Prospector: Discovering Parallelism via Dynamic Data-Dependence Profiling

Minjang J. Kim, Chi-Keung (CK) Luk‡, Hyesoon Kim
College of Computing, Georgia Institute of Technology, Atlanta, GA
‡Intel Corporation, Hudson, MA

Abstract—Multiprocessor architectures are increasingly common these days. Unfortunately, writing correct and efficient parallel programs remains very challenging. Part of the reason is that software tools like execution profilers for parallel programming is still premature. At the same time, while many compilers support automatic parallelization, we find that they usually fail to parallelize C/C++ programs. In this paper, we propose *Prospector*, a new binary-instrumentation based profiler which *dynamically* detects frequently executed loops and the data dependences that they carry. We demonstrate that Prospector is able to discover parallelizable loops that are missed by state-of-the-art compilers.

1 Introduction

Multiprocessors are becoming mainstream computing platforms nowadays. In order to fully utilize the abundant hardware parallelism, writing correct and efficient parallel programs has become a pressing need. However, parallel programming has had limited success so far compared with sequential programming. Other than the fact that parallel programs are fundamentally more difficult to write, we believe that the lack of software support for parallel programming is also a significant reason.

Ideally, compilers are the perfect tools for exploiting parallelism as they could potentially perform automatic parallelization. However, we find that even state-of-the-art compilers miss many parallelization opportunities in C/C++ programs. As a result, programmers are forced to manually parallelize applications. The success of manual parallelization heavily relies on the quality of execution profilers. Unfortunately, popular execution profilers like Gprof [1] and DevPartner [2] profile programs at the function and/or instruction granularity. This is insufficient for parallel programming as many programs are instead parallelized at the *loop* level.

In this study, we propose *Prospector*, a dynamic binary-level profiler which discovers potential parallelism by *loop profiling* and *data-dependence profiling*. The basic idea of Prospector is dividing the work between software tools and programmers to maximize the overall performance benefit. Prospector provides candidates of parallelizable loops to programmers which were discovered by dynamic profiling. Nevertheless, the decisions of how or whether to parallelize the identified loops are left to programmers. In other words, Prospector aims to bridge the gap between automatic and manual parallelization.

A typical parallelization process on a loop-intensive program consists of the following four steps: (1) finding candidate loops for parallelization, (2) analyzing data dependences in the loops, (3) parallelizing the loops, and (4) verifying and optimizing the parallelized loops. Prospector takes on the first and second steps but leaves the third and fourth steps to programmers for they are better in these steps. More specifically, Prospector provides loop execution profile such as trip counts and the number of instructions executed inside loops. It also dynamically detects *loop-carried data dependences*, which must be preserved during the parallelization process. Parallelization candidate loops are then reported to programmers, who could now focus their efforts on important parallelizable loops. Finally, they perform the actual parallelization using their own favorite programming APIs such as OpenMP [3] and Threading Building Blocks [4].

The rest of the paper is organized as follows. First, we provide insights into when compilers cannot identify parallelizable loops. Second, we discuss our proposed data-dependence profiling algorithm. Finally, we show the results of applying Prospector to a number of benchmarks which cannot be automatically parallelized by state-of-the-art compilers.

2 When Can't Compilers Parallelize Code?

While state-of-the-art compilers support automatic parallelization, we found that these compilers often fail to parallelize C/C++ programs. We present some case studies to demonstrate the limitations of automatic parallelization using Intel C/C++ compiler 10.1.022 (ICC) [5] and the Portland C/C++ compiler 8.0 (PGC) [6]. Both compilers support automatic vectorization and parallelization. We used them to parallelize the OmpSCR [7] benchmark suite, a set of scientific kernels that are already manually parallelized by the programmer using OpenMP pragmas. We used all compiler options that would maximize parallelization opportunities, such as interprocedural analysis and pointer analysis. For ICC, we also disabled cost/benefit analysis so that the compiler would parallelize whenever possible. We were unable to find the option that controls cost/benefit analysis in PGC. Our goal is to see how many of the manually parallelized loops can be automatically parallelized.

Table 1 summarizes the results of automatic parallelization with both compilers. The second column shows the number of manually parallelized loops by the programmer. The third and fifth columns show how many of these manually parallelized loops are automatically parallelized by ICC and PGC, respectively.[1] Overall, ICC parallelizes four of the 13 manually

• {*minjang, hyesoon*}*@cc.gatech.edu*
• ‡*chi-keung.luk@intel.com*

1. The compilers also parallelize other loops that are not parallelized by the programmers. Nevertheless, we found that most of these loops are either not frequently executed or not benefited much from parallelization.

TABLE 1: OmpSCR automatic parallelization results

Benchmark	Programmer	ICC #	Reason	PGC #	Reason
FFT	2	1	Recursion	0	No benefit
FFT6	3	0	Pointers	0	No benefit
Jacobi	2	1	Pointers	0	Pointers
LUreduction	1	0	Pointers	0	Pointers
Mandelbrot	1	0	Reduction	0	Multi-exits
Md	2	1	Reduction	0	Pointers
Pi	1	1	N/A	0	No benefit
QuickSort	1	0	Recursion	0	No benefit
TOTAL	13	4	N/A	0	N/A

```
for(int k = 0; k < N-1; k++)
  #pragma omp parallel for
  for (int i = k + 1; i < N; i++) {
    L[i][k] = M[i][k] / M[k][k];
    for (int j = k + 1; j < N; j++)
      M[i][j] = M[i][j] - L[i][k]*M[k][j];
  }
```

Fig. 1: LUreduction in OmpSCR

```
 1: #pragma omp parallel for reduction(+:outside)
 2: for(int i = 0; i < N; i++) {
 3:   complex z = pt[i];
 4:   for (int j = 0; j < MAXITER; j++) {
 5:     z = z*z + pt[i];
 6:     if (abs(z) > THRESOLD) {
 7:       outside++;
 8:       break;
 9:     }
10:   }
11: }
```

Fig. 2: Mandelbrot in OmpSCR

```
void FFT(Complex *D, int N, ...){
  ...
  #pragma omp parallel for
  for(i = 0; i <= 1; i++)
    FFT(D + i*n, N/2, ...);
  ...
```

Fig. 3: The FFT benchmark in OmpSCR

parallelized loops while PGC parallelizes none. Based on the diagnostic reports from the compilers, we estimate the failing reasons and briefly write them in the fourth and sixth columns. We discuss the major failing reasons in more details below.

2.1 Pointer-based Accesses

C/C++ programmers often prefer using pointers even if equivalent array expressions exist. C99 [8] supports `restrict` to minimize pointer overlapping, but this keyword is not widely used. For instance, the LUreduction code shown in Fig. 1 is manually parallelized at the second-level loop among the three nested loops. The two 2-dimensional `double` arrays, `L` and `M` were dynamically allocated, being accessed via `double**` pointers. ICC does not parallelize the second-level loop due to potential flow dependences and anti-dependences on `M`. However, if these two arrays are statically allocated (e.g., `double M[16][16]`), ICC can then parallelize the loop. Also, when loop bounds are not statically known, the compilers conservatively assume data dependences on arrays.

Pointer-linked data structures like linked lists and trees pose an even greater challenge to compilers. In many cases, compilers simply give up parallelization once pointer-linked data structures are accessed in loops.

2.2 Irregular Control Flows

Compilers may not parallelize loops that have irregular control flows such as branches, breaks, early returns, or function calls (including indirect and virtual function calls). Mandelbrot and Md were not parallelized for this reason. Fig. 2 shows the Mandelbrot benchmark in which the outer loop should be parallelizable by realizing that `outside` is a reduction variable. However, the fact that `outside` is conditionally updated at line 7 and the potential early exit at line 8 confuse the compiler. Consequently, the outer loop is not automatically parallelized.

2.3 Recursive Function Calls

FFT and Quicksort parallelize loops by recursion. The FFT code is shown in Fig. 3. The trip count of the loop is only two, which is equivalent to the fork-join style. However, the compilers fail to recognize that the data accessed by the two paths are independent. As a result, the loop is not parallelized.

3 Overview of Prospector

Prospector performs both loop profiling and data-dependence profiling on a given binary. In this section, we focus on the discussion of our data-dependence profiling algorithm.

Prospector dynamically detects loop-carried Flow (Read-After-Write or RAW in short), Anti (Write-After-Read or WAR), and Output (Write-After-Write or WAW) dependences [9]. Since all memory addresses are resolved at runtime, Prospector can overcome the compilers' limitations discussed in Section 2. In particular, Prospector handles loop-carried dependences for loops that include nested loops, function calls, or recursions. For each loop, the dependence profiling provides the following information:

- Types of loop-carried dependences and the number of their occurrences.
- Sources and targets of the dependences in terms of PC addresses, file names and line numbers, and variable names (if applicable).

The ultimate goal of Prospector is to provide good candidates for loop parallelization. Based on the dependence results, Prospector finds *potentially* parallelizable loops which have no or few dependences. A loop with loop-carried dependences are still potentially parallelizable under the following situations:

- Flow dependences: usually loops with flow dependences are not parallelizable without a significant change of the algorithm. However, some flow dependences from a scalar variable could be simply avoided if the variable is an *induction* or *reduction* variable.
- Output dependences: these dependences are mostly due to *temporary* variables. In general, a temporary variable on a stack is to be private to a thread after parallelization. A loop with this kind of temporary variables should be easily parallelizable.
- Antidependences: a number of code transformations can be used to remove antidependences. For example, antidependences on an array can be avoided by duplicating the array.

3.1 Basic Data-Dependence Profiling Algorithm

We propose a dynamic data-dependence profiling algorithm. For the purpose of parallelization, our algorithm profiles only loop-carried but *not* loop-independent data dependences. Our algorithm operates on three hash tables: the *pending* table, *history* table, and *conflict* table. The pending table captures the memory accesses in a *single* iteration of a particular loop. The history table keeps track of the memory accesses in *all* iterations of a particular loop. The conflict table remembers all data dependences found throughout the program execution. The algorithm is described below:

1) While executing an iteration of a loop, Prospector temporarily records memory accesses of the iteration in the pending table. To ignore loop-independent dependences, all reads from a location which is newly defined inside the same iteration are not recorded in the pending table.
2) On finishing an iteration, the pending table is checked against the history table using memory addresses as keys. All found data dependences are recorded in the conflict table. Then the current content of the pending table is copied to the history table and finally the pending table is flushed.
3) When all iterations of the current loop, say L, finish, the history table of L is merged into the pending table of the parent loop of L (only if L is not the outermost loop in the loop nest). Then the history table is flushed.
4) When the program finishes, Prospector reports the data dependences recorded in the conflict table.

Discovering data dependences across nested loops is essential because parallelizing an outer loop is usually more efficient than an inner loop. Prospector handles nested-loop dependences as if inner loops are completely unrolled. All memory accesses of an inner loop will be copied to its parent in Step 3 of the algorithm. Then, dependences from the nested loops will be recursively checked by Step 2.

3.2 Heuristics for Improving Dependence Profiling

Prospector attempts to report easily removable dependences caused by induction, reduction, and temporary variables. Compilers typically use data-flow analysis to identify these variables [10]. However, since Prospector is based on dynamic binary instrumentation, it does not have the complete flow graph as a compiler does and hence cannot rely on data-flow analysis. Instead, we have developed the following heuristics.

3.2.1 Induction and Reduction Variables

Many loops are bounded by a simple loop-counter variable, also known as an *induction variable* (*IV*). An *IV* incurs all three types of loop-carried dependences, but this type of variable virtually does not stop parallelization. Prospector reports a variable that *may* be an induction variable if:

- All three types of loop-carried dependences occur on an identical memory address (i.e., a scalar variable),
- A single instruction generates the address and its content shows a constant stride increment or decrement.

We use the above heuristics to detect reduction variables as well. For instance, the reduction variable `outside` in Fig. 2 can be identified by these heuristics. However, after Prospector identifies possible *IVs* and reductions, the programmer needs to manually make a final decision whether identified ones are correct. In the future work, we will improve this semi-auto process by implementing a more sophisticated static analysis for binaries.

3.2.2 Temporary Variables

In general, a temporary variable should not be considered incurring loop-carried flow dependences; it is initialized before being used in each iteration. Thus, its data-dependence pattern is a loop-carried output dependence followed by some loop-independent dependences. Once this pattern is detected, Prospector classifies the variable as a temporary variable.

When a temporary variable is detected, the programmer may or may not need to change the code for parallelization. If the temporary variable is allocated *inside* the loop as a local variable, it would be implicitly privatized by the compiler in the parallelized version. In contrast, if the temporary variable is allocated *outside* the loop (e.g., as a static variable), then the programmer would need to explicitly privatize the variable prior to parallelization.

```
   0         1         2         3
   1234567890123456789012345678901234
1: int A[4][4], B[4][4];
2: for (int i = 1; i <= 2; ++i)
3:   for (int j = 1; j <= 2; ++j) {
4:     A[i][j] = A[i][j-1] + 1;
5:     B[i][j] = B[i+1][j] + 1; }
```

Fig. 4: A simple example of data-dependence profiling

TABLE 2: Memory traces and dependences of Fig. 4

	j = 1	j = 2

i = 1	**A[1][1]** = A[1][0] + 1;
	B[1][1] = **B[2][1]** + 1;
		A[1][2] = **A[1][1]** + 1;
		B[1][2] = **B[2][2]** + 1;

i = 2	**A[2][1]** = A[2][0] + 1;
	B[2][1] = B[3][1] + 1;
		A[2][2] = **A[2][1]** + 1;
		B[2][2] = B[3][2] + 1;

(a) Memory traces on the arrays (Boldfaces: conflicting accesses)

Loop	Var	Source and Target	Dependences				Note
			T	#	T	#	
For-i (Outer)	B[]	(R,5,15)→(W,5, 5)	WAR	2			
	i	(W,2,27)↔(R,2,27)	RAW	1	WAR	1	IV
	i	(W,2,27)→(W,2,27)	WAW	1			IV
	i	(W,2,27)↔(R,2,17)	RAW	2	WAR	1	IV
	i	(W,2,27)↔(R,4,17)	RAW	2	WAR	1	IV
	i	...					
	j	(W,3,12)→(W,3,12)	WAW	1			Temp
	j	(W,3,12)→(W,3,29)	iWAW	1			Temp
	j	(W,3,12)→(R,3,29)	iRAW	2			Temp
	j	...					
For-j (Inner)	A[]	(W,4, 5)→(R,4,15)	RAW	2			
	j	(W,3,29)↔(R,3,29)	RAW	2	WAR	2	IV
	j	(W,3,29)→(W,3,29)	WAW	2			IV
	j	(W,3,29)↔(R,3,19)	RAW	4	WAR	2	IV
	j	(W,3,29)↔(R,4,20)	RAW	2	WAR	2	IV
	j	...					

(b) Dependence profiling result (Prefix `i` means loop-independent. The notation used by the source and target of a data dependence is (Read/Write, line number, column number), and dependence fields are pairs of its type(T) and # of occurrence(#).)

3.3 Example of Profiling a Loop Nest

The program in Fig. 4 is profiled by Prospector. Table 3a enumerates all memory accesses on the arrays and Table 3b summarizes the final dependence profiling results.

The array `A[]` has loop-carried RAWs with respect to the inner loop, but no dependences with respect to the outer loop. On the other hand, loop-carried WARs on `B[]` occur in the outer loop while no dependences in the inner loop. Two induction variables, `i` and `j` are found for the outer and inner loops, respectively. The data dependences incurred at `j` with respect to the outer loop are classified as temporary-variable accesses. In this example, the programmer can parallelize the outer loop after removing the WARs by duplicating `B[]`. The inner loop is not parallelizable due to the RAWs on `A[]`.

4 Experimental Results

Prospector is implemented on top of the Pin binary instrumentation framework [11]. Prospector statically analyzes x86

TABLE 3: Did Prospector find what programmers parallelized?

Benchmark	Total Loop #	Parallelized by programmers	Prospector	
			Parallelizable	Matched (Exec Weight)
FFT	6	2	3	2 (80%)
FFT6	28	3	16	3 (91%)
Jacobi	9	2	8	2 (94%)
LUreduction	6	1	4	1 (94%)
Mandelbrot	3	1	2	1 (99%)
Md	12	2	10	2 (97%)
Pi	1	1	1	1 (92%)
QuickSort	7	1	4	1 (77%)

binaries to (1) extract loops and (2) insert minimal instructions to trace loop behaviors and memory accesses. At runtime, profiling data is collected.

4.1 OmpSCR Results

Table 3 summarizes Prospector's results for OmpSCR. The second and third column show the total number of loops and the number of manually parallelized loops by the programmer, respectively. The fourth column has the number of parallelizable loops suggested by Prospector among all loops. The final column is the intersection of the third and fourth columns, which is the number of manually parallelized loops that are also suggested by Prospector. This column also includes the execution weight of matched loops, which is the number of instructions executed in the matched loops as a percentage of the total number of instructions.

Prospector successfully reports all the loops already parallelized by the programmers as parallelizable. The execution weight indicates that these loops dominate the execution time. The heuristics discussed in Section 3.2 effectively eliminate all dependences due to induction, reduction, and temporary variables. Without these heuristics, most loops would not be classified as parallelizable.

Our results also show that Prospector finds more parallelizable loops than what the programmer has already identified. For example, Prospector reports that 16 loops of FFT6 are parallelizable while the programmer parallelizes only three of them. We manually verified that all loops reported by Prospector as parallelizable can be safely parallelized. The programmer decided not to parallelize these extra loops probably because they did not add significant performance benefits.

5 Discussions

Prospector uses dynamic data-dependence profiling and binary instrumentation. We discuss their pros and cons below.

The Dynamic Approach: Dynamic data-dependence profiling alleviates the memory aliasing problem encountered by static dependence analysis. We have shown that the dynamic approach can find parallelism that is not easily discovered by state-of-the-art compilers. However, dynamic profiling is sensitive to inputs. Prospector can suffer from insufficient code coverage. In the future, we will improve this by using a statistical approach (e.g., using a set of randomized inputs).

Binary Instrumentation: Our binary-based approach allows Prospector to be independent of compiler vendors, compiler optimizations, and programming languages. Prospector does not need source code as well. However, as being unable to access high-level semantic information, filtering induction, reduction, and temporary variables needs extra effort. Prospector currently resorts to runtime heuristics and leverages debugging information to obtain symbol names and other information. In the future, we plan to improve the static analysis capability of Prospector and look for ways to pass high-level information from compilers to Prospector.

6 Related Work

A number of loop-profiling tools were proposed in the past [12], [13], [14]. However, they did not implement data-dependence profiling, which is critical for parallelization.

ParaScope Editor [15] built an interactive parallel programming interface which analyzed data dependences with visualization. However, this tool only targeted Fortran and used static data-dependence analysis.

Most recently, Embla [16] also proposed using dynamic data-dependence profiling on binaries. While Embla and Prospector share a similar approach, they have different focuses. Embla concentrates on exploiting fork-join parallelism by discovering general instruction-level data dependences among function calls. In contrast, Prospector concentrates on discovering loop-level parallelism by detecting loop-carried dependences.

7 Conclusions and Future Work

We have presented Prospector, a dynamic binary instrumentation tool that implements loop profiling and data-dependence profiling. We have demonstrated that Prospector can successfully discover parallelizable loops that are not automatically parallelized by state-of-the-art compilers. We believe that the Prospector approach is an interesting alternative between automatic and manual parallelization.

In the future, we plan to improve Prospector in the following ways: (1) incorporating cost-benefit analysis to project speedups, (2) optimizing the data-dependence profiling process, and (3) overcoming the limitations of the dynamic and binary-level approaches. We will also extend Prospector to explore SIMD-level parallelism.

References

[1] S. L. Graham *et al.*, "Gprof: A call graph execution profiler," in *SIGPLAN*, 1982.
[2] Compuware Corp., "Devpartner," http://www.compuware.com.
[3] "OpenMP," http://openmp.org, The OpenMP Architecture Review Board.
[4] J. Reinders, *Intel Threading Building Blocks*. O'Reilly, July 2007.
[5] *Intel Compilers*, Intel Corp., http://www.intel.com/software/products.
[6] *PGI C++ Workstation*, The Portland Group, http://www.pgroup.com/products/workpgcc.htm.
[7] A. J. Dorta *et al.*, "The OpenMP Source Code Repository," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*.
[8] *The ANSI C standard (C99)*, ISO/IEC, http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf.
[9] R. Allen *et al.*, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
[10] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
[11] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[12] J. Mellor-Crummey *et al.*, "Tools for application-oriented performance tuning," in *ICS*, 2001.
[13] T. Moseley *et al.*, "Identifying potential parallelism via loop-centric profiling," in *CF*, 2007.
[14] *Intel Performance Tuning Utility*, Intel Corp., http://softwarecommunity.intel.com/articles/eng/1437.htm.
[15] K. Cooper *et al.*, "The parascope parallel programming environment," *Proceedings of the IEEE*, vol. 81, no. 2, 1993.
[16] K.-F. Faxen *et al.*, "Embla - data dependence profiling for parallel programming," in *Complex, Intelligent and Software Intensive Systems*, 2008, pp. 780–785.