Preprints.org

Article

Not peer-reviewed version

CATS: A Tool for Semantically-Aware Cache Performance Analysis of C Programs

Vitaly Egunov *, Alla G. Kravets, Pavel Kravchenya, Anna Matokhina, Vladimir Shabalovsky

Posted Date: 21 May 2026

doi: 10.20944/preprints202605.1387.v1

Keywords: cache simulation; memory trace analysis; performance optimization; artificial intelligence; machine learning; dynamic tracing; static code analysis; semantic annotation; high-performance computing

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC, OpenAlex.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

CATS: A Tool for Semantically-Aware Cache Performance Analysis of C Programs

Vitaly Egunov [1,*], Alla G.Kravets [2,3], Pavel Kravchenya [4], Anna Matokhina [5] and Vladimir Shabalovsky [6]

[1] Computers and Systems Department, Volgograd State Technical University, Volgograd 400005, Russia
[2] CAD&RD Department, Volgograd State Technical University, Volgograd 400005, Russia
[3] Institute of System Analysis and Management, Dubna State University, Russian Federation, Moscow Region, Dubna 141982, Russia
[4] Computers and Systems Department, Volgograd State Technical University, Volgograd 400005, Russia
[5] CAD&RD Department, Volgograd State Technical University, Volgograd 400005, Russia
[6] CAD&RD Department, Volgograd State Technical University, Volgograd 400005, Russia
* Correspondence: vegunov@mail.ru

Abstract

The rapid development of Artificial Intelligence (AI) and Machine Learning (ML) poses new challenges for high-performance system developers. The performance of such systems is often limited not by computational power, but by the efficiency of memory subsystem interaction. Cache behavior optimization becomes critically important, yet existing analysis tools fail to meet the demands of modern AI applications. They either provide only aggregated statistics or are characterized by a "semantic gap", presenting data in machine addresses rather than source code, which makes them ill-suited for analyzing the complex software systems typical of AI. This paper introduces CATS (C Annotated Trace-based Cache Simulator), a novel hybrid method and toolset for detailed cache efficiency analysis, designed to overcome these limitations. CATS combines dynamic tracing with static source code analysis to generate semantically annotated memory traces. This approach is particularly relevant for optimizing AI applications, as it allows precise identification of which data structures (e.g., weight matrices, tensors, or input vectors) are causing cache misses. For analyzing long-running tasks, such as training AI models, our method leverages AI techniques, specifically ML, for intelligent trace sampling, significantly reducing analysis time without sacrificing representativeness. The paper describes the methodology and architecture of CATS and presents experimental evaluation results. In the long term, the data collected by CATS can be used to train AI models capable of automatically providing developers with code refactoring recommendations to improve performance. Early CATS application identifies and resolves cache issues before final implementation, cutting performance optimization costs

Keywords: cache simulation; memory trace analysis; performance optimization; artificial intelligence; machine learning; dynamic tracing; static code analysis; semantic annotation; high-performance computing

---

1. Introduction

The performance of modern computing systems is increasingly constrained not by the computational speed of processors, but by the latency of data access from main memory. This ever-widening gap between CPU and memory performance, widely known as the "memory wall" [1, 2], has made the efficient use of the memory hierarchy a critical factor in software performance. The multi-level cache system serves as the primary mechanism to mitigate this problem by storing frequently used data closer to the processor [3]. Consequently, for a wide spectrum of performance-

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

demanding applications, from scientific computing and database management [4] to machine learning [5], the ability to write cache-efficient code is paramount for achieving high performance.

This is particularly relevant for modern Artificial Intelligence (AI) systems. Despite the widespread use of high-level languages (such as Python) at the interface level, critical components, such as numerical library kernels, deep learning frameworks, and inference engines, are implemented in C/C++ to achieve maximum performance and control over memory usage. A Machine Learning Engineer (ML Engineer) developing high-performance industrial AI solutions, including for edge computing (Edge AI), must be proficient in applying C/C++ to implement efficient multi-threaded solutions, to adapt AI systems for specific hardware platforms with computational constraints (embedded and cyber-physical systems), and to develop solutions using GPUs and FPGAs for massive parallelization. In all these scenarios, the efficiency of the C/C++ code interaction with the memory subsystem and cache hierarchy directly determines the achievable performance and energy efficiency of the AI systems.

To write such code, developers require sophisticated tools capable of analyzing the interaction between the program and the cache hierarchy. The authors of this work have previously proposed a number of analytical methodologies aimed at increasing the efficiency of automatic vectorization [6], optimizing parallel linear algebra algorithms [7, 8], and reducing energy consumption through software-based cache management [9]. Nevertheless, the effectiveness of all these approaches fundamentally depends on how well the program interacts with the memory hierarchy, and existing tools for analyzing this interaction have fundamental limitations. Hardware profilers (e.g., `perf`, Intel VTune) offer low overhead by leveraging processor performance counters; however, they provide only aggregated statistics, identifying "hotspots" in the code without revealing their root cause or the specific data structures responsible for poor performance. On the other end of the spectrum are Dynamic Binary Instrumentation (DBI) tools, such as Valgrind, which provide high granularity by simulating every memory access. However, such accuracy is achieved at the cost of high overhead, significantly slowing down program execution. More importantly, these tools exhibit the "semantic gap" problem: by operating at the machine code level, they lose the connection to high-level source code constructs. A developer sees a cache miss at a specific memory address but has no direct way of knowing which array, variable, or structure field this address corresponds to. Finally, static analysis tools, while avoiding runtime overhead, are by nature conservative, oriented towards Worst-Case Execution Time (WCET) estimation, and are incapable of analyzing typical cache behavior for specific program inputs.

Consequently, there is a significant gap in the spectrum of existing performance analysis tools. No existing approach provides developers with a mechanism that is simultaneously accurate (based on real execution), detailed (tracking individual memory accesses), and, most crucially, interpretable in terms of source code semantics. This applies equally to traditional high-performance computing and modern AI systems, where an AI system developer using C/C++ needs to understand exactly which data structures (tensors, weight matrices, feature buffers, etc.) are leading to cache efficiency degradation.

To overcome these limitations, in this paper we present CATS (C Annotated Trace-based Cache Simulator), a cache simulator for C programs based on annotated traces. CATS implements a novel hybrid method that bridges the semantic gap by combining dynamic analysis for collecting accurate execution traces with static analysis for enriching these traces with high-level semantic information. The key artifact produced by CATS is a semantically annotated memory trace, where each memory access record contains not only the address but also a direct reference to the corresponding C source code construct (a global variable, a stack-allocated array, or a specific field in a heap-allocated structure). This approach ensures a level of detailed, source-code-oriented cache performance analysis that was previously unattainable, providing developers with actionable information for targeted code optimization, including within complex AI systems implemented in C/C++ and targeting specialized, resource-constrained hardware platforms. The main contributions of this work are as follows:

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

- Development of a hybrid cache performance analysis methodology that systematically links low-level architectural events (cache misses) to high-level source code constructs (variables, arrays, and data structures).
- Design and implementation of CATS, a toolset that implements this methodology for C programs and is capable of generating and analyzing semantically annotated traces.
- Experimental evaluation of CATS, demonstrating its ability to identify memory performance problems that are difficult or impossible to diagnose with existing tools, thereby supporting developers of high-performance C/C++ components.

The rest of the paper is organized as follows. Section 2 reviews related work in program analysis and cache simulation. Section 3 describes the methodology and architecture of CATS. Section 4 presents the experimental setup and evaluation results. Finally, Section 5 concludes the paper and outlines directions for future research.

2. Literature Review

To develop an effective methodology for analyzing program cache behavior, it is necessary to conduct a systematic review of existing approaches and tools. This section examines a wide range of performance analysis methods, with a particular emphasis on their capabilities and limitations in the context of studying program interaction with the memory subsystem.

The entire diversity of approaches can be broadly divided into two major categories: methods oriented towards program code analysis and methods focused on architectural simulation. The first category includes classical static, dynamic, and hybrid analyses. The second category comprises the use of specialized and full-system simulators to investigate program behavior on a detailed hardware model. The general classification of these methods and tools discussed in this chapter is presented in Figure 1.

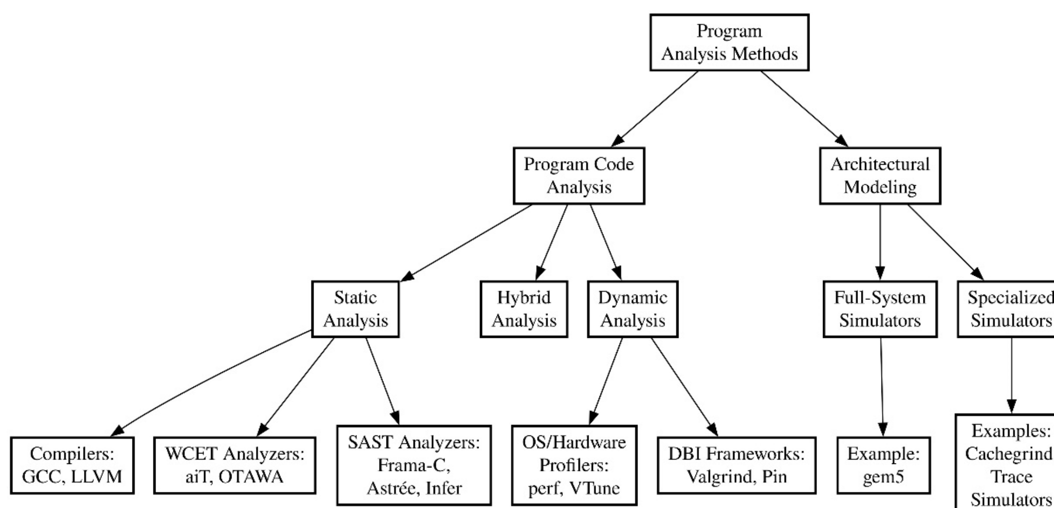

Figure 1. General classification of methods and tools for analyzing program performance and behavior in the memory subsystem.

2.1. Program Analysis Methods

There are various approaches to analyzing software systems, traditionally classified based on when the analysis occurs: before the program execution starts or during its operation. This division forms three fundamental classes of methods: static, dynamic, and hybrid.

Static analysis examines the source or binary code of a program without actually running it, aiming to establish properties that hold true under any usage scenario. Classical techniques, such as data-flow analysis [10], abstract interpretation [11], and pointer analysis [12], compute approximate, conservative program characteristics. The main limitation of this approach lies in the loss of precision:

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

to ensure the correctness of conclusions, static analysis is forced to overestimate the set of possible behaviors, which is particularly noticeable when dealing with complex C language constructs. In the context of cache analysis, this allows for calculating safe upper bounds, for example, in WCET analysis tasks [13, 14], but does not provide an opportunity to obtain an accurate performance profile for specific input data.

Dynamic analysis, in contrast, collects information about program behavior directly during its execution, observing a specific execution trajectory for given input data [15]. This approach is used for performance profiling [16], memory error detection [17, 18], and data access pattern analysis [19]. Technically, it is implemented through DBI using frameworks such as Valgrind or Intel PIN [20], or through instrumentation at the source code or Intermediate Representation (IR) level. The key advantage of dynamic analysis is its high accuracy for a specific run. However, this approach is associated with two significant drawbacks: substantial overhead, slowing down the program by orders of magnitude, and the "semantic gap"—the loss of connection between low-level events (memory addresses) and high-level source code constructs.

Hybrid methods aim to combine the advantages of both approaches. Typically, in the first, static stage, the program is analyzed or transformed to prepare for observation, and in the second, dynamic stage, data collection and subsequent analysis take place. A classic example is Profile-Guided Optimization (PGO) [16]. A similar scheme is applied in data locality analysis systems [21] and frameworks for analyzing parallel programs, such as TAU or Score-P [22, 23], where static instrumentation is used to control dynamic trace collection.

A distinct class of hybrid approaches is based on the interpretation or deep transformation of the IR. Tools such as CIL [24], AddressSanitizer [25], or the E-ACSL plugin for Frama-C [26] statically modify the IR or source code to embed checks that are executed dynamically. Although the KLEE symbolic execution system [27] also interprets IR, its primary goal is functional correctness rather than performance analysis. Despite hybrid methods being a promising direction, existing implementations are often either focused on tasks unrelated to cache behavior or still fail to fully resolve the "semantic gap" problem, which serves as the primary motivation for this work.

2.2. Architectural and Cache Simulators: Capabilities and Limitation

Full-system architectural simulators include tools such as gem5 [28], SimpleScalar [29], MARSSx86 [30], and a number of extended QEMU modifications [31] that supplement functional modeling with a detailed description of the microarchitecture. They model not only the processor core but also caches of all levels, interconnects and buses, DRAM controllers and modules, and other elements of the I/O subsystem. In full-system mode, such simulators reproduce the operation of the complete hardware-software environment—from OS and driver booting to the execution of user applications and system services [28–30].

Due to this high degree of detail, these tools are widely used in architectural research and in the design of microprocessors and memory subsystems. They allow for the study of new coherence protocols, replacement and associativity policies, prefetching algorithms, cache hierarchy and buffer structures, options for organizing inter-processor communication and buses, as well as new types of main memory. The ability to run unmodified operating systems and real applications makes such experiments representative of target workloads [28–30]. However, this comes at the cost of configuration complexity and an extremely high "cost" of simulation: execution speed is orders of magnitude lower than real hardware, which limits the use of full-system simulators to targeted analysis scenarios.

A lighter class of tools consists of specialized cache simulators and trace-driven approaches. Tools like DineroIV [32] and cache simulators based on dynamic binary instrumentation (e.g., Cachegrind within Valgrind) [17] model only the behavior of the cache hierarchy using pre-prepared memory access traces. Such traces, representing logs of events in the form of "address–operation type–data size" (sometimes specifying the thread, timestamp, and instruction address), are generated using DBI frameworks (Valgrind, Pin, etc.) or hardware-dependent profiling tools [17, 32]. They are

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

then replayed multiple times on various simulator configurations to investigate the impact of sizes, associativity, replacement policies, and prefetching on throughput and miss rates [33]. Because only the memory subsystem is simulated, these tools are simpler, faster, and particularly in demand at the early stages of architectural design.

A common limitation of both full-system and trace-driven simulators is their operation at the machine code level. They receive streams of instructions and memory accesses represented as addresses and operation codes (sometimes with minimal metadata: thread identifiers, program counters) as input [17, 20]. High-level source code entities, such as variable and array names, loop and function boundaries, and abstract data structures, are absent from their models. The simulator operates with registers, the stack, the heap, and sets of virtual/physical addresses [34]. Attempts to restore the connection to the source code through debug information (DWARF, etc.) [35] or post-processing of simulation results (Pin, Valgrind, HPCToolkit) [17, 20, 36] encounter aggressive compiler optimizations (loop unrolling and fusion, inlining, variable elimination, and register-only storage) [34]. In production binary files, debug information is often missing or incomplete [35, 36], making the mapping of addresses to variables and source text constructs laborious and inaccurate.

As a result, existing simulators are well-suited for measuring aggregated memory subsystem characteristics (miss counts, latencies, etc.) but are ill-suited for tasks that require linking these characteristics to specific arrays, data structures, and code sections in the C language.

2.3. Summary and Motivation for Trace-Based Analysis

A comparison of the approaches to performance and memory subsystem behavior analysis reviewed in the literature highlights a number of fundamental limitations of existing methods. Each of them solves its specific task, but none provides a comprehensive solution that would be simultaneously accurate, fast, and convenient for a developer interested in interpreting results in terms of the original C code. The key trade-offs lie in three dimensions: data accuracy and granularity, performance (overhead), and the preservation of the semantic connection to the source code.

On one hand, static methods and lightweight profilers ensure high performance and maintain a link to the code, but at the cost of low accuracy and detail. On the other hand, DBI tools and architectural simulators offer maximum accuracy, but their practical application is limited by colossal overhead and the "semantic gap," which consists in the loss of information about the program's high-level data structures.

Taken together, these limitations show that none of the "pure" approaches fully solves the problem of providing the developer with an accurate and simultaneously interpretable (in terms of the original C code) picture of cache subsystem behavior for specific program runs. This problem creates a need for methods capable of bridging this gap.

This is where the concept of trace-oriented performance analysis begins to play a key role, but with an important clarification. It is not enough to simply collect a memory access trace, as low-level tools do. Meaningful analysis requires annotating traces with source-code-level information (which variables, arrays, and structure fields were affected by each memory access). Such an approach, implemented within the framework of hybrid and partially dynamic methods, strives to combine the best of two types of analysis: the accuracy of dynamic analysis (since data is collected from real execution) and the semantic completeness of static analysis (since the source code context is preserved). This allows not just stating the fact of a cache miss, but answering the question: "A miss on which specific element of which array occurred and why?", which forms the basis for targeted code optimization by the developer.

The limitations listed above directly motivate the approach proposed in this paper, based on the use of annotated traces, which, already at the generation stage, combine low-level information (addresses, operation types, threads) with high-level annotations about the program structure (variable names and types, loop and function boundaries, belonging of accesses to arrays, etc.). This opens up the possibility of analyzing memory subsystem behavior directly in terms of the source code.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

3. Materials and Methods

3.1. Methodology

The central task of this research is the development and application of a methodology for the comprehensive evaluation of the cache efficiency of programs written in C. The fundamental metric for such an evaluation is the number of cache misses, which must be analyzed across cache hierarchy levels and program sections to obtain a complete picture of the program's interaction with the memory subsystem.

At the baseline level of evaluation, it is necessary to quantitatively measure the number of cache misses, categorizing them as follows:

- By cache level: separately for each level of the hierarchy.
- By access type: separately for read misses and write misses.

However, despite their quantitative accuracy, such aggregated assessments have limited diagnostic value for the developer. A report indicating a high total number of misses in the L2 cache describes a symptom (low performance) but does not reveal its cause. It does not provide information about specific flawed design decisions that need improvement and does not allow for effective program optimization. Faced with such information, a developer does not receive an answer to the main question: "What exactly in the code needs to be fixed?"

To overcome this limitation and provide the developer with actionable insights, our methodology requires a deeper level of granularity that links low-level architectural events (cache misses) to high-level semantic source code constructs. To achieve this, it is necessary to associate each cache miss with its source, obtaining information with detail according to the following parameters:

1. Association with data structures: It is necessary to obtain information about cache misses separately for each array, global, or stack variable. This allows identifying exactly which data structures in the program are organized inefficiently in terms of locality or cause conflicts in the cache. Such analysis directly points to the used memory access patterns and their effectiveness.
2. Association with code sections: Information about cache misses localized within individual functions or even specific code blocks (e.g., critical loops) is highly valuable. This helps determine which algorithmic parts of the program contribute most to inefficient memory usage.
3. Preservation of architectural detail: At the same time, for each of the above attributes (by data and by code), the baseline granularity, by cache level and access type (read/write), must be preserved.

Thus, the proposed methodology allows moving from a general and uninformative conclusion like "the program has 10 million L2 cache misses" to a specific and actionable conclusion, for example: "80% of read misses in the L2 cache occur in the matrix_multiply function when accessing columns of array B, which indicates a suboptimal access pattern (column-major traversal) and requires reorganizing the algorithm or data." This level of detail is a necessary condition for the targeted and effective optimization of program cache behavior.

3.2. Problem Statement

To develop targeted software optimization strategies, a methodology is required that allows for the quantitative assessment of the efficiency of a program's interaction with the memory subsystem. The fundamental metric for such an assessment is the number of cache misses.

Formally, the behavior of a program in the memory subsystem can be described by a memory access trace—an ordered sequence of events.

$$T = \langle e_1, e_2, \dots, e_n \rangle, \tag{1}$$

where each event e_i represents at least a tuple $(op_i, addr_i)$, consisting of the operation type $op_i \in \{Read, Write\}$ and the virtual address $addr_i$.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

Cache efficiency analysis boils down to applying a simulator function Sim to the trace T for a given cache configuration C:

$$M = Sim(T, C). \quad (2)$$

The result M is a set of aggregated cache miss counters, typically with a breakdown by cache level l and operation type op:.

$$M = \{M_{l,op}\}. \quad (3)$$

Despite its quantitative accuracy, this formulation has minimal diagnostic value for the developer. The key limitation lies in the definition of the trace T, which contains only low-level information (addresses) and lacks any connection to high-level semantic constructs of the source code. Upon receiving report M, a developer cannot answer the main questions: "Which specific data structures are causing these misses?" and "Which access patterns to them need to be optimized?"

To overcome this “semantic gap” and provide the developer with actionable information, it is necessary to extend the trace definition by enriching it with semantic context. Let us introduce the concept of a semantically annotated trace T_a:

$$T_a = \langle e_{a,1}, e_{a,2}, \ldots, e_{a,n} \rangle. \quad (4)$$

where each annotated event $e_{a,i}$ represents a tuple $(op_i, addr_i, S_i)$.. Here S_i is a semantic descriptor that identifies the source code object associated with the access at address $addr_i$. At a minimum, S_i must contain the identifier of a variable or data structure (e.g., an array name).

Accordingly, the analysis task becomes more complex. The new simulator function Sim_a must not simply count the total number of misses but calculate their distribution across semantic objects. The result of its operation is a mapping M_a, which assigns to each unique semantic descriptor s its own set of miss counters:

$$M_a(s) = \{M_{s,l,op}\}. \quad (5)$$

where $M_{s,l,op}$ is the number of misses for object s at cache level l during an operation of type op. Such granularity allows transitioning from a general conclusion "there are many misses in the program" to a specific diagnostic conclusion.

Thus, the central problem addressed in this paper is formulated as follows.

Development of a hybrid method and a toolset implementing it, capable of automatically generating a semantically annotated memory access trace T_a for a given C program and input data, and, based on it, computing a detailed distribution of cache misses $M_a(s)$ for each semantically significant source code object s.

To solve the posed problem, we have developed a hybrid method and a toolset implementing it, named C Annotated Trace-based cache Simulator (CATS). In the following section, we will examine its methodology and architecture in detail.

3.3. The CATS

The proposed method and the toolset implementing it, in accordance with the formulated requirements, must automatically obtain a semantically annotated memory access trace T_a: for a given C program and a specific set of input data, and, based on it, compute the distribution of cache misses $M_a(s)$ across semantically significant source code objects s. The CATS system implements this requirement as a two-stage pipeline:

1. Generation of a semantically annotated trace at a representation level close to the original C source code.
2. Subsequent trace-driven simulation of the cache memory hierarchy using this trace.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

Unlike traditional binary profilers and architectural simulators, CATS applies partially dynamic analysis before the stage of aggressive compiler optimizations. The CATS algorithm can be represented as follows (Figure 2).

The original C program is parsed into an Abstract Syntax Tree (AST), based on which a simplified IR is built, similar to CIL, Frama-C, or LLVM IR: complex expressions are normalized, loops are reduced to explicit branches and jumps, and expressions may be converted into Reverse Polish Notation (RPN), etc. This IR preserves the natural structure of the source code (functions, loops, arrays, structures), which is critical for subsequent semantic annotation.

An interpreter is implemented on top of the obtained IR, which executes the program step-by-step and intercepts all operations of interest, primarily memory accesses. At this stage, a semantically annotated trace T_a is formed, which is then passed to a separate cache simulation module.

Currently, CATS is implemented as a console application.

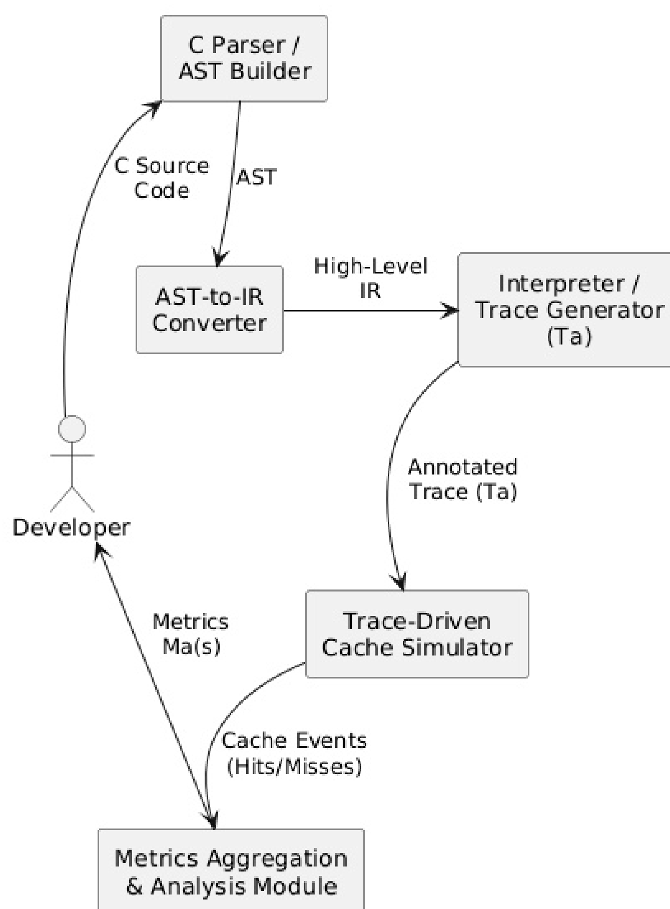

Figure 2. CATS General Algorithm.

3.3.1. Generation of Semantically Annotated Trace

Each event in the CATS trace corresponds to a single memory access and is represented by a tuple.

$$e_a = (addr, op, size, P). \quad (6)$$

where:

- $addr$ is the memory access address;
- $op \in \{Read, Write\}$ is the operation type;
- size is the size of the transferred data;
- P is a set of annotations carrying high-level semantic information.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

The set of parameters P includes, in particular:

- The name and type of the variable or array to which the access refers.
- When accessing structures, the field name and container type.
- The function name and (if necessary) call context.
- The position in the source file (file, line).
- Belonging to a specific loop iteration or logical block of the program at the C level.

Thus, the trace (4) is not a "flat" address log, as in classical trace-driven cache simulators [32, 33], but represents a semantically rich description of execution, directly linked to the source code. This directly implements the problem statement's requirement for a semantically annotated trace.

3.3.2. Trace-Driven Simulation

At the second stage, the annotated trace T_a is passed to the hierarchical memory simulation module, which implements the classical trace-driven approach. This module:

- Models the specified configuration of the cache subsystem (number of levels, sizes, associativity, replacement policies, prefetching, etc.).
- Processes the sequence of low-level events (6), calculating misses, latencies, and other metrics.

The key difference from traditional cache simulators is that when aggregating results, the CATS simulator uses the semantic annotations P. For each semantic object s (array, global or local variable, structure field, code section), metrics of the form (5) are generated.

This allows results to be presented not only in traditional architectural terms (L1/L2/LLC miss rate) but also in terms of the original data structures and functions, which is directly useful to the developer.

Separating the stages into "semantically rich" trace generation and subsequent inexpensive, repeatedly replayable cache simulation provides two key advantages:

- The accuracy of dynamic analysis is preserved (real execution paths and specific input data are taken into account).
- It enables the repeated investigation of various cache subsystem configurations without re-running and re-interpreting the program.

3.3.3. Assumptions and Scope of Applicability

To reduce overhead while maintaining the reliability of target metrics, CATS employs two practical assumptions.

Assumption 1 (subset of observed variables). *In the problem statement section, a semantically annotated memory access trace (4) and the target form of the analysis result as a mapping (5) were introduced. Within the framework of this assumption, the set of all objects that can principally be sources of memory accesses is introduced,*

$$S = \{s\}. \tag{7}$$

and an observed subset $S_{obs} \subseteq S$, for example

$$S_{obs} = S_{arr} \cup S_{struc(arr)}. \tag{8}$$

i.e., arrays and associated structure fields. For objects $s \in S_{obs}$ the metrics $M_a(s)$ are computed and aggregated fully, whereas accesses to objects $S_{ign} = S \setminus S_{obs}$ (in particular, to "purely scalar" variables that do not affect addressing and control structure: loop boundaries, branch conditions) may be omitted at the trace generation stage or excluded from subsequent aggregation.

It is thereby assumed that the contribution of ignored objects to the target metrics is small compared to the contribution of observed objects; ignored objects contribute no more than a fraction

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

ε of the total misses. Formally, for target analysis configurations, the fulfillment of a relation of the following form is expected

$$\frac{\sum_{s \in S_{ign}} M_{s,l,op}}{\sum_{s \in S} M_{s,l,op}} \leq \varepsilon, 0 \leq \varepsilon \ll 1, \quad (9)$$

which allows for a significant reduction in interpretation cost and trace volume without a noticeable distortion of the cache miss distribution across main data structures.

Within this assumption, it is permissible to use one of the following levels of detail:

- Account only for accesses to array elements.
- Account for accesses to array elements and a selected subset of scalar variables.
- Account for accesses to all program variables.

In the base configuration of CATS, the first option is applied; experimental results show that it provides acceptable accuracy with a significant reduction in simulation time.

Assumption 2 (filtering of non-essential operations). *In the problem statement, cache behavior metrics (5) are calculated based on the memory access trace (4) and the cache subsystem simulator*

$$M_a(s) = Sim_a(T_a, C). \quad (10)$$

Here, trace (4) contains memory access events and semantic annotations (6).

Let E^{full} denote the full execution of the program (the sequence of all elementary events, including computational operations and memory accesses), and let $\pi()$ be the operation of extracting the annotated memory access trace

$$T_a^{full} = \pi_a(E^{full}). \quad (11)$$

Let us introduce the set of essential operations O_{ess}, which must be modeled accurately because they can change the trace (4):

- All memory access operations and address arithmetic.
- Operations affecting control decisions (branch conditions), loop boundaries, and counters.
- Operations affecting the calculation of array indices (pointers).

Other computational operations, particularly arithmetic-logical transformations of array and structure element values that do not participate in address calculation and branch/loop conditions, belong to the set of non-essential operations O_{ign} and may not be modeled in detail.

Let E^{red} be the reduced execution obtained from E^{full} by removing operations from O_{ign} while maintaining accurate modeling of O_{ess}. The content of the assumption consists in the following

$$\pi_a(E^{red}) = \pi_a(E^{full}). \quad (12)$$

That is, the reduced simulation preserves the same annotated memory access trace, and consequently, for a fixed cache configuration C:

$$Sim_a(\pi_a(E^{red}), C) = Sim_a(\pi_a(E^{full}), C) \quad (13)$$

and the target metrics $M_a(s)$ coincide.

Within the framework of this assumption, the following simplifications are adopted:

- All memory access operations are always taken into account; it is they that form the trace supplied to the input of the cache simulator.
- Arithmetic and logical operations without side effects are not included in the trace and can be interpreted in a simplified manner if they:
  - do not modify data in memory;
  - do not participate in calculating memory access addresses;
  - do not affect branch conditions, loop counters, and other constructs determining the order of memory accesses;

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

- In particular, this applies to operations on scalar variables that are not used in array indices, address arithmetic, and control expressions.

Collectively, the described solutions ensure the realization of the goal set in the Problem Statement section: the construction of semantically annotated traces and the calculation of cache metrics in terms of the original C code with acceptable overhead, making CATS a practically applicable tool for developers and researchers.

Within the specified assumptions, the scope of applicability of CATS is limited to the analysis of the cache behavior of C programs executed on architectures with a classical processor cache hierarchy. The method is oriented towards CPU-initiated memory accesses and is primarily suitable for computationally intensive applications dominated by operations on arrays and other structured data. Direct accounting for interaction with I/O devices, DMA transfers, specialized accelerators (GPUs, etc.), as well as distributed memory systems, is not performed in the current version; such effects can only be considered indirectly through their influence on the sequence of memory accesses on the central processor side.

4. Results and Discussions

Computational experiments were conducted using a "naive" matrix multiplication implementation, specifically the standard algorithm employing three nested loops. In addition to matrix multiplication itself, the analyzed code also contains the initialization of arrays. To investigate software implementations of matrix multiplication algorithms, the following tools were used:

- The architectural simulator gem5 for simulating program execution on the target memory hierarchy.
- The Cachegrind tool from the Valgrind suite for dynamic analysis of cache memory efficiency.

All programs were run on the Ubuntu 24.04.1 LTS operating system, deployed within the Windows Subsystem for Linux 2 (WSL2) under Windows 10. The code was compiled with GCC 13.3.0. Experiments were conducted on a computing system based on the 11th Gen Intel(R) Core(TM) i9-11900 @ 2.50GHz microprocessor.

During the analysis in both cases, the following cache subsystem configuration was used:

- Number of levels: 2.
- Hierarchy type: Inclusive (a line present in the L1 cache is considered to be simultaneously present in the next-level L2 cache).
- Replacement algorithm: LRU (Least Recently Used), i.e., when space needs to be freed, the line that has not been used for the longest time is evicted from the cache.
- Write policy: Write-back, where modified data is first accumulated in the cache and written to external memory only upon line eviction.
- Write miss policy: Write-allocate, where on a write miss, the corresponding line is first loaded into the cache, and only then is the write operation performed in the cache (rather than directly to main memory).

Table 1. Characteristics of simulated cache levels.

Level	Size, KB	Associativity
L1	32	8
L2	256	8

For both levels, the cache line size is 64 bytes.

4.1. CATS vs gem5

The simulation in gem5 for the program under study was launched using the command:

```
gem5/build/ALL/gem5.opt gem5/configs/deprecated/example/se.py \
  --cmd=./mm \
```

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1


```
--cpu-type=X86O3CPU \
--caches --l2cache \
--l1d_size=32kB --l1d_assoc=8 \
--l1i_size=32kB --l1i_assoc=8 \
--l2_size=256kB --l2_assoc=8 \
--mem-type=DDR3_1600_8 × 8 --mem-size=1GB
```

Executing the command initiates simulation in the gem5 simulator in System Call Emulation (SE) mode with the x86 architecture and the following configuration of the processor, cache subsystem, and RAM:

- One compute core with the X86O3CPU model (x86 architecture from the gem5 model set, out-of-order superscalar core).
- Separate Level 1 (L1) caches for instructions and data with a size of 32 KB and 8-way associativity (the default line size is 64 bytes for se.py).
- Unified Level 2 (L2) cache with a size of 256 KB and 8-way associativity.
- Main memory of type DDR3-1600 with a size of 1 GB.

To conduct the simulation, the standard configuration script se.py was used, which is included in the gem5 distribution and designed for SE mode. The se.py script creates a model of a single-processor x86 architecture system and allows specifying the processor core type, cache subsystem parameters, and RAM from the command line, facilitating the execution of serial experiments with various computing system configurations.

Matrix multiplication simulation was performed considering various data placement options: automatic arrays (on the stack), static arrays, and dynamically allocated arrays (on the heap), as illustrated in Figure 3.

```
int matrix_mul()
{
  int a [10000];
  int b [10000];
  int c [10000];
```

a)

```
int a [10000];
int b [10000];
int c [10000];
int matrix_mul()
{
```

b)

```
int matrix_mul()
{
  int *a = (int*)malloc(10000 * sizeof(int));
  int *b = (int*)malloc(10000 * sizeof(int));
  int *c = (int*)malloc(10000 * sizeof(int));
```

c)

Figure 3. Data placement options: automatic arrays a), static arrays b), dynamic arrays c).

The gem5 simulator allows estimating only the total number of cache misses by category, but it does so quite accurately for the chosen target architecture, as during program execution gem5 step-by-step emulates each memory access, taking into account the specified cache subsystem configuration and eviction algorithm, without analytical assumptions and simplifications.

During the simulation process, data on read and write misses at two cache levels were collected for all three data placement options (automatic, static, and dynamic arrays). For each parameter, the difference between the values obtained by CATS was evaluated in accordance with the following expression:

$$diff = \frac{abs(V_{cats} - V_{gem5})}{V_{gem5}}, \tag{14}$$

where:

- V_{cats} is the parameter value obtained by CATS;
- V_{gem5} is the parameter value obtained by gem5.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

It can be seen (Figure 4) that at small dimensions (100) for the auto and static variants, the relative deviation of CATS estimates from gem5 results is about 5–7%, and at dimension 400, the difference increases to ≈10%; at other dimensions, the discrepancy does not exceed 2%. The increased deviation at N=100 is explained by the higher "noise" of measurements: due to the relatively small number of computational operations, the relative contribution of service memory accesses (stack, library code, etc.), which are not modeled in CATS but are accounted for in the gem5 simulator, increases significantly.

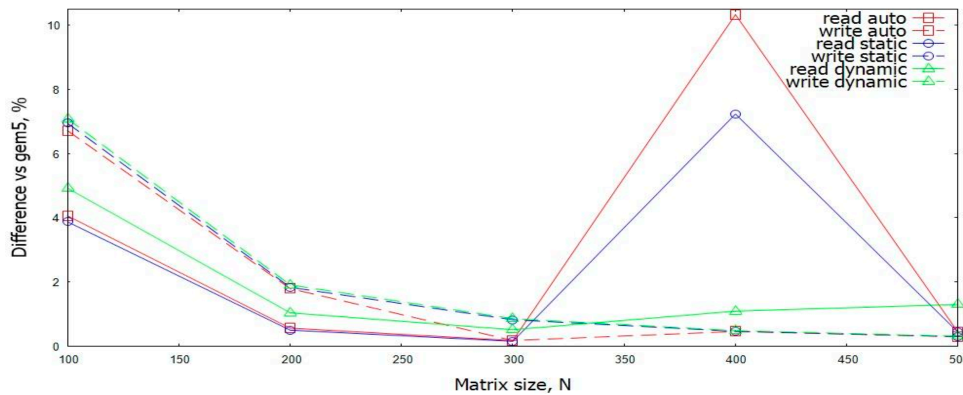

Figure 4. Difference CATS vs gem5 in the number of L1 cache misses.

At dimension 400, the CATS analytical model predicts significantly more L1 misses for arrays than measured in gem5. This is due to the fact that at these dimensions, the matrix row length (in cache lines) and the cache geometry (number of sets and associativity) form an unfortunate combination from the model's point of view, where cache lines of different matrix rows are formally projected into a limited subset of sets and are considered frequently conflicting. The actual sequence of memory accesses modeled by gem5 (taking into account real array base addresses, alignment, and additional accesses to the stack and other data) leads to fewer conflicts than the analytical scheme assumes. Thus, at these dimensions, the model provides an overestimated but conservative estimate of the number of misses.

It can be seen that for dynamic arrays, with the exception of small dimensions (100), the deviation does not exceed 1%. Such a deviation is insignificant, so it can be considered that for dynamic arrays, both systems give coinciding estimates. In scientific and engineering applications, dynamic arrays are used in the overwhelming majority of cases, for which the discrepancy between CATS and gem5 estimates does not exceed 1%. Therefore, taking into account the gem5 results, it can be considered that the CATS system provides reliable estimates of the number of cache misses for practically significant data placement options.

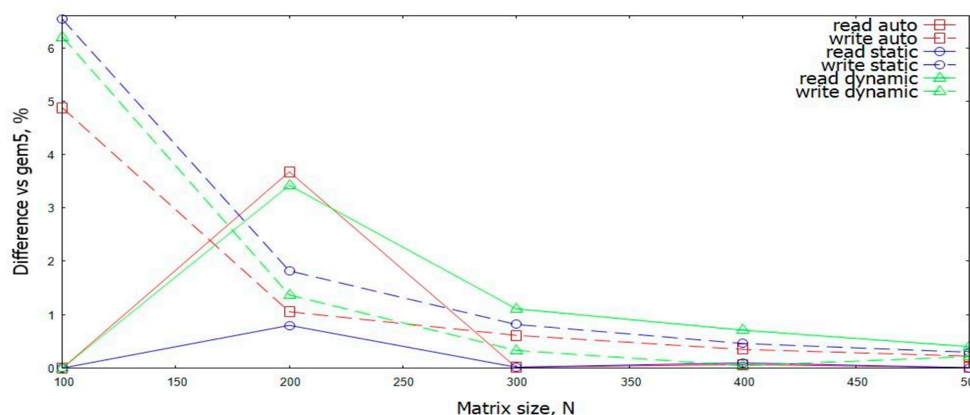

Figure 5. Difference CATS vs gem5 in the number of L2 cache misses.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

For the L2 cache, the results are similar across all data placement variants: the same deviations occur at small matrix dimensions, with subsequent convergence of results; moreover, for dimensions of 300 and above, the results are nearly identical and do not exceed 1%.

A comparison of the simulation results obtained in the CATS system with those from the gem5 architectural simulator has shown that, for all investigated matrix dimensions and data placement variants, the estimates of the number of L1/L2 cache misses and the relative trends are in nearly complete agreement. In the majority of cases, the discrepancy does not exceed a few percent, and for certain "resonant" dimensions, the deviations are of an explainable nature and, as a rule, are conservative in character (the analytical model slightly overestimates the number of conflict misses).

Since gem5 executes actual binary code and provides a detailed simulation of the processor microarchitecture and the memory subsystem, its results can be considered more comprehensive and accurate and can be used as a reference. It should be noted that simulation using gem5 takes an unacceptably long time and can hardly be used for the rapid analysis of a current software implementation during the design phase of a software system. Below are the graphs showing the dependence of the gem5 simulation time on matrix dimensions, as well as the graphs of the ratio of the gem5 simulation time to the CATS simulation time.

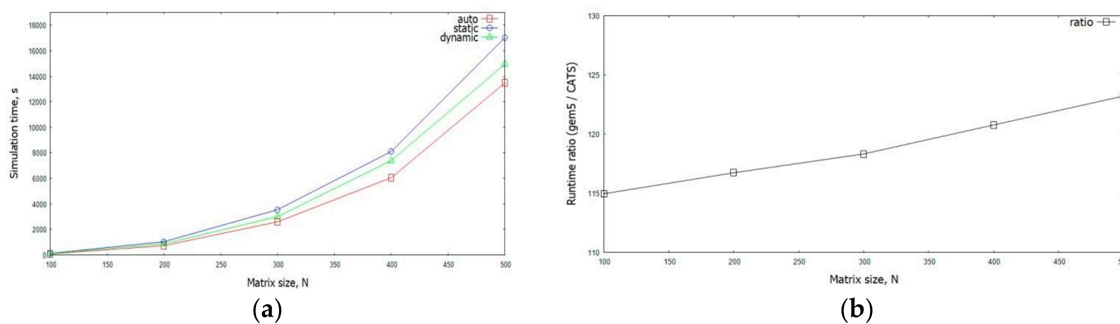

Figure 6. gem5 simulation time a) gem5/CATS runtime ratio (dynamic arrays) b).

Graph 5a) presents the simulation time. For example, for a matrix dimension of 500 (static variant), gem5 required approximately 5 hours (17,000 seconds) to complete the simulation. Graph 5b) shows that the CATS simulation time is two orders of magnitude lower (on average, 120 times faster) than the corresponding time for gem5.

Against this backdrop, the good agreement with gem5 confirms the validity of the proposed approach and allows CATS to be used for fast and sufficiently accurate evaluation of algorithm efficiency without the need for costly architectural simulation. Furthermore, CATS enables the analysis of program source code (or any portion of the source code) without the necessity of compilation and debugging.

Table 2 presents a comparison of CATS vs gem5.

Table 2. Comparison of CATS vs gem5.

Parameter	CATS	Gem5
Simulation accuracy	+	+
Simulation time	+	-
Statistics for individual arrays	+	-
Statistics for individual functions and code blocks	+	-
Applicability at the source code analysis stage	+	-

To summarize all of the above, it can be stated that gem5, like other architectural simulators, is a powerful tool for the detailed investigation of program behavior at the level of processor microarchitecture and memory hierarchy, providing high accuracy in performance evaluation but requiring significant simulation time. Against this backdrop, the developed system (CATS) enables

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

obtaining results close to those of gem5 with substantially lower computational costs, making it more convenient for rapid analysis and support of design decisions. Furthermore, the proposed system allows for obtaining more detailed statistics that can be utilized at the early stages of software system design (statistics on arrays, functions, and code blocks) through static analysis of the program source code.

4.2. CATS vs Valgrind

Compared to gem5, Valgrind allows obtaining more information, in particular, statistics on memory accesses and cache misses for each line of source code and each function, with separate accounting for reads and writes, as well as annotated profiles that directly link cache performance metrics and execution time to specific program sections. This makes it impossible to analyze cache misses separately for different arrays; however, by modifying the program code, it is possible to roughly estimate the values of these parameters. Below is an example of the source code modification required for further analysis of the Valgrind results

```
c[i*n+j] = 0;
for(k = 0; k < n; k=k+1) {
        c[i*n+j] = c[i*n+j] + a[i*n+k] * b[k*n+j];
```

→

```
c[i*n+j] = 0;
for(k = 0; k < n; k=k+1)
{
        c[i*n+j] = \
          c[i*n+j] + \
          a[i*n+k] * \
          b[k*n+j];
```

After the simulation completes, it is necessary to analyze the statistics file and aggregate the metrics from lines containing references to elements of the same arrays. For example, in the given case, to analyze cache misses for array c, it will be necessary to analyze the contents of three lines. For large codebases, it will be necessary to perform both preprocessing of the source code and subsequent analysis of the statistics file, which can become time-consuming or necessitate the development of custom tools. Furthermore, it must be taken into account that lines containing, for example, a reference like c[i*n+j], will contain aggregate information on accesses to elements of array c, as well as to variables i, n, and j.

Valgrind was run using the following command line

```
valgrind --tool=cachegrind \
        --cache-sim=yes \
        --I1=32768,8,64 \
        --D1=32768,8,64 \
        --LL=262144,8,64 \
        ./mm
```

To post-process the Valgrind results and obtain an annotated report by functions and source code lines, the cg_annotate utility was used, which allows for detailing the distribution of memory accesses and cache misses across program sections. The obtained results were compared with the results calculated by the CATS system for the L1 cache using the same comparison procedure as for the data obtained from the gem5 simulator.

Figure 7 shows the read-miss and write-miss curves for each data placement variant (automatic, static, and dynamic arrays).

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

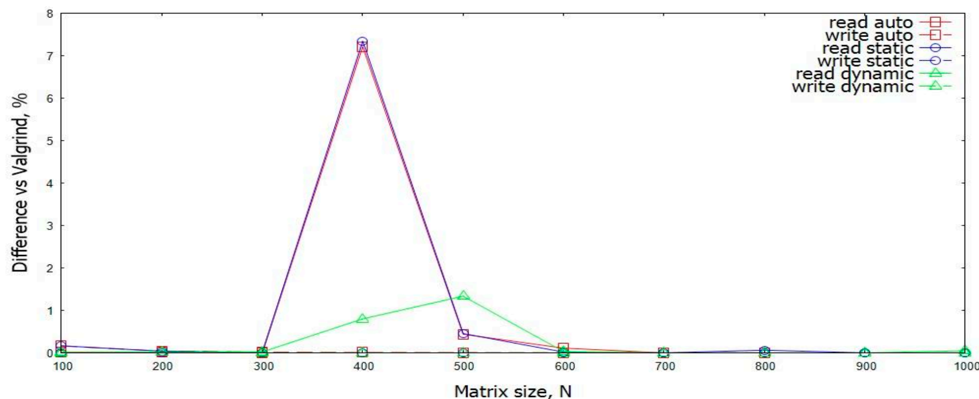

Figure 7. Difference CATS vs Valgrind in the number of L1 cache misses.

It can be seen that for all data placement variants, the relative deviation of CATS results from Valgrind results does not exceed fractions of a percent for most dimensions; in this sense, both systems can be considered to yield matching estimates. However, noticeable discrepancies are observed on each curve for two dimensions (400 and 500). For automatic and static arrays, the maximum deviation reaches approximately 7%, whereas for dynamic arrays, it does not exceed 1.4%. The reason is the same as in the comparison with gem5: at these dimensions, the model overestimates the number of cache misses. The maximum relative deviation is about 7% for automatic and static arrays and 1.4% for dynamic arrays, which allows CATS estimates to be considered accurate enough for practical applications. Moreover, since the 1.4% deviation is insignificant, it can be considered that for dynamic arrays, both systems provide matching estimates.

Below are the results of comparing Valgrind with the results calculated by the CATS system for the L2 cache (the top level of the hierarchy in the target system).

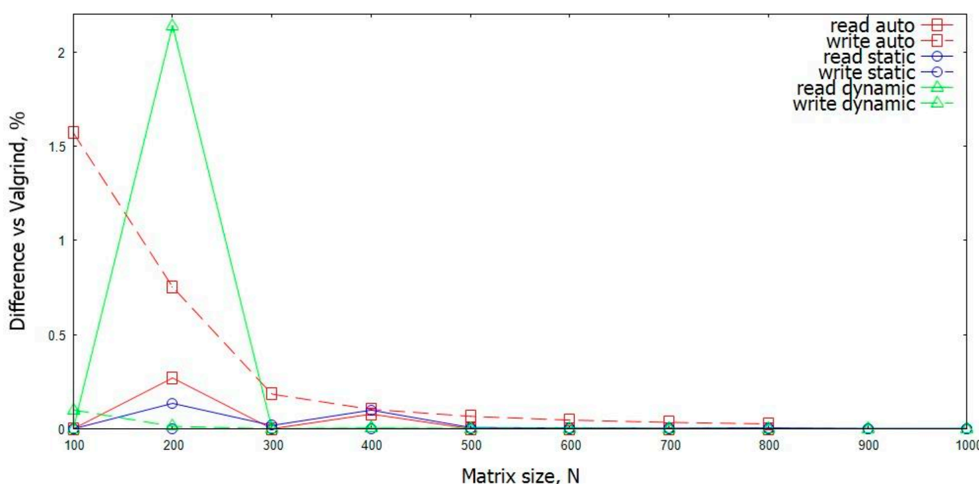

Figure 8. Difference CATS vs Valgrind in the number of LL cache misses.

As with the comparison to gem5, for the L2 cache, the results are similar across all data placement variants: small deviations (up to 2%) at small dimensions, with subsequent convergence of the results. Moreover, for dynamic arrays, the results for dimensions of 300 and above are practically identical and equal to 0; for automatic and static arrays, the differences are minimal and the values are also close to zero.

Unlike gem5, Valgrind performs simulation significantly faster. Below are graphs showing the dependence of Valgrind simulation time on matrix dimensions, as well as graphs of the ratio of CATS simulation time to Valgrind simulation time.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

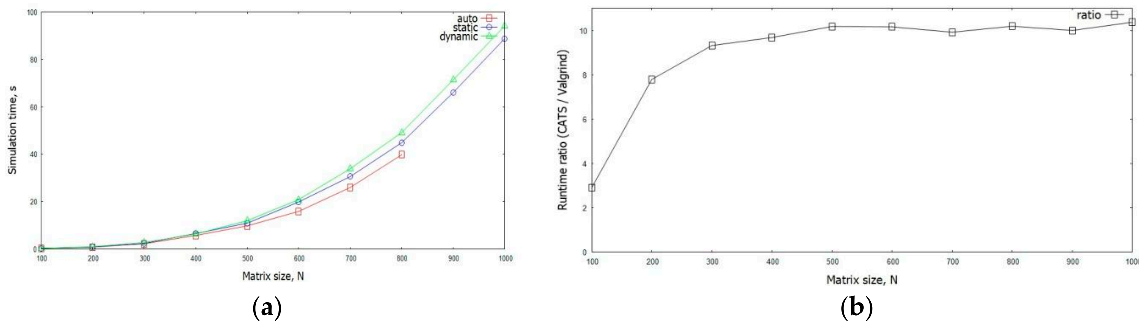

Figure 9. Valgrind simulation time a) CATS/Valgrind runtime ratio (dynamic arrays) b).

Analysis of the graphs presented in Figure 9 shows that in this case, Valgrind already performs simulation faster than CATS. Given this fact, Valgrind is more suitable than gem5 for analyzing the program under design during the design phase.

Table 3 presents a comparison of CATS vs Valgrind (Cachegrind).

Table 3. Comparison of CATS vs Valgrind (Cachegrind).

Parameter	CATS	Valgrind
Simulation accuracy	+	+
Simulation time	-	+
Statistics for individual arrays	+	+/-
Statistics for individual functions and code blocks	+	+/-
Applicability at the source code analysis stage	+	-
Number of cache levels	Any	2
Eviction algorithm	LRU, Random, FIFO, Pseudo-LRU	LRU
Write policy	write-back, write-through	write-back
Write miss policy	write-allocate, write-around	write-allocate
Cache hierarchy type	inclusive, exclusive	inclusive
Memory traffic accounting	+	-
Preliminary modification of source code	-	+/-
Post-processing of statistics file	-	+/-

The Cachegrind tool from the Valgrind suite is a powerful tool for dynamic analysis and cache memory subsystem modeling; simulation in it is performed faster than in the proposed CATS analytical model. However, this tool, like gem5, cannot be used in the early stages of program design, as it requires a compiled and debugged executable file. In addition, Valgrind has a number of limitations (see Table 3), which significantly narrows its scope of application. Finally, to obtain approximate statistical estimates separately for different arrays, it is necessary to modify the program source code and perform additional post-processing of the generated statistics file.

5. Conclusions and Future Work

An approach to designing efficient software systems has been proposed, based on partially dynamic profiling of the program under development at early stages of the lifecycle to improve memory subsystem efficiency. The developed CATS modeling system, a software complex for simulating cache memory subsystem operation based on annotated execution traces of C programs, is described.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

The proposed approach was compared with the results of two widely used tools: the gem5 architectural simulator and the Valgrind dynamic analysis tool. The gem5 simulator provides detailed modeling of the processor core and memory hierarchy at the microarchitectural level but requires significant time for simulation. Valgrind, in contrast, performs dynamic analysis of the cache subsystem during actual program execution, provides detailed statistics by function and code line, and operates substantially faster than gem5. However, it relies on a simplified cache model and, like gem5, requires a compiled executable file.

The comparison showed that the proposed CATS system provides a significant speedup in simulation, running tens to hundreds of times faster than gem5. At the same time, the results obtained by CATS are in good agreement with the estimates of both tools: for most of the investigated dimensions and data placement variants, the relative deviation in the number of cache misses did not exceed 2–3%. Unlike gem5 and Valgrind, which provide aggregate statistics, CATS generates detailed information on efficiency separately for arrays, functions, and code blocks. This, combined with the ability to operate at early lifecycle stages without a compiled executable file, allows CATS results to be used directly for the targeted improvement of the designed program's characteristics.

Future work in this direction will focus on both improving the CATS tool itself and expanding its scope of application.

First and foremost, we plan to expand the supported subset of the C syntax to ensure correct handling of more complex code constructs. Simultaneously, a key task is reducing simulation overhead. In particular, we plan to investigate intelligent trace sampling methods, where the full annotated trace is replaced by a set of representative fragments selected based on the program's phase behavior (including using machine learning methods). This will significantly reduce the volume of processed data without a noticeable loss of accuracy in key cache behavior metrics.

In addition to these main improvements, we see several promising directions for further research:

1. C++ Support: Extending the methodology to the C++ language is a logical and important step. This will require solving complex problems related to the analysis of object-oriented constructs, such as classes, templates, virtual functions, and polymorphism, which introduce an additional layer of abstraction between code semantics and memory accesses.
2. Multithreaded Program Analysis: The current version of CATS is oriented toward single-threaded applications. Adapting the method for the analysis of parallel programs (using OpenMP, pthreads, etc.) will allow for the investigation of critical aspects such as the impact of shared data on cache coherence and the occurrence of false sharing.
3. Power Consumption Modeling: Semantically annotated traces are an ideal basis for constructing detailed models of memory subsystem power consumption. By linking each memory access to an estimate of its energy cost (e.g., an L1 hit, an LLC miss, a DRAM access), it will be possible to analyze not only performance but also the energy efficiency of programs at the source code level.
4. IDE Integration and Automated Recommendations: To increase the tool's practical value, its integration with development environments (IDEs) such as VS Code is planned. This will allow cache miss information to be visualized directly in the code editor. In the longer term, the system can be extended with a module that, based on access pattern analysis, will automatically suggest practically significant optimization recommendations to the developer.

Author Contributions: Conceptualization, V.E. and A.K.; Investigation, V.E.; Methodology, V.E. and A.K; Software, V.E.; Visualization, V.E.; Writing—original draft, V.E.; Writing—review & editing, A.K.; Funding Acquisition, V.E, A.K., P.K., A.M. and V.S. All authors have read and agreed to the published version of the manuscript.

Funding: The study was supported by the Russian Science Foundation and the Administration of Volgograd region, project № 25-21-20073 (https://rscf.ru/en/project/25-21-20073/).

Data Availability Statement: Data are contained within the article.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026 doi:10.20944/preprints202605.1387.v1

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DBI	Dynamic Binary Instrumentation
WCET	Worst-Case Execution Time
CATS	C Annotated Trace-based Cache Simulator
IR	Intermediate Representation
AST	Abstract Syntax Tree
RPN	Reverse Polish Notation

References

1. Hennessy, J. L., & Patterson, D. A. (2011). Computer architecture: a quantitative approach. Elsevier.
2. Wulf, W. A., & McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. ACM SIGARCH computer architecture news, 23(1), 20-24.. DOI: https://doi.org/10.1145/216585.216588
3. Mittal, S. (2017). A survey of techniques for cache partitioning in multicore processors. ACM Computing Surveys (CSUR), 50(2), 1-39.. doi: 10.1145/3062394.
4. Leis, V., Haubenschild, M., Kemper, A., & Neumann, T. (2018, April). Leanstore: In-memory data management beyond main memory. In 2018 IEEE 34th International Conference on Data Engineering (ICDE) (pp. 185-196). IEEE..
5. Sze, V., Chen, Y. H., Yang, T. J., & Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE, 105(12), 2295-2329.. doi: 10.1109/JPROC.2017.2761740.
6. Egunov, V. A., & Kravets, A. G. (2024). The New Method for Automatic Vectorization Efficiency Increasing. In Cyber-Physical Systems: Data Science, Modelling and Software Optimization (pp. 195-208). Cham: Springer Nature Switzerland.. doi: 10.1007/978-3-031-67685-7_14.
7. Egunov, V., & Andreev, A. (2016, May). Implementation of QR and LQ decompositions on shared memory parallel computing systems. In 2016 2nd International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM) (pp. 1-5). IEEE.. doi: https://doi.org/10.1109/ICIEAM.2016.7911607.
8. Getmanskiy, V., Andreev, A. E., Alekseev, S., Gorobtsov, A. S., Egunov, V., & Kharkov, E. (2017, August). Optimization and parallelization of CAE software stress-strain solver for heterogeneous computing hardware. In Conference on Creativity in Intelligent Technologies and Data Science (pp. 562-574). Cham: Springer International Publishing. doi: 10.1007/978-3-319-65551-2_41.
9. Kravets, A. G., & Egunov, V. (2022). The software cache optimization-based method for decreasing energy consumption of computational clusters. Energies, 15(20), 7509.. doi: 10.3390/en15207509.
10. Kildall, G. A. (1973, October). A unified approach to global program optimization. In Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages (pp. 194-206)..
11. Cousot, P., & Cousot, R. (1977, January). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (pp. 238-252).
12. Nielson, F., Nielson, H. R., & Hankin, C. (2004). Principles of program analysis. Springer Science & Business Media..
13. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., ... & Stenström, P. (2008). The worst-case execution-time problem—overview of methods and survey of tools. ACM transactions on embedded computing systems (TECS), 7(3), 1-53.
14. Ferdinand, C., & Wilhelm, R. (1999). Efficient and precise cache behavior prediction for real-time systems. Real-time systems, 17(2), 131-181..
15. Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. IEEE transactions on software engineering, 27(2), 99-123..
16. Ball, T., & Larus, J. R. (1996, December). Efficient path profiling. In Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29 (pp. 46-57). IEEE..

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 May 2026
doi:10.20944/preprints202605.1387.v1

17. Nethercote, N., & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices, 42(6), 89-100.
18. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in Proc. USENIX Annu. Tech. Conf. (ATC), 2012, pp. 309–318.
19. Ding, C., & Zhong, Y. (2003, May). Predicting whole-program locality through reuse distance analysis. In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (pp. 245-257).
20. Luk, C. K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., ... & Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices, 40(6), 190-200..
21. Zhong, Y., Dropsho, S. G., Shen, X., Studer, A., & Ding, C. (2007). Miss rate prediction across program inputs and cache configurations. IEEE Transactions on Computers, 56(3), 328-343..
22. Shende, S. S., & Malony, A. D. (2006). The TAU parallel performance system. The International Journal of High Performance Computing Applications, 20(2), 287-311..
23. Knüpfer, A., Rössel, C., Mey, D. A., Biersdorff, S., Diethelm, K., Eschweiler, D., ... & Wolf, F. (2012, August). Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden (pp. 79-91). Berlin, Heidelberg: Springer Berlin Heidelberg..
24. Necula, G. C., McPeak, S., Rahul, S. P., & Weimer, W. (2002, March). CIL: Intermediate language and tools for analysis and transformation of C programs. In International Conference on Compiler Construction (pp. 213-228). Berlin, Heidelberg: Springer Berlin Heidelberg..
25. Serebryany, K., & Iskhodzhanov, T. (2009, December). ThreadSanitizer: data race detection in practice. In Proceedings of the workshop on binary instrumentation and applications (pp. 62-71)..
26. Signoles, J. (2015). E-acsl: Executable ansi/iso c specification language. Published electronically at http://frama-c. com/download/e-acsl/e-acsl. pdf.
27. Cadar, C., Dunbar, D., & Engler, D. R. (2008, December). Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI (Vol. 8, pp. 209-224)..
28. Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., ... & Wood, D. A. (2011). The gem5 simulator. ACM SIGARCH computer architecture news, 39(2), 1-7..
29. Burger, D., & Austin, T. M. (1997). The SimpleScalar tool set, version 2.0. ACM SIGARCH computer architecture news, 25(3), 13-25..
30. Patel, A., Afram, F., Chen, S., & Ghose, K. (2011, June). MARSS: A full system simulator for multicore x86 CPUs. In Proceedings of the 48th Design Automation Conference (pp. 1050-1055)..
31. Bellard, F. (2005, April). QEMU, a fast and portable dynamic translator. In Usenix ATC, Freenix Track (pp. 41-46)..
32. J. Edler and M. D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator." [Online]. Available: http://pages.cs.wisc.edu/~markhill/DineroIV. [Accessed: Mar. 15, 2026].
33. Lilja, D. J. (2005). Measuring computer performance: a practitioner's guide. Cambridge university press..
34. Muchnick, S. (1997). Advanced compiler design implementation. Morgan kaufmann.
35. DWARF Debugging Information Format Committee, " DWARF Version 5 Debugging Format Standard ". [Online]. Available: https://dwarfstd.org/dwarf5std.html. [Accessed: Mar. 15, 2026].
36. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., & Tallent, N. R. (2010). HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience, 22(6), 685-701..

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.