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, parallelism.

Keywords Performance analysis, parallel programming models, call path profiling, HPCTOOLKIT.

1. Introduction
Over the last several years, power dissipation has become a substantial problem for microprocessor architectures as clock frequencies have increased [14]. 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 parallelism 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 [15] and Cilk [8] 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 [9,10] 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 is obvious if one considers that string manipulation routines might be called from many distinct contexts 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
when the sample occurred. A difficulty for call path profiling of a programming model such as Cilk, which uses a work-stealing run-time system to partition and map work onto a thread pool, is that the stack of native procedure frames active on 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 are stolen and migrate between threads. 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) to higher-level models such as Cilk and OpenMP. 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 Intel’s thread building blocks and Pthreads, 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, §2 describes parallel idleness and overhead. Then, §3 defines logical call path profiles while §4 shows how to obtain them using logical stack unwinding. §5 describes the application of these ideas to Cilk. Finally, §6 discusses related work and §7 concludes.

2. Pinpointing parallel bottlenecks

To enable an average developer to quickly determine whether a multithreaded application is effectively parallelized, we describe two novel measurement and analysis techniques that direct one’s attention to areas of ineffective parallelization. We then show how these techniques facilitate effective analysis.

2.1 Quantifying insufficient parallelism

To quantify insufficient parallelism, we describe how to efficiently and directly measure parallel idleness, i.e., when threads are idling or blocked 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.

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_o, 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_o = n - n_w.

Consider a run-time system that has one worker thread per core. On each sample, each thread receives an asynchronous signal. 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_o and attributes a fractional sample n_w/n_o to an idleness metric for the sample context.

This 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 \( \sum n_w \) over all samples t 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.

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.

\[1\] As mentioned in §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.
Table 1. Using parallel idleness and overhead to determine if the given application and input are effectively parallel on n cores.

<table>
<thead>
<tr>
<th>parallel idleness</th>
<th>parallel overhead</th>
<th>interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>low</td>
<td>effectively parallel, focus on serial performance</td>
</tr>
<tr>
<td>low</td>
<td>high</td>
<td>coarsen concurrency granularity</td>
</tr>
<tr>
<td>high</td>
<td>low</td>
<td>refine concurrency granularity</td>
</tr>
<tr>
<td>high</td>
<td>high</td>
<td>switch strategies; e.g., consider task parallelization</td>
</tr>
</tbody>
</table>

For library-based parallel 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 intrinsics 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 §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 associated with them to parallelization overhead. The tags therefore partition the application code — the ‘work’ — into useful work and overhead (distinct from idleness).

This scheme has two important benefits. First, compiler generated tags may be designed to partition sources of overhead into multiple types, thereby providing detailed information to users or analysis tools. For example, it may be useful to distinguish between synchronization overhead and all other overhead. The second benefit 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 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 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 loops4 procedures, and the calling contexts of a program, we can directly assess parallel efficiency and provide guidance for how to improve it (see Table[2]). 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; perhaps the idle processors could be kept busy with a completely different type of work (functional parallelism).

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 instantiation at run time. A key aspect of this is recovering logical user-level calling contexts. 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. Note that this technique applies to both parallel and serial applications. In §3 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. Analogously, we obtain logical call paths by logically unwinding the call stack. We introduce and define the following terminology.

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

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

A note represents a frame; a chord a grouping of frames; and a bichord the association of physical stack frames (\(P_i\)) with logical (\(L_i\)) stack frames. Logical frames correspond to a user-level view of an application; physical frames correspond to an implementation-level realization of that view. The p-notes \(P_i = (p_{i,1}, \ldots, p_{i,m_1})\) 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 unwind is a sequence of bichords

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

where bichord \(\langle P_n, L_n \rangle\) represents the innermost set of frames or the top of the context stack. The logical call path for this logical
unwind is simply its reverse:

\[
\langle \langle P_1, L_1 \rangle, \langle P_2, L_2 \rangle, \ldots, \langle P_n, L_n \rangle \rangle
\]

where \( \langle P_1, L_1 \rangle \) is the program’s entry point. 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

\[
\langle \langle p_1,1, \ldots, p_i, m_1 \rangle, \ldots, \langle p_n,1, \ldots, p_n, m_n \rangle \rangle
\]

where \( p_{i,1} \) 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.

There are several ways of concretely representing a logical call path; on-line measurement demands a compact one. We represent a \( p \)-chord 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 this nested structure (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 a bichord’s \( p \)-chord and \( l \)-chord. Given a 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 \( \{0, 1, M\} \times \{0, 1, M\} \), where \( M \) represents \( \{n \mid n \geq 2\} \). In particular, 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 \( M \leftrightarrow 0 \). 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. \( M \leftrightarrow 1 \). 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. \( 1 \leftrightarrow M \). 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 \( 0 \leftrightarrow \{0, 1, M\} \) 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, \( M \leftrightarrow M \), 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 \( \forall i \mid |P_i| \geq 1 \), 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) [1]—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 \( \langle P_n, L_n \rangle \) 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 results in considerable space inefficiency. To reduce memory effects, we wish to share notes without losing any information. As shown in the Appendix, given two bichords, \( B_x = \langle P_x, L_x \rangle \) and \( B_y = \langle P_y, L_y \rangle \), \( B_x \) and \( B_y \) can completely or partially share representations if one of the following is true:

- \( P_x = P_y \) and \( L_x = L_y \)
- \( \langle (P_x \sqcup P_y) \cup (P_y \sqcup P_x) \rangle \) and \( L_x = L_y \)
- \( P_x = P_y \) and \( \langle (L_x \sqcup L_y) \cup (L_y \sqcup L_x) \rangle \)

(where \( \sqcup \) means “strict prefix” and is defined with respect the sequence of notes that form a chord).

### 4. Obtaining logical call paths 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 Cilk 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 unwind 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 unwind. 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. 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 execution, the return address instruction pointer located in the stack. This is natural because although the physical call stack may not have ‘gaps’, the logical unwind process of logical unwinding should be controlled by the physical stack.

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. 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.

4.3 An API for logical unwinding

Algorithm 1: The backtrace routine for obtaining a logical unwind.

```plaintext
let c be the unwind cursor, initialized with machine context and language-specific agents
while step-bichord(&c) ≠ END_OF.PROJECTION do
  let a be the bichord’s association (from c)
  while step-pnote(a) ≠ END_OF_CHORD do
    Record p-note (instruction pointer from c)
  end
  while step-lnote(&c) ≠ END_OF_CHORD do
    Record l-note (logical instruction pointer from c)
  end
  Form bichord from a and the lists of p-notes and l-notes
end
```

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 unwind (Algorithm[1] for allocating storage, we determined that it was more desirable to encapsulate the code for the unwind than to complicate each agent’s implementation. Consequently, the logical unwind 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).

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 the backtrace routine is shown in Algorithm[2]. Each level adopts semantics similar to libunwind[15]. This means that to find each bichord in the logical unwind ((P_0, L_0), (P_{n-1}, L_{n-1}), . . . , (P_1, L_1)), 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 \(m+1^{st}\) 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 unwind.

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 \(P_j\) of length \(m_j\), the \(m_j+1^{th}\) call to step-pnote both enumerates \(P_j\)’s p-notes and discovers the next p-chord. For example, consider a section of the physical projection representing p-chords \(P_i\) and \(P_{i+1}\):

\[
\langle \ldots, p_{i,m_i}, (p_{i+1,1}, \ldots) \rangle
\]

While iterating over the p-notes in p-chord \(P_i\), we first issue \(m_i\) calls to step-pnote. On the \((m_i+1)^{th}\) 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 for Cilk-5 [8] (currently 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 con-
Fig 1. HPCTOOLKIT workflow.

currency 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 implementation are freely available.

Our implementation is part of HPCTOOLKIT [17], 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. haprofile analyzes the application binary to recover program structure such as procedures and loop nests. hpcprof (bottom right) interprets the call path profile and correlates it with program structure, generating a database for interactive presentation 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 the logical calling context 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 \((n_w, n_{-w})\), 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 \(\text{work} \) metric metric total associated with this logical call path and the fractional sample \(n_{-w}/n_w\) of idleness is added to the \(\text{idleness} \) 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 by the Cilk compiler is encapsulated either by a method call or a function macro inserted into the intermediate C code. 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. 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.

\(^3\) Most of overhead code not caught by these things is either a declaration, continuation control flow (Duff’s device) or trivial.

\(^4\) When macros are expanded by the C preprocessor no indication of the originating source file is typically recorded. In contrast, if the function calls are inlined, the C compiler will effectively generate the appropriate #line directives.
int fib(WorkerState* ws, int n) { struct frame* fr;
    #line 28 "lush:parallel-overhead"
    CILK2C_INIT_FRAME(fr, ...);
    CILK2C_START_THREAD_FAST();
    #line 28 "fib.cilk"
    if (n < 2) { int t = n;
        #line 31 "lush: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 "lush:parallel-overhead"
            CILK2C_BEFORE_SPAWN_FAST();
            CILK2C_PUSH_FRAME(fr);
            #line 34 "fib.cilk"
            x = fib(ws, n-1);
            #line 34 "lush:parallel-overhead"
            CILK2C_XPOP_FRAME_RESULT(fr, 0, x);
            CILK2C_AFTER_SPAWN_FAST();
            #line 34 "fib.cilk"
        }
        ...
    }

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

Cilk. In particular, we implemented the logical unwind API (described in §4.3), developed a Cilk-specific agent, and modified the hpcprof profile interpretation and source code correlation tool to normalize the results. The design of the Cilk agent illustrates several important points.

To understand the 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 instance, it is “converted” to the slow version. 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):

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 ↔ 0 (since they are part of the logical call path), while user-level C routines correspond to an association of 1 ↔ 1.

B. There may be j frames corresponding to Cilk fast frames. Since the fast clone of the Cilk routine directly corresponds to the logical frame, physical frames in this segment correspond to bichords with association 1 ↔ 1 where the l-chord directly corresponds to the p-chord.

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 the type of the worker. Consider the first worker before it has stolen and become a thief. (There may be only one such worker since this generalizes to serial execution of a Cilk program.) This worker’s B segment will correspond exactly to a logical unwind because the root stack frame will correspond to the “main” routine. This implies that in this case segment C should be interpreted as 1 ↔ 0 or M ↔ 0. However, once a worker becomes a thief (once a thief...), the B segment corresponds only to a portion of the logical unwind. This means that to complete the unwind, it is necessary to walk the cactus stack upon encountering stack segment C. This implies that the first frame of segment C should now be interpreted as 1 ↔ M. That is, once segment C is reached, it is necessary to complete the logical unwind by walking the cactus stack to the root frame. To distinguish between these two situations, it is convenient to make a small modification to the Cilk run time to indicate when the first worker thread becomes a thief.

5.4 Case study

To demonstrate the power of attributing work, parallel idleness and parallel overhead to logical call path profiles, we consider a case study of a Cilk program for Cholesky decomposition. We used the Cholesky program that comes as an example with the Cilk 5.4.6 source distribution. We ran a problem size of 4000 × 4000 (40000 non-zeros) on one quad-core AMD Opteron 2360 SE (2.5 Ghz).

Figure 4 presents one view of the results displayed by hpcviewer. The view is divided into three main components. The navigation
and the source pane (top sub-pane) shows the procedure instances along the call paths. The selected line in the navigation pane context tree, partially expanded. One can see several procedure instances in addition to all of its callees. Although thread-specific metadata values are available, we first desire to know whether it is necessary that they represent values for the associated procedure instance in the metric pane to the right. Sibling entries are sorted with respect to the selected metric column (in this case “work (all)”).

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.

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. These metrics show summary values over the four worker threads for work, parallel idleness and parallel overhead, respectively. The metrics are inclusive in the sense that they represent values for the associated procedure instance in addition to all of its callees. Although thread-specific metric values are available, we first desire to know whether it is necessary to consult the more fine-grained information. This view shows that about 50% of of the total work of the program is spent in the top level call to cholesky; the top level call to num_blocks is a close second at 47.5%. We can also quickly see that about 13% and 14% of the total parallel idleness and overhead, respectively, occur in cholesky. However, because the idleness and overhead are very small with respect to the work (1% and 0.1%, respectively), it is clear that the parallelization is very effective.

Despite this fact, the idleness and overhead metrics allow us to pinpoint where the inefficiency is. To do this, 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, a summary idleness metric. Note that in contrast to Figure 4 these metric values are “exclusive” in the sense that they do not include values for a procedure’s callees.

Again we see with this view that total program idleness and overhead are very small — approximately 3% (4.46/146) and 0.3% (4.90/146), respectively. This is valuable information because it clearly confirms that we should not spend time tuning this program (assuming that the input values are representative). Nevertheless, spending a small amount of additional time scanning the routines with the top idleness values is instructive. The top two are versions of the C library routine free. When the calling contexts for these routines are expanded (not shown in the figure because of space), it is evident that they are both called by free_matrix, a non-Cilk helper routine for the Cholesky program. In other words, the Cholesky program uses a serial helper routine to deallocate the matrix, which consumes approximately 38% of the program’s idleness. Although insignificant from the perspective of total performance, hpcviewer immediately identifies it as the the primary source of idleness. Walking down the list reveals other serial helper routines that are part of the Cholesky program: num_nonzeros, mag and num_blocks are visible in this screen shot. The significance of this is that without even reading the source code — we have quickly identified all the primary sources of idleness in a program. Although it is not surprising that serial code is responsible for idleness, it is surprising that we can immediately identify and quantify its effect.

6. Related work

Our parallel idleness metric is similar to Quartz’s notion of “normalized time” gathered for procedures, synchronization objects and threads. Normalized time is computed by attributing 1/\(w\) to the relevant section of code on each sample. Intuitively, this metric inflates compute times in areas of poor parallelization to highlight portions of the code with poor concurrency. 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: \(n_{w}/w\). Unlike Quartz’s normalized time, our measure of parallel idleness may be quantitatively compared to total compute time and total idle 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 [7]. To predict parallel performance, Crovella and LeBlanc describe a “lost cycles analysis” [8] that separates parallel overhead from pure computation. They further subdivide 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 focus on physical call path profile projections [9][10] or logical call path profile projections, such as for Java [11][12]. 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 [13]. Our work is more general in the sense that we define a logical call path profile, explain how it can be efficiently represented, and describe a general API for obtaining one. Our parallel idleness and overhead metrics can be directly applied to OpenMP.

It is interesting to compare our performance analysis of Cilk to Cilk’s own performance metrics: critical path and total work (given a particular input). The significant advantages of Cilk’s metrics

Figure 5. A Callers (bottom-up) view of Cholesky.
are that they are “platform independent” and directly correspond to the theoretical model that underlies Cilk’s provably efficient scheduler. 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 highlights parallel inefficiency. Moreover, paired with hardware performance counter information, it distinguishes between different types of architectural bottlenecks in different regions of code.

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 user-level 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 identify serialization in an execution with only about 1% parallel idleness. 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.

Our work shows that it is possible to construct effective and efficient performance tools for multithreaded programs whose run-time cost 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 by the current project team Laksono Adhianto, Michael Fagan, Mark Krentel, and Gabriel Marin. Project alumni include Nathan Froyd and Robert Fowler.

References


Appendix: Efficiently representing logical CCTs

Definitions

We observe that some associations are naturally related. For example, \( 1 \leftrightarrow 0 \) is the natural ‘base case’ of \( M \leftrightarrow 0 \). Similarly, \( 1 \leftrightarrow 1 \) is the natural ‘base case’ of both \( 1 \leftrightarrow M \) and \( M \leftrightarrow 1 \). We therefore define the following association-classes:

- \( \mathcal{A} \leftrightarrow 0 = \{ 1 \leftrightarrow 0, M \leftrightarrow 0 \} \)
- \( \mathcal{A} \leftrightarrow 1 = \{ 1 \leftrightarrow 1, M \leftrightarrow 1 \} \)
• \(1 \leftrightarrow A = \{1 \leftrightarrow 1, 1 \leftrightarrow M\}\)

We define the functions \(ip\) and \(lip\) to extract the physical and logical instruction pointers from a \(p\)-note and \(l\)-note, respectively. Let the functions \(assoc\) and \(assoc\text{-class}\) return the association and association-class of a bichord, respectively.

**Sharing the representation of two logical call paths**

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):

\[
(⟨p_{i,b}, p_{i,a}⟩, (l_i, 1)) \ldots (1)
\]

\[
(⟨p'_{i,c}, p_{i,b}, p_{i,a}⟩, (l_i, 1)) \ldots (2)
\]

\[
\ldots (⟨p_{i,c}, p_{i,b}, p_{i,a}⟩, (l_i, 1)) \ldots (3)
\]

\[
\ldots (⟨p_{i,a}⟩, (l_i, 1)) \ldots (4)
\]

\[
(⟨p_{i,a}⟩, (l_i, 1)) \ldots (5)
\]

\[
\ldots (⟨p_{i,a}, p_{i,b}, p_{i,a}⟩, (l_i, 1)) \ldots (6)
\]

\[
\ldots (⟨p_{i,c}, (l_i, 1)), (⟨p_{i,b}, (l_i, 1)), (⟨p_{i,a}⟩, (l_i, 1)) \ldots (7)
\]

Unwinds (1)–(4), with bichords of association \(M \leftrightarrow 1\), could represent an interpreter implementing a high-level logical operation, signified by \(l\)-note \(l_i, 1\). Since none of the bichords are equal, treating these bichords as atomic units requires that common notes such as \(p_{i,a}\) be duplicated when the corresponding call paths are inserted into the logical calling context tree. In particular, note that unwinds (2) and (3) are not equal because the former contains a sample and must therefore be a leaf node. Because unwinds such as these occur frequently in practice, it is desirable to develop a more space-efficient representation. Consequently, it is necessary to know when it is both possible and profitable to share the notes of two bichords \(B_x = (P_x, L_x)\) and \(B_y = (P_y, L_y)\).

As a starting point, clearly \(B_x\) and \(B_y\) can be represented with the same structure if \(P_x = P_y\) and \(L_x = L_y\) and both are interior bichords. Now we must consider situations where neither the former nor latter condition hold. Suppose we only have that \(P_x \neq P_y\) or \(L_x \neq L_y\). Without loss of generality assume the latter. We divide the analysis into three cases, corresponding to unwinds (1)–(4).

• \(assoc = (B_x, B_y)\): Consider Unwinds (1) and (2) \((B_x\) and \(B_y\), respectively). Both have association \(M \leftrightarrow 1\). Clearly, the representation of each bichord can share notes \(p_{i,a}\) and \(l_i, 1\); what about \(p'_{i,c}\) and \(p_{i,b}\)? Observe that \(p'_{i,c}\) represents a sample point while \(p_{i,b}\) represents a call site. Because of this, in general \(ip(p'_{i,c}) \neq ip(p_{i,b})\) and it is not possible to share representations. However, since a sample can be taken at a call site, it is possible that \(ip(p'_{i,b}) = ip(p_{i,b})\) and therefore possible to share the note’s representation as long as a flag indicates that the node is both a sample point as well as an interior node. (Technically, call sites are represented with return addresses.) The presence of a non-zero metric count naturally satisfies this requirement.

This conclusion allows us to remove the interior node qualification previously attached to the case where \(B_x = B_y\), implying that it is possible to share \(p'_{i,c}\) and \(p_{i,b}\) in Unwinds (2) and (3) if \(ip = (p'_{i,c}, p_{i,b})\) holds.

• \(assoc \neq (B_x, B_y)\) but \(assoc\text{-class} = (B_x, B_y)\): Compare Unwinds (1) and (4) \((B_x\) and \(B_y\), respectively). In this case we have different associations but identical association classes. It is possible to share \(p_{i,a}\) without losing information (as shown below in recovery).

Observe that this implies that \(p_{i,a}\) may be shared not only between Unwinds (3) and (4) but also with Unwind (6), which results in sharing \(p_{i,a}\) between fully distinct contexts that happen to share a common ‘prefix’, i.e., \(P_y \subseteq P_x\). Indeed, this is the same criterion as in the previous case.

Note that as the above discussion about sharing leaf and interior nodes implies, it may be possible to share the representation of \(p_{i,a}\) between Unwinds (1) and (5) if \(ip = (p_{i,a}, p'_{i,a})\) holds.

*\(assoc \neq (B_x, B_y)\) and \(assoc\text{-class} \neq (B_x, B_y)\): This case is impossible given that \(L_x \neq L_y\).

As a final observation, it might seem that sharing is possible between Unwind (7) with Unwind (3). In this case lossless sharing between the two is not possible unless bichord boundary information is somehow duplicated. We wish to avoid this complication.

The above analysis applies when at least one of \(P_x = P_y\) or \(L_x \neq L_y\). The final case is when \(P_x \neq P_y\) and \(L_x \neq L_y\).

\[
(⟨p_{i,a}⟩, (l_i, 2, l_i, 1)) \ldots (8)
\]

\[
(⟨p_{i,a}⟩, (l_i, 1)) \ldots (9)
\]

\[
(⟨p_{i,a}⟩, (l_i, 1)) \ldots (10)
\]

Compare Unwinds (1) and (5) \((B_x\) and \(B_y\) respectively). Here, \(L_x \neq L_y\). In this case it might appear that sharing is possible even though neither association nor association class are equal. However, doing this would require duplicating association information. Similarly, comparing Unwind (1) against both (9) and (10) show examples where sharing is nominally possible between \(p\)-notes but not \(l\)-notes. However, any nominal sharing would again be offset by a requirement to duplicate information to maintain distinctions. Consequently, lossless sharing is not possible.

**Representing bichords**

The above conclusions directly translate into the following implementation which maintains the two-level distinction between bichords and notes implicitly. A bichord is represented by a list of \(X\)-structures and a logical calling context tree by a tree of \(X\)-structures. Each \(X\) contains an association \(assoc\) and a physical and logical instruction pointer \(ip\) and \(lip\), respectively. Given a bichord \((P_x, L_x)\), we need \(n\) \(X\)s \(X_1, \ldots, X_n\), where \(n = max(|P_x|, |L_x|)\) and \(X_1\) representing the outermost portion of the bichord. Let the function note-id return the index of an \(X\)’s note-id\((X_j) = j\). Note that \(ip(X_i) = NIL\) if \(|P_x| < j \leq n\); similarly for \(lip(X_i)\).

During the process of inserting a logical unwind into the logical calling context tree, we must be able to efficiently determine when nodes can be shared. This turns into the following problem: Given the call path fragment \(m' \sim n'\) \((m'\ calls\ n')\) and given an \(X\)-structure \(m\) in the logical CCT such that \(m' = m\), is it the case that \(\exists n\ s.t. n\ is\ a\ child\ of\ m\ and\ sharable?(n, n')\) holds? The prior section implies that:

\[
\text{sharable?}(n, n') : ip = (n, n') \land lip = (n, n') \land
\]

\[
\text{assoc-class} = (n, n') \land \text{note-id} = (n, n')
\]

These examples have \(|L_x| = |L_y| = 1\), which by symmetry covers associations except those in class \(A \leftrightarrow 0\); it is straightforward to extend this analysis to cover those in \(A \leftrightarrow 0\).