kernel, directly using the hardware. It also avoids the high cost of general-purpose POSIX signal injection. An event requires only 3800 cycles (user→kernel→user) on our system.

3.7 Evaluation

This section evaluates the first fully automatic solution for heartbeat scheduling, HBC. After describing the experimental settings, this section shows the higher performance obtained by HBC compared to the clang-based OpenMP compiler for irregular workloads. Then, this section compares the performance of the binaries automatically generated by HBC against those manually generated in the TPAL work [26]. Our results suggest that the automation done by HBC to generate binaries preserves the performance obtained by the intense manual work done by TPAL. By leveraging our automatic solution, this work shows the first empirical comparison between two signaling mechanisms for heartbeat scheduling: software polling and hardware interrupts. Our results counter-intuitively show that the former is as good as the latter, even when the latter is implemented with custom OS support. This result has the potential to help broaden heartbeat scheduling's adoption as it can now be used on the off-the-shelf Linux OS without sacrificing performance. This section also evaluates the need for adapting the chunk size at run-time and evaluates how well our runtime solution performs. Finally, this section ends by comparing HBC and OpenMP for regular workloads.

3.7.1 Experimental Settings

Next, we describe our test-bed, where we performed all empirical evaluations described in this work. We will open source HBC and all benchmarks/results described in this section.

HBC, TPAL, and OpenMP compilation. HBC builds upon NOELLE [81], a compilation framework that augments LLVM with additional dependence-oriented abstractions like the Program Dependence Graph [82]. We ported NOELLE to LLVM 14 because the public version only supports LLVM 9. HBC, OpenMP, and TPAL parallelize the same set of loops for all benchmarks. All

loops parallelized are DOALL.

LLVM 14 is the underlying compilation infrastructure used for all of our results. The TPAL and OpenMP binaries are generated using clang as included in LLVM 14. This enables a fair comparison between the three systems as they share a significant portion of their compilers, leaving parallelism granularity control to be the main difference between them. All benchmarks are compiled using `-O3 -march=native` with vectorization passes disabled (as done in evaluating TPAL [26]).

Benchmarks. We evaluated HBC on two sets of benchmarks. Table 3.1 lists the benchmarks we used to evaluate HBC, along with the input used, their regularity, and whether they were also evaluated by TPAL.

The first set of benchmarks is composed of all the iterative (eight) benchmarks that the prior work TPAL [26] was evaluated on. As HBC aims to replace all significant manual efforts behind the manual generation of the assemblies used in TPAL [26], it is important to evaluate HBC against the original benchmarks used by this prior work. To this end, we targeted the iterative benchmarks from this prior work since HBC targets loops and not recursive functions. In more detail, we imported the same implementation of benchmarks previously evaluated by TPAL [26]. We have also used the same matrix generator implementation for generating the different inputs of the benchmark *spmv*.

We have also evaluated HBC on a second set of benchmarks to further demonstrate HBC's effectiveness in targeting irregular workloads. To this end, we evaluated ten extra benchmarks whose computations are irregular. The rest of this subsubsection describes them. The first one is the benchmark *cg* from the NAS benchmark suite; we used its implementation in NPB3.0 [45]. We chose only *cg* from this suite because it is the only benchmark in the NAS suite whose input can

lead to an irregular workload. In more detail, we used a non-synthetic and irregular input because most synthetic inputs of *cg* (including the one included in the suite) lead to a regular workload. We used the input cage15 [83], which is a real-world matrix from the Univerity of Florida sparse matrix collection [84]. The second benchmark we targeted is *mandelbulb* [85], which is an extension of *mandelbrot* to handle 3D inputs.

We have also evaluated eight more benchmarks from two domains: sparse tensor computation and graph analytics applications. We targeted these two domains because applications in these domains tend to be highly irregular due to their computational sparsity. For the sparse tensor computation, we imported *TTV* and *TTM* from TACO [20], a domain-specific language for sparse tensor algebra. TACO asks programmers to supply an index expression of a tensor algebra kernel and specify each tensor's storage format (*dense* or *sparse*) in the index expression provided. This high-level expression is compiled to C/C++ code where the main kernel computation is a loop nest, within which all loops are DOALL. TACO disables nested parallelism by annotating OpenMP pragmas only on the outermost loop. We manually changed the C/C++ code generated by TACO to have multiple versions of the same benchmark by enabling (or disabling) parallelization of the nested loops. This enabled us to evaluate how both an OpenMP compiler and HBC handle nested parallelism. We used the same inputs that were used to evaluate TACO [20]; these inputs are obtained from the FROSTT Tensor Collection [88]. We use the storage format of *dense* for the first dimension of the input tensor and *sparse* for the rest of the dimensions.

For graph analytics applications, we imported *bfs*, *cc*, *pr*, *cf*, *pr-delta* and *sssp* from GraphIt [27], a domain-specific language for writing graph analytics. GraphIt enables programmers to write high-level computation with the freedom to apply various optimizations such as parallelization, cache partitioning, and data layout optimizations. GraphIt compiler transforms high-level graph computations to C/C++ code with OpenMP pragmas. The main kernel from the above graph ana-

Benchmark	Source	Input	Regularity
OpenMP pragmas are generated by programmers			
mandelbrot	TPAL [26], [86]	512 × 1024 × 40k	irregular
spmv-arrowhead		150 million rows 450 million nonzeros	irregular
spmv-powerlaw		16.7 million rows 402 million nonzeros	irregular
spmv-random		6 million rows 600 million nonzeros	regular
floyd-warshall		4k × 4k	regular
kmeans		10 million elements	regular
plus-reduce-array		100 billion elements	regular
srad		10k × 10k	regular
mandelbulb	3D Mandelbrot [85]	100 × 200 × 300 × 400	irregular
cg	NAS [45]	cage15 [83]	irregular
OpenMP pragmas are automatically generated			
ttv	TACO [20], [87]	nell-2 [88]	irregular
ttm			irregular
bfs	GraphIt [27], [89]	Twitter [90]	irregular
cc			irregular
pr			irregular
cf		LiveJournal [84]	irregular
pr-delta			irregular
sssp			irregular

Table 3.1: The benchmarks used in this work, with input used, their regularity, and if they were also evaluated by TPAL.

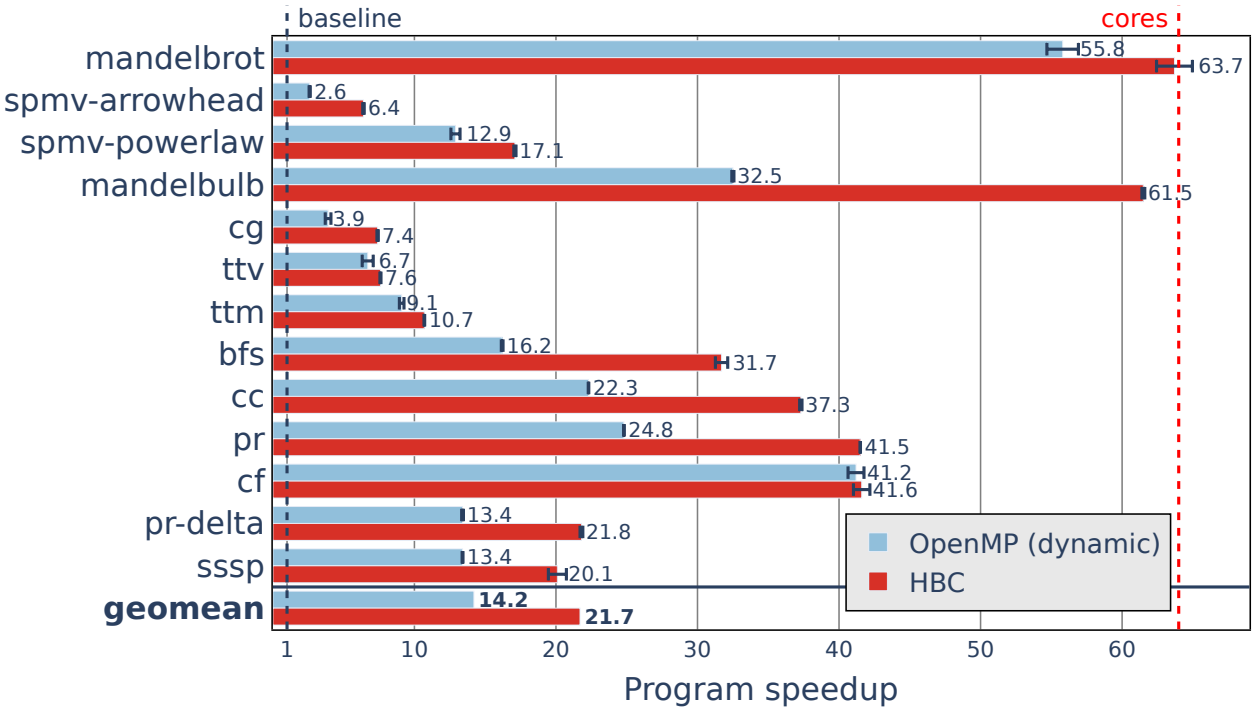


Figure 3.4: 64-core evaluation comparing OpenMP dynamic scheduling and HBC over irregular workloads.

lytics benchmarks is often a DOALL loop that goes through every node of a graph and applies an update function on outgoing neighbors of that node. We enabled parallelization for all benchmarks and used *DensePull* as the direction of applying the update function. We used social network graphs Twitter [90] and LiveJournal [84], which exhibit high irregularity for the input data. These graphs are the same inputs that the authors of GraphIt have used to evaluate their work.

Platform. All of our results are computed using an AWS instance that runs Linux kernel 5.19.0 equipped with an Intel Xeon Platinum 8375C processor featuring 64 cores over two sockets running at 3.0 GHz, with 2 MiB L1i, 3 MiB L1d, 80 MiB L2 and 108 MiB L3. Both hyperthreading and turbo-boost are disabled throughout the evaluation. We set the heartbeat rate to 100 μs following the tuning process proposed in TPAL [26]. We report the median result over 100 runs.

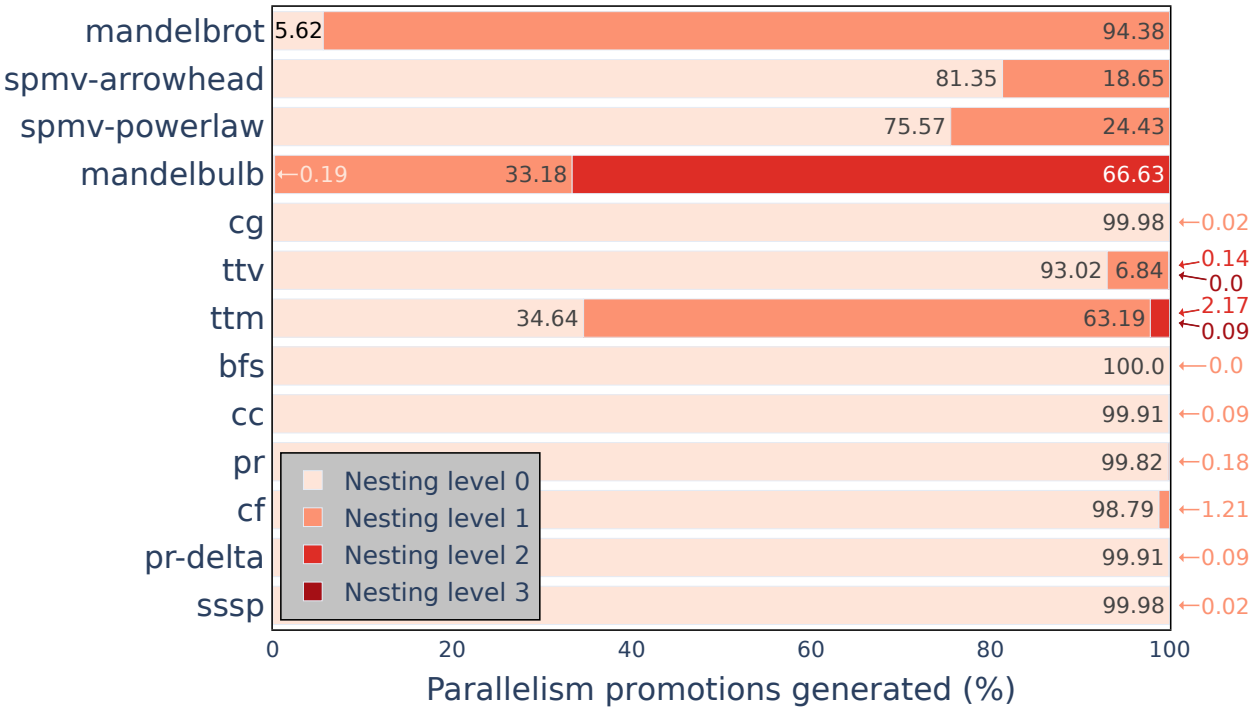


Figure 3.5: Parallelism is generated at different loop nesting levels.

3.7.2 HBC Outperforms OpenMP for Irregular Workloads

HBC binaries outperform OpenMP for all irregular workloads. Fig. 3.4 shows the speedups of these two sets of binaries on 64 cores. The speedup (on average) increases from 14.2× to 21.7× when switching from OpenMP to HBC. This shows HBC's effectiveness in generating optimized parallel binaries that benefit from heartbeat scheduling. HBC outperforms OpenMP by adapting the parallelism at run-time, following the heartbeat approach. Fig. 3.5 shows that HBC enables the program to generate parallelism at different loop nesting levels upon a heartbeat. This suggests that using a static granularity decision is sub-optimal as the best granularity depends on the input data. This also demonstrates that the granularity decisions can be offloaded to the compiler and runtime, reducing the burden on programmers.

3.7.3 HBC Automates TPAL's Prior Work

HBC automates the code generations and optimizations done manually in the TPAL work [26]. Hence, it is important to understand whether the automation performed by HBC delivers the same binary quality that was manually obtained by TPAL.

Fig. 3.6 shows the speedups of the HBC and TPAL binaries in our testbed. **HBC automatically delivers comparable or superior performance compared to the state-of-the-art and manually-generated code implemented using TPAL**. The TPAL-based heartbeat manual transformation uses rollforwarding to switch between the serial version of the code and rollforwarded code to promote parallelism. Its runtime reserves an extra interrupt ping thread to signal the arrival of heartbeats to all active worker threads. Similar to our solution, TPAL inserts the promotion handler at the end of each loop body. TPAL binaries perform chunking on all leaf loops using a static chunk size determined with a tuning process and defined at compile time per benchmark. These per-benchmark manual tunings performed in TPAL's binaries are fully automated in HBC (on top of the code parallelization, generation, and optimization). Next, we discuss in detail why HBC outperforms TPAL on some benchmarks.

HBC generates more parallelism. HBC obtains higher speedups than TPAL for *kmeans* (+13.7%), *mandelbrot* (+24.4%) and *srad* (+44.8%) because HBC runs in parallel all three tasks generated per promotion (two loop-slice tasks and a leftover task), while TPAL only runs two tasks in parallel, placing the third one in its critical path. According to the authors, this was done because running the leftover task in parallel would have further increased the already high time it took them to manually generate the code. The limitation was due to having the leftover task still referencing the execution environment (e.g., stack) of the parent task; in other words, the leftover task in TPAL did not have a complete closure. Notice that in the worst case, the number of possible static leftover

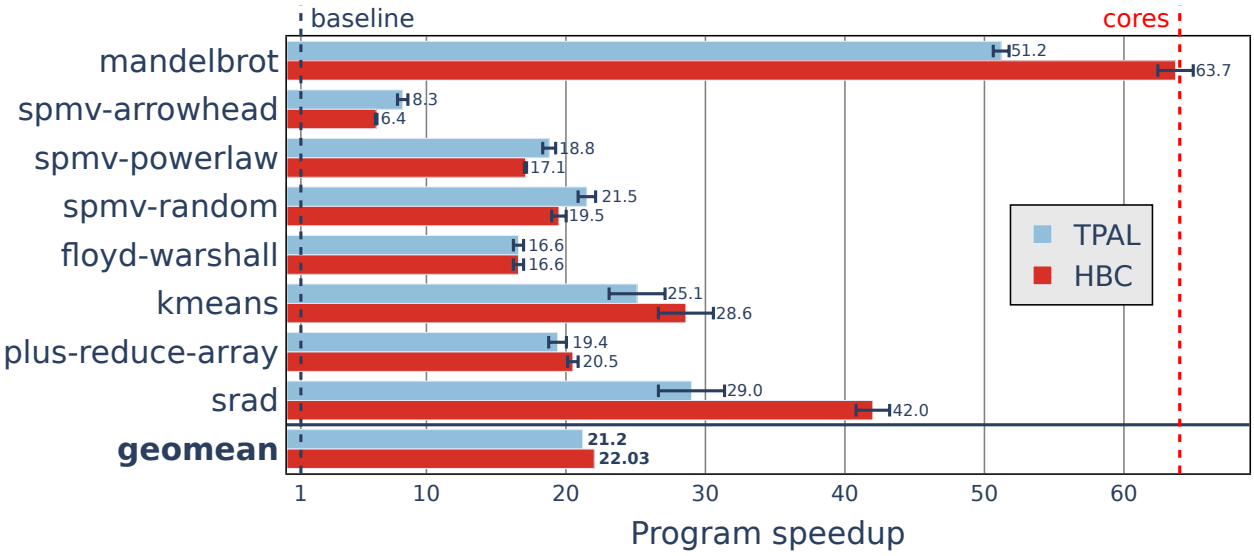


Figure 3.6: HBC automatically delivers comparable performance compared to the manually-generated TPAL binaries. These are the loop-based benchmarks used in [26] running on 64 cores.

tasks grows quadratically with the number of nested loops; writing all of them manually with their closure is impractical. Because HBC is fully automatic, it is able to automate the closure generation of all possible leftover tasks, which unlocks an extra level of parallelism. This additional level of parallelism becomes important when the time it takes to sequentially execute all leftover tasks is non-negligible, which is the case for *kmeans*, *mandelbrot* and *srad*.

HBC is penalized for chunk size transferring. In all *spmv*-based benchmarks, HBC performs worse than TPAL. The worst is *spmv-arrowhead*, which shows a 22% slowdown. This slowdown is caused by continuously tracking and updating the remaining chunk size before making a poll. This overhead is in the critical path for this input, starting from the second row. TPAL relies on interrupts and thus does not generate this extra overhead. The chunk size transferring cost decreases for HBC once there are more non-zeros to be processed per row, as shown in *spmv-powerlaw* (-9%) and *spmv-random* (-9%).

3.7.4 HBC Overhead Analysis

Next, we analyze the extra work performed by the HBC binaries compared to the baseline. To this end, we compile all the loops without enabling parallelism promotion and thus avoid the cost of task scheduling. Since now the program runs sequentially, all extra work that the HBC binaries perform comes from loop outlining, closure generation, loop chunking, promotion insertion, chunk size transferring and polling overhead. Fig. 3.7 shows the overhead results and their breakdowns.

Only *spmv-arrowhead* and *spmv-powerlaw* have significant overhead (others have less than 10% overhead). The extra work added by HBC for all benchmarks includes the parent loop passing via memory the iteration space for the nested loop to run. This overhead is negligible when the invoked loop runs many iterations, which is true among all these benchmarks. HBC inserts a promotion handler call at the end of the body of a loop, followed by a check on the call's return value. This does not generate much overhead because HBC applies chunking to all leaf loops. Hence, promotion insertion overhead becomes insignificant compared to the amount of work performed inside a chunk.

When the loop getting invoked repeatedly runs only a small number of iterations, the overhead generated by HBC cannot be amortized and becomes significant. This is because for *spmv-arrowhead* (+58.46%) and *spmv-powerlaw* (+22.14%), the binary needs to do chunk size transferring each time a leaf loop is invoked, while only processing a few non-zero elements. For the same reason, the overhead of promotion insertion at the outer loop accumulates quickly when a small leaf loop is invoked, and gets added into the critical path.

3.7.5 Software Polling is as Good as Hardware Interrupts

HBC supports both software polling and hardware interrupts for handling heartbeats. The former is the default one and it is the mechanism used for all results shown in this work outside this sub-

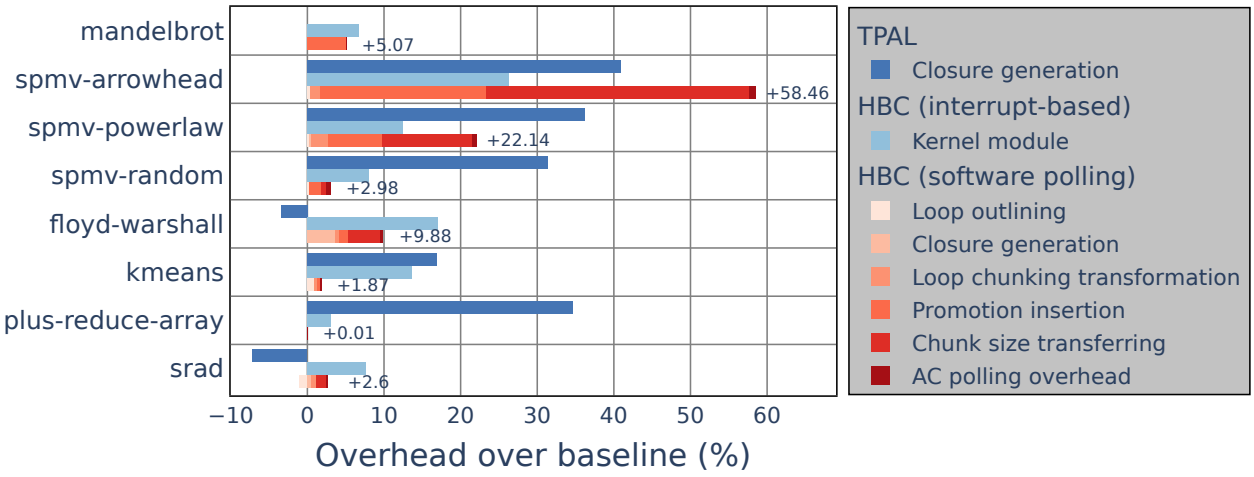


Figure 3.7: Overhead of HBC (w/ and w/o software polling) and TPAL.

section. We first analyze the overhead of software polling and then compare it with the interrupt-based mechanism implemented using rollforwarding.

Software polling overhead. Software polling has the potential to broaden the adoption of heartbeat scheduling as it does not require any hardware or OS support. The main problem with software polling is its overhead. We show, however, that adding a few optimizations (e.g., loop chunking) significantly reduces the polling overhead to the point that software polling becomes an interesting design choice. To show this, we start with an unoptimized implementation of a simple algorithm for software polling, showing its high overhead. Then, we slowly improve its quality, leading towards a more and more optimized algorithm, until we reach the final algorithm (with low overhead) described in §3.6.1, which is the default implementation of HBC.

Our simplest implementation disables the loop chunking transformation in HBC. Hence, a poll is performed at every loop iteration. The "No chunking" bar of Fig. 3.8 shows the overhead (in clock cycles) obtained by this simple implementation compared to the same baseline of Fig. 3.6. This is computed by running the generated code without doing promotions; hence, the execution stays sequential even if we perform the polls. The overhead is significant and completely erases

the benefits of parallelism, causing a 7.5x slowdown.

Our second implementation adds loop chunking using static chunk sizes used in TPAL [26]. Hence, a poll is performed per chunk. The bar "Static chunking" of Fig. 3.8 shows that the overhead of this algorithm is significantly lower.

Finally, we measured the overhead of polling generated by the algorithm described in §3.6.1 where the chunk size is decided by our runtime. This is the bar called "Adaptive Chunking" of Fig. 3.8. This low overhead led to the speedups shown in Fig. 3.6.

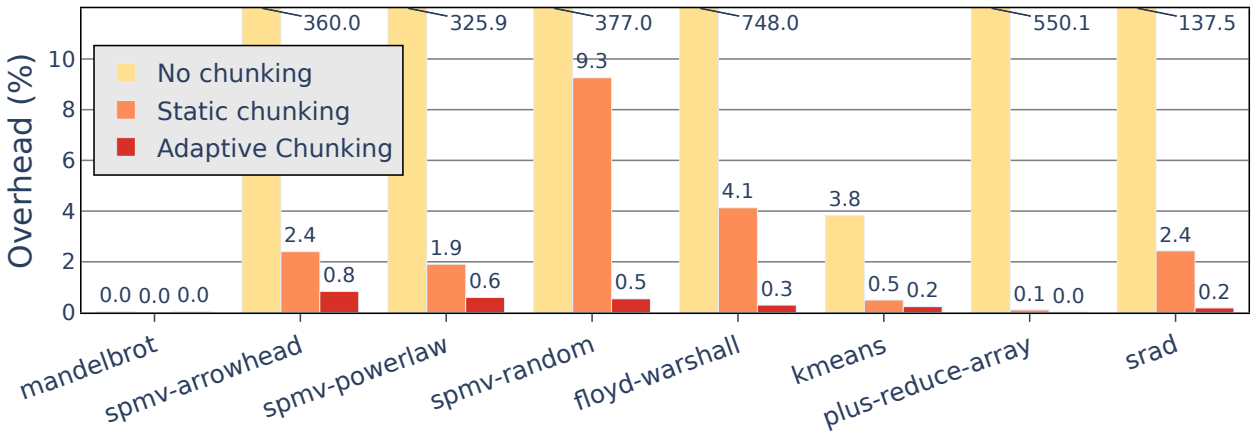


Figure 3.8: Software polling overhead with different chunking mechanisms. Both static and adaptive chunking significantly reduce overhead, with adaptive performing best.

Software polling and interrupt-based solutions are comparable. Prior work on heartbeat scheduling relied on hardware interrupts. HBC implements this solution as well as software polling, so for the first time, these techniques can be compared within the context of heartbeat scheduling.

TPAL work relies on a dedicated thread to send heartbeat interrupts to worker threads. HBC is the first compiler that automated the code generation needed by this technique (§3.5). To measure the efficiency of this solution, we used a user-space thread to send heartbeat interrupts as done

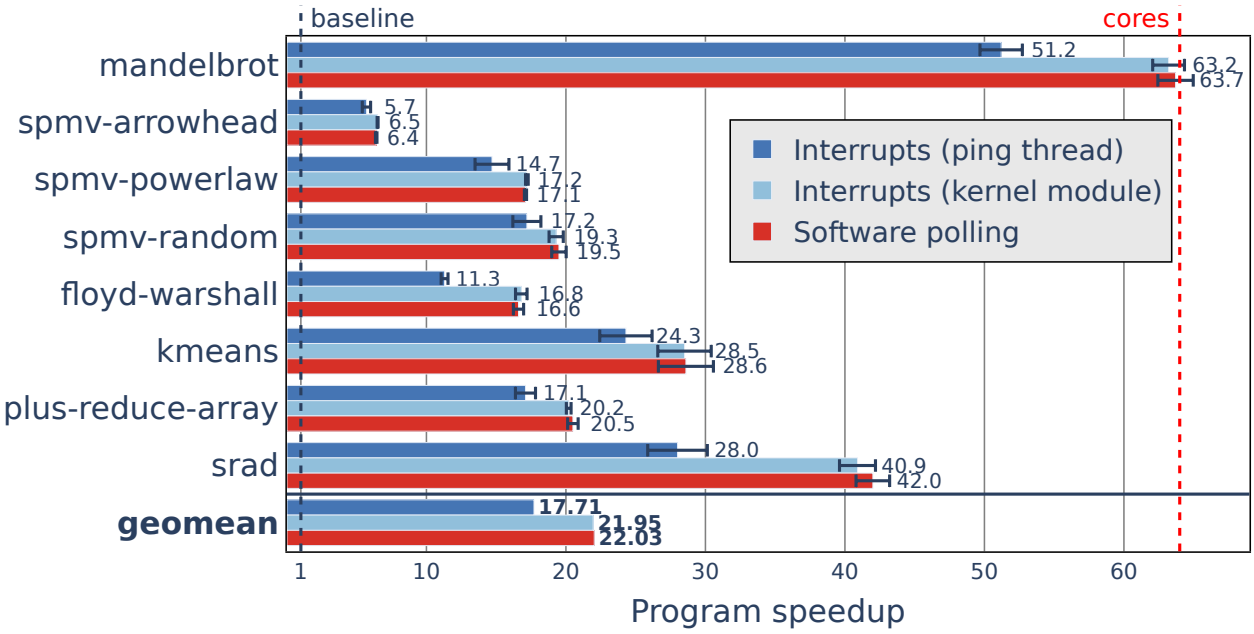


Figure 3.9: Software polling is as good as interrupt-based mechanisms.

by prior work [26]. Results obtained with this technique are shown by the bar ”Interrupts (ping thread)” of Fig. 3.9. This figure shows that software polling boosts the speedup obtained by this prior work technique from 17.7× to 22.0×. This is because of the high signaling overheads created by the interrupt ping thread, making it unable to deliver heartbeats at a rate that matches the desired one. This results in missing up to 45% of the heartbeats (and therefore generating less parallelism).

Because of this high overhead, we have implemented a custom OS support to reduce the latency of delivering heartbeats (§3.6.2). This is the bar ”Interrupts (kernel module)” of Fig. 3.9. While this new implementation is better than the one adopted by prior work, its performance is still compatible to that of software polling. This suggests that heartbeat scheduling can be embedded in programs without special OS support, increasing adaptability.

Why is software polling comparable? To understand why software polling is comparable to the interrupt-based solution adopted in prior work, we compared the overhead of these two techniques.

Fig. 3.7 shows such a comparison where promotion is disabled for both techniques; hence, the promotion cost and task scheduling cost are not included in the overhead.

Software polling pays (on average) less overhead than the best interrupt-based technique (the one with custom OS support). This is because the overhead for the latter per heartbeat is almost two orders of magnitude compared to a single poll. For each heartbeat, it can take 3800 cycles (§3.6.2). Instead, by our measurement, a poll takes 50 cycles. Ideally, a single poll per heartbeat is enough for the software polling technique. Even when 10 polls are performed per heartbeat, the total cost is still an order of magnitude less than the cost of an interrupt.

3.7.6 Chunking Needs to be Adapted at Runtime

The best chunk size is input-dependent and therefore it needs to be adapted at run-time. Next we are going to use *mandelbrot* as an example of code that highlights this need.

The need for adapting. Fig. 3.10 shows the execution time of *mandelbrot* using heartbeat scheduling on 64 cores with two different inputs, one with high latency (input 1) and the other with low latency (input 2). As we increase the static chunk size used from 2^0 to 2^9, input 2 performs better while input 1 performs worse. Therefore, the best chunk size setting for *mandelbrot* is input-dependent, and there is no single chunk size setting that is optimal across all inputs.

A common scenario in many applications is that an important loop gets invoked repeatedly and possibly with different inputs. To further show the limitations of setting the chunk size statically, we studied *mandelbrot* in this scenario by invoking its main loop 10 times, using input 1 and 2 five times each. We measured the time it takes to sequentially run these invocations (compiling the code with clang) and we consider this time to be the baseline for this experiment. The speedup obtained by HBC forcing a static chunk size or by adapting it at run-time are shown in Fig. 3.11.

Adapting the chunk size at run-time boosts the speedup from 17× (static chunk size set to 2) to 28×, an increase of 64%.

The benchmark *mandelbrot* is just an example that shows the need for adapting the chunk size at run-time. To show this, let us now observe *spmv* with different matrix inputs and use the number of non-zeros per row to show the impact of having different latencies per loop iteration. AC is included in our HBC solution and is needed when there are different latencies between loop iterations. This is because the code needs to perform the right amount of polls to minimize their overhead while avoiding missing heartbeats. The higher the latencies, the smaller the chunk size needs to be. Fig. 3.12 shows how the chunk size changes in reaction to loop iteration latency.

Adapting chunk size requires the right target. To adapt the chunk size of a loop, our runtime implements a sliding window algorithm (§3.6.1), controlled by two parameters: target polling count and window size. All HBC results shown in this work use 8 for both the target polling count and the window size for all loops in all benchmarks, which we will justify next.

Target polling count. Our primary goal is to avoid missing heartbeats while minimizing polling overhead, thus we compare how many heartbeats are detected to the total number of heartbeats generated. We perform this experiment at different target polling ratios. Our results in Fig. 3.13 show that a too-low target polling count results in missing a significant number of heartbeats (almost 50% for *spmv-powerlaw*), which leads to less parallelism. On the other hand, having a too-high target polling count results in extreme polling overhead. Setting this value to 4 captures over 99% of the heartbeats while paying a small overhead. Similar results are obtained for other benchmarks. Hence, we used the target count 4 for all loops in all benchmarks for every result shown in this work.

Window size. Theoretically, this parameter could impact the reaction speed of changing the chunk size at run-time to changes of the latency of loop iterations. In practice, our results show negligible impact for this parameter on all benchmarks used. Hence, we used window size 8 (anything ≥ 2 would have worked fine) for all results shown in this work.

3.7.7 Manual Granularity Control for OpenMP Compilers

All OpenMP-related results described in prior sub-sections are obtained by parallelizing only the outermost loops of a benchmark (and by using the default chunk size for each parallel loop, which is one). This is recommended as a good practice to control the scheduling overhead. However, if an OpenMP compiler can perform granularity control automatically (like HBC), then a better solution for programmers is to expose the parallelism of all DOALL loops of a benchmark without worrying about the scheduling overhead. This is what we did when we used HBC in the prior sub-sections.

OpenMP compilers rely on the programmer to make granularity control decisions such as determining which loops to parallelize and specifying the chunk size of a loop being parallelized. To show that the decisions that OpenMP programmers of our target benchmarks are reasonable, this sub-section performs the following experiments. We changed (manually) the selection of which loops to parallelize as well as their chunk size while using the OpenMP compiler to see if these changes can improve performance. We performed these experiments on all manually implemented benchmarks, where programmers have the full control over where and how OpenMP pragmas are generated. Our results show that both tuning the chunk size and parallelizing all DOALL loops (instead of parallelizing only the outermost loops) degrade the performance.

Tuning the chunk size. OpenMP programmers can manually perform granularity control of the parallelism of their code by changing the chunk size of a parallel loop. While tuning the chunk size for performance is often input-dependent and labor-intensive (and therefore not ideal), it is important to know how much performance can be gained by it (to understand how well the default chunk size performs). To this end, we run a sensitivity analysis over the chunk size of the OpenMP dynamic scheduler for all manually implemented benchmarks. As shown in Fig. 3.14, all benchmarks get their performance degraded when keep increasing the chunk size except for one benchmark, *cg*, whose performance is slightly improved. The worse performance is because when tasks get coarsened, chunking results in less balanced execution for irregular workloads.

Parallelizing all DOALL loops. The OpenMP results described in prior sub-sections have been obtained by parallelizing only the loops that the original author of the benchmarks has parallelized (which is only the outermost DOALL loops). However, some nested loops of the target benchmarks could be parallelized as DOALL as well. Enabling their parallelism (by adding more OpenMP pragmas) leads to finer-grained parallelism. To understand how OpenMP compilers handle more (fine-grained) parallelism, we ran an experiment where we exposed fork-join parallelism for all DOALL loops (as we did for HBC) for all manually implemented OpenMP benchmarks. This is achieved by explicitly invoking `omp_set_max_active_levels` routine at the beginning of the code and tagging all DOALL loops with OpenMP pragmas (using the *dynamic* scheduler and its default chunk size) of all loop nests. As shown in Fig. 3.15, all benchmarks, except for *cg*, have their performance significantly degraded when all DOALL loops are parallelized. *spmv-arrowhead* and *spmv-powerlaw* did not finish in time (2 hours). *mandelbulb* crashed because the OpenMP runtime failed to allocate necessary resources for 64 workers. The performance degradation is because enabling nested parallelism by parallelizing all DOALL loops in OpenMP generates too

many tasks, which overloads the system and wipes out the benefit of parallelism.

3.7.8 When Heartbeat Scheduling is Inefficient

We compared HBC and OpenMP for regular benchmarks to understand whether heartbeat scheduling can become the solo policy. HBC performs worse than OpenMP in this case, as shown in Fig. 3.16, because heartbeat scheduling incurs extra overhead that is not justified for workloads that have well-balanced loop iterations. The only exception is *kmeans*, which HBC outperforms OpenMP static scheduler by more than 50%. HBC obtains this performance gain because it is able to reduce an array of elements between all tasks in parallel. Instead, the OpenMP implementation performs the reduction operation over an array sequentially by the main thread, and this adds to the critical path of the computation. We used the unmodified OpenMP implementation *kmeans* from the Rodinia benchmark suite [91], which is the same implementation used by TPAL [26].

The static policy outperforms all dynamic policies (including heartbeat) for most regular benchmarks we studied. This is because a static decision about how to parallelize the code generates minimal run-time overhead. Therefore, an ideal compiler should include both heartbeat and static scheduling.

3.8 Related work

Lazy scheduling and clone optimization. Lazy scheduling (LS) dates back to the proposal for lazy task creation of Mohr et al. [92], and was later adapted to the specifics of the work-stealing scheduler [93], resulting in the *clone optimization*. Our HBC runtime uses the clone optimization to avoid paying synchronization costs (e.g., the execution cost of atomic instructions) between tasks that are executed by the same thread. For HBC, in more detail, when a heartbeat happens at time t, while a task k is executing, k splits into three tasks that cumulatively perform the same

computation that k still had to do to complete at time t. These three tasks execute in parallel and the continuation of k (the code to execute after k ends) can execute only when all three tasks end, which requires the three tasks to synchronize. However, when the three tasks are executed by the same thread that was executing k, they will execute sequentially and in the right order (thanks to the thread-local deque). In this case (the fast path), our runtime avoids performing the synchronization. However, if one of these three tasks is stolen by another thread and therefore it will execute on another core, then our runtime performs the necessary synchronization (the slow path).

Granularity control. A classic alternative to LS and heartbeat scheduling is granularity control (GC), a family of approaches that operate in a proactive manner to amortize task overheads. Like with LS, in GC, the program switches between serial and parallel modes of execution. However, switching in GC is guided by *predicted* amounts of *future* of work (LS and heartbeat scheduling are guided by *measured* amounts of *past* work). Manual granularity control [94] remains commonplace in spite of its limitations [23]; there have also been various proposals for automatic granularity control [95], [96], [97], [98], [99], [100]. Oracle-guided GC [101], [102] is the first to be backed by formal guarantees that bound task-related overheads and guarantee preservation of parallelism. However, these bounds require certain assumptions on the dynamic behavior of the program, which may be difficult to know in general, and the approach is not fully automatic. In particular, it requires application programmers to write annotations at fork points in the program that specify *abstract cost functions*, which require manual effort and are sometimes not practical.

Prior work has performed granularity control applied to a single program to dynamically adapt on changes to the hardware resources available while the parallel program executes (e.g., due to having multiple programs running on the same machine at the same time) [103], [104], [105],

[106]. Compared to heartbeat scheduling, these approaches react to change in available resources rather than changes of available parallelism of the target program.

Compiler support for parallelism Tapir [12] is a recent proposal to embed fork-join parallelism into the LLVM IR. The motivation is to unlock conventional middle-end optimizations (i.e., optimizations that target LLVM IR code) in LLVM to work within parallel constructs (e.g., loop invariant code motion across parallel loops). These optimizations are otherwise only applicable to serial regions of programs. Although our HBC extends the LLVM IR, our focus is to enable granularity control rather than to unlock existing optimizations of LLVM's middle end. Finally, notice that HBC can be extended to target TAPIR rather than LLVM IR. HBC and Tapir should compose well.

3.9 Conclusion

Obtaining efficient parallel execution still requires programmers to manually control the parallelism granularity of their programs. This leads to either platform-specific and hard-to-maintain codebases or inefficient programs. Heartbeat scheduling can solve this problem, but it requires the software to fit to an unconventional structure. This work introduces HBC, the first compiler that automatically transforms C/C++ programs with nested fork-join constructs into such unconventional structure, unlocking heartbeat scheduling for a wide programmer base. HBC outperforms the clang-based OpenMP compiler for irregular workloads, where even programmers (not tools) struggle to optimize.

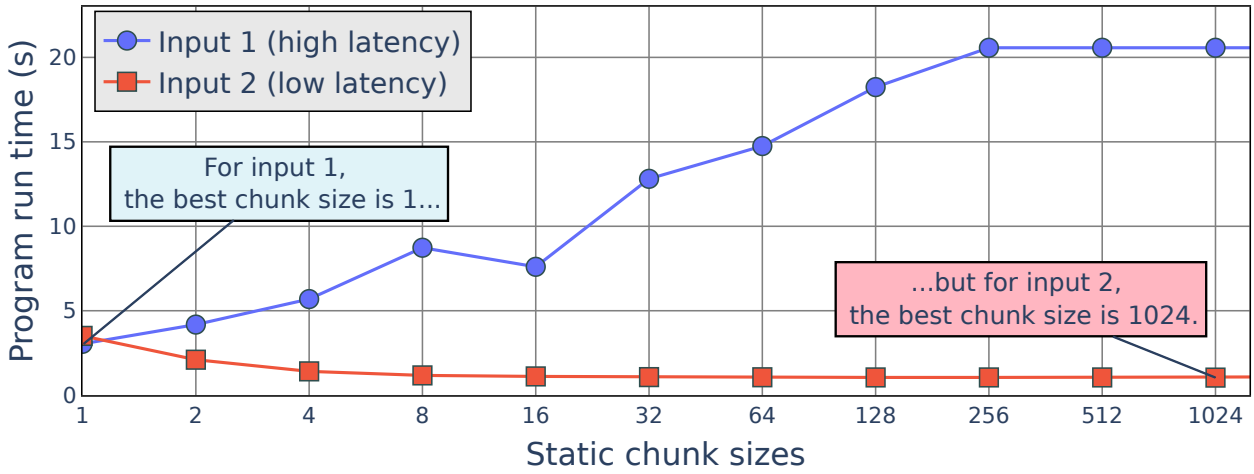


Figure 3.10: Optimal chunk size for *mandelbrot* is input-dependent.

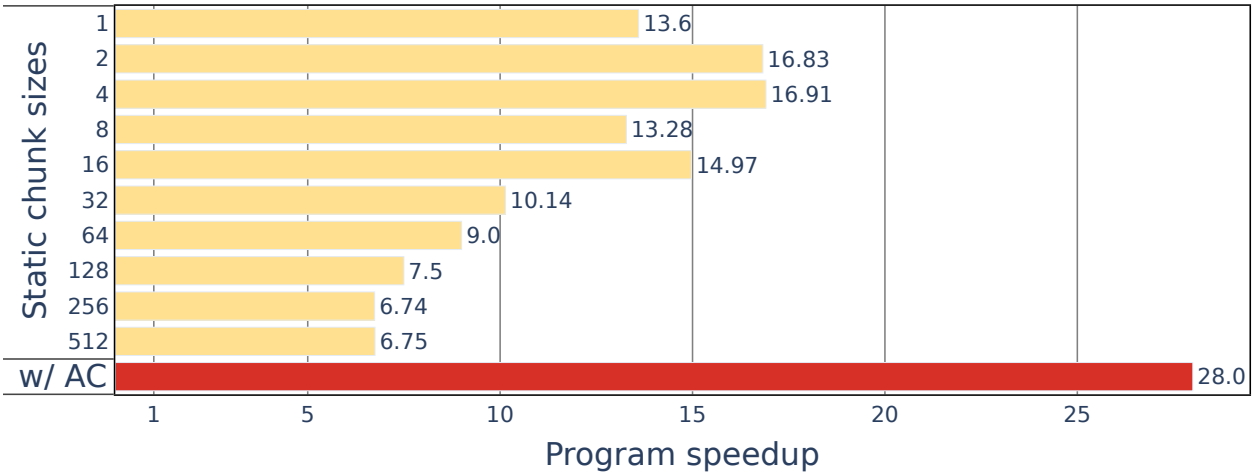


Figure 3.11: Speedup of invoking mandelbrot 10 times with different inputs, using static chunk size versus adapting chunk size at run-time.

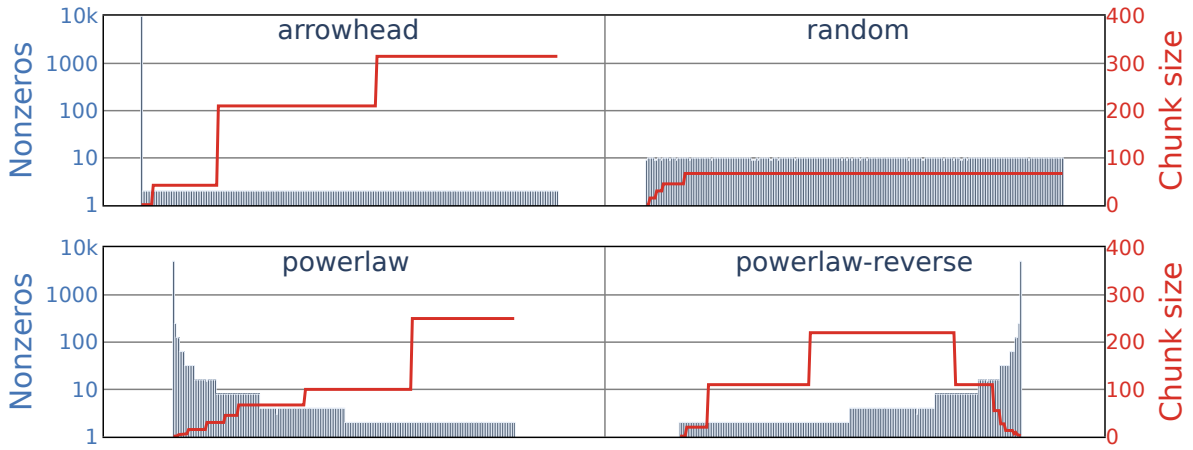


Figure 3.12: Visualization of Adaptive Chunking.

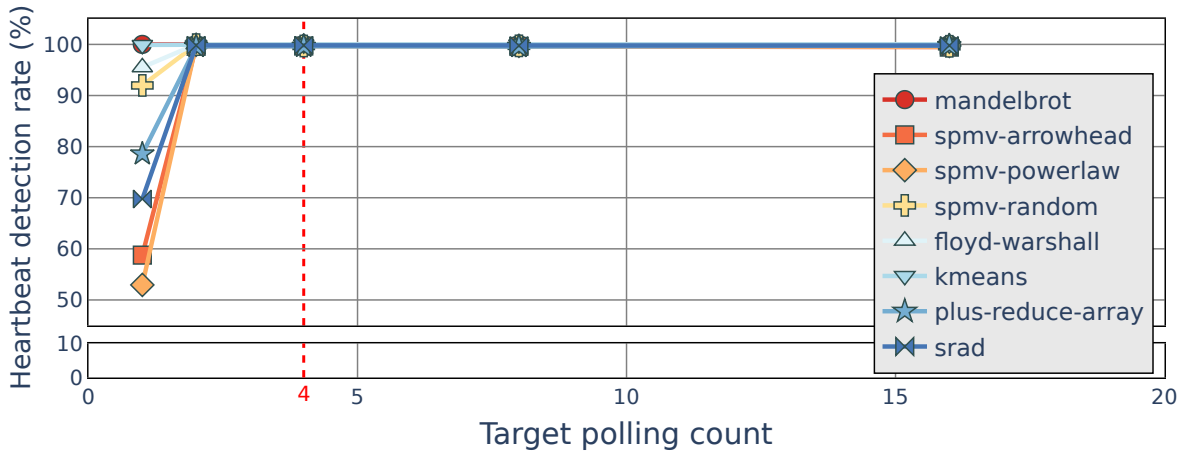


Figure 3.13: Heartbeat detection rate via AC. A target polling count value of 4 is efficient in capturing almost all heartbeats.

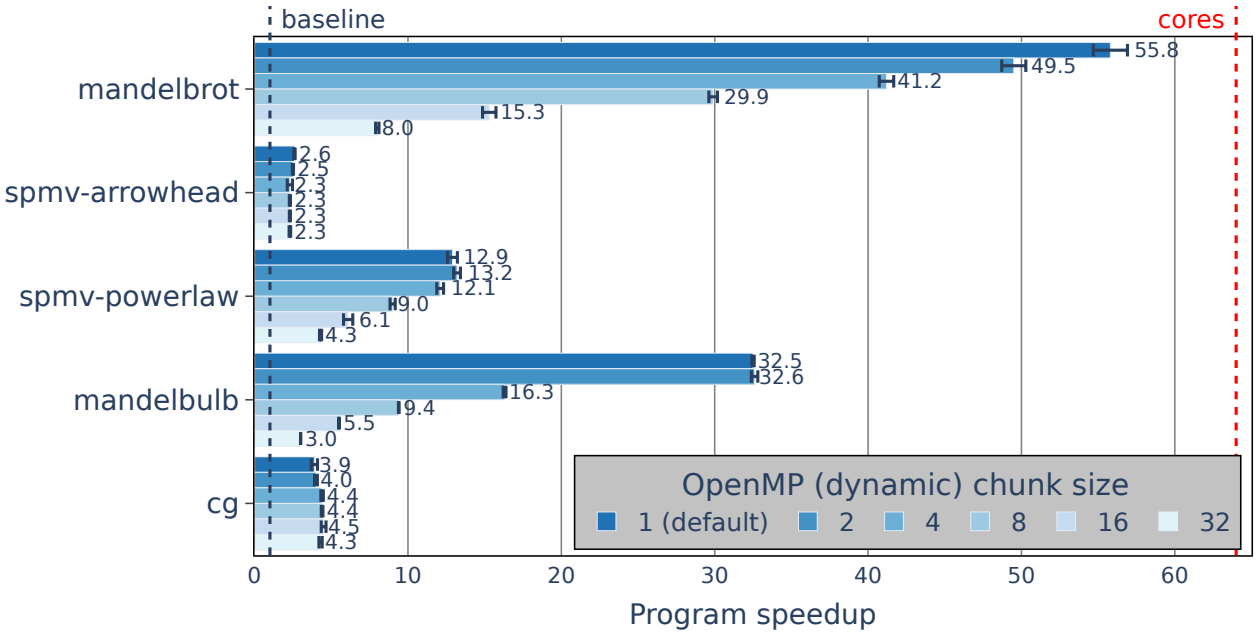


Figure 3.14: 64-core evaluation of OpenMP dynamic scheduling using varying chunk sizes. Only the outermost loop is parallelized.

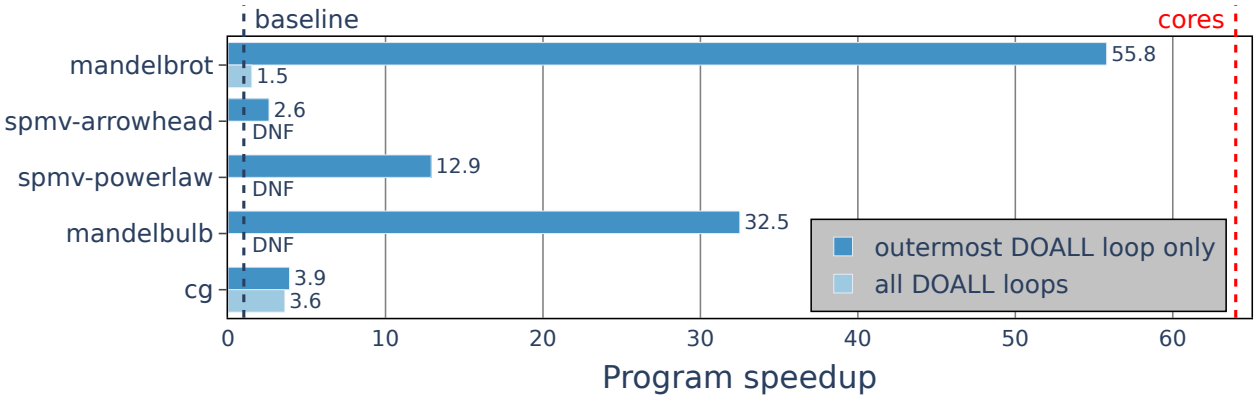


Figure 3.15: 64-core evaluation of OpenMP dynamic scheduling (using default chunk size) parallelizing the outermost loop only versus all DOALL loops. DNF means the program did not finish within the allowed time frame (2 hours) or crashes.

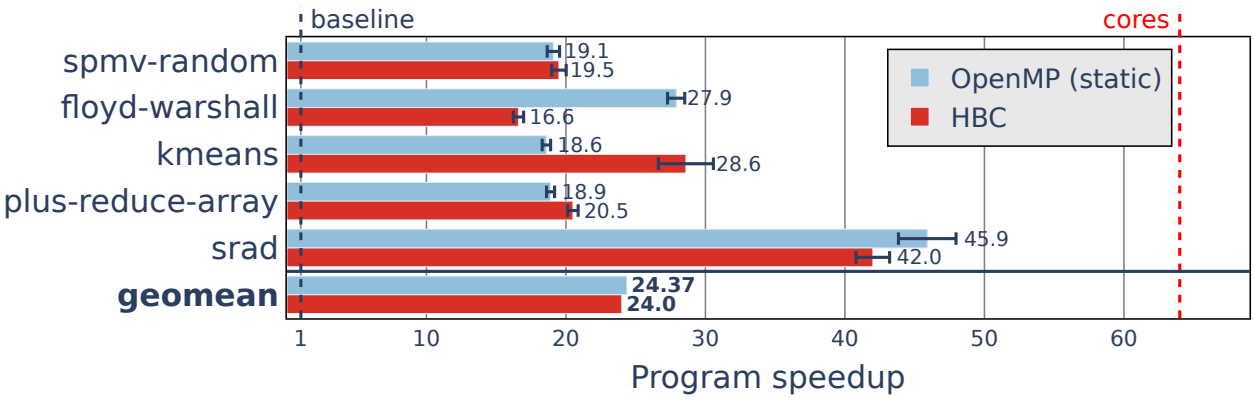


Figure 3.16: 64-core evaluation comparing OpenMP static scheduling and HBC over regular workloads.

CHAPTER 4

AUTOMATING MEMORY SAFETY ON DATA COLLECTIONS

4.1 Introduction

Modern C++ programming heavily relies on data-container abstractions such as `std::vector`, `std::list`, `std::set`, and `std::unordered_map`. These abstractions let developers express high-level data-structure operations without manually managing the underlying memory layout, allocation policy, ordering invariant, resizing strategy, or rehashing schedule. For example, using a `std::vector` abstracts over contiguous heap allocation, capacity growth, and possible reallocation, while an associative container such as `std::set` abstracts over balanced-tree organization and element ordering. The iterator abstraction further decouples algorithms from concrete container implementations by providing a uniform interface for traversal, dereference, comparison, and accessing data elements within a container. For example, the same algorithmic pattern to obtain an iterator using the `begin()` method, advance it, and dereference it, can be applied to both `std::vector` and `std::set` containers with completely different internal representations and implementations.

These abstractions are central to how developers reason about C++ applications, but they also introduce a subtle and well-documented class of memory-safety vulnerabilities. The validity of an iterator is not determined by its type or by the local expression that uses it. Instead, it is determined by the *history* of operations performed on the associated container: a single intervening update can change the container's internal storage or structure and thereby invalidate iterators that were previously obtained from it. A subsequent dereference, comparison, increment, decrement, or

container operation through such an iterator results in undefined behavior. This bug class is known as *iterator invalidation*.

Iterator invalidation is not a theoretical concern. It has produced exploitable vulnerabilities in some of the most widely deployed C++ codebases. CVE-2016-9079 [30], a zero-day in Mozilla Firefox's SVG animation engine, was an iterator-invalidation bug weaponized in the wild. CVE-2018-6088 [29], in Chrome's PDFium component, allowed remote code execution via a crafted PDF and was patched only after the bug had been exploited. CVE-2020-15678 [107] in Mozilla's compositor likewise stemmed from a function that "did not follow iterator invalidation rules," producing a use-after-free during graphical layer traversal. More recently, CVE-2026-2441 [108] was an actively exploited iterator-invalidation flaw in Chrome's CSS font-feature value map. These incidents share a common structural pattern: an iterator is acquired, an intervening control-flow path mutates the container, and the iterator is then reused without revalidation. The structural pattern is simple, but its detection at scale remains a difficult open problem.

Existing memory-safety tools provide only partial coverage of this bug. Low-level pointer analyses and dynamic sanitizers reason about allocations, frees, and raw memory accesses, but an invalidated iterator may still point into allocated storage (a vector iterator after `erase` that shifts elements) or may encode a position whose invalidity is determined by the container's invalidation semantics rather than by an underlying memory fault. Detection requires three forms of reasoning performed simultaneously: (i) tracking iterator state across iterator-arithmetic operations, including iterators derived from other iterators by increment, decrement, or container methods; (ii) understanding the precise container-operation semantics that distinguish, for example, an `erase` on `std::vector` from one on `std::list`; and (iii) preserving iterator validity information across non-trivial control flow, including branches in which only some paths mutate the container. Existing static analyzers each cover a strict subset of these requirements, and none generalizes

naturally to non-standard containers such as those provided by Boost [109], [110], Abseil [111], or LLVM's `ADT` [112].

To overcome these limitations, in this work, we presents LEDGER, a data-collection-oriented static analyzer that closes these coverage gaps by reasoning directly about data collections. LEDGER is built on three design choices. First, it extends MEMOIR [113]—a static-single-assignment (SSA) representation of data collections—with first-class iterator instructions, exposing iterator initialization, bidirectional traversal, iterator-based update, and iterator-based access as operations in the intermediate representation (IR). Second, it performs invalidation detection as a sparse data-flow analysis over collection-version def–use chains, combined with per-container invalidation models that distinguish stable, hash-table, and contiguous-storage semantics. Third, it lowers the cost of adoption through MEMOIR-backed drop-in replacement containers that expose familiar standard- and non-standard-container APIs while producing extended-MEMOIR IR underneath, so that programs benefit from the analysis without being rewritten in terms of low-level IR intrinsics.

The main contributions of this work are:

- LEDGER, a data-collection–oriented static analyzer for iterator-invalidation detection in C++ applications (§4.3).
- An extension of the MEMOIR intermediate representation with first-class iterator operations—initialization at container boundaries, bidirectional traversal, iterator-based update, and iterator-based element access—that preserves derived-iterator provenance and the associated collection version (§4.3.1).
- A sparse data-flow algorithm operating on the extended MEMOIR IR that detects iterator invalidation by walking collection-version def–use chains rather than propagating dense state across the control-flow graph (§4.3.2).

- A set of container invalidation models—stable, hash-table, and contiguous-storage—that decouples the collection-level abstraction from the concrete invalidation semantics of the underlying container, enabling uniform reasoning across both standard and non-standard containers (§4.3.3).
- A practical integration strategy in which MEMOIR-backed drop-in replacement containers expose familiar C++ container APIs while lowering operations to the extended MEMOIR IR, reducing adoption cost from a whole-program rewrite to a header and namespace change (§4.3.4).
- An evaluation against widely used C++ static analyzers showing that LEDGER matches their performance on existing public test suites while substantially improving recall on the synthetic benchmarks that combine derived iterators, non-trivial control flow, and non-standard containers (§4.4).

The remainder of this chapter is organized as follows. Section 4.2 reviews iterator invalidation, the MEMOIR IR, and the limitations of existing approaches. Section 4.3 presents the design of LEDGER. Section 4.4 reports the experimental evaluation. and Section 4.5 concludes.

4.2 Background

This section is organized in three parts. Section 4.2.1 reviews iterator invalidation in C++ containers and motivates the problem with an example used throughout the chapter. Section 4.2.2 describes MEMOIR, the collection-oriented intermediate representation on which LEDGER builds. Section 4.2.3 surveys existing approaches to iterator-invalidation detection and discusses why they fail to fully address the problem.

4.2.1 Iterator Invalidation

An iterator is a thin wrapper that denotes a position in a data container. Its concrete representation differs across container types. A `std::vector` iterator is typically implemented as a pointer into contiguous storage. A `std::list` iterator references a node in a doubly-linked structure. A `std::unordered_map` iterator references a bucket entry in a hash table. The C++ standard programming interface abstracts over these representations so that algorithms can be expressed uniformly in terms of iterator operations.

Iterator invalidation occurs when an iterator remains in scope after it no longer denotes a valid position in the underlying container. There are two principal causes. First, the container itself may reach the end of its lifetime, so that any outstanding iterator dangles. Second, and far more commonly, a mutation operation on the container triggers a structural change—reallocation, node deletion, element shifting, or hash-table rehashing—that invalidates previously obtained iterators.

Listing 4.1: Motivating example of iterator invalidation in `std::vector`.

```
1 std::vector<int> vec = {1, 2, 3, 4, 5};
2 auto it = vec.end();
3 --it;
4 if (COND) { /* ... */ }
5 else { vec.erase(vec.begin()); }
6 int value = *it;          // potential invalidated-iterator use
```

Listing 4.1 illustrates the structural pattern at the heart of this bug class. The iterator `it` is initialized at `vec.end()` and then advanced backwards by one. On the else-branch the container is mutated by an `erase` at position 0, which on `std::vector` shifts subsequent elements forward in storage. By the standard's invalidation rules for `vector`, every iterator at or after the erased

position is invalidated [114]. The dereference `*it` on the final line is therefore unsafe along the else-path, even though `it` remains a syntactically well-typed expression and the underlying storage remains allocated.

The exact invalidation rule depends jointly on the container type and the operation. The C++ standard specifies that erasing from a `std::vector` invalidates iterators and references at or after the erased element [114], whereas erasing from a `std::list` invalidates only iterators and references to erased elements [115]. Ordered associative containers such as `std::map` similarly preserve iterators to non-erased elements [116]. Hash-table containers have a more conditional rule: insertion may invalidate all iterators if it triggers rehashing, while erasure invalidates only the iterator referring to the erased element [117]. Because these rules are container-specific, precise invalidation analysis cannot operate purely at the level of memory accesses or pointer dereferences; it must distinguish the abstract operation performed on the container from its underlying memory effects.

4.2.2 MEMOIR

MEMOIR [113] is a compiler intermediate representation that lifts data collections into a static single-assignment (SSA) form. Where conventional SSA assigns each scalar variable exactly once, MEMOIR assigns each *collection* exactly once: every mutating operation on a collection produces a fresh SSA value representing the updated collection state, while leaving the previous SSA value semantically intact. MEMOIR introduces collection-level abstractions for sequences, maps, and sets, so that operations such as write, insert, and remove appear in the IR as collection operations rather than as opaque memory accesses. The syntax for MEMOIR types, collections, objects, and instructions is reproduced in Figure 4.1.

For memory-safety analysis, the MEMOIR representation provides two properties that LEDGER

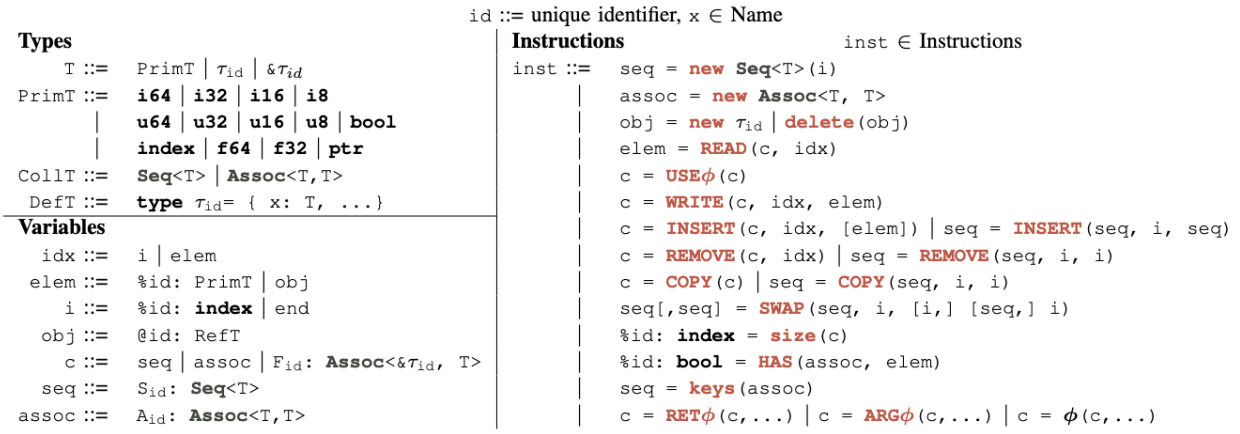


Figure 4.1: The syntax for MEMOIR types, collections, objects, and instructions. The collection-level types (`Seq`, `Assoc`) abstract over concrete container implementations while preserving structural identity through SSA versions.

exploits directly. First, it makes collection versions explicit: a collection before an update and the same logical collection after an update are distinct SSA values connected by a def–use edge. Second, it enables sparse analysis: instead of propagating dense state through every instruction in the control-flow graph, an analysis can walk the collection-level def–use chain between the version where an iterator was defined and the version where it is used. As a result, the analysis cost scales with the number of collection updates rather than with program size.

4.2.3 Existing Approaches

Existing approaches to memory safety on C++ data collections fall into three broad categories: language migration, dynamic analysis, and static analysis. None of them fully resolves the iterator-invalidation problem on C++ applications.

Language migration. A first family of approaches provides stronger safety guarantees by rewriting code in a language or programming model that prevents dangling references altogether. Rust

enforces memory safety without a garbage collector through ownership and borrowing, statically disallowing many of the patterns that lead to iterator invalidation [118]. The C++ Core Guidelines lifetime profile and the related Stroustrup–Sutter proposals [119] similarly aim to detect common dangling-reference errors within C++ itself. The cost of these approaches is substantial: migrating a mature C++ codebase to Rust requires a whole-program rewrite, and the lifetime profile imposes a discipline on annotation and ownership that existing code rarely satisfies. For performance-critical systems that depend on long-lived C++ libraries, migration is rarely the path of least resistance.

Dynamic analysis. Dynamic memory-safety tools such as Valgrind [120] and AddressSanitizer [121] are widely deployed and effective at catching memory errors that manifest during execution. Their effectiveness, however, is bounded by runtime path coverage and by the availability of bug-triggering inputs: a defect on an unexercised control-flow path produces no diagnostic. Iterator invalidation compounds this limitation. An invalidated `std::vector` iterator may still reference allocated, accessible memory after an `erase` that shifts elements, in which case the access produces no low-level memory fault and is silently incorrect. Dynamic memory checkers therefore systematically miss the class of invalidation bugs whose symptom is a semantic violation rather than an out-of-bounds or use-after-free access.

Static analysis. Static analyzers are complementary because they reason about unexecuted paths and do not require bug-triggering inputs. The Clang Static Analyzer [32] ships alpha-quality C++ iterator checkers [122] that flag uses of invalidated iterators based on symbolic execution and a fixed set of standard-library models. Cppcheck [31] targets undefined behavior and dangerous coding constructs in C and C++ while keeping its false-positive rate low; its invalidation detection is pattern-based and not path-sensitive. Clang-Tidy [123] provides linter-style checks for C and

Static Analyzer	Invalidation detection	Path-sensitive control flow	Derived iterators	Detection coverage	Diagnostic traces	Non-STL containers
Clang-Tidy [123]	No	–	–	–	–	–
Clang Static Analyzer [32]	Yes	Yes	No	Low	No	No
Cppcheck [31]	Yes	No	Yes	Low	Yes	No
LEDGER (this work)	**Yes**	**Yes**	**Yes**	**High**	**Yes**	**Yes**

Table 4.1: High-level capability comparison between LEDGER and representative C++ static analyzers along six axes relevant to iterator-invalidation detection. LEDGER combines path-sensitive reasoning, derived-iterator tracking, diagnostic traces, and explicit support for non-standard containers.

C++ and only includes a check for iterator invalidation in range-based `for` loops [124], but it does not model invalidation across general control flow.

Beyond these widely deployed tools, the research and security communities have developed more sophisticated infrastructures. Itergator [125], a CodeQL [126] library developed at Trail of Bits, uses declarative dataflow queries to identify acquire–invalidate–use patterns for iterators and has been used to find real-world bugs at scale. Infer [127], [128] applies bi-abductive separation logic to scale interprocedural memory-safety analysis to industrial codebases, and similar techniques have been used for other classes of memory-safety bug [129]. SVF [130] provides LLVM-based sparse value-flow analysis that has been instantiated for memory-leak and use-after-free detection. PVS-Studio's V789 check detects invalidation specifically within range-based `for` loops over containers [131].

Despite this body of work, no existing analyzer simultaneously provides the three forms of reasoning that precise iterator-invalidation detection requires. Surprisingly, none of Clang Static Analyzer, Cppcheck, or Clang-Tidy report an invalidation on the motivating example of Listing 4.1. First, Clang-Tidy implements pattern matching over the abstract syntax tree and does not model iterator invalidation at all in the general case. Second, Clang Static Analyzer's symbolic-execution

engine tracks iterator state but loses precision on iterators derived from other iterators by arithmetic or container methods, so the `--it` on line 3 of Listing 4.1 severs the analyzer's connection to the originating container. Third, Cppcheck's pattern-based approach does not perform path-sensitive reasoning about which control-flow paths mutate the container. Itergator's CodeQL queries [125] mitigate some of these issues by expressing invalidation as a dataflow query, but they remain tied to the syntactic and library-level patterns expressible in CodeQL and do not generalize to non-standard containers without substantial query authoring. Table 4.1 summarizes the resulting capability profile of widely deployed analyzers relative to LEDGER.

4.3 Ledger

This section presents LEDGER, a static analyzer that detects iterator invalidation by reasoning directly about data collections. LEDGER comprises three design pillars. First, it is built on an extended MEMOIR representation that exposes both data-collection and iterator operations in SSA form (§4.3.1). Second, it performs invalidation detection by walking collection-version def–use paths over the extended representation and applying container-specific invalidation models (§4.2.1, §4.3.3). Third, it improves practicality by providing MEMOIR-backed drop-in replacement containers that allow C++ applications to use the analysis without being rewritten in terms of low-level MEMOIR APIs (§4.3.4).

4.3.1 Representation

LEDGER extends MEMOIR with a family of first-class iterator instructions. The original MEMOIR IR exposes collection allocation, collection updates, and collection reads indexed by integer offsets or keys. To support invalidation analysis, LEDGER adds four classes of instructions: (i) *iterator initialization* at container boundaries, denoted `memoir_iter_begin` and `memoir_iter_end`;

(ii) *bidirectional traversal*, denoted `memoir_iter_stride`, which advances or retreats an iterator by a signed offset and is the IR-level counterpart of C++ iterator arithmetic such as `++it` and `--it`; (iii) *iterator-based updates*, in which collection-modifying operations such as insertion and deletion take an iterator argument denoting the position of the operation; and (iv) *iterator-based reads*, which dereference an iterator to yield the element it denotes.

Listing 4.2: The extended MEMOIR instructions for the motivating example of Listing 4.1, shown in LLVM IR syntax.

```
%col0    = call ptr @memoir_allocate(...)
%it_beg  = call ptr @memoir_iter_begin(ptr %col0)
%it_end  = call ptr @memoir_iter_end(ptr %col0)
%it_last = call ptr @memoir_iter_stride(ptr %col0, ptr %it_end, i64 -1)
%col1    = call ptr @memoir_remove(ptr %col0, ptr %it_beg)
%val     = call i32 @memoir_read(ptr %col1, ptr %it_last)
```

Listing 4.2 shows the extended-MEMOIR representation of the motivating example. The original collection SSA value `%col0` is produced at allocation and gives rise to two iterators at the container boundaries (`%it_beg` and `%it_end`) as well as a derived iterator `%it_last` obtained by a backward stride of 1. The `erase` call produces a new collection version `%col1`, and the final `memoir_read` accesses `%col1` through `%it_last`.

This representation has two properties that are essential for LEDGER's analysis.

Provenance of derived iterators. The representation captures derived iterator states without losing their connection to the originating collection. The C++ expression `--it` gets lowered into a `memoir_iter_stride` instruction whose first operand is the defining collection version of its source iterator. As a result, derived iterators are not opaque: they carry the provenance of

Algorithm 3 Iterator-invalidation detection in LEDGER.

Input: Extended-MEMOIR program P.
Output: A, the set of iterator-access instructions whose iterators may be invalidated.

```
1:  A ← ∅
2:  for each instruction i ∈ P do
3:      if i is not an iterator-access instruction then
4:          continue
5:      it ← i.GETITERATOR()
6:      C_0 ← it.GETDEFININGCOLLECTION()
7:      C_n ← i.GETACCESSINGCOLLECTION()
8:      path ← GETDEFUSEPATH(C_0, C_n)
9:      for each u ∈ path.GETUPDATINGINSTRUCTIONS() do
10:         if u is an insertion or removal then
11:             if u is not operated by it then
12:                 A ← A ∪ {i}
13:                 break
14: return A
```

the iterator from which they were computed, and they remain associated with the same defining collection version. This matters because most invalidation bugs do not arise from the immediate result of `begin()` or `end()`; they arise from iterators that have been advanced, decremented, copied, assigned, or returned by container methods.

Explicit collection versions. Every mutating operation on a collection produces a new SSA value. The pair (C_0, C_n) formed by an iterator's defining collection version and the collection version accessed at a later instruction is therefore well-defined and statically computable, and the sequence of intervening updates is precisely the def–use path between these two values. This is the structural invariant that LEDGER's invalidation detection exploits.

4.3.2 Invalidation Detection

Algorithm 3 presents LEDGER's invalidation-detection procedure over the extended MEMOIR representation. The algorithm takes a program P as input and returns the set A of iterator-access instructions whose iterators may be invalidated. The procedure iterates over every instruction in P. Non-iterator-access instructions are skipped, because they cannot directly dereference or otherwise use an iterator. For each iterator-access instruction i, LEDGER extracts the iterator it, the collection version C_0 on which the iterator was originally defined, and the collection version C_n accessed at i. It then computes GETDEFUSEPATH(C_0, C_n), the sequence of collection-version updates between the iterator's defining collection and the collection version used at the access site.

The central observation underlying the algorithm is that an iterator becomes *potentially invalid* when the collection has been structurally modified between the point where the iterator was defined and the point where it is used. LEDGER therefore inspects each updating instruction u on the def–use path. If u is an insertion or removal, it may change the structure of the underlying collection; if that update is not itself performed through it, LEDGER marks the access i as a potentially invalidated-iterator access and proceeds to the next instruction. The "not operated by it" condition is important: an iterator-based update performed through it either yields a fresh iterator as its result (as in the `erase` return-value idiom) or leaves it in a well-defined state by construction, and so does not invalidate the iterator under the analysis.

Applied to the motivating example, the algorithm proceeds as follows. The dereference `*it` corresponds to the `memoir_read(%col1,%it_last)` instruction. The iterator `%it_last` is defined on collection version `%col0`, and the access reads from `%col1`, so the def–use path is the single edge `%col0` $\rightarrow$ `%col1` produced by the `memoir_remove` instruction. This update is an insertion-or-removal operation performed through a different iterator (`%it_beg`), so the access is reported as potentially invalidated.

The algorithm is sparse in the sense that its cost is proportional to the number of collection updates along the def–use path between C_0 and C_n, rather than to the number of instructions in the control-flow graph. For programs that mutate collections rarely relative to their overall instruction count—a common pattern in practice—this yields a substantial constant-factor improvement over dense data-flow analyses formulated on the control-flow graph.

4.3.3 Container Invalidation Models

The detection procedure of Algorithm 3 reports *potentially* invalidated accesses, on the assumption that any intervening update may invalidate an iterator. To distinguish potential invalidation from actual invalidation, LEDGER must also reason about the concrete container's invalidation semantics. The same high-level MEMOIR abstraction can correspond to containers with different iterator-stability guarantees: both a `std::vector` and a `std::list` present as sequence-like collections, yet removing an element from a vector invalidates iterators to subsequent elements, while removing an element from a list preserves iterators to non-erased nodes.

LEDGER decouples the collection-level abstraction from concrete implementation details by assigning each container to an *invalidation model*. Three models suffice to cover the containers encountered in standard and widely used non-standard libraries: *stable*, *hash-table*, and *contiguous-storage*. Table 4.2 summarizes the iterator-stability behavior of each model, separately characterizing the iterator that performs the operation ("operating iterator") and any other live iterator on the same container.

The *stable* model captures containers whose structure preserves iterator validity for existing elements regardless of insertion or deletion. The *hash-table* model captures containers whose iterators may be invalidated by rehashing on insertion, and the operating-iterator-vs.-others distinction matters because the standard guarantees that the result of an `erase` returning an iterator remains

Model	Operation	Operating iterator	Other iterators
Stable	Insertion	Valid	Valid
	Deletion	Valid	Valid
Hash-table	Insertion	Valid	Invalidated
	Deletion	Valid	Valid
Contiguous-storage	Insertion	Valid	Invalidated
	Deletion	Valid	Invalidated

Table 4.2: Container invalidation models used by LEDGER. Each model specifies whether the iterator that performs the operation and any other live iterator on the same container remain valid after insertion or deletion.

valid even when other iterators are invalidated. The *contiguous-storage* model captures containers whose updates can shift elements or reallocate storage and therefore invalidate all but the operating iterator. The models are intentionally conservative when precise runtime properties are not statically available: if LEDGER cannot prove that a hash-table insertion will not rehash, it treats the insertion as invalidating other iterators.

The invalidation models extend LEDGER's reach beyond the standard library. For example, Boost's `stable_vector` [110] provides vector-like indexed access while preserving iterator stability and therefore fits the stable model; Boost's flat containers [109] use vector-like contiguous storage and invalidate iterators in the same way as other contiguous-storage containers; Abseil's flat hash tables [111] store values inline and invalidate iterators on rehash; LLVM's `DenseMap` [112] invalidates iterators on insertion. By mapping each of these containers to one of the three invalidation models, LEDGER avoids hard-coding an analysis tied to a single library, and a new container can be added simply by classifying it into the appropriate model.

4.3.4 Practicability

A collection-oriented IR is useful only if programs can be lowered to it without imposing unrealistic rewriting requirements on developers. The original MEMOIR interface exposes low-level collection intrinsics, and rewriting a large C++ application directly in terms of those intrinsics would be impractical: every `push_back`, every iterator-based loop, every range-based `for` would have to be expressed as a sequence of explicit `memoir_*` calls. LEDGER therefore introduces MEMOIR-backed *drop-in replacement containers* that preserve the public C++ container API while lowering operations to extended-MEMOIR intrinsics underneath.

A drop-in replacement container implements the same methods, the same type-level interface, and the same iterator concepts as its standard-library counterpart, but its method bodies emit MEMOIR intrinsics in addition to performing the underlying operation. A developer adopts LEDGER by replacing `#include <vector>` with `#include <MEMOIR/vector>` and substituting `MEMOIR::vector` for `std::vector` (or relying on a namespace alias when migrating an existing codebase). The application continues to use the familiar methods `begin()`, `end()`, `erase()`, `push_back()`, and the iterator dereference operator, and its semantics are unchanged; the only difference is that these operations become visible to LEDGER in the extended-MEMOIR IR. This strategy reduces the cost of adoption from a whole-program rewrite to a header and namespace change for supported containers.

LEDGER currently targets a representative set of standard and non-standard containers. The standard-library replacements include sequence containers (`vector`, `list`, `string`) and hash-table containers (`unordered_map`, `unordered_set`). The non-standard replacements include Boost flat containers (`flat_map`, `flat_set`). The same strategy generalizes to additional containers by implementing the relevant APIs and assigning the container to one of the invalidation models of §4.3.3.

4.4 Evaluation

This section evaluates LEDGER's ability to detect iterator-invalidation vulnerabilities. The evaluation is structured around a single research question:

> *RQ: How does* LEDGER *compare with widely deployed C++ static analyzers in detecting iterator-invalidation vulnerabilities, both on existing public test suites and on synthetic benchmarks that exercise harder cases?*

4.4.1 Experimental Settings

The evaluation compares LEDGER against two widely deployed C++ static analyzers: Clang Static Analyzer [32] with its alpha-quality C++ iterator checkers enabled, and Cppcheck [31] in its default configuration with all relevant inspection categories activated. Clang-Tidy [123] is omitted from the quantitative comparison because, as discussed in Section 4.2.3, it does not implement a general iterator-invalidation check.

Two test suites are used. The first comprises pre-existing iterator-invalidation tests aggregated from two sources: 16 tests under CWE-672 [132] ("Operation on a Resource after Expiration or Release") from the Juliet C/C++ test suite [133], a systematic collection of small C and C++ programs organized by Common Weakness Enumeration category, and 21 tests authored for iterator-invalidation checking in the Clang Static Analyzer [122]. Each test contains at most one iterator-invalidation vulnerability, with ground truth provided by the test suite. The second test suite is a set of synthetic benchmarks generated to exercise cases that are under-represented in the existing suites, described in Section 4.4.3 below.

The primary metric is recall:

$$\text{Recall} \;=\; \frac{TP}{TP + FN},$$

where TP is the number of true iterator-invalidation bugs reported by the analyzer and FN is the number of true bugs missed. Recall is the appropriate primary metric because the absolute concern in this domain is the rate at which analyzers fail to surface real vulnerabilities.

4.4.2 Existing Test Suites

On the combined 37 tests drawn from the Juliet and LLVM suites, LEDGER detects every iterator-invalidation vulnerability, achieving 100% recall. The two baseline analyzers also achieve 100% recall on these tests, confirming the public observation that existing tools are well-tuned to the simple, hand-written invalidation patterns these suites encode. The principal takeaway from this experiment is not that LEDGER surpasses the baselines on these tests, but rather that it is at least on par with them on the cases where they were intended to be strong, leaving no regression on simple invalidation patterns. The remaining question is whether the baseline analyzers' parity on these simple cases reflects their general effectiveness or whether the public suites fail to exercise the harder cases that motivated LEDGER.

4.4.3 Synthetic Testing

To stress-test iterator-invalidation detection, this work uses a program generator that produces small C++ programs exercising multiple live iterators on a single container. The generator randomly composes iterator increment and decrement, insertion, erasure, and dereference operations, and it introduces nested `if`, `if-else`, and `for` constructs to stress path-sensitive reasoning. Two test populations are produced: a standard-library population in which the underlying container is a member of the C++ STL, and a non-standard population in which the underlying container is drawn from Boost or Abseil. Ground truth is established by manual inspection.

Table 4.3 reports recall across the three test populations. On synthetic standard-library tests,

Test suite	Clang Static Analyzer	Cppcheck	LEDGER
Existing Juliet/LLVM tests	100.0%	100.0%	100.0%
Synthetic, STL containers	15.2%	22.8%	87.5%
Synthetic, non-STL containers	0.0%	0.0%	92.5%

Table 4.3: Recall on iterator-invalidation detection across the three test populations. Both baselines saturate on the pre-existing hand-written tests but degrade sharply on synthetic tests with derived iterators and non-trivial control flow, and both produce zero detections on non-standard containers in the evaluated configuration.

LEDGER achieves 87.5% recall, compared with 15.2% for Clang Static Analyzer and 22.8% for Cppcheck. The gap reflects the limitations identified in Section 4.2.3: the baseline analyzers lose track of derived iterators after stride operations and do not reason precisely about which branches mutate the container. On synthetic non-standard tests, LEDGER achieves 92.5% recall, while both baselines report no detections. The zero-recall result for the baselines is consistent with their documentation: their invalidation checks are written against hard-coded STL container models and do not generalize to Boost or Abseil containers in the evaluated configuration.

Two conclusions follow from these results. First, existing public test suites do not fully capture the complexity of iterator invalidation in realistic C++ code: tools that perform well on simple invalidation tests can still miss the large majority of cases that combine derived-iterator states with non-trivial control flow. Second, a collection-oriented representation materially improves recall because it preserves the information the analysis needs at the right level of abstraction: the iterator's defining collection version, the collection-version updates that produce later versions, and the invalidation model of the concrete container. The combination of an SSA-versioned collection representation and per-container invalidation models is what enables the analyzer to generalize beyond the STL without being rewritten for each new container.

4.5 Conclusion

This work introduces LEDGER, a data collection-oriented static analyzer for detecting iterator invalidation vulnerabilities in C++ applications. LEDGER extends MEMOIR with first-class iterator instructions and detects invalidation following a sparse data-flow analysis. The introduced invalidation models allow the same analysis to reason about stable type, hash-table type, and contiguous-storage type containers, including non-standard containers. The drop-in replacement containers make LEDGER practical by exposing MEMOIR semantics through C++ APIs. LEDGER detects invalidation cases that are missed by existing analyzers, especially when programs contain derived iterators, complex control flow, and non-standard containers.

CHAPTER 5
CONCLUSION AND FUTURE WORK

This chapter concludes my thesis by summarizing my contributions and discussing directions for future work.

5.1 Conclusion

Modern hardware is fundamentally parallel, and exploiting it forces developers to take on a series of burdens writing parallel code such as selecting a parallel execution plan and tuning its granularity to the input the program will process. Today's production compilers leave each of these burdens with the developer. This dissertation introduces new compiler abstractions and compilation techniques that allow optimizing compilers to release these burdens on the developer's behalf.

The first burden concerns the static aspects of the plan: the strategy a developer encodes is fixed, but the best strategy depends on the target hardware. Existing compiler intermediate representations preserve the developer's plan but treat it as immutable. We propose the *Parallel-Semantics Program Dependence Graph* (PS-PDG) [5], which simultaneously represents the developer's parallel semantics and the compiler's dependence analysis, enabling the compiler to reason about the full space of legal plans and select one tailored to the hardware. On eight NAS benchmarks, our PS-PDG-based optimizing compiler GINO improves average speedup from 6.8× to 7.8× on a 56-core machine and outperforms the developer's plan by up to 46.6%.

The second burden concerns the dynamic aspects of the plan, most prominently the granularity of parallel tasks, which is input-dependent but today's PPMs ask developers to manually determine them in source code. We propose the *Heartbeat Compiler* (HBC) [33], the first compiler that trans-

lates C/C++ programs with high-level fork-join constructs into binaries that automatically tune task granularity at runtime through heartbeat scheduling. On irregular workloads from TACO, GraphIt, and Rodinia, HBC improves average speedup from 14.2× under OpenMP dynamic scheduling to 21.7× on a 64-core machine, matching binaries that previously required months of manual heartbeat-scheduling work.

Beyond performance, developers also bear the cost of reasoning about correctness. As a by-product of this dissertation, we present *Ledger*, a data collection-oriented static analyzer that detects iterator invalidation vulnerabilities—a bug class that has produced exploitable use-after-free errors in Chrome [29], [108] and Firefox [30], [107]. On a synthetic benchmark suite stressing derived iterators and non-trivial control flow, Ledger achieves 87.5% recall, compared to 22.8% for prior state-of-the-art analyzers.

Taken together, these contributions move responsibilities that today rest on parallel-programming developers into the compiler: selecting a hardware-appropriate execution plan, adapting its granularity to the input at runtime, and catching a class of memory-safety bug that has repeatedly produced security vulnerabilities in production C++ applications.

5.2 Other Contributions from Collaborative Work

In support of this thesis, I have collaborated with other researchers and made the following contributions:

- PROMPT: A Fast and Extensible Memory Profiling Framework, a memory profiling framework that factors profiling into a generic frontend–backend pipeline with reusable instrumentation, event handling, and parallel data structures. This work is published at OOPSLA 2024 [134] in collaboration with Ziyang Xu, Yebin Chon, Zujun Tan, Sotiris Apostolakis, Simone Campanoni and David I. August.

- Revisiting Computation for Research: Practices and Trends, A survey paper including responses from 106 researchers across disciplines and documenting the gap between available computational resources and researchers' ability to fully utilize them. This work is published at SC 2024 [135] in collaboration with Jeremiah Giordani, Ziyang Xu, Ella Colby, August Ning, Bhargav Reddy Godala, Ishita Chaturvedi, Shaowei Zhu, Yebin Chon, Greg Chan, Zujun Tan, Galen Collier, Jonathan D. Halverson, Enrico Armenio Deiana, Jasper Liang, Federico Sossai, Atmn Patel, Bangyen Pham, Nathan Greiner, Simone Campanoni, and David I. August.
- NOELLE Offers Empowering LLVM Extensions, an open-source compilation framework that provides higher-level compiler abstractions, including PDG, SCCDAGs, and loop-centric analyses that enable 11 advanced code transformations. This work is published at CGO 2022 [37] in collaboration with Angelo Matni, Enrico A. Deiana, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, Brian Homerding, Tommy McMichen, David I. August, and Simone Campanoni.

5.3 Future Directions

The contributions of this thesis open several directions for future work. The directions below extend the thesis's central theme—moving parallel-programming burdens from the developer into the compiler—to settings that this work did not address.

5.3.1 Extending PS-PDG Beyond Shared-Memory CPUs

Currently, PS-PDG and GINO target shared-memory CPUs only. Modern hardware is increasingly heterogeneous: GPUs are now standard accelerators across scientific computing, machine learning, and graphics workloads, and large-scale systems combine CPUs, GPUs, and distributed nodes

connected over interconnects with very different bandwidth and latency characteristics. Extending PS-PDG to these platforms requires the abstraction to capture parallel semantics it does not capture today, including host-device memory transfers, SIMT execution divergence, and inter-node communication. A natural next step is to enrich PS-PDG's trait system with annotations that encode these costs, and to extend GINO with cost models capable of comparing plans that span heterogeneous backends—e.g., choosing whether a loop should be parallelized on the host, offloaded to a GPU, or split across both. Prior work on GPU-aware OpenMP optimization [136], [137] provides a foundation for the kinds of analyses such an extension would build on.

A related direction is extending PS-PDG to distributed-memory programming models such as MPI, where communication is explicit and the compiler must reason about data movement as a first-class concern. Hybrid MPI+OpenMP execution is the dominant model in HPC, and reactive load-balancing systems built on top of it [138] have shown that runtime task migration across MPI ranks is a productive direction. A compiler that could reason about both layers of parallelism within a single representation would relieve a burden that today rests entirely on application developers.

Beyond hardware portability, GINO's plan-selection heuristics are hand-engineered. Machine learning has been shown to outperform hand-engineered heuristics across a wide range of compiler decisions [139], [140]. Training a model to score PS-PDG-derived candidate plans, conditioned on hardware features and profile data, could improve plan selection beyond hand-engineered manual heuristics.

5.3.2 Extending HBC Beyond Statically-Known Loop Structures

HBC currently requires the structure of a parallel loop nest to be statically known at compile time, so that the compiler can generate the split-and-rollforward code for each promotion site ahead of time. Many real workloads do not satisfy this assumption: recursive algorithms (e.g., divide-and-

conquer parallelism), task-parallel programs with dynamically-spawned work, and loops whose nesting depth is dynamically determined at runtime all fall outside HBC's current scope. Extending HBC to these dynamic structures would require either runtime code generation or a more flexible promotion mechanism that can synthesize the necessary split points on the fly. Prior work on lazy task creation in dynamically-structured programs [21], [22] offers a starting point, but reconciling those approaches with heartbeat's provable overhead bound is an open question.

A second limitation is that HBC's granularity decisions are made within a single parallel region. In practice, programs often consist of multiple parallel regions in sequence, where the right granularity for one region depends on the work in the next. Adaptive granularity control *across* loop boundaries—for instance, fusing the tail of one parallel loop with the head of the next when their iteration spaces and dependences permit, or carrying granularity state between regions to amortize promotion overhead—would extend HBC's reach to programs whose parallelism is exposed gradually across multiple phases.

5.4 Acknowledgement

I thank the National Science Foundation, ARCANA Lab, and Prescience Lab at Northwestern University for supporting my thesis work. The PS-PDG [5] work is supported by the U.S. Department of Energy under contract number DE-SC0022268. It is also based upon work supported by the National Science Foundation under Grants CCF-1901381, CCF-2107241, CCF-2115104, NSF-2119069, CCF-2119352, NSF-2107042, NSF-2028851, and NSF-1908488, and via 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 the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357. The HBC [33] work is supported by the National Science Foundation under Grants NSF2119069, NSF-2148177, NSF-2107042, and

NSF-2107257.

REFERENCES

[1] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 365–376, Jun. 2011.

[3] OpenMP Architecture Review Board, *OpenMP application program interface*.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, no. 1, pp. 55–69, 1996.

[5] Y. Su et al., "The parallel-semantics program dependence graph for parallel optimization," in *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2026, pp. 348–361.

[6] LLVM Project, *Clang: A C language family frontend for LLVM*, Accessed: [Insert Date Here].

[7] R. M. Stallman and the GCC Developer Community, *Using the gnu compiler collection*, Boston, MA: Free Software Foundation, 2003.

[8] LLVM Project, *A Compiler's View of OpenMP*, `https://www.openmp.org/wp-content/uploads/A-compilers-view-of-OpenMP.pdf`.

[9] OpenMP Architecture Review Board, *OpenMP application program interface version 5.0*, Nov. 2018.

[10] D. H. Bailey et al., "The NAS parallel benchmarks—summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91, Albuquerque, New Mexico, USA: Association for Computing Machinery, 1991, pp. 158–165, ISBN: 0897914597.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[12] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into LLVM's intermediate representation," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17, Austin, Texas, USA: Association for Computing Machinery, 2017, pp. 249–265, ISBN: 9781450344937.

[13] V. Sarkar and B. Simons, "Parallel program graphs and their classification," in *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Berlin, Heidelberg: Springer-Verlag, 1993, pp. 633–655, ISBN: 3540576592.

[14] V. Sarkar, "Analysis and optimization of explicitly parallel programs using the parallel program graph representation," in *LCPC*, 1997.

[15] H. Srinivasan and M. Wolfe, "Analyzing programs with explicit parallelism," in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, Berlin, Heidelberg: Springer-Verlag, 1991, pp. 405–419, ISBN: 354055422X.

[16] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer, "INSPIRE: The insieme parallel intermediate representation," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 7–17.

[17] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar, "A transformation framework for optimizing task-parallel programs," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 1, Apr. 2013.

[18] *OMP dialect - mlir*, 2024.

[19] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[20] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[21] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, no. 5, pp. 212–223, May 1998.

[22] T. B. Schardl and I.-T. A. Lee, "Opencilk: A modular and extensible software infrastructure for fast task-parallel code," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23, Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 189–203, ISBN: 9798400700156.

[23] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin, "Lazy binary-splitting: A run-time adaptive work-stealing scheduler," in *Symposium on Principles & Practice of Parallel Programming*, 2010, pp. 179–190.

[24] A. Tzannes, G. C. Caragea, U. Vishkin, and R. Barua, "Lazy scheduling: A runtime adaptive scheduler for declarative parallelism," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 3, Sep. 2014.

[25] U. A. Acar, A. Charguéraud, A. Guatto, M. Rainey, and F. Sieczkowski, "Heartbeat scheduling: Provable efficiency for nested parallelism," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, Philadelphia, PA, USA, 2018, pp. 769–782, ISBN: 978-1-4503-5698-5.

[26] M. Rainey et al., "Task parallel assembly language for uncompromising parallelism," in *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '21, New York, NY, USA: ACM, Jun. 2021.

[27] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[28] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*, Ieee, 2009, pp. 44–54.

[29] MITRE, *CVE-2018-6088: Iterator-invalidation bug in PDFium in Google Chrome prior to 66.0.3359.117 allowing remote code execution via a crafted PDF*, Common Vulnerabilities and Exposures, Accessed: 2026-05-22, 2018.

[30] MITRE, *CVE-2016-9079: Iterator invalidation in nsSMILTimeContainer::NotifyTimeChange leading to use-after-free in Mozilla Firefox SVG animation*, Common Vulnerabilities and Exposures, Accessed: 2026-05-22, 2016.

[31] Cppcheck Project, *Cppcheck manual*, Project documentation, Accessed 2026-05-21, 2026.

[32] LLVM Project, *Available checkers: Clang static analyzer*, Project documentation, Accessed 2026-05-21, 2026.

[33] Y. Su et al., "Compiling loop-based nested parallelism for irregular workloads," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 232–250, ISBN: 9798400703850.

[34] *LLVM/OpenMP design and overview*, 2023.

[35] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "HELIX: Automatic parallelization of irregular programs for chip multiprocessing," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, San Jose, California: ACM, 2012, pp. 84–93, ISBN: 978-1-4503-1206-6.

[36] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, IEEE, 2007, pp. 49–59.

[37] A. Matni et al., "NOELLE offers empowering LLVM extensions," in *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '22, Virtual Event, Republic of Korea: IEEE Press, 2022, pp. 179–192, ISBN: 9781665405843.

[38] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August, "Perspective: A sensible approach to speculative automatic parallelization," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 351–367, ISBN: 9781450371025.

[39] S. Apostolakis, Z. Xu, Z. Tan, G. Chan, S. Campanoni, and D. I. August, "SCAF: a speculation-aware collaborative dependence analysis framework," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 638–654, ISBN: 9781450376136.

[40] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[41] A. R. Hurson, J. T. Lim, K. M. Kavi, and B. Lee, "Parallelization of DOALL and DOACROSS loops—a survey," in *Advances in computers*, vol. 45, Elsevier, 1997, pp. 53–103.

[42] LLVM Project, *AliasAnalysis Class Reference*, https://mlir.llvm.org/doxygen/classmlir_1_1AliasAnalysis.html.

[43] LLVM Project, *Side effects & speculation*, https://mlir.llvm.org/docs/Rationale/SideEffectsAndSpeculation/.

[44] LLVM Project, *MLIR Language Reference*, https://mlir.llvm.org/docs/LangRef/.

[45] Omni Compiler Project, *NAS-C-OpenMP3.0*, https://benchmark-subsetting.github.io/cNPB/, 2014.

[46] LLVM Project, *Auto-Vectorization in LLVM*, https://llvm.org/docs/Vectorizers.html.

[47] H. Vandierendonck, S. Rul, and K. De Bosschere, "The paralax infrastructure: Automatic parallelization with a helping hand," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, Vienna, Austria: Association for Computing Machinery, 2010, pp. 389–400, ISBN: 9781450301787.

[48] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," *SIGPLAN Not.*, vol. 29, no. 6, pp. 171–185, Jun. 1994.

[49] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "HPVM: heterogeneous parallel virtual machine," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18, Vienna, Austria: Association for Computing Machinery, 2018, pp. 68–80, ISBN: 9781450349826.

[50] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, San Diego, California, USA: Association for Computing Machinery, 2007, pp. 211–222, ISBN: 9781595936332.

[51] M. A. Hassaan, D. D. Nguyen, and K. K. Pingali, "Kinetic dependence graphs," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 457–471, ISBN: 9781450328357.

[52] G. Blelloch and J. Greiner, "Parallelism in sequential functional languages," in *Proceedings of the seventh international conference on Functional programming languages and computer architecture - FPCA '95*, La Jolla, California, United States: ACM Press, 1995, pp. 226–237, ISBN: 978-0-89791-719-3.

[53] M. Fluet, M. Rainey, J. Reppy, and A. Shaw, "Implicitly-threaded parallelism in Manticore," p. 12,

[54] S. Westrick, R. Yadav, M. Fluet, and U. A. Acar, "Disentanglement in nested-parallel programs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, Jan. 2020.

[55] R. H. Halstead, "Implementation of multilisp: Lisp on a multiprocessor," in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, ser. LFP '84, New York, NY, USA: Association for Computing Machinery, Aug. 1984, pp. 9–17, ISBN: 978-0-89791-142-9.

[56] A. Arvind, R. Nikhil, and K. Pingali, "I-Structures: Data structures for parallel computing.," *ACM Trans. Program. Lang. Syst.*, vol. 11, pp. 598–632, Oct. 1989.

[57] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," *ACM SIGPLAN Notices*, vol. 28, no. 7, pp. 102–111, Jul. 1993.

[58] P. Li, S. Marlow, S. Peyton Jones, and A. Tolmach, "Lightweight concurrency primitives for GHC," in *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, ser. Haskell '07, New York, NY, USA: Association for Computing Machinery, Sep. 2007, pp. 107–118, ISBN: 978-1-59593-674-5.

[59] S. Marlow, "Parallel and concurrent programming in haskell," in *Central European Functional Programming School: 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, V. Zsók, Z. Horváth, and R. Plasmeijer, Eds., Berlin, Heidelberg: Springer, 2012, pp. 339–401, ISBN: 978-3-642-32096-5.

[60] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. Chakravarty, "Harnessing the multicores: Nested data parallelism in Haskell," *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 2, Dec. 2008.

[61] A. Guatto, S. Westrick, R. Raghunathan, U. Acar, and M. Fluet, "Hierarchical memory management for mutable state," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 81–93, Feb. 2018.

[62] R. Raghunathan, S. K. Muller, U. A. Acar, and G. Blelloch, "Hierarchical memory management for parallel programs," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2016, New York, NY, USA: Association for Computing Machinery, Sep. 2016, pp. 392–406, ISBN: 978-1-4503-4219-3.

[63] K. C. Sivaramakrishnan, L. Ziarek, and S. Jagannathan, "MultiMLton: A multicore-aware runtime for standard ML," *Journal of Functional Programming*, vol. 24, no. 6, pp. 613–674, Nov. 2014, Publisher: Cambridge University Press.

[64] U. Klusik, R. Loogen, S. Priebe, and F. Rubio, "Implementation skeletons in eden: Low-effort parallel programming," M. Mohnen and P. Koopman, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2001, pp. 71–88, ISBN: 978-3-540-45361-1.

[65] S. Ghosh, M. Cuevas, S. Campanoni, and P. Dinda, "Compiler-based timing for extremely fine-grain preemptive parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20, Atlanta, Georgia: IEEE Press, 2020, ISBN: 9781728199986.

[66] K. C. Hale, C. Hetland, and P. A. Dinda, "Automatic hybridization of runtime systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16, Kyoto, Japan: Association for Computing Machinery, 2016, pp. 137–140, ISBN: 9781450343145.

[67] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks, "HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 217–228, ISBN: 978-1-4799-4394-4.

[68] D.-K. Chen, H.-M. Su, and P.-C. Yew, "The impact of synchronization and granularity on parallel systems," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 239–248, 1990.

[69] S. Debray, M. V. Hermenegildo, and P. López García, "A methodology for granularity-based control of parallelism in logic programs," *Journal of symbolic computation*, vol. 21, no. 4-6, pp. 715–734, 1996.

[70] M. Girkar and C. D. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," *IEEE transactions on parallel and distributed systems*, vol. 3, no. 2, pp. 166–178, 1992.

[71] E. Frantar and D. Alistarh, "Sparsegpt: Massive language models can be accurately pruned in one-shot," 2023.

[72] D. Mosberger, P. Druschel, and L. L. Peterson, "Implementing atomic sequences on uniprocessors using rollforward," *Software: Practice and Experience*, vol. 26, no. 1, pp. 1–23, 1996.

[73] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks, "HELIX-UP: Relaxing program semantics to unleash parallelization," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15, San Francisco, California: IEEE Computer Society, 2015, pp. 235–245, ISBN: 978-1-4799-8161-8.

[74] N. Murphy, T. Jones, R. Mullins, and S. Campanoni, "Performance implications of transient loop-carried data dependences in automatically parallelized loops," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016, Barcelona, Spain: ACM, 2016, pp. 23–33, ISBN: 978-1-4503-4241-4.

[75] E. A. Deiana, V. St-Amour, P. A. Dinda, N. Hardavellas, and S. Campanoni, "Unconventional parallelization of nondeterministic applications," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, Williamsburg, VA, USA: ACM, 2018, pp. 432–447, ISBN: 978-1-4503-4911-6.

[76] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August, "Perspective: A sensible approach to speculative automatic parallelization," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 351–367, ISBN: 9781450371025.

[77] J. Ma et al., "Paths to openmp in the kernel," in *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri,*

USA, November 14 - 19, 2021, B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds., ACM, 2021, 65:1–65:17.

[78] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "HELIX: Automatic parallelization of irregular programs for chip multiprocessing," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, San Jose, California: ACM, 2012, pp. 84–93, ISBN: 978-1-4503-1206-6.

[79] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers principles, techniques & tools*, 2007.

[80] V. Leis, K. Kundhikanjana, A. Kemper, and T. Neumann, "Efficient processing of window functions in analytical sql queries," 10, vol. 8, VLDB Endowment, Jun. 2015, pp. 1058–1069.

[81] A. Matni et al., "NOELLE Offers Empowering LLvm Extensions," in *International Symposium on Code Generation and Optimization, 2022. CGO 2022.*, 2022.

[82] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[83] A. van Heukelum, G. T. Barkema, and R. H. Bisseling, "DNA electrophoresis studied with the cage model," *Journal of Compututational Physics*, vol. 180, pp. 313–326, 1 Jul. 2002.

[84] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[85] D. White, *3d mandelbrot generator*, `https://www.fountainware.com/Funware/Mandelbrot3D/Mandelbrot3d.htm`, 2008.

[86] M. Rainey, *Tpal github*, `https://github.com/mikerainey/tpal/tree/master`, 2021.

[87] F. Kjolstad, *Taco github*, `https://github.com/tensor-compiler/taco`, 2017.

[88] S. Smith et al. "FROSTT: The formidable repository of open sparse tensors and tools."

[89] Y. Zhang, *Graphit github*, `https://github.com/GraphIt-DSL/graphit`, 2018.

[90] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" In *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10, Raleigh, North Carolina, USA: Association for Computing Machinery, 2010, pp. 591–600, ISBN: 9781605587998.

[91] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09, USA: IEEE Computer Society, 2009, pp. 44–54, ISBN: 9781424451562.

[92] E. Mohr, D. A. Kranz, and R. H. Halstead Jr., "Lazy task creation: A technique for increasing the granularity of parallel programs," in *Conference record of the 1990 ACM Conference on Lisp and Functional Programming*, New York, New York, USA: ACM Press, Jun. 1990, pp. 185–197.

[93] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *PLDI*, 1998, pp. 212–223.

[94] Intel, *Intel threading building blocks*, `https://www.threadingbuildingblocks.org/`, 2011.

[95] J. S. Weening, "Parallel execution of lisp programs," Computer Science Technical Report STAN-CS-89-1265, Ph.D. dissertation, Stanford University, 1989.

[96] J. Pehoushek and J. Weening, "Low-cost process creation and dynamic partitioning in Qlisp," in *Parallel Lisp: Languages and Systems*, ser. Lecture Notes in Computer Science, T. Ito and R. Halstead, Eds., vol. 441, Springer Berlin / Heidelberg, 1990, pp. 182–199.

[97] L. Huelsbergen, J. R. Larus, and A. Aiken, "Using the run-time sizes of data structures to guide parallel-thread creation," in *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, ser. LFP '94, Orlando, Florida, USA, 1994, pp. 79–90, ISBN: 0-89791-643-3.

[98] H.-W. Loidl and K. Hammond, "On the granularity of divide-and-conquer parallelism," in *Proceedings of the 1995 Glasgow Workshop on Functional Programming*, 1995, pp. 1–10.

[99] K. Shen, V. Santos Costa, and A. King, "Distance: A new metric for controlling granularity for parallel execution," *Journal of Functional and Logic Programming*, vol. 1999, pp. 1–23, 1999.

[100] A. Duran, J. Corbalan, and E. Ayguade, "An adaptive cut-off for task parallelism," in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, 2008, pp. 1–11.

[101] U. A. Acar, A. Charguéraud, and M. Rainey, "Oracle scheduling: Controlling granularity in implicitly parallel languages," in *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011, pp. 499–518.

[102] U. A. Acar, A. Charguéraud, and M. Rainey, "Oracle-guided scheduling for controlling granularity in implicitly parallel languages," *Journal of Functional Programming (JFP)*, vol. 26, e23, 2016.

[103] M. K. Emani, Z. Wang, and M. F. P. O'Boyle, "Smart, adaptive mapping of parallelism in the presence of external workload," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013, pp. 1–10.

[104] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," *ACM SIGARCH computer architecture news*, vol. 39, no. 1, pp. 199–212, 2011.

[105] S. Campanoni, T. Jones, G. Holloway, G. Y. Wei, and D. Brooks, "The helix project: Overview and directions," in *DAC Design Automation Conference 2012*, Jun. 2012, pp. 277–282.

[106] F. Sironi et al., "Metronome: Operating system level performance management via self-adaptive computing," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12, San Francisco, California: ACM, 2012, pp. 856–865, ISBN: 978-1-4503-1199-1.

[107] MITRE, *CVE-2020-15678: Iterator invalidation in APZCTreeManager::ComputeClippedCompositionBou leading to use-after-free in Mozilla Firefox, Thunderbird, and Firefox ESR*, Common Vulnerabilities and Exposures, Accessed: 2026-05-22, 2020.

[108] MITRE, *CVE-2026-2441: Iterator-invalidation flaw in the CSSFontFeatureValuesMap implementation in Google Chrome, actively exploited in the wild*, Common Vulnerabilities and Exposures, Accessed: 2026-05-22, 2026.

[109] Boost Project, *Boost.container flat_map*, Boost documentation, Accessed 2026-05-21, 2026.

[110] Boost Project, *Boost.container stable_vector*, Boost documentation, Accessed 2026-05-21, 2026.

[111] Abseil, *Abseil container guide*, Project documentation, Accessed 2026-05-21, 2026.

[112] LLVM Project, *LLVM Programmer's Manual: Densemap*, Project documentation, Accessed 2026-05-21, 2026.

[113] T. McMichen, N. Greiner, P. Zhong, F. Sossai, A. Patel, and S. Campanoni, "Representing data collections in an SSA form," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024.

[114] cppreference.com, *std::vector::erase*, C++ reference, Accessed 2026-05-21, 2026.

[115] cppreference.com, *std::list::erase*, C++ reference, Accessed 2026-05-21, 2026.

[116] cppreference.com, *std::map::erase*, C++ reference, Accessed 2026-05-21, 2026.

[117] cppreference.com, *std::unordered_map*, C++ reference, Accessed 2026-05-21, 2026.

[118] S. Klabnik, C. Nichols, and the Rust Community, *The rust programming language: Understanding ownership*, Rust documentation, Accessed 2026-05-21, 2026.

[119] B. Stroustrup, H. Sutter, et al., *Lifetime safety: Preventing common dangling*, ISO C++ paper P1179R1, 2019.

[120] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 2007.

[121] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference*, 2012.

[122] G. Balogh, *Finding iterator-related errors with clang static analyzer*, EuroLLVM Developers' Meeting presentation, Accessed 2026-05-21, 2018.

[123] LLVM Project, *Clang-tidy documentation*, Project documentation, Accessed 2026-05-21, 2026.

[124] LLVM Project, *clang-tidy: bugprone-misplaced-iterator-on-stl-container and related iterator-invalidation checks*, Project documentation, `https://clang.llvm.org/extra/clang-tidy/`, Accessed: 2026-05-22, 2026.

[125] K. Higgs, *Detecting Iterator Invalidation with CodeQL*, Trail of Bits Blog, and `trailofbits/iterga` GitHub repository, `https://blog.trailofbits.com/2020/10/09/detecting-iterator-invalidation-with-codeql/`, Accessed: 2026-05-22, 2020.

[126] GitHub, *CodeQL: A semantic code analysis engine*, `https://codeql.github.com/`, Accessed: 2026-05-22, 2026.

[127] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling Static Analyses at Facebook," *Communications of the ACM*, vol. 62, no. 8, pp. 62–70, 2019.

[128] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional Shape Analysis by Means of Bi-Abduction," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009, pp. 289–300.

[129] R. Mihai, F. Pop, and B. Tudor, *A Static Analysis Framework for Detecting Use-After-Free Bugs in C++*, arXiv preprint arXiv:2410.23764, Accessed: 2026-05-22, 2024.

[130] Y. Sui and J. Xue, "SVF: Interprocedural Static Value-Flow Analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction (CC)*, 2016, pp. 265–266.

[131] PVS-Studio, *V789: Iterators for the container used in the range-based `for` loop become invalid upon a function call*, Project documentation, `https://pvs-studio.com/en/docs/warnings/v789/`, Accessed: 2026-05-22, 2017.

[132] MITRE, *CWE-672: Operation on a resource after expiration or release*, Common Weakness Enumeration, Accessed 2026-05-21, 2026.

[133] NIST SAMATE, *Juliet c/c++ 1.3 test suite*, Software Assurance Reference Dataset, Accessed 2026-05-21, 2017.

[134] Z. Xu et al., "Prompt: A fast and extensible memory profiling framework," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024.

[135] J. Giordani et al., "Revisiting computation for research: Practices and trends," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 1–14.

[136] E. Tiotto, B. Mahjour, W. Tsang, X. Xue, T. Islam, and W. Chen, "OpenMP 4.5 compiler optimization for GPU offloading," *IBM Journal of Research and Development*, vol. 64, no. 3/4, 14:1–14:11, 2020.

[137] J. Doerfert et al., *GPU first – execution of legacy CPU codes on GPUs*, 2023. arXiv: 2306.11686 [cs.DC].

[138] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. S. Müller, "CHAMELEON: Reactive load balancing for hybrid MPI+OpenMP task-parallel applications," *Journal of Parallel and Distributed Computing*, vol. 138, pp. 55–64, 2020.

[139] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[140] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys*, vol. 51, no. 5, 96:1–96:42, 2018.