Prospector: A Dynamic Data-Dependence Profiler To Help Parallel Programming

Minjang Kim
College of Computing
School of Computer Science
Georgia Tech
minjang@gatech.edu

Hyesoon Kim
College of Computing
School of Computer Science
Georgia Tech
hyesoon@gatech.edu

Chi-Keung Luk
Technology Pathfinding and Innovations
Software and Services Group
Intel Corporation
chi-keung.luk@intel.com

ABSTRACT

Multiprocessor architectures are increasingly common these days. In order to fully and efficiently utilize the abundant hardware parallelism, translating many sequential programs into parallel programs is a pressing need. Although many compilers support automatic parallelization, most programmers are still manually parallelizing their applications. To help parallelizing applications especially legacy programs written by other programmers, we propose *Prospector*. Prospector is a profile-based parallelism identification tool using dynamic data dependence profiling results. It also advises on how to parallelize the identified sections. We demonstrate that Prospector is able to discover potentially parallelizable loops that are missed by state-of-the-art production compilers. We illustrate its potential on guiding parallel programming for programmers who did not know serial code. This paper also discusses technical challenges on implementing Prospector.

1. INTRODUCTION

Multiprocessors are becoming mainstream computing platforms nowadays. 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 as compared with sequential programming. In addition to the fact that parallel programs are fundamentally more difficult to write, we believe that the lack of software-level support, including libraries and tools for parallel programming, is also a significant reason for parallel programming's limited success.

Compilers may be the ideal 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 small C/C++ programs due to a limitation of static pointer analysis. Automatic parallelization, however, fundamentally has a limited applicable domain because only data-dependence-free loops could be parallelized. As a result, programmers are forced to manually parallelize applications. Nonetheless, a few software tools have been introduced. For example, popular profilers like GNU gprof [8] and DevPartner [4] are not targeted for parallel programming. Recently, Intel Parallel Studio [11] and CriticalBlue [6] have been introduced to help parallel programming for C/C++ and embedded domain, respectively. They provide several useful features, including data race detection, locks-and-waits or concurrency profiling, and some degree of data-dependence profiling, but we find a lot of room for improvement.

In this position paper, we propose *Prospector*, a profiling-based parallelism extraction and advising tool that allows for easier parallel programming. Assume a typical parallelization process that consists of the following four steps: (1) finding candidates for parallelization, (2) understanding the candidates, mostly by analyzing data dependences in the targets, (3) parallelizing the targets, and (4) verification and optimization. The goal of Prospector is to assist with the first three steps.

Prospector aims to bridge the gap between automatic and manual parallelization. Prospector divides the work between software tools and programmers to maximize the overall benefit. It provides programmers with candidates of parallelizable targets that were discovered by dynamic profiling. However, the decisions of how or whether to parallelize the identified loops are left to programmers although Prospector provides advice.

In this paper, we show state-of-the-art compiler and software support, and argue that Prospector is necessary and useful by giving our initial results. We also discuss technical challenges and visions of Prospector.

2. MOTIVATION

We discuss our argument that a profiling-based parallelism extraction tool is necessary by showing the weaknesses of automatic parallelizing compilers. We then present the current status of the latest software tools, and discuss their weaknesses.

2.1 Why a Profiling-based Tool?

The most critical limitation of compiler-based automatic parallelization is the limitation of pointer analysis. For languages like C/C++ where arbitrary pointers and dynamic allocations are allowed, it is known that precise pointer analysis is an undecidable problem [2]. There is also a large body of research related to the approximation of pointer analysis. *Data-dependence profiling* is an alternate approach to addressing this problem by checking data dependences at runtime using actual memory addresses. Data-dependence profiling has been used in various parallelization efforts, including speculative multithreading [22] and profiling-assisted parallelization [24]. It is also being deployed in commercial tools like Intel's Parallel Studio [11] and CriticalBlue's Prism [6].

Unfortunately, dynamic data-dependence profiling inherently has an input dependency problem. Discovered data dependences are only the results of particular input sets. Parallelism extracted from the data-dependence profiling is *potential* parallelism since we cannot prove it. However, even with this weakness, we believe this approach is very helpful for programmers, and it is worth exploring as a research topic and practical tool. Our argument is that parallelization is mostly done for frequently executed code. In such a code section, important data dependence patterns are highly unlikely to be affected by different input sets. Our early experiments also support this claim.

Note that this input dependency problem is inevitable for all dy-

Table 1: OmpSCR Automatic Parallelization Results
How many loops can compilers parallelize automatically?

Benchmark	#	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

namic program analysis techniques. The majority of memory leaks and data-race detectors are implemented as dynamic tools. Fixing all discovered memory leaks and races does not guarantee the safety of the program. However, these tools are critical for successful and efficient software development. Our profile-based data dependence analysis is also based on the same premise. Since manual parallelization is so painful for many programmers, *hints* from Prospector on where and how to parallelize should be helpful.

2.2 Automatic Parallelization in C/C++

This section briefly summarizes the weaknesses of automatic parallelization of the two latest production compilers, which provides the motivation for using Prospector. While state-of-the-art compilers support automatic parallelization, they often fail to parallelize C/C++ programs. We present case studies to demonstrate the limitations of automatic parallelization using Intel C/C++ compiler 11.1 (ICC) [10] and the Portland C/C++ compiler 8.0 (PGC) [20]. We used them to parallelize the OmpSCR [7] benchmarks, a set of scientific kernels that are manually parallelized by programmers using OpenMP pragmas. We use all compiler options that maximize parallelization opportunities.

Table 1 summarizes the results of automatic parallelization with both compilers. The second column shows the number of loops manually parallelized by the programmer. The third and fifth columns show how many of these manually parallelized loops are automatically parallelized by ICC and PGC, respectively. Overall, ICC parallelizes four of the 13 manually parallelized loops, while PGC parallelizes none (mostly due to lack of cost-benefit analysis). Based on the diagnostic reports from the compilers, we estimate the reasons for failure and briefly write them in the fourth and sixth columns. We then discuss the major reasons for failure.

```
1: double** L = malloc(...);                      (a) LUreduction
2: double** M = malloc(...);
3: for(int k = 0; k < N-1; k++)
4:   for (int i = k + 1; i < N; i++) {
5:     L[i][k] = M[i][k] / M[k][k];
6:     for (int j = k + 1; j < N; j++)
7:       M[i][j] = M[i][j] - L[i][k]*M[k][j];
8:   }
```

```
1: for (j = 0; j < N; ++j)                        (b) FFT6
2:   for (k = 0; k < N; ++k)
3:     V[j*N + k] = complex_pow(wn, j*k);
```

```
1: for(int i = 0; i < N; i++) {                   (c) Mandelbrot
2:   complex z = pt[i];
3:   for (int j = 0; j < MAXITER; j++) {
4:     z = z*z + pt[i];
5:     if (abs(z) > THRESOLD) {
6:       outside++;
7:       break;
8:     }
```

```
1: void FFT(Complex *D, int N, ...){              (d) FFT
2:   ...
3:   for(i = 0; i <= 1; i++)
4:     FFT(D + i*n, N/2, ...);
```

Figure 1: Four OmpSCR Benchmarks

2.2.1 *Pointer-based Accesses*

C/C++ programmers often prefer using pointers even if an equivalent array expression exists. C99 [13] supports the `restrict` keyword to minimize pointer overlapping, but this keyword is not widely used. LUreduction in Figure 1.a can be parallelized at the second-level loop among the three nested loops. The two 2D `double` arrays, `L` and `M`, were dynamically allocated, being accessed via `double**` pointers. ICC cannot parallelize the second-level loop due to potential flow dependences and anti-dependences on `M`. However, if these two arrays are statically allocated like `double M[8][8]`, ICC parallelizes it. We also used `restrict` where it is applicable, but the results remained the same. The compilers also failed to parallelize arrays whose bounds are unknown at compile time. Figure 1.b illustrates this case.

2.2.2 *Complex Control Flows*

Compilers may not parallelize loops that have *irregular control* flows such as branches, breaks, early returns, and function calls. Mandelbrot and Md in OmpSCR were not parallelized for this reason. Figure 1.c shows Mandelbrot 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 6 as well as the potential early exit at line 7 confuses the compiler.

FFT and Quicksort are parallelized by recursion, shown in Figure 1.d. The trip count of the loop is only two, but the loop is parallelizable. It is equivalent to the fork-join parallelization. However, the compilers fail to recognize that the data accessed by the two paths are independent.

We also found that code that contains C++ virtual functions and indirect function calls (e.g., callbacks) were not parallelizable. Some loops will be parallelizable if these function calls could be inlined.

2.2.3 *Insufficient Cost-Benefit Analysis*

The compilers statically analyze the cost and benefit of parallelization. However, as seen in the results, PGC misjudged many hot loops. It concludes that many hot loops are providing no benefit. ICC's cost-benefit analysis also has weaknesses. When internal cost-benefit analysis was enabled, only two loops were parallelized. A lack of dynamic execution information of loops is also another major limitation for automatic parallelization. Recently, Tournavitis et al. [24] proposed a profile-driven mechanism that can address this problem.

2.3 State-of-the-art Software Support

Intel recently released Parallel Studio Suite [11] for C/C++ developers. Parallel Studio is composed of several components. Among these components, we focus on *Parallel Advisor Lite*. It is one of the state-of-the-art tools that guide parallel programming. We summarize its approach by an illustration shown in Figure 2.

Suppose a programmer wants to parallelize a sudoku program that generates a hundred sets of Sudoku puzzle and its solution. First, the programmer finds that the for-loop at line 128 is the hottest spot by the profiler. Second, the programmer should analyze data dependences of the loop. Parallel Advisor requires annotations by programmers. Programmers manually insert macros (ANNOTATE_SITE_*) to specify where Parallel Advisor will perform data dependence analysis. Another set of macros (ANNOTE_TASK_*) are used to indicate the loop would be parallelized. This step is shown in Figure 2.b. The third step is a dynamic analysis based on the annotations. We obtain results as shown in Figure 2.c. The programmer sees a data communication (i.e., read-after-write and write-after-read dependences) on line 23 and 25 at `Random` function. To avoid this dependence, the programmer now uses OpenMP's

Figure 2: sudoku example with Intel Parallel Advisor Lite in Parallel Studio Suite

```
125: int main()
126: {
127:   Grid::initialize();
128:   for (int i = 0; i != 100; ++i) {
129:    generate(METHOD_BOX_LINE);
130:   }
131:   return 0;
132: }
```

(a) Original Source Code

```
132: int main()
133: {
134:   Grid::initialize();
135: ➡ANNOTATE_SITE_BEGIN
136:   for (int i = 0; i != 4; ++i) {
137:   ➡ANNOTATE_TASK_BEGIN
138:     generate(METHOD_BOX_LINE);
139:   ➡ANNOTATE_TASK_END
140:   }
141: ➡ANNOTATE_SITE_END
142:   return 0;
143: }
```

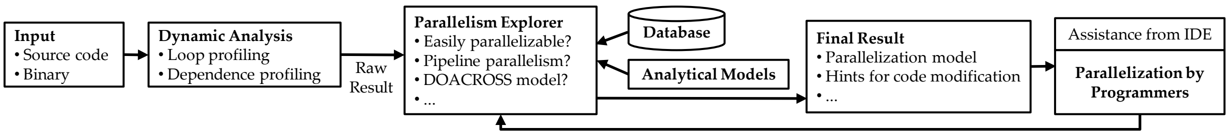
(b) Annotated and modified source code for checking

```
21: Random::Random()
22: {
23:   if (!initialized) {
24:     std::srand(time(0));
25:     initialized = true;
26:   }
27: }         #pragma omp critical
...
34: bool Random::initialized = false;

Problem: Data communication
 Read:  random.cxx:23
 Write: random.cxx:25
```

(c) Dynamic analysis result and its correction

Figure 3: Overview of Prospector



critical section.[1] However, Parallel Advisor does not give advice on how to avoid found dependences.

2.4 Additional Requirements for Dependence Profiler

To our knowledge, Parallel Studio is the first product-level software tool that helps parallelism extraction and parallel programming (CriticalBlue's Prism performs similar tasks, but is limited in the embedded domain). The success of such tools (i.e., how easily and effectively they guide parallel programming) is dependent on the quality of its dependence profiler. Unfortunately, the data-dependence profilers proposed so far are insufficient to meet the goal of Prospector. We discuss the necessary conditions of a data-dependence profiler for Prospector.

First, we need accurate profiling results. Accuracy in this context means that Prospector profiles without losing accuracy for a given input. A simple sampling technique would lose accuracy of profiling results; an-parallelizable loop may be mistakenly reported as parallelizable and vice versa. We found several examples that even losing small accuracy can mislead programmers thereby hurting productivity. Achieving accurate dependence profiling results is the first requirement.

Second, detailed profiling results are necessary. Dependence profiling has been used in thread-level speculation (TLS) [3, 16]. A simple sampling-based dependence profiler that obtains approximated dependence distance and density would mostly suffice for TLS. To parallelize applications for non-speculative hardware, i.e., today's processors, a profiler needs to provide exact locations and frequencies of data dependencies along with other information such as data independence information.

Finally, we need a scalable and low-overhead data dependence profiler while satisfying the above two requirements. All previous profilers often fail to provide good scalability for both memory consumption and time overhead [14, 24]. For example, Parallel Studio cannot profile 6 out of 9 selected C/C++ SPEC 2006 benchmarks (with the train input) due to excessive memory overhead (more than 10GB). Average slowdowns are typically 100-1000x. Observe that the programmer reduced the loop trip count in Figure 2.b from 100 to 4. This is because the dependence profiling takes too much time (typically more than a 300x slowdown).

[1]In practice, the data dependence in `random` can be avoided by using a thread-safe random number generator.

From such observations from the compilers and software tools, we propose Prospector, which helps parallel programming.

3. PROSPECTOR

Recall the parallelization steps: (1) finding candidates for parallelization, (2) analyzing data dependences in the candidates, and (3) parallelizing them. Prospector effectively assists in these three steps. An overview of Prospector is sketched in Figure 3. Prospector takes an input program (an executable or source code) and performs two analyses: (1) instrumentation-time analysis and (2) runtime analysis. During the instrumentation time, an input program is analyzed and instrumented. In the runtime analysis, Prospector performs *loop* and *data* dependence profiling. Various statistics collected during the runtime are post-processed to refine the results. Finally, programmers use the results to parallelize their program.

3.1 Instrumentation-Time Analysis

This analysis is performed at static time. The profiler performs control-flow analysis to identify loop structures. Data structures representing functions and loops are created, and instructions are instrumented for the loop and data dependence analysis. Static data dependence analysis can be also performed at this stage. The input to Prospector can be either source code or binaries. Prospector is implemented in both source-level and binary-level instrumentation frameworks.

3.2 Runtime Analysis: Loop Profiling

Loop-profiling [19] results are essential to choosing the right candidates for parallelization. Especially, hot loop information is provided, which tells programmers where to focus. Statistics about iterations and invocations are also important for efficient parallelization. Even if a loop does not have loop-carried dependences, the benefit of parallelization is diminished if the loop is called too many times and has few iterations. We maintain the correct loop structures at runtime in the presence of complex control flows, including recursive calls, loop nests, and early exit.

3.3 Runtime Analysis: Dependence Profiling

A typical profiler instruments memory loads and stores, and tracks down its execution histories. A naive approach to detect dependences is a pair-wise comparison [14]. Obviously, this simple approach could result in extreme memory usage (more than 10 GB memory consumption for many SPEC 2006 benchmarks with the train inputs). Furthermore, it incurs huge time overhead, usually a 100+ times slowdown.

To solve these scalability problems, we introduce three methods: (1) a stride-based compression technique and a new dependence calculation algorithm, (2) a data-level parallelization framework that is aware of stride behavior, and (3) a pipelined parallelization framework that minimizes overhead of stride detection. Our implementation can save memory consumption more than 30x against the baseline pair-wise algorithm and achieve up to 27x speedup in profiling overhead when 32 cores are used (exploiting multiple machines). We were able to profile entire SPEC 2006 benchmarks with train inputs. We briefly summarize our solutions.

3.3.1 Stride-based Dependence Profiling Algorithm

The main reason for the huge memory overhead is mostly from *affined* memory accesses (e.g., `A[a+b*i]`) inside loops. Our approach dynamically discovers strides and directly checks data dependences with strides. First, we implement a finite state machine to detect stride behavior per memory instructions (i.e., PC). We carefully design the state machine to capture stride-behavior accesses, non-strides, fixed-location accesses, and unpredictable accesses. Stride accesses are stored in a compressed format like {start, end, stride}. Non-strides are stored as points.

The biggest challenges of trace compression is data dependence testing. A number of dynamic program analysis studies have addressed huge memory overhead. For example, Larus used the SEQUITUR compression technique for whole program path profiling [15]. Marathe et al. exploited regular patterns to analyze memory access behavior [18]. However, such compression techniques only solved the trace compression problem. We need an efficient online algorithm that calculates data dependences while saving memory overhead at the same time. Otherwise, we have to decompress traces every time to perform data-dependence profiling.

To perform a data dependence test in a compression format, we need to find data dependences between strides and strides or strides and points. If all memory accesses are stored in the point format, then dependence checking (i.e., given an address, were there previous accesses on this address?) can be done by simple hash table-based searching. However, strides represent intervals, thus we cannot simply use hash tables to do this dependence checking. Instead, we implement an interval-tree-based data structure [5] that expedites collision detection between strides. Once we find overlapped strides and points, we finally calculate exact number of conflicted memory accesses for correct profiling results. We extend the GCD testing so that it can be applicable to our dynamic stride compression format.

Based on our algorithms, Prospector efficiently profiles (a) sources and sinks, (b) frequencies, and (c) distance of dependences. It also considers (d) complex control flows and (e) dynamic execution context. In particular, we must distinguish loop-carried and loop-independent data dependences, as they play a critical role in judging the parallelizability of a loop. Current Parallel Studio could not detect this sensitivity of a loop, which could make an incorrect conclusion in the parallelization step.

3.3.2 Parallelized Dependence Profiling Algorithm

The stride-based compression can solve only the memory consumption problem, but not the slowdown problem. We naturally came up this question: *Can data-dependence profiling itself benefit from parallelization as well?* Fortunately, data-dependence profiling has abundant data-level parallelism that does not require any synchronization between cores. For example, say we have four cores. We divide the memory space into four sub-regions in an interleaved fashion (e.g., addr % 4). Each core then processes its own sub-regions. Data dependences of different memory addresses are not related at all. However, in order to achieve higher speedup while keeping low memory overhead, we need to solve a number of challenges.

First, we can simply divide memory loads and stores to cores based on their addresses, but the events related to loop and function execution must be duplicated to entire cores because these events are critical for the correct data-dependence calculation. Second, load unbalancing is a problem since memory accesses often have high locality. Third, we should enhance the stride detection algorithm on the parallelization framework to maintain low memory overhead. Otherwise, a long stride could be detected as many short strides, which significantly hurts the memory overhead. Finally, the cost of the stride detection is not trivial work. In a naive data-level parallel approach, the same stride detection tasks are done on all cores. To minimize such overhead, we pipeline stride-based data-dependence profiling algorithms.

4. AN ILLUSTRATION: HELPING PARALLEL PROGRAMMING

In this section, we demonstrate the usefulness of Prospector with 179.art. Although 179.art is considered as an easy problem for parallelization in the TLS domain, production compilers cannot automatically parallelize it. A typical programmer who does not know the algorithm detail would easily spend a day or even more paral-

Table 2: Profiling result of `scan_recognize:5` in 179.art

Loop Profiling	79% execution coverage; 1 invocation and 20 iterations; Standard deviation of iteration lengths: 3.5%
Dependence Profiling	Loop-carried WAWs on `f1_layer` ... Temporary variables on `i`, `k`, `m`, `n` Induction variable on `j` at line 5 Reduction variable on `highest_confidence`
	No Loop-carried RAWs on this loop: *parallelizable*

```
 1: void scan_recognize(startx, starty, endx, endy, stride)
 2: {
 3:   ...
 4:   #pragma omp for private (i,k,m,n)
 5:   for (j = starty; j < endy; j += stride)
 6:     for (i = startx; i < endx; i += stride){
 7:       ..
 8:       pass_flag = 0;
 9:       match();
10:       if (pass_flag == 1) {
11:         if (set_high[tid][0] == TRUE) {
12:           highx[tid][0] = i, highy[tid][0] = j;
13:           set_high[tid][0] = FALSE;
14:         }
15:         if (set_high[tid][1] == TRUE) {
16:           ...
18:     }  // End of for-i
...
30: void match()
31: {
32:   reset_nodes();
34:   while (!matched) {
35:     ...
48:     int match_cnfd = simtest2();
41:     if ((match_cnfd) > rho) {
43:        pass_flag = 1;
44:        if (match_cnfd > highest_confidence[tid][winner]){
45:          highest_confidence[tid][winner] = match_cnfd;
46:          set_high[tid][winner] = TRUE;
47:        }
48:     ...
```

```
70: void reset_nodes()
71: {
72:   for (i=0;i<numf1s;i++) {
73:     f1_layer[tid][i].W = 0.0;
74:     Y[tid][i].y = 0.0;
```

Need parallel reduction

Figure 4: Simplified parallelization steps 179.art on multicore by Prospector's result: (1) Privatize global variables; (2) Insert OpenMP pragmas; (3) Add a reduction code (not shown here)

lelizing. 179.art is profiled by Prospector with the train input set. The result and source code changes are summarized in Table 2 and Figure 4, respectively. The detailed steps are as follows:

1. Prospector finds that the loop `scan_recognize:5` in Figure 4 is the hottest loop with 79% execution coverage, only one invocation and 20 iterations. Every iteration has almost an equal number of executed instructions (implying good balance). Hence, the loop should be a good candidate.
2. Fortunately, the loop has *no* loop-carried *flow* dependences except a reduction variable. Strictly speaking, there are dependences on induction (i.e., loop counter variables) and reduction variables, but they can be filtered, as they virtually do not prevent parallelization.
3. `scan_recognize:5` has many loop-carried output dependences on global variables such as `f1_layer` and `Y` (See `reset_nodes()` in Figure 4). These variables are not automatically privatized to threads, so we explicitly allocate thread-local private copies of these global variables. This step is known as privatization, but is not shown in Figure 4.
4. A number of temporary variables (in the scope of the loop:5) are also found such as `i` at line 6. In this code, we need to insert `i` to OpenMP's private list. Note that if line 6 has `int i` (i.e., a temporary variable), then Prospector ignores it.
5. A potential reduction variable (in the scope of the loop:5) `highest_confidence` is also identified. We observe the variable is intended to calculate the maximum value. Given the structure of the original program, using OpenMP's built-in reduction is difficult. Instead, we can modify the code to collect each thread's highest value and compute the final answer. This modification is not shown in Figure 4.

Programmers do not need to know any details of the algorithms and data structures of 179.art. By only interpreting Prospector's loop and dependence profiling results, programmers can easily understand the program and finally parallelize it. It is true that Prospector cannot prove its parallelizability prediction. However, when programmers manually parallelize a loop that cannot be automatically parallelized by compilers, they must prove its correctness by hand or verify it empirically using several tests. In such cases, we claim Prospector is very valuable for programmers.

5. FINDING ADVANCED PARALLELISM

The examples of 179.art and sudoku could be categorized as easy-level parallelization programs, as they do not require significant algorithm changes. However, the natural next question is: *Can we help programmers even if non-trivial data dependence exists?* We cannot simply insert a critical section to preserve dependences since the order of execution is also not guaranteed. Instead, we consider a couple of specific parallelism models.

5.1 Extending Parallelization Models

Prospector attempts to find advanced parallelization models by post-processing the dependence profiling results. We first describe the discovery algorithm for *pipeline parallelism*. Thies et al. showed that pipeline parallelism could be found by dynamic data-flow analysis with programmers' annotations that specify potential pipeline stages [23]. Prospector extends their work by discovering pipeline parallelism *automatically* (i.e., no annotations). After finishing initial profiling, we decide whether a loop could be written in a pipeline model. One of the criteria is whether a loop has a certain pattern of loop-carried flow dependences. Then, post-analysis is done to determine whether feasible pipeline parallelism could be exploited. The critical step is automatic reasoning of potential pipeline stages. We exploit some hints from source code: A loop that will be beneficial from pipeline parallelism generally has several sub-routine calls. Otherwise, we use a sort of greedy algorithm that infers pipeline stage candidates. Based on candidate stages, we abstract data flows among the stages by building a data-flow graph and judge whether it is suitable for pipeline parallelism. If Prospector finally suggests pipeline parallelism, programmers can easily try the suggested model by using the pipeline template of TBB [12]. However, the automatically suggested model may not satisfy the programmer's demand. Prospector then interactively communicates with the programmer to find an alternative model via annotations or other interfaces. This interactive step does not require re-profiling.

Another interesting and feasible pattern is numerical partial differential equations like the fluidanimate benchmark in the PARSEC suite [1]. Significant data-level parallelism exists, but there are steps that communicate with neighbor data cells, which require synchronizations. We extract the characteristics of the already parallelized program in this way and exploit this database to find hidden parallelism for an input program.

5.2 Extending Parallelization Target

So far we have only focused on multicores, but Prospector can be extended to GPGPU and SIMD domains. Given that GPGPU applications are likely to be data parallel and have less synchronization, we strongly expect Prospector could be very helpful for predicting parallelizability for GPGPU. To do so, we not only judge parallelizability, but also predict suitability for GPGPU. For such predictions, we need to make an analytic model that can estimate the overhead of data communication between GPU and host.

SIMD is also a good target for Prospector. Intel's AVX and Larrabee native instructions support 256-bit and 512-bit vector operations, respectively [9, 21]. Automatic vectorization by compilers is also often ineffective due to similar reasons discussed in Section 2.2. Given an analytical model for SIMD and GPGPU, Prospector can exploit extensive ranges of hardware parallelism. In such a hybrid parallelism model, we will have an efficient mapping problem (i.e., how much work should be divided by CPU and GPU), but there is a proposed solution for this problem [17].

6. CONCLUSIONS

In this paper, we proposed Prospector, a profile-based tool to support parallel programming. Prospector exploits loop-level and data-dependence profiling. It identifies easy-to-parallel-loops, which could be easily parallelizable using simple OpenMP pragmas or hard-to-parallel-loops, which require more complex code changes such as inserting critical sections and condition variables. Prospector also provides hints as to how to parallelize loops such as the pipeline model and the GPGPU model. We demonstrate the benefit of Prospector by the illustration of 179.art. We also showed the limitations of the current state-of-the-art production compilers and the parallel programming supporting tools. We believe that a tool like Prospector can significantly improve the productivity of parallel programming. Future work should focus on providing more sophisticated code changes for hard-to-parallel loops.

Acknowledgments

We thank the anonymous reviewers for their comments. We also thank Geoff Lowney, Nagesh B Lakshminarayana, Richard Vuduc, and HParch members for their feedbacks on improving the paper. We gratefully acknowledge the support of NSF CCF0903447, NSF/SRC task 1981, Intel Corporation, and Microsoft Research.

7. REFERENCES

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, 2008.

[2] V. T. Chakaravarthy. New results on the computability and complexity of points–to analysis. *ACM SIGPLAN Notices*, 38(1):115–125, January 2003.

[3] T. Chen, J. Lin, X.Dai, W. Hsu, and P. Yew. Data dependence profiling for speculative optimizations. In *Proc. of 14th Int'l Conf on Compiler Construction (CC)*, 2004.

[4] Compuware Corp. Devpartner. `http://www.compuware.com`.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[6] Criticalblue. *Prism: an analysis exploration and verification environment for software implementation and optimization on multicore architectures*. `http://www.criticalblue.com`.

[7] A. J. Dorta, C. Rodríguez, F. de Sande, and A. Gonzalez-Escribano. The OpenMP Source Code Repository. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 2005.

[8] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN*, 1982.

[9] Intel Corporation. *Advanced Vector Extensions*. `http://software.intel.com/en-us/avx/`.

[10] Intel Corporation. *Intel Compilers*. `http://www.intel.com/software/products`.

[11] Intel Corporation. *Intel Parallel Studio*. `http://software.intel.com/en-us/intel-parallel-studio/home`.

[12] Intel Corporation. *Intel Threading Building Blocks*.

[13] ISO/IEC. *The ANSI C standard (C99)*. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf`.

[14] J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Trans. Parallel Distrib. Syst.*, 4(7):812–826, 1993.

[15] J. R. Larus. Whole program paths. *SIGPLAN Not.*, 34(5), 1999.

[16] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: a tls compiler that exploits program structure. In *PPoPP '06*, 2006.

[17] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Micro-42*, 2009.

[18] J. Marathe, F. Mueller, T. Mohan, S. A. Mckee, B. R. De Supinski, and A. Yoo. *ACM Transactions on Programming Languages and Systems*, 2007.

[19] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 2007 International Conference on Computing Frontiers*, May 2007.

[20] The Portland Group. *PGI C++ Workstation*. `http://www.pgroup.com/products/workpgcc.htm`.

[21] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson. Programming model for a heterogeneous x86 platform. In *PLDI*, 2009.

[22] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. *SIGARCH Comput. Archit. News*, 28(2):1–12, 2000.

[23] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO-40*, pages 356–369, 2007.

[24] G. Tournavitis, Z. Wang, B. Franke, and M. O'Boyle. Towards a holistic approach to auto-parallelization integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI*, 2009.