LoopProf: Dynamic Techniques for Loop Detection and Profiling

Tipp Moseley[†], Dirk Grunwald[†], Daniel A. Connors[†],
Ram Ramanujam[‡], Vasanth Tovinkere[‡], Ramesh Peri[‡]

[†]Department of Computer Science
University of Colorado
Boulder, CO 80309

[‡]Intel Corporation
Software and Solutions Group

Abstract

As processors transition to multithreaded, multi-core designs, 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 addition to instruction-level parallelism (ILP) and increased clock frequency for improving performance. While many tools exist that attempt to automatically parallelize code, manual techniques remain the predominant method of exploiting TLP. Deciding where to try to parallelize code is difficult, especially for large, complex applications or those where the original developers have moved on. In general, loops have been relatively easy targets for parallelization efforts. In order to get the best scaling, it is important to focus on larger grain parallelization which means moving up loop hierarchies. To aid this task, programmers need to have a hierarchical view of how much time is spent in a loop and loops nested within it.

Our tool, LoopProf, is primarily intended to guide the developer in increasing parallelism in predominantly sequential codes. It can detect loops and collect detailed profile information including self and total instruction counts and histogram information for trip counts. In the past, collecting this type of profile information has relied on compiler or hand instrumentation. However, LoopProf is designed to work transparently with unmodified binary programs with no compiler dependence other than debug information. We present algorithms for dynamically detecting program loops in the absence of a control flow graph and accounting for instructions in a hierarchy of nested loops. The paper concludes with a case study evaluating the tool on several SPEC 2000 benchmarks.

1. Introduction

The transition to multithreaded, multi-core processor designs means that 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. Loops are easy targets for parallelization. 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 [2] [6], and regions that are the most active, or hot, are then given special attention by the programmer. Similarly, tools like gprof [7] 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), time spent in each function including children (total time). Performance analysis tools designed with the intent to analyze sequential code are very mature, however 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. In the best case, the programmer can simply use OpenMP directives to parallelize loops. In other cases, some work must be done to ensure correct semantics and prevent data races. 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. Therefore, one part of an ideal tool would create a loop graph profile similar in nature to the call graph, but identify parent-child relationships and self and total instruction counts of loops instead of functions.

Our tool, LoopProf, is implemented as a Pintool using the Pin [9] 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. Dynamic instrumentation is particularly beneficial for this type of tool because it captures the execution of arbitrary shared libraries in addition to the main program, and it has no dependence on the instrumented application's compiler. Requiring only a binary and being compiler independent does not imply that the tool is useful without source code. Instead, it provides flexibility for the tool to be language independent and it can be used with any compiler toolchain that produces a common binary format Furthermore, it does not require the user to modify the build environment to recompile the application with special profiling flags.

Since LoopProf relies on dynamic instrumentation and is compiler-independent, loop detection must be done in the absence of a control flow graph. Therefore loop detection is based on the dynamic execution of the program. Section 3 describes in detail the algorithms used. LoopProf calculates a hierarchical loop profile, logically similar to a call graph, for loops in a program. For every loop detected in an execution of the program, LoopProf is capable of providing the following information:

- The number of entries to the loop.
- The total number of loop iterations.
- Histogram information for the number of iterations executed per loop entry.
- Self and total instruction counts.
- Per-loop self and total counts for each type of instruction (e.g. load, store, fp, etc.).
- The loop's nesting depth.
- The parent and child loops in the loop call graph.

These features are intended to guide optimization on many different levels. From a static perspective, the compiler can use this profile information to unroll loops and perform inter-procedural code transformation. The programmer can use this information to help identify loops have the most potential gain if converted to multiple threads. It can also be used to identify areas where the program is data or compute intensive. More interestingly, the loop detection algorithm's independence from the compiler makes it a viable tool for use in a run-time optimization system.

The rest of this paper is organized as follows. Section 2 describes related profiling work. Section 3 discusses the loop profiling methodology. Section 4 presents performance overhead and a case study. Section 5 describes future work. Finally, Section 6 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[2], OProfile, or Intel® VTune™ naturally identify such inner loops as 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[7], 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. Since dynamic instrumentation incurs a high and variable overhead, LoopProf uses executed instructions instead.

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 detects loops at analyzes program traces to detect loops and characterize their different properties [8]. De Alba *et. al.* also perform several loop studies[1] aiming to improve path prediction inside of loops[3][5] and enable dynamic loop unrolling[4]. Loop-back, or "loop termination" edges are frequently mispredicted. Sherwood *et. al.*[10] propose both hardware and software techniques for "loop termination prediction" to increase branch prediction accuracy with loop branches.

The algorithm for loop detection proposed in this paper is based on an algorithm used by Shye *et. al.*[11] 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 was similar to the one proposed in this paper, except loop back edges were used to terminate a path instead of define a loop.

In a more germane study[13], 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 LoopProf tool. 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 time 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.

3. Methodology

3.1. Background

Since LoopProf works without any compiler support, it dynamically discovers loops in a program based on the basic block (BBL) path of execution. Pin, like other dynamic instrumentation systems, does not characterize basic blocks in the same sense as the compiler; basic blocks 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.

```
// Data Structures
class LoopInfo {
  // Loop accounting variables for iterations,
  // self/total instructions, and children
};

class StackFrame {
  list<BblPathInfo *> path;
};

class BblPathInfo {
  addr_t head;
  // Other accounting information
};

// Variables
list<StackFrame> callStack;
HashTable< addr_t, LoopInfo *> loops;

// Loop Detection Algorithm
void processLoop(StackFrame frame,
                 BblPathInfo *bpi) {
  if( !loops.hasKey(bpi->head) ) {
    loops[bpi->head] = new LoopInfo();
  }

  doInstructionAccounting(frame, bpi)
  attachGraphEdges(frame, bpi);

  popBbls(frame, bpi);
}

// Called each time Bbl is executed
void processBbl(addr_t bblhead,
                addr_t sp) {
  // Adjust the callstack
  adjustStack(sp);

  StackFrame &frame = callStack.back();

  if( frame.path.contains(bblhead) ) {
    // Loop detected; do instruction accounting
    // and pop BBLs above this BBL.
    BblPathInfo *bpi = findBPI(frame, bblhead);
    processLoop(bpi);
  } else {
    newBpi = new BblPathInfo(bblhead, sp);
    frame.path.push(newBpi);
  }
}
```

Figure 1. C++ pseudocode for loop detection.

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

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

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 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 `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. The `BblPathInfo` object contains the address of the BBL and other information necessary for profiling. Again, because of recursion, this accounting may not be kept with a global BBL object because one BBL may be on 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

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. 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 accounted for in H'.
- 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 top level instructions from the function are added to the self instructions of the calling BBL, and the total counts from any top-level loops in the function are added to the total instructions of the calling BBL. Later, when the calling BBL is removed from the path, its total instructions are attributed to the loop it is associated with.

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 top-level loops in that frame are added to a list in the last BBL on the path after the frame is popped. This BBL, B, is 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.

4. Results

As output, LoopProf produces two files. The first file contains the flat instruction and iteration profile information. The second is a list of edges in the loop call graph. There is usually too much raw information to deal with visually, so LoopProf includes several post-processing tools to manipulate the data. In a large program, thousands or

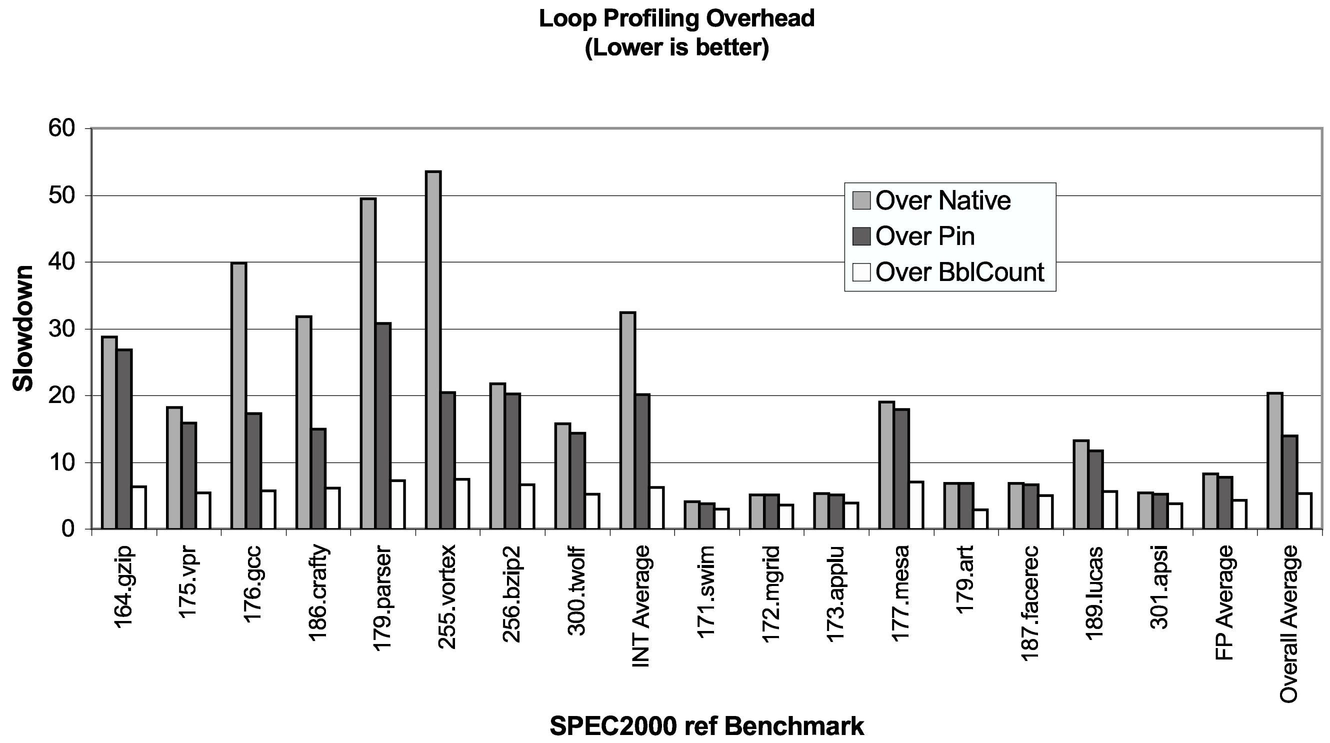

Figure 2. The runtime overhead of the LoopProf tool when compared to applications run natively, executed with no instrumentation under Pin, and instrumented by the BblCount Pintool.

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 [12] benchmarks tested, many thousands of loops were detected, but this pruning threshold was sufficient to reduce the number of loops to tens or hundreds. It is also useful to be able to sort the loops on any number of fields, such as self or total instructions, iterations, memory reads and writes, floating point operations, etc. In addition, a simple GUI is used to view the loop call graph.

4.1. Performance

Complex binary instrumentation tools are known to incur high overhead, and LoopProf is no different. Figure 2 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 (32), and FP benchmarks are quite reasonable (8). 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 20.4 times slower than native execution. Because using this tool shouldn't generally require frequent or iterative profiling, we believe this number to be reasonable. A typical use case might involve profiling a program with a few different input sets (since the profile is execution-dependent), then optimizing the program based on those results.

In an effort to reduce the overhead 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).

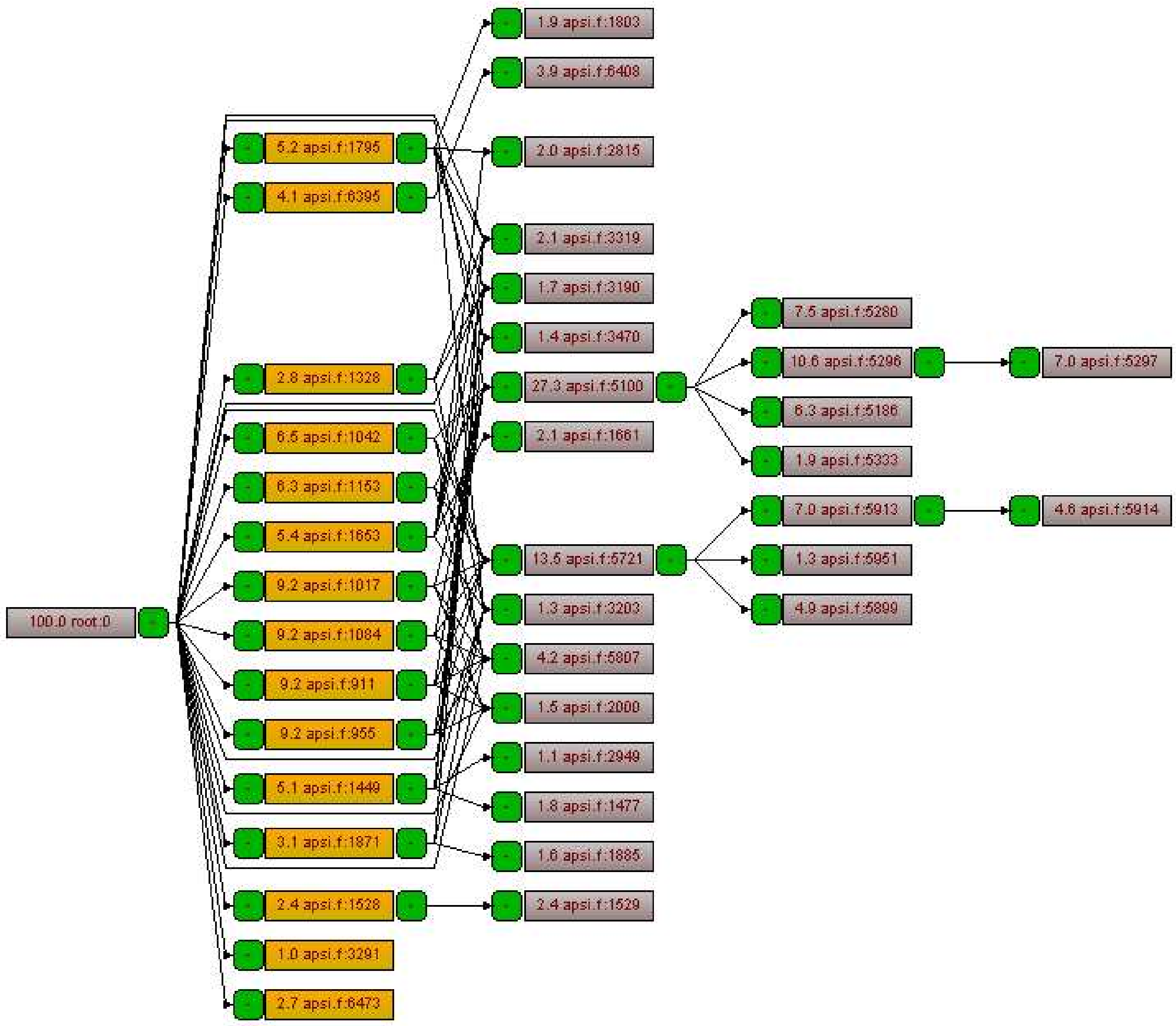

Figure 3. A loop call graph for the sequential version of 301.apsi in the SPEC2000 CPU benchmark suite. Each node contains the percent of total execution, filename, and line number for a loop. Highlighted nodes were parallelized by the programs' authors in the SPEC2001 OMP suite, and gray nodes were not parallelized.

4.2. 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 50 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.

4.3. Case Study: SPEC OMP

To demonstrate the capabilities of the LoopProf tool, 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*.

The loop call graph in Figure 3 shows what is displayed by the GUI viewing utility. Each node presents the percentage of total execution that a loop contributes, and when the mouse rolls over the node, more detailed information about the loop is displayed. Because there are hundreds of loops

```
---------------------------------------------
Loop at 0x08066253 (apsi.f:5100)
   Total Iterations:    Self Ins:   Total Ins:
            1365129    587005470  17419956126
                            0.92 %      27.27 %

    TripCount     Entries
            3      455043
---------------------------------------------
Loop at 0x0806c965 (apsi.f:5721)
   Total Iterations:    Self Ins:   Total Ins:
             818646    256236198   8656089922
                            0.40 %      13.55 %

    TripCount     Entries
            3      272882
---------------------------------------------
Loop at 0x08067fdb (apsi.f:5296)
   Total Iterations:    Self Ins:   Total Ins:
            3640344   2328455031   6778320528
                            3.64 %      10.61 %

    TripCount     Entries
            1      455043
            7      455043
---------------------------------------------
Loop at 0x08052a4f (apsi.f:1017)
   Total Iterations:    Self Ins:   Total Ins:
              44640     32160240   5897752560
                            0.05 %       9.23 %

    TripCount     Entries
          >10       44640
---------------------------------------------
Loop at 0x08067ffc (apsi.f:5297)
   Total Iterations:    Self Ins:   Total Ins:
            5460516   4449865497   4449865497
                            6.97 %       6.97 %

    TripCount     Entries
            0     3640344
            6      910086
---------------------------------------------
Loop at 0x08056075 (apsi.f:1795)
   Total Iterations:    Self Ins:   Total Ins:
              91440    104509440   3293521920
                            0.16 %       5.16 %

    TripCount     Entries
          >10       91440
---------------------------------------------
```

Figure 4. An excerpt LoopProf's flat instruction and iteration profile for 301.apsi. Loops are sorted descending by the percentage of total execution.

in the program, only loops that contribute greater than 1% to total execution are displayed.

Figure 4 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. The iteration buckets are a list of the trip counts and entries. For example, there were 910,086 entries to apsi.f:5297 where it iterated 6 times before exiting. For space and readability, 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. LoopProf's 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 contain the hot, inner loops.

The SPEC2001 OMP benchmark suite contains benchmarks that have been parallelized using OpenMP. Several 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 three SPEC benchmarks. First, we profile the sequential versions of the programs to identify loops that are good targets for parallelism. Then, we search for these candidates in the OMP benchmark source code to see if they are parallelized. We place some degree of faith in the decisions of the benchmarks' respective authors.

Figure 3 shows the loop call graph generated for the sequential benchmark, 301.apsi. The full graph is large and unwieldy, so this graph only shows loops that accounts for more than 1% of total execution. For this program, a traditional profiler might focus on the loops to the far right in 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 further to the left have the most potential. The highlighted loops were actually parallelized in the OpenMP benchmark, 325.apsi, and those in gray were not. All of the loops that were parallelized were high-level, and the flat profile shows that they executed very few self instructions.

The tool can also be used to give some insight into how well the authors of 301.apsi did. For this data set, all of the significant top-level loops have been parallelized. This implies that the best opportunities for performance gains were exercised. However, loops that were parallelized don't account for a significant portion of execution. From the 301.apsi source code, 28 loops were parallelized with OpenMP pragmas. However, only 15 outer loops account for over 99% of total execution. Therefore, either the authors of 301.apsi did too much work, or other input sets may stress different parts of the program.

The benchmarks 179.art and 171.swim yield similar results. This shows that LoopProf is effective at identifying the same loops that an experienced human would choose to parallelize.

5. Future Work

For parallelization, the information that LoopProf provides is a considerable improvement over what current profilers offer. However, there is still much work to be done. In the short term, some simple features will be added: the tool GUI will include weights on the edges in the loop call graph and combine the function call graph with the loop call graph to get a more detailed view of program structure. More significant enhancements include profiling loop-carried data dependences and identifying potential race conditions that could be exposed by parallelizing loops. We believe that 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.

Another issue with the LoopProf tool is profiling overhead. For some benchmarks, the tool's overhead is quite reasonable, but in many cases it is above what a normal programmer may tolerate. We are currently investigating potential solutions to mitigate this problem. We have investigated methods of building the CFG ahead of time by using heuristics to reason about the effects of indirect branches. The results were quite similar to those presented in this paper for both overhead and information reported. We are also developing a statistical sampling-based scheme to collect the same type of information, and preliminary results are promising.

6. 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. LoopProf targets this problem. By generating a loop call graph and profiling self and total instruction counts, it effectively discovers which loops the developer should try to parallelize. Because the loop detection algorithm does not rely on the compiler, the tool can be used with any compiler toolchain that produces Linux binaries for Intel® ARM, IA32, EM64T (64-bit x86), and Itanium® architectures. 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.

Acknowledgments

The authors would like to thank Robert Cohn and the rest of the Pin team for the exceptional support provided and answering many questions. Also, we wish to thank J. Bradley Chen for his input to the project.

References

[1] M. R. D. Alba, D. R. Kaeli, and E. Kim. Analisis dinamico de bloques iterativos. In *Tercer Congreso Internacional en Control, Instrumentacion Virtual y Sistemas Digitales*, pages 93–106, Ciudad de Mexico, Mexico, Agosto 2001.

[2] 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.

[3] M. de Alba and D. Kaeli. Runtime predictability of loops. In *The 4th Annual IEEE International Workshop on Workload Characterization, pages 91–98, Austin, TX, December 2001.*, 2001.

[4] M. de Alba and D. Kaeli. "characterization and evaluation of hardware loop unrolling". Technical report, Electrical and Computer Engineering Department at Northeastern University, 2002.

[5] 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.

[6] 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.

[7] 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.

[8] M. Kobayashi. Dynamic characteristics of loops. *IEEE Trans. Computers*, 33(2):125–132, 1984.

[9] 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.

[10] 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.

[11] 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.

[12] Standard Performance Evaluation Corporation. The SPEC CPU 2000 benchmark suite, 2000.

[13] J. Tubella and A. Gonzlez. 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.