An Extended GPROF Profiling and Tracing Tool for Enabling Feedback-Directed Optimizations of Multithreaded Applications

Sandro Bartolini
Dipartimento di Ingegneria
dell'Informazione, Università di Siena, Italy
email: bartolini@dii.unisi.it

Antonio C. Prete
Dipartimento di Ingegneria
dell'Informazione, Università di Pisa, Italy
email: prete@iet.unipi.it

Abstract

This paper presents an approach for profiling and tracing multithreaded applications with two main objectives. First, extend the positive points and overcome the limitations of GPROF tool when used on parallel applications. Second, focus on gathering information that can be useful for extending the existing GCC profile-driven optimizations and to investigate on new ones for parallel applications. In order to perform an insightful profiling of a multithreaded application, our approach proposes to gather intra-thread together with inter-thread information. For the latter, Operating System activity, as well as the usage of programmer-level synchronization mechanisms (e.g., semaphores, mutex), have to be taken into account.

The proposed approach exposes various per-thread information like the call-graph, and a number of intra-thread ones like blocking relationship between threads, blocking time, usage patterns of synchronization mechanisms, context switches.

The approach introduces a relatively low overhead which makes it widely applicable: less than 9% on test multithreaded benchmarks and less than 3.9x slowdown for the real MySQL executions.

1 Intro and motivation

Parallel and, in particular, multithreaded programming is very common especially in general-purpose applications (*e.g.*, office automation, OS services and tools, web browsers) and in special-purpose systems like web- and DB-servers, but is gaining increasing importance also in the embedded domain due to the market demand for more and more complex portable applications, and the technological offer of growingly powerful devices. In addition, the trend towards on-chip parallel architectures enforces the general interest towards managing parallel applications along the entire software development process, (*i.e.*, from the design and programming phases, down to compiling, optimizing, debugging, testing, and running phases) even if it is far more complicated than in case of sequential applications [1].

The simple, but still very useful, profiling capabilities provided by *gprof* GNU tool [11] for mono-process, mono-threaded applications is not applicable for gathering insightful information for multi-threaded ones because of two main reasons: a) the collected information are per-process and, therefore, are not able to investigate on the thread-specific behavior; b) there is no way to gather inter-thread information, which are related to both cooperation and competition for shared resources, which the threads use through the Operating System (OS) primitives for synchronization (*e.g.*, semaphores).

In order to tune the performance of applications through specific optimizations [5] (manually and/or automatically), each thread profile has to be available, as well as specific information on the interaction between threads. For instance, some feedback-directed optimizations for cache performance, like Pettis and Hansen one [6], are already present in GCC and rely on the function call-graph, which is collectable by *gprof* on mono-threaded applications. Additional statistics for the analysis of temporal and spatial locality of functions, which could enable more sophisticated optimizations [7][8][9], are still missing even for mono-threaded applications. For multi-threaded applications the *gprof* tool only collects the statistics on the main thread, which can constitute a negligible part of the executed instructions and of the execution time of the application. This work aims to provide a profiling framework that can put the bases for the profiling/tracing of multithreaded applications so that existing and, possibly, new feedback-directed optimizations can be investigated.

In addition, for debugging and testing purposes, the history of the parallel execution should be made available at a granularity that can allow following the execution through each function of each thread, to inspect the scheduling/descheduling events in the OS, and go through the synchronization operations. Essentially, in order to have a precise snapshot of the runtime behavior a multithreaded application, the parallel execution has to be collected in a way that both thread activities and the interaction between threads, mediated by the OS, could be ideally *re-played* offline.

These requirements are far more complicated than the corresponding mono-process ones, especially because of the tight interaction between the application and the OS services, which forces to investigate also on the behavior of the OS itself during the application execution.

Another crosscutting issue it that the profiling activity should have low overhead because parallel applications tend to be complex (no big slowdown is typically affordable) and, in particular, parallel applications can be very sensitive to the probe-effect of profiling itself, which may artificially alter the execution time of specific code fragments and, consequently, the relative speed of the involved concurrent activities.

2 Proposed approach

We analyzed existing approaches for profiling parallel applications and none of them had the ability to easily collect all the information highlighted in the previous paragraphs and, at the same time, being easily portable on a variety of platforms. For instance, the ATOM tool [2] allows gathering detailed information on applications executing on Alpha processors, relying on special hardware features of such processors. From an other perspective, full-system simulators can be adapted to collect any information on the applications running in the simulated system, provided that specific probes are inserted in the simulator framework. However, the slowdown could be not negligible for a number of applications. In Figure 1, a number of existing profiling/tracing tools is shown, along with the time in which each one operates in the overall application development process. For space reasons, we will not go into the details of each tool.

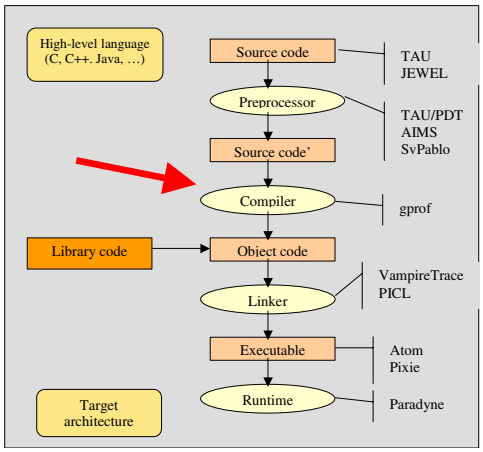

Figure 1: various profiling tools, each one operating in a specific moment of the software generation process.

Essentially, our proposed solution instruments the code using the standard interface of the *gprof* tool and implements the extension of its profiling/tracing capabilities to multithreded applications. In this way, thread-specific profile data are collected and can be later elaborated and inspected.

We instrumented in the same way also the GNU runtime libraries thanks to the unmodified *gprof* [10][11] interface towards GCC toolchain.

In addition, information on the Linux kernel services and on the scheduling activity are collected through a specifically modified version of the Linux Trace Toolkit [3] and interfaced coherently together with the statistics of the userspace threads. In this way, it is possible to insert the kernel events inside each thread statistics and in the thread execution history so that the whole picture of the parallel application behavior can be collected. In particular, the collected per-thread information comprise:

- Execution statistics of each function (*e.g.*: number of invocations);
- Function Call graph and history, comprising synchronization operations (*e.g.*: wait, posts);
- Non-active time (blocked or de-scheduled), active time;
- Average number of functions executed in active times;
- Statistics for each synchronization operation (*e.g.:* number of invocations, number of times it was blocking);

Inter thread information comprise:

- A graph indicating the statistics on the threads that switch execution between them and, in particular, in which position of their code this happens;
- The history of synchronization events (*e.g.*, wait, signal) between threads for various synchronization mechanisms (*e.g.*, semaphore, mutex, spinlock, signal).

Specific attention was put in the design of the buffers of tracing/profiling events so that the runtime overhead could be maintained low. In addition, we implemented the profiling/tracing tool so that it could work at various levels of details and, correspondingly, induce more or less overhead.

2.1 Gprof scheme

Gprof [10][11] allows two main kinds of profiling: the so-called *flat profile*, which aims to show the execution time spent in each function, and an "event count profiling", which can collect the function call graph and the source-line execution count. Profiling of the execution time is based on program counter sampling and, for this reason, it can be inaccurate in giving insight into "short" functions. For multi-threaded applications, the running time of each thread can be even less interesting because the overall execution time of the application can be highly affected by inter-thread phenomena (*e.g.*, synchronizations) and I/O blocking situations, which may spend time in kernel mode (not computed in the thread running time) or hidden by the execution of other threads/processes. For this reasons, time profiling, especially sampling-based, will not be addressed here and we will mainly focus on the *event count* profiling.

Gprof profiling works in three steps: (i) program instrumentation through a specific compilation, (ii) execution of the instrumented program to gather profile information, and (iii) post-processing/interpretation of profile data.

The instrumented compilation can be triggered by the *–pg* GCC [12] command line switch. Also an instrumented version of the *libc* can be used (flag *–lc_p*) to get profile information also from library calls. Otherwise, the possibly interesting events (*i.e.*, function invocation) occurring within the library would be completely hidden. In a similar way, profiling can be limited to only a sub-set of the application modules through a selective "*-pg* flag" compilation, achieving a reduced overhead and, as a consequence, the impossibility to investigate both non-profiled modules and events that cross the border between profiled and non-profiled code (*e.g.*, a function call).

During the execution of the instrumented program, profile information is gathered and, upon program termination, all information is written in a profile file.

The post-processing and analysis of collected information is done through the *gprof* tool. In Figure 2 a possible *gprof* output is shown: time-spent in each function, number of calls to each function and the call-graph can be derived. For example, in row *[3]*, *report* function is called one time and, in turn, calls eight times *timelocal*. *skipspace* is called eight times by *report* out of 16 overall invocations in the whole program (8/16).

```
index % time self children  called     name

[1] 100.0    0.00 0.05          start [1]
             0.00 0.05    1/1     main [2]
             0.00 0.00    1/2     on_exit [28]
             0.00 0.00    1/1     exit [59]
-----------------------------------------
             0.00 0.05    1/1     start [1]
[2] 100.0    0.00 0.05    1     main [2]
             0.00 0.05    1/1     report [3]
-----------------------------------------
             0.00 0.05    1/1     main [2]
[3] 100.0    0.00 0.05    1     report [3]
             0.00 0.03    8/8     timelocal [6]
             0.00 0.01    1/1     print [9]
             0.00 0.01    9/9     fgets [12]
             0.00 0.00   12/34    strncmp <cycle 1> [40]
             0.00 0.00    8/8     lookup [20]
             0.00 0.00    1/1     fopen [21]
             0.00 0.00    8/8     chewtime [24]
             0.00 0.00    8/16    skipspace [44]
```

Figure 2: *gprof* tool sample output. Time spent into each program function (self) and in the corrisponding children functions (called), along with the invocation count.

The instrumentation mechanism for functions (call graph and execution count) works at compile time (see Figure 1) and inserts a call to a profiling function (*mcount*) at the start of each function.

In this way, upon function invocation, *mcount* can increase the execution counter of the function and can update the call-graph retrieving the caller identity through the return address in the stack, and the callee identity through current program counter.

Source-line execution count is performed through the instrumentation of each basic-block with an increment instruction over a basic-block counter variable. Basic-block profiling can introduce a significant overhead and, by default, it is not performed along with function-level profiling.

Profile data is written into the output file by a specific profile function (*mcleanup*), which is called upon program termination.

2.2 Proposed scheme for multithreaded intra-thread profiling

In case of a multithreaded application, *gprof* is able to profile only the main thread as far as the *flat profile,* and causes the collection of only the profile, for functions and for the call graph, of the main thread. This happens because the profile library has only one profile structure per profiled process.

We have modified the profile library in order to support one structure per thread. This was done allocating one specific profile structure per thread, as soon as the first *mcount* function is invoked in each thread. The new thread-specific structure will be used for collecting statistics throughout all thread execution. In this way, a separated profile file is created for each thread. *pthread_self* function, which returns

```
Event              Time                 PID   Length Description
#####################################################################
Syscall exit     1042493319696474  N/A   7
Syscall entry    1042493319696492  N/A   12 SYSCALL : open; EIP : 0x0804A509
File system      1042493319696637  N/A   20 START I/O WAIT
Sched change     1042493319696696  1191  19 IN : 1191; OUT : 1215; STATE : 2
File system      1042493319696707  1191  20 SELECT : 3; TIMEOUT : 0
File system      1042493319696713  1191  20 SELECT : 4; TIMEOUT : 0
File system      1042493319696716  1191  20 SELECT : 6; TIMEOUT : 0
File system      1042493319696718  1191  20 SELECT : 7; TIMEOUT : 0
File system      1042493319696722  1191  20 SELECT : 9; TIMEOUT : 0
File system      1042493319696724  1191  20 SELECT : 11; TIMEOUT : 0
Memory           1042493319696737  1191  12 PAGE FREE ORDER : 0
Syscall exit     1042493319696745  1191  7
Syscall entry    1042493319696994  1191  12 SYSCALL : read; EIP : 0x0804D028
File system      1042493319696997  1191  20 READ : 11; COUNT : 4096
Syscall exit     1042493319697012  1191  7
Trap entry       1042493319697722  1191  13 TRAP : page fault; EIP : 0x4126C309
Trap exit        1042493319697729  1191  7
Syscall entry    1042493319698007  1191  12 SYSCALL:gettimeofday;EIP: 0x0804D028
Syscall exit     1042493319698010  1191  7
…
File system      1042493319722332  1037  20 SELECT : 17; TIMEOUT : 12000
Timer            1042493319722334  1037  17 SET TIMEOUT : 12000
Sched change     1042493319722337  1191  19 IN : 1191; OUT : 1037; STATE : 1
File system      1042493319722340  1191  20 SELECT : 3; TIMEOUT : 2
…
Syscall entry    1042493319722926  1191  12 SYSCALL : write; EIP : 0x0804D028
File system      1042493319722927  1191  20 WRITE : 3; COUNT : 8
Socket           1042493319722929  1191  16 SO SEND; TYPE : 1; SIZE : 8
Process          1042493319722935  1191  16 WAKEUP PID : 1037; STATE : 1
Syscall exit     1042493319722939  1191  7
Sched change     1042493319722942  1037  19 IN : 1037; OUT : 1191; STATE : 0
File system      1042493319722946  1037  20 SELECT : 1; TIMEOUT : 12000
```

Figure 3: portion of the output of our modified LTT. Various kernel events are recorded, along with the timestamp, PID and some additional description. Context switches (*Sched_change*), a page fault (*Trap entry*) and a socket IO operation (*Socket*) are highlighted.

the thread-id of the running thread, and thread-specific global data allowed to achieve a significant parallelism in the profiling activity, so that almost negligible synchronization overhead or synchronization probe effect was introduced on the application execution by the mechanism.

Another hidden problem in the *gprof* approach resides in the mechanism that writes the collected profile data to file. In fact, the standard *gprof* interface writes the data when the main thread ends. This is a problem because the main thread may end before all the children threads and thus cause only partial profile data to be written. In addition, a number of applications have their core execution in the children threads, while the main one is devoted only to initialization procedures.

We have modified this point, allowing each thread to write its own profile data upon its termination. In this way, profile data from one thread are available as soon as it terminates and the profile structures in memory can contain only the data of the running threads.

Print functions, and other auxiliary library functions used within *mcount* should be used without profiling, otherwise a possible infinite recursion could happen (*e.g.*, *mcount* calls *libFun1*, which is instrumented and therefore calls *mcount* again and so on). For this purpose, g*prof* uses a global disabling variable that is set at the start and reset at the end of *mcount* function. In our multi-threaded environment we replicate the disabling variable for each thread because, otherwise, a context-switch of thread-1 during *mcount* execution in disabled mode, would stop the profiling of all the threads that would execute before that thread-1 could execute again and re-enable the profiling.

The collected per-thread information comprises statistics for each function (*e.g.*, number of invocations), and for the relationship between functions (*e.g.*, call graph). *Gprof* native profiling interface is able to identify the identity of the just invoked function, from the program counter value before *mcount* invocation, and of its caller function, from the inspection of the return address on the stack frame. In addition, we have enabled the recoding of specific selectable events (*e.g.*, the execution of a basic-block), along with a temporal time-stamp, so that the history of these events could be available.

2.3 Proposed scheme for inter-thread profiling

To collect information on the relationship between the execution of the application threads, we had the need to know when a thread was de-scheduled and scheduled again and, if possible, to have some more insight into the reasons for a thread to be de-scheduled (*e.g.*, time quantum expired, blocking IO operation, blocking synchronization).

2.3.1 Kernel events profiling

In order to profile the OS activities, we have modified the Linux Trace Toolkit (LTT)[3], which is a powerful tracing/profiling tool for linux kernel. LTT is able to intercept and record a number of kernel events with a limited overhead [4] (*i.e.*, within 2.5%) and providing a good flexibility to be extended according to our needs.

We have inserted the activation of the LTT tracing inside the modified *gprof* initialization code, so that only the system activity occurring during our application execution is recorded. Accordingly, LTT tracing is stopped when the application profiling ends.

One of the modifications of the LTT tool included the addition of a timestamp to the collected events, in the same format as the records taken for thread execution. This timestamp allows establishing a relationship

between the "user" events collected within the threads and the events collected in the kernel, as well as between the events of the various threads. This feature allows reconstructing an overall picture of the whole execution of the multithreaded application, both for debugging and for profiling purposes.

Our overall profiling scheme relies on a number of parallel and largely independent profiling activities (*i.e.*, one per thread and one for the kernel), which collect data and end up writing a file with statistics and an event record. Then, ad-hoc post-processing and interpretation tools allow rebuilding the whole picture and analyzing the overall execution features of the application. We think that this approach can help reducing the profiling overhead and can be quite flexible to be further extended and adapted with extremely limited modifications to more complex environments like multi-core architectures. In particular, the approach is already suitable for a symmetric multi-processor architecture.

Figure 3 shows a sample of the events gathered from a LTT run. In particular, context switch events (*Sched_change*) allow tracing the segments of execution where each of the thread executes, along with its recorded events and statistics. Some of the information associated with the kernel events allows giving insight on the reason for the context switches. For instance, Figure 3 shows two context switches: the first one is due to the expiration of the time quantum of the scheduler, in fact a *Timer* event occurs immediately before the context switch. The second is due to the network IO operation issued by one process, probably to send data to the other one, which is immediately woken up and sent into execution.

2.3.2 Profiling of synchronization mechanisms

Even if LTT is able to record the basic synchronization events in the kernel (*Wait* and *Signal*), an insightful application profiling/tracing would necessitate the record of events which are directly linked to the various synchronization mechanisms used by the programmer: semaphores, mutexes, condition variables, spinlocks and signals.

We modified the primitives implementation of semaphores, mutexes, condition variables and spinlocks both for the main operations (*e.g.*, "lock", "unlock", "trylock"), and for the *destroy* operations. Each structure is identified by its address in the program. The destroy operations are crucial because allow to understand the *end of the scope* of a synchronization structure during the execution of the application and thus identify if the same physical structure is re-allocated afterwards, possibly with a different logical semantics.

As far as for the signals, for simplicity reasons we decided to record only the signal send events and not specifically the ones where the program configures the events which it is interested in. However, in this way it is possible to have into the recorded trace both the signals raised and, in an indirect fashion, the reaction of the receiving task.

The synchronization event (*synch-event*) records comprise the following fields: [*Syncr.Type*, *Structure-Id* (*address*), *Op.Type*], where *Syncr.Type* distinguishes between the synchronization mechanisms, *Structure-Id* between the instances of the same mechanism and *Op.Type* between the mechanism's operations. For signals, the *synch-event* is simpler: [*Syncr.Type*, *Sig.Num*], where *Syncr.Type* specifies the signal mechanism and the *Sig.Num* the number of the raised signal.

Synch-events are recorded inside the event trace of the thread that uses the synchronization mechanism, so that the ordered sequence of synchronization operations performed by each thread is known, as well as the ordered sequence of operations performed on every synchronization structure. In this way, it is possible to analyze the features of the interaction between parallel threads for performance/profiling purposes. Moreover, this information eases the debugging/testing of parallel applications through the inspection of the history of synchronization operations. This can be a very critical and important activity both in case of incorrect (debugging) and correct (testing and certification) executions.

2.4 Buffer design

The buffer structure is essentially based on the logical organization shown in Figure 4. The structure aims to be efficient from the performance point of view, allowing fast access from the running threads. A hash function operates on the thread-id and outputs the index of the array *A*, which holds the list *B* of the thread profile data that collide in the hash generation. A sequential search in the list *B* allows to find the correct thread structure *C*, which comprises both control fields and profile data.

A careful tuning of the size of array *A* allows limiting the possibility of hash collisions. This is a global structure shared between all threads but, apart from when a thread is created/destroyed, it needs only to be read by the parallel threads. For this reason, it does not induce significant synchronization overhead between threads.

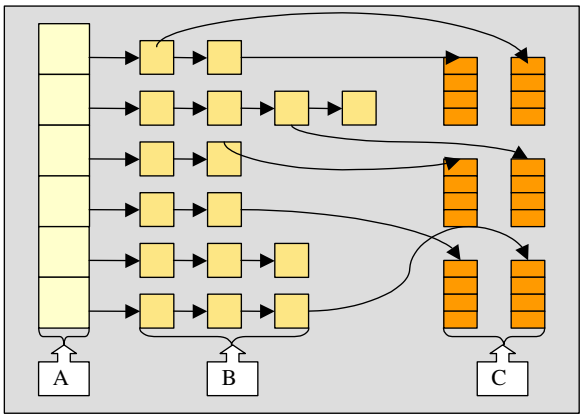

Figure 4: Logic organization of the tracing structure. A hash function elaborates the thread-id and outputs the index of array *A* where the profiling structure is. A sequential search is done in list *B* to find the correct structure among the ones with colliding hash. The structure *C* comprises control fields and profile data.

The main structure (array *A*) is statically allocated, while the thread-specific sub-structures (elements of list *B* and profile data *C*) are allocated by each thread as soon as in they first invoke *mcount* function.

We chose to let each thread write its profile information to disk as soon as its *pthread_exit* function is invoked. In this way, it is possible to overcome the limitations of the standard *gprof* scheme, which writes upon termination of the main thread. This choice allows freeing the profile structure from the thread data as soon as the thread terminates.

During profiling, if the gathered profile data fill up the available buffer, *mcount* manages this overflow condition and writes to disk (flush). This operation, when performed, can introduce a significant overhead in the thread execution because it can be relatively long. In addition, it can introduce a momentary significant probe-effect because it uses functions from the IO library and services from the IO subsystem of the kernel. However, the size of collected data and the rate of collection allow allocating relatively big structures, which permit to keep low the flush overhead in the overall thread execution. In the result section we will quantitatively analyze the overhead of the solution.

2.5 Usage of the solution

The setup steps for using our solution are essentially the following.

- Install of the modified LTT tool on the host Linux platform. It is almost identical to the original installation of LTT. This is needed only if interested in profiling kernel events.
- Substitution of the standard profile library (for *gprof* tool) with our improved one.
- Install of the profile-version of the libraries of interest: libc, libpthread, libm, etc. In general, all the libraries that have also the source code available can be easily recompiled through the standard and portable compilation command with profiling (*gcc –pg*). In this way, the code will be instrumented through the same mechanism used for *gprof* instrumentation but the profile behavior will be ours. Please note that it is always possible to mix not instrumented code with instrumented one, for instance in case of libraries whose source code is not available or when a part of the application is not interesting for profiling.
- Compile the application, or the interesting modules, with the profiling option (*gcc –pg*).
- Configure the profiling configuration file in the running directory to override the default options (*e.g.*, buffer sizes) and select the desired level of profiling. Possible choices include: *full* (application, synchronization events, full Kernel), *full-sched* (full profiling with only the scheduling kernel events), *applic-only* (application, synchronization, no kernel).
- Run the application.
- Analyze the results using the available programs for interpreting the output files. Per thread call graph, synchronization event history, number of blocking synchronization events, inter-thread call graph, per-thread wait-time, scheduling activity are some of the already deployed programs even if it is easy to implement new ones for analyzing different features of the execution.

3 RESULTS

We have tested the proposed solution on a real Linux workstation using some simple multithreaded benchmarks: 10 dining philosophers, a client-server service with 30 clients, vehicle simulation in a crossroad (10 cars), a processor farm. Then we have applied the technique to a more complex, real application: MySQL database manager. We performed also some load-test on the benckmarks, pushing on the problem size: 100 philosophers, 80 concurrent clients, 1000 cars on the crossroad, etc. The profiling buffers were chosen 10 millions elements wide (profile A) and 500 elements wide (profile B) in order to investigate two opposite situations: a target system that has plenty of spare RAM memory so that it can allocate tens of MByte of memory to the profiling buffers (profile A), and a more constrained system that can allocate only few hundreds Kbytes (profile B). Each element is able to collect a few basic events.

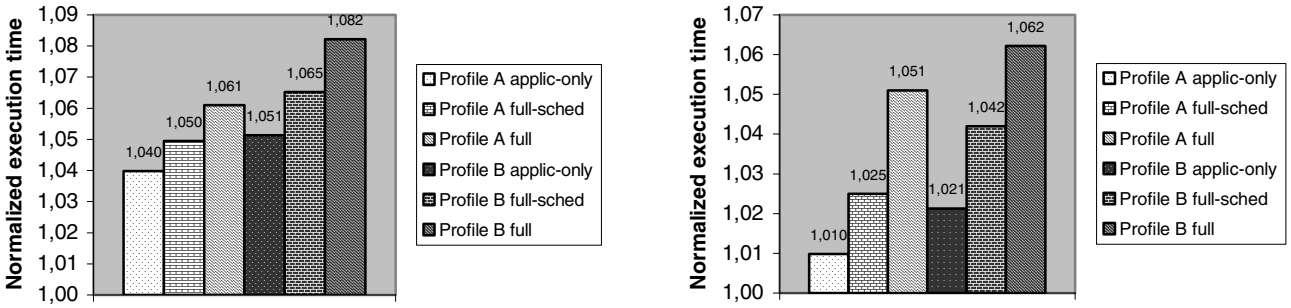

Figure 5: Client-server (left) and Dining Philosophers (right) overhead for 80 clients and 100 philosophers, respectively.

Figure 5 shows that the runtime overhead for simple benchmarks, even in high load conditions, is very limited: from 5% to less than 9% for the *full* profile. Using less detailed collection, profiling of the sole OS scheduler or no OS profiling at all, the overhead is even smaller. This enables a low intrusiveness profiling of multithreaded applications. In addition, Figure 5 shows also a limited difference in the overall overhead caused by the usage of big and small profiling buffers (*i.e.*, profile A and profile B, respectively), even if, depending on the application, the slowdown of profile B over profile A can be almost double as the slowdown of profile A over the original application (see Figure 5-*right*).

Then, we analyzed some runs of the MySQL database server in order to test the capabilities of the proposed profiling framework on a real-world multithreaded application. Some existing MySQL self-testing procedures were used both for testing that the instrumentation would not introduce any bug in the application, and for the performance analysis.

Figure 6 shows that the overhead of MySQL can be significantly higher (up to 4x slowdown) than in case of simpler benchmarks, especially in some use cases (*e.g.*, test-check procedure). However, the absolute value is small enough to enable the runtime profiling of real-executions of a multithreaded application, without introducing a dramatic slowdown. In this way, many applications can be profiled in their real environment without the need of, and the probe error due to, an artificial test conditions.

In addition, Figure 6 confirms that the full profiling is not tremendously slower than the application-only one (about 4x instead of 3-3.3x for test-check, about 1.3 instead of 1.01-1.04x for test-lock, and 2.1-2.5x instead of 1.4-1.5x for test-myisam). Finally the figure shows that even in a complex application, the overhead difference between small (*profile B*) and huge (*profile A*) profile buffers is quite small. This is probably due to the combined effect of the features of our buffer (organization of the structure, independence

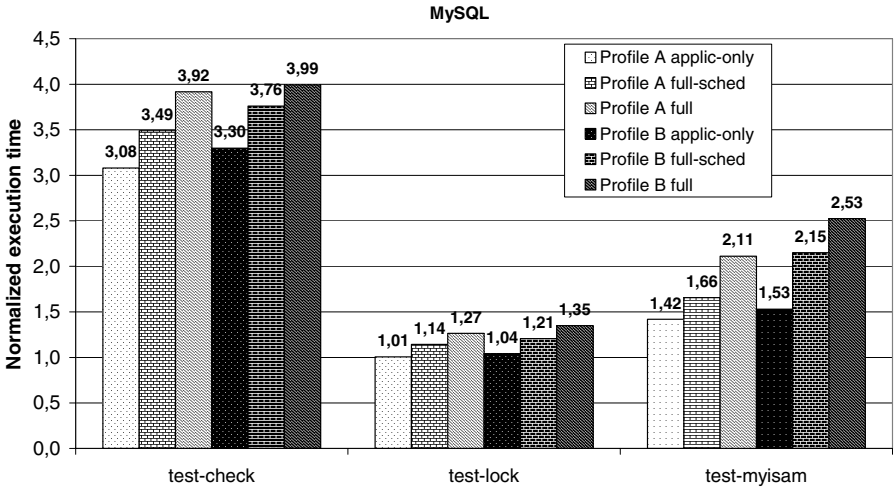

Figure 6: profiling overhead of three MySQL self-test procedures (test-check, test-lock and test-myisam). The overall slowdown depends on the particular procedure and, in the worst analysed case, was less than 4x, with a limited difference between application-only profling (3x) and full profiling. On average, small buffers (proflie B) do not introduce much more overhead than huge ones (profile A).

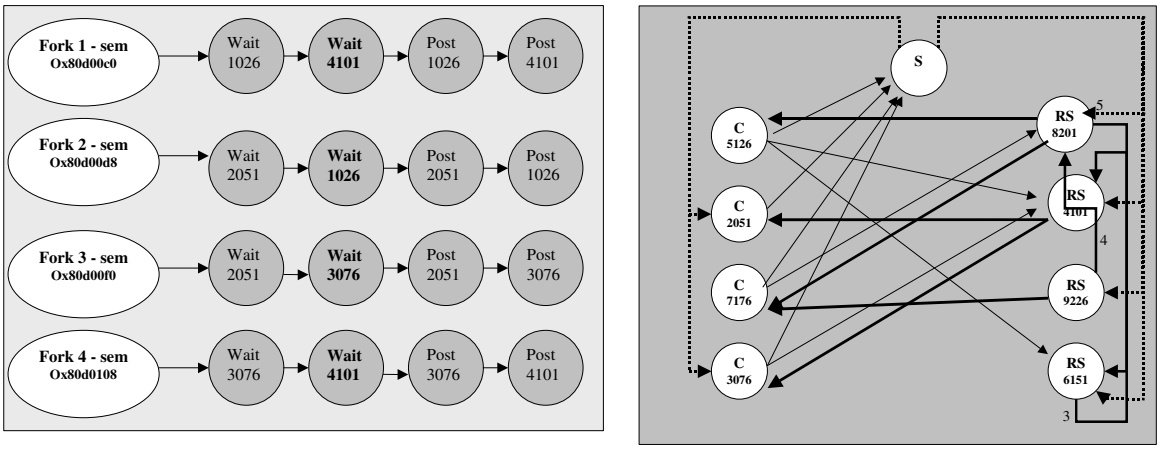

Figure 7: (*left*) usage pattern of the synchronization structures (semaphores) which protect the critical sections for the usage of each fork by the *dining philosophers*. Each semaphore has an Id (its memory address) and a list of actions on it: operation and thread-Id of the operator. (*right*) Inter-thread relationship (scheduling or descheduling) between the threads of the client-server test. Clients are *C*, main server is *S* and dedicated servers are *RS*.

of the various profiling/tracing activities in the threads and in the O.S.) and the filesystem buffering of the host operating system.

As a whole, these result show that the runtime overhead of the proposed mechanism is very limited, considering that it is enough to gather complete information on the parallel behavior of the application. This is mainly due to two reasons. First, the granularity of the gathered information is quite coarse (function-level). Second, the timestamping of events occurring in independent profile activities (threads, O.S.) allows for low-overhead during execution, but enable the offline reconstruction of high-level overall information.

In the following we sketch some simple examples of the information collectable from the proposed approach.

In Figure 7-*left*, the history of the synchronization events occurred per synchronization structure allows to analyze the usage patterns (and, if needed, the correctness) of synchronization operations between threads. In particular, the figure shows the sequence of synch-operations (*e.g.*, wait and post) and the thread-Id that performed the operation itself. Using the timestamp of these events, it is also possible to reconstruct the precedence graph of parallel execution of the involved threads.

Figure 7-*right* shows the inter-thread graph of a client-server application, which shows which threads (nodes) tend to suspend their execution (*e.g.*, due to a blocking) for having which other (arch) thread go into execution. For instance, this could be useful to highlight bottlenecks in the parallel application due to unbalanced software design and/or unusual use cases that stress the software structure.

Table 2 gives an example of other profile information that are useful for investigating the synchronization behavior of the application. The table shows the number of times that an application-level synchronization function was called by each thread and, in case of a possible blocking function (*e.g.*, wait), the number of times that it actually blocked the thread. For instance, the thread with 2051 Id called three times the M*utex-lock* function and one time it actually was blocking.

Table 1 further digs into the synchronization behavior of the application and shows the non-running time percentage of each thread, *i.e.* the thread life-time fraction in which it did not execute, and the quote of this

Table 2: Usage of the synchronization structures is a sample run of the *dining philosophers*. For each thread and for each sync-function, the total and the blocking execution count is shown.

Function/Thread	4101	1026	2051	3076
MutexLock	3/0	3/0	3/1	3/0
MutexUnLock	3/-	3/-	3/-	3/-
Wait	2/2	2/1	2/0	2/1
Post	2/-	2/-	2/-	2/-
Lock Destroy	1/-	1/-	1/-	1/-

Table 1: sample run of the *dining philosophers*. For each thread, the percentage of execution time in which it does not execute is shown, as well as the subfraction in which it is blocked on a shared resource.

Thread_id	Not-running time (%)	Not-running time (blocked) (%)
1026	72	30
2051	63	20
3076	71	31
4101	75	37

time that is due to a blocking condition on a synchronization structure. For instance, the thread with the 1026 Id was not executing for 72% of its lifetime, but only 30% of the time due to a blocking state. The remaining quote is due to its stay in the OS ready queue while other threads execute.

References

[1] J.K.Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX Conference, 1996.

[2] A. Srivastava, A. Eustace, "ATOM: a system for building customized program analysis tools". In Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (Orlando, Florida, United States, June 20 - 24, 1994). PLDI '94. ACM Press, 196-205.

[3] Linux Trace Toolkit (LTT), http://www.opersys.com/LTT/index.html

[4] K. Yaghmour, M.R. Dagenais, "Measuring and Characterizing System Behavior Using Kernel-Level Event Logging", Proc. 2000 USENIX Annual Technical Conference, 2000.

[5] S. McFarling, "Procedure Merging with Instruction Caches". Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 1991, 71-79.

[6] K. Pettis, R.C. Hansen, "Profile Guided Code Positioning". Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, June 1990, pp.16-27.

[7] A. Hashemi, D.R. Kaeli, B. Calder, "Efficient Procedure Mapping using Cache Line Coloring". Proceedings of the Int. Conference on Programming Language Design and Implementation, June 1997, 171-182.

[8] J. Kalamatianos, A. Khalafi, D.R. Kaeli, W. Meleis, "Analysis of Temporal-Based Program Behavior for Improved Instruction Cache Performance". IEEE Transactions on Computers, Vol. 48, No. 2, 168-175.

[9] S. Bartolini, C.A. Prete, "Optimizing instruction cache performance of embedded systems". Trans. on Embedded Computing Sys. 4, 4 (Nov. 2005), 934-965.

[10] S.Graham, P.Kessler, M.McKusick, "gprof:A Call Graph Execution Profiler", Proceedings of SIGPLAN '82 Symposium on compiler Construction, SIGPLAN Notices, Vol. 17, No 6, pp. 120-126, June 1982.

[11] GPROF tool, http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html.

[12] GNU GCC Compiler Collection, http://gcc.gnu.org/.