Effective Performance Measurement and Analysis of Multithreaded Applications

Nathan R. Tallent   John M. Mellor-Crummey
Rice University
{tallent,johnmc}@rice.edu

Abstract

Understanding why the performance of a multithreaded program does not improve linearly with the number of cores in a shared-memory node populated with one or more multicore processors is a problem of growing practical importance. This paper makes three contributions to performance analysis of multithreaded programs. First, we describe how to measure and attribute *parallel idleness*, namely, where threads are stalled and unable to work. This technique applies broadly to programming models ranging from explicit threading (*e.g.*, Pthreads) to higher-level models such as Cilk and OpenMP. Second, we describe how to measure and attribute *parallel overhead*—when a thread is performing miscellaneous work other than executing the user's computation. By employing a combination of compiler support and post-mortem analysis, we incur no measurement cost beyond normal profiling to glean this information. Using *idleness* and *overhead* metrics enables one to pinpoint areas of an application where concurrency should be increased (to reduce idleness), decreased (to reduce overhead), or where the present parallelization is hopeless (where idleness and overhead are both high). Third, we describe how to measure and attribute arbitrary performance metrics for high-level multithreaded programming models, such as Cilk. This requires bridging the gap between the expression of logical concurrency in programs and its realization at run-time as it is adaptively partitioned and scheduled onto a pool of threads. We have prototyped these ideas in the context of Rice University's HPCTOOLKIT performance tools. We describe our approach, implementation, and experiences applying this approach to measure and attribute work, idleness, and overhead in executions of Cilk programs.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Measurement techniques, Performance attributes.

General Terms Performance, Measurement, Algorithms.

Keywords Performance Analysis, Call Path Profiling, Multithreaded Programming Models, HPCTOOLKIT.

1. Introduction

Over the last several years, power dissipation has become a substantial problem for microprocessor architectures as clock frequencies have increased [19]. As a result, the microprocessor industry has shifted its focus from increasing clock frequencies to delivering increasing numbers of processor cores. For software to benefit from increases in core counts as new generations of microprocessors emerge, it must exploit threaded parallelism. As a result, there is an urgent need for programming models and tools to support development of efficient multithreaded programs. In this paper, we address the challenge of creating tools for measuring, attributing, and analyzing the performance of multithreaded programs.

Performance tools typically report how resources, such as time, are consumed rather than wasted. For parallel programs, it is typically most important to know where time is wasted as a result of an ineffective parallelization. To enable an average developer to quickly assess the quality of the parallelization in a multithreaded application, tools should pinpoint program regions where the parallelization is inefficient and quantify their impact on performance. Two aspects of a parallelization in particular are important for efficiency: whether there is adequate parallelism in the program to keep all of the processor cores busy, and whether the parallelism is sufficiently coarse-grain so that the cost of managing the parallelism does not become significant with respect to the cost of the parallel work. In this paper, we describe novel techniques for assessing both of these aspects of parallel efficiency.

For performance tools to be useful, they must apply to the multithreaded programming models of choice. Over the last decade, high-level programming models such as OpenMP [22] and Cilk [10] were developed to simplify the development of multithreaded programs. These programming models raise the level of abstraction of parallel programming by partitioning the problem into two parts: the programmer is responsible for expressing the logical concurrency in a program and a run time system is responsible for partitioning and mapping parallel work efficiently onto a pool of threads for execution. Without appropriate support for tools, the nature of this run-time mapping of work to threads is obscure and renders ineffective tools that measure and attribute performance directly to threads in the run-time system.

In our work, we focus on using *call path profiling* [12] to attribute costs in a program execution to the calling contexts in which they are incurred. For modular programs, it is important to attribute the costs incurred by each procedure to the different contexts in which the procedure is called. The need for context is obvious if one considers that string manipulation routines might be called from many distinct places in a program. Of particular interest is providing this capability for high-level multithreaded programming models such as Cilk. However, for high-level multithreaded parallel programming models, using call path profiling to associate costs with the context in which they are incurred is not as simple as it sounds. At each sample event, a call path profiler must attribute the cost represented by the sample (*e.g.*, time) to the current execution context, which consists of the stack of procedure frames active

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.
Copyright © 2009 ACM 978-1-60558-397-6/09/02...$5.00

when the event occurred. For programs written in a programming model such as Cilk, which uses a work-stealing run-time system to partition and map work onto a thread pool, the stack of native procedure frames active within a thread represents only a suffix of the calling context. Cilk's work-stealing run-time system causes calling contexts to become separated in space and time as procedure frames migrate between threads as work is stolen. As a result, a standard call path profile of a Cilk program during execution will show fragments of call paths mapped to each of the threads in the run-time system's thread pool. Since frames can be stolen, even the mapping between even an individual procedure frame and a thread may not be one to one. As a result, a standard call path profile of a Cilk program will yield a result that is at best cumbersome and at worst incomprehensible. For effective performance analysis of multithreaded programming models with sophisticated run-time systems, it is important to bridge the gap between the abstractions of the user's program and their realization at run time.

This paper makes the following contributions for understanding the performance of multithreaded parallel programs:

- A technique for measuring and attributing *parallel idleness*—when threads are idling or blocked and unable to perform useful work. This technique applies broadly to programming models ranging from explicit threading (*e.g.*, Pthreads [7]) to higher-level models such as Cilk, OpenMP and Threading Building Blocks [23]. The technique relies on minor modifications to the run-time systems of multithreaded programming models.
- A technique for measuring and attributing *parallel overhead*—when a thread is performing miscellaneous work other than executing the user's computation. This technique could be applied to both library-based programming models such as Pthreads and Threading Building Blocks, as well as compiler-based programming models such as Cilk and OpenMP. By employing a combination of compiler support and post-mortem analysis, we incur no measurement cost beyond normal profiling to glean this information.
- The definition of and a method for efficiently collecting *logical call path profiles*—a generalization of call path profiles that enables one to measure and correlate execution behavior at different levels of abstraction. We develop this approach here to relate the execution of a multithreaded program by a work-stealing run-time system back to its source-level representation.

We believe these complementary techniques are necessary for effective performance measurement and analysis of high-level multithreaded programming models. Logical call path profiles are the key for mapping measurements of work, idleness and overhead back to the source-level abstractions in high-level multithreaded parallel programming models. Our *idleness* and *overhead* metrics enable one to pinpoint areas of an application where concurrency should be increased (to reduce idleness), decreased (to reduce overhead), or where the present parallelization is hopeless (where idleness and overhead are both high). To show the utility of these techniques, we describe their implementation for Cilk and show how they bridge the gap between the execution complexity of a Cilk program and the relative simplicity of the Cilk programming model. Our tool attributes work, idleness, and overhead to Cilk source code lines in their full *logical* user-level calling context.

This paper is organized as follows. First, Section 2 describes parallel idleness and overhead. Then, Section 3 defines logical call path profiles while Section 4 shows how to obtain them using logical stack unwinding. Section 5 describes the application of these ideas to Cilk. Finally, Section 6 discusses related work and Section 7 concludes.

2. Pinpointing parallel bottlenecks

We describe two novel measurement and analysis techniques that enable an average developer to quickly determine whether a multithreaded application is effectively parallelized. If the application is not effectively parallelized, our techniques direct one's attention to areas of the program that need improvement.

2.1 Quantifying insufficient parallelism

To quantify insufficient parallelism, we describe how to efficiently and directly measure *parallel idleness*, *i.e.*, when threads are idle and unable to perform useful work. Our measurements of idleness are based on sampling of a time-based counter such as the wall clock or a hardware cycle counter. Measurement overhead is low and controllable by adjusting the sampling frequency. When a sample event occurs, a signal handler collects the context for the sample and associates the sample count with its context.[1] Collecting parallel idleness on a node with n processor cores requires minor adjustments to traditional time-based sampling. The first adjustment is to extend the run-time system to always maintain n_w and $n_{\overline{w}}$, the number of working and idle processor cores, respectively. This can be done by maintaining a node-wide counter representing n_w. When a core acquires a unit of useful work (*e.g.*, acquiring a procedure activation using work stealing or plucking a unit of work from a task queue), it atomically increments n_w. Similarly, when a core finishes a unit of work, it atomically decrements n_w to indicate that it is no longer actively working. In this scheme, $n_{\overline{w}} = n - n_w$.

Consider a run-time system that has one worker thread per core. On a sample, each thread receives an asynchronous signal, resulting in a per-thread sample event. If a sample event occurs in a thread that is not working, we ignore it. When a sample event occurs in a thread that is actively working, the thread attributes one sample to a *work* metric for the sample context. It then obtains n_w and $n_{\overline{w}}$ and attributes a fractional sample $n_{\overline{w}}/n_w$ to an *idleness* metric for the sample context. Even though the thread itself is not idle, it is critical to understand what work it is performing when 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.

For example, if three threads are active on a quad core processor, whenever a sample event for the cycle counter interrupts a working thread, the working thread will record one sample of work in its *work* metric, and $1/3$ sample of idleness in its *idleness* metric. The $1/3$ sample of idleness represents its share of the responsibility for the core that is sitting idle.

After measurement is completed, idleness can be computed for each program context. Since samples are accumulated during measurement, the idleness value for a given thread and context is $\sum n_{\overline{w}_i}/n_{w_i}$ over all samples i for that context. It is often useful to express this idleness metric as a percentage of the total idleness for the program. Total idleness may be computed post-mortem by summing idleness metric over all threads and contexts in the program. The idleness value may be converted to a time unit by multiplying by the sample period. One can also divide the idleness for each context by the application's *total effort*—the sum of work and idleness everywhere across all threads—to understand the fraction of total effort that was wasted in each context.

A variant of this strategy applies to situations where the number of threads n_T may not equal the number of cores (such as with Pthreads programs or OpenMP's nested parallelism). If $n_T > n$, then n_w may not exceed n; if $n_T < n$, then n_w cannot exceed n_T.

[1] As mentioned in Section 1, we attribute costs to their full calling context using call path profiling. In this section, we use the term *context* rather than *calling context* since idleness can be measured with or without full calling context.

parallel		
idleness	overhead	interpretation
low	low	effectively parallel; focus on serial performance
low	high	coarsen concurrency granularity
high	low	refine concurrency granularity
high	high	switch parallelization strategies

Table 1. Using parallel idleness and overhead to determine if the given application and input are effectively parallel on n cores.

2.2 Quantifying parallelization overhead

Parallel overhead occurs when a thread is performing miscellaneous work other than executing the user's computation. Sources of parallel overhead include costs such as those for synchronization or dynamically managing the distribution of work.

For library-based programming models such as Pthreads, identifying parallel overhead is easy: any time spent in a routine in the Pthreads library can be labeled as parallel overhead. For language-based parallel programming models, one must rely on compiler support to identify inlined sources of parallel overhead. A compiler for a multi-threaded programming model, such as OpenMP or Cilk, can tag statements in its generated code to indicate which are associated with parallelization overhead. In Section 5.2, we describe how we mark sources of parallel overhead for Cilk. In a post-mortem analysis, we recover compiler-recorded information about overhead statements, identify instructions associated with overhead statements and run-time library routines, and attribute any samples of work (as defined in Section 2.1) associated with them to parallelization overhead. The tags therefore partition the application code into instructions corresponding to either useful work or overhead (distinct from idleness).

A benefit of this scheme is that tags are only meta-information: they can be inserted and overhead can be associated with them using post-mortem analysis without affecting run time performance in *any* way. In addition, the tags may be refined to partition sources of overhead into multiple types. For example, it may be useful to distinguish between synchronization overhead and all other overhead. Such a refinement would provide more detailed information to users or analysis tools.

In particular, tags do *not* have any associated instrumentation. While the mapping between instructions and tags consume space, it need not induce any run time cost. For example, the mapping can be located within a section of a compiled binary that is not loaded into memory at run time or maintained in a separate file.

The tags we propose could take several forms, but one particularly convenient one is to associate overhead instructions with special unique procedure names within the line map. For example, synchronization code could be tagged with the special procedure or file name `parallel-overhead:sync`.

2.3 Analyzing efficiency

In a parallel program, we must consider two kinds of efficiency: parallel efficiency across multiple processor cores and efficiency on individual processor cores. With information about parallel idleness and overhead attributed hierarchically over loops,[2] procedures, and the calling contexts of a program, we can directly assess parallel efficiency and provide guidance for how to improve it (see Table 1). If a region of the program (*e.g.*, a parallel loop) is attributed with high idleness and low overhead, the granularity of the parallelism could profitably be reduced to enhance parallel efficiency. If the overhead is high and the idleness low, the granularity of the parallelism should be increased to reduce overhead. If the overhead is high and there is still insufficient parallelism, the parallelism is inefficient and no granularity adjustment will help; keeping the idle processors busy requires a different parallelization. For instance, one might use a combination of data and functional parallelism rather than one alone.

One can assess the efficiency of *work* and identify rate limiting factors on individual processor cores by using metrics derived from hardware performance counter measurements. Many different factors can limit an application's performance such as instruction mix, memory bandwidth, memory latency, and pipeline stalls. For each of these factors, information from hardware performance counters can be used to compute derived metrics that quantify the extent to which the factor is a rate limiter. Consider how to assess whether memory bandwidth is a rate limiter. During an execution, one can sample hardware counter events for *total cycles* and *memory bus transactions*. By multiplying the sampling period by the sample count for each instruction, one can obtain an estimate of how many bus transactions are associated with each instruction. By multiplying the number of bus transactions by the transaction granularity (*e.g.*, the line size for the lowest level cache), one can compute the amount of data transferred by each instruction. By dividing the amount of data transferred by instructions within a scope (*e.g.*, loop) by the total number of cycles spent in that scope, one can compute the memory bandwidth consumed in that scope. By comparing that with a model of peak bandwidth achievable on the architecture, one can determine whether a loop is bandwidth bound or not. Attributing metrics to static scopes such as loops and dynamic contexts such as call paths to support such analysis of multithreaded programs is the topic of the next section.

3. Logical call path profiles

To enable effective performance analysis of higher-level programming languages it is necessary to bridge the gap between the user's abstractions and their implementation. A key aspect of this is recovering user-level calling contexts. As mentioned previously, when Cilk programs execute, user-level calling contexts are separated in space and time by work stealing. Mapping measurements during execution back to a source program requires reassembling user-level contexts, which have been fragmented during execution. The next two sections extend the notion of call path profiling by defining *logical call paths* and describing how to generally and efficiently obtain logical call path profiles using a *logical calling context tree*. Logical call path profiling applies to both parallel and serial applications. In Section 5, we describe how this technique forms an essential building block for measurement and analysis of multithreaded Cilk program executions by a work-stealing run-time system.

3.1 Logical call paths

A sampling-based call path profiler obtains a call path by unwinding the call stack at a sample point to obtain a list of active procedure instances, or frames. Such a call path may not correspond directly to a user-level calling context. We introduce the notion of *logical* call paths to bridge this gap. We obtain *logical* call paths by *logically* unwinding the call stack. To support a precise discussion of this concept, we introduce and define the following terminology.

A *bichord* is a pair $\langle P_i, L_i \rangle$ consisting of a p-*chord* P_i and a l-*chord* L_i where each *p*-chord (or *l*-chord) is is a sequence of *p*-notes (*l*-notes), *e.g.*:

$$\langle P_i, L_i \rangle = \langle (p_{i,1}, \ldots, p_{i,m_1}), (l_{i,1}, \ldots, l_{i,m_2}) \rangle$$

[2] Because we collect performance metrics using statistical sampling of hardware performance counters, which associates counts directly with instructions, and use binary analysis to associate instructions with higher-level program structures such as loops, we can directly compute and attribute metrics at the level of individual loops.

A note represents a frame; a chord a grouping of frames; and a bichord the association of a group of physical stack frames (P_i) with a group of logical (L_i) stack frames. Logical frames correspond to a user-level calling context; physical frames correspond to an implementation-level realization of that view. The *p*-notes $P_i = (p_{1,1}, \ldots, p_{1,m_1})$ that form *p*-chord P_i represent the bichord's physical call path fragment, while the *l*-notes form the logical call path fragment. We say that the length $|P_i|$ of *p*-chord P_i, is the number of *p*-notes contained therein, *i.e.*, m_1 in the above example; similarly, $|L_i| = m_2$.

A *logical call path* is a sequence of bichords

$$\langle \langle P_1, L_1 \rangle, \langle P_2, L_2 \rangle, \ldots, \langle P_n, L_n \rangle \rangle$$

where $\langle P_1, L_1 \rangle$ is the program's entry point and where bichord $\langle P_n, L_n \rangle$ represents the innermost set of frames. It is natural to speak of the p-*chord projection* for the logical call path as

$$\langle P_1, \ldots, P_n \rangle$$

and the p-*note projection* as

$$\langle (p_{1,1}, \ldots, p_{1,m_1}), \ldots, (p_{n,1}, \ldots, p_{n,m_n}) \rangle$$

where $p_{1,1}$ represents the physical program entry point and the projection represents the physical call path from the entry point to the sample point. *Logical projections* are analogous.

To provide intuition for a discussion of bichord forms, it is useful to consider a concrete representation. We represent a *p*-note projection as a list of instruction pointers, one for each procedure frame active at the time a sample event occurs. The first instruction pointer of the unwind (p_{n,m_n}) is the program counter location at which the sample event occurred. The rest of the list contains the return address for each of the active procedure frames. Similarly, each *l*-note in a logical call path contains an opaque logical instruction pointer that represents the logical context.

Defining a logical call path to consist of a sequence of bichords formed of notes enables us to preserve interesting relationships between the physical and logical call path. To formalize these relationships, we first observe that a logical call path's *p*-note projection should always have a non-zero length because the physical stack is never empty. Moreover, intuitively, every *l*-chord must be associated with at least one *p*-note. This implies that no bichord should have a zero length *p*-chord. Equivalently, we observe that a *p*-note projection should not have 'gaps', *i.e.*, a machine cannot return to a 'virtual' logical frame — an *l*-note without an associated *p*-note — and then return back to a physical frame. From this starting point, we consider the possible relationships, or *associations*, between the lengths of a bichords's *p*-chord and *l*-chord. Given bichord $B_i = \langle P_i, L_i \rangle$, there are several possible associations between $|P_i|$ and $|L_i|$ that we describe with a member from the set $\{0, 1, \mathbf{M}\} \times \{0, 1, \mathbf{M}\}$, where $\mathbf{M}$ (a mnemonic for multi or many) represents any natural number $m \geq 2$. We are interested in the following four categories accounting for five of the possible association types:

1. $1 \leftrightarrow 1$. One *p*-note directly corresponds to one *l*-note—the typical case for C or Fortran code where a physical procedure frame corresponds to a logical procedure frame.
2. $1 \leftrightarrow 0$ and $\mathbf{M} \leftrightarrow 0$. A *p*-chord corresponds to an *empty* *l*-chord. This situation typically arises when run-time support code is executed. For example, a sample event that interrupts the run-time system's scheduler may find several physical frames that correspond to no logical procedure frame.
3. $\mathbf{M} \leftrightarrow 1$. This association often describes the run-time system implementing a high level user routine. For example, a Python interpreter may require a chain of procedure calls (several *p*-notes) to implement a user level call to sort a list.
4. $1 \leftrightarrow \mathbf{M}$. At first sight, this association may seem esoteric. However, it has important applications. It directly corresponds to using Cilk's scheduling loop as a proxy for walking the cactus stack of parent procedures that are stored in the heap and have no physical presence on the stack. As another example, a Java compiler could form one physical procedure from a 'hot' chain of user-level procedures.

Three observations are apropos. First, as previously discussed, associations $0 \leftrightarrow \{0, 1, \mathbf{M}\}$ are excluded meaning that the length of a *p*-chord is always non-zero. Second and in contrast, association (2) implies that it is possible to have a zero-length *l*-chord. The final omitted association, $\mathbf{M} \leftrightarrow \mathbf{M}$, can always be represented as some combination of categories (1-4) above.

We now concisely define a logical call path as a sequence of bichords $\langle \langle P_1, L_1 \rangle, \langle P_2, L_2 \rangle, \ldots, \langle P_n, L_n \rangle \rangle$ where $n \geq 1$ and $\forall i[|P_i| \geq 1]$, but where it is possible that $|L_i| = 0$ for any i.

3.2 Representing logical call path profiles

At run-time, we wish to efficiently obtain and represent a logical call path profile, *i.e.*, a collection of logical call paths annotated with sample counts with the time dimension removed. Our approach is to form a logical calling context tree—an extension of a *calling context tree* (CCT) [2]—that associates metric counts with logical call paths.

3.2.1 Weighted logical calling context trees

We first define a very simple logical CCT. Given a logical unwind

$$\langle \langle P_n, L_n \rangle, \langle P_{n-1}, L_{n-1} \rangle, \ldots, \langle P_1, L_1 \rangle \rangle$$

where $\langle P_n, L_n \rangle$ is a sample point, the straightforward extension of a CCT ensures that the path

$$\langle \langle P_1, L_1 \rangle, \langle P_2, L_2 \rangle, \ldots, \langle P_n, L_n \rangle \rangle$$

exists within the tree, where $\langle P_1, L_1 \rangle$ is the root of the tree and where $\langle P_n, L_n \rangle$ is a leaf node. Metrics such as sample counts are associated with each leaf node (sample point); in this example metrics at $\langle P_n, L_n \rangle$ are incremented.

We define the *physical projection* of a logical CCT to be the CCT formed by taking the *p*-chord projection of each call path in the logical CCT. The *logical projection* of a logical CCT is defined analogously.

3.2.2 Efficiently representing logical calling context trees

While this logical CCT representation is simple, treating bichords as atomic units can result in considerable space inefficiency. To reduce memory effects, we wish to share notes without losing any information represented in the logical CCT. The Appendix describes when sharing is possible and develops a more efficient and practical implementation.

4. Obtaining logical call path profiles

Given the definition of a logical call path and the representation of a call path profile using a logical calling context tree, we now turn our attention to obtaining a logical call path profile. To provide low controllable measurement overhead, we use statistical sampling and form the logical calling context tree by collecting and inserting logical call paths on demand for each sample. 'Physical' call path profilers use stack unwinding to collect the call path. Since the physical calling context alone is insufficient for obtaining the logical call path, we develop the more general the notion of *logical stack unwinding* to collect the logical call path.

4.1 Logical stack unwinding

Consider a contrived example where a Python driver calls a Java routine that calls a Cilk solver. Though unusual, this example shows that each bichord in a logical call path could potentially derive from a different run-time system. Because run-time systems use the system stack in their implementation, this suggests that the actual process of logical unwinding should be controlled by the physical stack. This is natural because although the physical call stack may represent the composition of calls from many different languages, it conforms to a known ABI. In addition, using a physical unwind naturally corresponds to our requirement that a *p*-note projection not have 'gaps', *i.e.*, there is at least one representative stack frame for each *l*-chord in the logical unwind. However, since a physical stack unwinder alone cannot determine either the association of the bichord or the length of the *p*-chord or the content of the *l*-chord, some sort of additional information must be available to construct the bichord. This information can be obtained using a language-specific plug-in or *agent* to assist a 'physical' stack unwinder. Each agent would understand its corresponding language implementation well enough to determine the particulars of reconstructing an *l*-chord given the *start* of a *p*-chord. It is important to emphasize a *p*-chord's *start* because assistance from the agent will in general be necessary to determine the *p*-chord's length, *e.g.*, 1 vs. $\mathbf{M}$.

There must be some way of selecting which agent to use at any point in the logical unwind. In the example above, one must know when to use the Cilk, Java and Python agents, respectively, to obtain the relevant bichords. Observe that at any point in the execution, the return address instruction pointer located in the stack frame should map to at most one run-time system and therefore one agent. Consequently, the frame's return address serves a proxy for the specific agent that should be consulted to assist formation of the bichord. During a program's execution, the mapping of code segments within the address space (the load map) can typically be determined by interrogating the operating system.

4.2 Thread creation contexts

Often it is useful to know the context in which a thread was created. The *creation context* of a thread is defined as the calling context at the time the thread was created. For example, consider a solver using fork-join parallelism where a pool of Pthreads is created using several calls to `pthread_create`. It is desirable to capture the calling context of the `pthread_create` so that the Pthread can be rooted within the context of the solver. The thread creation context may be captured and maintained as an extension to the thread's physical stack.

4.3 An API for logical unwinding

We have designed and implemented a general API for obtaining logical unwinds given language specific agents. Technically, there are two sub-APIs, one for collecting logical unwinds (using agents) and one describing the interface to which language specific agents must conform and the assumptions they may make.

The API for logical unwinding is designed to place as much burden as possible on the non-agent library routines so that agent implementation is as easy as possible. For example, an agent is not required to perform any look-ahead to determine the length of an *l*-chord. Although this information could be used by the logical unwinder (Algorithm 1) for allocating storage, we determined that it was more desirable to complicate the code for the unwinder than to complicate each agent's implementation. Consequently, the logical unwinder ensures that enough buffer space is always available to store a bichord. As another example, the agent interface sub-API promises a small amount of functionality to ease agent implementation, such as a means to inspect the address space and a safe memory allocator (`malloc` may not be safe).

Algorithm 1: logical-backtrace: performs a logical unwind

```
let $c$ be the unwind cursor, initialized with the machine
    context and language-specific logical unwind agents
while step-bichord(&$c$) $\neq$ END_UNWIND do
    let $a$ be the bichord's association (from $c$)
    while step-pnote(&$c$) $\neq$ END_CHORD do
        Record $p$-note (instruction pointer from $c$)
    while step-lnote(&$c$) $\neq$ END_CHORD do
        Record $l$-note (logical instruction pointer from $c$)
    Form bichord from $a$ and the lists of $p$-notes and $l$-notes
```

The logical unwinding API is divided into a two-level hierarchy corresponding to the division between bichords and notes. In particular, the top level addresses finding the bichords within a logical unwind while the other level targets finding the notes of a chord. An outline of of the backtrace routine is shown in Algorithm 1. Each level adopts semantics similar to libunwind [20]. This means that to find each bichord in the logical unwind $\langle\langle P_n, L_n\rangle, \langle P_{n-1}, L_{n-1}\rangle, \ldots, \langle P_1, L_1\rangle\rangle$,[3] n successive calls to step-bichord are required along with an additional call that returns a special value to indicate the unwind is completed. The advantage of these semantics is that they help ensure agents do not have to perform contextual look ahead. For example, to examine all *l*-notes within the *l*-chord $(l_{i,1}, \ldots, l_{i,m})$, $m+1$ calls are issued to step-lnote. This means that the agent need not know that $l_{i,1}$ is the last *l*-note in the *l*-chord unwind until the $m + 1^{\text{st}}$ call to step-lnote. This fact is particularly useful for an agent to a multi-threaded run time system because thread-specific state need not be maintained within the agent. Rather, all state for the unwind can be maintained by a fixed-sized thread-specific cursor allocated by the logical unwinder.

As discussed previously, logical unwinding is driven by a stack unwind. On each call to step-bichord, the library determines if a valid physical stack frame exists. If so, it extracts the return address instruction pointer and determines if it maps to any agent. If it does, that particular agent is used complete the discovery of the bichord. Otherwise, the 'identity' agent is used to create a 1 ↔ 1 bichord representing native code.

Observe that the asymmetry between *p*-chords and *l*-chords plays a critical role in the unwind process. For a *p*-chord P_i of length m_i, the $m_i + 1^{\text{th}}$ call to step-pnote both completes enumeration of P_i's *p*-notes and discovers the next *p*-chord. For example, consider a section of the physical projection representing *p*-chords P_i and P_{i+1}:

$$(\ldots, p_{i,m_i})(p_{i+1,1}, \ldots)$$

While iterating over the *p*-notes in *p*-chord P_i, we first issue m_i calls to step-pnote. On the $m_i + 1^{\text{th}}$ call, the agent discovers that there are no more *p*-notes in P_i, but only because it has found *p*-note $p_{i+1,1}$, the beginning of *p*-chord P_{i+1}. This means that the *p*-note portion of the cursor is pointing to the beginning of P_{i+1} *before* the cursor has stepped to P_{i+1}. This 'peeking' behavior is important because we must know the initial portion of P_{i+1} in order to know which agent to assign the responsibility of the next bichord. In contrast, step-lnote need not 'peek' ahead in to the next *l*-chord. Indeed, it should not because the next *l*-chord may be handled by a different agent and may have length 0.

5. Measurement and analysis of Cilk executions

To demonstrate the power of using our parallel idleness and overhead metrics in combination with logical call path profiling, we developed an implementation of a profiler for Cilk-5 [10] (currently

[3] A *logical unwind* is simply the reverse of a logical call path.

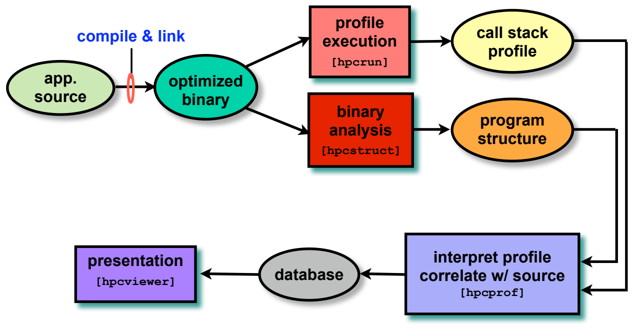


Figure 1. HPCToolkit workflow.

at version 5.4.6). We chose Cilk for several reasons. First, Cilk lets parallel programmers focus on specifying logical concurrency, while its run-time system handles the details of executing that logical concurrency efficiently. The power of Cilk's abstraction of logical concurrency is something that will be critical if programmers are to routinely write scalable multithreaded applications. (Indeed, Cilk is being developed into a commercial product.) Second, Cilk pioneered a sophisticated work-stealing scheduler that is provably efficient assuming the availability of sufficient concurrency. Third, the Cilk compiler and run-time implementations are freely available.

Our profiler implementation is part of HPCToolkit [1, 24], a performance toolkit whose workflow is shown in Figure 1. `hpcrun` (top, middle), a sampling-based call path profiler, measures the performance of fully-optimized executables. `hpcstruct` analyzes application binaries to recover program structure such as procedures and loop nests. `hpcprof` (bottom right) interprets call path profiles and correlates them with program structure, generating databases for interactive exploration with `hpcviewer`.

To support measurement and analysis of work, idleness, and parallel overhead in executions of multithreaded Cilk programs, we extended `hpcrun` to collect logical call path profiles for Cilk. In the following sections, we describe our approach, along with minor supporting modifications to the Cilk run-time system. After measurements are complete, we use logical calling contexts to correlate our measurements of work, idleness, and parallel overhead with the Cilk source program and interactively explore the performance data using `hpcviewer`.

5.1 Parallel work and idleness

To support measurement of our idleness metric, we modified the Cilk scheduler to classify threads as working or non-working and to maintain the number of working and idle threads (n_w and $n_{\overline{w}}$, respectively). These modifications were straightforward. Each worker thread executes a scheduling loop that acquires work (through a steal, if necessary) and then performs that work. Since the work is executed via a method call, the scheduling loop is 'exited' to perform the work and then re-entered as the worker thread waits to acquire more work. To identify a thread as actively working or idle, we set a thread-specific state variable just before the thread exits or enters the scheduling loop, respectively. At the same time, a global counter representing the number of working threads is atomically incremented or decremented as each thread exits and enters the scheduling loop, respectively. When a sample event interrupts a worker thread, one of two things happen: if the worker is idle, the sample event is ignored; if the worker is active, the logical call path for the work being executed is collected, one sample is attributed to the *work* metric metric total associated with this logical call path and a fractional sample $n_{\overline{w}}/n_w$ of idleness is added to the *idleness* metric associated with this logical call path.

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

Figure 2. Fragment of a Cilk program (Fibonacci numbers).

5.2 Parallel overhead

To attribute parallel overhead to logical calling contexts we use several mechanisms (described below) to identify all overhead inserted by the Cilk compiler into a Cilk application binary. At run-time, samples associated with parallel overhead will be attributed as *work* to the logical calling context in which they arise. After an execution of a Cilk program completes, in a post-mortem analysis phase we partition sample counts of the *work* into *useful work* and *parallel overhead* based on compile-time information.

Our strategy for identifying the parallel overhead within a Cilk application binary relies on the `hpcstruct` binary analysis tool for recovering program structure from a binary. `hpcstruct` analyzes an application binary to recover a mapping between object code and program structure. In particular, `hpcstruct` recovers the structure of procedures, including a procedure's loop nests, and identifies code that has been inlined therein. Thus, `hpcstruct` will naturally identify overhead-related code in a procedure if that code appears to have been inlined. We accomplish this is by using `#line` compiler directives to simulate inlining.

Given this overall strategy, we used two different methods to ease the implementation effort. The Cilk compiler compiles Cilk source code to C and then uses a vendor C compiler to generate an executable. It turns out that nearly all parallel overhead inserted into the intermediate C code by the Cilk compiler is encapsulated either by a call to a method or macro.[4] Consequently, it is possible to identify essentially all overhead by 1) tagging about 45 Cilk run time library routines with `#line` directives, and 2) inserting appropriate `#line` directives surrounding the appropriate macro references before the generated C code is fed to the vendor compiler.[5] Given this fact, and given our unfamiliarity with the Cilk compiler's source code, we determined that instead of modifying the compiler it would be easier to 1) appropriately tag the Cilk run time routines and 2) write a Cilk post-processor that inserted the appropriate tags in the intermediate C file. To preserve the ability to recover sensible structure for a routine and use a debugger with the resulting executable, our post-processor preserves the line number of the original source file. A sanitized example of an original Cilk routine and its corresponding post-processed C code is shown in Figures 2 and 3. (Note that the 'odd' formatting in the post-processed C, such as the declaration on the first line, is critical for aligning the line numbers of the generated code with the source.)

[4] Parallel overhead that derives neither from a method nor macro call is either continuation control flow, a declaration, or trivial.

[5] When a macro is expanded by the C preprocessor, no indication of its originating source file is typically recorded. In contrast, if a function call is inlined, a C compiler will effectively generate the appropriate `#line` directives.

```
int fib(WorkerState* ws, int n) { struct frame* fr;
#line 28 "hpctoolkit:parallel-overhead"
  CILK2C_INIT_FRAME(fr, ...);
  CILK2C_START_THREAD_FAST();
#line 28 "fib.cilk"

  if (n < 2) { int t = n;
#line 31 "hpctoolkit:parallel-overhead"
    CILK2C_BEFORE_RETURN_FAST();
#line 31 "fib.cilk"
    return t;}
  else {
    int x; int y;
    { fr->header.entry=1; fr->scope0.n = n;
#line 34 "hpctoolkit:parallel-overhead"
      CILK2C_BEFORE_SPAWN_FAST();
      CILK2C_PUSH_FRAME(fr);
#line 34 "fib.cilk"
      x = fib(ws, n-1);
#line 34 "hpctoolkit:parallel-overhead"
      CILK2C_XPOP_FRAME_RESULT(fr, 0, x);
      CILK2C_AFTER_SPAWN_FAST();
#line 34 "fib.cilk"
    }
    ...
```

Figure 3. Post-processed C fragment from the Cilk compiler (corresponding to Figure 2). Parallel overhead is demarcated with `#line` directives.

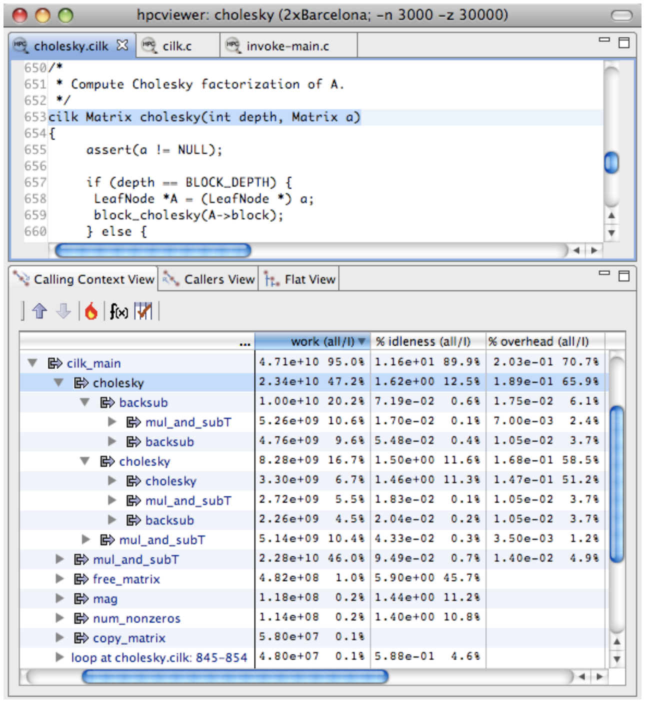
5.3 Cilk call path profiles

To attribute parallel idleness and overhead to logical calling contexts, we modified `hpcrun` to collect logical call path profiles for Cilk. In particular, we implemented the logical unwind API (described in Section 4.3), developed a Cilk-specific agent, and modified `hpcprof`'s profile interpretation and source code correlation to normalize the results. The design of the Cilk agent illustrates several important points. Although discussing this agent necessarily involves details about the Cilk implementation, it is important to note that the API remains language independent. For example we are working on agents for other models such as OpenMP.

To understand the Cilk agent, it is necessary to review some high-level details about the Cilk-5 implementation. For each source Cilk routine, the Cilk compiler generates two clones, a 'fast' and 'slow' version. The fast clone is very similar to the corresponding C procedure, and is executed in the common case. Importantly, whenever a procedure is spawned, the fast version is executed. The slow clone is executed only when parallel semantics are necessary such as when a procedure is stolen.

Each worker thread maintains a deque (stored in the heap) of ready procedure instances, which together form a 'Cactus stack', *i.e.*, a tree where the root corresponds to the bottom (outermost frame) of the stack. Local work is pushed and popped from the tail of the deque (top or inner frames) while thieves steal from the head (bottom or outer frames). Execution proceeds on the thread's stack even though a 'shadow' continuation is maintained on the deque. Whenever a thief steals a procedure's continuation, it resumes it using the slow version of that procedure. Since frames may only be stolen from the deque's head (bottom of cactus stack), this implies that the descendants of a fast procedure may only be fast procedures themselves.

We may infer the following invariants about the frames on a worker's stack (in top-down order):



```
/*
 * Compute Cholesky factorization of A.
 */
cilk Matrix cholesky(int depth, Matrix a)
{
    assert(a != NULL);

    if (depth == BLOCK_DEPTH) {
     LeafNode *A = (LeafNode *) a;
     block_cholesky(A->block);
    } else {
```

...	work (all/I) ▼	% idleness (all/I)	% overhead (all/I)
▼ cilk_main	4.71e+10 95.0%	1.16e+01 89.9%	2.03e-01 70.7%
▼ cholesky	2.34e+10 47.2%	1.62e+00 12.5%	1.89e-01 65.9%
▼ backsub	1.00e+10 20.2%	7.19e-02 0.6%	1.75e-02 6.1%
▶ mul_and_subT	5.26e+09 10.6%	1.70e-02 0.1%	7.00e-03 2.4%
▶ backsub	4.76e+09 9.6%	5.48e-02 0.4%	1.05e-02 3.7%
▼ cholesky	8.28e+09 16.7%	1.50e+00 11.6%	1.68e-01 58.5%
▶ cholesky	3.30e+09 6.7%	1.46e+00 11.3%	1.47e-01 51.2%
▶ mul_and_subT	2.72e+09 5.5%	1.83e-02 0.1%	1.05e-02 3.7%
▶ backsub	2.26e+09 4.5%	2.04e-02 0.2%	1.05e-02 3.7%
▶ mul_and_subT	5.14e+09 10.4%	4.33e-02 0.3%	3.50e-03 1.2%
▶ mul_and_subT	2.28e+10 46.0%	9.49e-02 0.7%	1.40e-02 4.9%
▶ free_matrix	4.82e+08 1.0%	5.90e+00 45.7%	
▶ mag	1.18e+08 0.2%	1.44e+00 11.2%	
▶ num_nonzeros	1.14e+08 0.2%	1.40e+00 10.8%	
▶ copy_matrix	5.80e+07 0.1%		
▶ loop at cholesky.cilk: 845-854	4.80e+07 0.1%	5.88e-01 4.6%	

Figure 4. A Calling Context (top-down) view of Cholesky.

A. There may be i frames corresponding to Cilk run time routines (*e.g.*, creation of continuation information) or user level C routines. Cilk run time routines correspond to a bichord with association $1 \leftrightarrow 0$ (since they are not part of the logical call path), while user-level C routines correspond to an association of $1 \leftrightarrow 1$.

B. There may be j frames corresponding to Cilk fast frames. Since the fast clone of a Cilk routine directly corresponds to a physical frame and a logical frame, the pair corresponds to a bichord with association $1 \leftrightarrow 1$.

C. There is always at least one frame corresponding to the Cilk scheduler.

These segments may not be interchanged.

The exact interpretation of segment C depends upon whether there are additional ancestor frames in the Cactus stack. That is, when a worker steals any procedure other than 'main', that procedure's logical context is represented as a chain of ancestor frames within the Cactus stack. In this case, the scheduler frame has association $1 \leftrightarrow \mathbf{M}$. Otherwise, if the innermost frame in segment B corresponds to 'main', which has no logical calling context, the scheduler frame has association $1 \leftrightarrow 0$.

5.4 Case study

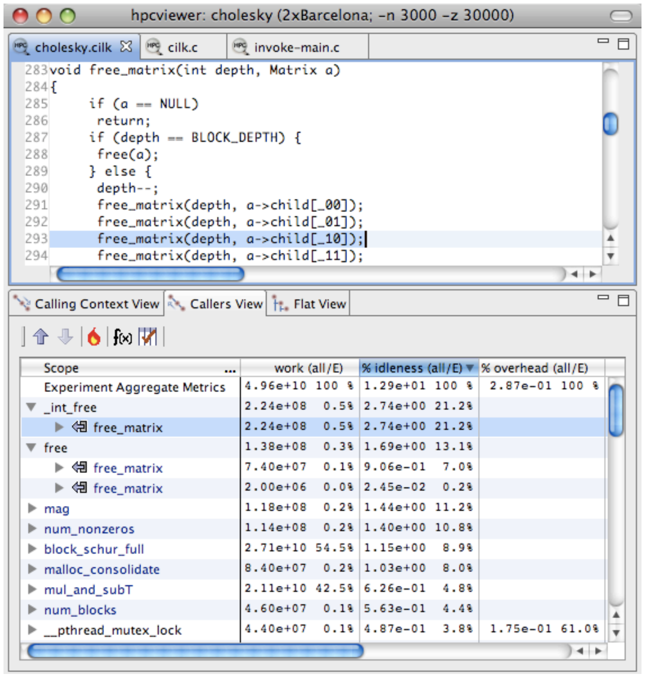
To demonstrate the power of attributing work, parallel idleness and parallel overhead to logical call path profiles, we apply our method to analyze the performance of a Cilk program for Cholesky decomposition. We used the example Cholesky program in the Cilk 5.4.6 source distribution. We ran a problem size of 3000 × 3000 (30,000 non-zeros) on an SMP with dual quad-core AMD Opterons (2360 SE, 2.5 Ghz) and 4 GB main memory. We profiled the execution using `hpcrun`, which gathers separate data for each thread, and processed the results using `hpcprof`.

Figure 4 presents one view of the aggregated results displayed by `hpcviewer`. The view has three main components. The navigation pane (lower left sub-pane) shows a top-down view of the calling context tree, partially expanded. One can see several user-level procedure instances along the call paths. (Physical procedure instances are not shown.) The selected line in the navigation pane and the source pane (top sub-pane) shows the procedure `cholesky`. Each entry in the navigation pane is associated with metric values in the metric pane to the right. Sibling entries are sorted with respect to the selected metric column (in this case 'work (all/I)'). Observe at the bottom of the navigation pane a loop, located within the context of `cilk_main`. The loop is detected by `hpcstruct`'s program structure analysis; the navigation pane actually contains a fusion of the dynamic logical calling contexts and the `hpcstruct`'s static context information.

The metric columns in Figure 4 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. In the idleness and overhead columns, the values in scientific notation represent the aforementioned percentages; the values shown as percentages to their right give an entry's proportion of the total idleness or overhead, respectively. The metrics are *inclusive* (hence the 'I' qualifier) in the sense that they represent values for the associated procedure instance in addition to all of its callees. Thus, the metric name 'work (all/I)' means inclusive work summed over all threads.

Because Cilk-5 emphasizes recursive decompositions of algorithms — parallelism is exposed through asynchronous procedure calls — call chains can become quite long. Nevertheless, expanding the calling context tree to the first call to `cholesky` and noting the metrics on the right is very informative. Figure 4 shows that about 47.2% of of the total work of the program 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 46.0%. We can also quickly see that about 12.5% and 65.9% of the total parallel idleness and overhead, respectively, occur in `cholesky`. However, because this idleness and overhead are small with respect to effort (about 1.62% and 0.189%, respectively), it is clear that the parallelization of `cholesky` is very effective for this execution. In contrast, the parallelization of the entire program (for which we can use `cilk_main` as a proxy) is less effective, with overhead essentially remaining the same, but idleness accounting for about 11.6% of total effort.

To pinpoint exactly where inefficiency occurs using the idleness and overhead metrics, we turn to the 'Callers' or bottom-up view in Figure 5. If the top-down view looks 'down' the call chain, the *bottom-up* view looks 'up' to a procedure's callers. Thus at the first level, the bottom-up view lists all the procedures in the program, rank-ordered according to the selected metric—in this case, relative idleness, the most troubling inefficiency. Note that in contrast to Figure 4, these metric values are 'exclusive' (signified with an 'E') in the sense that they do not include values for a procedure's callees. The top two routines in the rank-ordered list are versions of the C library routine `free` and together account for about 34.3% of the program's idleness. When the callers for these routines are expanded, it is evident that they are both called by `free_matrix`, a *non*-Cilk, *i.e.*, *serial*, helper routine that deallocates the matrix for the Cholesky driver. Continuing down the list reveals that every routine shown in the screen shot except `mul_and_subT` is a serial helper. Since each of these serial routines except `block_schur_full` is related to initialization or finalization, it is immediately evident that to reduce parallel idleness either the size of the matrix must be increased or the initialization and finalization routines must be parallelized. The significance of this conclusion is that *without having any prior knowledge of the source code*, our techniques have enabled us to quickly make strong and precise statements about the parallel efficiency of this program. Although it is not surprising that serial code is responsible for idleness, the fact that we can immediately quantify and pinpoint its impact on parallel efficiency shows the effectiveness of our methods.


hpcviewer: cholesky (2xBarcelona; -n 3000 -z 30000)

```
void free_matrix(int depth, Matrix a)
{
     if (a == NULL)
      return;
     if (depth == BLOCK_DEPTH) {
      free(a);
     } else {
      depth--;
      free_matrix(depth, a->child[_00]);
      free_matrix(depth, a->child[_01]);
      free_matrix(depth, a->child[_10]);
      free_matrix(depth, a->child[_11]);
```

Scope	work (all/E)	% idleness (all/E)	% overhead (all/E)
Experiment Aggregate Metrics	4.96e+10 100 %	1.29e+01 100 %	2.87e-01 100 %
▼ _int_free	2.24e+08 0.5%	2.74e+00 21.2%	
▶ free_matrix	2.24e+08 0.5%	2.74e+00 21.2%	
▼ free	1.38e+08 0.3%	1.69e+00 13.1%	
▶ free_matrix	7.40e+07 0.1%	9.06e-01 7.0%	
▶ free_matrix	2.00e+06 0.0%	2.45e-02 0.2%	
▶ mag	1.18e+08 0.2%	1.44e+00 11.2%	
▶ num_nonzeros	1.14e+08 0.2%	1.40e+00 10.8%	
▶ block_schur_full	2.71e+10 54.5%	1.15e+00 8.9%	
▶ malloc_consolidate	8.40e+07 0.2%	1.03e+00 8.0%	
▶ mul_and_subT	2.11e+10 42.5%	6.26e-01 4.8%	
▶ num_blocks	4.60e+07 0.1%	5.63e-01 4.4%	
▶ __pthread_mutex_lock	4.40e+07 0.1%	4.87e-01 3.8%	1.75e-01 61.0%

Figure 5. A Callers (bottom-up) view of Cholesky.

6. Related work

Our parallel idleness metric is similar to Quartz's [3] notion of 'normalized time' to highlight code with poor concurrency. Normalized time is computed by attributing $1/n_w$ (using the notation from Section 2.1) to the relevant section of code on each sample of a working thread, inflating compute times in areas of poor parallelization. While our idleness metric is similar in that it also highlights code sections with poor concurrency, it is different in that it is a direct measure of parallel idleness: $n_{\overline{w}}/n_w$. This quantitative/qualitative distinction is important because Quartz's qualitative metric can be ambiguous. Consider a program that executes with n threads (on n cores) with two phases named X and Y, where each phase executes for an equal amount of time, t. During phase X, procedure x executes serially; during phase Y, n instances of procedure y execute without any loss to overhead. Unintuitively, the normalized times $\|T_{\mathsf{x}}\|$ and $\|T_{\mathsf{y}}\|$ for procedures x and y are identical ($t/1$ and nt/n, respectively) even though $n-1$ threads are idle for the whole duration of phase X. In contrast, our idleness metric would yield values of $\mathcal{I}_{\mathsf{x}} = (n-1)t$ and $\mathcal{I}_{\mathsf{y}} = 0$. Although Quartz eliminates this ambiguity by using n counters for each procedure, assigning t to counter x_1 and 0 to counters $\mathsf{x}_2 \ldots \mathsf{x}_n$, this solution requires a comparison between n counters to convey the same thing as $\mathcal{I}_{\mathsf{x}}$. Additionally, we attribute idleness to full logical calling contexts, even in the presence of a work-stealing run time.

The idea of computing parallel overhead is not new. For example, 'cycle accounting' is a powerful methodology for partitioning stall cycles during the execution of serial code [9, 17]. To predict parallel performance, Crovella and LeBlanc describe a 'lost cycles analysis' [8] that separates parallel overhead from pure computation. They further divide parallel overhead into sub-categories useful for differentiating between different performance problems. However, they lament that "[m]easuring lost cycles directly for the entire environment space is still impractical." Our method directly measures parallel overhead without any run-time cost.

Several tools for obtaining call path profiles have been developed, but they collect only physical call path profile projections [4, 11, 13, 18, 21] or logical (user-level) call path profile projections, such as for Java [5, 25, 26]. In parallel but independent work, Itzkowitz *et al.* describe an OpenMP API that enables a statistical call path profiler to correlate user-level call paths with run-time metrics about whether a thread is working or waiting [16]. Our work is more general in the sense that we define logical call path profiles, explain how they can be efficiently represented, and describe a general API for obtaining them. Although the two idleness metrics are similar, we additionally collect and attribute a parallel overhead metric without any run-time cost.

It is interesting to compare our performance analysis of Cilk to Cilk's own performance metrics. Cilk computes two metrics that directly correspond to the theoretical model that underlies Cilk's provably-efficient scheduler. The first is total work or the time for a serial execution of the program with a given input. The second is critical path, or a prediction of the execution time on an infinite number of processors. The significant advantages of Cilk's metrics are that they are 'platform independent' and provide a theoretical upper bound on the scalability of a program with a given input. However, they share two important disadvantages. First, Cilk's metrics are computed using extremely costly instrumentation — which itself disturbs the application's performance characteristics. Second, these metrics do not aid the programmer in pinpointing *where* in the source code inefficiency arises. In contrast, our method *immediately pinpoints parallel inefficiency in user-level source code*. Moreover, paired with hardware performance counter information, our method can help distinguish between different types of architectural bottlenecks in different regions of code.

Critical path is a classic metric for understanding parallel programs. While Cilk computes the critical path's lower bound for a program and given input, it is also possible to determine the actual critical path for an execution. Intel's VTune [15] computes the actual critical path for an execution, though at the native thread level. The classic problem with critical path information is that after expending much effort to reduce its cost, a completely different critical path may emerge, slightly less costly than the original. Therefore, it is much more useful to know how much 'slackness' exists in the critical path. Intel's Thread Profiler [6, 14] not only computes critical path but classifies its segments by concurrency level and thread interaction. Given a segment where n_T threads execute on n cores ($n > 1$), the tool classifies that segment's concurrency level as either serial ($n_T = 1$), under-subscribed ($1 < n_T < n$), fully parallel ($n_T = n$), or oversubscribed ($n_T > n$). These categories are then qualified by three interaction effects such as cruise, impact and blocking. *Cruise time* is time that a thread does not delay the next thread on the critical path while *impact time* is the opposite. If a thread on the critical path waits for some external event, it accumulates *blocking time*. Thus, performance tuners should focus on areas of serial or under-subscribed impact time rather than fully parallel cruise time. The disadvantages of Thread Profiler are that it uses costly instrumentation, reports information at the native (Win32) thread level, and does not provide contextual information.

An interesting observation about our idleness and overhead metrics is that, in the context of Cilk, they approximate a quantitative measure of critical path slackness, tied to full calling context. To see this, note that a Cilk worker thread is idle only if it is waiting for another worker thread to 1) make asynchronous calls or 2) release a lock. Therefore, if a thread's idleness is high in a certain context, then that context was on one of the 'interesting' critical paths. One deficiency of our profile data is that it does not distinguish between idleness (or overhead) that is the result of a few calls to a long-running function as opposed to many calls to a fast one. However, given the properties of the Cilk scheduler, we can compute metrics similar to Thread Profiler's but for a fraction of the overhead.

7. Conclusions

Because of the growing need to develop applications for multicore architectures, effective tools for quantifying and for pinpointing performance bottlenecks in multithreaded applications are absolutely essential. This will be increasingly true as less skilled application developers are forced to write parallel programs to benefit from increasing core counts in emerging processors.

We have shown that attributing work, parallel idleness and parallel overhead to *logical* calling contexts enables one to quickly obtain unique insight into the run-time performance of Cilk programs. In particular, we demonstrated the power of our method by using it to pinpoint and quantify serialization in a Cilk execution. A strength of our approach is that our performance metrics are completely intuitive and can be mapped back to the user's programming abstractions, even though the run-time realization of these abstractions is significantly different. While we described a prototype tool for measurement and analysis of multithreaded programs written in Cilk, our underlying techniques for computing parallel idleness, parallel overhead, and obtaining logical call path profiles are more general and can be applied directly to other multithreaded programming models such as OpenMP and Threading Building Blocks.

Our work shows that it is possible to construct effective and efficient performance tools for multithreaded programs. The run-time cost of our profiling can be dialed down arbitrarily low by reducing the sampling frequency. We have also 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.

Acknowledgments

Development of the HPCToolkit performance tools is supported by the Department of Energy's Office of Science under cooperative agreements DE-FC02-07ER25800 and DE-FC02-06ER25762. HPCToolkit would not exist without the contributions of the other project members: Laksono Adhianto, Michael Fagan, Mark Krentel, and Gabriel Marin. Project alumni include Robert Fowler and Nathan Froyd.

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. Technical Report TR08-06, Rice University, 2008.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, New York, NY, USA, 1997. ACM Press.

[3] T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. *SIGMETRICS Perform. Eval. Rev.*, 18(1):115–125, 1990.

[4] Apple Computer. Shark. `http://developer.apple.com/tools/sharkoptimize.html`.

[5] W. Binder. Portable and accurate sampling profiling for Java. *Softw. Pract. Exper.*, 36(6):615–650, 2006.

[6] C. P. Breshears. Using Intel Thread Profiler for Win32 threads: Philosophy and theory. http://software.intel.com/en-us/articles/using-intel-thread-profiler-for-win32-...threads-philosophy-and-theory, August 2007.

[7] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[8] M. E. Crovella and T. J. LeBlanc. Parallel performance using lost cycles analysis. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 600–609, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[9] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. *SIGPLAN Not.*, 41(11):175–184, 2006.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[11] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proceedings of the 19th annual International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM Press.

[12] R. J. Hall. Call path profiling. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 296–306, New York, NY, USA, 1992. ACM Press.

[13] Intel Corporation. Intel performance tuning utility. Linked from http://whatif.intel.com/.

[14] Intel Corporation. Intel thread profiler. http://www.intel.com/software/products/tpwin.

[15] Intel Corporation. Intel VTune performance analyzers. http://www.intel.com/software/products/vtune/.

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

[17] D. Levinthal. Execution-based cycle accounting on Intel Core 2 Duo processors. http://www.devx.com/go-parallel/Link/33315.

[18] J. Levon *et al.* OProfile. http://oprofile.sourceforge.net/.

[19] M. Monchiero, R. Canal, and A. Gonzalez. Power/performance/thermal design-space exploration for multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 19(5):666–681, May 2008.

[20] D. Mosberger-Tang. libunwind. http://www.nongnu.org/libunwind/.

[21] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 143–152, New York, NY, USA, 2007. ACM.

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

[23] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, Sebastopol, CA, 2007.

[24] Rice University. HPCToolkit performance tools. http://hpctoolkit.org.

[25] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani. An efficient online path profiling framework for Java just-in-time compilers. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 148, Washington, DC, USA, 2003. IEEE Computer Society.

[26] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 263–271, New York, NY, USA, 2006. ACM.

Appendix: Efficiently representing logical CCTs

Recall that Section 3.2.1 defined a logical calling context tree (L-CCT) as a tree of bichords. Accordingly, two distinct call paths in the tree may be partially shared if and only if they they share a common prefix of bichords. (All paths share a common root.) One issue that arises during a straight-forward implementation of L-CCTs is that common notes between multiple bichords are unnecessarily duplicated. We illustrate this problem with an example.

Suppose over the course of several samples, we obtain several logical unwinds of the forms below (where inner frames are on the left and a sample point, if relevant, is underlined):

$$\ldots \langle (p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{1}$$

$$\langle (\underline{p'_{i,\mathsf{b}}}, p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{2}$$

$$\langle (\underline{p'_{i,\mathsf{c}}}, p_{i,\mathsf{b}}, p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{3}$$

$$\ldots, \langle (p_{i,\mathsf{c}}, p_{i,\mathsf{b}}, p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{4}$$

$$\langle (\underline{p'_{i,\mathsf{a}}}), (l_{i,1}) \rangle, \ldots \tag{5}$$

$$\ldots, \langle (p_{i,\mathsf{e}}, p_{i,\mathsf{f}}, p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{6}$$

$$\ldots, \langle (p_{i,\mathsf{c}}), (l_{i,1}) \rangle, \langle (p_{i,\mathsf{b}}), (l_{i,1}) \rangle, \langle (p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{7}$$

$$\ldots \langle (p_{i,\mathsf{a}}), (l_{j,1}) \rangle, \ldots \tag{8}$$

$$\ldots \langle (p_{i,\mathsf{a}}), (l_{i,2}, l_{i,1}) \rangle, \ldots \tag{9}$$

Unwinds (1)–(6), with bichords of association $\mathbf{M} \leftrightarrow 1$ and $1 \leftrightarrow 1$, could represent an interpreter implementing a high-level logical operation, signified by *l*-note $l_{i,1}$. Although none of these bichords are equal, all share $l_{i,1}$; and all but (5) share $p_{i,\mathsf{a}}$. However, a L-CCT treats each bichord as an atomic unit, thereby requiring that any common notes be duplicated when the corresponding call paths are inserted into the L-CCT. (Even the bichords in Unwinds (3) and (4) must be distinct because the former contains a sample and should therefore be a leaf node.) In general, the $\mathbf{M}$-portion of these bichords may be long and the frequent sample rate identifies most, if not all, of the unique prefixes. An analogous situation occurs in our Cilk profiler, where the root bichord of (almost) all call paths has association $1 \leftrightarrow \mathbf{M}$. As a result, several seemingly unnecessary *p*-notes exist with the L-CCT. For compact representation of an L-CCT, it is desirable to know when it is both possible and profitable to share the notes of two bichords.

Terminology

Observe that some associations are naturally related. For example, $1 \leftrightarrow 0$ is the natural 'base case' of $\mathbf{M} \leftrightarrow 0$. Similarly, $1 \leftrightarrow 1$ is the natural 'base case' of both $1 \leftrightarrow \mathbf{M}$ and $\mathbf{M} \leftrightarrow 1$. We therefore define the following *association classes*:

- $\mathcal{A} \leftrightarrow 0 = \{1 \leftrightarrow 0, \mathbf{M} \leftrightarrow 0\}$
- $\mathcal{A} \leftrightarrow 1 = \{1 \leftrightarrow 1, \mathbf{M} \leftrightarrow 1\}$
- $1 \leftrightarrow \mathcal{A} = \{1 \leftrightarrow 1, 1 \leftrightarrow \mathbf{M}\}$

Let the functions ip and lip return the physical and logical instruction pointers given a *p*-note or *l*-note, respectively. The functions assoc and $\mathsf{assoc\text{-}class}$ return the association and association-class of a bichord, respectively. For convenience, we also define $\mathsf{assoc\text{-}class}{=}$ to test whether two bichords have identical association classes, respectively.

Sharing within bichords

We first consider the limits of sharing within bichords. Sharing between any two bichords may either be full or partial. If two paths partially share a bichord, they may still be able to partially share another bichord (*cf.* Unwinds (4) and (7)). However, partially sharing either bichord requires that the paths diverge in some fashion (otherwise they would be equal). Additional sharing requires that paths merge again, turning the tree into a graph and creating ambiguous calling contexts. Therefore, two bichords may be partially shared only if they are both roots of their respective call paths or their respective call path predecessors are fully shared. After partial sharing, paths must diverge.

The next task is to clearly define when partial sharing may occur between two bichords $B_x = \langle P_x, L_x \rangle$ and $B_y = \langle P_y, L_y \rangle$. We divide the analysis into two cases.

Case 1. $P_x = P_y$ or $L_x = L_y$. Without loss of generality assume the latter.

- $\mathsf{assoc\text{-}class}{=}(B_x, B_y)$: Compare Unwinds (1)–(6). Although these bichords represent at least three fully distinct contexts and two different associations, they have identical association classes. Each *p*-chord (except (5)) has a common prefix beginning with *p*-note $p_{i,\mathsf{a}}$. In general, several other types of non-prefix sharing are possible (*e.g.*, suffixes). However, prefix sharing naturally corresponds to tree structure whereas non-prefix sharing effectively requires that a path diverges, skips one or more *p*-notes, and then re-merges.

  Therefore we formulate the prefix condition for partially sharing two bichords B_x and B_y:

  - $((P_x \sqsubset P_y) \vee (P_y \sqsubset P_x))$ and $L_x = L_y$
  - $P_x = P_y$ and $((L_x \sqsubset L_y) \vee (L_y \sqsubset L_x))$ (by symmetry)

  where $=$ and $\sqsubset$ ('strict prefix') are defined with respect to the sequence of notes that form a chord.

  The one issue is that B_x and B_y may have different associations; prefix sharing is not effective if associations must be duplicated. However, because we know the bichord's association classes are identical, we know that if their associations are different, one association must be the 'base case' of the other. For example, Unwinds (1) and (2) have associations $1 \leftrightarrow 1$ and $\mathbf{M} \leftrightarrow 1$, respectively. We show below how to implement an implicit 'base-case flag' that preserves this information.

  It turns out that the prefix condition can be relaxed slightly. Consider Unwinds (2) and (3), which may share *p*-note $p_{i,\mathsf{a}}$ by the above condition. Observe that $p'_{i,\mathsf{b}}$ represents a sample point while $p_{i,\mathsf{b}}$ represents a call site. Although in general $\mathsf{ip}(p'_{i,\mathsf{b}}) \neq \mathsf{ip}(p_{i,\mathsf{b}})$, a sample can be taken at a call site (technically, a return address), meaning that it is possible that $\mathsf{ip}(p'_{i,\mathsf{b}}) = \mathsf{ip}(p_{i,\mathsf{b}})$. We show below how to implement an implicit 'sample-point flag' that enables us to extend the prefix condition to allow sharing in this case. The flag indicates that the note both is and is not a sample point.

- $\mathsf{assoc\text{-}class}{\neq}(B_x, B_y)$: An enumeration of the possibilities for B_y for each of the five possible associations for B_x shows that this case is impossible (by the assumption $L_x = L_y$).

Case 2. $P_x \neq P_y$ and $L_x \neq L_y$.

- $\mathsf{assoc\text{-}class}{=}(B_x, B_y)$: Note that neither association may be in association class $\mathcal{A} \leftrightarrow 0$; otherwise $L_x = L_y$.

  We now consider the two other association classes and focus, without loss of generality, on $\mathcal{A} \leftrightarrow 1$. There are three cases. First, both bichords may have association $1 \leftrightarrow 1$. Second, one bichord has association $1 \leftrightarrow 1$ and the other $\mathbf{M} \leftrightarrow 1$. Third, both bichords have association $\mathbf{M} \leftrightarrow 1$.

  In the first case, no sharing is possible (since neither chord is equal). In the second and third cases, prefix sharing among *p*-notes may be possible. However, *l*-notes must be duplicated to maintain distinct logical calling contexts (*cf.* Unwinds (2) and (8)). Therefore, partial sharing is not profitable.

- $\mathsf{assoc\text{-}class}{\neq}(B_x, B_y)$: Since association classes are fully distinct, partial sharing is not possible without duplicating association information (*cf.* Unwinds (2) and (9)).

Implementation

We now translate the above conclusions into a practical implementation for the L-CCT.

We maintain the two-level distinction between bichords and notes implicitly. A bichord is represented by a list of X-structures. Each X contains an association (assoc) and a physical and logical instruction pointer (ip and lip, respectively). Given a bichord $\langle P_x, L_x \rangle$, we need n Xs $X_1, \ldots, X_n$ where $n = \mathsf{max}(|P_x|, |L_x|)$ and where X_1 represents the outermost portion of the bichord. Let the function note-id return the index of an X-structure within a bichord: $\mathsf{note\text{-}id}(X_j) = j$.[6] Note that $\mathsf{ip}(X_j) = \mathsf{NIL}$ if $|P_x| < j \leq n$; similarly for $\mathsf{lip}(X_k)$.

Given this representation, a logical call path is simply a list of X-structures $X_1, \ldots, X_n$. A bichord begins at every X_i where $\mathsf{note\text{-}id}(X_i) = 1$. A L-CCT is a tree of X-structures. Each X in the L-CCT may have a vector of metric values. A non-zero metric count naturally implements the 'sample-point flag' mentioned above. To implement the 'base-case flag', we simply ensure that when a $1 \leftrightarrow 1$ bichord shares the root of, say, an $\mathbf{M} \leftrightarrow 1$ bichord, the root X has association $1 \leftrightarrow 1$. Thus, the bichords in Unwinds (1) and (2) would be represented as two Xs $\ldots X_1, X_2 \ldots$ where $\mathsf{assoc}(X_1) = 1 \leftrightarrow 1$, $\mathsf{assoc}(X_2) = \mathbf{M} \leftrightarrow 1$; where X_2 has a non-zero metric value; and where X_1 is an interior node.

The final item is to describe an efficient way to insert a logical call path into the L-CCT in a way that corresponds to the full and partial sharing of bichords described above. To ensure the L-CCT is rooted, we prefix a synthetic root node to the beginning of every call path, implying that every call path has a length of at least two. Inserting a path into the L-CCT therefore turns into the following problem: Given the call path fragment $m' \rightarrow n'$ (as X-structures) and given a node m in the L-CCT such that $m' = m$, is it the case that $\exists n$ such that n is a child of m and $\mathsf{sharable?}(n, n')$ holds? If the answer is yes, n may be shared and insertion proceeds to the children of n and n'. Otherwise, a new path for n is spliced into the tree.

To define sharable?, we first consider a physical calling context tree where X-structures only contain a physical instruction pointer (ip). In this case we simply have:

$$\mathsf{sharable?}(n, n') : \mathsf{ip}{=}(n, n')$$

To extend this definition to a L-CCT, we observe that both ips and lips should be equal if bichords are equal or if one is a prefix of the other. To properly compute a prefix, bichords must be demarcated and aligned which we can ensure by also testing note-id(). Consulting note-id() also forces path divergence after partial sharing. Finally, we need to ensure that sharing is only permitted when at least one of $P_x = P_y$ and $L_x = L_y$ holds. We can check this by additionally examining assoc-class. This results in the following simple test:

$$\begin{aligned}\mathsf{sharable?}(n, n') : \mathsf{ip}{=}(n, n') \wedge \mathsf{lip}{=}(n, n') \wedge \\ \mathsf{assoc\text{-}class}{=}(n, n') \wedge \mathsf{note\text{-}id}{=}(n, n')\end{aligned}$$

[6] In implementation, assoc and note-id may be combined into one bit-field, since the former only needs 3 bits; we use 8 and pre-compute association classes.