SCOZ: A system-wide causal profiler for multicore systems

Minwoo Ahn | Donghyun Kim | Taekeun Nam | Jinkyu Jeong

1College of Information and Communication Engineering, Sungkyunkwan University, Suwon, South Korea
2Samsung Electronics, Suwon, South Korea
3Apposha, Seoul, South Korea

Abstract
The increased complexity of hardware and software makes it difficult to analyze programs with conventional profilers. The causal profiling technique is introduced to solve the problem of conventional profilers. The causal profiling technique finds the bottleneck of the program and shows the effect of optimizing it. COZ, the newest causal profiler, exploits a technique called virtual speedup to perform causal profiling without actually optimizing program codes. However, it can only profile multithreaded applications, and cannot profile multiprogram applications and operating system (OS) kernel codes, thereby limiting the use of causal profiling. This article introduces SCOZ, a system-wide causal profiler that addresses these limitations. The proposed profiler changes the target of virtual speedup from threads to CPU cores, thereby expanding the profiling coverage to diverse applications as well as OS kernel codes. To verify our profiler, we profiled multithreaded and OS kernel-intensive applications. For multithread applications, our profiler shows identical results to what COZ provides. For the OS kernel-intensive applications, our profiler identifies identical bottlenecks that previous OS scalability studies have pinpointed. Finally, we verified the profiling capability of the proposed profiler by profiling and optimizing multiprocess applications in the NAS parallel benchmark suite.

KEYWORDS
causal profiler, Linux perf subsystem, multicore system, virtual speedup

1 INTRODUCTION

As commercial processors evolve into multicore systems, programmers need a programming model that enables an application to utilize multiple cores concurrently. Currently, applications use multithread or multiprocess programming models, which help in improving the performance of applications. However, the complexity of applications has increased due to factors such as synchronization primitives (e.g., lock, condition variable) and I/O operations. Thus, traditional profilers (e.g., perf, 1 oprofile, 2 gprof 3), which show the application's bottlenecks as an order of CPU resource consumption, have limitations in finding an accurate critical path and improving the performance of applications. A great effort is necessary to optimize a bottleneck, but once the optimization is done, the critical path becomes changed to a different execution path due to the complex use of synchronization primitives. Consequently, the optimization is no longer connected to the performance improvement of the application.
To address these issues, Curtsinger and Berger\textsuperscript{4} introduced COZ, a causal profiler for Linux. This causal profiler utilizes a technique, called \textit{virtual speedup}, that identifies bottlenecks and calculates the potential speedup by optimizing those bottlenecks. Using the results of causal profiling, programmers can work on the bottlenecks to optimize their applications for improved performance. The key idea behind the virtual speedup is to delay other concurrently running threads when a thread executes a speedup code line. This has a virtual effect of speeding up the selected line. By using the virtual speedup technique, COZ can predict the performance improvement of a bottleneck without actually optimizing the bottleneck.

COZ is a state-of-the-art causal profiler; however, it has two limitations. First, its profiling scope is limited to the user space. Many multithread programs interact with the operating system (OS) kernel heavily,\textsuperscript{5,6} and their performance optimization usually requires optimizing the OS kernel. Because the scope of COZ is limited to user space only, the application of virtual speedup to the kernel is not supported. Second, COZ cannot profile multiprocess applications since the virtual speedup is applied at the thread-level.

In this article, we propose a system-wide causal profiler called SCOZ to address these limitations. The proposed scheme expands the scope of profiling to multiple processes and the OS kernel. This is achieved by extending the virtual speedup from the thread-level to the physical core-level. This facilitates profiling not only multiple processes but also the OS kernel.

The proposed scheme is implemented on a Linux system, and its profiling performance is validated in three aspects. First, we compare the profiling results of SCOZ with COZ for multithread applications tested in the COZ article.\textsuperscript{4} With multithread applications, both profilers identify the same bottleneck and similar potential speedups of applications. Second, we test whether the proposed profiler can identify and provide virtual speedup in the OS kernel. We run the benchmarks used in the file system scalability analysis article.\textsuperscript{6} The profiling results identify the identical bottlenecks in the Ext4 file system in the kernel and their potential performance improvement. Finally, we profile MPI-based multiprocess applications in the NAS Parallel benchmark.\textsuperscript{7} We optimized two applications in the benchmark, and we confirmed that the trends in the actual performance improvement follow the virtual speedup trend reported by SCOZ.

The contributions of this article can be summarized as follows:

- We propose SCOZ, a system-wide causal profiler that uses core-level virtual speedup and can profile not only multithread applications but also profile multiprocess applications as well as the OS kernel.
- We validate the profiling results by our scheme with multithread applications and applications with kernel-level scalability bottlenecks. The results have matched with the profiling results provided in the literature.
- We optimize two MPI applications (IS and FT) in the NAS Parallel benchmark\textsuperscript{7} using our profiler. We find that the trends in performance improvement in both applications follow the speedup trends in the profiling results.

The rest of this article is organized as follows. In Section 2, COZ is described briefly to provide the background for our research. The design and implementation details of SCOZ are presented in Section 3. In Section 4, we demonstrate the validity of our profiler by comparing the profiled bottleneck with those obtained using COZ and previous studies. We introduce related works in Section 5 and provide our conclusions in Section 6.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Virtual speedup

To generate the same effect as that of the actual speedup, a virtual speedup delays other tasks every time a speedup for a code line is executed. COZ\textsuperscript{4} randomly selects a line of the code and the amount of the speedup when the application is started. It utilizes the thread-level virtual speedup by delaying all other threads, except the thread that executes the code line of speedup at runtime. Toward the termination of the application, COZ uses three values to calculate the potential speedup if the experimented line is optimized: runtime with virtual speedup, original runtime, and the sum of all inserted delays.

Figure 1 illustrates how to calculate the potential speedup through the virtual speedup when two threads are running, and the speedup line is B (speedup = 1). Figure 1(C) indicates that every time B is executed, it delays other threads by 1. The potential speedup is calculated using Equation (1).

\[
\text{Speedup} = (\text{Original runtime}) + (\text{Sum of all inserted delay}) - (\text{Runtime with virtual speedup}). \tag{1}
\]
In this case, the original runtime is 14, the sum of all inserted delays is 3, runtime with virtual speedup is 16, and the potential speedup is calculated as 1. The calculated potential speedup is equal to the actual speedup in Figure 1(B).

### 2.2 COZ

COZ is a causal profiler that finds bottlenecks in applications through the thread-level virtual speedup technique. It helps programmers identify the bottlenecks of an application and calculate their respective potential speedups by virtually optimizing each point. Implementing a thread-level virtual speedup requires the monitoring of the instructions that each thread executes. COZ collects these data through the Linux `perf` subsystem. It collects the IP and callchain of each thread and processes them for use in virtual speedup. However, the threads are not delayed immediately when the code line that COZ intends to speedup is sampled. Theoretically, the virtual speedup requires the other threads to be immediately delayed whenever each thread executes a code line to speedup; however, this results in a profiling overhead. To address this problem, COZ processes the collected samples by batching them every 10 ms.

Whenever a batch of samples of a thread is ready, COZ translates each sample to a code line and checks whether the code line is the target of virtual speedup. If so, it delays the execution of other threads. To delay the execution of threads, COZ maintains two values: local delay and global delay. The local delay is a per-thread variable and is the sum of the delays the corresponding thread enforces to the other threads. The global delay is a shared variable and is the total amount of delay enforced to the application. Hence, whenever to process a virtual speedup, the thread increases its local delay and the global delay. Then, the thread sleeps for the difference between the global delay and its local delay.

COZ also considers runtime dependencies between threads and specially handles the delay during a thread wake-up (Figure 2). When a thread (wakee) is woken up by another thread (waker), the delay enforced to the blocked thread should be carefully handled. If a delay is charged to the two threads while the wakee is being blocked, the wakeup time is delayed by the amount of delay (d2 in Figure 2(B)) because of the delay consumption by the waker (d1 in the figure). However, if the wakee is woken up and consumes the same delay again (d3), the wakee thread is delayed by two times of the originally intended delay (d2 + d3 in the figure). To avoid this problem, when a thread is blocked, COZ preserves the thread’s delay by saving the difference between its local delay and the global delay; this operation is called `pre_block()`. When the thread is woken up by another thread, the preserved difference is recovered by assigning the wakee’s local delay to the global delay minus the saved difference value; this operation is called `post_block()`. To ensure that the delay during the blocking event is correctly consumed by the waker thread (d1 in Figure 2(C)), when a thread calls a wake-up function (e.g., `unlock()` or `cond_signal()`), the thread is enforced to consume its enforced delay; this is called `catch_up()`. The three operations are properly installed in pthread APIs: `pre_block()` and `post_block()` in blocking APIs such as `pthread_mutex_lock()`, and `catch_up()` in signaling APIs such as `pthread_mutex_unlock()`.

### 2.3 Limitations

Although COZ is a state-of-the-art causal profiler, it has two limitations. First, it cannot profile the case when OS services become bottlenecks. However, many studies have revealed that the scalability problems of parallel applications
running in multicore systems are often caused by OS kernel codes.\textsuperscript{5,6} When the actual bottleneck of the application is the kernel code, COZ identifies the user code causing that kernel activity as the bottleneck and cannot accurately calculate the potential speedup. These profiling results take away the opportunity for programmers to optimize their applications. The virtual speedup should be applicable to parallel applications regardless of the programming model. Currently, COZ is not sufficiently useful for virtual speedup because it operates on a thread-level basis.

Second, COZ cannot profile multiprocess applications because the delay operates at a thread-level in the virtual speedup designed for COZ. Furthermore, COZ intercepts the pthread APIs for handling the delay-related processing for the virtual speedup (i.e., `pre_block`, `post_block`, `catch_up`). Applications that COZ can profile are limited to multi-thread applications using the pthread library with the patches for correct delay accounting. Therefore, other programming models such as multiprocessing cannot readily use the virtual speedup technique although it is still widely used in many areas, such as scientific computation, big data analysis, and database systems.

### 3 | SCOZ

#### 3.1 | Overview

The goal of SCOZ is to widen the profiling scope to any software on a system. To accomplish this, we change the level of virtual speedup from threads to cores. The following section (Section 3.2) describes why the core-level virtual speedup is able to provide the virtual speedup of codes and also provides broader support to synchronization between threads or processes without any special supports.

Then, we describe our approach to supporting system-wide causal profiling. Typical causal profilers have five steps during their profiling. First, it selects a target code line to virtually speedup. The target code can be explicitly specified by a user or implicitly selected by the sampling of IP and callchain. Second, the profiler performs sampling of IP and callchains while the target program is running. Third, the profiler checks whether collected samples match to the target line for virtual speedup. Fourth, if so, the profiler accounts for the local and global delays. Finally, the profiler enforces to consume delays if there is a mismatch between local and global delays. We describe how the five steps are changed to widen the profiling scope to the entire software and to support the core-level virtual speedup in Sections 3.3–3.5.
3.2 Core-level virtual speedup

The goal of SCOZ is to widen the profiling scope to any software on a computer system. To achieve this, SCOZ profiles not only user-level codes but also kernel codes. To this end, our idea is to change the level of virtual speedup from threads to physical cores. The rationale behind this is that whenever a certain code is running on a core, our scheme delays the execution of the codes on the rest of the cores. This can lead to the effect of virtually speeding up a certain code.

To achieve this goal, we need to consider two cases: (1) when all cores are running and (2) when some cores are idle. First, when all cores are running codes, whenever a core runs a target code line to virtually speed up, the codes running on the rest of the cores are delayed immediately. If the rest of the cores are running the threads in the same process of the core running the target code line for virtual speedup, the virtual speedup technique is identical to that in the original causal profiling.4

Second, when to speedup a target code line running on a core and if some of the cores are idle, we need to carefully charge the delay of the idle cores. We need to consider two idle core cases: (1) when a core is woken up by another core; hence having dependencies between cores, and (2) when an idle core is woken up by external interrupts. Handling the latter case is straightforward because enforcing delays while the core is idle is like delaying the external functions such as disk I/O operations. This is identical to delaying a blocked thread without dependency in the thread-level virtual speedup technique.4

The former case, however, should be carefully addressed. When an idle core (wakee) is woken up by another core (waker), the delay charged to the wakee core during the idle state should not be consumed. At the same time, the waker core should consume its charged delay during the idle time of the wakee core. This dependency handling is identical to handling dependency between threads in the thread-level virtual speedup (Section 2.2). Hence, our scheme uses the three operations, pre_idle(), post_idle(), and catch_up(), while waking up an idle core that has intercore dependency; details are explained in Section 3.5.1.

In this subsection, we clarify whether our core-level virtual speedup technique handles the dependency between threads correctly, even when a core involved in the thread wakeup is idle or not. Figure 3 shows the case when a thread (T1) wakes up another thread (T4). When Core 3 (T5) runs a target code line for virtual speedup, its delay is charged to the two cores (core 1 and core 2). We have two cases here: (1) Core 2 is not idle (Figure 3(B)), and (2) Core 2 is idle (Figure 3(C)). As shown in Figure 3(B), when Core 2 is not idle, it consumes the charged delay (d2) by Core 3. Then, T4 is woken up at the right time. Meanwhile, when Core 2 is idle during the thread wakeup, it is forced not to consume the delay charged by Core 3, but still T4 is woken up at the right time because of the intercore dependency handling. Hence, whether a core involved in interthread dependency is idle or not, our scheme provides the same virtual speedup results as the original causal profiler provides.

(A) Original program  (B) Core-level virtual speedup  (C) Core 2 is in idle state before thread 4 wakes up

FIGURE 3 Illustration of core-level virtual speedup with the dependency between threads: (A) The original case, (B) the case without idle cores, (C) the case with an idle core. (B) and (C) are correctly handled with the proposed core-level virtual speedup technique [Colour figure can be viewed at wileyonlinelibrary.com]
3.3 | Per-core handler thread

To support the core-level virtual speedup, our profiler spawns a thread for each core, as shown in Figure 4. The threads are denoted as per-core handler threads. Each of the threads is pinned onto the designated physical core and performs all the five steps for the virtual speedup.

Among the five steps, sampling IP/callchain is a crucial part of causal profiling because sampling significantly lowers the overheads of typical profiling. In our scheme, we use the sampling not in thread-level but in the core-level. Hence, we use the Linux perf subsystem, and we use the event type `PERF_COUNT_SW_CPU_CLOCK` to sample IP/callchain on each core. In addition to the collection of IP/callchain, SCOZ records the `pid` (process ID) of the task of the collected samples. Because all the tasks that can run on each core can be sampled, a pid is needed to distinguish which task the collected samples are from. This collected pid is used to determine whether SCOZ needs to build an additional mapping table on-demand, and the details are described in Section 3.4.

Each handler thread periodically receives collected samples via a shared memory between the thread and the perf subsystem. Whenever a batch of samples are collected, the thread performs mapping between IP/callchain and code line. The sampling is used in both first and second steps, but we assume that the selection of the target code line to virtually speedup is done. Hence, when a sample is collected, the handler checks whether the sampled IP/callchain is the virtual speedup target. For this checking, `mapping tables` play an important role, which is described in the following subsection. Figure 5 describes the actions of the handler thread with corresponding sections having the descriptions of the actions.

3.4 | Mapping tables

The mapping table is a data structure used to translate the sampled IP and callchain to the line number of the application code. Based on the `DWARF` debug information of the target binary, the profiler builds the mapping tables statically at the start of the run. The original COZ has an option to specify the target binaries; this option is denoted as `binary scope`. If the binary scope is specified, the binaries in the option are loaded during the startup of the profiler, and the mapping table for

---

**FIGURE 4** The structure of SCOZ

**FIGURE 5** Flowchart of the per-core handler thread
each binary is built. The original COZ also has an option to limit the scope of source codes for the translation; this option is denoted as source scope. The profiler abandons the sample and processes the next sample if this option is specified, but the IP or callchain of the current sample does not belonging to any of the source files specified by the option. This option is effective in limiting the source codes to profile. SCOZ inherits these two options.

However, since the target of profiling in our profiler is the entire software running on a system, it is sometimes difficult to specify target binaries or target source codes. To handle these cases, the profiler supports to build mapping tables on demand when a new executable first appears. This on-demand building of the mapping table is enabled only when the option binary scope is disabled.

Our profiler supports multiprocess applications, and processes sometimes share the same virtual memory layout (e.g., fork()). Our profiler does not maintain a mapping table for each process but provides a way to share a mapping table between multiple processes. This facilitates to save memory for mapping table and to save time to avoid building unnecessarily duplicated mapping tables. For example, an MPI program spawns multiple processes, and a first process appeared to the sampling cause the building of a mapping table for the new executable. The program has other processes sharing the same executable, and once their samples appear, processes are new but executable files are not new. In that case, by checking the executable of the pid in a sample, the pid can share the existing mapping table with the same executable file.

Finally, our profiler provides an option called kernel scope. When it is specified, the profiler builds the kernel mapping table using the kernel boot image specified by the option. Our profiler can also specify the source scope using the kernel source files. Since the kernel text segment is shared between all the processes in a system, the profiler maintains only a single mapping table for the OS kernel.

After mapping tables are ready, the handler thread processes each sample. Hence, it first translates the collected IP/callchain to the code line using the mapping table. If no translation is made, it processes the next sample. If a translation is found, it checks whether the translated line is the target of virtual speedup. A sample of IP/callchain can be the target of virtual speedup in two cases: (1) IP is the target line or (2) any return address in the callchain belongs to the target line. If the sample is the target line, the handler thread processes to account for the delay of other threads.

### 3.5 Delaying cores

Our profiler inherits the use of local and global delays in COZ. Our scheme, however, maintains local delay at a per-core basis. In addition, the local delays and global delay are managed by the handler threads, but the actual delays each core needs to consume are done inside the OS kernel. To facilitate the sharing of the delay values between the handler threads and the OS kernel, we build a shared memory between the user-space and the kernel space. A shard page is allocated and the local delays for the cores and the global delay are stored in the page. Atomic instructions are used to modify the values.

Consequently, when the handler thread needs to apply the virtual speedup because it finds a sample that matches to the target line of virtual speedup, it increases the local delay of the current core and the global delay by the amount of the delay it wants to inject. We inherit the random delay injection in COZ.4

After processing all samples in this period, the handler thread synchronizes the local delay of the current core with the global delay. Hence, it forces the current core to sleep for the difference between the local delay and global delay. Different from the thread-level virtual speedup, which requires to pause the current thread for the specified amount of time, our core-level virtual speedup needs to pause the current core. This can be performed by the help of the OS kernel. To this end, we use a special system call to make the current core to pause. Hence, the handler calls the system call and the rest is done by the OS kernel.

Our profiler runs with the Linux kernel. Hence, we add a new system call in the kernel to consume the delay of the current core (the global delay minus the local delay of the current core). When the handler thread calls this system call, the Linux kernel uses the ndelay interface to spin for a given time period. Moreover, before and after calling ndelay interface, context switching is prevented by invoking preempt_disable() and preempt_enable(), respectively. Therefore, the Linux kernel results in the same effect as the core stopped.

### 3.5.1 Handling idle cores

In addition, the Linux kernel manages delays of cores when a core enters an idle state and is woken up from the idle state via an IPI. As explained in Section 3.2, the delay management around the idle core state is to avoid overdelay
of a core waken up by another core. To this end, we implement three operations, \texttt{pre_idle()}, \texttt{post_idle()}, and \texttt{catch_up()} and modify three parts of the OS kernel: (1) entering an idle loop, (2) exiting from the idle loop, and (3) waking up an idle core.

First, \texttt{pre_idle()} remembers the gap between the core’s local delay and global delay. This operation is called when the core enters the idle loop, which is a loop halting the current core and then scheduling the runnable thread if any existing. Otherwise, the loop repeats to halt the current core. Accordingly, calling the idle loop is the signal the core has become idle. Accordingly, we preserve the gab of the local and global delay to avoid the delays enforced while the core is in the idle state.

Second, \texttt{post_idle()} recovers the saved delay gap. This operation is called when the core is woken up. However, we carefully call the \texttt{post_idle()} only when it is desired to recover the saved delay gap. For example, if a core is woken up by external interrupts such as disk interrupts, the core should not restore the saved gap because the core should consume the delays enforced while the core is idle. On the contrary to this, when the core is woken up by another core via an interprocessor interrupt, the core should not consume the delays enforced during its idle state. Hence, that case is the one to call the \texttt{post_idle()} operation. To this end, while exiting the idle loop, we check a flag that specifies another core has woken up the core. If the flag is set, it calls \texttt{post_idle()} to restore the saved delay gap.

Third, \texttt{catch_up()} enforces a core to consume its accumulated delay by calling the \texttt{n_delay} function with preemption disabled. This operation needs to be called when a core wakes up an idle core. The Linux kernel has a function \texttt{try_to_wake_up()} to wake up a task in the core scheduler. The function can place the woken-up task on an idle core. In that case, the function also sends an interprocessor interrupt to signal the idle core. If this step happens, the function sets the flag to notice the woken-up core of that the core needs to call the \texttt{post_idle()} operation. Accordingly, the waker core calls the operation \texttt{catch_up()} in this case to make sure the woken-up core indirectly consumes the delays during its idle state.

3.5.2 Handling thread migration

As explained in Section 3.2, the core-level virtual speedup conceptually works when delaying other cores synchronously occurs. However, our profiler relies on the sampling and the batching of sample processing. These are effective in reducing runtime overheads. However, these inevitably makes the delaying operation asynchronous.

The asynchronous delaying operation raises an issue in SCOZ. For example, when a target line for virtual speedup appears on a core, the rest of the cores should be delayed. However, the actual delay will be enforced later on the rest of the core. Before the delays are consumed in the cores, if the thread that executed the target line has migrated to one of the cores, consuming the delay on the core means that the target line is delayed. As a result, incorrect virtual speedup results would be generated due to the asynchronous delay consuming nature.

To address this problem, the local delay in the destination core should be made to synchronize with the global delay at the time when a thread migration decision is made. To this end, we modify the Linux kernel so that when to select a destination core for thread migration, cores that do not consume its accumulated delay are excluded. To implement this, \texttt{select_idle_sibling()}, a function that selects the destination core during wake-up of a thread, and \texttt{load_balance()}, a function of the Linux load balancer, are modified. When selecting a destination core for thread migration, if the global delay is greater than the core’s local delay, the core is excluded from the selection. This modification may change the thread-core mapping during runtime. However, thread migration does not frequently occur in many parallel applications because they usually pin threads to cores or match the number of threads to the number of cores.

4 EVALUATION

In this section, we evaluate how well SCOZ profiles various types of applications. First, we verify the profiling performance of our profiler with multithread in the PARSEC benchmark suite. For the result verification, we compare the results of our profiler with the results of COZ on our system and the results in the COZ article. We select two applications, \texttt{ferret}, \texttt{fluidanimate} since the virtual speedup graphs of the two applications are available in the original COZ article.

Second, we verify how well our profiler profiles the kernel codes. Since COZ has no results in kernel profiling, we compare the profiling results with the bottleneck information disclosed in the literature. For kernel profiling, we use two benchmark programs: \texttt{Dbench} and \texttt{Filebench}.
Finally, we profile and optimize multiprocess applications. We profile two MPI-based applications (IS and FT) in NAS Parallel Benchmark suite.\textsuperscript{7} SCOZ reports bottlenecks in each application, and we analyze and optimize the bottleneck points.

We performed the experiments on a Dell R930 rack server with four sockets. An Intel Xeon Processor v4 (E7-8890), which has 24 physical cores, is installed in each socket. We used the 4.2.0 version of the Linux kernel on Ubuntu 16.04. Moreover, we disabled dynamic voltage and frequency scaling to avoid nondeterministic delays caused by frequency scaling. All profiling results are averaged for at least ten repetitions to eliminate unexpected speedup results because of any outliers.

\section{Multithread applications}

In this experiment, the location of the inserted progress point for each application is the same as that in the COZ article.\textsuperscript{4} The profiling results for each application are depicted in Figures 6 and 7. The top of the graph in the figure indicates the bottleneck point (filename with a line number). The values indicate the potential speedup with respect to the line speedup. The profiling results for the multithread application found the bottleneck identical to that of COZ in both experiments, and similar speedup results are obtained. The details of each application are shown as follows.

\subsection{ferret}

Ferret is an application that performs content similarity search. The application consists of six pipelined stages. Except for the first and last stages, the four middle stages are the main stages for image searching. The main stages perform four operations: image segment, feature extraction, indexing, and ranking. The first and last stages are handled by one thread each, and the main stages are executed by multiple threads in parallel. There is a shared queue between the adjacent stages, where the threads receive data from the previous stage and send it to the next stage after processing. The threads belonging to each stage are executed concurrently as a pipelined structure.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{ferret.png}
\caption{Profile results of ferret using COZ and our scheme on our machine (A) and the result in the COZ article (B) [Colour figure can be viewed at wileyonlinelibrary.com]}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fluidanimate.png}
\caption{Profile results of fluidanimate using COZ and our scheme on our machine (A) and the result in the COZ article (B) [Colour figure can be viewed at wileyonlinelibrary.com]}
\end{figure}
As same as in Reference 4, we inserted a progress point in the end of the output stage (398th line in original ferret-parallel.c) and specified the ferret-pthreads.c file as a source scope. The number of threads for each main stage is 24 (96 threads are running concurrently in total). Figure 6(A) shows the profiling result of using SCOZ and COZ on our experimental setup. As a result, the 360th code line of ferret-pthreads.c is selected as the bottleneck. This line is also pointed out in Reference 4, as shown in Figure 6(B). The line number is not identical since there are two additional lines in the ferret pthread.c file in our experimental configuration. However, both results indicate the same line calling the function cass_table_query(). The virtual speedup result is verified to become actual speedup by optimizing the specified code line in the COZ article.

4.1.2 fluidanimate

Fluidanimate is an application that simulates incompressible fluid using the extension of the Smoothed Particle Hydrodynamics method. It performs eight steps of calculation for each frame, and each step is separated by a barrier. We inserted a progress point at the end of the barrier (line 1151 in pthreads.cpp). The source scope is not specified. Since the number of threads is limited to the power of 2, we used 64 worker threads. Both our profiler and COZ on our machine pointed the line 151 in parsec_barrier.cpp as the bottleneck, as shown in Figure 15(A). This result is identical to the result in the COZ article, as shown in Figure 7(B). This code is included in parsec_barrier_wait() function that is customized barrier implementation and the next line (152nd line) is a loop that calls pthread_mutex_trylock() repeatedly. This virtual speedup result is verified by achieving the actual speedup of the application in the original COZ article. In more detail, by replacing the customized barrier to the pthread_barrier, the application showed improved performance.

One might notice that the graphs show negative speedup; this is denoted as a downward slope. However, this downward slope does not necessarily mean that the application is slowed down if the target line is sped up. The downward slope is a strong indication of contention. As evidence, the COZ article optimized three target lines showing downward sloping in the PARSEC benchmark suite, and fluidanimate is one of the three applications.

4.2 Kernel profiling

4.2.1 DBench

We used the DBench benchmark to verify that SCOZ can profile kernel codes. DBench is a benchmark in which a large number of clients perform a series of operations consisting of create, delete, append, read, write, and rename and generate a large amount of I/O operations. Each client corresponds to one process or thread. In this experiment, we set a client as a process. The number of clients for the experiment is 96, and a progress point is inserted at the end of processing each operation. We also entered the path to the vmlinux file to profile the kernel code. Figure 8 shows the profiling results of Dbench.

![Kernel profile results of DBench](image-url)
SCOZ found bottlenecks in lines 3335 and 3894 of the fs/namei.c file of the kernel source code. These lines are the parts that call the path_openat() and do_unlinkat() functions, respectively, and are involved in creating and deleting files, respectively. To verify this profiling result, we refer to the scalability study of the file system. This study mentioned that a scalability problem occurs because various objects for synchronization are used in the process of metadata update, file deletion/creation, and file write. However, it is difficult to determine whether the results found by SCOZ are caused by synchronization objects. Although the lines captured by the profiler are related to file system metadata operations, it is uncertain which synchronization objects (e.g., locks or condition variables) are related to the scalability problem. Therefore, to verify that SCOZ can detect bottlenecks caused by the synchronization object, we conducted additional experiments by limiting the source scope to fs/.*.

Figure 9 shows the results of experimenting by limiting the scope to files in the fs/ directory. Line 141 of the fs/ext4/fsync.c file is the code that calls the jbd2_complete_transaction() function. This function uses journal->j_state_lock as the synchronization object. It is clear from the previous file system scalability study that this object causes scalability bottlenecks in ext4. Line 312 of fs/ext4/ext4_jbd2.h is the code that calls the __ext4__journal_start_sb() function. This function contains codes that increment t_handle_count atomically. This code is also found to be a bottleneck in previous file system scalability studies. Therefore, it is confirmed that SCOZ can find bottlenecks in the kernel codes.

4.2.2 Varmail

Varmail in Filebench is an application that emulates the mail server’s I/O activity. The workload consists of operations such as create-append-sync, read-append-sync, read, and delete in a single directory. In this experiment, the number of threads is set to 96. Progress points are inserted in the loop that handles each operation.

Figure 10(A) shows the result of profiling varmail. The profiler points to line 3894 in fs/namei.c as the bottleneck. This line is about deleting files. Similar to the profile results of DBench, it is difficult to point specific objects causing the contention problem.
Therefore, further experiments are conducted with the source scope limited to `fs/*`, and the result is shown in Figure 10(B). The profile result points out the bottleneck as the line 141 of `fs/ext4/fsync.c` which is locking the lock variable `journal->j_state_lock`. This bottleneck point is identical to that in Figure 9(A) since both applications are file system metadata-intensive workloads so as to incur high contention to file system journaling. Therefore, we can verify that SCOZ provides adequate profiling results of the kernel codes.

### 4.3 Profiling and optimizing multiprocess applications

In this subsection, we verify the profiling capability of SCOZ by profiling and optimizing multiprocess applications. We profile applications in the MPI version of the NAS Parallel Benchmark suite. Based on the profiling results, we select two applications, integer sort (IS) and Fourier transform (FT), and optimize the two applications.

#### 4.3.1 Integer sort

The IS workload generates uniform random numbers during startup and sorts them using the counting sort algorithm. We insert the progress point at the end of each iteration in the main loop (Figure 11). Then, we profile the D class of IS with 64 MPI processes. After we profile the application, line 618 of the main loop is selected as the bottleneck, as shown in Figure 12(A).

To optimize the bottleneck point, we need to understand the operations in the main loop. In the main loop, the number of input keys (`NUM_KEYS`) is $2^{25}$, the number of buckets (or the size of array `bucket_ptrs`) is $2^{10}$, and the value of `shift` is 17. The array `key_buff1` is a large array divided into $2^{10}$ buckets, and each element in `bucket_ptrs` has the first index of the corresponding bucket. Hence, line 618 assigns the current key to the end of the target bucket and increases the bucket pointer by one. In other words, a bucket is filled sequentially due to the increment operator to `bucket_ptrs` at line 618.

With considering these behaviors, we assume that the smaller the number of buckets, the better cache memories are utilized because of the spacial locality in accessing the buckets and temporal locality in accessing the array `bucket_ptrs`; a small number of buckets makes the array size small thereby making it always fit in the cache memory. Because of the sequential access in each bucket, only one cache line for each bucket needs to be cached in cache memory. Hence, having a small number of buckets has advantages in caching because a few numbers of cachelines are necessary.

---

**Figure 11** The main loop of integer sort pointed as the bottleneck by SCOZ

```plaintext
615: for( i = 0; i < NUM_KEYS; i++ )
616: {
617:     k = key_array[i];
618:     key_buff[ bucket_ptrs[k >> shift]+ ] = key;
619: }
```

**Figure 12** Profile result and optimization result of integer sort workload. Profile result (A) includes the virtual speedup (black line) provided by our profiler and the actual speedup (red line) achieved by our optimization with regard to the speedup of the target line. Optimization result (B) shows the normalized throughput with changing the number of buckets in the main loop [Colour figure can be viewed at wileyonlinelibrary.com]
to be cached. Based on this assumption, we vary the number of buckets from 256 to 4096 and measure the performance (Figure 12(B)), and cache hit ratios (Figure 13). As shown in the table and figures, the smaller the number of buckets, the higher the throughput of the application with higher cache hit ratios (the last-level cache and the level-1 data cache).

In order to verify the virtual speedup graph generated by our profiler, we plot the actual speedup points in Figure 12(A) (the red line). We plot only four points because the original virtual speedup graph is with 4096 buckets, and we have the rest of four bucket numbers. We obtain the speedup of the target line by using the CPU time consumed by the target line; when the target line is sped up, its CPU time is reduced. As shown in the figure, the actual speedup line does not exactly match with the virtual speedup line. The reason for this phenomenon is because the change to the number of buckets affects other parts of the program. Hence, with the small number of buckets, each bucket has larger values to be sorted; this increases the execution time for sorting the number in each bucket. As a result, even if the main loop is fully sped up, the total program’s speedup does not match with the projected speedup because some other parts in the program are slowed down.

4.3.2 Fourier transform

The FT application performs the fast FT to solve a 3D partial differential equation. We insert the progress point in the loop inside the subroutine named cffts1 (line 921 in ft.f). Then, we profile the FT application using SCOZ. We profile the C class of FT using 64 MPI processes. The profiler result points line 1336 in the ft.f file as the bottleneck, as shown in Figure 15(A).

The bottleneck is a loop inside the subroutine named transpose2_local and is depicted in Figure 14. The subroutine transposes an input matrix xin of size n1*n2 and stores the output into the output matrix xout. The element of each matrix is double complex (16 bytes). The matrix z is a temporal buffer during the transpose operation. The size of z is transblock*transblock. In our test case, the values of n1, n2, and transblock are 2^9, 2^12, and 32, respectively. The matrix z, a unit of the matrix transpose operation in the subroutine, is a transblock * transblock matrix of type double complex (16 KB). Since the subroutine performs its operation at the granularity of matrix z, the size of z is crucial to the efficient use of the private caches of each core.

---

**Figure 13** Hit ratios of L1 data cache and last level cache

**Figure 14** The main loop of Fourier transform pointed as the bottleneck by SCOZ

```
1330: do jj = 1, transblock
1331:   do ii = 1, transblock
1332:     z(jj,ii) = xin(i*ii, j+jj)
1333:     end do
1334: end do
1335:
1336: do ii = 1, transblock
1337:   do jj = 1, transblock
1338:     xout(j+jj, i+ii) = z(jj, ii)
1339:     end do
1340: end do
```
Figure 15 shows profile results and optimization results of Fourier transform workload. Profile results (A) include the virtual speedup (black line) provided by our profiler and the actual speedup (red line) achieved by our optimization with regard to the speedup of the target line. (B) shows the normalized throughput by changing the size of transblock; the baseline is the default configuration of the benchmark application. [Colour figure can be viewed at wileyonlinelibrary.com]

Figure 15(B) depicts the result of measuring the throughput of FT with different values of transblock. When the value of transblock is 128, the highest average throughput is obtained, up 3.48% from the default case. The size of $z$ with a transblock value of 128 is 256 KB, which is the same as the L2 cache of the processor we used.

To verify the virtual speedup result, we plot two cases of the execution of the application with our optimization. Hence, we configure the transblock to 128 and run the workload five times. Then, we plot the average and maximum throughput cases onto the virtual speedup graph (the red line in Figure 15(A). The speedup value of the code line is obtained by using the amount of CPU time consumed for the subroutine transpose2_local. Whenever a subroutine is optimized, its CPU time is reduced. By using this characteristic, we can obtain the speedup value of the target subroutine. Consequently, the actual speedup line using our optimization mostly overlaps with the virtual speedup line drawn by our profiler.

5 | RELATED WORKS

Profilers are being actively studied to help identify bottlenecks and provide programmers with optimization opportunities. Conventional profilers were developed that identify bottlenecks by profiling the CPU usage and execution time for each function in an application. However, these profilers cannot identify bottlenecks accurately for parallel applications, because the function that uses the majority of CPU resources may not be a critical path in an execution. Many profilers have been developed to address this problem. Recently, profilers have been developed to profile off-CPU tasks such as disk I/O and networks, as well as profilers that support the presence of synchronization among the application tasks. Moreover, some profilers have been developed to obtain critical paths for parallel applications.

However, none of these profilers could determine the quantitative impact of each bottleneck on the performance of an application. Therefore, programmers are able to know the performance impact of a bottleneck only after the bottleneck is actually optimized. To address this problem, COZ, a causal profiler, was developed to evaluate the performance improvement quantitatively without actual optimization of bottlenecks. COZ exploits the virtual speedup technique and can profile multithread applications. COZ+ is an extended version of COZ to profile web-browser-based applications. OMP-WHIP, TASKPROF, and TASKPROF2 are the profilers that support causal profiling of parallel applications, and dprof is the profiler that supports causal profiling on distributed systems. Unlike SCOZ, the profiling scope of these profilers is limited to user space whereas SCOZ can profile codes in user space as well as kernel space.

6 | CONCLUSION

Identifying and optimizing the application’s bottlenecks are crucial for programmers. The causal profiling technique guides programmers on where to focus on optimization. However, COZ, a state-of-the-art causal profiler, can profile only multithread applications at the user level. SCOZ overcomes this limitation by designing a kernel mapping table
and the core-level virtual speedup. Our profiling results for multithread applications showed identical bottlenecks with similar potential speedups, as that shown by COZ. The kernel scope profiling results confirmed that the proposed profiler obtained the same bottlenecks as did the previous research on the Linux kernel file system. Finally, we provided a use case for the proposed profiler that optimizes the MPI-based multiprocess applications in the NPB. By overcoming the limitations of COZ, SCOZ can provide programmers with improved optimization opportunities.

ACKNOWLEDGMENTS
This work was supported partly by Korea Institute for Advancement of Technology (KIAT) grant funded by the Korea Government (MOTIE) (N0001883, The Competency Development Program for Industry Specialist) and by Samsung Electronics.

ORCID
Jinkyu Jeong https://orcid.org/0000-0002-4905-9244

REFERENCES


**How to cite this article:** Ahn M, Kim D, Nam T, Jeong J. SCOZ: A system-wide causal profiler for multicore systems. *Softw Pract Exper*. 2021;51:1043–1058. https://doi.org/10.1002/spe.2930