IDENTIFYING PERFORMANCE BOTTLENECKS IN WORK-STEALING COMPUTATIONS

Nathan R. Tallent and John M. Mellor-Crummey, *Rice University*

Work stealing is an effective load-balancing strategy for multithreading, but when computations based on it underperform, traditional tools can't explain why. To resolve a computation's key performance obstacles, tools must pinpoint and quantify parallel idleness and overhead.

With the increase in clock frequencies over the past several years, power dissipation has become a substantial problem for microprocessor architectures. In response, the microprocessor industry has shifted its focus to designs with a higher processor core count. For software to benefit from this shift, it must exploit threaded parallelism, which in turn requires programming models that will facilitate the development of efficient multithreaded programs.

However, even with new programming models, many shared-memory algorithms that scale to eight cores are unlikely to scale to next-generation machines with scores of threaded cores. Because many scaling problems are hard to diagnose, programmer productivity demands helpful performance tools. However, the most promising multithreaded programming models have complex runtime characteristics that prevent traditional performance analysis strategies from pinpointing bottlenecks.

Recognizing the critical need for new performance analysis methods and tools, we have developed a profiling strategy for identifying bottlenecks in programs that use work stealing—a powerful, practical, and influential scheduling model for dynamically mapping multithreaded computations onto multicore processors. Although work stealing provides good load balancing, there is currently no way to pinpoint scalability bottlenecks or provide user-level insight into their causes.

Our strategy, implemented in HPCToolkit (http://hpctoolkit.org), addresses both these deficiencies.[1] It quantifies parallel idleness (when threads wait for work) and overhead (when threads work on non-user code). It then pinpoints regions of the user's application that need more or less concurrency (to reduce idleness or overhead, respectively) and that employ hopeless parallelization (because both idleness and overhead are high). In contrast, existing profilers compute less insightful metrics and fail to distinguish between source code and the scheduler. Finally, our strategy incurs no measurement overhead beyond that for normal sampling-based profiling (typically 1 to 5 percent).

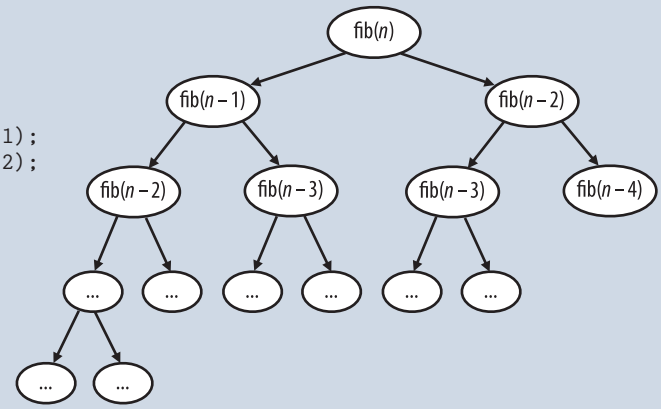
To gauge our strategy's effectiveness, we analyzed the performance of a program for Cholesky decom-

 Published by the IEEE Computer Society

position written in Cilk, a multithreaded language based on work stealing. Without any insight into the source code, we were able to make strong, precise statements about the program's parallel efficiency. Moreover, our method applies to any work-scheduling implementation and to other multithreaded programming models such Cilk++, OpenMP, and Intel's Threading Building Blocks.

```
cilk int fib (n) {
  if (n < 2)  return n;
  else {
    int x, y;
    x = spawn fib (n - 1);
    y = spawn fib (n - 2);
    sync;
    return (x + y);
  }
}
```

(a)



(b)

Figure 1. Example of Cilk's simplicity and expressiveness. A successful parallel language for multicore applications must be able to express complex patterns of parallelism relatively easily. In (a), asynchronous calls (spawns) in a simple Cilk program to compute the *n*th Fibonacci number create logical tasks that block only at a sync, which (b) quickly creates significant logical parallelism.

MULTICORE PROGRAMMING LANGUAGES

To become widely adopted, a parallel programming language must have four key properties. First, expressing parallelism should be simple. Second, the language must be expressive enough to easily combine different parallel programming models. Many large-scale parallel programs are based on data parallelism, in which the same computation is mapped across many data elements. Although this model dominates high-performance computing, many other applications contain both data and task parallelism, and in irregular ways. Third, the language must make it possible to exploit parallel resources efficiently. Finally, the language should ensure against future architectural changes by transparently scaling to increasing core counts.

From its conception, Cilk possessed these four properties.[2] It has proven influential, spawning a commercial counterpart, Cilk++, and serving as an exemplar for Intel's Threading Building Blocks and Microsoft's Concurrency Runtime, as well as for ongoing research projects.

Cilk, an extension of C, provides two keywords for expressing parallelism. A *spawn* transforms a sequential (blocking) function call into an asynchronous (nonblocking) call. A *sync* blocks a function's execution until all its spawned children have completed.

Figure 1a shows a sample Cilk program for computing the *n*th Fibonacci number. The function computes fib(n) as the sum of fib(n – 1) and fib(n – 2). (This program is for illustration only; there are more efficient algorithms for this computation.) Because the recursive calls to fib are spawned, Cilk's runtime system can execute them in parallel. Because the expression (x + y) depends on the results of both calls, the *sync* ensures that both have completed before the addition begins.

In Figure 1b, which represents the computation's (simplified) logical parallelism, the spawns and syncs form a tree of dependencies, in which each interior node (not a leaf) depends directly on its two children. The tree is slightly unbalanced to indicate that more work is on the left side than on the right.

The Cilk runtime system must efficiently map logically independent calls onto computational cores. Each asynchronous call can be thought of as a lightweight thread, or task. The Cilk work-stealing model combines lazy task creation with a work-stealing scheduler. To execute the program, the runtime system creates a pool of worker threads at the operating-system level, one per available core. Then, as Figure 2 shows, the first worker thread begins executing the first task. If no other worker threads are in the pool, program execution continues sequentially, without any additional task creation. Whenever the thread pool contains an idle worker, that worker attempts to steal a task from a working thread.

The Cilk work-stealing model is attractive because, although a spawn identifies an independent task, the overhead of assigning this work to a separate thread occurs only when a worker thread is idle. Moreover, as long as worker threads execute enough spawns, work stealing will naturally achieve very good load balance. Both these features mean that, as long as the program has sufficient logical parallelism, the same Cilk program can execute efficiently on one core or many.

THE CHALLENGE OF ANALYZING PERFORMANCE

If a Cilk program is slow and does not scale well, the Cilk compiler can insert instrumentation to compute an abstract measure of the program's critical path as well

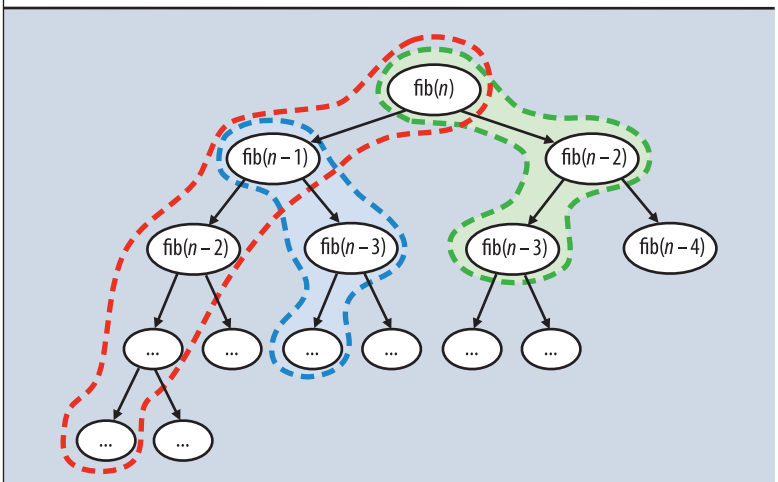


Figure 2. Scheduling work through work stealing. Execution of the Cilk program in Figure 1a begins with the Cilk runtime system assigning the whole computation to worker thread 1 (red). This worker starts elaborating the call tree in a depth-first order and continues down the far-left branch, as in a serial execution. An idle thread, worker thread 2 (green), steals the continuation associated with fib(*n*), which promptly spawns a second asynchronous call to compute fib(*n* – 2). A second idle thread, worker thread 3 (blue), has two threads to steal from. It randomly chooses to steal from thread 1 and then selects the next piece of available work, the continuation associated with fib(*n* – 1), a call to fib(*n* – 3).

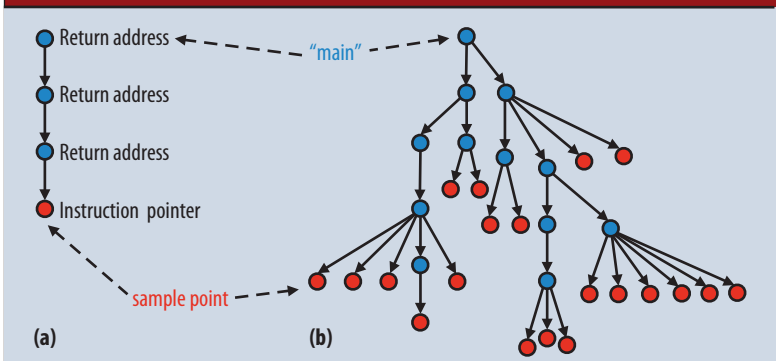


Figure 3. A sampling call path profiler. The profiler initializes a timer or hardware counter that generates a signal when it expires or overflows. For each sample event, the profiler gathers the profiled application's calling context using stack unwinding, yielding (a) a call path sample represented as a list of instruction pointers, with the leaf being the sample point. A collection of samples naturally forms (b) a calling context tree whose root is the program's entry point.

as its degree of parallelism relative to that path.[3] These metrics are attractive because they are largely machine independent, and the measure of parallelism provides predictive insight for scaling the application to a system with more cores.

But what if the aim is to *resolve* a scaling bottleneck? Where in the source code does it arise? How severe is it? Unfortunately, Cilk's metrics provide little insight into a problem's severity, location, or resolution. Moreover, abstract metrics are a double-edged sword. Although they approximate the performance of a machine-independent model, they obscure important architectural details, such as memory-system performance, which might be critical to an application's actual performance. In addition, the instrumentation to compute Cilk's metrics is costly, significantly dilating execution time.

To pinpoint performance bottlenecks in a program based on work stealing, performance metrics must reflect the quality of the program's parallelism. A parallel program executes efficiently when the granularity of parallel tasks is small enough to keep every core busy, but large enough to avoid needless task management. Thus, two aspects of parallelism are particularly important for efficiency: whether the program contains enough parallelism to occupy all the processor cores and whether that parallelism is sufficiently coarse grained so that the cost of managing it is not large relative to the cost of performing the program's actual work.

Additionally, because most programs are written in a modular coding style, to identify performance bottlenecks, performance metrics must be attributed to the calling context in which they occur. Unfortunately, work stealing makes this very difficult.

THE CHALLENGE OF ATTRIBUTING METRICS

Call path profilers attribute metrics to calling contexts. To achieve low overhead, sampling-based call path profilers use statistical sampling rather than instrumentation. Figure 3 shows how such call path profilers work. The key advantage of sampling over instrumentation is that sampling overhead is proportional to sampling frequency, not call frequency. Moreover, sampling naturally elides unimportant data; if a region of the profiled program receives no samples, that region's cost is negligible.

Cilk's work-stealing scheduler renders even sophisticated call path profilers useless. To understand why, consider the calling contexts in Figure 4, which are derived from Figure 2. What happens if worker thread 3 receives a sample? Because that thread began its execution with a steal, its stack of native procedure frames represents only a suffix of the full calling context. In fact, the rest of thread 3's context is separated in both space and time—space because thread 1 contains its parent context, and time because thread 1 continues executing rather than blocking and waiting for thread 3 to complete the asynchronous call. As execution progresses, call paths can become even more

fragmented because procedure frames migrate between threads during steals. Consequently, standard call path profiling of a Cilk program yields a result that is at best cumbersome and at worst incomprehensible. For effective performance analysis, it is important to bridge the gap between user-level abstractions and their realization at runtime by attributing costs to their full logical calling context. We call this logical call path profiling.

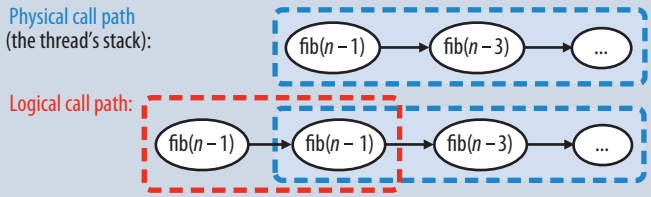


Figure 4. A case for logical call path profiling. Suppose that thread 3 (blue) from the example in Figure 2 receives a sample. Because that thread began its execution with a steal, sampling cannot capture the full context. The rest of the context is separated in space and time. Logical call path profiling attributes metrics to their full logical calling context.

QUANTIFYING INSUFFICIENT PARALLELISM

The total computational effort of a program is the sum of work and idleness, where "idleness" is time that threads wait for "work." To quantify insufficient parallelism, our method directly and efficiently measures and attributes the idleness component of an execution using logical call path profiling.

Cilk's work-stealing scheduler creates one worker thread per core. When a sample event occurs during profiling, each thread receives an asynchronous signal. Worker threads are either working or idle. If a worker thread is idle, it is spinning within a scheduling loop waiting for another thread to create a task that it can steal. A standard call path profiler attributes samples on the basis of first-person knowledge of what a thread itself is doing. Consequently, working threads accumulate samples where they work, but idle threads accumulate samples in the scheduling loop.

Although this method quantifies parallel idleness by classifying samples received within the scheduler as idleness, the results are not actionable because they do not pinpoint the cause of idleness. To identify cause, an idle thread must also have third-person knowledge about which other threads are responsible for its own idleness. In a work-stealing computation, a thread is idle precisely because other threads have no extra tasks available to steal. Therefore, when a thread is idle, the current working threads are culpable because they are not generating enough logical parallelism to keep all threads busy. Thus, our method changes how samples are attributed for idle threads by forming an idleness metric that blames actively working threads for not spawning enough tasks to keep all workers busy.

To compute idleness, we first adjusted the Cilk scheduler to always maintain the number of working threads, n_w, and idle threads, $n_{\overline{w}}$. To do this, the scheduler simply maintains a nodewide counter to represent n_w. When a thread begins a task, it atomically increments n_w. When a thread completes its current task, it atomically decrements n_w to indicate that it is no longer working. Then, at any given time, $n_{\overline{w}} = n - n_w$, where n is the number of worker threads.

We also modified our sampling strategy. If a sample event occurs in a thread that is not working, the thread ignores it. When a sample event occurs in a working thread, the thread attributes one sample unit to the work metric for its sample context. The thread then obtains n_w and $n_{\overline{w}}$ and attributes a fractional sample $n_{\overline{w}}/n_w$ to the idleness metric for the sample context. Even though the thread itself is not idle, it is critical to understand what work that thread is performing while other threads are idle. Our strategy charges the thread its proportional responsibility for not keeping the idle processors busy at that moment at that point in the program.

As an example, consider a sample of a Cilk execution, in which five threads are working and three threads are idle. Each working thread records one sample of work in its work metric, and a 3/5 sample of idleness in its idleness metric. The total amount of work and idleness charged for the sample is thus 5 and 3, respectively.

QUANTIFYING PARALLELIZATION OVERHEAD

Once we had quantified parallel idleness, we wanted to measure parallel overhead. We had already defined a program's effort as the sum of work and idleness, where work is nonidle time, but to measure parallel overhead, we had to refine the work metric to distinguish useful work from parallel overhead. Thus, a program's work becomes the sum of useful work and overhead, where overhead is the time spent executing something other than the user's computation. Sources of parallel overhead include task synchronization and bookkeeping operations to prepare tasks for the possibility of being stolen.

The key challenge is that on a sample event, a nonidling thread must be able to efficiently determine whether to charge the sample to the overhead or useful-work metric. Because at the binary level overhead instructions are intermingled with useful-work instructions, the two categories are indistinguishable without prior arrangement. A compiler could insert instrumentation, but that would be extremely costly.

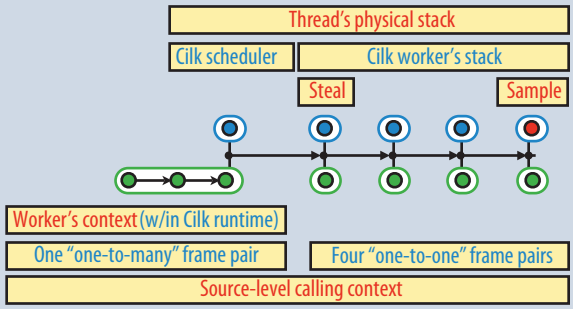


Figure 5. Recovering the logical context of a Cilk worker thread. The logical call path has five pairs, with the outermost frame at the left. For each pair, source-level frames are on the bottom (green nodes) and native frames (red and blue nodes) are on the top. Thus, the top frames represent the native frames of a worker thread's stack. The outermost native frame represents Cilk's scheduler loop, and the next native frame represents a steal point. Because of the steal point, the outermost native frame corresponds to several source-level frames that represent the context of the steal. In contrast, each native frame after the steal point corresponds to only one source-level frame.

The key insight is this: Because distinguishing overhead instructions from useful-work instructions can be done with static analysis, a performance tool can actually create an overhead metric postmortem. For example, we modified the Cilk compiler to specially tag instructions that are associated with parallelization overhead. These tags could take several forms, but a particularly convenient one associates overhead instructions with special file or procedure names within the binary's debugging information. A postmortem analysis tool recovers the compiler-recorded tags, identifies instructions associated with overhead, and attributes any samples of work associated with them to the overhead metric.

Tags have several attractive properties. Because they are only metainformation, they can be created and used without affecting runtime performance in *any* way. Although tags consume space, they need not be loaded into memory at runtime. In addition, tags can be refined to partition overhead sources into several types, providing more detailed information to users or analysis tools.

ATTRIBUTION WITH LOGICAL CALL PATH PROFILING

To attribute parallel idleness and overhead metrics to full calling contexts, we developed logical call path profiling to bridge the gap between Cilk's source-level calling contexts and their realization at runtime when load is being dynamically rebalanced through work stealing.

As Figure 4 shows, mapping measurements during execution back to a source program requires reassembling source-level contexts, which have been fragmented during execution. Figure 5 gives a simplified example of recovering the logical context of a Cilk worker thread. A logical call path is a list of pairs in which each pair consists of a list of source-level frames and their corresponding realization as native frames.

Logical unwinding is applicable to other parallel and serial languages. Because the details of mapping native to source-level frames are different for each language implementation, our profiling strategy provides a general application programming interface for obtaining logical unwinds given a language-specific plug-in.

IDENTIFYING PARALLEL BOTTLENECKS

Collecting parallel idleness and overhead metrics and attributing them to their logical calling context yields actionable insight that helps a developer resolve bottlenecks. With information about parallel idleness and overhead attributed hierarchically over loops, procedures, and the calling contexts of a program, it is possible to directly assess parallel efficiency and provide guidance about how to improve it. (Using a postmortem binary analysis, our tools can attribute metrics to loops without any profiling overhead.)

If a region of the program (such as a parallel loop) is attributed with high idleness and low overhead, a decrease in the parallelism's granularity could enhance parallel efficiency. If the overhead is high and the idleness low, a granularity increase could reduce overhead. If the overhead is high and there is still insufficient parallelism, the parallelism is inefficient and no granularity adjustment will help. In this case, keeping the idle processors busy requires a different parallelization: for example, a combination of data and functional parallelism instead of only one or the other.

It is also necessary to consider efficiency on an individual core. With logical call path profiling, a profiler can associate hardware-performance-counter-based metrics to Cilk program contexts. Using hardware-performance counters, for example, enables the computation of memory bandwidth or memory latency for all loops in a program—in their full calling context.

ANALYZING CHOLESKY DECOMPOSITION

To demonstrate the power of attributing work, parallel idleness, and parallel overhead to logical call path profiles, we used our method to analyze the performance of the Cholesky program from the Cilk 5.4.6 source distribution. We profiled a problem size of 3000 × 3000 (30,000 nonzeros) on a symmetric multiprocessor with dual quad-core AMD Opterons (2360 SE, 2.5 GHz) and a 4-Gbyte main memory.

The big picture

In Figure 6, our presentation tool is displaying the calling context view of the aggregated results. The screenshot

has three main components. The *navigation pane* shows a top-down view of the calling context tree, partially expanded. The pane contains several source-level procedure instances along the call paths. (Physical procedure instances are not shown.) The *source pane* shows the procedure cholesky, which corresponds to the selected line in the navigation pane. Each entry in the navigation pane is associated with metric values. In the idleness and overhead columns, the values in scientific notation represent idleness and overhead as percentages of total effort; the values shown as percentages to their right give an entry's proportion of the total idleness or overhead, respectively. The "I" qualifier at the top of the column denotes that the metrics are inclusive—they represent values for the associated procedure instance in addition to all its callees. Thus, the metric name "work (all/I)" means inclusive work summed over all threads. The viewer sorts sibling entries with respect to the selected metric column, which in the figure is "work (all/I)."

At the bottom of the navigation pane is a loop within the context of `cilk_main`. The navigation pane actually contains a fusion of the dynamic logical calling contexts and static loop contexts.

To obtain an overview of the program's performance, the user can expand the calling context tree to the first call of cholesky and then note the metrics on the right. As Figure 6 shows, 50.7 percent of the total work is spent in the top-level call to cholesky; the top-level call to `mul_and_subT` (which verifies the factorization) is a close second at about 47 percent. It is also immediately apparent that 19.9 percent of the total parallel idleness and 54.7 percent of the total overhead occur in cholesky. However, because this idleness and overhead are only about 2.45 and 1.28 percent of the total effort, respectively, the parallelization of cholesky is very effective for this execution. In contrast, the parallelization of the entire program (using `cilk_main` as a proxy) is less effective, with overhead accounting for a relatively low 2.33 percent, but idleness increasing to 12.1 percent of the total effort.

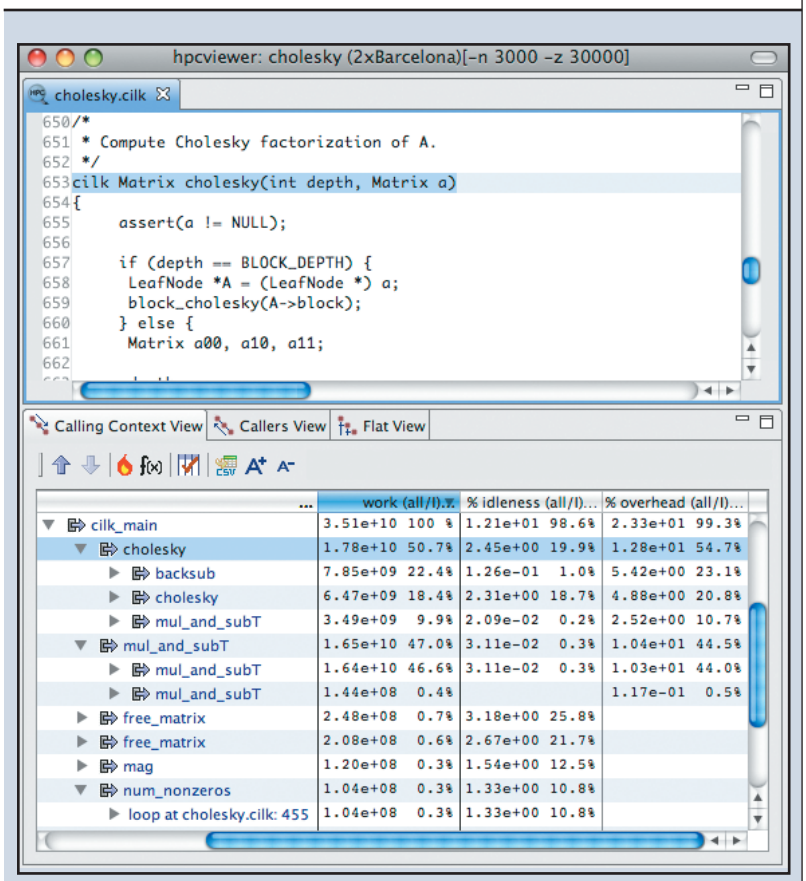


Figure 6. A calling context view of the Cholesky program. The selected line in the navigation pane and the source pane (top) show the cholesky procedure. The navigation pane (bottom left) shows a top-down view of the calling context tree, partially expanded. The metric columns (bottom right) show summed values over the eight worker threads for work (in cycles), parallel idleness, and parallel overhead (yielding the "all" qualifier in their names). Both idleness and overhead are shown as percentages of total effort, where effort is the sum of work, idleness, and overhead.

The details

To pinpoint exactly where inefficiency occurs, the callers view is often useful because it looks up from a procedure to the procedures that call it. Thus, at the first level, the callers view lists all the program's procedures. If multiple instances of a given procedure appear in the calling context view, the callers view aggregates those instances.

In Figure 7, the navigation pane is sorted according to relative idleness, the most troubling inefficiency. Unlike the metrics in the calling context view, these metrics have exclusive ("E") values because they do not include values for a procedure's callees. The top two routines in the list are versions of `free`, a C library routine. Together they account for about 35.8 percent (20.8 percent + 15.0 percent) of the program's idleness. When the callers for these routines are expanded, it is evident that both are called by `free_matrix`, a serial helper routine that deallocates the matrix for the Cholesky driver. Moving down the list reveals that every routine shown in the screenshot is a serial helper. Because each of these serial routines except `block_schur_full` is related to initialization or finalization, the user can see immediately that reducing the fraction of parallel idleness requires either a larger matrix or parallelization of the initialization and finalization routines.

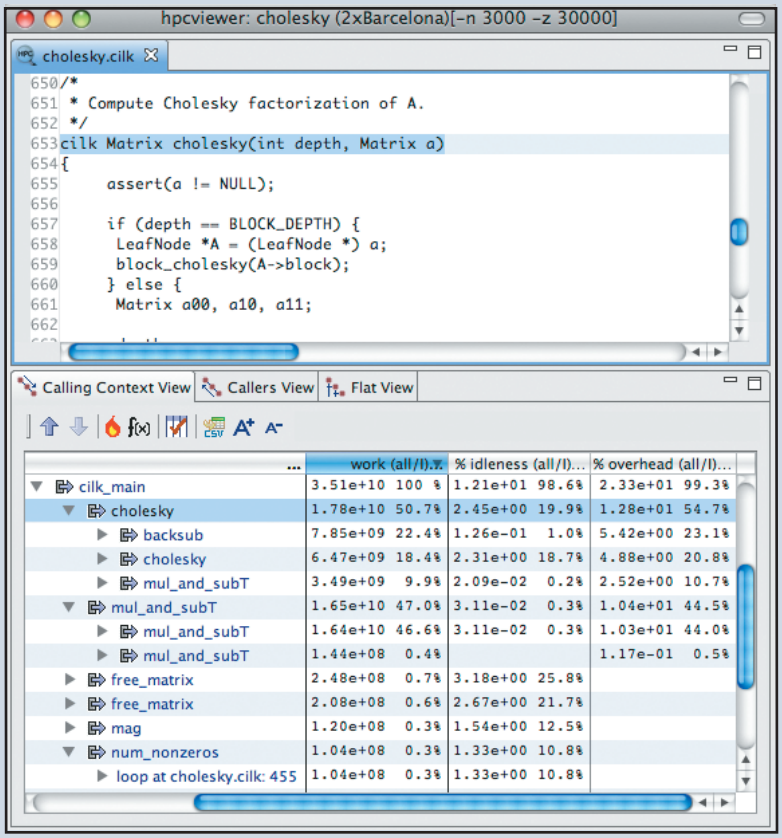


Figure 7. A callers view of the Cholesky program. This view is a bottom-up look at the call chain, which reveals the source of the inefficiency. The top pane shows a list of procedures rank-ordered according to the metric of interest, in this case, relative idleness. The bottom pane shows an expanded view of the callers for these routines. The expanded view shows instantly that `free_matrix`, a serial routine, is calling the top two routines.

In hindsight, it is hardly surprising that serial code was responsible for idleness. What *is* surprising is that without any prior knowledge of the program, we could immediately pinpoint serial code and quantify its impact on parallel efficiency.

The growing need to develop applications for multicore architectures makes tools that can pinpoint and quantify performance bottlenecks in multithreaded applications absolutely essential. Such tools will become critical as less-skilled application developers are forced to write parallel programs to benefit from increasing core counts in emerging processors.

Our method demonstrates that attributing work, parallel idleness, and parallel overhead to logical calling contexts quickly provides insight into the runtime performance of a Cilk program. Although we have focused on Cilk, our techniques can be applied to other work-stealing-based languages. Our approach is effective because it relies on metrics that are completely intuitive and can be mapped back to the user's programming abstractions, even though the runtime realization of these abstractions is significantly different. We have also shown that an effective approach can be highly efficient: The runtime cost of our profiling can be made arbitrarily low by reducing the sampling frequency. Finally, we have shown that it is possible to collect implementation-level measurements and project detailed metrics to a much higher level of abstraction without compromising their accuracy or utility.

References

1. N.R. Tallent and J. Mellor-Crummey, "Effective Performance Measurement and Analysis of Multithreaded Applications," *Proc. Symp. Principles and Practice of Parallel Programming* (SIGPLAN 09), ACM Press, 2009, pp. 229-240.
2. M. Frigo, C.E. Leiserson, and K.H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," *Proc. Conf. Programming Language Design and Implementation* (SIGPLAN 98), ACM Press, 1998, pp. 212-223.
3. Supercomputing Technologies Group, MIT Laboratory for Computer Science, *Cilk Reference Manual*; http://supertech.csail.mit.edu/cilk.

Nathan R. Tallent *is a PhD candidate in the Department of Computer Science at Rice University. His research interests include the performance measurement, attribution, analysis, and interpretation of parallel programs. He received an MS in computer science from Rice University. Contact him at tallent@rice.edu.*

John M. Mellor-Crummey *is a professor in the Departments of Computer Science and Electrical and Computer Engineering at Rice University. His research interests include compilers, tools, and runtime libraries for multicore processors and scalable parallel systems. He received a PhD in computer science from the University of Rochester. He is a member of the IEEE Computer Society and the ACM. Contact him at johnmc@rice.edu.*