

Identifying Potential Parallelism via Loop-centric Profiling

Tipp Moseley, Daniel A. Connors,
Dirk Grunwald
Department of Computer Science
University of Colorado at Boulder
{moseleyt,dconnors,grunwald}@colorado.edu

Ramesh Peri
Software and Solutions Group
Intel Corporation
ramesh.v.peri@intel.com

ABSTRACT

The transition to multithreaded, multi-core designs places a greater responsibility on programmers and software for improving performance; thread-level parallelism (TLP) will be increasingly relied upon in addition to instruction-level parallelism (ILP) and increased clock frequency. Deciding where to try to parallelize code is difficult, especially for large, complex applications or those where the original developers have moved on. Outer loops are relatively easy targets for parallelization, but traditional profilers focus primarily on functions and hot inner loops. To aid in programmers' parallelization efforts, we introduce the concept of loop-centric profiling to provide a hierarchical view of how much time is spent in a loop and the loops nested within it.

This paper introduces two techniques for loop profiling. First, we describe an instrumentation-based approach that gathers highly detailed and accurate information about loop behavior. Second, we present a sampling approach that achieves similar results with negligible overhead. The paper concludes with a case study evaluating the tool on several SPEC 2000 benchmarks.

Categories and Subject Descriptors

D.1.3 [Software]: Programming Techniques—*Concurrent Programming*

General Terms

Performance

Keywords

Loop profiling, parallelization

1. INTRODUCTION

With processor designs becoming increasingly multithreaded and multi-core, sequential programs will no longer achieve historical performance gains from advances in technology. This trend places a greater responsibility on programmers and software for program optimization. Vectorization and thread-level parallelism (TLP) will

be increasingly relied upon in place of instruction-level parallelism (ILP) and increased clock frequency for improving performance. Many techniques exist that attempt to automatically parallelize loops, but they are most effective on inner loops. In contrast, coarse-grain parallelism is difficult to detect. High level loops may be nested across function and file boundaries, and their significance cannot be detected with current tools. Currently, manual techniques remain the predominant method of detecting and introducing high-level TLP into a program.

Performance analysis of sequential programs often relies on profilers that provide information about how much time is spent in different regions of code [4] [8], and regions that are the most active, or hot, are then given special attention by the programmer. Similarly, tools like gprof [10] analyze programs at the function level. Intel® VTune™, for example, creates a call graph profile that shows the structure of the application at the function level, including parent-child relationships and time spent in each function (self time) and time spent in each function including children (total time). Performance analysis tools designed with the intent to analyze sequential code are very mature, but their focus is not well suited for the task of extracting thread-level parallelism from existing applications.

When trying to parallelize sequential code, a logical first step might be to find which loops are doing the most work. However, even deciding which loops to target can be very time consuming. Current tools do not adequately capture the context, from a loop perspective, of where time is spent. The function call graph profile leaves guess work to the programmer to identify loops that are good candidates for parallelization. In fact, some functions can be extremely long; some individual functions in the GNU C compiler are over 1000 lines of code. Clearly, it is inadequate to focus our attention just on hot outer functions.

We propose the concept of *loop-centric* profiling to give the programmer a more complete view of where time is spent in a program. Loop-centric profiling is similar in nature to the traditional call graph, but also identifies parent-child relationships and self/total execution counts for loops in addition to functions.

Many profiling tools require compiler support, thus the developer must recompile the entire application to gather profile data. Since large applications may take hours to recompile, it can become impractical to profile this way. Therefore, the algorithms presented in this paper operate only on binary programs and are compiler-independent (except for debug information). Requiring only a binary does not imply that the tool is useful without source code. Instead, it provides the flexibility for the tool to be language independent, and thus it can be used with any compiler toolchain.

First, we present an instrumentation-based tool, named Loop-Prof, that uses a dynamic, stack-based algorithm to detect and ac-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CF'07, May 7–9, 2007, Ischia, Italy.

Copyright 2007 ACM 978-1-59593-683-7/07/0005 ...\$5.00.

count for loops. LoopProf collects an abundance of information, including:

- The number of entries to each loop.
- The total number of loop iterations.
- Histogram information for the number of iterations executed per loop entry.
- Self and total instruction counts for loops and functions.
- Per-loop self and total counts for each type of instruction (e.g., load, store, fp).
- A hierarchical *loop-call graph* containing both loops and functions.

In addition, we introduce a lightweight, sampling-based tool named LoopSampler. In contrast to LoopProf, LoopSampler has negligible overhead, but it does not collect detailed information such as loop entries and iterations.

The rest of this paper is organized as follows: Section 2 describes related profiling work. Section 3 discusses instrumentation-based loop profiling. Section 4 presents a sampling mechanism for loop profiling. Section 5 discusses performance overhead and a case study. Section 6 describes future work. Finally, Section 7 concludes.

2. RELATED WORK

2.1 Traditional Profiling

Since applications often spend a large portion of execution time in a small portion of code, typically inner loops, performance tuning often focuses on optimizing code found within these loops. Tools such as DCPI[4], OProfile [2], or Intel® VTune™ naturally identify such inner loops since they are the most frequently executed regions of code, but they do not focus on identifying loops themselves. Therefore, such tools fail to communicate information about the overall structure of loops in a program.

The function call graph profile and associated features common to Intel® VTune™, gprof[10], and other standard tools are the basis for another type of program optimization. Each function profiled includes the execution time spent in that function (self time) and the execution time spent in that function and all child functions (total time). These tools also collect a call graph that shows each function’s callers and callees, and the number of times each is called. The best way to look at LoopProf is that it is a “loop call graph profiler,” providing functionality for loops that is analogous to what prior tools provide for functions. One notable difference is that gprof *et. al.* use time as the profiling metric. LoopSampler also uses time, but since dynamic instrumentation incurs a high and variable overhead, LoopProf uses executed instructions instead.

OProfile [2] supports call graph profiling using a sampling mechanism similar to the one presented in this paper. However, OProfile does not recognize loops and relies on kernel support.

2.2 Loop Profiling

Most previous work related to loop profiling is targeted at improving hardware branch prediction and ILP. Several studies aim to characterize properties of loops to enable traditional optimization and improve dynamic branch prediction. Kobayashi analyzes program traces to detect loops and characterize their different properties [11]. De Alba et al. also perform several loop studies[3] aiming to improve path prediction inside of loops[5, 7] and enable dynamic

loop unrolling[6]. Loop-back, or “loop termination” edges are frequently mispredicted. Sherwood et al.[14] propose both hardware and software techniques for “loop termination prediction” to increase branch prediction accuracy with loop branches.

The instrumentation-based algorithm for loop detection proposed in this paper is based on an algorithm used by Shye et al.[15] for software path profiling. The study used performance counters on modern hardware to collect a trace of branch execution, then used dominator and post-dominator information to construct traces of execution and a statistical model for which paths were hot. To validate their results, a full path profile was gathered using a Pin tool. The algorithm used to capture the full path profile is similar to the one proposed in this paper, except loop back edges are used to terminate a path instead of define a loop.

In a more germane study[17], Tubella et al. propose dynamic loop detection and thread speculation in hardware. The results show that there is significant thread-level parallelism that can be achieved when loops are parallelized. Specifically, for machine configurations with 2, 4, 8, and 16 contexts, the proposed mechanism achieves an average of 1.65, 2.6, 4, and 6.2 correctly speculated threads per cycle for SPEC95 benchmarks. However, the study does not describe how data dependence and speculation issues are solved.

Since the above studies are all focused on hardware loop detection, for performance reasons, all of them assume that a loop is defined by the region between a branch with a negative offset and its target. This assumption is often true, but for correctness the algorithms in this paper strive to define loops in a more precise manner.

The C and Fortran compilers developed by Sun Microsystems® are capable of automatically parallelizing loops when static analysis determines that it is safe and profitable to do so. Until version 5.0, the Sun WorkShop™ program performance analysis suite shipped with a tool called LoopTool that performed some similar analyses as our tools. Instead of identifying loops to parallelize, LoopTool was intended to analyze how well the parallelization implemented by the compiler performed. The key feature of Sun’s LoopTool is a table of loop timings, or how much time was spent in each loop, including those parallelized by the compiler. This is analogous to the total instructions feature of LoopProf. LoopTool is useful to determine total time spent in each loop, but it does not provide a window into how loops are structured and nested, and it is dependent on compile-time information and compiler-based instrumentation.

An initial revision of the LoopProf algorithm was presented in [13]. This paper introduces a low-overhead sampling-based profiling technique and extends the instrumentation-based algorithm to provide better support for recursive functions.

3. INSTRUMENTATION-BASED LOOP PROFILING

LoopProf is implemented using the Pin [12] run-time binary instrumentation system. This has the immediate benefit of working transparently with unmodified Linux binaries on Intel® ARM, IA32, EM64T (64-bit x86), and Itanium® architectures.

3.1 Background

Since LoopProf relies on dynamic instrumentation and is compiler-independent, loop detection must be done in the absence of a complete control flow graph (CFG). Static analysis can provide most information about a CFG, but indirect branch targets aren’t known. In most programs, it is unlikely that indirect branches would signif-

icantly effect the profiling results, but for correctness the algorithm presented in this section does not assume a known CFG. Instead, LoopProf works without any compiler support; it dynamically discovers loops in a program based on the basic block (BBL) path of execution. Interestingly, we also investigated an instrumentation-based tool that used the partial CFG, and it achieved very similar results in terms of both data and performance.

Another effect of not having a CFG is a unique characterization of BBLs. Pin, like other dynamic instrumentation systems, does not characterize BBLs in the same sense as the compiler. BBLs reported to Pintools are a multiple-entry, single-exit series of instructions, also known as dynamic BBLs (DBBL). This implies that multiple DBBLs include the same instructions, and that Pin translates each DBBL separately in its code cache. This difference is significant for identifying loops, so LoopProf must dynamically split DBBLs into traditional, disjoint BBLs when it discovers two DBBLs have the same tail instruction address but do not share the same head instruction address.

3.2 Overview

LoopProf discovers loops by tracing each BBL as it is executed by the program. A stack of BBLs is kept to represent the program's path of execution. Each time a new BBL is encountered, it is pushed onto a stack. This stack is referred to as the *path* of execution. If a BBL is encountered that is already on the path, it is marked as a loop head. It is important to note that this description will detect loops that occur across function boundaries, such as recursive function calls.

Because of recursion, it does not make sense to search the entire path of execution for a BBL. For instance, returning from a series of recursive function calls may appear to LoopProf as a loop over the BBL containing the return instruction. The tool keeps track of the function call stack as well and the loop detection is limited to the basic blocks that have executed in the current frame. This allows the same BBL to be on the path multiple times without necessarily defining a loop, and also prevents recursive procedures to be discovered as loops.

Since LoopProf performs loop detection and instruction profiling in a novel way, we define some of the following common terminology in a slightly different way than traditional control flow and loop analysis:

- *loop back* - a jump (or fall through) to a BBL that is already on the path.
- *loop head* - the first BBL in a loop and the target of a *loop back*.
- *loop entries* - the number of times a loop has been entered from above.
- *loop iterations* - the number of times a loop iterates before it is exited.
- *self instructions* - the number of dynamic instructions executed within a loop; exclusive instructions.
- *total instructions* - the sum of dynamic instructions executed within a loop and its children; inclusive instructions.

3.3 Loop Detection

First, we will describe the data structures used in loop detection:

- `LoopInfo` - This structure contains accounting information such as self/total instruction counts, iteration counts, and child edges. Each loop that is discovered is allocated a `LoopInfo` object.

```

1 // Data Structures
2 class LoopInfo {
3     // Loop accounting variables for iterations,
4     // self/total instructions, and children
5 };
6
7 class StackFrame {
8     list<BblPathInfo *> path;
9 };
10
11 class BblPathInfo {
12     addr_t head;
13     // Other accounting information
14 };
15
16 // Variables
17 list<StackFrame> callStack;
18 HashTable< addr_t, LoopInfo *> loops;
19
20 // Loop Detection Algorithm
21 void processLoop(StackFrame frame,
22                 BblPathInfo *bpi) {
23     if( !loops.hasKey(bpi->head) ) {
24         loops[bpi->head] = new LoopInfo();
25     }
26
27     doInstructionAccounting(frame, bpi)
28     attachGraphEdges(frame, bpi);
29
30     popBbls(frame, bpi);
31 }
32
33 // Called each time Bbl is executed
34 void processBbl(addr_t bblhead,
35               addr_t sp) {
36     // Adjust the callstack
37     adjustStack(sp);
38
39     StackFrame &frame = callStack.back();
40
41     if( frame.path.contains(bblhead) ) {
42         // Loop detected; do instruction accounting
43         // and pop BBLs above this BBL.
44         BblPathInfo *bpi = findBPI(frame, bblhead);
45         processLoop(bpi);
46     } else {
47         newBpi = new BblPathInfo(bblhead, sp);
48         frame.path.push(newBpi);
49     }
50 }

```

Figure 1: C++ pseudocode for loop detection.

- `StackFrame` - Each activation on the call stack is represented by a `StackFrame`. Within each `StackFrame`, there is a list of `BblPathInfo` objects that represent the current path of execution.
- `BblPathInfo` - This structure represents a BBL on the path and contains temporary accounting information.

Figure 1 shows high level C++ pseudocode for the analysis that occurs every time a basic block is executed. The full code is omitted in favor of the following descriptions. Notice that every stack frame has its own path. The following is a description of the actions that occur in the `processBbl()` function:

1. Call the `adjustStack()` function. This function serves two purposes. First, it simply pops activation stack frames

so that the top of the stack is in line with the current stack pointer. Second, the top level loops in the frame to be popped must be remembered because they must be connected to the loop graph and accounted for. That technique is described below.

2. If the path contains the current BBL (`bblhead`), call the `processLoop()` function to do accounting and pop nodes that occur after this BBL in the path. Otherwise, push a `BblPathInfo` for `bblhead` on the path. Again, because of recursion, this accounting cannot be kept with a global BBL object because one BBL may be in multiple paths.

When a loop back is detected, the `processLoop()` function is called to do iteration and instruction accounting and to attach edges on the loop graph. Several things happen in the `processLoop()` function:

1. If this is the first time this loop has been encountered, a new `LoopInfo` object is created and added to the `loops` hash table.
2. Increment the `iterations` counter for this `BblPathInfo` object.
3. Attach edges on the loop graph.
4. Perform instruction accounting for the elements after this BBL in the path.
5. Pop BBLs above this BBL (but not the loop head) in the path. The loop head `BblPathInfo` is left on the path to keep track of temporary loop accounting information. The `LoopInfo` object is only updated when a loop head is popped off the path.

3.4 Instruction Accounting

The detailed algorithms associated with accounting are omitted from this paper for brevity, but a full description is provided.

Accounting is performed each iteration of a loop, before the BBLs are popped off. BBLs are kept on the path in a `BblPathInfo` object. This object contains temporary accounting information that is valid while a BBL is on the path, and this information is stored permanently before a BBL is removed from the path. The path is traversed in reverse order until the loop head BBL (H) is reached, and there are three cases that must be dealt with:

- A BBL that is not associated with any loop. This BBL is associated with and accounted for in H .
- A BBL that was previously associated with a loop head (H') in the path other than H . This BBL is reassociated and accounted for in H . This condition can occur when an inner loop iterates zero or one times and is not initially detected by the algorithm.
- A BBL, B , that is a loop head (but is not H). The global loop statistics for this loop head are updated. Then any associations with this B are removed. The accounting information for B is attributed to the total instruction count of H , and an edge $H - B$ is added to the loop graph.

When a function returns, the call stack is traversed and the self instructions from the popped function are added to the total counts of each loop and function above it in the path. To prevent double counting due to recursion, the algorithm will only add once to functions or loops that appear on the stack multiple times

3.5 Loop Call Graph Generation

Loop call graph generation is performed in much the same way as instruction accounting. When the path is being traversed, if a BBL, B , is a loop head and it is associated with another loop head, H , then B is a child of H .

When the `adjustStack()` function is called, if a frame is popped off the stack, the popped frame is attached to the last BBL on the path after the frame is popped. This BBL, B , is the BBL that called the function. Later, when B is popped off the path, its children are attached to the loop that B is associated with.

3.6 Optimization

In an effort to reduce the overhead of loop profiling, we have optimized LoopProf in several ways. Instead of performing the analysis each time a BBL is executed, BBLs are buffered and processed in batch to increase code and data locality. Since each BBL would require a path traversal to see if it does exist on the path, each activation record has a table of the BBLs it contains. When a BBL is placed on the path, it is marked as such in the table. Then, the accounting traversal only occurs when a BBL is encountered that has already been marked (a loop-back edge).

3.7 Multithreaded Programs

Since many real-world programs are already multithreaded to some degree, a practical profiling tool should support profiling multithreaded programs as well. To support this, LoopProf maintains a separate context for each thread at runtime, and at the end of execution, per-thread and aggregate statistics are reported.

3.8 Limitations

Designing a loop profiling tool to work without compiler guidance is not without its limitations. For most applications, the overhead is reasonable, but for some the overhead is over 70 times slower than native execution. Secondly, due to the dynamic nature of loop detection, loops with trip counts of zero will not be detected until the loop head itself has been detected as a loop. Finally, some trip counts may suffer from off-by-one errors as a result of early loop exits from *break* or *return* instructions.

4. SAMPLING-BASED LOOP PROFILING

Like LoopProf, LoopSampler uses Pin to shepherd program behavior. However, instead of using dynamic instrumentation, LoopSampler leverages Pin's "probe" feature to run the application natively and only take control when specific events occur. LoopSampler works by periodically interrupting an application and examining the contents of the call stack to determine what functions and loops are currently active.

First, LoopSampler needs a periodic interrupt to examine the state of the call stack. One approach would be to implement this in the kernel, but kernel-based solutions are less portable and often difficult to install. Another alternative would be to use the UNIX `setitimer` function, but `setitimer` only supports the `SIGALRM`, `SIGVTALRM`, and `SIGPROF` signals. Since applications may already use these signals, overriding them could potentially break the program. Instead, we implement an approach that is similar to `setitimer` but allows an arbitrary signal to be specified. When an application begins, LoopSampler first registers a signal handler for the specified signal. Then, the tool starts another process, called the "signaler", to send the signal back to the original process at a specific frequency. One drawback of this approach is that in Linux, the signaler cannot send signals faster than the `USER_HZ` kernel variable, which is 100 for most machines.

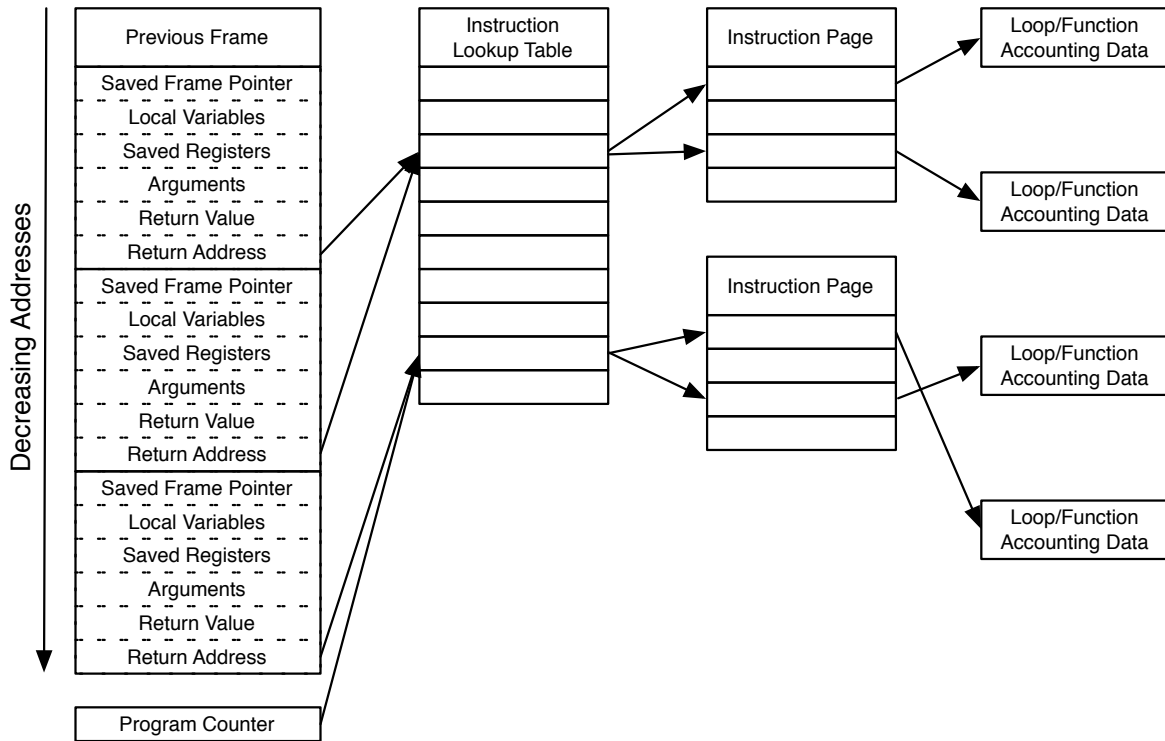


Figure 2: The process of looking up functions and loops by unwinding the activation stack.

Increasing `USER_HZ` would require recompiling the kernel. Fortunately, experiments show that 100Hz is sufficient to collect very accurate profiles. Note that this approach is primarily aimed toward ease of use. Higher fidelity sampling could easily be achieved via kernel modification.

At program initialization, LoopSampler registers a callback with Pin to be notified when new binary images are loaded. Each time a new image is loaded, it is parsed and its partial CFG is used to detect loops using traditional dominator analysis. Some heuristics are used to predict indirect branch targets, but the fundamental assumption here is that indirect branches will not frequently cross loop boundaries. Note that this is different from the LoopProf algorithm, which uses a BBL stack to identify loops without using the CFG.

Once all of the loops have been detected, a lookup table is created to map every program address to an accounting data structure for the loop and function in which it is contained. To enable fast lookup, we use a page table with large pages so neighboring functions will likely be on the same page. Since instructions may be only be byte aligned, the table entries must be byte aligned as well. Therefore, in a 32-bit address space, the table itself requires four times the space of the original program. For each function and loop in the program, there is an accounting data structure that contains self/total counts and a list of edges to nested functions and loops.

Call graph generation and accounting happens within the signal handler. Instead of self/total instruction counts, we use self/total samples here. When an interrupt occurs, the operating system provides the interrupted context as an argument to the signal handler. First the self sample counts in the loop and function structures for the current program counter are updated. Then the call stack is traversed using the `libunwind` [1] library. In each frame, the loop and function accounting structures for the return address are located

and their total sample count is updated. Figure 2 shows the process of unwinding the stack and looking up accounting structures. To prevent double counting due to recursion, a list of visited nodes is kept as the stack is unwound, and each loop or function only gets updated once. As frames are traversed, edges are added to each loop and function to build the dynamic loop-call graph.

4.1 Multithreaded Programs

For multithreaded programs, each thread must receive a signal so that its context may be examined. Therefore, the signaler process sends a signal to each thread in the application. Like LoopProf, per-thread and aggregate statistics are reported at the end of execution.

5. RESULTS

As output, both tools report self/total counts and a list of edges in the loop-call graph. In addition, LoopProf provides counts for entries to loops and a histogram of iterations per entry. There is usually too much raw information to deal with visually, so there is a post-processing tool to manipulate the data. In a large program, thousands or hundreds of thousands of loops may be detected, so it is useful to be able to focus on the ones that contribute the most *total* instructions by filtering out loops that contribute less than a given percent to total execution. We have found just that pruning loops that contribute less than 0.1% from the loop call graph has a significant effect on readability. For some of the SPEC2000 [16] benchmarks tested, many thousands of loops were detected, but this pruning threshold was sufficient to reduce the number of nodes to tens or hundreds. It is also useful to be able to sort the loops and functions on any number of fields, such as self or total instructions, iterations, memory reads and writes, floating point operations, etc. The graphs in this section are generated using the Graphviz tool [9].

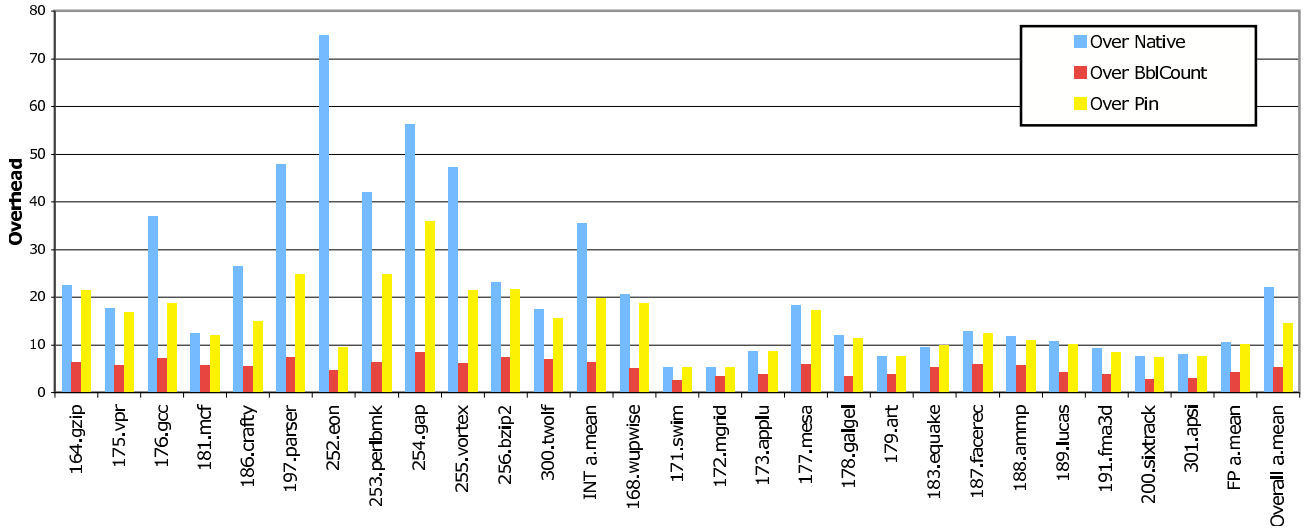


Figure 3: The runtime overhead of the LoopProf tool when compared to applications run natively, executed with no instrumentation under Pin, and instrumented by the BblCount Pintool.

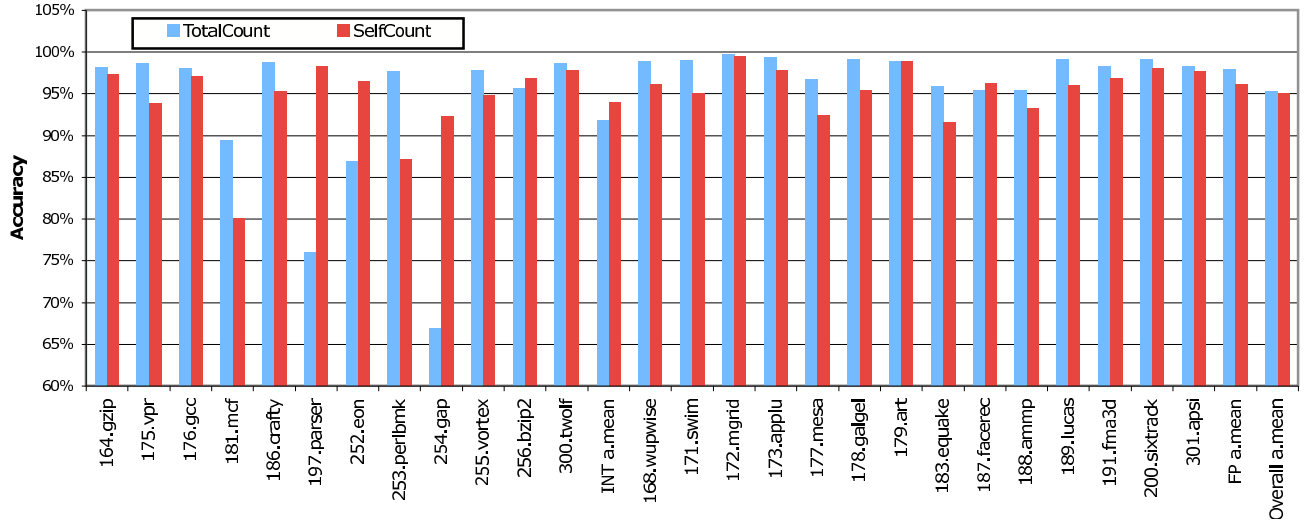


Figure 4: Accuracy of sampling-based profiling compared to instrumentation-based profiling.

5.1 LoopProf Performance

Complex binary instrumentation tools commonly incur high overhead. Figure 3 shows the overhead for running the tool on various SPEC2000 benchmarks. For each benchmark tested, vertical bars show the overhead of LoopProf when compared to three items: native execution, Pin executing the program with no instrumentation, and a simple Pintool that counts basic block frequency. Basic block counting requires each BBL to be instrumented by Pin, but the instrumentation code is a simple increment. Therefore, BblCount is a more useful point of comparison than native execution because it represents an optimal lower bound. LoopProf is on average 5 times slower than BblCount.

In comparison to native execution, the overhead of loop profiling is rather bimodal; INT benchmarks have exceptionally high overhead (35), and FP benchmarks are quite reasonable (10). This is likely due to the more regular code structure and greater number of instructions per BBL found in the FP benchmarks. The results show that on average, LoopProf is 22 times slower than native ex-

ecution. Because most programs shouldn't generally require frequent or iterative profiling, we believe this number to be reasonable for cases where the developer requires more detailed information about loops. In addition, a known accurate baseline is necessary to validate the results of the sampling technique.

5.2 LoopSampler Performance

During initialization, LoopSampler can take 2-3 seconds to process the program and all of the libraries as they are loaded. After that, there is no measurable overhead incurred by the sampler at a frequency of 100Hz.

In addition to having no operational overhead, LoopSampler is also very accurate compared to an instrumented profile. Accuracy is measured by the following definition:

$$Accuracy = \sum_{L_{inst} \in loops_{inst}} (1 - abs(L_{inst_{pct}} - L_{smp_{pct}})) * L_{inst_{pct}} \tag{1}$$

loop: 0x8061343	apsi.f:5296		
Iterations	SelfIns	TotalIns	
52M	1G	146G	
TripCount	Entries		
0	17436831		
3	17436832		

loop: 0x804ffa3	apsi.f:906		
Iterations	SelfIns	TotalIns	
7K	1M	157G	
TripCount	Entries		
> 10	70		

loop: 0x8060198	apsi.f:5100		
Iterations	SelfIns	TotalIns	
34M	8G	163G	
TripCount	Entries		
2	17436832		

loop: 0x80647ba	apsi.f:5721		
Iterations	SelfIns	TotalIns	
20M	4G	206G	
TripCount	Entries		
2	10459008		

loop: 0x8053e45	apsi.f:1967		
Iterations	SelfIns	TotalIns	
3M	3G	315G	
TripCount	Entries		
> 10	31151		

loop: 0x8054365	apsi.f:2087		
Iterations	SelfIns	TotalIns	
3M	3G	315G	
TripCount	Entries		
> 10	31151		

Figure 5: An excerpt LoopProf’s flat instruction and iteration profile for 301.apsi. Loops are sorted descendingly by the percentage of total execution.

This equation measures the weighted difference in contribution for each loop in the instrumented profile. $loops_{inst}$ is the set of loops detected by instrumentation. Each of $loops_{inst}$ is compared to the corresponding sampled loop, if it exists, and weighted by the percentage of execution that it accounts for. The accuracy results for LoopSampler are calculated for both self and total samples, and the results are shown in Figure 4. The graph shows that, on average, the sampler is accurate within 95% of the instrumentation-based mechanism. For the outlying benchmarks that have lower accuracy, we believe this to be mostly an effect of comparing *time* versus *instructions*. That is, the low correlation between counts in sampled and instrumented profiles is probably a result of different regions of code having much different IPC.

5.3 Case Study: SPEC OMP

To demonstrate the capabilities of loop-centric profiling, this section evaluates a profile collected for the 301.apsi benchmark in the SPEC2000 benchmark suite. In the following discussion, loops will be referred to in the format *filename:lineno*, which is gathered from the program’s debug information.

The loop call graph in Figure 6 shows an example loop call graph. Each node presents the function’s name or loop’s filename and line number. In addition, each node contains the percentage of self and total execution that it contributes. Loops are identified by oval nodes and functions by rectangular nodes. Because there are hundreds of loops in the program, only those that contribute greater than 1% to total execution are displayed.

Figure 5 shows an excerpt from the flat profile generated for 301.apsi. This profile shows, for each loop, its address, self and total instruction counts, and iteration buckets. Note that the sampled version of this file does not contain iteration data. The iteration buckets are a list of the trip counts and entries. For example, there were 10,459,008 entries to apsi.f:5721 where it iterated 2 times before exiting. For space and readability, more detailed instruction profiles were omitted and the maximum number of buckets was limited to 10.

Given a sequential program, deciding where and how to parallelize the code is often tedious and time consuming. Traditional profilers are well suited to analyzing hot spots in code and increasing performance in inner loops, but effective thread-level parallelization requires coarse granularity that is not exposed by modern profiling tools. The loop call graph feature displays the hierarchy of loop execution and the percentage of total execution each loop accounts for.

With this view of structure combined with an execution profile, deciding which loops to try to parallelize is greatly simplified. Those loops that reside high in the hierarchy and account for the largest percentage of total execution are the best targets. These loops may execute very few instructions themselves, but loops may be nested deeply within them. In contrast, traditional profiling tends to focus on hot inner loops and functions that are most frequently called. This information is misleading to would-be parallelizers; they could either try to parallelize the wrong loops or spend hours figuring out which outer loops ultimately contain the hot, inner loops.

The SPEC2001 OMP benchmark suite contains benchmarks that have been hand-parallelized using OpenMP. Most of these benchmarks are modified versions of their sequential counterparts found in the SPEC2000 CPU suite. To evaluate the effectiveness of our tool, we use it to compare the sequential and parallel versions of four such SPEC benchmarks. First, we profile the sequential versions of the programs to identify loops that are good targets for parallelism. Then, we search for corresponding loops in the OMP benchmark source code to see if they are parallelized. In the benchmarks studied, we confirm the decisions of the benchmarks’ respective authors.

Figure 6 shows the loop call graph generated for the sequential benchmark, 301.apsi. For this program, a traditional profiler might focus on the loops at the bottom of the graph, those that are deeply nested and where most execution time is spent. For the purposes of creating coarse-grain parallelism, however, the loops near the top have the most potential. The highlighted loops were actually parallelized in the OpenMP benchmark, 325.apsi. All of the loops that were parallelized were high-level, and the flat profile shows that they executed very few self instructions.

If a traditional profiler were used to help find coarse-grained parallelism in 301.apsi, the user would first be drawn to the `run` function and its immediate children. The `run` function alone is 651 lines of code, and its children contain many thousands more. In addition, there are many loops in those functions that are not significant. This makes finding the best loops tedious and time consuming. Loop-centric profiling expedites the process of identifying the outer loops that are the best candidates for parallelization.

The tool can also be used to give some insight into how effectively the authors of 301.apsi were able to parallelize the sequential code. For this data set, all of the significant top-level loops have been parallelized, implying that the best opportunities for performance gains were exercised. However, many loops that were parallelized do not account for a significant portion of execution. From the 301.apsi source code, 28 loops were parallelized with OpenMP pragmas. However, only 16 outer loops account for over 99% of total execution. Therefore, either the authors of 301.apsi did too much work, or other input sets, which are not part of the SPEC suite, may stress different parts of the program.

Figure 7 shows the loop call graph for 191.fma3d. We see a similar trend as in 301.apsi; good target loops are high in the graph and account for nearly all of total execution. Interestingly, there are two high-level loops in 191.fma3d that did not get parallelized. Both of those loops correspond to code that reads data and builds data structures, a task that is much harder to parallelize.

We have found that most of the other SPEC OMP benchmarks exhibit similar properties, specifically 179.art and 171.swim. This shows that loop-centric profiling is effective at identifying the same loops that an experienced human would choose to parallelize.

6. FUTURE WORK

For parallelization, the information that loop-centric profiling tools provide is a considerable improvement over what current profilers offer. However, there is still much work to be done. Loop-Sampler is very good at identifying hot outer loops, but lacks some of the detailed information that the (slower) LoopProf can offer. We are working on applying code patching at specific application points to collect information about loop entries and iterations without significantly slowing down the program. More significant enhancements include profiling loop-carried data dependences and identifying potential race conditions that could be exposed by parallelizing loops. We believe this level of information will be immensely useful in guiding developers to create coarse-grain threading into their applications, and could ultimately aid in automatic parallelization tools.

7. CONCLUSION

One of the major obstacles to developing parallel code, especially in large applications, is identifying the best places to parallelize. Loops are an easy and common target for parallelization, but without sufficient profile information, it is difficult to decide which loops can be parallelized efficiently and with low overhead. We introduce loop-centric profiling to target this problem. By generating a loop call graph and profiling self and total time, loop-centric profilers effectively discover which loops the developer should try to parallelize. Because the loop detection algorithms do not rely on the compiler, they can easily be adopted into current software development cycles. This paper shows that the technique can be effective in finding which loops to parallelize, but there is still much work to be done in aiding the developer in discovering “how” to parallelize.

8. REFERENCES

- [1] The libunwind project. <http://www.hpl.hp.com/research/linux/libunwind/>.
- [2] OProfile. <http://oprofile.sourceforge.net>.
- [3] M. R. D. Alba, D. R. Kaeli, and E. Kim. Analisis dinámico de bloques iterativos. In *Tercer Congreso Internacional en Control, Instrumentación Virtual y Sistemas Digitales*, pages 93–106, Ciudad de Mexico, Mexico, Agosto 2001.
- [4] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.
- [5] M. de Alba and D. Kaeli. Runtime predictability of loops. In *IEEE International Workshop on Workload Characterization*, pages 91–98, Austin, TX, December 2001., 2001.
- [6] M. de Alba and D. Kaeli. "characterization and evaluation of hardware loop unrolling". Technical report, Electrical and Computer Engineering Department at Northeastern University, 2002.
- [7] M. de Alba and D. Kaeli. Path-based hardware loop prediction. In *The 4th International Conference on Control, Virtual Instrumentation and Digital Systems*, August 2002.
- [8] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 292–302, December 1997.
- [9] J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2003.
- [10] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM Press.
- [11] M. Kobayashi. Dynamic characteristics of loops. *IEEE Trans. Computers*, 33(2):125–132, 1984.
- [12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.
- [13] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri. LoopProf: Dynamic Techniques for Loop Detection and Profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.
- [14] T. Sherwood and B. Calder. Loop termination prediction. In *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing*, pages 73–87, London, UK, 2000.
- [15] A. Shye, M. Iyer, T. Moseley, D. Hodgdon, D. Fay, V. Janapareddi, and D. Connors. Analysis of path profiling information generated with performance monitoring hardware. In *Proceedings of the 9th Workshop on Interaction between Compilers and Computer Architecture*, February 2005.
- [16] Standard Performance Evaluation Corporation. The SPEC CPU 2000 benchmark suite, 2000.
- [17] J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 14, Washington, DC, USA, 1998. IEEE Computer Society.