LAMPVIEW: A LOOP-AWARE TOOLSET FOR FACILITATING PARALLELIZATION

THOMAS RORIE MASON

A THESIS FOR THE DEGREE
OF MASTER OF SCIENCE IN ENGINEERING

DEPARTMENT OF
ELECTRICAL ENGINEERING

ADVISOR: DAVID I. AUGUST

AUGUST 2009

Abstract

A continual growth of the number of transistors per unit area coupled with diminishing returns from traditional microarchitectural and clock frequency improvements has led processor manufacturers to place multiple cores on a single chip. However, only multi-threaded code can fully take advantage of the new multicore processors; legacy single-threaded code does not benefit. Many approaches to parallelization have been explored, including both manual and automatic techniques.

Unfortunately, research in this area is impeded by the innate difficulty of exploring code by hand for new possible parallelization schemes. Regardless of whether it is a researcher attempting to discover possible automatic techniques or a programmer trying to make manual parallelization, the benefits of good dependence information are substantial. This thesis provides a profiling and analysis toolset aimed at easing a programmer or researcher's effort in finding parallelism. The toolset, The Loop-Aware Memory Profile Viewing System (LAMPView), is developed in three parts.

The first part is a multi-frontend, multi-target compiler pass written to instrument the code with calls to the Loop-Aware Memory Profiling (LAMP) library. The compile-time instrumentation was partially developed previously and has been augmented here with additional features. The second part is a post-runtime processing pass that translates the output of the profiling run from a machine-level view to a source-level view. As it translates, it also processes and sorts dependence information. The third and final part is a pair of stand-alone utilities that takes the translated information and provides the user with human-readable output that is searchable by various parameters. In various discussions with potential users, it has been seen that the utility eases analysis of parallelism opportunities.

Acknowledgments

I would like to first thank my advisor Professor David August for welcoming me into the Liberty Research Group and for his guidance throughout the writing of this thesis. Additionally, I would like to thank the entire Liberty Research Group for their help throughout my graduate career – their support as well as their suggestions were invaluable. I have learned a great deal from working with the Liberty Group that I would not have learned without them. I extend a special thank you to Prakash for helping me get started in the group and to Arun for making me a part of his project in my earlier months and for all of his recommendations that made the LAMPView toolset better.

I also thank my friends at Princeton who have made this experience more interesting than it possibly could have been alone. Our good times together made Princeton really feel like my new home. Finally, I would like to thank my mother, my father, and my sister for their support throughout my life and especially as I worked through my graduate career at Princeton.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

For decades, continual exponential increases in clock speed alongside microarchitectural improvements yielded improvements for all programs without any change to the programs themselves. However, starting around 2004, this was no longer the case. This has caused a decrease in rate of improvement of application performance seen as seen in the SPEC Benchmark performance graph in Figure 1.1 [29]. The change was caused primarily by two factors. The first of these was that the exponential clock speed increase for uniprocessor systems was no longer sustainable as power became a primary design constraint. These power constraints led many manufacturers to scale down clock frequencies. The second of these factors was that microarchitectural advancements, faced with greatly increased design complexity as well as this power wall, were giving diminishing returns in terms of performance.

Nonetheless, Moore's law has continued to hold, granting processor designers higher numbers of transistors per die. With traditional design improvements no longer able to deliver improved performance, designers and manufacturers shifted to manufacture multi-core processors. Multi-core processors allow designers to use the extra transistors while

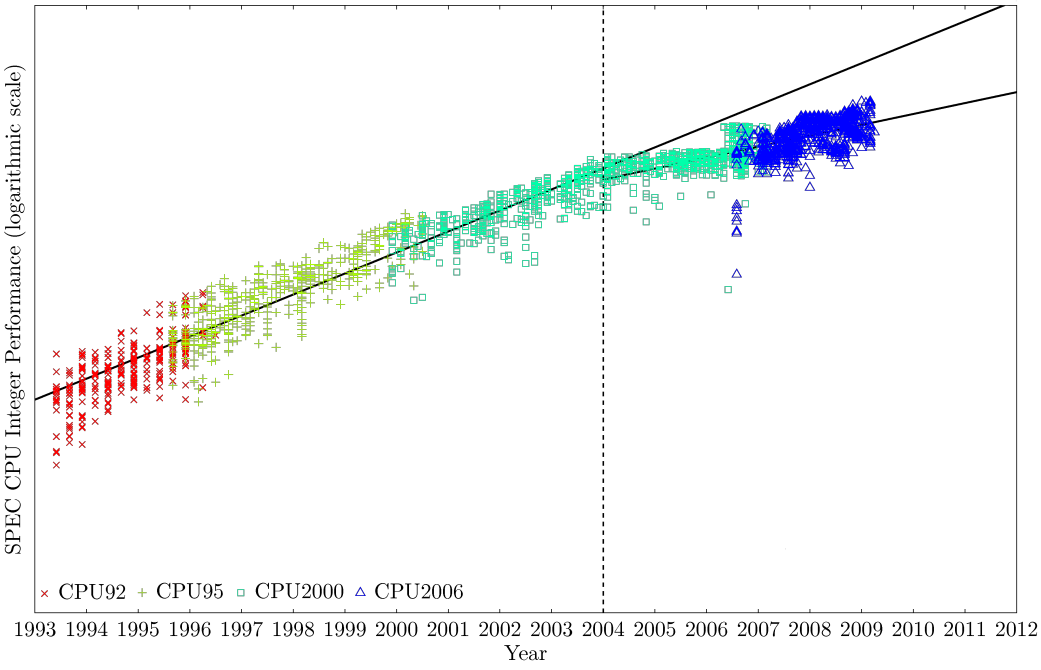


Figure 1.1: Normalized Performance of SPEC 92, 95, 2000, 2006 integer benchmark suites. Each version of the suite is normalized to the previous version using matching hardware with only the highest performance results for each time period shown. Upper line is a linear regression on all points prior to 2004 while the lower is a linear regression on all points during and after 2004. Source Data from [29]

working within the constraints that have made uniprocessor techniques no longer feasible.

In a sense, the problem of using transistors was solved. However, the declining performance gains of an individual piece of software is not solved by this paradigm shift. While multi-threaded applications benefit from higher numbers of parallel processors, legacy single-threaded code does not benefit at all. As a result, many techniques have developed to make single-threaded programs parallel. Automatic techniques and tools are useful as the programmer faces many problems in parallelizing programs including race conditions, deadlock, livelock, and more [10][19][11]. Sequential code, not designed with parallelism in mind, poses numerous difficulties in determining dependences that exist and transforming the code to a parallel form. Manual parallelization, therefore, is difficult. Research in the automatic parallelization domain is not exempt from these difficulties, however. Often, prior to automation, parallelization techniques are applied by hand to a small set of applications. This poses significant challenges to a researcher investigating new automatic parallelization techniques.

1.1.1 Contributions and Summary

The contribution of this thesis is LAMPView, an instrumentation and profiling system built partially within the LLVM compiler and partially as a stand-alone utility. The system serves as a toolset to a programmer or researcher gathering dependence information about a program. It also provides a basis for further integration of profiling into an optimizing compilation framework such as LLVM.

The second chapter provides further background on existing techniques that exploit parallelism as well as introducing profiling in its capacity for discovering profitable parallelizations. The third chapter introduces the Loop-Aware Memory Profile Viewing toolset (LAMPView). Included in this work is a set of profiling and analysis assistance tools augmented or developed as a part of this thesis to aid researchers in manual exploration of parallelization techniques. The third chapter concludes with a demonstration of the utility

of these tools in analyzing several benchmarks. The fourth chapter mentions some potential for extension work.

Chapter 2

Background

There are already many techniques for use in automatically parallelizing code. The techniques for extracting parallelism from loops fall into three broad categories including independent multi-threading (IMT), cyclic multi-threading (CMT), and pipelined multi-threading (PMT). Each of these depend heavily on data dependence information which must be provided either by static analysis or by profiling and, if possible, by both.

Profiling involves instrumenting the source code or binary with code to collect data at run time. This instrumentation can occur either offline prior to execution or online in an on-the-fly fashion. Instrumented code runs on representative sets of data in order to collect information about the program. Information collected from the profiling runs can be used as a supplement for analysis as it provides information that may not be determined practically (or at all) by static analysis. Profile information can be useful to any programmer or researcher trying to apply the various techniques to be discussed in the background section of this thesis. Furthermore, a loop-sensitive profiler such as the one described in this thesis provides greater utility to a programmer looking for opportunities not simply for parallelization but also for speculation to enhance parallelization.

2.1 Independent Multi-Threading

2.1.1 Introduction

Independent multi-threading techniques prohibit cross-thread dependences. The most well known of these techniques, DOALL parallelization, was originally introduced to parallelize loops in array-based scientific programs. In this technique, a loop is divided into several iteration groups, and each group is executed concurrently. This technique, with no inter-thread communication necessary for worker threads while it executes its iterations, is highly scalable. In fact, it scales linearly with the number of worker threads.

2.1.2 DOALL Parallelization

In DOALL parallelization, two iterations are executed simultaneously without any synchronization. In 1966, Bernstein [5] described three conditions for two program blocks *i* and *j* to be safely executed in parallel. If a loop iteration does not have any other loop exit condition except an iteration condition, it can be considered as a program block, and Bernstein's conditions can be applied at an iteration level. The conditions can be translated into the following dependence conditions which can be checked by a compiler in dependence analysis. If a loop satisfies all conditions, the loop can be parallelized in a DOALL manner.

1. no loop-carried WAR dependences
2. no loop-carried RAW dependences
3. no loop-carried WAW dependences
4. no loop-carried control dependences except an iteration condition

The example in Figure 2.1 can be safely parallelized in a DOALL manner because it satisfies all DOALL dependency conditions. As an example, this loop is parallelized into M threads. Each thread is given a unique ID from 0 to M-1 which will be used by the thread to

```
1  int a[N], b[N], c[N];
2  for(i=0;i<N;i++){
3    c[i] = a[i]+b[i];
4  }
```

Figure 2.1: DOALL code example

```
1  Sequential Program Variables:
2    int a[N], b[N], c[N], tid[M];
3
4  Parallel Code:
5    void doall(int *tid){
6      int id = *tid;
7      for(i=id*N/M; i<(id+1)*N/M; i++){
8        c[i] = a[i]+b[i];
9      }
10   }
```

Figure 2.2: DOALL parallelization for Example 2.1

select iterations to execute in the parallelized code. The iterations are divided thus divided into M groups. Two typical grouping schemes are that of modular value and that of range. In a modular-value-based scheme, thread *i* executes iteration *j* where $i = j$ MOD N. In a range-based scheme, thread *i* executes iteration *j* where $i \in [i * N/M, (i + 1) * N/M)$. In general, range-based grouping shows better performance than a modular-based grouping because it has higher cache locality. The code in Figure 2.2 shows the parallelized code with range-based grouping, and Figure 2.3 shows its execution model.

2.1.3 Augmenting Independent Multi-Threading

Even though we can apply DOALL parallelization in this example, the conditions become too restricting in many cases. Fortunately, some dependences can be ignored in special conditions or by using certain transformations. The simplest of these is privatization.

In the example in Figure 2.4, there are loop-carried WAW and WAR dependences on *tmp* so it cannot be parallelized using DOALL because the wrong *tmp* value can be read after for loop execution and *tmp* updated in a later iteration can be read. If *tmp* is not read

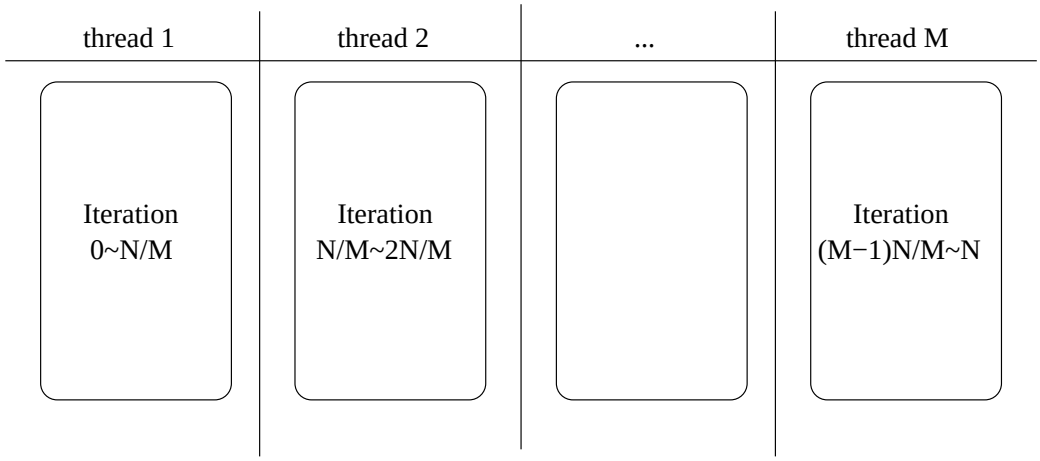


Figure 2.3: DOALL execution model for Example 2.1

```
1  int a[N], b[N], c[N];
2  int tmp;
3  for(i=0;i<N;i++){
4    tmp = a[i]+b[i];
5    d[i] = tmp*c[i];
6  }
```

Figure 2.4: Sequential code for privatization example

after the for loop, the loop-carried WAW dependence can be ignored. If *tmp* is not live-out, the WAW dependence can be ignored. The WAR dependence can be also ignored if *tmp* is privatized. The code in Figure 2.5 takes advantage of these facts to parallelize the loop in a DOALL parallelization.

The second common transformation to enable independent multi-threading parallelization is that of reduction. Operations such as summing the elements of an array are common across many programs. However, since there is a loop-carried dependence on the sum variable, the loop cannot be parallelized by the DOALL technique directly. In such cases, reduction is useful. Reduction can be applied not only to summing elements, but to all commutative operations, including multiplication, finding the maximum or minimum element in an array and counting the number of elements meeting some condition. These latter two reductions are called as max/min reduction and count reduction [3].

```
1  Sequential Program Variables:
2    int a[N], b[N], c[N], d[N], tid[M];
3
4  Parallel Code:
5  void doall(int *tid){
6    int id = *tid;
7    int tmp;
8    for(i=id*N/M;i<(id+1)*N/M;i++){
9      tmp = a[i]+b[i];
10     d[i] = tmp*c[i];
11   }
12 }
```

Figure 2.5: Parallelized code for privatization example from Example 2.4

```
1 int a[N], b[N];
2 int sum = 0;
3 for(i=0;i<N;i++){
4   sum = sum + a[i]*b[i];
5 }
6 printf("sum: %d\n", sum);
```

Figure 2.6: Sequential code for DOALL reduction example

The example in Figure 2.6 is useful for demonstration of the utility of sum reduction. As mentioned before, this loop cannot be parallelized with the DOALL technique because there are loop-carried WAW and RAW dependences on sum. Add operations are commutative and associative, and so any set of add operations can be performed in any order. A sum operation, therefore, can be divided into several groups and then the sub-sums from each group can be accumulated after computation, yielding the correct result. In this manner, the code in Figure 2.7 can be generated.

Sub-sums are initialized in the first loop, and they are calculated in each group in the second loop. In the last loop, the final sum is calculated by accumulating sub-sum results. Unlike the previous code, in this code, L1 can be parallelized using DOALL because L2 loops for different *j* can be executed at the same time.

```
1       int a[N], b[N], s[M];
2       int sum = 0;
3       for(j=0;j<M;j++){
4         s[j] = 0;
5       }
6   L1: for(j=0;j<M;j++){
7   L2:   for(i=j;i<N;i+=M){
8           s[j] = s[j] + a[i]*b[i];
9         }
10      }
11      for(j=0;j<M;j++){
12        sum = sum + s[j];
13      }
14      printf("sum: %d\n", sum);
```

Figure 2.7: Parallelized code for DOALL reduction example from Example 2.6

2.1.4 Advanced Transformations for Independent Multi-Threading

While the DOALL technique and the extensions discussed above are frequently useful in parallelization of scientific code (and occasionally more general purpose programs), their utility is still severely limited by the structure of the program constructed by the original programmer. The structure may inhibit parallelism entirely or it may simply make it unprofitable to parallelize in its current form because results must be communicated and synchronized too frequently. There are, however, many other code transformations that can be done by a programmer or a compiler.

One technique, perhaps the simplest technique, that can be applied to make parallel execution more profitable is that of *loop interchange* [2]. The code seen in Figure 2.8 provides such an example. The loop at statement 2 is immediately parallelizable by the DOALL technique. Ignoring the *i* indices which would remain constant across an invocation of the loop, this is precisely the same as the loop seen at the beginning of Section 2.1.2. However, assuming these loops execute for a large number of iterations as is often the case in practice, results must be communicated N times which is undesirable. However, in its current form, there is a dependence across iterations of the outer loop (statement 1) which prohibits parallelization of the outer loop, a parallelization which would reduce communication.

```
1 for (i = 0; i < N; i++) {
2   for (j = 0; j < M; j++) {
3     A[i+1][j] = A[i][j] + B[i][j];
4   }
5 }
```

Figure 2.8: Code example for loop interchange

```
1 for (j = 0; j < M; j++) {
2   for (i = 0; i < N; i++) {
3     A[i+1][j] = A[i][j] + B[i][j];
4   }
5 }
```

Figure 2.9: Code for interchanged loops from Example 2.8

The loops, however, can be interchanged without altering the resultant array. The code for the interchanged loops can be seen in Figure 2.9. In this transformed code, the dependence recurrence is now across the inner loop while iterations of the outer loop fulfill the criteria for making a successful DOALL parallelization. The iterations of the outer loop can now be distributed to different threads such that the results need only be communicated at the end.

Another useful loop transformation that can obtain DOALL parallelism in scientific programs where none exists in its initial form is *loop distribution*. Loop distribution takes what is originally a single loop with multiple statements with inter-iteration dependences and splits it into multiple loops where there are no inter-iteration dependences. Figure 2.10 is an example of code that can benefit from such a transformation. There is an inter-iteration flow dependence on the array *A* that prevents immediate DOALL parallelization. However, all of the computations in statement 2 could be completed independently of the calculations in statement 3 (statement 2 does not depend on statement 3). Statement 3, in fact, could wait and be executed after all calculations from statement 2 were executed. Recognizing that statement 3 depends on statement 2 but that statement 2 has no loop-carried dependences, this loop can be distributed.

```
1  for(i = 0; i < N; i++) {
2    A[i] = A[i] + C[i];
3    B[i] = A[i - 1] + 10;
4  }
```

Figure 2.10: Code example for loop distribution

```
1  for(i = 0; i < N; i++)
2    A[i] = A[i] + C[i];
3  for(i = 0; i < N; i++)
4    B[i] = A[i - 1] + 10;
```

Figure 2.11: Code for distributed loops from Example 2.10

The code generated by distributing this loop is shown in Figure 2.11. The statements in the first loop (statements 1 and 2) very closely resemble the first DOALL example from Section 2.1.2; it is absolutely a good candidate for the DOALL technique. The second loop now depends on the first loop but has no loop-carried dependences. It, too, may be further transformed by the DOALL technique. Both of these techniques are common techniques for enabling IMT and have been recorded in numerous publications and texts [4].

However, these transformations are only a small sample of the vast toolbox of transformations that enable further DOALL parallelism. Other techniques for creating parallelism for non-parallel loops include *loop reversal* and *loop skewing*. While the loop transformation techniques discussed here unlock parallelism in well-nested loops, imperfectly nested loops require other techniques to find parallelism. For this, it is appropriate to try *multilevel loop fusion* [37]. Many of these techniques have been combined into a single algorithmic approach to extracting parallelism from scientific code called the *affine transform*[17].

2.1.5 Speculation

While DOALL parallelizations can be applied to a wide range of loops without inter-iteration dependences and transformations can unlock parallelism where none previously existed, this still limits the programmer or compiler in instances where an inter-iteration

```
1 int a[N], b[N], c[N];
2 for(i=0;i<N;i++){
3   if (a[i] < 0 || b[i] < 0) break;
4     c[i] = a[i]+b[i];
5 }
```

Figure 2.12: Sequential code for DOALL speculation example

dependence *might* exist. It is common, for example, to insert error-checking code in an otherwise DOALL parallelizable loop, as in the code in Figure 2.12.

The above loop is almost exactly the same as the one mentioned in Section 2.1.2 except that there is one conditional branch for error-checking. This early-exit branch adds inter-iteration dependence edges that prevent DOALL parallelization because it cannot be determined whether the next iteration is going to be executed or not depending on the value of a[i] and b[i] in this iteration, though the early branch may never happen at runtime.

The solution is to speculate this rarely occurring branch [40]. To re-enable DOALL parallelization using speculation, dependences that should be speculated must be determined. In the case of this example, the control dependences created by statement 2 that make the best, most helpful candidates for speculation. Speculating that this exit will not occur, code must be inserted for detecting misspeculation in order to guarantee correctness. Misspeculation recovery functionality is usually needed to recover if misspeculation occurs. This misspeculation detection and recovery can be done either with hardware support, using something such as transactional memory, or in software, with software transaction memory. A compiler can transform the original loop into the one in Figure 2.13. All loop iterations can now be executed speculatively in parallel. DOALL parallelism is re-enabled.

Now, as before, more threads can be spawned to run the loop speculatively. If the early exit condition is never satisfied, the loop will execute in parallel with no extra cost. If there is an error in the input data and the loop does exit early, there must be some recovery mechanism. There is a trade-off between enabling more DOALL parallelism and the cost of misspeculation in real world applications. A compiler can decide when and where to

```
1  Thread 1:
2
3  misspec = 0;
4  for (i = 1; i< N/2; i++} {
5       if (a[i] < 0 || b[i] < 0) {
6            misspec = 1;
7            break;
8       }
9       c[i] = a[i] + b [i];
10 }
11 if (!misspec)
12      receive(thread2);
13
14 Thread 2:
15
16 for (i = N/2 ; i <N; i++) {
17       if (misspec == 1)
18            break;
19       if (a[i] < 0 || b[i] < 0)
20            break;
21       c[i] = a[i] + b[i];
22 }
23 send(thread1);
```

Figure 2.13: Parallelized code for DOALL speculation example from Example 2.12

speculate a value or branch to remove data or control dependences in the PDG at compile time.

In real world programs, such as scientific applications, vector calculations are very common and, in fact, often dominate. While many array and vector computations can be parallelized in a DOALL fashion, it is also common to have bounds or threshold-checking conditional branches as we saw in our simple example. As a practical example, in a support vector machine program, the goal is to compute a division plan that divides the data set best. The computation loop finishes when the result is good enough. That is, some certain threshold is reached. This rarely-taken early-exit branch inhibits a large amount of otherwise parallelizable code. With the speculative DOALL technique, the parallelism is restored with only one misspeculation.

2.2 Cyclic Multi-Threading

2.2.1 Introduction

While DOALL scales very well with the number of iterations in a loop, its applicability is limited by the existence of loop-carried dependences. DOACROSS parallelization is able to exploit parallelism in loops with cross iteration dependences. describe how speculation is employed in DOACROSS to enable more parallelism. Thread Level Speculation (TLS) will be introduced as a widely recognized example, and the TLS technique known as speculative parallel iteration chunk execution (Spice) will be discussed.

2.2.2 DOACROSS Parallelization

DOACROSS schedules each iteration on a processor with synchronization added to enforce the dependences. In Figure 2.14 there is a loop which contains loop-carried dependences. The corresponding program dependence graph was formed in Figure 2.15. If we assign

```
1  1:  while(node) {
2  2:    ncost = calc(node);
3  3:    cost += ncost;
4  4:    node = node->next; }
5  5:  ...
```

Figure 2.14: DOACROSS example code

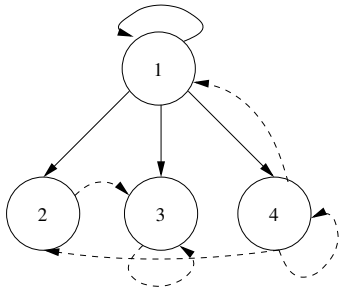


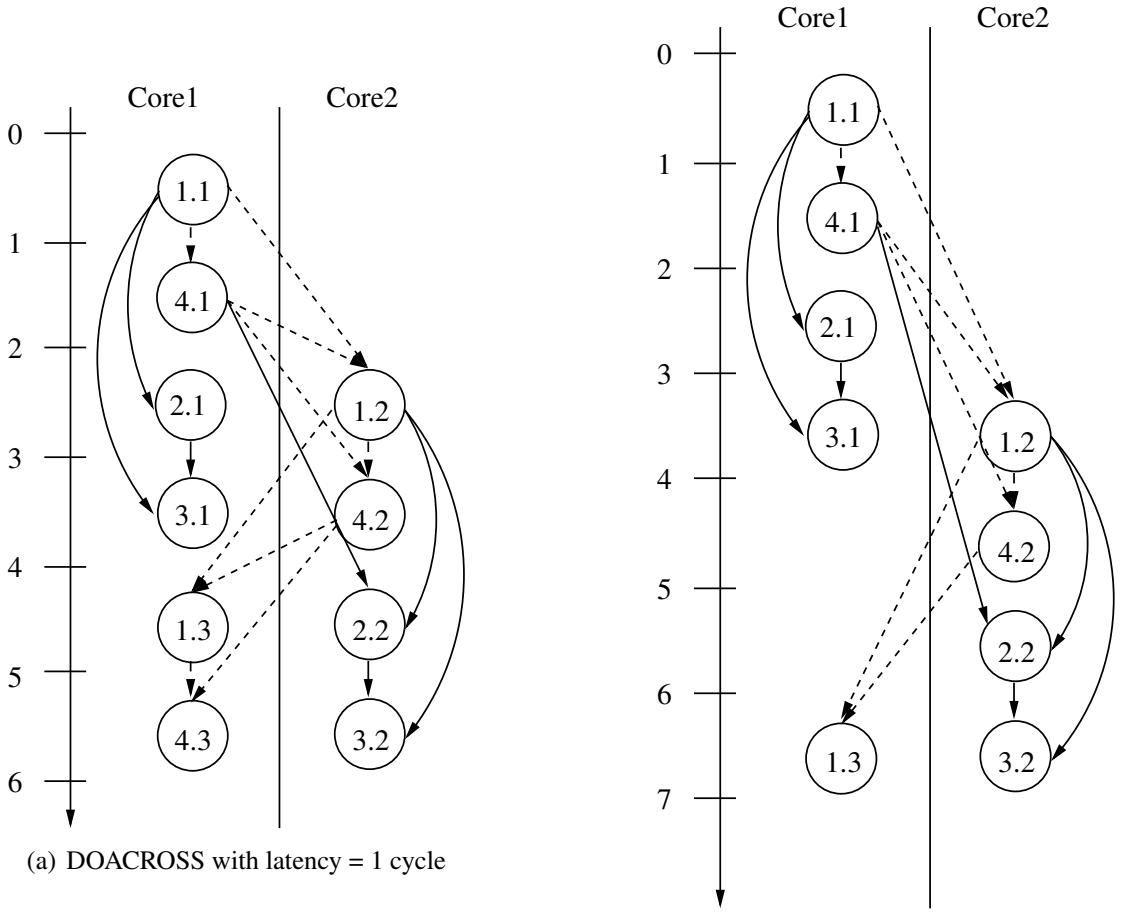
Figure 2.15: Program Dependence Graph for Example 2.14

different iterations of the loop onto different cores, synchronizations are communicated between different cores.

Figure 2.16(a) shows the DOACROSS execution model for this loop. It assumes an inter-core communication latency of only 1 cycle and thus achieves a speedup of 2 over single threaded execution. However, since synchronizations between different cores are put on the critical path, DOACORSS is quite sensitive to the communication latency. Increasing the communication latency to 2 cycles, for example, results in execution as shown in Figure 2.16(b), and it then takes 3 cycles instead of 2 to finish an iteration.

To do a DOACROSS transformation, the loop carried dependence graph is constructed first and then the synchronization instructions are inserted into the loop to enforce the dependences.

Different synchronization schemes have been proposed in previous work. Some work has proposed compiler algorithms that insert special synchronization instructions into DOACROSS loops [21]. Another example computes the access orders of each data element at compile time and then inserts data synchronization instructions to enforce those dependences [32]. Other research has taken aim at distributed shared-memory multiprocessors [31]. It uses di-



(a) DOACROSS with latency = 1 cycle

(b) DOACROSS with latency = 2 cycles

Figure 2.16: DOACROSS example

```
1 for (i=0; i<N; i++) {
2   A[i] = B[i-2] + C[i];
3   B[i] = A[i] + D[i];
4 }
```

Figure 2.17: Sample loop

```
1 doacross I = 1, N
2       test(R, 2)
3 1:  A[I] = B[I-2] + C[I];
4 2:  B[I] = A[I] + D[I];
5       testset(R)
6 end doacross
```

Figure 2.18: Loop after DOACROSS transformation

rect communication and a static message passing method. It proposes compiler algorithms to generate communication primitives for DOACROSS loops. The primitives use the nearest shared memory (NeSM) as communication buffers to eliminate major inter-processor communication overhead.

DOACROSS code generation depends on the synchronization scheme. Here, the algorithm proposed by Midkiff is used as example [21] in the loop shown in Figure 2.17.

The only cross-iteration dependence exists from statement 3 to statement 2. Before statement 2 in iteration i can execute, it needs to make sure that statement 3 in iteration i-2 has already finished execution.

To enforce the dependence, synchronization instructions will be inserted into the loop. In this example, test and testset are used. The call test(R,2) makes sure that before statement 1 can execute in the present iteration i, iteration i-2 has already finished executing because only after that will the value R be set to the iteration number by instruction testset(R). Now different iterations now can be run in parallel on different cores. This transformed loop can be seen in Figure 2.18

2.2.3 Thread Level Speculation

The algorithms introduced above depend on the accurate detection of dependence distances across iterations. However, for more general purpose programs, dependences can be ambiguous or not manifest in all cases. To reduce the communication latency between processors, speculation is necessary. Profiling as is described later in this thesis is often used to determine if speculation will be profitable. Without profiling, poor speculation decisions can easily be made whether it is by a programmer or by the compiler. Additionally, as will be seen in thread level speculation techniques, the parallelized code must often know to speculate *all* loop-carried dependences, and this is something a profiler that is loop-aware can help a programmer or researcher determine much more rapidly than they could do by hand.

Typically, in a DOACROSS style parallelization, transformed code will speculatively execute iterations in parallel as if there are no loop-carried dependences. This is called thread level speculation (TLS). There are two most common types of speculation:

- Memory alias speculation: This assumes that loads in later iterations do not conflict with stores in the earlier iterations. If the speculation turns out to be false due to dependences between the iterations, the speculatively executed iterations are squashed and restarted. The alias speculation works as long as the conflict between loads and stores in different iterations are infrequent. If the dependences between loop iterations manifest frequently, alias speculation suffers from high mis-speculation rates, causing a slowdown when compared with single-threaded execution. This can be overcome by synchronizing those store-load pairs that conflict frequently.

- Value Prediction: This uses value predictors to predict the live-ins for future iterations and speculatively executes the future iterations with these predicted values. Different value predictor techniques, such as last value predictors, stride predictors, and trace-based predictors have been proposed. Some common uses of value specu-

```
1 while(node) {
2   ncost = calc(node);
3   cost += ncost;
4   if (cost > T)
5     break;
6   node = node->next;
7 }
```

Figure 2.19: Sequential code for TLS example

lation are biased branch speculation (which speculates that a condition for a branch is unlikely to be met) load prediction (which predicts what value will be loaded), and silent store speculation (which speculates that the store will not actually change the existing value).

A third type of speculation called memory value speculation combines elements of both of the above speculation types by speculating upon the value read from memory at a certain point, without specific regard to the memory location from which it is read.

Architectural support or substantial software support is needed for mis-speculation detection and recovery. First, whether a store and a load conflict during execution must be detected. Second, if mis-speculation happens, any changes to the state made by the speculative thread must be undone. Undoing changes to register state requires saving and restoring some register values which can be done in software. Undoing changes to memory requires special support such as hardware transactional memory or other memory systems that buffer speculative state and discard it on mis-speculation.

The example code in Figure 2.19 will be seen to be parallelizable using the TLS technique.

According to the PDG in Figure 2.20 for this example code, there are two loop-carried dependences. The first one is in statement 4 which decides whether the loop should exit or not. The second one is in statement 6 which provides the node value for the next iteration. As for the dependence caused by statement 3, since cost is a reduction variable, we can use the technique mentioned in last section to resolve the dependence.

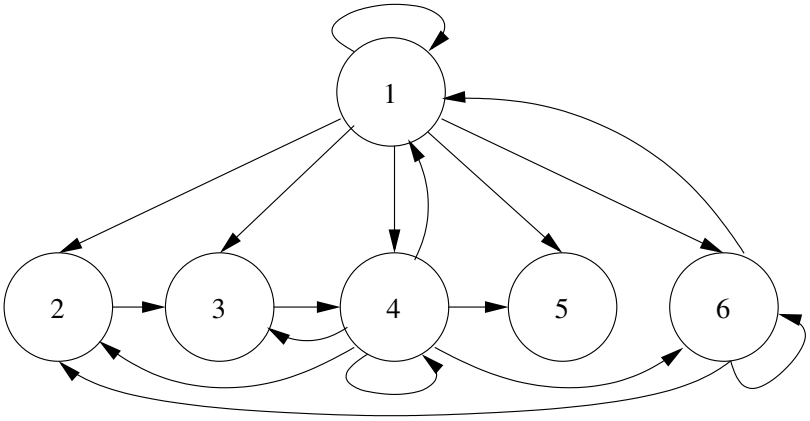


Figure 2.20: PDG for example loop in Example 2.19

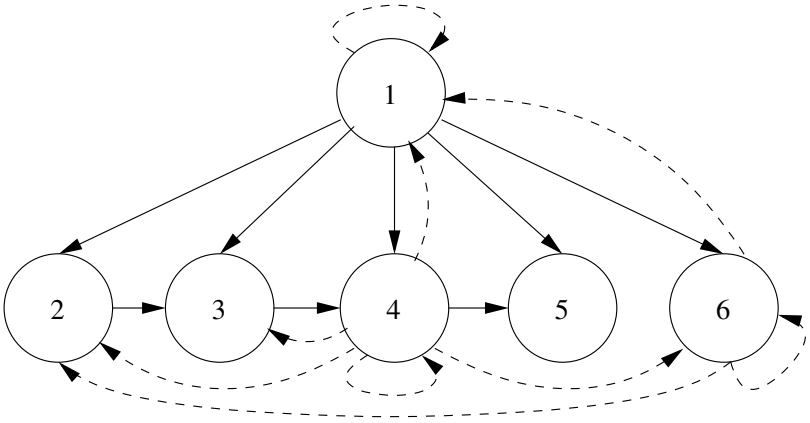


Figure 2.21: PDG for code in Example 2.19 after speculation

In order to delete all of the loop-carried dependences, we need to speculate:

1. The value of cost is always less or equal to the value of T, which means the branch in statement 4 is not taken.

2. The values of nodes are the same in different invocations of the loop, which means we can use the node value in former invocations to predict the node value in the present invocation.

Figure 2.21 shows the PDG after speculation. Now if we have two threads, we will transform the example code into the code seen in Figure 2.22.

Examining this code, a question remains as to how to find the "predicted node" used by thread 2 as a starting position. One possible answer is a technique known as *speculative*

```
1  Thread1:
2
3  mispred = 1;
4  while(node) {
5    ncost = calc(node);
6    cost += ncost;
7    if (cost > T)
8      break;
9    node = node->next;
10   if (node == predicted_node) {
11     mispred = 0;
12     break;
13   }
14 }
15 if (!mispred) {
16   receive(thread2, cost2);
17   cost += cost2;
18 }
19
20 Thread2:
21
22 node = predicted_node;
23 cost = 0;
24 while(node) {
25   ncost = calc(node);
26   cost += ncost;
27   if (cost > T)
28     break;
29   node = node->next;
30 }
31 send(thread1, cost);
```

Figure 2.22: Parallel code after TLS for Example 2.19

parallel iteration chunk execution (Spice) [28]. Spice determines which loop-carried live-ins require value prediction. It then inserts code to gather values to be predicted. On the first invocation of the loop, it collects the values – in this case, it is a pointer to a location somewhere in the middle of our list. Since many loops in practice are invoked many times, the next invocation can then speculate that the same value will be used. So long as the predicted node was not removed from the list (even if nodes around it were removed), our speculation will succeed.

Since this example has only two threads, only one node value needs to be predicted. Thread 1 is executed non-speculatively. It keeps executing until the node value is equal to the predicted node value for thread 2. If that happens, it means there is no mis-speculation, and the result from thread 2 can be combined with the result in thread 1. However, if the node value in thread 1 is never equal to the predicted value, thread 1 will execute all of the iterations and a mis-speculation will be detected in the end. When mis-speculation happens, all the writes to registers and memory made by thread 2 must be undone and the result of thread 2 is discarded.

There has been a great deal of work done with thread level speculation techniques beyond what has been mentioned in this section. Some efforts have investigated hardware support and design for TLS techniques as with Stanford's Hydra Core Multiprocessor (CMP) [13], among many others[1] [20] [36] [30] [33]. Other TLS papers have investigated software support for these techniques [22][9]. Bhowmik and Franklin investigate a general compiler framework for TLS techniques [6]. Other work by Kim and Yeung examines compiler algorithms [15]. Work by Johnson, Eigenmann, and Vijaykumar examines program decomposition [14]. Further work by Zilles and Sohi examines a TLS technique involving master/slave speculation [41]. The interested reader is directed to these publications for more information about the numerous TLS techniques and support systems.

2.3 Pipeline Multi-Threading

2.3.1 Introduction

A pipeline is a chain of processing elements, instructions in this case, arranged in a way that each earlier element produces the input that will be used by a later element. Sometimes buffering is needed to communicate between two elements. The first pipeline parallelization technique proposed was DOPIPE. It was initially proposed alongside DOACROSS to parallelize scientific code with recurrences[25]. DOPIPE splits the original loop into several stages and spreads them among multiple threads. The number of threads are fixed at compile time. The dependences among all threads are forced to be unidirectional, which means no cyclic cross-thread dependences.

2.3.2 Decoupled Software Pipelining

Decoupled software pipelining (DSWP) [24] is a technique proposed more recently that partitions the code into several stages and executes them in a pipelined fashion. Like DOPIPE, communication is restricted to be unidirectional – from an earlier stage to a later stage in the pipeline only. DSWP differs from DOPIPE in that 1) DSWP targets general purpose programs instead of scientific code; 2) DSWP uses communication queues for inter-thread communication. Figure 2.23(b) shows the execution model of DSWP.

All types of pipeline parallelizations rely on the loop containing pipelineable stages. That is, DSWP works only if the loop can be split into several stages that do not form cyclic dependences in the PDG. The example code shown in Section 2.2.2 will not be a good candidate for DSWP since the two instructions contained in the loop form a cyclic data dependence. Therefore DSWP does have the same universal applicability as DOACROSS. However, DSWP is not as sensitive to communication cost as DOACROSS because the execution of the next iteration overlaps with communication. Communication latency only affects performance as a one-time cost, no longer being incurred once the pipeline is filled.

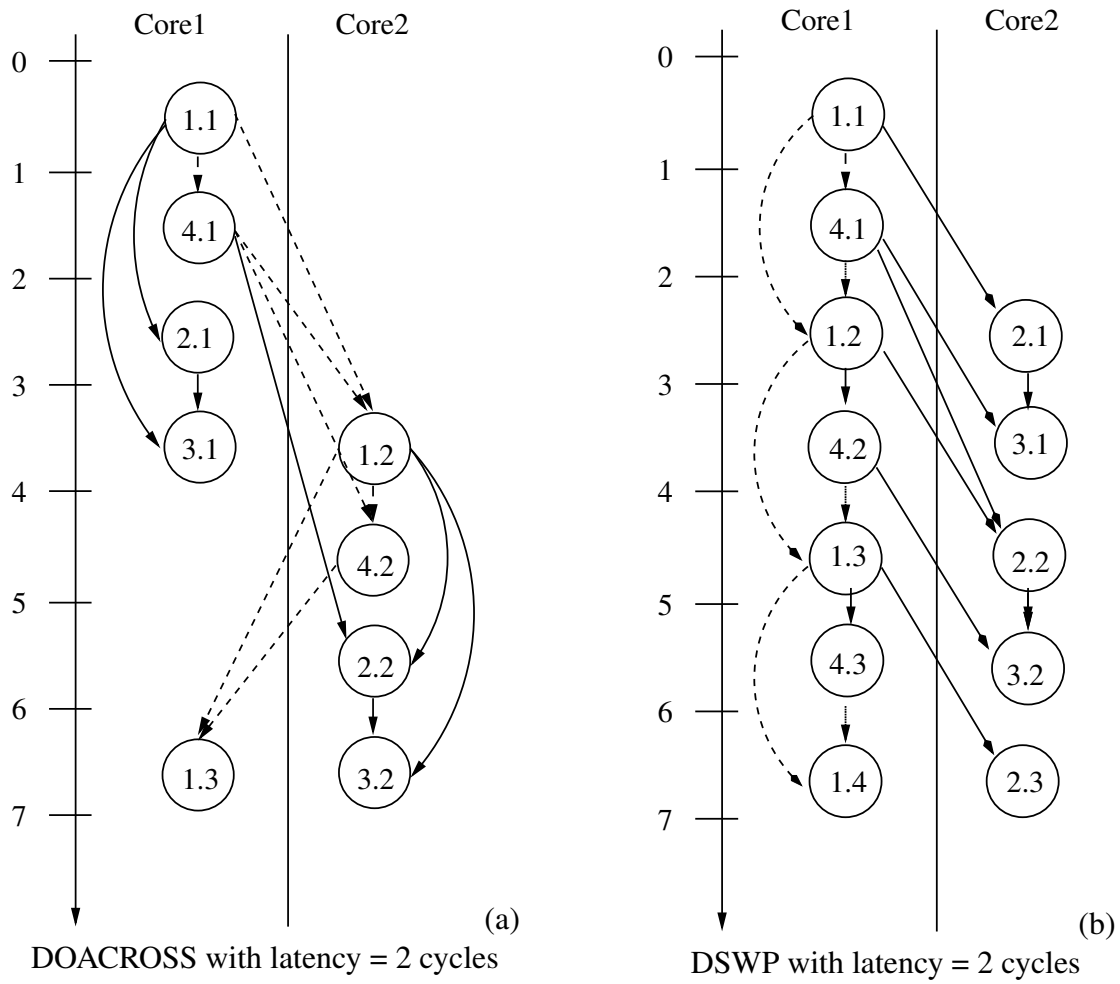
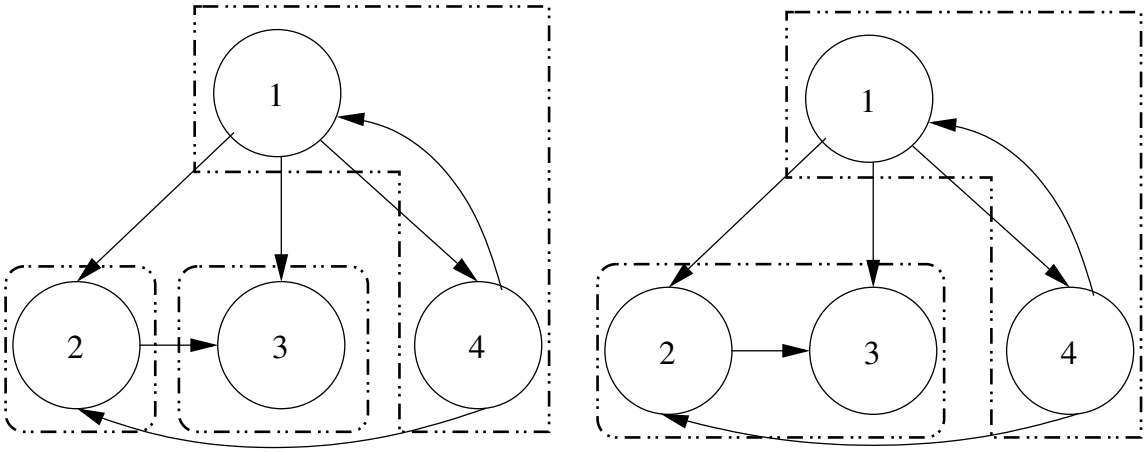


Figure 2.23: DSWP vs. DOACROSS with communication latency

Therefore, pipeline parallelization is more latency tolerant than DOACROSS technique. In Figure 2.16, DOACROSS performs poorly with longer latency whereas DSWP allows overlapping of communication with execution of the same stage in the next iteration to hide the latency as in Figure 2.23.

The linked list traversal loop example in Figure 2.14 and its PDG in Figure 2.15 from the previous section (Section 2.2) is useful for illustrating DSWP. For purposes of further discussion, self-dependence edges are omitted – each statement in a partition will be entirely within a single partition, so these edges will inevitably be respected.

To split the loop, DSWP creates an acyclic thread dependence graph based on PDG. It first identifies strongly connected components (SCCs) in the PDG. Contracting each of



(a) DAG_{SCC} of sample loop (b) DAG_{SCC} Partitioned

Figure 2.24: DSWP code partitioning for Example 2.14

the SCCs to a single vertex, the resultant DAG_{SCC} graph is seen in Figure 2.24(a). It is this graph that is partitioned into a fixed number of threads without cyclic inter-thread dependences as in Figure 2.24(b). In this example, instructions 1 and 4 form an SCC that cannot be split without creating a cyclic inter-thread dependence. They are placed on the same stage in the pipeline. Since many possible partitions exist, heuristics are commonly used to find the best load balance while minimizing communication costs[23].

Given this code partition, the next step is to generate the parallel code for the parallel threads to execute. This example DSWP parallelization of this loop splits the loop into two slices: one linked list traversal thread and one computation thread, as shown in Figures 2.25 (a) and (b) respectively. The produce function enqueues the pointer into a communication queue, and the consume function dequeues one element (a pointer) from the queue. In this way, dependences are communicated forward in the pipeline; the isolation of thread-local variables in conjunction with this forward communication of intra-iteration dependences allows intra-iteration anti-dependences to be ignored. If the queue is full when produce function is called, the produce will be blocked and will wait for an empty slot; if the queue is empty when consume function is called, that consume thread will be blocked to wait on more data.

This DSWP parallelization decouples the execution of the two code slices and allows

```
1   while(node) {
2         produce (node, queue);
3         node = node->next;
4   }
```

(a) produce thread

```
1  while ((node =consume(queue)) {
2    ncost = cacl(node);
3    cost += ncost;
4  }
```

(b) consume thread

Figure 2.25: DSWP partitioned code

the code execution to overlap with the inter-thread communication. Furthermore, the loop is split into smaller slices so that the program can take better use of cache locality.

2.3.3 Speculative Decoupled Software Pipelining

In the case of DOACROSS and DOALL parallelization, applicability was limited considerably by inter-iteration dependences and MAY ALIAS relations that could not be disproved at compile time. Through speculation as seen in Section 2.2.3, the applicability of DOACROSS parallelization is greatly improved in instances where cost of misspeculation detection and misspeculation recovery time can be kept relatively low and while frequency of misspeculation remains low.

The case for speculation in pipeline parallelization is similar. One typical case is that of a conditional branch for an error condition. Such a branch may be rarely taken, and may in fact never manifest in normal execution, but it can introduce a dependence recurrence. Consider the loop in Figure 2.26, a loop that is exactly the same as the loop examined in previous sections on pipeline parallelism except that it introduces an early exit condition if the computed cost exceeds a threshold.

```
1  while(node) {
2    ncost = calc(node);
3    cost += ncost;
4    if (cost > T)
5      break;
6    node = node->next;
7  }
```

Figure 2.26: Pipeline speculation code example

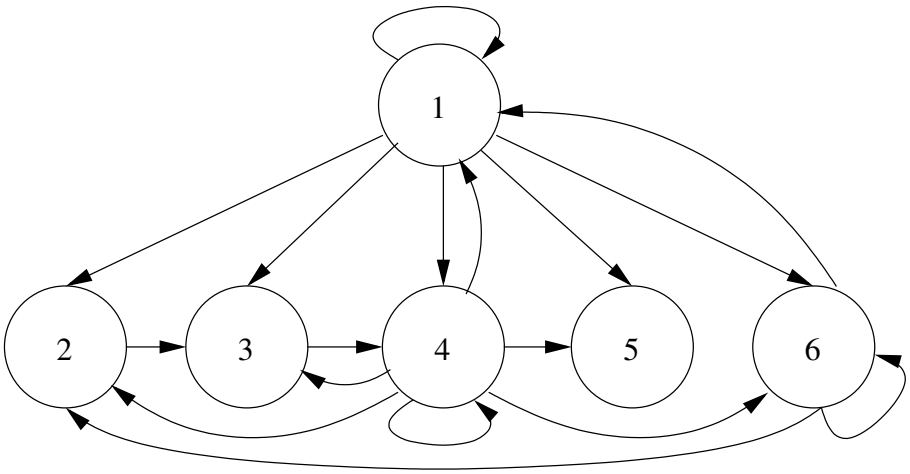


Figure 2.27: PDG for Example 2.26

This section will reference the code in Figure 2.26 as an example. The PDG formed by this code is shown in Figure 2.27 and is clearly not partitionable by DSWP as described previously. The dependence recurrence created by the introduction of the early exit condition must be broken in order to achieve a DSWP style pipeline parallelization. One must recognize that this is a recurrence that would be useful to break, however.

The most common speculative pipelining technique is known as Spec-DSWP[35]. In this technique, determining what to speculate is decided in a way similar to the way it was answered in Section 2.2.3. Possible types of speculation are the same as those seen previously. As review, these can include:

1. Value speculation - using a predicted value instead of actual value; includes biased branch speculation, load prediction, and silent store

2. Memory alias speculation - speculating that two memory operations, one of which is a write, will not access the same memory location

3. Memory value speculation - combining elements of both of the above and speculating that a value read from memory will be the same regardless of the memory location from which it is read

There are, however, important differences that make profiling even more important in

Spec-DSWP than it was in TLS. The most integral difference lies in the fact that TLS techniques generally speculate only on loop-carried dependences as their goal is to reduce inter-thread communication. They need only discover what dependences are loop-carried. Meanwhile, pipeline parallelization, which does not focus on reducing inter-thread latency, instead looks to remove dependence recurrences. Speculating such recurrences breaks large SCCs and reduces the size of pipeline stages, a desirable result for pipeline parallelism. This difference of focus allows speculation on *any* dependence that will break an SCC. Since loop-carried dependences may not be easily speculatable while another dependence along the cycle may be, the compiler has increased flexibility when selecting which edges to speculate. For a researcher or programmer investigating possible parallelizations by hand, this means much more work.

The fact that speculation in pipeline parallelizations can be intra-iteration speculation has other implications. Unlike DOACROSS speculation techniques, no iteration executes non-speculatively. The resultant work is committed when all stages of the pipeline have completed without misspeculation. The iterations must be guaranteed to commit in proper order.

Though it is now clear that it is desirable to select dependence edges to speculate which will remove a recurrence with the least likelihood of causing excessive misspeculation, the question of how to select these dependences remains. Limiting the likelihood of excessive misspeculation can be done using profiling. Loop-sensitive profiling provides even better information to a compiler looking to limit misspeculation. Representative loop-sensitive profiling information can indicate rarely taken control edges, possible memory aliases that rarely or never manifest, and operations that frequently return the same result. The same sort of profiling information that is indispensable to a compiler can also be used to greatly ease the efforts of the programmer or researcher investigating techniques.

In our example in Figure 2.26, it is assumed that we know through profiling that the early exit for cost greater than a threshold (cost $>$ T) is rarely taken and thus a good candi-

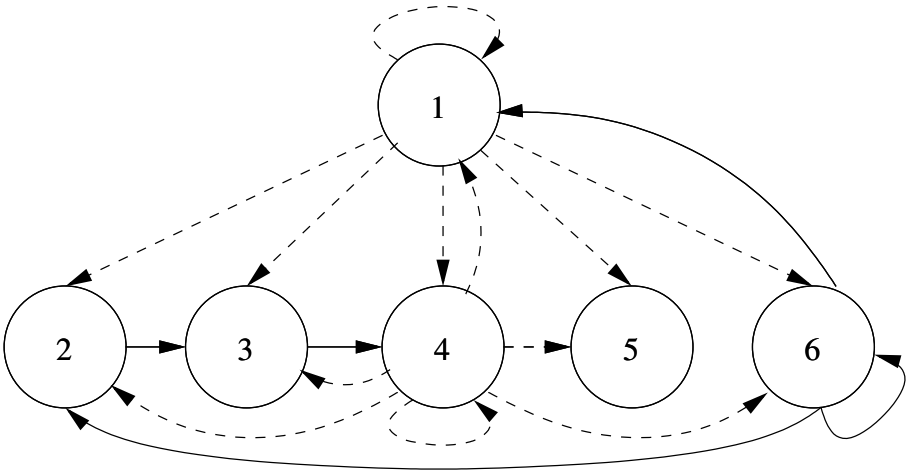


Figure 2.28: Prospectively speculated dependences for Example 2.27

date for speculation. We will also assume that the exit due to a null node is also rarely taken (profiling reveals we have long lists) and thus a possible candidate for speculation as well. Figure 2.28 shows dashed lines representing the candidates for speculated dependences.

The second issue in selecting dependences is actually choosing dependences that break a recurrence. It may be, and in fact often is, the case that it is necessary to speculate more than one dependence in order to break a recurrence. Ideally, one would want to consider speculating all sets of dependences. Unfortunately, exponentially many dependence sets exist and so a heuristic solution is necessary. In Spec-DSWP, this is handled in a multi-step manner. After determining which edges are easily speculatable, all of these edges are prospectively removed. The PDG is then partitioned in the same manner as in DSWP with these edges removed. Edges that have been prospectively removed and would break an inter-thread recurrence that originates later in the pipeline become speculated. Other edges that do not fit these criteria are effectively returned to the PDG. That is, the dependences are respected and not actually speculated.

In the example seen in Figure 2.29, it is seen that the DSWP partitioning algorithm has elected to place statements 1 and 6 on a thread, statement 2 on a thread, and statements 3, 4, and 5 on a thread. Using this partitioning, it is clear that speculating the dependences on the early exit due to cost exceeding a threshold (outbound edges from 4) is necessary.

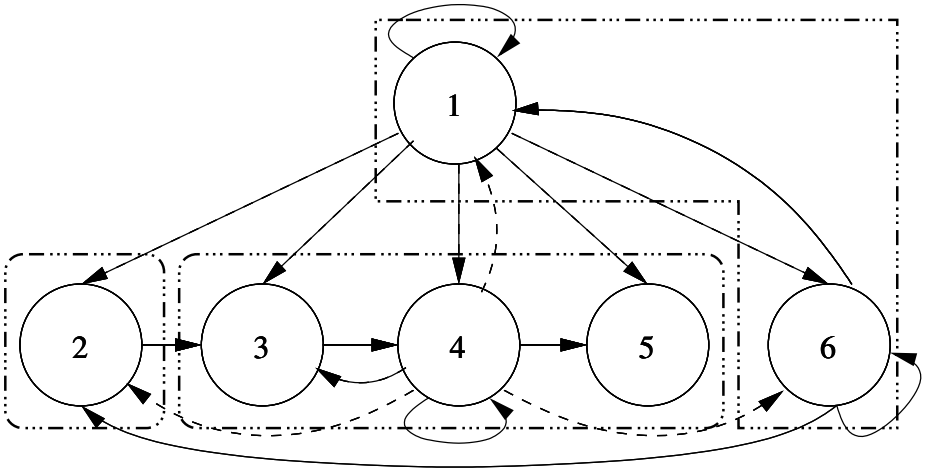


Figure 2.29: Partitioned PDG for Example 2.26

However, edges speculated due to a null node do not break any inter-thread recurrences. These dependences, therefore, will not be speculated.

Though the previous paragraphs describe what is necessary to select dependences to speculate, checking for misspeculation is just as important. This is partly a problem of code generation and partly a problem of support systems. Unlike speculation in DOACROSS techniques (i.e. TLS), every iteration is speculative and no single thread executes a whole iteration. This means that there must be some commit unit to guarantee a proper order of commit and handle recovery. Furthermore, the version memory system must allow multiple threads to execute inside the same version at one time. The system must check for proper speculation across the entire version before it can be committed. These checks for misspeculation may either be inserted into the code by the compiler or handled in a hardware memory system. Spec-DSWP handles most speculation types by inserting checks to flag misspeculation. The noteworthy exception is the case of memory alias speculation where the memory system is left to detect cases where a memory alias did exist.

In this example, code has been inserted to flag misspeculation if it occurs in the final thread. Three threads will be generated from the partitioned PDG in the same way as DSWP. The code generated is shown in Figure 2.30.

In all cases, the final requirement is a method by which speculative state can be buffered

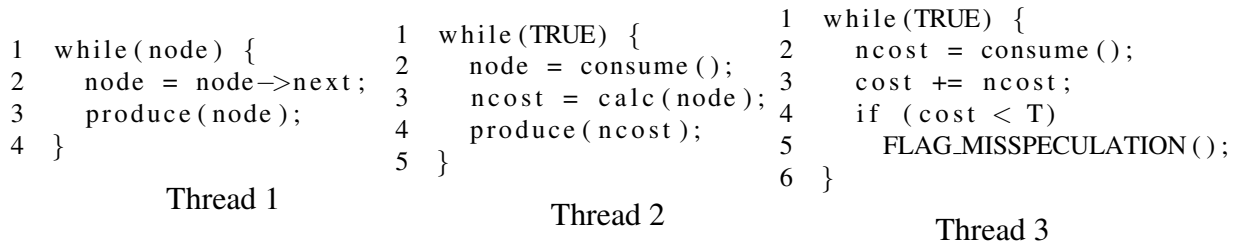


Figure 2.30: Parallelized code from Example 2.26

prior to commit and a unit to commit non-speculative state. When misspeculation is signaled, the non-speculative state must be recovered and the work must be recomputed. Version memory systems that allow multiple threads to operate on a given version and commit only when the version has been completed and well-speculated have been proposed in hardware and in software but will not be discussed here [34].

The considerations made in this section can be summarized as a step-by-step procedure:

1. Build the PDG for the loop to be parallelized
2. Select the dependence edges to speculate
3. Remove the selected edges from the PDG
4. Apply the DSWP transformation to the PDG without speculated edges
5. Insert code necessary to detect misspeculation for dependences to be speculated
6. Insert code for recovery from misspeculation

2.4 Parallel Stage Decoupled Software Pipelining

Spec-PS-DSWP in particular draws from the pipelining execution model of DSWP with the use of speculation to break recurrences seen in Spec-DSWP and combines them with an idea drawn that partially mirrors DOALL by replicating stages that could be executed independently. The result is a highly applicable technique that is latency-tolerant and highly scalable. As an example, take the code in Figure 2.31.

```
1  while(node) {
2    p = node -> list;
3    while(p) {
4      if (p->val == 0) exit();
5      p->val++;
6      p = p->next; }
7    node = node -> next;
8  }
```

Figure 2.31: Code example for Spec-PS-DSWP

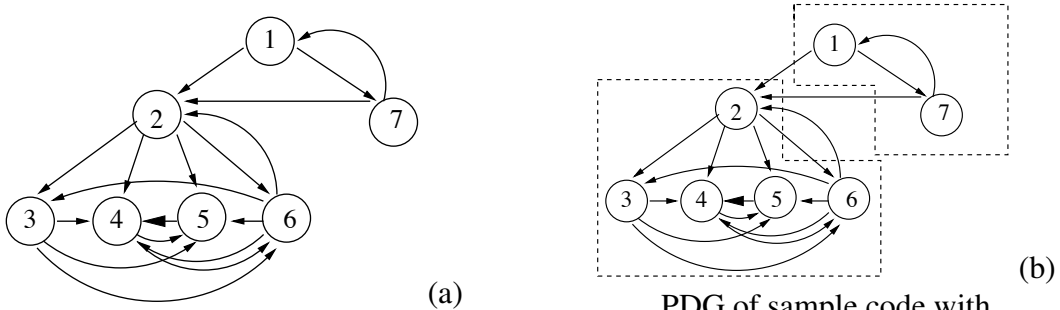


Figure 2.32: PDG of sample code from Example 2.31 before and after partitioning

Examining this code and its PDG shown in Figure 2.32(a), there is the familiar list traversal from which we were able to form our DSWP pipeline. There is also an inner loop control dependence from the early loop exit condition that causes a dependence recurrence. With this control dependence removed via speculation, a two-stage pipeline could be formed as previously. This can be seen in the partitioned PDG in Figure 2.32(b).

However, the second stage is likely to be much longer than the first stage – it contains a whole linked-list traversal of its own. This imbalance greatly damages the gains from a Spec-DSWP parallelization. However, each invocation of the inner loop (lines 3-6) is independent of one another, if the memory analysis confirms that each invocation of the inner loop accesses different linked list (if memory analysis could not determine it, profiling and speculation would again be indispensable). This stage could be replicated and all invocations of this loop could be executed in parallel. This *parallel stage* can be executed in parallel on many cores and can be fed with the node found in the first *sequential stage*

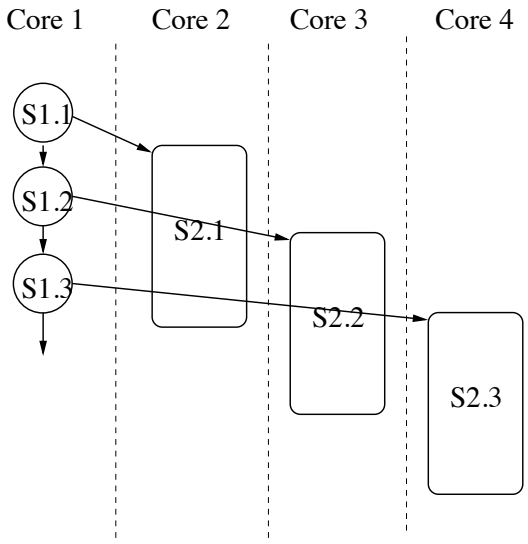


Figure 2.33: Execution model of Spec-PS-DSWP

of the pipeline, as shown in Figure 2.33. The parallel stages need not communicate with each other, similar to the way that DOALL threads need not communicate with each other. Their only common link is the first DSWP pipeline stage. Because the parallel stage may be replicated many times, Spec-PS-DSWP is a highly scalable technique, achieving better speedup as the number of cores increases.

2.5 Profiling

Profiling for memory dependence information has been repeatedly mentioned throughout this background chapter as a necessary component for intelligent speculation. Profiling, as described before, involves instrumenting code, either offline prior to runtime or online, on-the-fly prior to actual execution, in order to collect information about loads and stores that alias at runtime. Finding parallelism is inherently hard and though profiling has been described so far as a tool for automatic techniques, when properly presented, it is equally as useful to a programmer or researcher trying to analyze sequential programs for potential parallelism opportunities. Without reasonable profiling information, the programmer must take great pains to determine all dependences, including dependences that do not mani-

fest at runtime. While this may be trivial in a simple program, analyzing a program with deep call graphs and potential memory aliasing is much more difficult. Furthermore, if the dependences can be greatly impacted by the input set, it may be difficult to determine which dependences will manifest in common cases. Profiling may show that certain dependences a programmer expects to manifest never do or, conversely, may show that some dependences the programmer did not expect manifest frequently.

Much work has been put into using profiling for memory dependence analysis. As early as 1995, ATOM was introduced to provide a system for instrumenting individual instructions of a binary for various purposes, including dependence analysis and data trace generation[12]. Individual instruction instrumentation is useful for generating detailed memory dependence information, but ATOM was not targeting parallel programming. DynamoRIO, like ATOM, is an infrastructure for instrumenting binaries rather than source. DynamoRIO, however performs online instrumentation, using dynamic runtime code manipulation to instrument and profile a running binary[39]. While it has been used more extensively for profiling applications, DynamoRIO was also not primarily concerned with aiding in parallelization.

More recently, many research compilers have integrated profiling tools with the more explicit purpose of using profile information for finding parallelism. TEST is a hardware-based profiler that finds minimum dependence distances between loop iterations and uses it to slice Java programs into speculative threads in a TLS scheme[8]. POSH is similar to TEST in that it also targets a TLS scheme but it further profiles many constructs including loop iterations and subroutines and continuations of these constructs[18]. LAMP is the profiling infrastructure component developed to aid speculation in the VELOCITY compiler which was capable of various parallelization transformations based on such input[27][7]. As it forms the basis of this thesis, it is discussed in great detail in Chapter 3. TEST, POSH, and LAMP all focus on profiling true (RAW) dependences and omits WAW and WAR dependences. WAR and WAW dependences are often omitted from profilers because they

require much larger amounts of data to be maintained in order to track all possible cases which can become prohibitive.

The three profiling tools mentioned above are meant only for purposes of a compiler performing automatic parallelization. Other work interprets profiling data for the purposes of a programmer or a researcher interested in analyzing a program for potential parallelism opportunities in a manner more similar to the work presented in this thesis. ParaMeter provides an interactive analysis and visualization system for large traces in order to facilitate identification of parallelism[26]. It provides organized information for massive traces and uses BDD to provide fast trace compression. The LAMP utility detailed in this work does not deal with full traces but rather with organized dependence information from the code instrumented with the LAMP libraries.

In work most closely related to this thesis, Alchemist provides detailed profile analysis for the purposes of parallelism identification[38]. Alchemist provides instrumentation and profile data for all types of dependences for C programs. It bases its analysis on the dependence distance, reporting possible "violations" for loop-carried dependences that are within a certain dependence distance. Dependences beyond the set distance are presumed to be safe as the data will already be available and are not reported. While the Alchemist system does very well in its presentation of data and its representation of all dependence types, the system is unable to produce the correct result in the presence of recursion. Alchemist does not report intra-iteration dependences in any fashion. The LAMP-based utility described in the following chapter, while it does not provide the same information with regards to all dependence types, does provide dependence distance information and produces the correct result in the presence of recursion.

Chapter 3

Loop-Aware Profiling and Analysis Utility

3.1 Introduction

As discussed in the previous chapter, dependences limit the extractable parallelism. In addition to analysis that provides static information about possible dependences, profiling tools can give information about which dependences manifest. Whether it is an automatic technique, a researcher investigating new techniques, or a programmer attempting to parallelize a program entirely by hand, dependence information is crucial to a correct parallelization. In the case of the automatic techniques, the compiler needs information in a form it can easily manage and manipulate while a human-readable form is necessary in the other two cases.

This chapter introduces a complete toolset for profiling, processing, and viewing dependence information to aid in parallelization. The tool, titled LAMPView, is formed from three parts, each discussed in a separate section of this chapter.

Section 3.2 discusses the loop-aware memory profiling tool titled "LAMP" which instruments a program to generate loop-aware memory profiling, originally developed in

work by Raman for the VELOCITY Compiler and ported to the LLVM compiler infrastructure as a part of this work [7][27][16]. Section 3.3 explains a compiler pass developed in the LLVM compiler infrastructure to read LAMP profile data and translate it to user-readable source-level information. Section 3.4 describes a stand-alone utility for a programmer or researcher to retrieve the processed information which is usually not easy enough to read as to be useful. The utility makes dependence information searchable by various parameters, allowing a programmer to find true dependences that manifest regardless of call depth or aliasing that would otherwise make efforts in manual parallelization even more difficult than usual. It also provides means to output all loop-carried dependences with respect to a given loop.

Throughout this chapter, the C code shown in Figure 3.1 will be used to demonstrate the various elements of these profiling tools. The code's purpose is trivial: it consists of three functions, *Fibonacci* which computes the nth Fibonacci number based on a global variable *n*, *fact_g* which computes n factorial based on the same global variable, and *fact* which computes n factorial based on a passed parameter *z*. These functions are called in similarly trivial loops in a main function. While the program is simple, the dependences that manifest serve to demonstrate each element of the LAMPView toolset.

3.2 Loop-Aware Memory Profiling

The original loop-aware memory profiling tool (LAMP) was developed as a component of work with speculation in the VELOCITY Compiler [7]. LAMP acts in one sense like all other profilers in that it provides runtime dependence information between store/load pairs. However, LAMP provides further information on dependences than a count of occurrences of the store/load alias as most profilers do. LAMP profile data tracks and returns other information useful in parallelization including loop header information and dependence distance. The profile is interpreted with respect to the loop whose header identification

```
1  #include <stdio.h>
2
3  int n;
4
5  long int Fib()
6  {
7          int i=0;
8          long int f1=1, f2=1, fnext;
9          for (i=0; i < n; i++)
10         {
11           fnext = f1 + f2;
12           f1 = f2;
13           f2 = fnext;
14         }
15         return f2;
16 }
17
18 long int fact_g()
19 {
20         int i=0;
21         int bars=1;
22
23         for (i=n-1; i > 0; i--)
24         {
25            bars = bars*i;
26         }
27
28         bars = bars*n;
29         return bars;
30 }
31
32 long int fact(int * z)
33 {
34         int i=0;
35         int bars=1;
36         for (i=*z; i > 1; i--)
37         {
38            bars = bars*i;
39         }
40         return bars;
41 }
42
43 int main()
44 {
45         long int fi = 0, fa = 1, i=0;
46
47         do {
48            for (i=1; i < 9; i++)
49            {
50               n = i;
51               fi = Fibonacci();
52               n = i%2 ? i%9 : i%8;  // 1,2,3...7,0
53               fa = fact_g();
54               printf("%d %ld %d %d\n", i, fi, fa, n);
55            }
56            for (i=1; i < 3; i++)
57            {
58               fa = fact(&n);
59               printf("%d %ld %d\n", i, fi, fa);
60            }
61         }
62         while (fa != 1);  // actually iterates only once
63
64         return 0;
65
66 }
```

Figure 3.1: LAMPView example code

number is given, indicating that a load reads a value written by a store executed distance iterations ago a given number of times. The maximum dependence distance of interest can be altered, thus dropping any dependences that exceed that distance. The profiling execution runs to collect this information are run on sequential programs; the LAMP libraries are not currently thread-safe.

The implementation of this profiler is done in two parts. The first is an instrumentation pass written as a part of the compiler. Originally written as a part of the VELOCITY Compiler in work by Raman, part of this work ported it to the LLVM infrastructure so that it might be used with the full range of frontends and backends provided under the LLVM infrastructure. In this instrumentation pass, unique identifier numbers are assigned to all memory operations including loads, stores, and library calls. In particular, memory allocation and deallocation calls are instrumented. Loops are also given unique identifier numbers for use as the header identifier described above. These numbers, generated in sequence as instructions and loops are encountered, are used as in the calls to the LAMP library functions that instrument each memory operation. Memory operation instrumentation takes the identifier, the address accessed, and the size. Loop preheaders and loop exit blocks are also instrumented with calls to library functions indicating the start and end of loop invocations. Loops are also instrumented with calls indicating the beginning and ends of iterations. Finally, initialization LAMP API calls with the total number of instructions and loops are added at the start of the program to configure LAMP, and a finalization LAMP API call is added at the end of the program to trigger LAMP to write the profile results.

After discussion with potential users of the system, it was decided that mapping between LAMP loop identification numbers and the source line number where the loop began would be of great use in parallelizing loops within a program. As a result, in addition to doing the instrumentation, the pass also optionally will output the loop id, loop line number pairs with function name if the bytecode has been instrumented with debug information.

(Format: loop_id line_number function_name)

The LAMP library itself is almost unchanged in this work but Raman's work on the LAMP API is mentioned here for completeness[27]. LAMP maintains a **timestamp** that increments whenever a call to the library is made. It also maintains a **loop nest stack** on which loop header ids are pushed whenever a new loop is invoked, thus maintaining information about the nesting of loops. Information is also maintained about the current **loop iteration** in a queue structure for each loop; this queue can be simplified in cases where only loop-carried versus intra-iteration information is needed. A **memory writer map** is used to map between locations written and the id of the instruction that wrote that location. This structure is integral to identifying the most recent instruction to have written a location when a load is encountered. Finally, the full profile information is maintained in the **LAMP arcs map**. In this map is maintained the dependence information discussed in the first paragraph of this section (count, loop header, dependence distance) and is updated whenever the load instrumentation call is made. When such an update occurs, the inner-most loop that encloses both the load and the store is found from the loop nest stack, and the distance is determined from the loop iteration queue. At completion of the program, the LAMP arcs map is dumped to a file with all dependence information to be read by a second compiler pass for use in annotation or other purposes.

After some discussion over possible usefulness of the system as a whole with potential users, it was determined that augmenting the LAMP profile libraries with counters to track the number of iterations each loop iterated would be helpful. While the original VELOCITY compiler determined relative frequencies of manifesting dependences in other ways, this capability was no longer present in LLVM. This relative frequency is particularly useful when a programmer or compiler is trying to determine possible candidates for speculation as described in the background sections on speculation (Sections 2.1.5,2.2.3, and 2.3.3). As a result, a **loop iteration count** vector was added to the LAMP infrastructure. Without altering the LAMP API, it tracks iteration counts for the invoked loop using the loop nest

stack to determine the loop id on each call to the LAMP loop iteration function. The total iteration counts are dumped to the end of the same file as the rest of the profile data. The profile thus provides total iteration counts for use in determining the fraction of times the dependence manifests. A sample file is shown in the next section in Figure 3.2 with the original profile results in lines 2 to 21 and the added loop iteration counts shown in lines 23 to 25.

3.3 LAMP Profile Reader

Unlike the previous LAMP implementation in the VELOCITY Compiler, the new LLVM infrastructure does not currently maintain instruction-level annotations (and thus unique identifiers for instruction) in the intermediate representation. Therefore, the ordering and number of instructions must be preserved until LAMP profile data can be read and mapped with the appropriate instruction. The LLVM pass developed as a part of this work operates in three phases:

1. First, the pass uses the uninstrumented bytecode to reissue the same unique identifiers to each loop. As it does this, it creates two maps, one from the LAMP id number to the LLVM basic block that forms the header of the loop and one that maps from the LLVM header basic block back to the LAMP id.

2. Second, the pass uses the uninstrumented bytecode to reissue the same unique identifiers to each memory instruction. As it does this, it creates two maps, one from the LAMP id number to the LLVM instruction and one from the LLVM instruction back to the LAMP id.

3. Finally, it reads the profile generated by running a LAMP-instrumented program. The read profile is then used to identify the instructions that depend upon each other.

```
 1  .  .  .
 2  (115  0  6  112  (1  0  )  )
 3  (115  1  4  142  (7  0  )  )
 4  (120  0  6  112  (1  0  )  )
 5  (120  1  4  142  (7  0  )  )
 6  (123  0  6  112  (1  0  )  )
 7  (123  1  4  142  (3  0  )  )
 8  (127  1  4  142  (4  0  )  )
 9  (130  0  4  124  (4  0  )  )
10  (130  0  4  128  (4  0  )  )
11  (135  0  4  131  (8  0  )  )
12  (136  0  6  112  (1  0  )  )
13  (136  1  4  142  (7  0  )  )
14  (137  0  4  118  (8  0  )  )
15  (138  0  4  133  (8  0  )  )
16  (141  0  6  112  (1  0  )  )
17  (141  1  4  142  (7  0  )  )
18  (144  0  6  112  (1  0  )  )
19  (144  1  4  142  (8  0  )  )
20  (152  0  6  147  (1  0  )  )
21  (152  1  5  158  (1  0  )  )
22  .  .  .
23  1  44
24  2  36
25  3  2
26  4  9
27  5  3
28  6  1
29  .  .  .
```

Figure 3.2: LAMP profile excerpt

Since the LAMP ids are meaningless to a user, the current LAMP reader compiler pass translates these profile results into information that is more useful to a programmer or researcher. As an example, Figure 3.2 shows a few lines from the profile of the code in Figure 3.1. The basic structure of the first portion (lines 2-21) of the profile is: (Mem1 DependenceDistance LoopID Mem2 (Count 0)) where Mem1 depends on Mem2. The second portion (lines 23-28) is: LoopID LoopIterationCount[1]

As an example, the first two lines of the profile are those indicating that *i* depends upon itself in the loop on line 48 in the code in Figure 3.1. Furthermore, discussing possible formats for output with potential users of the system, it was clear that LLVM IR data is of almost no use to a programmer. Such information requires the programmer to delve into LLVM bytecode to decipher information which would render the tool more of a hassle than an aid. For these reasons, in most circumstances, the pass also retrieves information about the original source form of the code. The second part of the profile is also hard to interpret as one must associate loop IDs back to the dependences for it to be helpful – e.g. it is hard to see that the dependence on line 3 in Figure 3.2 manifests 7 times in a total of 9 loop iterations as seen on line 26. This all takes place in the final phase listed above. Below is a further breakdown of the final phase of the LAMP reader pass:

1. The pass determines if the entry is a loop-carried dependence or an intra-iteration dependence so that output will be sent to a different file for each.

2. The entry is parsed and associated (using the previously described maps) with its LLVM instruction.

3. The operands of both the store and the load instruction are traced recursively backwards through the LLVM bytecode until original source code variable names are discovered.

[1]The observant reader will note that the iteration counts include the "iteration" when the loop exit condition is met and no meaningful work is completed. This can distort the fraction of times manifested to total iterations by a factor no greater than the number of invocations of the loop. Nonetheless, the fractions still provide a meaningful measure.

4. If debug instrumentation was used at compile time, the pass then determines the original source line from which the dependence originated.

5. The LAMP-generated dependence information augmented with the information of interest to the programmer is output to file. This information includes the variable name, LLVM basic block name, source function name, and line number of both the store and load of the profiled dependence. The ratio of dependence manifestations to total loop iterations is also calculated here.

As stated previously, the resultant processed profile exists in two files, one for loop-carried and one for intra-iteration dependences. The reason for this was a request by a potential user of the system for loop-carried dependences, which are often of greater interest, to be easily able to quickly distinguish from other true dependences. The basic format of the files in both instances is: [variable1 @ function1 * basicblock1 l# lineNumber – variable2 @ function2 * basicblock2 l# lineNumber] Count P:Fraction L: LoopID

where variable1 depends on variable2. LoopID is omitted for intra-iteration dependences. Originally, only the variable, function, and LLVM basic block name along side the count were available in the processed profile output. This set of data, while useful, was seen to be non-ideal after some discussions with potential users. This is the case because the exact location of the data use is unknown. While the basic block information gives a general offset from the start of the function, it is an LLVM IR construct and difficult for a programmer not interested in examining LLVM bytecode to associate back to the original code. As a result, debug profiling information added at compile time was used to extract original source line numbers. Source line numbers resolved the issue and had the secondary positive characteristic of allowing a programmer to examine dependence information by line number alone, as will be seen in Section 3.4. LoopIDs were also included because of the usefulness of information regarding a specific loop when targeting loops for parallelization. Further discussion of this component is found in 3.4.4

```
1 [f1 @ Fibonacci * bb 1# 11 -- f1 @ Fibonacci * bb 1# 12] 28 P:0.6364 L
2 [f2 @ Fibonacci * bb 1# 11 -- f2 @ Fibonacci * bb 1# 13] 28 P:0.6364 L
3 [f2 @ Fibonacci * bb 1# 12 -- f2 @ Fibonacci * bb 1# 13] 28 P:0.6364 L
4 [i @ fact_g * bb 1# 25 -- i @ fact_g * bb 1# 23] 21 P:0.5833 L:2
5 [i @ fact_g * bb 1# 23 -- i @ fact_g * bb 1# 23] 21 P:0.5833 L:2
6 [i @ fact_g * bb3 1# 23 -- i @ fact_g * bb 1# 23] 28 P:0.7778 L:2
7 [i @ main * bb4 1# 50 -- i @ main * bb7 1# 48] 7 P:0.7778 L:4
8 [i @ main * bb4 1# 52 -- i @ main * bb7 1# 48] 7 P:0.7778 L:4
9 . . .
```

Figure 3.3: LAMP Reader excerpt – Loop-carried output file

Furthermore, it was pointed out by potential users that the information provided by the profile for count was, for reasons mentioned earlier, somewhat insufficient to a programmer trying to decide on candidates for speculation. The programmer would have to determine total iteration counts for themselves. Potential users noted that manually comparing such counts with raw dependence manifestation counts was tedious and greatly diminished the usefulness of the tool. As a result, the total loop iteration counts generated by the new LAMP library code was used to calculate the fractional number of times the dependence manifests in relation to the number of times the loop executes[1].

Examining the samples, the loop-carried dependences for f1 and f2 in the Fibonacci's sequence calculating function "Fibonacci" is seen in 3.3. The first line indicates that f1 at line 11 "Fibonacci" depends upon f1 the value stored to f1 in line 12 of "Fibonacci" in a previous iteration. Similarly, the intra-iteration dependence file shown in 3.4 shows that the initial values from line 8 are used at line 11 and that fnext at line 13 depends upon the value stored to fnext at line 11. The final numbers on the lines in both cases indicate the number of times the dependence manifests and the fraction of times it manifests in relation to the number of times the loop executes.

While this final phase is largely tailored to the goal of extracting information of use to a programmer or researcher interested in examining the source, the general structure of the pass lends itself to relatively minor modifications that would allow the compiler to use

```
 1 [f1 @ Fibonacci * bb 1# 11 -- f1 @ Fibonacci * entry 1# 8] 8 P:0.8889
 2 [f2 @ Fibonacci * bb 1# 11 -- f2 @ Fibonacci * entry 1# 8] 8 P:0.8889
 3 [f2 @ Fibonacci * bb 1# 12 -- f2 @ Fibonacci * entry 1# 8] 8 P:0.8889
 4 [fnext @ Fibonacci * bb 1# 13 -- fnext @ Fibonacci * bb 1# 11] 36 P:0.8
 5 [n @ main * bb7 1# 54 -- n @ main * bb7 1# 52] 8 P:0.8889
 6 [i @ fact_g * bb 1# 25 -- i @ fact_g * entry 1# 23] 7 P:0.7778
 7 [i @ fact_g * bb 1# 23 -- i @ fact_g * entry 1# 23] 7 P:0.7778
 8 [i @ fact_g * bb3 1# 23 -- i @ fact_g * entry 1# 23] 8 P:0.8889
 9 [z @ fact 1# 36 * entry -- n @ main * bb7 1# 52] 1 P:1
10 [n @ fact_g * bb4 1# 28 -- n @ main * bb7 1# 52] 8 P:0.8889
11 [i @ main * bb7 1# 54 -- i @ main * bb 1# 48] 1 P:1.000
12 [fi @ main * bb7 1# 54 -- fi @ main * bb4 1# 51] 8 P:0.8889
13 [fa @ main * bb7 1# 54 -- fa @ main * bb7 1# 53] 8 P:0.8889
14 [i @ main * bb7 1# 48 -- i @ main * bb 1# 48] 1 P:1.000
15 [n @ fact_g * entry 1# 23 -- n @ main * bb7 1# 52] 8 P:0.8889
16 . . .
```

Figure 3.4: LAMP Reader excerpt – Intra-iteration output file

the profiled dependence information. For example, with the maps already in place to map LAMP id back to LLVM instruction, modifying edges of a PDG with profiled information would be possible. In its current implementation, however, the extracted information is used in conjunction with a utility program to present dependence information requested by a programmer.

3.4 LAMP Dependence Viewer Utility

While the dependence information generated to file by the LAMP compiler pass described above in Section 3.3 is structured so that it can be inspected manually, tracing dependence information by hand is intractable for large programs. In fact, even in this simple example program, it is difficult to find dependences of interest amongst the 63 dependence entries generated. To alleviate the complications from tracing dependence data by hand, a viewing utility was created as a part of this work.

3.4.1 Overview

Based upon user request for loop-carried data only or all data, the viewing utility reads the files generated by the LAMP reader, expecting them in a format like the ones seen in Figures 3.3 and 3.4. As it reads data, it constructs a hash-table. Each entry in the hash-table corresponds to a specific instance of a variable from the LAMP reader output with the same name, function, basic block, and source line number. Hashes are done based on variable name only and entries are constructed for all variables that are encountered in parsing the file, whether it is due to a store or to a load. Duplicate entries produced by the profile reader due to multiple IR instructions contributing to a single source code operation are ignored at this point. As will be seen in Section 3.5, removal of duplicate entries notably decreases the number of dependences with which a programmer must contend.

After the appropriate hash-table entry is found or constructed and added to the table, a *required by* node is linked from the entry for the variable that requires it. A *requires* node is linked from the entry for the variable that uses the value generated. If this exact dependence has been seen before, a duplicate node is not created. The entire dependence structure is created in this fashion.

3.4.2 Simple Search Options

Once the dependence table is fully constructed, the user is presented with a menu that allows search by various parameters. Any time a dependence or dependence chain is displayed, the user is given all information available including variable name, function name, and source line number (in cases without debug instrumentation, line number is identified as *unknown*). Simple search options include:

1. List all variables with dependences discovered in a function
2. Find all *required by* relations associated with a variable name
3. Find all *requires* relations associated with a variable name

```
1 Found n in function "main" (block bb7 line 52), feeds:
2  -- n in function "main" (block bb7 line 54)
3  -- z in function "fact" (block entry line 36)
4  -- n in function "fact_g" (block bb4 line 28)
5  -- n in function "fact_g" (block entry line 23)
6 Found n in function "main" (block bb4 line 50), feeds:
7  -- n in function "Fibonacci" (block bb5 line 9)
```

Figure 3.5: LAMP Viewer example search for *required by* tree for variable *n*

Finding all variables with profiled dependences in function *Fibonacci* yields the list fnext, n, f1, f2, i. This option can be particularly useful in a first pass over code, finding which variables (in particular, global variables) are used and thus may need to be dealt with by privatization or communication when parallelizing code. Using the second option to find values needed by global *n* yields the output seen in Figure 3.5. The first entry indicates that the value stored by n at line 52 is used by n at lines 52, 28, and 23, and aliases with z used at line 36. The second entry indicates similar information. Note that the four dependences on n at line 52 displayed here correspond directly to entries at lines 5, 9, 10, and 15 respectively in the original output file in Figure 3.3. This listing displays two particularly useful elements of this system when it comes to parallelizing code. It organizes and sorts dependence information that even in a cleaner form of the output shown in Figures 3.2, 3.3, and 3.4 would be difficult to parse. More importantly, while a programmer may not immediately see which global variables are required in functions deeper in the call graph, particularly when they alias as in the case of z at line 36, a query on any global variable will identify where they are used.

Using the third option to find lines that use the value stored to f1 yields the output seen in Figure 3.6. This indicates the same information seen before in the first line of both Figure 3.3 and Figure 3.4, that f1 at line 11 uses the value written to variable f1 at line 8 and the value written at line 12. If loop-carried only had been selected, only the second dependence would appear.

```
1 Found f1 in function "Fibonacci" (block bb line 11), is flow-dependent
2  -- f1 in function "Fibonacci" (block entry line 8)
3  -- f1 in function "Fibonacci" (block bb line 12)
```

Figure 3.6: LAMP Viewer example search for *required by* tree for variable *f1*

3.4.3 Advanced Search Options

These simple options only give immediate dependence information which is often not completely helpful since there may be multiple call levels through which the memory location passes and is used before the modified value is loaded again for use. Discussion with potential users of the tool led to the addition of a feature that would try to uncover these dependence chains. To provide the programmer with greater information on dependences, the full dependence chain can be displayed. Each of the chain displays operates by recursively, first finding a given variable and all of its required by or requires relations and then finding all of the same relations for those variables. The recursion ignores self-dependences and terminates at a maximum depth based upon a command line argument. Because this element of the utility is more powerful and can generate large amounts of output. In response to problems examining lengthy output by potential users, more options were provided to limit results to those of interest. In particular, rather than generate all results for a variable of a given name, results may be restricted to those only within a given function. Advanced search options therefore include:

1. Find dependence chains for what x is *required by* (flow dependence from x to y)

2. Find dependence chains for what x *requires* (flow dependence from y to x)

3. Find dependence chains for what x is *required by* starting within only one specific function

4. Find dependence chains for what x *requires* starting within only one specific function

5. Find dependence chains for what line number x is *required by* within a specific function

6. Find dependence chains for what line number x *requires* within a specific function

These chain dependence options allow a programmer to see what functions far deeper in the call stack require variables read or written in a certain line of code at a much higher level.

The first option (and the similar options three and five) allows a programmer to trace far into a program what values and calculations a variable assigned at one point will influence. The example seen earlier with *n* in Figure 3.5 is a good example of this use and would be enhanced in larger programs by these expanded options.

The second option (and the similar options four and six) allows a programmer to examine what variables and values contribute to a result. As an example of the second option, Figure 3.7 shows the full "requires" dependence chain. It does give a rather complete view of the various values variable *bars* uses. While useful in many cases, the chain is large and contains many fragments of information. In some cases, a programmer is more interested in information limited to a particular function in question and would prefer to reduce the amount of data displayed. In this case, it is more likely that determining what values influenced the return value of function *fact_g* is of interest. Using the sixth option instead would filter the results for this purpose, as seen in Figure 3.8 showing the discovered "requires" dependence chain for line 29 in function *fact_g*. In this case, the report shows that fact_g directly uses the value of fact_g generated at line 28 which in turn depends on the values of fact_g at line 25 and line 21.

3.4.4 Augmentations for Loop and Code Region Analysis

In addition, the utility provides an option to strip out variable names and basic block naming information in the instance that a programmer wishes to consider dependences based

```
1  bars in function "fact" (block bb5 line 40) is flow-dependent on:
2     bars in function "fact" (block entry line 35)
3  bars in function "fact_g" (block bb4 line 29):
4     bars in function "fact_g" (block bb4 line 28)
5        bars in function "fact_g" (block bb line 25)
6           bars in function "fact_g" (block entry line 21)
7           bars in function "fact_g" (block bb line 25)
8        bars in function "fact_g" (block entry line 21)
9  bars in function "fact_g" (block bb4 line 28):
10    bars in function "fact_g" (block bb line 25)
11       bars in function "fact_g" (block entry line 21)
12       bars in function "fact_g" (block bb line 25)
13    bars in function "fact_g" (block entry line 21)
14 bars in function "fact_g" (block bb line 25):
15    bars in function "fact_g" (block entry line 21)
16    bars in function "fact_g" (block bb line 25)
```

Figure 3.7: Flow-dependency chain for variable *bars*

```
1  bars in function "fact_g" (block bb4 line 29) is flow-dependent on:
2     bars in function "fact_g" (block bb4 line 28)
3        bars in function "fact_g" (block bb line 25)
4           bars in function "fact_g" (block entry line 21)
5           bars in function "fact_g" (block bb line 25)
6        bars in function "fact_g" (block entry line 21)
```

Figure 3.8: Flow-dependency chain for variable *bars* limited to line 29 only

only on line number. This was provided in response to feedback that indicated that potential users really often only wanted to examine information on a line-by-line basis. Line-only output is useful in getting a broader source-level view of dependence information which may be of more use to a programmer or researcher. The line number-based tool also provides the deepest possible dependence chains as it traces all dependences created by that line of code, not just that of a particular variable.

Performing the same request for values line 29 depends on as in Figure 3.8 generates the much more detailed trace seen in Figure 3.9. As was evident from inspection of this simple program, this output shows (in line 6) that the returned value depends on the value of n set in the main function at line 52. Knowledge that the global variable n is necessary in a much lower level computation may be critical to the way a programmer attempts to parallelize a program.

Going into functions to find the return line is, of course, unmanageable in larger programs. However, the lines-only function of the utility makes this problem easily avoidable. Instead, requesting information on what lines require the values stored at a high level source line will allow a programmer to get the same valuable information. Figure 3.10 shows the results of requesting the "required by" dependence information for line 52 in function *main*. While in some cases, information deeper in the call graph is useful, in this instance, it is evident at line 4 and line 6 that *n*'s value is needed in functions *fact_g* and *fact*.

When potential users were shown the lines-only version of the utility, the most common complaint was that it is troublesome to examine multiple lines in succession, in particular all the lines of a loop or function of interest. As a result, two final options were added to the lines-only version of the utility:

1. Find dependence chains for what line numbers n through m are *required by* within a specific function

2. Find dependence chains for what line numbers n through m *require* within a specific function

```
1  function "fact_g" (line 29) is flow-dependent on:
2     function "fact_g" (line 28)
3        function "fact_g" (line 25)
4           function "fact_g" (line 21)
5           function "fact_g" (line 23)
6              function "main" (line 52)
7                 function "main" (line 52)
8                 function "main" (line 48)
9              function "fact_g" (line 23)
10          function "fact_g" (line 25)
11       function "fact_g" (line 21)
12       function "main" (line 52)
13          function "main" (line 52)
14          function "main" (line 48)
15             function "main" (line 48)
```

Figure 3.9: Flow-dependency chain for line 29 in function *fact_g*

```
1  function "main" (line 52) feeds:
2     function "main" (line 54)
3     function "main" (line 52)
4     function "fact" (line 36)
5        function "fact" (line 36)
6     function "fact_g" (line 28)
7        function "fact_g" (line 29)
8           function "fact_g" (line 29)
9     function "fact_g" (line 23)
10       function "fact_g" (line 23)
11       function "fact_g" (line 25)
12          function "fact_g" (line 28)
13             function "fact_g" (line 29)
14                function "fact_g" (line 29)
15          function "fact_g" (line 25)
```

Figure 3.10: *Required by* dependency chain for line 52 in function *main*

Examining the output from such options was largely satisfactory but often led to unreadably long output charts. Potential users requested that the structured output be generated to file for these options. Figure 3.11 shows the full file output for the entirety of the loop in Fibonacci (lines 9-14). Figure 3.12 shows the more interesting loop-carried-only output for the lines within the loop.

The line-range output can be useful for loops, but unfortunately sometimes missed variables that were ONLY modified deep in the call stack. Take, for example, the code in Figure 3.13. The line-range option identifies no dependences. This omitted information, while avoidable if the user used the tool to query lower-level functions or picks out variables of interest instead, greatly diminished the tool's value in the eyes of potential users. As a result, a final augmentation was made to allow the user to generate files containing all dependences that were loop-carried over a loop beginning on a certain line. While the other options provide utility for targeting functions and variables, this final output option is of the greatest use when targeting a loop for parallelization. This option uses the loop-ID-to-source-line-number translation information generated by the LAMP instrumentation pass in conjunction with the loop ID information preserved by request in the reader output. With this information, it identifies all dependences that manifest with respect to the loop beginning on a line requested by the user. Requesting loop-carried dependences with respect to line 19 in the code from Figure 3.13 quickly yields the obvious information shown in Figure 3.14 – there is a loop-carried dependence on n with respect to this loop. If desired, this information can be output in the same chain of dependence manner as in previous options. It also can be generated with line-numbers only.

The final change requested by the potential users was more verbose menus and outputs that completely described all options. These changes have been incorporated and the current form is what exists in the current version of the utility.

While the code used to demonstrate the utility has looked largely at its usefulness in diagnosing dependences around global variables, it is worth mention that the utility is most

```
function "Fibonacci" (line 9) feeds:
   function "Fibonacci" (line 9)


function "Fibonacci" (line 11) feeds:
   function "Fibonacci" (line 13)
      function "Fibonacci" (line 15)
         function "Fibonacci" (line 15)
      function "Fibonacci" (line 12)
         function "Fibonacci" (line 11)
            function "Fibonacci" (line 13)
               function "Fibonacci" (line 15)
               function "Fibonacci" (line 12)
               function "Fibonacci" (line 11)
      function "Fibonacci" (line 11)
         function "Fibonacci" (line 13)
            function "Fibonacci" (line 15)
               function "Fibonacci" (line 15)
            function "Fibonacci" (line 12)
               function "Fibonacci" (line 11)
            function "Fibonacci" (line 11)
               function "Fibonacci" (line 13)


function "Fibonacci" (line 12) feeds:
   function "Fibonacci" (line 11)
      function "Fibonacci" (line 13)
         function "Fibonacci" (line 15)
            function "Fibonacci" (line 15)
         function "Fibonacci" (line 12)
            function "Fibonacci" (line 11)
               function "Fibonacci" (line 13)
         function "Fibonacci" (line 11)
            function "Fibonacci" (line 13)
               function "Fibonacci" (line 15)
               function "Fibonacci" (line 12)
               function "Fibonacci" (line 11)


function "Fibonacci" (line 13) feeds:
   function "Fibonacci" (line 15)
      function "Fibonacci" (line 15)
   function "Fibonacci" (line 12)
      function "Fibonacci" (line 11)
         function "Fibonacci" (line 13)
            function "Fibonacci" (line 15)
               function "Fibonacci" (line 15)
            function "Fibonacci" (line 12)
               function "Fibonacci" (line 11)
            function "Fibonacci" (line 11)
               function "Fibonacci" (line 13)
   function "Fibonacci" (line 11)
      function "Fibonacci" (line 13)
         function "Fibonacci" (line 15)
            function "Fibonacci" (line 15)
         function "Fibonacci" (line 12)
            function "Fibonacci" (line 11)
               function "Fibonacci" (line 13)
         function "Fibonacci" (line 11)
            function "Fibonacci" (line 13)
               function "Fibonacci" (line 15)
               function "Fibonacci" (line 12)
               function "Fibonacci" (line 11)
```

Figure 3.11: *Required by* dependency chain information for loop in function *Fibonacci*

```
1  function "Fibonacci" (line 9) feeds:
2     function "Fibonacci" (line 9)
3
4
5  function "Fibonacci" (line 12) feeds:
6     function "Fibonacci" (line 11)
7
8
9  function "Fibonacci" (line 13) feeds:
10    function "Fibonacci" (line 12)
11       function "Fibonacci" (line 11)
12    function "Fibonacci" (line 11)
```

Figure 3.12: Loop-carried-only required by dependency chain information for loop in function *Fibonacci*

```
1  #include <stdio.h>
2  int n;
3
4  void inc()
5  {
6     n++;
7  }
8
9  void check()
10 {
11    printf("n is %d\n", n);
12    inc();
13 }
14
15 int main ()
16 {
17    printf("Begin.\n");
18
19    while (n < 5)
20       check();
21
22    return 0;
23 }
```

Figure 3.13: Loop-carried analysis example code

```
1 n at 7 in inc feeds:
2     n in function "main" (block bb1 line 21)
3     n in function "check" (block entry line 12)
4     n in function "inc" (block entry line 7)
```

Figure 3.14: Loop-carried analysis output

useful when dealing with pointers and memory locations. Memory aliasing accesses can be quickly traced in the same way as seen in these last examples. Determining what memory locations are accessed in what portions of the program is helpful in picking a scheme for parallelization or developing a new scheme to match data access patterns. Determining whether a possible memory alias manifests also allows speculation as seen in Chapter 2 to be used to increase parallelism.

3.5 Case Studies

As a proof of concept, several benchmarks (plus the toy program described above) were examined and then discussed with potential users of the toolset. In seven of eight cases, information useful in devising a parallelization strategy was quickly discovered. All of these are C language benchmarks. In the final case, that of par2cmdline, C++ name mangling convoluted output and hindered the usefulness of the tool but verification of parallelization strategies determined from manual inspection was still possible. In addition to the code analysis performed using the utility and the LAMP profile reader output, timing overheads were also measured for each of the five cases. Time overhead statistics are reported here for the toy program but further analysis is omitted from this section.

In the case of all benchmarks studied, the loop-carried dependences were of greatest interest in determining opportunities for parallelization. The total number of discovered dependences in the LAMP profile was in all cases much higher than the number of loop-carried dependences as processed by the LAMP profile reader. Once the LAMP viewing utility further processed the LAMP profile reader's output and eliminated redundancy cre-

Benchmark	Total Dependences	Loop-Carried Dependences	Unique Loop-Carried Dependences	In loop of interest
toy example	65	19	13	N/A[2]
130.lisp	1503	161	120	20
181.mcf	812	160	116	1
197.parser	22097	3682	1940	303
256.bzip2	3563	1062	530	7
456.hmmer	2965	495	334	190
par2cmdline	931	55	19	0
crc32	35	6	5	3

Table 3.1: Benchmark dependence statistics

ated by a multiplicity of IR instructions for each source line, the total number of loop-carried dependences for examination by the programmer was reduced to a manageable level. When a loop was ultimately identified for a possible parallelization, selecting the loop limited the output number of loop-carried dependences to a small subset that was easy to examine and analyze meaningfully. This data is presented in Table 3.1. Analysis of each individual benchmark led to the discovery of only a few loop-carried dependences that were important in parallelization as will be discussed in the following subsections.

The overhead of the system is described in Table 3.2. 181.mcf was tested on the reference input, a 2.7MB input file. 130.lisp was tested in batchmode running four simple scripts. 197.parser was tested using the test input. 256.bzip2 was tested using the reference input, a 1MB input file. 456.hmmer was tested using the test input. Par2cmdline was tested with a 13KB file. CRC32 was tested with two 25MB files. All of these inputs were selected to be moderately sized, representative input sets. All tests were done on a 2GHz Pentium 4 CPU with 512KB L2 cache and 512MB main memory. Runtimes are shown in comparison to the original runtime of the programs. Instrumentation time, LAMP reader time, and utility startup time are reported in seconds. Utility startup time includes all building of data structures for the variable-name-included version with all dependences included.

Benchmark	Runtime Increase	Instrumentation	LAMP Reader	Utility Startup
toy example	345.0x	0.060	0.032	0.001
130.lisp	60.00x	3.080	2.324	0.0040
181.mcf	491.5x	0.9360	0.6600	0.0040
197.parser	529.2x	4.784	5.744	0.092
256.bzip2	2012x	0.4199	0.3801	0.0840
465.hmmer	2442x	12.925	8.117	0.056
par2cmdline	115.1x	13.95	26.30	0.001
crc32	428.9x	0.076	0.072	0.001

Table 3.2: Benchmark overhead statistics

3.5.1 130.lisp

The 130.lisp benchmark is a lisp interpreter from the Spec95 benchmark suite and is written in the C language. It was the first benchmark analyzed using the tool suite. As is the case in all of the benchmarks analyzed, the loop-carried dependence output was examined for parallelism opportunities. In particular, the loop-carried dependences on the global state of the interpreter were of interest. Determining the loop-carried dependences across the main interpreter loop was done in two steps. First, queries were performed for variables in the global state using the variable name-included utility. The only loop-carried dependence on global state of any interest was on the interpreter's stack pointer, xlstack. The value of xlstack set on line 71 in xlload (file xlread.c) is required by xlsave on lines 330, 328, and 325 (file xleval.c).

Further queries on the functions of interest in parallelization of 130.lisp with respect to the batch loop yields no further loop-carried dependences that would impede parallelization with respect to that loop. The tool option to dump all loop-carried dependences on the outer loop (main line 47) verified this fact. However, the dependence on the xlstack pointer occurs on every iteration of the batch loop. The question then became whether it would be profitable to speculate the xlstack pointer. Examination of the value loaded to xlstack

showed that on every iteration of the loop the same value was loaded. This made it a strong candidate for speculation. Since no other loop-carried true dependences existed across the loop of interest, the rest of the lisp interpreter's state could be privatized and the batch loop could be parallelized.

3.5.2 181.mcf

The 181.mcf benchmark is from the SPEC CPU2000 suite and uses a network simplex algorithm. In this benchmark, the loop in *primal_net_simplex* is profiled to take approximately 70% of the execution time and thus was the starting point for analysis with the utility. Because of extensive memory operations and pointer-passing between functions, it was helpful in the case of this benchmark to view all dependences rather than just loop-carried dependences. Using the utility in this manner in parallel with dumping the loop-carried dependences for the loop showed that there was only one that might pose a problem for this loop. Within the function *refresh_potential* there is a loop-carried dependence around node with respect to the change of *node->potential*. Examination of the percentage of times this manifested showed that the dependence only rarely occured. Therefore, it would be possible to speculate that no changes were made to any node in the tree. In this way, a DSWP style parallelization scheme was seen to be feasible.

3.5.3 197.parser

The 197.parser benchmark is a benchmark from the SPEC CPU2000 suite that is a syntactic parser of the English language using link grammar. Of all the benchmarks with which LAMPView was tested, 197.parser had a larger number of profiled dependences by an order of magnitude. Nonetheless, LAMPView proved useful in finding opportunities for parallelization. The loop analyzed with the tool was the loop in the function *batch_process* which processes a series of lines given in a batch file. When the loop-carried analysis tool is first run on the outer-most loop in this function, it presents a daunting 300 dependences.

Examining the output on the whole, it is clear that the vast majority of the dependences exist only through the array for storing the sentence, variable *s*. A glance at the code shows that this sentence is actually replaced on each iteration and the variable could thus be safely privatized. This removes the majority of the dependences that are loop-carried with respect to the outer batch loop. Only four loop-carried dependences remain. The first is a dependence that is concerned entirely with the echo of input to the output stream. If the echo output is placed in a sequential pipeline stage, this will be respected.

The remaining three dependences, on variables *mn_free_list*, *lookup_list*, and *string_list*, are of the same type. Each is a list pointer variable that is a part of the state of the system. The use of these variables is very similar to the state variables in the 130.lisp interpreter – each is reset to its original value at the beginning of the iteration. Therefore, they, like the 130.lisp state variables, can be speculated to hold the same value at the start of each iteration. In this way, all loop-carried dependences can be handled.

3.5.4 256.bzip2

The 256.bzip2 benchmark is a compression program from the SPEC CPU2005 benchmark suite. It is written in the C language. Analysis of the 256.bzip2 benchmark again focused on the loop-carried dependences in a dominant outer loop. In this case, the considered target for parallelization was the *compressStream* function. The first step taken in using the utility was to get function reports from both the lines-only and variable names-included versions of the utility. The resultant output indicated loop-carried dependences inside the *compressStream* function at lines 2880, 2879, and 2872 on variables *blockNo* and *combinedCRC*.

Further queries on these dependences showed that they were only self-dependent and no dependence chains manifested when that option was invoked. Queries were then performed on the functions called by compressStream and cross-referenced with the loop-carried output files to determine that there were no other interesting loop-carried dependences with

respect to this outer loop in *compressStream*. The *blockNo* and *combinedCRC* dependences manifest every time. However, this self-loop-carried dependence can be moved into a final sequential stage in a PSDSWP parallelization. Since no true loop-carried dependences exist on the remaining variables, the state can be privatized and those that are necessary can be passed forward in the pipeline.

3.5.5 456.hmmer

The 456.hmmer benchmark is a computational biology application that uses Profile Hidden Markov Models to search for patterns in DNA sequences. Analysis of the 456.hmmer started by examining the outer-most loop. Requesting the output for all loop-carried dependences with respect to this loop yielded 190 dependences. However, a quick inspection of these dependences quickly revealed that they were almost all the same few variables read multiple times in a function. There were only six truly unique dependences to be examined in three separate functions.

Manual inspection of the first pair revealed them to be counting operations, a dependence that could be removed by simple reduction. Examination of the second pair revealed them to be a resizing of a chunk of memory used by the model but no value updates. The final pair was the interesting loop-carried dependence. This dependence was with regards to the internal state of a random number generator used by the sequence. As the ordering of the calls to a random number generator does not matter so long as the distribution is maintained, this function could be marked commutative. With this loop-carried dependence thus removed, no loop-carried dependences remain and the outer-most loop is parallelizable.

3.5.6 Par2cmdline

Par2cmdline is a utility that was developed for the purposes of data-recovery based on a Reed-Solomon code and is used to create and repair data files. It is written in C++ and for this reason proved to be the most difficult to analyze using the utility. The first problem

encountered was that the binary calls additional start up methods prior to calling main. This results in multiple LAMP library calls prior to LAMP initialization and thus requires special care to be taken to instrument the **first** function that will make the first LAMP library call with the initializer library call. The more substantial problem is that the LLVM compiler name-mangles C++ method names and variable names in such a way that it is very difficult to retrieve them in the original source code form in the LAMP reader pass. The resultant names are difficult to decipher and often unreadably long. Queries by function or variable name are impossible without examining the LAMP reader output files to find possible candidates of interest. As a result, the lines-only version of the utility is of much greater use.

Despite the problems with Par2cmdline, the toolset was still helpful. Using the loop-carried dependence dump on the loop within the method *ProcessData* showed it to have no loop-carried dependences that manifest, thus making it easily parallelizable. This query was done simply with the line number of the loop, allowing complete avoidance of the issues created by name mangling.

While identifying this loop was easily done with the LAMPView toolset, identifying other functions for possible parallelism opportunities was complicated due to the name mangling. However inspecting the code in conjunction with the LAMP reader output files and the utility was able to quickly verify another possible parallelization strategy proposed in work by Zhang et al [38]. In particular, it was possible to verify that there was only one true loop-carried dependence with respect to the loop in the method *OpenSourceFiles* and it only occurred through deeper function calls when closing a file.

The experience with Par2cmdline shows that despite its shortcomings when dealing with C++ files, this toolset is still useful in its ability to verify conclusions reached through other means when reasoning about C++ files. Furthermore, if one identifies a loop in advance without investigating dependences by variable name, the tool provides valuable information.

3.5.7 CRC32

CRC32 is a commandline utility that computes a 32-bit cyclic redundancy check on any number of files specified on the command line. Running LAMPView on the program yields only five loop-carried dependences. Generating the loop-carried dependence output for the outer-most loop that performs the batch processing of multiple files yields only three loop-carried dependences that manifest.

All three of these are trivial dependences. Two involve the counting through the args. This could easily be parallelized in many ways including by creating a first sequential pipeline stage to go through each of the arguments. The second is the errors flag. A quick glance at the errors flag shows that it is a simple reduction operation, OR, that is performed each iteration.

Chapter 4

Future Extensions and Conclusions

We have seen the development of many parallelization techniques in the history of computing. While the earlier days saw the development of techniques focusing on scientific code, this second era of parallel computing has seen the proposal of many general purpose parallelization techniques. As more cores are available on each successive generation of processor, researchers must continue finding new ways to exploit parallelism. This often requires difficult program analysis. As the parallel computing era continues, finding these new opportunities will only become more difficult.

New tools are needed to facilitate work in this area. Many useful tools have been developed already[26][38]. This work develops one more tool to the interested researcher or programmer. The LAMPView toolset provides flow dependence information in a structured format, searchable by various parameters including source line number, source variable name, and function names. It presents information about number of occurrences of a dependence as well as fraction of times with respect to total number of loop invocations. Importantly, it also distinguishes between loop-carried and intra-iteration dependences and can be queried for dependence distance. Based on feedback from potential users, all of this information is among the most important information a programmer needs in trying to parallelize a program.

Future work for the toolset includes an improved interface with enhanced display options. A graphical option that could display subsets of dependence chain information would, according to users, be of great use. Furthermore, dependence information for output dependences and anti-dependences would be a very substantial improvement to the toolset. This addition presents a significant challenge in that one must maintain large amounts of backwards-looking data as it is not known in advance whether a certain location will ever be written again. Modifications to allow LAMP to run on multi-threaded applications is also a future goal. Such a modification would allow iterative parallelization as well as aid a programmer in discovering possible data races.

Bibliography

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–236, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[2] J. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 233–246, June 1984.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, California, 2002.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[5] A. J. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, Oct. 1966.

[6] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 99–108, August 2002.

[7] M. J. Bridges. *The VELOCITY Compiler: Extracting Efficient Multicore Execution from Legacy Sequential Codes*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[8] M. Chen and K. Olukotun. Test: A tracer for extracting speculative threads. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:301, 2003.

[9] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 13–24, New York, NY, USA, 2003. ACM.

[10] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.

[11] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software: Practice and Experience*, 29(7):577–603, 1999.

[12] A. Eustace and A. Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the Winter 1995 USENIX Conference*, January 1995.

[13] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, January 2000.

[14] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 59–70, June 2004.

[15] D. Kim and D. Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Transactions on Computing Systems*, 22(3):326–379, 2004.

[16] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[17] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998.

[18] W. Liu, J. Tuck, L. Ceze, K. Strauss, J. Renau, and J. Torrellas. Posh: A profiler-enhanced tls compiler that leverages program structure. 2005.

[19] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.

[20] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *Proceedings of the 13th International Conference on Supercomputing*, pages 365–372, New York, NY, USA, 1999. ACM Press.

[21] S. P. Midkiff and D. A. Padua. Compiler algorithms for synchronization. *IEEE Trans. Comput.*, 36(12):1485–1495, 1987.

[22] C. E. Oancea and A. Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 23–32, New York, NY, USA, 2008. ACM.

[23] G. Ottoni and D. I. August. Communication optimizations for global multi-threaded instruction scheduling. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–232, March 2008.

[24] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, pages 105–116, November 2005.

[25] D. A. Padua Haiek. *Multiprocessors: discussion of some theoretical and practical problems*. PhD thesis, Champaign, IL, USA, 1980.

[26] G. D. Price, J. Giacomoni, and M. Vachharajani. Visualizing potential parallelism in sequential programs. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 82–90, New York, NY, USA, 2008. ACM.

[27] E. Raman. *Parallelization Techniques with Improved Dependence Handling*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, June 2009.

[28] E. Raman, N. Vachharajani, R. Rangan, and D. I. August. Spice: Speculative parallel iteration chunk execution. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, 2008.

[29] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

[30] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, February 2005.

[31] H.-M. Su and P.-C. Yew. Efficient doacross execution on distributed shared-memory multiprocessors. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 842–853, New York, NY, USA, 1991. ACM.

[32] P. Tang, P.-C. Yew, and C.-Q. Zhu. Compiler techniques for data synchronization in nested parallel loops. *SIGARCH Comput. Archit. News*, 18(3b):177–186, 1990.

[33] J. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[34] N. Vachharajani. *Intelligent Speculation for Pipelined Multithreading*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[35] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, September 2007.

[36] T. Vijaykumar, S. Gopal, J. E. Smith, and G. Sohi. Speculative versioning cache. *IEEE Transactions on Parallel and Distributed Systems*, 12(12):1305–1317, 2001.

[37] Q. Yi and K. Kennedy. Improving memory hierarchy performance through combined loop interchange and multi-level fusion. *International Journal of High Performance Computing Applications*, 18(2):237–253, 2004.

[38] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 47–58, Washington, DC, USA, 2009. IEEE Computer Society.

[39] Q. Zhao, J. E. Sim, W.-F. Wong, and L. Rudolph. Dep: detailed execution profile. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 154–163, New York, NY, USA, 2006. ACM.

[40] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *The 14th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2008.

[41] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

Appendix A

Practical LAMPView Information

This appendix shows the steps for the full process to use LAMPView. Appendix Section A.5 shows a full example to run LAMPView. Appendix Section A.6 summarizes all files generated. Refer to the body of the thesis for more information about the files generated at each step and their formats. Appendix B shows common issues a first-time user might encounter.

A.1 Setting up for use with LAMPView

LAMPProfiling.cpp should be compiled and installed with LAMPView defined in order to use it with LAMPView.

Prior to running LAMP instrumentation, you must use llvm-gcc to emit bytecode (with debug information) and link ALL files to be instrumented into one bytecode file. The linking is accomplsihed with llvm-link. e.g.:

```
llvm-gcc -g -emit-llvm -c SOURCEFILES
llvm-link OBJECTFILES -o BYTECODEFILE
```

The loopsimplify optimization should also be run on the full bytecode file in order to fix loops so that the LAMP instrumentation pass can handle the loops. e.g.:

```
opt -loopsimplify < BYTECODEFILE > LS_BYTECODEFILE
```

A.2 Running LAMP Instrumentation

To run instrumentation passes using LLVM's opt, you must run ALL LAMP instrumentation passes at once. Note the optional parameter lamp-init-fn that changes the default function to be instrumented from main to a provided function name; this function MUST be the FIRST among all instrumented functions to be called. Otherwise it will crash.

```
opt -load LLVMINSTALL_DIR/lib/libLAMP.so -debug
    -lamp-insts -insert-lamp-profiling -insert-lamp-loop-profiling
    [-lamp-init-fn FUNCTIONNAME] -insert-lamp-init
    < LS_BYTECODEFILE > INSTRUMENTEDBCFILE
```

The only file generated here as a result of running this optimization is *loops.out*.

A.3 Running LAMP-Instrumented Code

Before running LAMP-Instrumented Code, it must be translated into an executable and linked with the LAMP library. Note that the original lamp_hooks file does not include LAMPView's extra profile section for LAMP loop iteration counts. Preparing to run simply requires linking:

```
llc < INSTRUMENTEDBCFILE > NATIVEASMFILE
g++ -o BINARYFILE NATIVEASMFILE LAMPLIBRARY
```

Running the BINARYFILE then generates the *result.lamp.profile*

A.4 Running the LAMP Reader

Run load profile passes of the LAMP Reader using LLVM's opt. File *result.lamp.profile* expected to be present.

```
opt -load LLVMINSTALL_DIR/lib/libLAMP.so -debug
    -lamp-inst-cnt -lamp-map-loop -lamp-load-profile
    < LS_BYTECODEFILE > FILENAME
```

Running this pass generates files *lcout.out, dout.out,* and *auxout.out*. At this point, any utility can be used. It expects all of the .out files to be present in the working directory. Utility use is detailed in the thesis body.

A.5 Full LAMPView Example

Below is a possible instance of running LAMP.

```
llvm-gcc -g -emit-llvm -c *.c
llvm-link *.o -o main.bc
opt -loopsimplify < main.bc > main.ls.bc
opt -load ~/llvm/llvm-install/lib/libLAMP.so -debug -lamp-insts
    -insert-lamp-profiling -insert-lamp-loop-profiling -lamp-init-fn first
    -insert-lamp-init < main.ls.bc > main.lamp.bc
llc < main.lamp.bc > main.lamp.s
g++ -o main.lamp.exe main.lamp.s lamp_hooks.o

./main.lamp.exe

opt -load ~/llvm/llvm-install/lib/libLAMP.so -debug -lamp-inst-cnt
    -lamp-map-loop -lamp-load-profile < main.ls.bc > /dev/null
```

A.6 Generated Files

Five files are always generated as a result of the LAMPView process and there are three more types of files that can be generated by user request. The first six files are:

1. *loops.out* – Generated as a result of running LAMP Instrumentation with a LAMP build enabled for LAMPView. This is the loop id to line number conversion file.

2. *result.lamp.profile* – Generated as a result of running the LAMP-instrumented program. This is the full dependence profile dumped by LAMP at program completion.

3. *lcout.out* – Generated as a result of running the LAMP Reader pass for LAMPView. This file contains all loop-carried dependences in translated form. This has a 1-to-1 correspondence with the original profile's loop-carried dependence information.

4. *dout.out* – Generated as a result of running the LAMP Reader pass for LAMPView. This file contains intra-iteration flow dependences for all loops profiled. This has a 1-to-1 correspondence with the original profile's intra-iteration dependence information for dependences that occur in loops.

5. *auxout.out* – Generated as a result of running the LAMP Reader pass for LAMPView. This file contains both loop-carried and intra-iteration information. It is the result of the LAMP Reader recursively searching for the variable names in the bytecode in cases when simple iteration is insufficient.

The second type have variable names depending on data requested from the utilities. These are:

1. *firstline#_lastline#_fnname* – When the LAMPDirByLine utility is queried for data on a range of lines, files with names of this format are generated with the result.

2. *Loop#.result* – When the LAMPDumpLoop utility is used, it generates files of this structure where # is the LAMP loop id associated with the loop in question.

3. *Loop#.LO.result* – When the LAMPDumpLoopLines utility is used, it generates files of this structure where # is the LAMP loop id associated with the loop in question.

Appendix B

Common New User Problems

This section covers possible problems encounted from misuse of LAMPView.

B.0.1 Instrumentation fails

1) Did you use the loop-simplify pass first? If you do not, there WILL be a segfault on all but the simplest programs.

2) Did you emit llvm bitcode properly? Otherwise, this should not fail.

B.0.2 Program crashes after instrumentation

1) Is LAMP_init getting called before ANY other LAMP library functions? The error message is usually a vector-out-of-range error in this case. Make sure you passed the FIRST called function to -lamp-init-fn in the instrumentation pass.

2) Did you link ALL object files of interest into a single object file before performing instrumentation? If not, it may be the case that your init function was not found causing the above problem.

3) Is there a HUGE recursive call-depth that calls through loops? If so, you are likely to get an error message similar to the one above. The version of lamp_hooks.cxx used in LAMPView is modified to deal with huge recursive call-depths by providing a larger loop

stack, but if you used the original version or changed this value, this may be a problem.

B.0.3 Unexpected Results

1) Are there more than 1000 loops in your program? If so, your program maxed out on counters for the total loop iterations. Increase the number that is #defined in lamp_hooks.cxx and LAMPReader.cpp.

2) Is LAMP Reader generating incomplete or strange-looking files? Unfortunately, this is likely the result of base limitations in the LLVM bytecode. I can only retrieve name information in the majority of cases, not all (failed attempts at variable names in lcout.out and dout.out get recursively tried and then spit to auxout.out). Sometimes debug information cannot be retrieved properly and so line numbers will appear as UNKNOWN in lcout.out, dout.out, and auxout.out. In this case for loops, loops.out will show loops as occuring on line 0 in this case, and occasionally loops be assumed to be two lines above where they actually are. This is a problem with the way debug information is retrieved and is currently unavoidable.

B.0.4 LAMP Reader fails

1) Did you use the proper version of lamp_hooks.cxx? LAMP Reader expects a prefix to the original profile results file that gives total iteration counts of all loops.

2) Can LAMP Reader write to this directory? LAMP Reader expects the profile result in current working directory and writes its output to this directory.

B.0.5 Utility Issues

LAMPDumpLoops/LAMPDumpLoopsLO can't find loops.out. Did you #define LAMPView when compiling LAMPProfiling? This is what causes loops.out to be produced during the instrumentation pass.

C++ files look bad because of the bytecode that is generated. In these cases, the LAMP Reader spews blank bb and variable names and painfully long function names. Since the utilities expect properly constructed *.out files, the current method of recourse is to preprocess the out files to add bogus names like ”nul” in the blanks. Since C++ name mangling is troublesome, you ought use only the Lines-Only utility versions.