Program State Element Characterization

Enrico Armenio Deiana
Northwestern University, USA

Brian Suchy
Northwestern University, USA*

Michael Wilkins
Northwestern University, USA

Brian Homerding
Northwestern University, USA

Tommy McMichen
Northwestern University, USA

Katarzyna Dunajewski
Northwestern University, USA

Peter Dinda
Northwestern University, USA

Nikos Hardavellas
Northwestern University, USA

Simone Campanoni
Northwestern University, USA

Abstract

Modern programming languages offer abstractions that simplify software development and allow hardware to reach its full potential. These abstractions range from the well-established OpenMP language extensions to newer C++ features like smart pointers. To properly use these abstractions in an existing codebase, programmers must determine how a given source code region interacts with Program State Elements (PSEs) (i.e., the program's variables and memory locations). We call this process Program State Element Characterization (PSEC). Without tool support for PSEC, a programmer's only option is to manually study the entire codebase. We propose a profile-based approach that automates PSEC and provides abstraction recommendations to programmers. Because a profile-based approach incurs an impractical overhead, we introduce the Compiler and Runtime Memory Observation Tool (CARMOT), a PSEC-specific compiler co-designed with a parallel runtime. CARMOT reduces the overhead of PSEC by two orders of magnitude, making PSEC practical. We show that CARMOT's recommendations achieve the same speedup as hand-tuned OpenMP directives and avoid memory leaks with C++ smart pointers. From this, we argue that PSEC tools, such as CARMOT, can provide support for the rich ecosystem of modern programming language abstractions.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → *Parallel programming languages.*

Keywords: code optimization, dynamic analysis, program characterization

CGO '23, February 25 – March 1, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0101-6/23/02…$15.00
https://doi.org/10.1145/3579990.3580011

ACM Reference Format:
Enrico Armenio Deiana, Brian Suchy, Michael Wilkins, Brian Homerding, Tommy McMichen, Katarzyna Dunajewski, Peter Dinda, Nikos Hardavellas, and Simone Campanoni. 2023. Program State Element Characterization . In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23), February 25 – March 1, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579990.3580011

1 Introduction

Programming languages evolve to give programmers powerful abstractions that improve performance, energy savings, and code clarity. For example, abstractions introduced by the OpenMP language extensions enable programmers to obtain additional performance by taking advantage of the multiple cores available in a chip. Unfortunately, programmers often struggle to use abstractions properly, which leads to performance and correctness issues. The difficulty in using abstractions lies in the implicit requirement that programmers must be omniscient regarding the behavior of the whole program. For example, wrapping a code region in an OpenMP abstraction to express its parallelism requires programmers to understand all possible data and control flows that connect the outside code (potentially from anywhere in the program) to the inside of the code. More generally, programmers need to understand how variables and memory locations evolve as the program executes from the point of view of a target code region (e.g., a memory object is always written before being read in a code region) to properly use modern programming language abstractions. Variables and memory locations (i.e., globals, heap, and stack) form the state of a program. We refer to them as Program State Elements (PSEs).

We observe that many abstractions rely on a common piece of information related to the access pattern of PSEs. Our approach studies this access pattern for the code region where the abstraction is to be applied. We define a new concept that summarizes the impact of this access pattern, which we call Program State Element Characterization (PSEC). PSEC describes: (a) which, where, and how PSEs are used in a code region, (b) how data of PSEs flows across code region boundaries, and (c) the reachability relationships between different PSEs. Intuitively, the PSEC of a code region

* Now at Google.

assists programmers by formalizing their mental process when thinking about abstractions. For example, when parallelizing a program's loop using OpenMP, a programmer needs to understand how PSEs are defined in the program, which ones are only read, which are written, and in what order. Programmers then use this information to identify the PSEs that can be shared between threads, the PSEs that need to be privatized, those that can be reduced, and the code regions that have to become OpenMP critical sections as they access the same PSEs in parallel. Only after gathering all of this information can programmers build an OpenMP pragma and the necessary preparation code that might come with it; for example, the extra code needed to privatize (when possible without changing the program's semantics) PSEs that reside in memory. PSEC presents this information to programmers in human-readable form by reporting source code level information as instances of the target abstraction.

Building a tool capable of automating PSEC is challenging for two reasons. First, automating PSEC with only static code analyses is a challenge due to potential memory aliasing and caller-callee relations that are unknown at compile time. Second, performing PSEC at run-time is challenging because of its computation and memory requirements. This is because PSEC requires tracking the whole memory of a program as well as all variables and all their run-time accesses performed within the abstraction's code region. Notice that although tools exist to track the memory of a program, such as Valgrind [42] and AddressSanitizer [47], they cannot perform PSEC because they only track memory accesses. In other words, they ignore the vast bulk of PSE accesses (§2.3), those to function's variables (allowing them to operate with low overhead). Further, existing tools have no need to integrate information about accesses, which is required for PSEC, and therefore they do not need to preserve such information, allowing them to operate with low memory.

We are the first to automate PSEC. We do it with the *Compiler And Runtime Memory Observation Tool (CARMOT)*, an open-source tool that includes three components: an LLVM-based compiler with PSEC-specific optimizations, a co-designed runtime that profiles how the target PSEs evolve, and a custom Pintool that tracks PSEs in pre-compiled code. A programmer invokes CARMOT with the abstraction they would like to use on a given code region (e.g., OpenMP parallel for). CARMOT then generates abstraction recommendations by synthesizing an instance of the target abstraction using the PSEC of the target program.

To demonstrate the power and flexibility of PSEC, we use CARMOT to generate recommendations for five important abstractions ranging from traditional to emerging. The abstractions are: OpenMP critical section, OpenMP parallel for, OpenMP task, C++ smart pointers, and the Input-Output-State abstraction needed by the STATS compiler for non-deterministic programs [21]. We showcase the power of CARMOT's recommendations by automatically checking

```
0  int work(int a, int b){
1      int i, x, y;
2      y = 42;
3      for (i = 0; i < 10; ++i){
4          #pragma carmot roi{
5              x = i/(a + b);
6              y /= a*x + b;
7          }
8      }
9      return y;
10 }
```

Figure 1. CARMOT automatically builds the PSEC containing the information to parallelize this for-loop.

for correctness of existing OpenMP pragmas, and generating new pragmas for benchmarks from PARSEC [17], SPEC CPU 2017 [11], and NAS [16]. On average, the pragmas that result from CARMOT's recommendations speed up program execution by 8× compared to the original, serial version, and they match or outperform the manually (and labor-intensive) parallelized version. We also show that CARMOT finds and breaks cycles in reference counting garbage collection used by C++ smart pointers so programmers can safely use this abstraction without introducing memory leaks. For example, CARMOT was able to easily discover a complex reference cycle in the SPEC CPU 2017 benchmark `nab` that spans across multiple files and functions. Finally, the STATS abstraction automatically generated by CARMOT in a few minutes outperforms those that the STATS authors manually defined after years of work.

We summarize our contributions as follows:

- We observe that PSEC is the common information needed to use numerous abstractions correctly and at their full potential. We illustrate this need through an in-depth study of five popular abstractions (§2).
- We formally define PSEC (§3).
- We make PSEC practical by introducing novel compiler and runtime optimizations within CARMOT (§4).
- We evaluate CARMOT on a wide range of well-known benchmark suites and illustrate its benefits in terms of generating abstractions' recommendations with reasonable overhead (§5).

2 Background and Motivation

Using programming language abstractions in a large codebase is challenging and error-prone due to the lack of tools to assist programmers [48]. Here we describe three use cases exploring five different abstractions and how they support and challenge programmers. We show how PSEC is the common information necessary to build tools that help programmers use these abstractions. Finally, we show that prior work is insufficient as they use either static analysis only [1, 15, 18, 45] or limited dynamic strategies [35, 39, 43, 51] that limit them to a few simple abstractions.

2.1 Challenges in Adopting Abstractions

Program Parallelization/Synchronization. OpenMP has high-level abstractions to parallelize loops (*#pragma*

omp parallel for), synchronize parallel accesses (*#pragma omp critical/ordered*), and asynchronously execute units of computation (*#pragma omp task*). Using these pragmas to their full potential quickly becomes complex as they often require both the specification of their attributes and extra code to prepare the target code region for efficient parallel execution. For example, *#pragma omp parallel for* requires programmers to understand which PSE variable needs to be privatized per thread (using the *private* attribute), which can be shared (*shared* attribute), and which code statement uses PSEs involved in true data dependencies that should be in a *#pragma omp critical/ordered* section. Also, programmers need to understand how a *private* PSE variable interacts with the code outside the target loop. Variables that are written before the loop and read inside need to be declared as *first private*, while variables written inside the loop and read after need to be *last private*. Furthermore, programmers might have to write additional preparation code to, for example, clone PSEs that are more complex than variables (e.g., arrays, objects). This requires knowledge of where and how these PSEs are allocated (e.g., their size, type, alignment). Similarly, *#pragma omp task* requires an understanding of which PSEs are consumed/produced by the task through the *depend(in/out)* attribute. Failing to correctly classify PSEs or their code statements results in invalid or inefficient code.

Managing Dynamic Memory. C++ programmers used to manually manage the dynamic memory of a program. To help with this task, modern C++ standards (as of C++11) have added the smart pointer abstraction. Smart pointers manage dynamic memory using reference counting, which tracks the number of pointers to a dynamic PSE object and deletes it when the count drops to zero. Unfortunately, using smart pointers can lead to memory leaks when there are cycles in the reference counting graph of PSEs. Programmers have limited tool support to detect when cycles occur and no support to identify and break them. This is particularly challenging when reference cycles cross many functions and source code files, making manual detection difficult.

Declaring State Dependences with STATS. STATS [21] is a compiler for parallelizing non-deterministic programs. STATS requires a programmer to follow a given code structure (i.e., the Input-Output-State abstraction of STATS), which makes the compiler aware of the STATS *state dependence* (i.e., a Read-After-Write (RAW) dependence that can be satisfied in an alternative way following the STATS execution model). To do so, a programmer needs to classify the PSEs accessed by the code region where STATS operates into three classes: 1) Input class (PSEs that are only read), 2) Output class (PSEs that are written first), 3) State class (PSEs that are read first and then written). Understanding which PSE goes into which class often requires a programmer to understand the behavior of the entire program. Misclassifications lead to performance degradation or an incorrect program.

2.2 Benefits of PSEC

PSEC conveniently summarizes the knowledge of how the program state is affected by a code region that abstractions of §2.1 require. We now give an intuition about how to collect and apply PSEC to use the abstractions *#pragma omp critical/ordered* and *#pragma omp parallel for*. We formalize this process for the other abstractions in §3.2. Consider the loop in Figure 1 to be the code region where a programmer wants to apply *#pragma omp parallel for*. PSEC would classify the PSEs affected by the loop's body as follows: variables a and b are only read, variables x and i are always written before being read, and variable y is read and then written. With this information, CARMOT generates the *#pragma omp parallel for* with the correct attributes and critical section. In more details, CARMOT lists variables a and b as *shared* because they are only read, hence multiple threads can read their value at the same time without making extra copies. Variables x and i are instead declared as *private* because their value changes throughout loop iterations, hence multiple threads must have private copies to work on. Finally, variable y introduces a RAW loop-carried data dependence and cannot be put into a *reduction* clause because of the division operation performed on it. So, CARMOT recommends wrapping the statement that uses y (i.e., line 8) in *#pragma omp ordered* because the division operation is not commutative, hence the order of operations must be preserved.

2.3 Overhead of PSEC

On top of tracking memory accesses (i.e., reads and writes) like other memory-tracking tools do [42, 47], PSEC needs to track accesses to function variables for a target code region to obtain complete information of the program state. We measured the increase of accesses and observed 8× more accesses on average that need to be tracked for PSEC. Other tools do not have to track function variables because their only goal is to validate memory accesses. Hence, these tools are able to invoke many general-purpose compiler optimizations, which are not compatible with PSEC (e.g., mem2reg disrupts the mapping between source code and IR variables). This is why our approach requires a more involved compiler-based solution including several PSEC-specific compiler and runtime optimizations.

2.4 Limitations of Current Dynamic Analyses

As §2.1 shows, PSEs are explicitly used in many modern programming language abstractions. Despite their important role, this paper is the first one to consider PSEs as first class citizens. Dynamic analyses of prior works [35, 43, 51] are instead based on dependencies or memory footprint of instructions. This limits prior work to imprecise and overly-conservative information that can defeat the purpose of using an abstraction altogether. Figure 2 shows an example of the fundamental limitations of such dynamic analyses. In this example i spans from *0* to *N-1*, while j assumes the values *{1,0,0,2,3,...,N-2}*. The corresponding dependence graph and

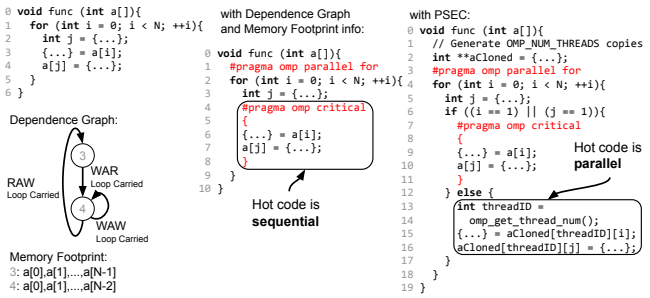

Figure 2. State of the art dynamic analyses based on dependence graph and/or memory footprint of instructions miss important parallelization opportunities compared to PSEC.

memory footprint for the relevant instructions (i.e., *3* and *4*) are reported in Figure 2. If programmers want to parallelize the for-loop in the example following the information provided by the dependence graph and memory footprint of the program, they need to be conservative and assume that any element of the memory object *a* can be involved in the loop-carried RAW dependence. This results in placing the most computationally intensive part of the loop body in a critical section, which runs sequentially and defeats the purpose of parallelizing the loop in the first place. This fundamental limitation becomes even more severe as the size of *a[N]* grows. PSEC instead reports to the programmer that only a small portion of *a* (*a[1]* in our example) is involved in the loop-carried RAW dependence. Hence, the programmer can considerably shrink the critical section and clone the rest of *a* to remove the loop-carried WAR and WAW dependencies, regaining parallelism.

3 Program State Element Characterization

The generation of abstractions of §2.1 requires three essential pieces of information about PSEs: classification (e.g., only read PSEs), contextualization (when and where PSEs are used in the program), and reachability (PSEs that reference other PSEs). In this section we describe how PSEC provides this information and how it is used to automatically generate the abstractions we target.

3.1 Components of PSEC

PSEC has three components (Table 1): *Sets* to classify PSEs, *Use-callstacks* to contextualize computation, *Reachability Graph* to represent reachability relationships between PSEs.

Sets. We define Program State Elements as the set of memory locations (stack and heap) and variables (local and global) of a program at the source code level. Also, we define Region of Interest (*ROI*) as a single-entry single-exit code region [28]. Examples of an ROI are a single statements, an if-then-else code block, a loop, or a function. PSEC is related to an ROI and contains information about how PSEs are read and/or written by that ROI.

PSEC classifies the ROI's access to PSEs into four *Sets*. Each set indicates how an ROI in the source code interacts with (i.e., reads/writes) PSEs. The sets that comprise a PSEC for a dynamically invoked ROI *Z* are:

Input set: PSEs read by a dynamic invocation of *Z* before being written by any invocation of *Z*. This set represents the input of *Z* as these data are generated by the code outside *Z* and consumed by *Z*.

Output set: PSEs written in a dynamic invocation of *Z* and read outside *Z*. This set represents the output of *Z* as this data is generated by *Z* and consumed by the code outside *Z*.

Cloneable set: PSEs written by more than one invocation of *Z* where no subsequent invocation reads them before overwriting them. This set represents data locations reused by invocations of *Z* without triggering RAW data dependences.

Transfer set: PSEs written by an invocation of *Z* and then read by a subsequent invocation of *Z* before any potential overwrites. This set represents the data generated by an invocation of *Z* and consumed by a subsequent invocation of *Z*, triggering a RAW data dependence.

Three pieces of information are necessary to classify PSEs in the correct set of a PSEC. First, we need to know *where* PSEs are allocated in the source code. Second, we need the *context* of such allocations. As context we use the callstacks that lead to the code statements that performed such allocations. The context of allocations is necessary because the same static code statement that generates PSEs can be used in different parts of the program, and the programmer must be able to distinguish them. For example, custom allocators are widely used in large codebases. Without knowing the callstack all allocations would look like they are coming from the allocation statement in the custom allocator, which is not useful information. Third, we need to *record reads and writes* the ROI performs on all PSEs to characterize them correctly. We call these accesses *uses* of PSEs.

Use-Callstacks. The program statements in an ROI (i.e., the uses of PSEs) can be executed multiple times from different parts of a program. To take this into account, we record the callstack of each statement invocation. We refer to these statements and their recorded callstacks as *Use-callstacks* of a PSEC. Knowing *Use-callstacks* enables us to disambiguate a static statement when invoked from different parts of the program, which can lead to a PSE being classified in different *Sets* of a PSEC. This is useful, for example, to report precisely which statement must be in a critical section.

Reachability Graph. PSEs can reference other PSEs in different points of a program. PSEC collects reference information through its *uses* of PSEs. Specifically, recording pointer escapes of PSEs. Escapes are recorded in the PSEC *Reachability Graph* where nodes are PSEs allocated within the PSEC's ROI and edges are references that point to other PSEs. We use this information to keep track of how PSEs reference each other to, for example, identify reference cycles.

3.2 From PSEC to Abstractions

Programmers declare the abstraction to apply to a given ROI to CARMOT. Then, CARMOT uses an ROI's PSEC to

Table 1. Different abstractions need different parts of PSEC.

Abstraction	PSEC		
	Sets (I,O,C,T)	Use-callstacks	Reachability Graph
OMP parallel for (and critical/ordered)	✓	✓	✗
OMP task	✓	✗	✗
Smart Pointers	✓	✗	✓
STATS	✓	✗	✗

automatically generate new source code with the requested abstraction in it and customizes it with the correct attributes (Table 1 illustrates which parts of PSEC are necessary to generate each abstraction). Next we describe the generation of abstractions from PSEC.

Program Parallelization/Synchronization. To generate a *#pragma omp parallel for* with the correct attributes CARMOT uses the *Sets* of the PSEC as follows. For every PSE e in the Cloneable set, CARMOT extracts the callstack for the element's allocation. These PSEs and their callstacks tell us what needs to be cloned to remove WAR and WAW data dependences between invocations of the related ROI (e.g., the body of a loop). If e is a variable, then CARMOT privatizes it in the generated pragma. Variables that are also in the Output set are declared as *lastprivate*, since they can be read after the ROI. Similarly, variables that are also in the Input set are declared as *firstprivate*, since they were first read inside the ROI. If e is a memory location, then CARMOT advises programmers to clone the PSE (CARMOT's output provides the allocation site and its callstack to help programmers understanding how to perform the cloning) and use the OpenMP API *omp_get_thread_num()* to access the correct clone of that allocation in the ROI. PSEs that belong only to the Input set are declared as *shared* in the pragma because they are only read. Finally, for each PSE e in the Transfer set CARMOT retrieves its *Use-callstacks*. If e is a variable, then CARMOT checks each use of e to understand if the computation performed on e is reducible (i.e., the statement uses one of the OpenMP-supported reduction operators such as +). If the computation is reducible, then CARMOT includes e and the supported operation in the *reduction(operator:variable)* attribute. Otherwise, all statements that access e are wrapped in a *#pragma omp critical* or *#pragma omp ordered* section. Note that CARMOT leaves the decision as to which abstraction to use, either critical or ordered, to programmers as they know whether it is necessary to preserve the loop iteration order. CARMOT generates dependences for *#pragma omp task* as follows. The Input and Output sets of a computational spoor are mapped to the *depend* attribute of *#pragma omp task*. All PSEs e in the Input set are declared as *depend(in:e)*. Similarly, all PSEs e in the Output set are declared as *depend(out:e)*.

Managing Dynamic Memory. Cycles between PSEs allocated in an ROI are detected using the PSEC *Reachability Graph*, which tracks references between PSEs. CARMOT reports detected reference cycles to programmers and can also

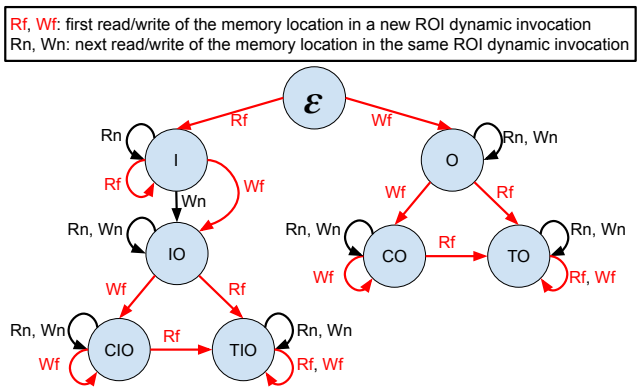

Figure 3. PSEC follows a Finite State Automaton (FSA).

suggest which reference should become a weak pointer[1] to break a detected reference cycle. It does so by identifying the node in that cycle that has the oldest access time. This enables programmers to gradually port ROIs within a large codebase to use smart pointers without introducing cycles.

Declaring State Dependences with STATS. The STATS abstraction Input-Output-State can be mapped directly from the *Sets* of an ROI's PSEC. PSEs classified in the Input, Output, or Transfer sets are respectively mapped to the Input, Output, State classes of the STATS abstraction. The STATS abstraction requires the target ROI to be explicitly moved into a separate function; hence, PSEs in the Cloneable set are declared locally in that function. This localization enables the STATS compiler to spawn independent parallel threads to execute the related ROI.

4 CARMOT

Here we describe how CARMOT performs PSEC, its compiler, Pintool, runtime, and PSEC-specific optimizations.

4.1 PSEC with CARMOT

CARMOT performs PSEC of an ROI independently of other ROIs. When a PSE (e.g., variable) is accessed within an ROI, CARMOT classifies it into the *Sets* of that ROI's PSEC following the Finite State Automaton (FSA) shown in Figure 3. Each PSE has an instance of this FSA. PSEs start in the ε state. A PSE is added to the PSEC of an ROI Z upon its first access within Z. Subsequent accesses of a PSE in Z might change its FSA's state for Z. At the end of a program's execution, the final FSA state of a PSE for Z reflects the set (or sets) that the PSE belongs to with respect to ROI Z. In more detail, if the terminal FSA state includes an `I`, `O`, `C`, and/or `T`, then the related PSE belongs to the Input, Output, Cloneable, and/or Transfer set respectively. Note that a PSE can never be both in the Cloneable and Transfer sets ($C \cap T = \emptyset$).

Let us consider the loop in Figure 1 and the PSE variable y. In the first dynamic invocation of ROI Z, PSE y is first read and then written, hence y transitions from ε to `I` (R_f) and then to `IO` (W_n). In the subsequent dynamic invocation of

[1] A weak pointer does not increment an allocation reference counter.

Z a read of y happens (R_f), which causes a trasition to `TIO`. `TIO` is a sink state, so when the program finishes, CARMOT classifies y in the Transfer, Input, and Output sets.

The FSA only operates on reads and writes that happen within ROIs. This design decision enables CARMOT to avoid profiling code outside ROIs, but it also makes the assumption that PSEs written in an ROI will be read outside the ROI, so they will be part of the Output set. This assumption is conservative and does not affect the correctness of the PSEC.

4.2 Advantages of CARMOT's Dynamic Approach

CARMOT performs PSEC by profiling a specific run of the target program. We envision CARMOT users will perform PSEC on a program multiple times to cover many program inputs, and combine the generated PSEC. Combining PSECs can be done through set union. For example, if PSE e is classified in the Input and Output sets in the first run, and in the Cloneable and Output sets in the second run, the PSEC across runs classifies e in the Cloneable, Input, and Output sets. The only exception to this union rule is when e is in the Cloneable set for one run and in the Transfer set for another run. In this case, the conservative answer is to classify e in the Transfer set. Currently, and only for engineering reasons, CARMOT's users need to manually apply these rules to merge multiple PSEC to gain a more comprehensive understanding of the target program.

CARMOT's dynamic approach goes beyond what can be determined with static code analyses, and provides programmers recommendations and support for programming language abstractions at the source code level for a specific program execution. The advantage of providing recommendations, as opposed to making automatic semantic changes to the code that should not be modified by anyone but the tool itself, is that it makes CARMOT more accessible to programmers. These recommendations allows for a better understanding of code behaviour and provide a starting point to tune abstractions to the programmer's needs. The disadvantage is that verifying the correctness of such recommendations for all prossible program executions has to be done manually. However, we argue that such process is more suitable for humans rather than tools. Programmers can leverage domain specific knowledge about a program to make a decision, while an automatic, semantic-changing tool has to make conservative assumptions when trying to build an abstraction that is sound for all possible program executions. This conservativeness hides the true behavior of the execution of a program, which prevents programmers to reason about their programs and the abstractions they want to use.

4.3 CARMOT as a System

CARMOT implements the compilation flow in Figure 4. CARMOT's compiler (§4.4) generates a binary from C/C++ source files including code instrumentation. Complementarily, CARMOT loads its Pintool (§4.5) into memory to cover code that lacks available sources. CARMOT's runtime (§4.6) is embedded within the generated binary as a static library. The runtime processes the reads and writes provided both by the instrumentation generated by CARMOT's compiler and by its Pintool. This generates the PSEC of each ROI specified by CARMOT's pragma included in the program's source code (Figure 1). The PSEC is then translated into programming language abstraction recommendations for the abstraction chosen by the CARMOT user.

4.4 Compiler

CARMOT's compiler uses `clang` with debugging symbols enabled, but without optimizations, to translate a C/C++ program to LLVM's IR and guarantee a reversible mapping between source code and IR. The advantage of performing PSEC at the IR level rather than at the binary level is two-fold. First, we can easily implement precise and effective specialized code analyses and optimizations. Second, the amount of instrumentation is considerably reduced compared to binary instrumentation where spilling of variables onto memory already occurred, generating extra memory loads and stores. The disadvantage of performing PSEC on unoptimized IR is the high overhead of the profiling phase, making PSEC infeasible for a large codebase. For this reason, CARMOT uses the following PSEC-specific code analyses and optimizations.

1) Subsequent accesses. The FSA of Figure 3 shows that transitions of PSEs to a different state happen only upon the first read or write (R_f, W_f) of a new dynamic invocation of an ROI. The only exception is a subsequent write (W_n) in the same ROI dynamic invocation when the PSE is in the `I` state. Following this observation, this optimization aims to instrument only the first read and write of a PSE and avoid instrumenting subsequent accesses that are proved to always access the same PSE.

We developed a new intra-procedural data-flow analysis to identify where a PSE must have been accessed already since the beginning of an ROI. For this data-flow analysis, predecessors and successors of basic blocks that are outside or leave an ROI are not followed during the data-flow value propagation as only instructions within an ROI need to be considered. We do so by considering the entry point of an ROI as the entry point for our analysis. Elements in the GEN, IN, and OUT sets are the variables and memory locations (i.e., PSEs) of the target program. Given an instruction i that is either a load or a store, the sets for the data-flow analysis are defined as follows. The GEN set of i is the PSE a that a load is guaranteed to access or a store must write to ($GEN[i] = \{a\}$). The IN set of i is first initialized to be the union of all PSEs, and then refined to be $IN[i] = \bigcap_{\forall p \in preds(i)} OUT[p]$, where p are the predecessors of i. The OUT set of i is initially empty, and then refined to be $OUT[i] = IN[i] \cup GEN[i]$. This data-flow analysis runs until a fixed point is reached for each ROI. Elements in the IN set of an instruction i are the PSEs that must have been accessed between the entry of the ROI and i. Hence, CARMOT reduces profiling overhead by avoiding

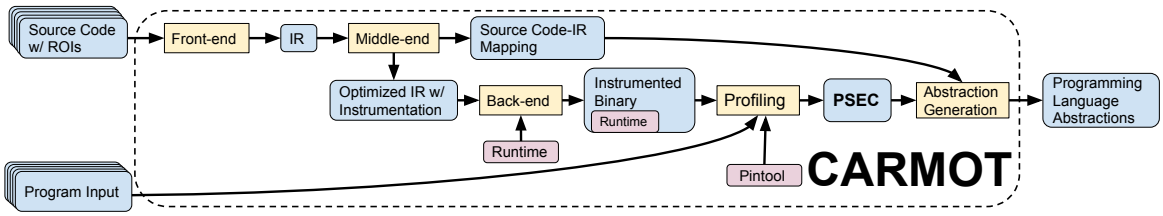

Figure 4. CARMOT produces the mapping between source/IR code and runs an instrumented binary to build the PSEC, which is then used to generate the target abstraction information for the programmer at the source code level.

instrumenting instructions i where the PSE accessed by i belongs to $IN[i]$.

2) PSEs aggregation. Normally, uses of PSEs are instrumented singularly. However, contiguous PSEs that can be indexed (e.g., arryas), for which the same operation is performed at every ROI's dynamic invocation (e.g., they are always only read or only written) are instrumented altogether at once. Currently we limit this optimization on ROIs that wrap the body of a loop for which the loop governing induction variable indexes the contiguous PSEs.

3) Fixed setting of FSA state for PSEs. The FSA in Figure 3 shows that PSEs that are always only read will always be classified as Input in the PSEC. Hence, PSEs that can be verified to be only read at compile time can be instrumented only once and still be correctly classified in the Input set. Although an ROI is a general code region, we currently enable this optimization only for ROIs that wrap the body of a loop. We determine whether a PSE is only read by verifying that the corresponding load instruction is loop invariant. Similarly, the FSA classifies PSEs that are always only written as Output or Cloneable. At compile time, we determine whether a PSE is only written using the PDG. If the store instruction that writes the PSE has no incoming memory dependence edge where the source of the edge is an instruction in the ROI, we set the FSA state of that PSE to be Output. Then, if the considered ROI wraps the body of a loop, we use the loop governing induction variable to determine whether the store to the PSE is executed more than once. If so, we set the FSA state of that PSE to also be Cloneable.

4) Selective mem2reg. The mem2reg optimization is extremely beneficial for performance and to enable further analysis and optimizations, but it cannot be generally applied when performing PSEC (Section 2.3). However, some allocations of PSEs that are local variables can be promoted to registers without affecting the correctness of PSEC. For example, local variables that are never used in any ROIs can be safely promoted to registers because they will not be part of a PSEC, and instrumentation of such variables can be safely removed. Also, local variables with a specific role in an ROI have to be promoted to registers to be identified (e.g., loop governing induction variables). We built a wrapper around the LLVM mem2reg optimization that allows the promotion of specific local variables to registers only when it is safe to do so.

5) Call graph-based optimization. This optimization aims to select functions that can be optimized with conventional transformations while preserving the aforementioned IR-to-source-code mapping needed for PSEC. This optimization is based on the observation that if a function f cannot be in the callstack when any ROI starts, then f can be optimized with conventional optimizations (such as `-O3`) without breaking the IR-to-source-code mapping. This is because any PSE allocated in the stack by f will not be part of the PSEC of any ROI. This holds even if f is invoked within an ROI, as its stack is freed before returning to its caller and therefore such stack PSEs cannot be involved in any data dependences that cross the boundaries of an ROI. Therefore, only PSEs that are heap allocated by f need to be tracked, which are preserved by the optimizations included in `-O3` of `clang`.

To perform this optimization, CARMOT identifies the functions that cannot be in the callstack when any ROI starts by computing the complete callgraph of a program (i.e., a callgraph where the lack of an edge (f_i, f_j) means f_i cannot invoke f_j). Unfortunately, the callgraph provided by LLVM is not complete. To generate the complete callgraph, CARMOT computes the program dependence graph (PDG) to automatically discover the possible callees to which a pointer could refer. CARMOT uses the same memory alias analyses used by the previous optimization. Armed with the complete callgraph, CARMOT identifies the set of functions that can be optimized. For every ROI, CARMOT takes the function f where the ROI belongs to and traverses the edges of the callgraph backwards from f and tags all functions reached, including f. All functions in the program that are not tagged are optimized by CARMOT invoking the `-O3` optimizations of `clang`.

6) Reducing Pin instrumentation. CARMOT uses the call graph to also reduce Pin instrumentation by enabling the Pintool only when it cannot guarantee that a call will not jump to precompiled code.

7) Callstack clustering. CARMOT needs to record the callstack of every PSE allocation. In a typical function, many PSEs are allocated. In a naive implementation, every time an allocation of a PSE occurs the callstack must be computed and assigned to that PSE. However, allocations made within the same invocation of a function share the same callstack. To avoid recomputing the callstack for each PSE allocation within a function, CARMOT computes the callstack only

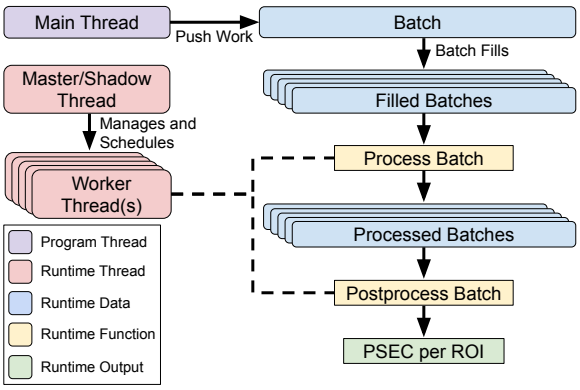

Figure 5. The runtime utilizes batching, shadow profiling, and pipeline parallelism to efficiently perform PSEC.

once at the beginning of the function. The instrumentation that documents allocations can now collectively share the computed callstack instead of producing redundant callstack records that are clones of one another.

4.5 Pin Instrumentation

When the target program includes code outside the available sources (e.g. precompiled libraries), it is impossible to track all PSEs information with a purely compiler-based approach. However, the activity of PSEs outside the available sources must be tracked in order for the PSEC to be correct and complete. To perform this tracking, CARMOT uses dynamic binary instrumentation through a Pintool that builds upon the Pinatrace memory access tracing tool from Intel [10]. The key challenge is to efficiently communicate between the Pintool and the compiler/runtime environment of CARMOT. To overcome these challenges we use compiler injected calls to invoke our Pintool, which tracks allocations and accesses of PSEs in precompiled code and communicates them to the CARMOT runtime. This is a costly operation, but necessary to generate a correct PSEC.

4.6 Runtime

CARMOT's runtime processes the uses of PSEs provided by either compiler-injected instrumentation or the Pintool. This information needs to be processed at run-time as the large amount of data collected makes storage a bottleneck.

The primary structures the runtime builds are the Active State Member Table (ASMT) and the ROIs' PSEC. The ASMT captures metadata about *active* PSEs such as the callstack of their allocation and size in bytes. The runtime generates a PSEC by enacting the FSA (§4.1) upon PSEs for each ROI.

Figure 5 shows the components and the processing flow of the runtime. The *Main Thread* runs the target program. The compiler-injected instrumentation and Pintool push requests into a *batch*. Once a batch fills, it is pushed into an ordered queue of filled batches, and the instrumentation calls begin filling a new batch. The *Master/Shadow Thread* schedules filled batches for processing by *Worker Threads*. Each processed batch is then added to a second ordered queue for final processing. The results of processed batches are updates to the ASMT and PSEC. The batches are processed following a parallel pipeline:

Processing batches. This stage processes the instrumentation calls to build the ROIs' PSEC for all PSEs. It does so by implementing the FSA in Figure 3 on active PSEs. Once the batch has been processed, it is then queued to the next pipeline stage and the next batch can be processed.

Postprocessing batches. This stage adds contextual information to the ROIs' PSEC and connects metadata to PSEs. This includes the callstack, escaped pointers, source code information for PSEs (file and line), and accesses.

5 Evaluation

To show the effectiveness of CARMOT, we evaluate 15 benchmarks from the SPEC CPU 2017, NAS [16], and PARSEC (version 3.0) [17] benchmark suites. We include every benchmark from all of these suites that already use, or are well suited for, the abstractions that CARMOT currently supports. When evaluating the performance benefits of CARMOT we used the "reference", "class C", and "native" inputs, respectively. When evaluating the overhead of CARMOT we used the "test", "class A", and "simsmall" inputs. The difference in inputs for performance and overhead results reflects how we see CARMOT being used. The larger, production level inputs are used for the performance results as this is indicative of the actual speedup that programs developed with CARMOT can attain. Smaller, representative inputs follow CARMOT being used to determine the PSEC of the program at development time.

Our evaluation is done on a dual socket server with two Intel Xeon Silver 4116 CPU running at 2.1GHz. Each processor has 12 cores with 2-way hyper-threading, and 16.5 MB of last level cache. The cores are supported by 125 GiB of main memory at 2400 MHz. The OS is Red Hat Enterprise Linux 8.2 (kernel 4.18.0-193.6.3). CARMOT is built on top of LLVM 9.0.0 [34], Pin 3.13 [36], and NOELLE 9.3 [37]. The baseline for both performance and overhead evaluation we show is the sequential version of each benchmark, compiled with `clang -O3 -march=native`.

5.1 OpenMP Use Case

Using PSEC, CARMOT is able to automatically generate *#pragma omp parallel for*, *#pragma omp critical*, *#pragma omp ordered*, and *#pragma omp task* annotations, and can be used by developers to verify the correctness and improve the performance of existing pragmas for a specific program execution. Many of the benchmarks we consider for this use case already use OpenMP pragmas. In this case, we choose as ROIs for PSEC the code regions of the already present OpenMP pragmas and we verified that CARMOT's recommendations matched the original pragmas and our understanding of the parallelism in the benchmark. In cases where the benchmark is parallelized using pthreads (e.g., `swaptions` from PARSEC 3), we use as ROI the entry point function of such threads to

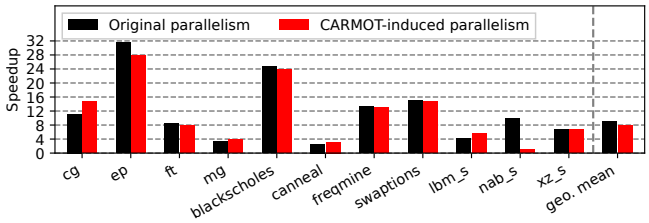

Figure 6. CARMOT-generated OpenMP pragmas achieve the same speedup of the original program parallelism manually implemented by a programmer. These experiments use the production-size inputs.

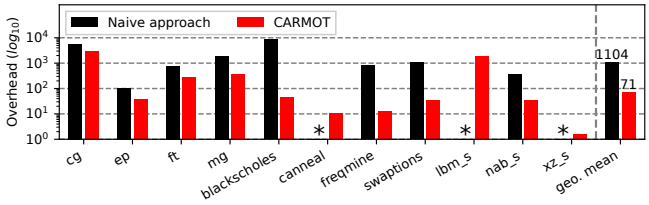

Figure 7. The CARMOT overhead to generate OpenMP pragma information is two orders of magnitude less than a naive approach.

build equivalent parallelism using CARMOT's recommended OpenMP pragmas. Furthermore, we use CARMOT to implement additional parallelization opportunities; for example, we add some OpenMP task parallelism to `mg` from NAS.

Figure 6 shows the speedup benefits of automatic CARMOT-generated pragmas (either verified pragmas or brand new ones) versus the original (manually extracted by the benchmarks' authors) parallelism (either through omp pragmas or pthread) for each benchmark we consider. This data shows that with CARMOT-generated pragmas, we are able to achieve speedups that are as good as or better than pragmas implemented manually by a programmer. For benchmarks like `canneal` and `swaptions`, where the only original source of parallelism comes from pthreads, the new pragmas generated by CARMOT match the performance of the labor-intensive pthreads parallelism. The only exceptions are `ep` and `nab` for which CARMOT was unable to extract all parallelism potential. In both cases the main source of parallelism in these benchmarks comes from general OpenMP *#pragma omp parallel* sections that include synchronization mechanisms such as *#pragma omp barrier* or *#pragma omp master* that are abstractions currently not supported by CARMOT.

When designing new development tools, striking a balance between effectiveness and feasibility is paramount. The feasibility of CARMOT as a tool is measured by the computational overhead required to perform PSEC. Figure 7 shows the computational overhead of CARMOT when automatically generating OpenMP pragmas information. We compare the CARMOT overhead with a naive approach that does not employ any PSEC-specific optimization, but can still generate a correct PSEC. CARMOT outperforms the naive approach

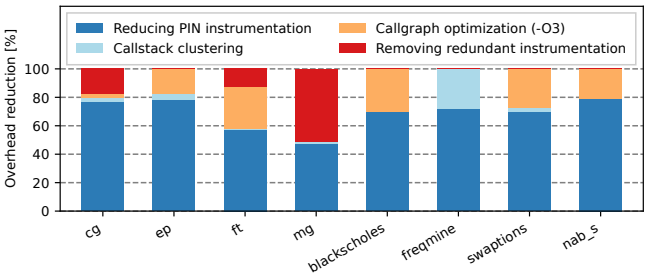

Figure 8. Overhead reduction of Figure 7 characterized per CARMOT optimization.

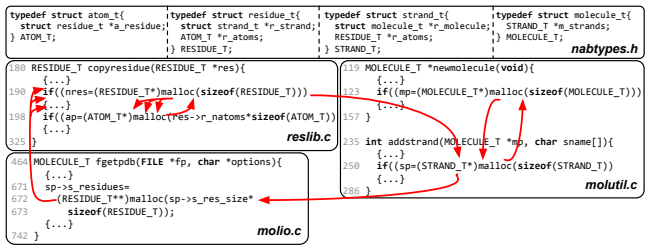

Figure 9. CARMOT-identified reference cycle across files, functions, and data structure in the `nab` benchmark.

by lowering the overhead of performing PSEC by two orders of magnitude. In some cases the execution of the naive approach required an excessive amount of memory and did not complete, we mark the missing data with ∗.

To showcase the power of PSEC-specific optimizations, Figure 8 shows the impact of the optimizations described in §4.4. For the benchmarks where the naive approach finished successfully, we show in percentage the breakdown of the delta between the black and red bars of Figure 7 for every CARMOT optimization. The reduction of Pin instrumentation and the callgraph-based optimization, enabled by the complete callgraph of NOELLE, have the highest impact. Optimizations from 1) to 4) of §4.4 collaboratively enable each other to remove redundant instrumentation, for this reason we consider them together. Because they heavily rely on alias analysis, they have the highest impact in the more regular benchmarks from NAS.

5.2 Smart Pointers Use Case

Here we show the versatility of CARMOT on a use case unrelated to parallelization: identifying reference cycles in an ROI. In this use case we choose the ROI for PSEC to be the entire program (i.e., the entire *main()* function), since we are interested in any possible reference cycle in the program.

Figure 9 shows an example of a reference cycle that CARMOT identified in the `nab` benchmark of the SPEC CPU 2017 suite. This cycle spans across several different files, functions, and data structures and demonstrates the complexity of porting an existing application to use smart pointers correctly. We measure the benefit that utilizing smart pointers for this reference cycle would generate for the benchmark. After correcting a naiveness in the original `nab` code which over allocates memory, we measure the total bytes leaked by

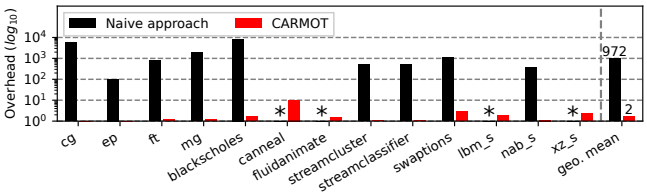

Figure 10. The CARMOT overhead for identifying reference cycles is two orders of magnitude less than a naive approach.

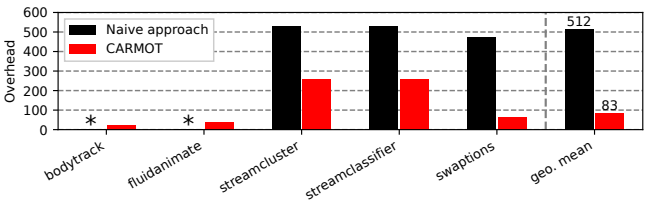

Figure 11. The CARMOT overhead to generate the STATS abstraction is one order of magnitude less than a naive approach.

the application as 230,537 bytes. The total bytes leaked by the application that would have been realized by correctly porting this reference cycle to smart pointer is reduced to 127,633, a reduction of 44.6%.

Figure 10 shows the overhead of CARMOT when finding reference cycles. In this use case CARMOT only needs to track allocations of PSEs and the *Reachability Graph* of such allocations. For this reason, CARMOT's overhead is two orders of magnitude smaller than a naive approach that lacks PSEC-specific optimizations.

5.3 STATS Use Case

Here we show that CARMOT can be used to build the Input, Output, and State classes required by the STATS abstraction. In the benchmarks considered we choose the ROI for PSEC to be the code region of the STATS state dependence. CARMOT is able to accurately generate the Input, Output, State classes required by the STATS abstraction, such that they match those manually implemented by the authors of STATS. Furthermore, CARMOT was able to identify some misclassifications of PSEs made by the authors of STATS. While these misclassifications have no impact on correctness, they lead to extra unnecessary copies of variables. In this case, the misclassification does not lead to a noticeable speedup. However, CARMOT's ability to outperform the manual and labor-intensive classification lends to its usefulness as a tool for abstractions that are difficult for developers to use correctly.

Figure 11 shows CARMOT's overhead for classifying PSEs into STATS's Input, Output, and State classes. We can see that the CARMOT overhead is one order of magnitude lower than a naive approach. This is due to the STATS abstraction not requiring the tracking of all *Use-callstacks*, a very costly operation, and the PSEC-specific optimizations of CARMOT.

6 Related Work

While CARMOT is the only tool capable of computing a complete and correct PSEC, there are other tools that enable programmers to better understand a program's behavior and how to improve it. Next, we compare CARMOT to these tools with respect to their ability to track PSEs, build aspects of the PSEC, and report back information to the user at the source code level. We categorize these tools in three sets.

Memory analysis. These tools investigate memory correctness such as memory leaks, double frees, and buffer over/underflow [2–10, 23, 42, 47] or memory bottlenecks [32, 38, 40, 44]. Some of these tools report some source-code level information such as the callstack of the error site; however, none of them track any aspect of PSEC. The most notable tools that perform some tracking of PSEs are: AddressSanitizer [47], Valgrind [42], and the Pintool Pinatrace [10]. However, none of these tools track PSEs that are variables or are able to distinguish between different stack locations or globals. AddressSanitizer and Valgrind can detect memory leaks due to reference cycles that should have been garbage collected, but they cannot identify the actual cycles in the source code responsible for the leak.

Parallelism discovery. These tools [13, 14, 19, 24–27, 29–31, 33, 35, 39, 41, 43, 45, 46, 49–53] analyze the memory utilization of a program to identify potential parallelization opportunities using static [27, 49] and/or dynamic analysis [14, 19, 41], and profiling techniques. Their objective is orthogonal to CARMOT and its PSEC. Once potential parallel regions of a program are discovered, CARMOT can be used on those regions to understand exactly how they can be parallelized using the supported parallelism-related abstractions, verify the presence of actual parallelism, and improve it if possible.

Reference cycle discovery. Only two approaches are able to aid programmers in finding reference counting cycles at the source code level: Xcode [12] and Distefano *et al.* [22]. Xcode only works for swift and objective-c, but does not currently handle C++ smart pointers. Distefano *et al.* uses a static approach, which is limited by the accuracy of memory analysis that is known to be challenging for unmanaged languages. Neither of them can detect cycles formed in precompiled libraries.

7 Conclusion

Programmers are left alone to understand how to use modern programming language abstractions. We show that understanding how to use many of these abstractions needs the same fundamental knowledge: the PSEC of the ROI where the abstraction is applied. We formalize PSEC and describe a new open-source tool, CARMOT, capable to perform PSEC with reasonable overhead. We hope that CARMOT will help programmers to better understand their programs and to properly use the abstractions that are becoming increasingly prevalent and necessary in modern applications.

Acknowledgments

This project was supported by the U.S. National Science Foundation via awards CCF-2107042, CCF-1908488, CCF-2118708, CCF-2028851, CCF-2119069, CNS-1763743, by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by the U.S. Department of Energy Office of Science under Contract DE-AC02-06CH11357.

Data-Availability Statement

The artifact [20] to reproduce the results of this paper is available here: https://doi.org/10.5281/zenodo.7374843.

A Artifact Appendix

A.1 Abstract

This artifact is a podman image containing the CARMOT system and its dependencies, and it generates the main results of this paper in text format. All benchmark suites are included in the artifact, except for SPEC CPU 2017, which we cannot share directly (please refer to `README.md` in the artifact on how to include SPEC CPU 2017 results). This artifact requires podman (or docker) to load and run the image, and a network connection to download additional dependencies of the CARMOT system. The execution of this artifact requires an Intel multicore processor with shared memory.

A.2 Artifact check-list (meta-information)

- **Algorithm: No**
- **Program: NAS, PARSEC3**
- **Compilation: LLVM9.0.0., included**
- **Transformations: No**
- **Binary: No**
- **Data set: Included with the benchmark suites**
- **Run-time environment: No**
- **Hardware: No**
- **Run-time state: Yes**
- **Execution: Sole user**
- **Metrics: Execution time**
- **Output: Individual file output for each benchmark suite**
- **Experiments: The experiments can be run using the included `bin/carmot_experiments` script. Further information on how to customize the execution of the experiments can be found in the `README.md` of the running podman container.**
- **How much disk space required (approximately)?: 200GB**
- **How much time is needed to prepare workflow (approximately)?: Several hours**
- **How much time is needed to complete experiments (approximately)?: 4 days**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: MIT License**
- **Data licenses (if publicly available)?: No**
- **Workflow framework used?: Experiments are executed by the `bin/carmot_experiments` script included in the podman container. Refer to `README.md` in the podman container for additional customization of experiments.**
- **Archived (provide DOI)?: 10.5281/zenodo.7374843**

A.3 Description

A.3.1 How delivered

The artifact (i.e., podman image) can be downloaded through the provided DOI.

A.3.2 Hardware dependencies

Intel multicore processor with shared memory. To reproduce execution time results accurately frequency scaling mechanisms (e.g., TurboBoost) have to be disabled. A minimum of 125 GiB of main memory is required. The amount of disk space required by the fully unpacked artifact is approximately 200 GB.

A.3.3 Software dependencies

To run the artifact a Linux-based system with podman (or docker) installed is necessary.
The host machine `/proc/sys/kernel/yama/ptrace_scope` must be set to 0 to ensure proper collection of Pin-related results. The remaining dependencies are included in the podman image or installed when running the included scripts. The only exception is SPEC CPU 2017, which we cannot include in the artifact because of licensing. If reviewers would like to generate SPEC CPU 2017 results, they have to add the SPEC CPU 2017 benchmark suite manually themselves (refer to `README.md` inside the artifact to know how).

A.3.4 Data sets

The data sets are included in the podman image. The only exception is the SPEC CPU 2017 data set, which we cannot include in the artifact because of licensing. If reviewers would like to generate SPEC CPU 2017 results, they have to add the SPEC CPU 2017 benchmark suite manually themselves (refer to `README.md` inside the artifact to know how).

A.4 Installation

Download the artifact (i.e., podman image `carmot.tar`) following the provided DOI.
Load the image using:

```
podman load < carmot.tar
```

Run the image interactively using:

```
podman run --rm -it carmot /bin/bash
```

A.5 Experiment workflow

The following workflow is automatically executed when invoking the `bin/carmot_experiments` script in the running podman container.

1. The CARMOT system and all benchmark suites dependencies (except for SPEC CPU 2017) are downloaded.
2. CARMOT and its dependencies (noelle, virgil runtime) are compiled, the remaining dependencies (Intel tbb, boost, pin) are already included in the podman image.
3. All baselines for Figures 6, 7, 10, 11 are compiled and execution time data are generated.
4. Execution time data of Figure 6 are generated.
5. Speedup of Figure 6 are computed and placed under `results/current_machine`.
6. Execution time data of CARMOT for Figures 7, 10, 11 are generated.
7. CARMOT overhead of Figures 7, 10, 11 are computed and placed under `results/current_machine`.

Once the podman container is running interactively, please read `README.md`:

```
vim README.md
```

Then, run the experiments using the `bin/carmot_experiments` script, which can be executed in the background:

```
./bin/carmot_experiments &
```

Optionally, view the finer grain progress of the script:

```
tail -f carmot_experiments_output.txt
```

A.6 Evaluation and expected result

After loading and running the podman image, the entry point to generate the Minimal set of results of the paper is the script `bin/carmot_experiments`, which has to be invoked from the home directory of the running podman container as follows:
`./bin/carmot_experiments &`
This script takes no arguments and follows the previously described workflow. Generating the Minimal results takes approximately 2 days.
The generated results can be found in `results/current_machine`. The authors' results can be found in `results/authors_machine`. We expect the generated results of the artifact to be in line with the authors' results.

A.7 Experiment customization

The Full set of results of the paper can be generated by setting the environment variable `CARMOT_FULL` to 1:
`export CARMOT_FULL=1 ; ./bin/carmot_experiments &`
Generating the Full set of results takes approximately 4 days.
The number of runs of every data point can be customized by setting the environment variable `CARMOT_NUM_RUNS` prior to running `bin/carmot_experiments`:
`export CARMOT_NUM_RUNS=5 ; ./bin/carmot_experiments &`
The default is 3. Changing `CARMOT_NUM_RUNS` will affect the total amount of time to generate the results.
Additionally, SPEC CPU 2017 results can be generated following the instructions in the artifact `README.md`. Including the generation of SPEC CPU 2017 results will also increase the amount of time required to run the experiments.

A.8 Notes

More information can be found in the `README.md` of the artifact.

A.9 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

[1] [n. d.]. Clang Tidy. https://clang.llvm.org/extra/clang-tidy/. Accessed: 2023-01-15.

[2] [n. d.]. dmalloc. https://dmalloc.com/. Accessed: 2023-01-15.

[3] [n. d.]. Duma. https://www.linuxlinks.com/duma/. Accessed: 2023-01-15.

[4] [n. d.]. Electric Fence. https://linux.die.net/man/3/efence. Accessed: 2023-01-15.

[5] [n. d.]. Intel Inspector. https://software.intel.com/en-us/inspector. Accessed: 2023-01-15.

[6] [n. d.]. jemalloc. http://jemalloc.net/. Accessed: 2023-01-15.

[7] [n. d.]. MemWatch. https://www.linkdata.se/sourcecode/memwatch/. Accessed: 2023-01-15.

[8] [n. d.]. Mpatrol. http://mpatrol.sourceforge.net/. Accessed: 2023-01-15.

[9] [n. d.]. Mtrace. http://man7.org/linux/man-pages/man3/mtrace.3.html. Accessed: 2023-01-15.

[10] [n. d.]. Pinatrace. https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/. Accessed: 2023-01-15.

[11] [n. d.]. SPEC CPU 2017. https://www.spec.org/cpu2017. Accessed: 2023-01-15.

[12] [n. d.]. Xcode. https://developer.apple.com/xcode/. Accessed: 2023-01-15.

[13] Frances Allen, Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. 1988. A framework for determining useful parallelism. In *Proceedings of the 2nd international conference on Supercomputing*. https://doi.org/10.1145/55364.55385

[14] Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Greg Chan, Simone Campanoni, and David I. August. 2020. SCAF: A Speculation-Aware Collaborative Dependence Analysis Framework. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3385412.3386028

[15] Hamid Arabnejad, Joao Bispo, Jorge G. Barbosa, and Joao M.P. Cardoso. 2019. An OpenMP based parallelization compiler for C applications. *Proceedings - 16th IEEE International Symposium on Parallel and Distributed Processing with Applications, 17th IEEE International Conference on Ubiquitous Computing and Communications, 8th IEEE International Conference on Big Data and Cloud Computing, 11t* (2019). https://doi.org/10.1109/BDCloud.2018.00135

[16] David Bailey, E. Barszcz, Barton J.T, Browning D.S, Carter R.L, Dagum D, Fatoohi R.A, Paul Frederickson, Lasinski T.A, Robert Schreiber, Horst Simon, Venkat Venkatakrishnan, and Weeratunga K. 1991. The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications* (1991). https://doi.org/10.1177/109434209100500306

[17] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. https://doi.org/10.1145/1454115.1454128

[18] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. 2008. PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System. *PLDI 2008 - 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2008).

[19] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2015. HELIX-UP: Relaxing Program Semantics to Unleash Parallelization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. https://doi.org/10.1109/CGO.2015.7054203

[20] Enrico Armenio Deiana. 2022. *Program State Element Characterization*. https://doi.org/10.5281/zenodo.7374843

[21] Enrico A. Deiana, Vincent St-Amour, Peter A. Dinda, Nikos Hardavellas, and Simone Campanoni. 2018. Unconventional Parallelization of Nondeterministic Applications. In *ASPLOS*. https://doi.org/10.1145/3173162.3173181

[22] Dino Salvo Distefano, Cristiano Calcagno, and Dulma Churchill. 2019. Detecting and remedying memory leaks caused by object reference cycles. US Patent 10,296,314.

[23] D. Evans and D. Larochelle. 2002. Improving security using extensible lightweight static analysis. *IEEE Software* (2002). https://doi.org/10.1109/52.976940

[24] Saturnino Garcia, Donghwan Jeon, Christopher M Louie, and Michael Bedford Taylor. 2011. Kremlin: rethinking and rebooting gprof for the multicore age. *ACM SIGPLAN Notices* (2011). https://doi.org/10.1145/1993316.1993553

[25] Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. 2009. Profiling java programs for parallelism. In *2009 ICSE Workshop*

on Multicore Software Engineering. https://doi.org/10.1109/IWMSE.2009.5071383

[26] Yuxiong He, Charles E Leiserson, and William M Leiserson. 2010. The Cilkview scalability analyzer. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. https://doi.org/10.1145/1810479.1810509

[27] Nick P Johnson, Jordan Fix, Stephen R Beard, Taewook Oh, Thomas B Jablin, and David I August. 2017. A collaborative dependence analysis framework. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2017.7863736

[28] Richard Johnson, David Pearson, and Keshav Pingali. 1994. The Program Structure Tree: Computing Control Regions in Linear Time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/773473.178258

[29] Ken Kennedy, Kathryn S Mckinley, and Chau-Wen Tseng. 1991. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel & Distributed Systems* (1991). https://doi.org/10.1109/71.86108

[30] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. 2010. SD3: A scalable approach to dynamic data-dependence profiling. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. https://doi.org/10.1109/MICRO.2010.49

[31] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. 2009. How much parallelism is there in irregular applications? *ACM sigplan notices* (2009). https://doi.org/10.1145/1594835.1504181

[32] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: A Memory Profiler for NUMA Multicore Systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. https://doi.org/10.5555/2342821.2342826

[33] James R Larus. 1993. Loop-level parallelism in numeric and symbolic programs. *IEEE Transactions on Parallel and Distributed Systems* (1993). https://doi.org/10.1109/71.238302

[34] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* https://doi.org/10.1109/CGO.2004.1281665

[35] Zhen Li, Rohit Atre, Zia Huda, Ali Jannesari, and Felix Wolf. 2016. Unveiling parallelization opportunities in sequential programs. *Journal of Systems and Software* (2016). https://doi.org/10.1016/j.jss.2016.03.045

[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* (2005). https://doi.org/10.1145/1064978.1065034

[37] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, David I. August, and Simone Campanoni. 2022. NOELLE Offers Empowering LLVM Extensions. In *International Symposium on Code Generation and Optimization, 2022. CGO 2022.* https://doi.org/10.1109/CGO53902.2022.9741276

[38] C. McCurdy and J. Vetter. 2010. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. https://doi.org/10.1109/ISPASS.2010.5452060

[39] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. 2013. Parallelizing Sequential Programs with Statistical Accuracy Tests. *ACM Trans. Embed. Comput. Syst.* (2013). https://doi.org/10.1145/2465787.2465790

[40] Svetozar Miucin, Conor Brady, and Alexandra Fedorova. 2016. End-to-End Memory Behavior Profiling with DINAMITE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. https://doi.org/10.1145/2950290.2983941

[41] Niall Murphy, Timothy Jones, Robert Mullins, and Simone Campanoni. 2016. Performance Implications of Transient Loop-carried Data Dependences in Automatically Parallelized Loops. In *Proceedings of the 25th International Conference on Compiler Construction*. https://doi.org/10.1145/2892208.2892214

[42] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices* (2007). https://doi.org/10.1145/1273442.1250746

[43] Mohammad Norouzi, Felix Wolf, and Ali Jannesari. 2019. Automatic construct selection and variable classification in OpenMP. *Proceedings of the International Conference on Supercomputing* (2019). https://doi.org/10.1145/3330345.3330375

[44] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. 2010. Locating Cache Performance Bottlenecks Using Data Profiling. In *Proceedings of the 5th European Conference on Computer Systems*. https://doi.org/10.1145/1755913.1755947

[45] Aloor Raghesh. 2011. A Framework for Automatic OpenMP Code Generation. *M. Tech thesis, Indian Institute of Technology, Madras, India* (2011).

[46] Atanas Rountev, Kevin Van Valkenburgh, Dacong Yan, and P Sadayappan. 2010. Understanding parallelism-inhibiting dependences in sequential Java programs. In *2010 IEEE International Conference on Software Maintenance*. https://doi.org/10.1109/ICSM.2010.5609588

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A fast address sanity checker. *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012* (2012). https://doi.org/10.5555/2342821.2342849

[48] Scott D. Stoller, Michael Carbin, Sarita V. Adve, Kunal Agrawal, Guy E. Blelloch, Dan, Stanzione, Katherine A. Yelick, and Matei A. Zaharia. 2019. Future Directions for Parallel and Distributed Computing: SPX 2019 Workshop Report. In *NSF Workshop Reports*.

[49] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. https://doi.org/10.1145/2892208.2892235

[50] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. 2008. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. https://doi.org/10.1145/1345206.1345234

[51] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael F. P. O'boyle. 2014. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Transactions on Architecture and Code Optimization* (2014). https://doi.org/10.1145/2579561

[52] Peng Wu, Arun Kejariwal, and Călin Caşcaval. 2008. Compiler-driven dependence profiling to guide program parallelization. In *International Workshop on Languages and Compilers for Parallel Computing*. https://doi.org/10.1007/978-3-540-89740-8_16

[53] Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. 2009. Alchemist: A transparent dependence distance profiling infrastructure. In *2009 International Symposium on Code Generation and Optimization*. https://doi.org/10.1109/CGO.2009.15

Received 2022-09-02; accepted 2022-11-07