A New Approach for Performance Analysis of OpenMP Programs

Xu Liu
Rice University
Dept. of Computer Science
MS 132
P.O. Box 1892, Houston, TX
77251-1892
xl10@rice.edu

John Mellor-Crummey
Rice University
Dept. of Computer Science
MS 132
P.O. Box 1892, Houston, TX
77251-1892
johnmc@rice.edu

Michael Fagan
Rice University
Dept. of Computer Science
MS 132
P.O. Box 1892, Houston, TX
77251-1892
mfagan@rice.edu

ABSTRACT

The number of hardware threads is growing with each new generation of multicore chips; thus, one must effectively use threads to fully exploit emerging processors. OpenMP is a popular directive-based programming model that helps programmers exploit thread-level parallelism. In this paper, we describe the design and implementation of a novel performance tool for OpenMP. Our tool distinguishes itself from existing OpenMP performance tools in two principal ways. First, we develop a measurement methodology that attributes blame for work and inefficiency back to program contexts. We show how to integrate prior work on measurement methodologies that employ directed and undirected blame shifting and extend the approach to support dynamic thread-level parallelism in both time-shared and dedicated environments. Second, we develop a novel deferred context resolution method that supports online attribution of performance metrics to full calling contexts within an OpenMP program execution. This approach enables us to collect compact call path profiles for OpenMP program executions without the need for traces. Support for our approach is an integral part of an emerging standard performance tool application programming interface for OpenMP. We demonstrate the effectiveness of our approach by applying our tool to analyze four well-known application benchmarks that cover the spectrum of OpenMP features. In case studies with these benchmarks, insights from our tool helped us significantly improve the performance of these codes.

Categories and Subject Descriptors

C.4 [**Performance of systems**]: Measurement techniques, Performance attributes; D.2.8 [**Metrics**]: Performance measures.

Keywords

OpenMP, performance measurement, performance analysis, software tools.

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

1. INTRODUCTION

Developing multithreaded programs for shared memory systems can be tedious, especially if one directly uses a primitive threading library. To simplify development of multithreaded programs, the high performance computing community has developed the OpenMP Application Program Interface (OpenMP API) [15]. The compiler directives, library routines, and environment variables collectively known as the OpenMP API define a portable model for shared-memory parallel programming. OpenMP is the preferred model for adding threading to MPI applications for recent supercomputers based on highly-threaded processors.

Given the importance of OpenMP, it is necessary to develop effective tools that pinpoint and quantify causes of performance losses in OpenMP applications. To address that goal, our research aims to develop effective performance analysis strategies and techniques for OpenMP, prototype these strategies and techniques in a tool to evaluate their utility, and ensure that these strategies and techniques are embodied in a general OpenMP performance tools API that is suitable for inclusion in the OpenMP standard. This paper describes the results of our research.

There are several challenges for building effective performance tools for OpenMP. First, a tool must have low runtime overhead. Otherwise, any measurements it records will be suspect. Second, one must attribute performance measurements back to the program. In previous work [1], we have shown that attributing performance metrics to full calling contexts in parallel programs is essential to understand context-dependent performance bottlenecks, such as synchronization delays. Third, in parallel programs, identifying *symptoms* of performance losses is easy; gaining insight into their *causes* requires a more sophisticated approach.

To address these challenges, we developed a novel framework for measurement and analysis of OpenMP programs. Our approach supports *all* OpenMP features including nested parallel regions, work-sharing constructs, synchronization constructs, and OpenMP tasks. We have developed two novel methods to measure and analyze the performance of OpenMP programsm. First, we developed a measurement methodology that attributes blame for work and inefficiency back to program contexts. We show how to adapt prior work on measurement methodologies (blame shifting, see section 2) to OpenMP programs. We extended these methods to support dynamic thread-level parallelism in both time-

shared and dedicated environments. Furthermore, we describe a more space-efficient mechanism for handling locks. Second, we developed a novel deferred context resolution method that supports on-the-fly attribution of performance metrics to full calling contexts within an OpenMP program execution. This approach enables us to collect compact call path profiles for OpenMP program executions *without* the need for traces.

We demonstrate the utility of our tool in case studies of four well-known application benchmarks using a version of the GNU OpenMP (GOMP) implementation, augmented with lightweight instrumentation and an API to support our measurement approach. Using insights provided by our tool, we were able to significantly improve the performance of these codes.

The next section of the paper surveys major OpenMP profiling tools and distinguishes their various approaches from our approach. Section 3 describes how we attribute blame for performance losses in OpenMP programs. Section 3 also explains our online deferred context resolution method for attributing performance metrics to full calling contexts in OpenMP programs. Section 4 discusses implementation issues for our tool. Section 5 demonstrates the effectiveness of our solution by studying four OpenMP benchmarks and identifying nontrivial improvement opportunities. Section 6 summarizes our conclusions and describes our ongoing work.

2. RELATED WORK

Tools for profiling and tracing OpenMP programs use either instrumentation or sampling as their principal measurement approach.

Tools such as TAU [18], ompP [7] and Scalasca [25] use the OPARI [14] source-to-source instrumentation tool to insert monitoring code around OpenMP constructs and the POMP [14] monitoring API to support measurement. POMP was proposed as a potential standard tool API for OpenMP; however, it was not embraced by the standards committee. Concerns about POMP include its reliance on instrumentation, the complexity of the API, and its runtime overhead. IBM provides a nearly complete implementation of the POMP API as part of their High Performance Computing Toolkit (HPCT) [3] and uses binary rewriting to add POMP calls to application executables. Due to concerns about overhead, IBM does not support the POMP API in optimized versions of their OpenMP runtime.

In contrast, tools such as Intel Vtune Amplifier XE 2013 [9], PGI's Group PGPROF [22], and OpenSpeedShop [17] use sampling to monitor OpenMP programs with low measurement overhead. While Cray's CrayPat [4] principally uses sampling, it relies on Cray and PGI compilers to add tracepoints to record entry/exit of parallel regions and worksharing constructs. However, none of these tools pinpoint code regions that cause idleness or lock waiting in an OpenMP program. With the exception of PGPROF, which relies on special characteristics of PGI's OpenMP runtime, these tools cannot attribute performance information to user-level calling contexts and instead present an implementation-level view that separates a main thread from OpenMP worker threads.

The tool most similar to ours is Oracle Solaris Studio (OSS) [16]. OSS is a full-featured, sampling-based performance tool that supports analysis of OpenMP programs built using Oracle's OpenMP compiler and runtime [24]. A key component of Oracle's OpenMP runtime is an API that provides information to support sampling-based performance tools [10]. Using the API requires no compiler support. Oracle's API confers two important advantages to OSS. First, it enables OSS to collect intricate performance metrics like overhead, idleness and work. Second, it enables OSS to attribute these metrics to full user-level calling contexts. A drawback of OSS, however, is that to collect profiles, OSS must record sample *traces* and then resolve full calling contexts from these traces during post-mortem analysis. Such traces quickly grow large as a program executes. A significant difference between OSS and our work is the approach for measurement and attribution of waiting and idleness. OSS attributes idle or spinning threads to contexts in which waiting occurs rather than trying to relate the idleness back to its causes. For example, OSS does not blame sequential regions for thread idleness. Consequently, it does not highlight sequential regions as opportunities for improving program performance.

To address the generic problem of attributing performance losses to their causes, Tallent and Mellor-Crummey [19] proposed a method for performance analysis now known as *blame shifting*. Their approach is based on the insight that an idle thread is a symptom of a performance problem rather than the cause. They describe a strategy for sampling-based performance analysis of Cilk programs executed by a work stealing runtime. Rather than a thread attributing samples to itself when it is idle, it announces its idleness to working threads using two shared variables. These two variables count the number of idle and working threads separately. At any asynchronous sample event, a working thread reads both variables and computes its share of the blame, which is proportional to the number of idle threads divided by the number of working threads, for not keeping idle threads busy. We refer to this strategy as *undirected* blame shifting as a thread has no knowledge of what other thread or threads will share the blame for its idleness.

Later, Tallent at el. [21] proposed a different blame shifting technique for lock spin waiting. In the extended technique, A thread waiting for a lock shifts blame for its waiting to the lock holder. We refer to this strategy as *directed* blame shifting, since an idle thread arranges to transfer blame to a specific thread. When OpenMP programs execute, various circumstances require either a directed or undirected blame shifting strategy to attribute idleness or lock waiting to its underlying causes.

In this paper, we integrate and extend both directed and undirected blame shifting to handle the full range of OpenMP programs. We elaborate our approach in the next section.

3. APPROACH

Providing insight into the performance of OpenMP programs requires careful design and implementation of mechanisms for measuring and attributing execution costs. To achieve low overhead, we use a measurement approach principally based on asynchronous sampling. The efficacy of sampling-based profiling for analyzing program performance is well known, e.g., [20]. A sampling-based performance tool can collect a flat profile that identifies source lines and routines where an OpenMP program spends its time without any special support from an OpenMP runtime system. Such an approach can identify *symptoms* of inefficiencies,

such as spin waiting at barriers or for locks. However, providing insight into *causes* of inefficiencies in OpenMP programs requires designing tool measurement capabilities for this purpose and assistance from OpenMP runtime systems. We designed and prototyped lightweight instrumentation for OpenMP runtime systems that enables our tool to attribute idleness to causes. To make our approach attractive as a potential standard OpenMP tools API, important goals of our work were to minimize the runtime overhead of our instrumentation and developer effort needed to add it to any OpenMP runtime.

The following two sections elaborate our approach. Section 3.1 discusses how to attribute performance metrics to their causes for OpenMP programs. Section 3.2 describes how to attribute metrics to full calling contexts on the fly for OpenMP programs.

3.1 Attributing idleness and lock waiting

Our tool measures and attributes an OpenMP thread's execution time into four categories:

- *Idleness*: The time a thread is either spin waiting or sleep waiting at a barrier for work. For example, OpenMP threads waiting at a barrier inside or outside a parallel region are idle. If a thread is not idle, the thread is busy.
- *Work*: The time a busy thread spends executing application code, including both serial and parallel regions.
- *Overhead*: The time a busy thread spends executing code in the OpenMP runtime system.
- *Lock waiting*: The time a thread spends spin waiting for locks.

These four metrics reflect different aspects of an OpenMP program execution. Depending on which metrics are prominent, different tuning strategies apply. If idleness dominates work attributed to a code region, that code region is insufficiently parallelized. If lock waiting associated with a code region is high, that code is associated with significant lock contention and the use of mutual exclusion in that code region needs review. If overhead for a code region is high but idleness is low, increasing the granularity of parallelism may reduce the overhead. If overhead is low and idleness is high, then decreasing the granularity of parallelism may reduce the idleness. Finally, if the overhead and idleness for a region are both high, then changing the granularity of parallelism will not help performance and the parallelization of the code region merits review.

To collect these four metrics, we modified a version of the GNU OpenMP runtime to explicitly maintain a state variable for each thread. When a thread takes a sample, it can query the runtime to determine which metric should be adjusted. It is worth of noting that the sum of these four metrics represents the aggregate execution time across all threads.

Finally, we use different techniques to attribute these metrics to their causes. Since work and overhead belong to the threads performing them, we attribute both metrics to a thread receiving a sample. For idleness, we apply an *undirected blaming* technique to attribute it to busy threads, highlighting the fact that threads suffering from excessive work are the causes of idleness. For lock waiting, we apply a *directed blaming* technique to attribute it to lock holders to identify problematic locks.

Section 3.1.1 and 3.1.2 discuss the challenge and our solution for the undirected and directed blaming technique in OpenMP programs respectively. Section 3.1.3 describes how we integrate undirected and directed blaming to attribute idleness and lock waiting in OpenMP programs.

3.1.1 Undirected blaming for idleness

As discussed in Section 2, previous work by Tallent and Mellor-Crummey [19] considers how to blame idleness among threads in a static thread pool. Their approach is insufficient for OpenMP, because OpenMP allows the number of worker threads to change dynamically during execution. In contrast, our undirected blaming technique supports attribution of idleness for OpenMP programs in the presence of dynamic thread-level parallelism in both time-shared and dedicated environments. Equation 1 shows how we compute idleness for a program context in the presence of dynamic parallelism.

$$\begin{aligned} I_c &= \sum_{k=1}^{s_c} I_{c,k} = \sum_{k=1}^{s_c} p\frac{i_{c,k}}{b_{c,k}} = p\sum_{k=1}^{s_c}(\frac{i_{c,k}+b_{c,k}}{b_{c,k}} - 1) \\ &= p\sum_{k=1}^{s_c}(\frac{t_{c,k}}{b_{c,k}} - 1) \end{aligned} \quad (1)$$

I_c is the total idleness attributed to a specific context c in the program over time. I_c is measured in processor cycles or microseconds and represents the aggregate idleness across all threads while one or more threads executed code in context c. $I_{c,k}$ is the idleness attributed to context c for the k^{th} sample. p is the sampling period, which is constant during program execution. s_c is the total number of samples taken in the context c. $i_{c,k}$ is the number of idle threads in the system when sample k occurs in context c. $b_{c,k}$ is the number of busy threads in the system when sample k occurs in context c. Like Tallent and Mellor-Crummey [19], we apportion the blame for instantaneous idleness in the system among the busy threads at the time. $t_{c,k}$, shown as the sum of $i_{c,k}$ and $b_{c,k}$, is the total number of threads considered by our undirected blaming technique in the context c when sample k is taken. Equation 1 shows how we transform our idleness method to compute total idleness for a context by replacing the need for $i_{c,k}$, a measure of instantaneous idleness, with $t_{c,k}$, a measure of instantaneous thread-level parallelism. After the transformation, there is no need for our tool to maintain a shared variable reflecting the number of idle threads in the system.

Note, however, that choosing an appropriate value for $t_{c,k}$ depends on the environment. To see why, consider the appropriate action concerning sequential code occurring *before* the first OpenMP parallel region. In a time-shared environment, an OpenMP program should *not* be charged for idleness outside a parallel region. In contrast, in a dedicated environment, this sequential code *should* be charged for the idleness of allocated cores even if threads have not yet been allocated to run on them. In the text that follows, we will elaborate the details of how our tool automatically chooses the appropriate value of $t_{c,k}$, depending on the environment.

For a time-shared environment, we should compute the idleness incurred by OpenMP threads required by each code region. If a code region in an OpenMP program does not use

all hardware threads, these threads are available to other co-running programs and the OpenMP program should not be charged for unused threads. Thus, a sequential code region of an OpenMP program is not charged for idleness because it requires only one thread and keeps it busy. To handle this case, in a time-shared environment, we let $t_{c,k}$ to be the number of threads in use when sample k is received in context c. In this case, $t_{c,k}$ is a variable of c and k. Fortunately, OpenMP provides an API to query the number of threads in use at any time. Thus, we can compute $I_{c,k}$ eagerly at each sample k and accumulate it to I_c.

For a dedicated environment, we propose two ways to choose $t_{c,k}$ values to compute two different idleness metrics to reflect inefficiency with respect to different viewpoints. To compute *core idleness*, we set $t_{c,k}$ to be the total number of cores[1] in a dedicated system. This setting blames working threads in an OpenMP program if a core in the system is unused. For this case, the value of $t_{c,k}$ is constant during execution and can be read from system configuration information. Alternatively, to compute *thread idleness*, we want $t_{c,k}$ to be the maximum number of threads ever in use during a program's execution. This idleness metric is useful when assessing the scalability of an OpenMP program on different numbers of threads. In this case, the maximum number of threads used by a program will be unknown until the program terminates because an OpenMP program can dynamically adjust its thread count using an API. To handle this case, we add two callbacks to the OpenMP runtime system so our tool is notified when a thread is created or destroyed. In these callbacks, our tool maintains an instantaneous global thread count using atomic increment/decrement. During execution, our tool records the maximum instantaneous thread count observed. When computing *thread idleness*, $t_{c,k}$ represents the maximum instantaneous thread count during execution. Since our tool won't know its value until the end of an execution, we separately accumulate two terms in Equation 1, $p\sum_k \frac{1}{b_{c,k}}$ and $-ps_c$, for each context. At the end of an execution, we compute *both* core and thread idleness metrics for each context c according to Equation 1 by simply multiplying $\sum_k \frac{1}{b_{c,k}}$ through by the appropriate value of $t_{c,k}$, which is constant for all k samples in both cases, and adding it to the second term, which is proportional to the number of samples in c. Both of the aforementioned methods attribute idleness to sequential code regions in OpenMP programs, but relative to different baselines. In our case studies, we use the thread idleness metric to identify performance bottlenecks for benchmarks running in a dedicated environment.

3.1.2 Directed blaming for lock waiting

Tallent at el. [21] developed a version of directed blaming for pthread programs, described in Section 2. Their method, which uses function wrapping to override pthread locking primitives, is not applicable to OpenMP runtime systems for three reasons. First, OpenMP runtimes have several different mutual exclusion implementations: locks, critical, ordered, and atomic sections. There is no defined interface that a tool can wrap. Second, efficiently wrapping a locking primitive requires intimate knowledge about the representation it uses. However, OpenMP runtime vendors are unwilling to commit to a fixed lock representation—they want the freedom to use whatever representation they deem best for a particular architecture because the choice of lock representation may affect a program's speed and scalability. Third, the approach in [21] associates extra state with each live lock to store blame information. Here, we describe a new approach for directed blaming for OpenMP programs that efficiently supports directed blaming for all mutual exclusion constructs in an OpenMP runtime system. Furthermore, it requires only modest additional space for managing blame; namely, proportional to the maximum number of concurrently contended locks rather than the maximum number of live locks.

Our directed blaming technique works in three parts: lock acquisition, asynchronous samples, and lock release. To minimize the overhead of lock acquisition, we do not require that any code be added to the critical path of an uncontended lock acquisition in an OpenMP runtime. Only if a lock acquisition fails and a thread is forced to wait does our tool record the lock we are trying to acquire in a thread-local variable. In this case, recording the lock will have little or no effect on execution time since this operation is done only when a thread begins to wait.

When a thread receives an asynchronous sample, our performance tool's signal handler queries the thread state maintained in the OpenMP runtime. Besides returning the thread state, if the thread is currently in the *lockwait* state, the state query API we defined for the OpenMP runtime will also return the address of the lock that the thread is awaiting. If the thread is in the *lockwait* state, the signal handler uses the lock address as the key for a hash table. Our tool uses a hash table of fixed, modest size, e.g. 1024 entries, to maintain a map of blame that must be charged to the thread holding any particular lock.[2] The hash table is initialized when our tool is first attached to the program. When the signal handler tries to accumulate blame for a lock in the hash table, it allocates an entry for the lock in the table on demand if one doesn't already exist. Hash table entries are tagged with the lock address. If two held locks hash to the same entry, blame for one will be discarded. When monitoring is enabled and a lock release callback is registered with the OpenMP runtime, then every lock release will invoke the callback with the lock address as an argument. At this point, we look up the lock in the hash table, accept any blame that has accumulated in the table for the lock, and reset the table entry's state to free so that it is available to be reallocated on demand. If the blame accumulated in the table is greater than 0, then our tool unwinds the call stack and attributes the accumulated blame for lock waiting to the calling context of the lock release.

When designing the protocol for concurrently maintaining the hash table to accumulate blame, we considered two types of implementations: a heavyweight algorithm that carefully handles any data race that might arise when two locks hash to the same table entry, and a lighter weight implementation that doesn't use atomic operations to manage data races. The heavyweight approach uses atomic operations and has higher runtime overhead. The lighter weight method might cause a lock to receive additional blame or some blame to be lost. The odds of this happening in practice are extremely

[1] Alternatively, one could do this for SMT hardware thread contexts.

[2] The idea of only maintaining blame for *actively contended* locks in a hash table is due to Alexandre Eichenberger (IBM T.J. Watson Research Center).

small. As a result, we opted for the lighter weight protocol. It works exceptionally well in practice because (a) the number of locks actively contended at the same time is much smaller than the size of hash table so collisions are rare, and (b) only locks with very large sample counts charged to them are worthy of attention by an analyst. While data races can change sample counts slightly, infrequent errors of this sort are extremely unlikely to change an analyst's assessment as to whether a particular lock is a problem or not.

3.1.3 Integrating directed and undirected blaming

When an OpenMP program has threads in both idle and lockwait states at the same time, it suffers performance losses from both thread idleness and lock contention. To handle this case properly, directed and undirected blaming techniques need to coordinate. The *lockwait* metric needs no special care because we can always blame it directly to the context of lock holders. However, the attribution of *idleness* in such circumstances deserves careful attention.

Consider an OpenMP program execution at a point in time when there are n_i threads idle, n_w threads working, and n_l thread waiting for locks. There are three reasonable ways to consider attributing blame for *idleness* among these threads. First, we consider using undirected blaming to only charge idleness of the n_i idle threads to the n_w working threads. Any working thread that receives an asynchronous sample in this execution state charges itself for $\frac{n_i}{n_w}$ units of idleness. This blaming scheme highlights code regions being executed by working threads as contributing to the idleness of other threads, but it ignores threads holding locks as deserving extra attention. The second method uses undirected blaming to charge an equal fraction of idleness to each thread that is working or waiting for a lock. This method attributes $\frac{n_i}{n_w+n_l}$ idleness to the context that receives an asynchronous sample in a working or waiting thread. This scheme charges idleness not only to the context of each working thread that receives an asynchronous sample, but also to the context of any thread waiting for a lock. The rationale of this scheme is that an idle thread is waiting for all working and waiting threads. Note, however, that each thread waiting for a lock is waiting for a lock holder. Our third method uses undirected blaming technique to charge idleness to threads both in working and lock waiting state, and then uses directed blaming to charge idleness inherited by a thread waiting for a lock to its lock holder. For example, if all n_l waiting threads are spinning at the same lock, this method attributes $\frac{n_i}{n_w+n_l}$ idleness to each context that receives an asynchronous sample in a working thread, $\frac{n_i n_l}{n_w+n_l}$ idleness to the lock release, and no idleness to the context in each waiting thread. We choose the third of these methods because it transfers any blame for idleness received while waiting for a lock directly to the lock holder.

3.2 Deferred context resolution

Attributing metrics to full calling context is essential for understanding the performance of parallel programs and the underlying causes of inefficiencies. Consequently, interpreting the runtime stack for both master and worker threads in an OpenMP program is paramount. Most OpenMP runtimes manage master and worker threads separately. In fact, too our knowledge, only PGI's OpenMP runtime uses a cactus stack to readily provide both master and worker threads with a unified call stack view.

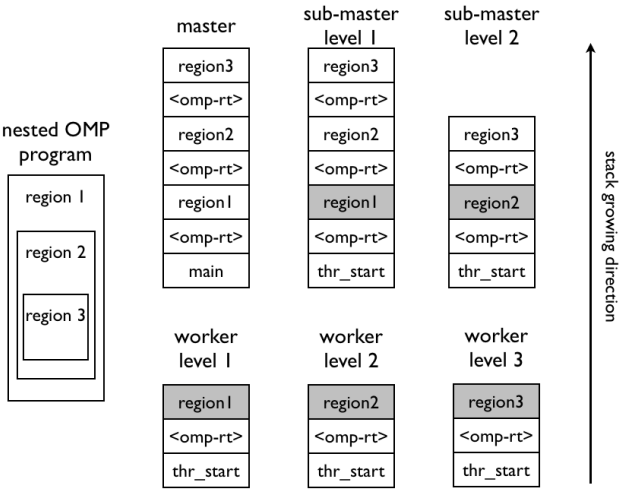

Figure 1: The call stack for master, sub-master and worker threads in an OpenMP program with three levels of nested parallelism. As shown, stacks grow from bottom to top. The calling contexts for the parallel regions shown in grey are scattered across two or more threads.

To illustrate the problem tools face interpreting call stacks of OpenMP threads, consider Figure 1, which shows the call stacks of several threads in the presence of nested parallelism. When a worker thread in an outer parallel region encounters nested parallelism, it becomes a master thread for the inner region. Threads other than the master thread lack full calling context for computations they perform. Assembling the complete user-level calling context for work in a parallel region requires information about the calling context for the region itself, as well as that of any enclosing regions. Thus, to assemble the full calling context for region 3 being executed by the level 3 worker, the level 3 worker requires the location in region 3 from its own stack, the context in which region 3 was invoked by the level 2 sub-master thread, the context in which region 2 was invoked by the level 1 sub-master thread, and finally the context in which region 1 was invoked by the master thread.

A straightforward way to support assembly of distributed context information involves the following steps when encountering a parallel region:

1. Create a unique ID for the parallel region instance.
2. If the master thread encounters a parallel region, unwind its call stack to capture the region's creation context and store the resulting context in a region table where it can be looked up by region ID.
3. If a sub-master thread or a worker encounters a new parallel region, unwind its call stack to the parallel region at the base of its stack, look up the calling context for that parallel region in the region table, assemble the full calling context of the new region instance, and register it in the region table.

When a thread receives an asynchronous sample as the program runs, the thread merely needs to unwind its own call

stack, determine the parallel region at the base of its stack, look up the prefix of its calling context in the region table, and assemble the final result. To support this approach, the OpenMP runtime system needs to supply a callback at the entry of each parallel region instance and an API that the tool can use to query a parallel region instance ID.

Capturing the context using the outlined approach, however, is more expensive than it needs to be. The overhead of this approach can be reduced in some cases. If a parallel region is small and worker threads in the region run quickly enough so that no asynchronous samples are taken while they are executing work in the region, then there is no need to recover the full calling context for the region. To avoid the cost of context capture when no worker thread has been sampled in the region, Itzkowitz [10] proposed a *deferred* context resolution technique that is used in Oracle Solaris Studio [16].

Rather than capturing the calling context context for a new a parallel region instance as it is entered, Oracle Solaris Studio defers capturing context until the end of the region instance. They maintain a global map that indicates whether a thread received an asynchronous sample within a parallel region instance. If not, then one can skip capturing the region's context. While the deferred context resolution can dramatically reduce measurement overhead for small, but frequently called parallel regions, it introduces a new problem: a thread can no longer immediately assemble the full context for its work when it receives an asynchronous sample. The context for any enclosing parallel regions will only become available *after* the regions exit. In Oracle Solaris Studio, Itzkowitz et al. cope with this by recording available partial context information for each sample in a trace. Then, in a post mortem phase, full context is reconstructed using context for enclosing parallel regions that later becomes available. Supporting deferred context resolution places an additional constraint on an OpenMP runtime system—a callback from region *exit* is required.

To avoid the need for large traces and post-mortem context assembly when using deferred context resolution, we devised an approach for *online* deferred context resolution. Whenever the master thread receives an asynchronous sample, it unwinds its call stack and enters its call path into the reference calling context tree (CCT). Whenever an OpenMP worker thread receives an asynchronous sample, the thread queries its parallel region ID. Its full calling context won't be known at this point because the context of the enclosing parallel region won't be known until after the region exits. Let us denote the call stack suffix the worker thread recovers at this point as s. Since the calling context for the enclosing parallel region is not yet known, s will have a placeholder at its base associated with its enclosing parallel region instance ID r_1. The thread next records s in a thread-private CCT rooted at r_1. This CCT is stored in a *thread CCT map* indexed by r_1. When the thread receives a subsequent sample, it again unwinds its call stack. If it is still in in the parallel region instance with ID r_1, it adds its new call stack suffix to its calling context tree rooted at r_1 in the *thread CCT map*.

When a master or sub master exits a parallel region r, it checks the *region instance sample map* to see if a sample was processed by any worker or sub-master thread in that parallel region instance. If so, the thread unwinds its call stack to acquire the calling context s_2 of the parallel region instance. s_2 is entered in the *resolved region map* indexed by the parallel region ID r.

When a worker or sub-master thread receives an asynchronous sample, if it finds itself in a parallel region different than the one for the previous sample, it inspects the *thread CCT map* to see if any context information for its CCTs is available from the *resolved region map*. If so, it prepends context information from the *resolved region map* to its relevant CCTs. Any CCT whose context is now fully resolved is taken out of the *thread CCT map* and placed into the reference CCT. When a thread exits or a process terminates, all remaining deferred contexts are resolved and any remaining information in the *thread CCT map* is folded into the reference CCT.

We compare our online method for deferred context resolution with the offline method used by Oracle Solaris Studio using the LULESH application benchmark, which is one of our case studies described in Section 5.2. For a 48-thread execution of LULESH using the same program input and sampling rate, Oracle Solaris Studio records a 170MB performance data file to perform deferred context resolution postmortem. Using our online deferred context resolution strategy, our tool records a compact profile for each thread, 8MB total—a reduction in data volume of more than 20x.

Like Oracle Solaris Studio, our prototype tool distinguishes OpenMP runtime frames on the call stack from others by simply identifying that they belong to the shared library for the OpenMP runtime. For the OpenMP standard, a solution suitable for statically-linked programs is needed as well. We are part of the OpenMP standards committee working group designing a new tools API that provides the necessary support. The new API has the OpenMP runtime record stack addresses for procedure frames that enter and exit the OpenMP runtime. When one uses call stack unwinding, this interface will enable one to identify sequences of procedure frames on the stack that belong to the OpenMP runtime rather than user code, even for statically linked programs. We are in the process of evaluating this design to provide feedback to the OpenMP tools API subcommittee.

A note about context resolution for OpenMP tasks.

OpenMP 3.0 tasks [2] support work stealing execution. With respect to context resolution, this introduces an additional consideration. Since task creation may be separated from task execution, tasks have an additional component to their context—the *creation* context.

To record the creation context of a task, we add an 8-byte field in the OpenMP task data structure maintained by the runtime system. Once a non-immediate-run task is created, a callback function is invoked to pass the 8-byte field to the performance tool. The performance tool unwinds the call stack to collect the task creation context in the callback function invoked at each task creation point. It then updates the 8-byte field in the task structure with the task creation context. When a task is executed, the performance tool uses two query APIs to get the task creation context and the root frame of the executing task. We combine the two to get the full context of the task. One complicated but practical case is that one task can be nested in another task. We can concatenate the outer task creation context with the inner task creation context to resolve the full context of the inner task. However, if the program recursively creates nested tasks, the call site of inner-most tasks may be deep

in the call stack. Long recursive call paths aren't typically of much interest to a performance analyst. For that reason, we collapse the call stack for recursive tasks. If the root frame of one task is the same as that of its parent, we fold the performance data of the task into its parent and skip its frames on the call stack.

Incorporating the creation context into the complete context for every OpenMP task is expensive. The overhead of eagerly gathering this information cannot be tied to the sampling rate, and therefore it is uncontrolled. Consequently, we do not employ *creation-extended* task context construction by default. Rather than providing a full creation context for a task by default, we simply associate the task with its enclosing parallel region. If a performance analyst needs the full creation context for tasks, our tool will collect it upon request, though this extra information increases monitoring overhead. Our case studies in Section 5 did not require full creation contexts for tasks to provide performance insights.

4. TOOL IMPLEMENTATION

To evaluate our ideas, we implemented them in the HPCToolkit [1] performance tools. HPCToolkit is an open-source, multi-platform, sampling-based performance tool for measurement and analysis of application performance on parallel systems. We extended HPCToolkit's existing call path profiler to support the techniques described in the previous section. Using these techniques, our extended HPCToolkit is able to analyze the performance of OpenMP and OpenMP+MPI programs by both collecting measures of work, idleness, overhead, and lock waiting; and then attributing these measures to full calling contexts in a unified user-level view. To analyze performance data, we use HPCToolkit's `hpcviewer` graphical user interface. `hpcviewer` associates both measured and derived metrics to full calling contexts, thereby representing a unified view reconstructed using our online deferred context resolution strategy.

The implementation of our techniques, however, requires support from the OpenMP runtime system. Consequently, we took special care to minimize the required support. Our initial prototype was based on an open-source OpenMP implementation — GNU OpenMP (GOMP) [6]. To support both our blaming strategies and online deferred context resolution, we modified GOMP source code to insert necessary callbacks, query functions, and data structures associated with parallel regions and tasks. Our modification encompassed 5 files and less than 50 lines of code. The software development cost to add the necessary OpenMP runtime support for our methods is quite low. Also, as we show in Section 5, the runtime impact of our proposed support is similarly low.

To quantify the improvement one can obtain after optimization, we compute two derived metrics from each raw metric we defined in Section 3.1. For example, we derive *absolute idleness* and *relative idleness* metrics from the raw *idleness* metric. Absolute idleness quantifies the idleness in a given context relative to the entire effort of the program. It shows the maximum possible (percentage) improvement if the idleness is eliminated for that context. Relative idleness reflects the parallel efficiency for the context. A high relative idleness value means the context is making poor use of parallel resources. A good rule of thumb is to focus optimization efforts on contexts with both high absolute idleness and high relative idleness.

Note that both derived metrics are computed after the raw data has been collected. The defining equations are shown in equation 2. The notation key in equation 2 is as follows: $I_{c,abs}$ is the absolute idleness computed for context c; $I_{c,rel}$ is the relative idleness computed for context c; I_c, W_c, O_c, and L_c are idleness, work, overhead, and lock waiting respectively attributed to the context c; I_r, W_r, O_r, and L_r are idleness, work, overhead, and lock waiting respectively aggregated for the whole program.

$$\begin{aligned} I_{c,abs} &= \frac{I_c}{I_r + W_r + O_r + L_r} \times 100\% \\ I_{c,rel} &= \frac{I_c}{I_c + W_c + O_c + L_c} \times 100\% \end{aligned} \quad (2)$$

It is worth noting that analogous derived metrics exist for *work*, *overhead* and *lock waiting* as well.

5. CASE STUDIES

In this section, we evaluate the utility and overhead of our measurement approach using a set of application benchmarks that employ OpenMP parallel loops with and without nested parallel regions, OpenMP tasking, and OpenMP in conjunction with MPI. Below, we briefly describe the benchmarks that we studied.

- AMG2006 [13], one of the Sequoia benchmarks developed by Lawrence Livermore National Laboratory, is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. The driver for this benchmark builds linear systems for various 3D problems. It consists of sequential and parallel regions that both affect its performance. AMG2006 is a MPI-OpenMP hybrid benchmark written in C.

- LULESH [12] is an C++ application benchmark developed by Lawrence Livermore National Laboratory, solving the Sedov blast wave problem for one 3D material. LULESH is implemented using a variety of parallel programming models; here, we study the performance of its OpenMP version. The OpenMP implementation of LULESH contains parallel regions inside deep loop nests so the number of parallel region instances in an execution is more than 20K.

- NAS BT-MZ [11] is a multi-zone parallel benchmark written in Fortran that employs two levels of nested OpenMP parallelism. It divides the data into several zones which are computed in the first level parallel regions. In each zone, computation is also parallelized in second level parallel regions. This benchmark manages load balance in the outer-level parallel regions.

- HEALTH [5] is a benchmark that is part of the Barcelona OpenMP Tasks Suite. With the medium input, it creates more than 17 million untied tasks during execution. The benchmark is used to evaluate task creation and scheduling policies inside an OpenMP runtime system.

We compiled each of these benchmarks using the GNU 4.6.2 compiler with `-O3` optimization and linked with our modified version of GOMP library from GCC 4.6.2. We measured the performance of these codes on a system with four 12-core AMD Magny-Cours processors and 128 GB

Benchmark	native GOMP	modified GOMP	profiling
AMG2006	54.02s	54.10s	56.76s
LULESH	402.34s	402.56s	416.78s
NAS BT-MZ	32.10s	32.15s	34.23s
HEALTH	71.74s	72.20s	74.27s

Table 1: Running times for our case study applications with (a) the unmodified GOMP runtime library (no sampling), (b) the modified GOMP library with lightweight instrumentation to support blame shifting (no sampling), and (c) the modified GOMP library while profiling at 200 samples per second per thread. Execution times represent the average over three executions.

memory. For AMG2006, we studied executions consisting of 4 MPI processes and 8 OpenMP threads per process. For LULESH, we studied executions with 48 OpenMP threads. For NAS BT-MZ, we studied executions that used 4 threads for outer parallel region and 8 threads for inner parallel regions. For HEALTH, we studied runs on 8 OpenMP threads. We measured the performance of these codes using asynchronous sampling at a rate of 200 samples/s per thread.

We first evaluate the measurement overhead associated with our modified GOMP library and profiler. Table 1 compares the execution time of each benchmark code using the native GOMP library, our modified GOMP library with performance measurement hooks, and the modified GOMP library with our profiler attached. The capabilities employed by our profiler include using undirected, directed as well as integrated blaming for *idleness*, *work*, *overhead* and *lock waiting*, and using online deferred context resolution for parallel regions, resolving task contexts to their enclosing parallel region. By comparing the times in the first and second columns of the table, we see that our GOMP modifications to support performance tools add almost no overhead. Comparing the first and third columns shows that the measurement overhead using our profiler is less than 5% for each of these codes.

Note that if we resolve the *full* creation context for tasks to their creation contexts in HEALTH, the run time of a profiled execution jumps from 74.27s to 431.48s. This is more than a 6x slowdown compared to its unmonitored execution. Resolving the full creation context for tasks in HEALTH is so expensive because HEALTH creates 17 million tiny tasks as it executes, with each task requiring a call stack unwind to recover its full creation context. To keep measurement overhead low, our profiler's default measurement approach is to only resolve task contexts to their enclosing parallel region.

In the following four sections, we illustrate the effectiveness of our profiling approach by describing insights it delivered for each of our benchmark applications and how they helped us identify optimizations that yielded non-trivial improvements to these codes.

5.1 AMG2006

Figure 2 shows a time-centric view of traces of AMG2006 executing on 8 MPI ranks with 8 OpenMP threads per rank. Trace lines for each of the MPI processes and OpenMP threads are stacked along the Y axis and time flows left

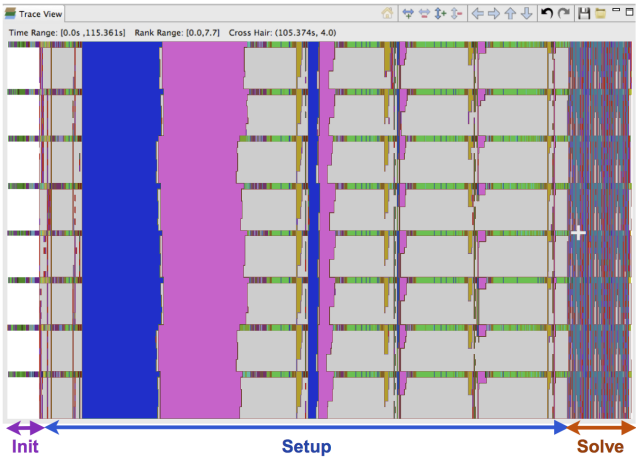

Figure 2: HPCToolkit's time-centric view rendering of a complete execution of AMG2006 on 64 cores – 8 MPI ranks with 8 OpenMP threads/rank.

to right. The color at each point in a thread's time line represents the procedure executing at that time. An execution of AMG2006 consists of three phases: MPI initialization, problem setup, and solve.

During initialization, only the master thread in each MPI rank is active. The initial white space in the traces of OpenMP worker threads shows that they are idle before they are created. During the setup phase shown in Figure 2, the light gray color for OpenMP worker threads represents worker threads blocked in `pthread_cond_wait`. This view illustrates that OpenMP threads are idle most of the time; the largest component of that idleness occurs while the master thread in each MPI rank executes serial code for `hypre_BoomerAMGCoarsen`. Our code-centric view for the full execution (not shown) reports that 34.9% of the total effort in the execution corresponds to idleness while `hypre_BoomerAMGCoarsen` executes; the relative idleness our tool attributes to this routine is 87.5%—exactly what we would expect with one of 8 cores working within an MPI rank. Our tool's quantitative measure for idleness attributed to this serial code is a result of our blame shifting approach. Neither VTune nor Oracle Solaris Studio directly associates idleness with serial code. During the setup phase, short bursts of activity by OpenMP worker threads separate long gray intervals of idleness. However, the intervals of activity by the worker threads are of unequal lengths, which indicates that even when threads are working, their work is imbalanced. While the setup phase in this benchmark execution accounts for a large fraction of the total time, the solve phase dominates in production computations. For that reason, we focus the rest of our analysis on the performance of the solve phase. Figure 3 shows an expanded view of one iteration of the solve phase. The imbalanced intervals of gray indicate idleness caused by a severe load imbalance in OpenMP loops employed by `hyper_BoomerAMGRelax`.

To analyze the solver phase in more detail, we use HPCToolkit's code-centric *hpcviewer* to analyze measurement data collected for just the solve phase of the benchmark in an execution by 4 MPI ranks with 8 OpenMP threads per rank. Figure 4 attributes *idleness*, *work* and *overhead* to the com-

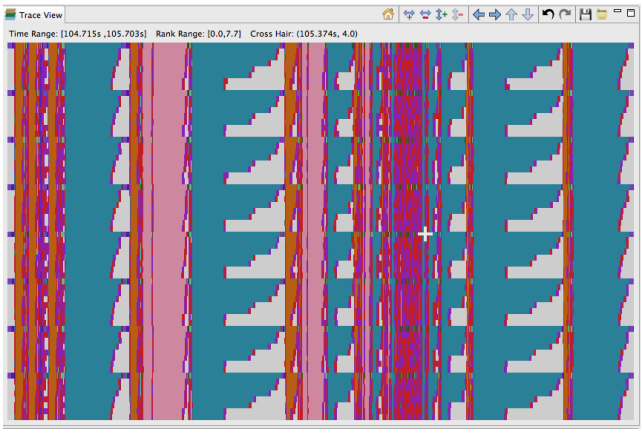

Figure 3: HPCToolkit's time-centric view rendering of a single iteration of AMG2006's solver on 64 cores – 8 MPI ranks with 8 OpenMP threads/rank.

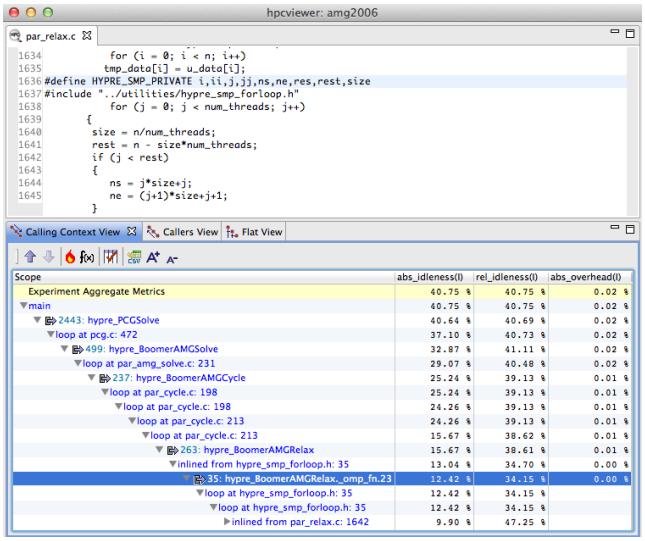

Figure 4: A calling context view of AMG2006's solver phase. More than 12% of the execution time of all threads is spent idling in the highlighted parallel region.

plete calling context of parallel regions. *hpcviewer* consists of three panes. The top one is the source code pane; the bottom left one shows program contexts such as routines, loops and statements; the bottom right one shows the metrics associated with each program context. Both inclusive and exclusive values are computed for all metrics. Figure 4 shows only inclusive metric values. The 40.75% aggregate absolute idleness metric for the solve phase means that threads are working only about 60% of the time. The OpenMP runtime overhead for the solve phase is negligible—only 0.02%. Drilling down the call path from `main`, we find that a large part of the idleness in the solve phase is attributed to a parallel region—`hyper_BoomerAMGRelax._omp_fn.23`, which is highlighted in Figure 4. Work by that routine (and routines it calls) accounts for 12.42% of the idleness in the solve phase. The relative idleness measure of 34.15% for this program context indicates that this parallel region is active, roughly one third of the threads are idle. The discussion in Section 3.1 indicates that if overhead is low and idleness is high, one should reduce the granularity of parallel work to increase parallelism and improve load balance.

To apply this optimization, we examined the source code corresponding to `hyper_BoomerAMGRelax._omp_fn.23` in the source pane of Figure 4. We found that this parallel region decomposes the computation into one piece of work for each thread and each thread is statically assigned a piece of work. Since each piece of work computes an equal number of iterations in the parallel loop, it would appear that each thread has equal amount of work. However, the execution time of iterations differs significantly due to characteristics of the input dataset, resulting in load imbalance. To reduce the load imbalance that causes the high *idleness* in this parallel region, we decomposed the work into a number of smaller pieces and used dynamic scheduling to balance the work among threads. Decomposing the work into five chunks per thread provided enough flexibility for dynamic scheduling to reduce load imbalance without adding too much overhead for the scheduling itself. The optimization reduced the *idleness* of `hyper_BoomerAMGRelax._omp_fn.23` from 3.05e+11 to 4.53e+09 CPU cycles, reducing the running time of the solve phase by 11%.

5.2 LULESH

In Figure 5, we analyze measurements from a 48-thread execution of LULESH using a bottom-up view which attributes metric consumption to routines and their calling contexts. We sorted the results to identify routines with the highest exclusive absolute idleness. The top routine is the `madvise` system call, which is called by `free` from both the application and the OpenMP runtime. The idleness associated with `free` amounts to 4.6% of the total thread execution time. The percentage of relative idleness indicates that only the master thread calls `free` and the other 47 threads are idle. It is worth noting that the performance tool reports that `free` accounts for only 0.1% of total execution time. Without idleness blame shifting, it would not appear that memory management is a potential performance problem. To see if we could improve performance by using a better memory allocator, we used Google's TCMalloc [8] (a high-performance threaded allocator) instead of `glibc`'s implementations of `malloc` and `free`. Surprisingly, the execution time drops from 402s to 102s—a 75% improvement!

To understand the improvement, which was well beyond our expectations, we compared profiles of the original and optimized versions of LULESH. As we expected, the absolute idleness associated with `free` drops from 4.6% to less than 0.1%. However, we also found that the work associated with some parallel regions shrank by a factor of 20. Further examination showed that data is allocated immediately before entering these regions on each iteration of an enclosing loop. Pseudo code in Figure 6 shows the structure of these problematic parallel regions. Because `glibc`'s `free` releases memory pages to the operating system, a subsequent `malloc` causes pages to be added back to the application's address space. As threads attempt to write these newly-allocated pages in the parallel region, they fault and stall as the operating system lazily zero-fills the pages. Repeatedly freeing, reallocating, and zero-filling pages is surprisingly costly. Un-

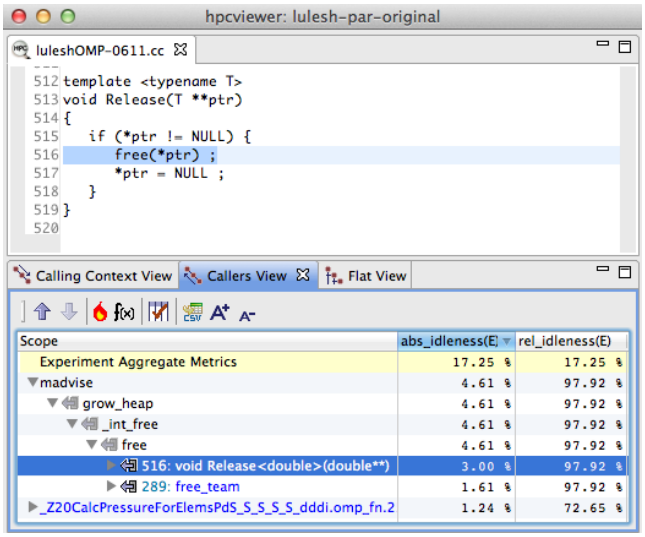

Figure 5: A bottom-up view of call path profiles for LULESH. `madvise`, which is called while freeing memory, is identified as the routine with the most idleness.

```
for( ... ) {
    malloc a[n], b[n], c[n];
    #pragma parallel for
    for(i=0; i<n; i++)  {
        a[i] = ...;
        b[i] = ...;
        c[i] = ...;
    }
    free a, b, c;
}
```

Figure 6: Pseudo code showing how memory is allocated, initialized, and freed in LULESH.

like `glibc`, TCMalloc does not return deallocated pages to the operating system so it avoids the cost of repeated zero-filling, leading to a 75% speedup.

While our idleness measure didn't help us predict the full benefit of changing the memory allocator, it did uncover the "tip of the iceberg," which helped us improve the performance of LULESH. The high overhead of zero-filling pages that we missed with our measurements was masquerading as work since it was incurred in parallel regions. An analysis based on sampling measurements of real time and graduated instructions might have provided some additional insight into operating system overhead that was unobserved.

5.3 NAS BT-MZ

To evaluate HPCToolkit's support for measurement and analysis of OpenMP programs with nested parallel regions, we used it to study the performance of the NAS BT-MZ benchmark. Figure 7 shows the unified calling contexts that HPCToolkit recovers for nested parallel regions in BT-MZ. In the figure, the parallel region `MAIN__._omp_fn.3` is nested inside `MAIN__._omp_fn.0`, showing that our method properly reconstructs calling contexts for nested parallel regions. The aggregate *absolute idleness* is larger than *absolute work*, which means that threads are idle more than half of the time during execution. Examination of the inclusive idleness measurements for the calling context of the nested parallel regions shows that most of the idleness comes from the innermost region. To optimize BT-MZ, we refactored the `exch_qbc` routine highlighted in Figure 7 because it has 1.55% *absolute work* but 11.1% *absolute idleness*. The *relative idleness* for this routine is more than 87%. Looking deeper along the call path inside the innermost parallel region, we see that `pthread_create` is responsible for idleness that amounts to roughly 3% of the overall execution cost. This cost is incurred because GOMP does not use a persistent thread pool for threads in nested parallel regions. Threads used in nested parallel regions are created at region entry and destroyed at region exit. For a nested parallel region in a loop, frequent thread creation and destruction can degrade performance.

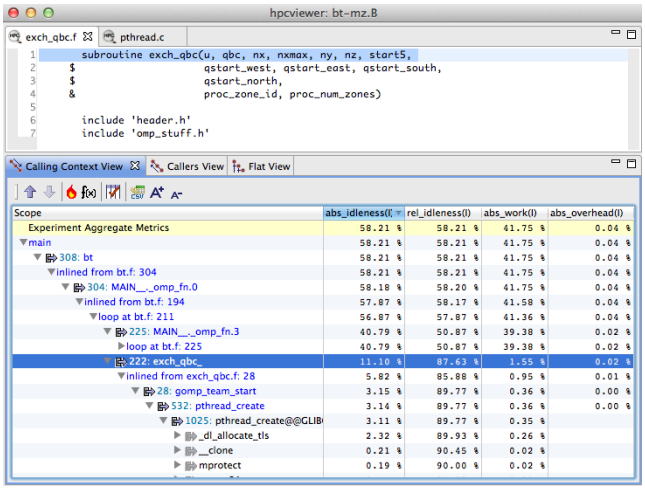

Figure 7: A calling context view of call path profiles for BT-MZ shows that the amount of idleness reported for `exch_qbc` is over seven times larger than the work it performs.

Changing the inner parallel region inside `exch_qbc` to a sequential region eliminated the *idleness* caused by thread creation and destruction, improving performance by 8%.

5.4 HEALTH

To evaluate HPCToolkit's support for measurement and analysis of programs based on OpenMP tasking, we studied the HEALTH benchmark. In this execution, we used HPCToolkit's (low-overhead) default context resolution for tasks, which attributes tasks back only to their enclosing parallel region rather than their full creation context. In experiments with HEALTH on eight threads, we found that its absolute idleness was much less than 1%, indicating that each thread was working on a parallel task most of the time. However, severe lock contention in the benchmark caused threads to spend much of their time spin waiting. Figure 8 shows a bottom-up view of the lock waiting metrics for an execution of HEALTH. The absolute lock wait metric shows threads in HEALTH spent roughly 75% of their total execution time waiting for locks. As described in Section 3.1.2,

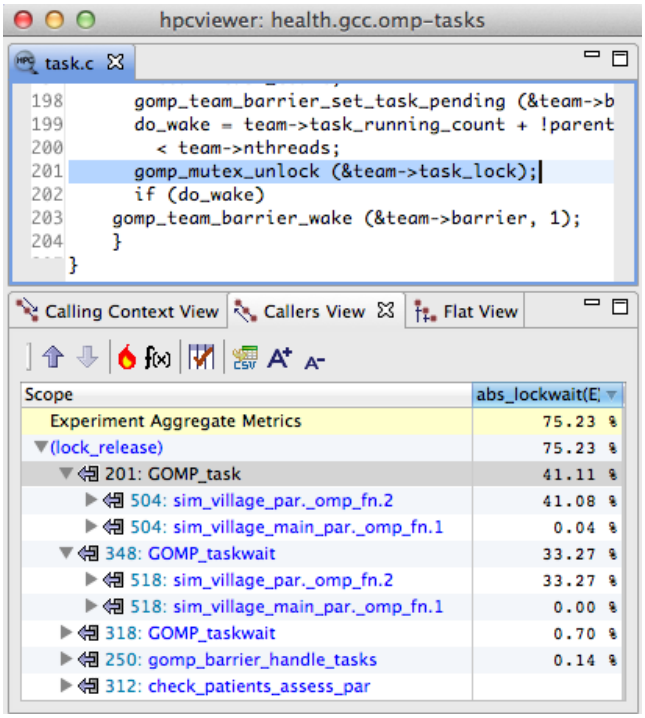

Figure 8: Bottom up view of HEALTH benchmark showing lock contention in the GOMP library.

blame for lock waiting gets attributed to the context where the lock holder releases the lock. Figure 8 shows blame being attributed up call paths from lock release to contexts where the lock was released.

The highly contended locks are released on line 201 of `GOMP_task` and line 348 of `GOMP_taskwait`. Locks released at these points account for significant waiting—roughly 41% and 33% of the thread execution time respectively. The OpenMP task `sim_village_par._omp_fn_2` is the common caller of `GOMP_task` and `GOMP_taskwait`, where lock contention leads to significant waiting. The lock release highlighted in the source code pane shows the problematic lock—`task_lock`. Note that `task_lock` is used in the GOMP library, not the user's code. `task_lock` is the lock used to manipulate the task queue created in a parallel region. It causes excessive contention in `sim_village_par._omp_fn_2`, as this routine recursively spawns 17 million tiny tasks and all threads contend for the lock to access the task queue. The `task_lock` used by `sim_village_main_par._omp_fn.1` causes little contention, so there is no need to optimize this parallel region. The best way to reduce contention for the task queue inside `sim_village_par._omp_fn_2` is to reduce the number of tasks by increasing task granularity. Another version of the program achieves this objective by using a cutoff to stop spawning tasks once granularity falls below a threshold. This improvement reduced program execution time by 82%, which is consistent with what our lock waiting measurements predicted.

6. CONCLUSIONS & ONGOING WORK

This paper proposes mechanisms to support low-overhead, sampling-based performance analysis of OpenMP programs. We demonstrate that an implementation of these techniques in HPCToolkit provides deep insight into the performance of threaded program executions by measuring and attributing informative metrics including idleness, work, overhead, and lock waiting. Our OpenMP profiler employs online deferred context resolution to efficiently and accurately attribute these metrics to full calling contexts in compact profiles, avoiding the space overhead of traces required by prior tools. Reducing the space overhead is an important aspect of our strategy that will enable it to scale to large parallel systems. The case studies reported in this paper validate the effectiveness and low overhead of our approach. Table 2 indicates the problems we identified using our tool in each of the application benchmarks, summarizes the optimizations we applied to improve each program, and shows the percent improvement that we achieved.

app	problem	optimization	improvement
AMG2006	high idleness in parallel regions	use dynamic scheduling to eliminate load imbalance	11%
LULESH	high idleness caused by a sequential routine `free`	use Google's tcmalloc threaded allocator	75%
BT-MZ	high idleness caused by frequent thread creation and exit in nested parallel regions	eliminate unnecessary inner parallel regions	8%
HEALTH	high lock contention for the task queue in OpenMP runtime	coarsen the task granularity	82%

Table 2: Summary of performance bottlenecks identified using our tool and our code optimizations in response.

We are presently working with the OpenMP standards committee to define a standard tools API for OpenMP. The emerging interface developed with the committee provides full support for our blame shifting approach. This includes interfaces that enable us to attribute performance metrics to full calling contexts as well as to pinpoint and quantify root causes of thread idleness and lock waiting. The new tools API provides support sufficient for analysis of both statically and dynamically linked applications. IBM has added support for a draft of this interface to a new lightweight OpenMP runtime system for their XL compilers for Blue Gene/Q. An implementation of HPCToolkit employs this draft interface to measure OpenMP programs executed by this runtime.

In the future, we plan to generalize our approach for deferred context resolution to support assembly of global view call path profiles on a heterogeneous system that offloads computation from a host node populated with conventional multicore chips to attached manycore accelerators, such as Intel MIC chips.

Acknowledgments

We thank Laksono Adhianto and Mark Krentel for their contributions to HPCToolkit. Without their efforts, this work would not have been possible. We thank Alexandre Eichenberger for his enthusiastic collaboration on the emerging OMPT OpenMP tools API. This research was supported in part by the DOE Office of Science under cooperative agreement number DE-FC02-07ER25800 and by Sandia National Laboratories under purchase order 1293383.

7. REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22:685–701, 2010.

[2] E. Ayguadé et al. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, Mar. 2009.

[3] I.-H. Chung. IBM high performance toolkit, 2008. `https://computing.llnl.gov/tutorials/IBM.HPC.Toolkit.Chung.pdf`.

[4] Cray Inc. Using Cray performance analysis tools, April 2011. Document S-2376-52, `http://docs.cray.com/books/S-2376-52/S-2376-52.pdf`.

[5] S. R. Das and R. M. Fujimoto. A performance study of the cancelback protocol for time warp. *SIGSIM Simul. Dig.*, 23(1):135–142, July 1993.

[6] Free Software Foundation. GOMP—an OpenMP implementation for GCC. `http://gcc.gnu.org/projects/gomp`, 2012.

[7] K. Fürlinger and M. Gerndt. ompP: A profiling tool for OpenMP. In *Proc. of the First and Second International Workshops on OpenMP*, pages 15–23, Eugene, Oregon, USA, May 2005. LNCS 4315.

[8] Google Inc. TCMalloc: Thread-Caching Malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`. Last accessed April 3, 2013.

[9] Intel. Intel VTune Amplifier XE . `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe`, July, 2012.

[10] M. Itzkowitz, O. Mazurov, N. Copty, and Y. Lin. An OpenMP runtime API for profiling. `http://www.compunity.org/futures/omp-api.html`.

[11] H. Jin and R. F. V. der Wijngaart. Performance characteristics of the multi-zone NAS parallel benchmarks. *J. Parallel Distrib. Comput.*, 66(5):674–685, May 2006.

[12] Lawrence Livermore National Laboratory. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). `https://computation.llnl.gov/casc/ShockHydro`. Last accessed April 3, 2013.

[13] Lawrence Livermore National Laboratory. ASC Sequoia Benchmark Codes. `https://asc.llnl.gov/sequoia/benchmarks`, 2012.

[14] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23:105–128, 2002.

[15] OpenMP Architecture Review Board. OpenMP application program interface, version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

[16] Oracle. Oracle Solaris Studio. `http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html`.

[17] M. Schulz et al. OpenSpeedShop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 16(2-3):105–121, Apr. 2008.

[18] S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.

[19] N. R. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–240, New York, NY, USA, 2009. ACM.

[20] N. R. Tallent, J. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proc. of the 2009 ACM SIGPLAN Conf on Programming Language Design and Implementation*, pages 441–452, New York, NY, USA, 2009. ACM.

[21] N. R. Tallent, J. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[22] The Portland Group. PGPROF Profiler Guide Parallel Profiling for Scientists and Engineers. `http://www.pgroup.com/doc/pgprofug.pdf`, 2011.

[23] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 392–403, New York, NY, USA, 1995. ACM.

[24] R. van der Pas. OpenMP Support in Sun Studio. `https://iwomp.zih.tu-dresden.de/downloads/3.OpenMP_Sun_Studio.pdf`, 2009.

[25] F. Wolf et al. Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In *Tools for High Performance Computing*, pages 157–167. Springer Berlin Heidelberg, 2008.