Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

1 Overview of Contributions and Artifacts

1.1 Paper's Main Contributions

This paper presents the following major contributions:

- C_1 We propose an enhanced data-centric profiling approach to capture the causal relationship of cache interactions at the variable level by monitoring cache line evictions and data locality.
- C_2 We visualize variable-level cache interactions using a Cache Interaction Graph (CIG) to help interpret data access patterns and diagnose the root cause of cache inefficiency.
- C_3 We extend Cachegrind to capture CIGs.
- C_4 We demonstrate the benefits of CIGs using real-world cases.
- C_5 We evaluate the performance of the approach on the latest computer system, such as the Intel Sapphire Rapids.

1.2 Computational Artifacts

Valgrind is distributed under the GNU General Public License. Our modified version of Cachegrind (based on Valgrind 3.23.0), with all sample applications used in the paper, is available in the following Git repository, which contains all computational artifacts related to our contributions.

A_1 https://github.com/Jin-Chao/Valgrind-Cacheusage.git

Please contact the first author (c.jin@uq.edu.au) of the paper to get access to this Git repository. The following table summarizes the relationship between computational artifiacts, contributions and the elements in the paper.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1 C_2 C_3 C_4 C_5	Figures 4-14 Tables 4-5

2 Artifact Identification

2.1 Computational Artifact A_1

Relation To Contributions

This artifact is a CIG extension of Cachegrind, built on Valgrind 3.23.0. Running this extended Cachegrind on an application monitors both incoming and evicted data for each cache line eviction during the simulation. The simulator collects these events, aggregates them by user space variables, and writes them into an output file, which is visualized using a graph generator script.

It also contains all applications used in the paper to demonstrate the effectiveness of our cache behaviour analysis approach.

Expected Results

Our tool captures cache behaviour of a target application during its execution. The cache behaviour is impacted by both system and application level factors, such as cache line replacement policies, memory allocation policies, application algorithm and data size. Given an identical setup, we expect the cache interaction patterns captured by our tool are consistent for repeated runs and the performance evaluation remains reproducible.

Cachegrind is a high precision cache simulator. Repeated executions with the same configuration experience negligible differences in the number of cache misses. Mapping cache lines to user space variables may vary slightly for different runs due to differences in actual memory layouts, even using the same memory allocation policy. We expect Cache Interaction Graphs exhibit a consistent pattern across repeated runs of the same application, when the following 3 key conditions are satisfied: 1) the hardware/software platform is identical, 2) the same compiler optimizations are used, and 3) the data access pattern is primarily determined by deterministic application algorithms and the data set is large enough. Running the applications included in our Cachegrind distribution under the above conditions should generate the figures consistent to those presented in the paper.

Expected Reproduction Time (in Minutes)

The expected Artifact Setup time is around 30 minutes. Artifact Execution, such as running the enhanced Cachegrind against all included samples with pre-defined configurations, typically takes 20 minutes. Artifact Analysis, which involves generating CIGs from the simulation results using `cig_generator.in` takes less than 1 minute.

The above expected computational time is obtained on an Intel Sapphire Rapids compute node.

Artifact Setup (incl. Inputs)

Hardware. Cachegrind is included in the Valgrind distribution and supports x86, AMD64 and other platforms. Our CIG extension introduces no additional hardware requirement. We have tested CIG Cachegrind on two compute nodes, with the following configurations.

1) Intel system:
   Memory: 1TB of memory
   Processors: Dual-socket configuration with 32 core Intel(R) Xeon(R) Gold 6430 CPU per socket.
2) AMD system:
   Memory: 2TB of memory
   Processors: Dual-socket configuration with 48 core AMD EPYC 7643 Milan CPU per socket.

Software. The CIG-enhanced Cachegrind is built using GCC compiler and runs on Linux. The graph generator script, `cig_generator.in`, requires Python 3, NetworkX, matplotlib and ipywidgets.

CIG Cachegrind has been tested using the following software.
OS: Rocky Linux 8.10
Compiler: GCC 12.3.0
Python stack: Python/3.13.3, NetworkX 3.4.2, matplotlib 3.10.1, and ipywidgets 8.1.5.

Datasets / Inputs. CIG Cachegrind handles any manually instrumented application that exposes memory addresses of variables for monitoring. After downloading the distribution using the provided

link, cachegrind/cig directory contains all applications discussed in the paper, including Himeno, IRSmk and PolyBench Correlation.

Installation and Deployment. Installing CIG Cachegrind follows Valgrind instructions, which are detailed in its README. Typically, it includes the following steps:

1) Clone the code from GIT.
   git clone https://github.com/Jin-Chao/Valgrind-Cacheusage
2) cd into the source directory.
3) Run ./autogen.sh to setup the environment.
   You need the standard `autoconf` tools for this step.
4) Run ./configure, with some options if you wish.
   Set the installation path using –prefix=/where/you/want/it/installed.
5) Run "make".
6) Run "make install".
   You may need to run this command as `root`, if necessary.

In the source directory, `cachegrind/cig/ProTools2025` contains all examples discussed in the paper and scripts to process simulation results, and its directory structure is shown below.

Directory `cachegrind/cig/ProTools2025` descriptions:

- `README.md` – Top-level documentation.
- `Makefile` – Build script for all examples.
- `Simple` – Examples of matrix multiplication and conflict.
- `Himeno` – Himeno benchmark.
- `IRSmk` – Radiation solver micro-kernel.
- `PolyBenchC-4.2.1` – PolyBench benchmark suite.
- `Scripts` – Python scripts for visualization and automation.
- `results` – Output and logs of profiling runs.
- `Util` – Utility functions used across benchmarks.

README.md provides details on each benchmark application, such as the URL to download the original source code and how to build them. The `Simple` directory contains examples of matrix multiplication and conflict (discussed in Section 4.3), and the `Scripts` directory contains helper scripts. Each of `Simple`, `Himeno`, `IRSmk` and `PolyBench` directories contains the original source code and optimized versions, and provides separate Makefile to build individual applications. The top-level Makefile can build all applications at once. The `results` directory contains output files used to create the figures and tables presented in the paper.

Experiment Workflow. Running the extended Cachegrind on each application produces an output file that keeps the CIG simulation result. The graph generator script, `cig_generator.in`, reads the output file to generate CIGs either interactively through an Jupyter Notebook interface or directly writing them to PDF files.

To run the CIG simulation on a target application, the application must be instrumented manually to expose the addresses of allocated memory spaces for major variables.

The experiment workflow typically consists of two steps: T_1 and T_2. Task T_1 runs a target application using CIG Cachegrind, which generates an output file. Task T_2 invokes `cig_generator.in`, located in `cachegrind/cig`, to visualize the output file generated by T_1.

To run Cachegrind on a program `prog` for cache simulation, using valgrind –tool=cachegrind –cache-sim=yes `prog`. Cachegrind specific options:

–I1=<size>,<associativity>,<line size>
–D1=<size>,<associativity>,<line size>
–LL=<size>,<associativity>,<line size>

are used to specify the size, associativity and line size for the level 1 instruction cache, the level 1 data cache and the last-level cache respectively.

Each CIG Cachegrind simulation generates 4 output files:
`cachegrind.out.<pid>`,
`cacheusage.d1.out.<pid>`,
`cacheusage.ll.out.<pid>`, and
`cacheusage.cr.out.<pid>` (<pid> is the program's process ID).
The last file, `cacheusage.cr.out.<pid>`, contains CIG data and can be visualized by running `cig_generator.in` on it.

The following is an example of running Himeno with CIG Cachegrind:

```
valgrind –tool=cachegrind –cache-sim=yes \
–I1=32768,8,64 –D1=49152,12,64 –LL=62914560,15,64 \
Himeno/bin/bmt.O3 -s M -l 10 -v varinfo.txt
```

The above example executes the Himeno binary, `bmt.O3`, which was built with gcc -O3. Among parameters passed to it, -s specifies the problem size, -l defines the loop count, -v specifies the file containing memory addresses of variables, which is read by the simulator. The file name `varinfo.txt` is hardcoded in the simulator. Please do not change it and make sure no file with the same name exists before running the simulation.

The `Scripts` directory contains automation scripts, such as `cig_cachesim.in`, that runs simulations on all benchmark applications presented in the paper using identical configurations. Another script, `figures.sh`, processes the simulation results to generate Figures 4-11 and 13-14 enclosed in the paper. The same directory also provides other scripts to compare performance of different versions of Himeno benchmark (Figure 12), and to evaluate performance slowdown introduced by the CIG simulation (Table 4 and 5). Please refer to README.md in the `Scripts` directory for details.

Artifact Analysis (incl. Outputs)

There are two modes to generate CIGs from an output file:
1) Manipulate CIGs interactively using Jupyter Notebook, and
2) Plot graphs to a PDF file.

To generate CIGs interactively, follow these steps:
1) Launch Notebook
   python -m notebook
2) Create a new Python 3 notebook
   Click New → Python 3
3) Enter the command in a code cell:
   %run `cig_generator.in` –cig `cacheusage.cr.out.<pid>`

This approach allows the user to manipulate the generated CIGs interactively, such as omitting unnecessary details, and selecting its code scope and choosing cache miss types displayed.

Helper scripts, such as `figures.sh`, are provided to invoke the graph generator on the outputs of simulating applications located in `cachegrind/cig/ProTools2025` and to produce PDF plots using the same settings as presented in the paper. Repeated runs are expected to generate figures that exhibit consistent cache interaction patterns as those included in the paper.