Predicting Potential Speedup of Serial Code via Lightweight Profiling and Emulations with Memory Performance Model

Minjang Kim Pranith Kumar Hyesoon Kim
School of Computer Science, College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
{*minjang, pranith, hyesoon*}*@gatech.edu*

Bevin Brett
Software and Services Group
Intel Corporation
Nashua, NH, USA
bevin.brett@intel.com

***Abstract*—We present *Parallel Prophet*, which projects potential parallel speedup from an annotated serial program before actual parallelization. Programmers want to see how much speedup could be obtained prior to investing time and effort to write parallel code. With Parallel Prophet, programmers simply insert *annotations* that describe the parallel behavior of the serial program. Parallel Prophet then uses lightweight *interval profiling* and *dynamic emulations* to predict potential performance benefit. Parallel Prophet models many realistic features of parallel programs: unbalanced workload, multiple critical sections, nested and recursive parallelism, and specific thread schedulings and paradigms, which are hard to model in previous approaches. Furthermore, Parallel Prophet predicts speedup saturation resulting from memory and caches by monitoring cache hit ratio and bandwidth consumption in a serial program.**

We achieve very small runtime overhead: approximately a 1.2-10 times slowdown and moderate memory consumption. We demonstrate the effectiveness of Parallel Prophet in eight benchmarks in the OmpSCR and NAS Parallel benchmarks by comparing our predictions with actual parallelized code. Our simple memory model also identifies performance limitations resulting from memory system contention.

I. Introduction

Assisting serial program parallelization is important to both programmer productivity and program performance in the multicore era. In this paper, we wish to answer one of the essential questions on parallelization: *How much speedup could be gained after parallelizing my program?* Before programmers invest their effort into writing parallel programs or parallelizing sequential programs, they would like to see whether parallelization would indeed provide performance benefits. Unfortunately, few practical methods that estimate parallel speedup by analyzing serial code have been developed.

Simple analytical models, such as Amdahl's law [5] and its extensions including Eyerman and Eeckhout [10], and Gustafson's law [12] are effective in obtaining an ideal limit to parallelization benefit. However, such models are not explicitly designed to predict parallel speedup practically because analytical models have difficulty considering realistic and runtime characteristics.

Recently, several tools that employ dynamic analysis and profiling have been introduced including Cilkview [13], Kismet [17], and Suitability analysis in Intel Parallel Advisor [16]. Cilkview analyzes the scalability of a program, but programmers should *parallelize* the program before the prediction. Kismet tries to estimate speedup with an *unmodified* serial program, but this requires huge overhead-critical path analysis. Kismet estimates only an upper bound of the speedup, so it cannot predict speedup saturation.

Intel Parallel Advisor is different than these two approaches because programmers' annotations are needed. However, we should note that annotating is much easier than actual parallelization. Annotations not only enable fast profiling, but also capture programmers' parallelization strategies. Our main goal is to provide fast profiling so that programmers can interactively use the tool to modify their source code. Hence, we also use the annotation approach.

Parallel Advisor provides good speedup estimation for some applications, but it is limited to certain parallel programming patterns. Furthermore, it does not model performance degradations caused by memory bottlenecks. Hence, we propose *Parallel Prophet* to solve these two problems.

Parallel Prophet predicts potential speedup by performing fast *interval profiling* on an annotated program using two emulation algorithms: (1) *fast-forwarding* emulation and (2) *program synthesis-based* emulation. Both emulation methods mimic the parallelized behavior of a given serial program from profiled data and annotations, thereby predicting potential speedups. Our emulators emulate a wide range of parallel programming patterns while accurately considering scheduling policies and parallel overhead. The prediction targets of Parallel Prophet are a subset of the popular parallel programming patterns, as illustrated in Section III, such as loop/task/nested/recursive parallelisms and synchronizations by OpenMP [2] and Cilk Plus [15].

We also propose a simple *memory performance model* to estimate the effects of memory behavior on speedup in parallelized programs. Parallel Prophet exploits low-overhead hardware performance counters and calculates predicted penalty factors of memory operations, which we call

burden factors. One of the biggest performance degradations caused by memory in a parallel system is memory resource contention, such as bandwidth and queuing delays [7, 9]. Hence, as the first-order performance estimation tool and also a very lightweight profiling tool, we only predict the memory resource contention in this mechanism.

The contributions of Parallel Prophet are as follows:

- We propose a new low-overhead mechanism to predict parallel speedups from only annotated serial code. In particular, our new dynamic emulation method, program synthesis-based emulation, precisely and easily models various parallel program paradigms.
- Parallel Prophet employs a simple memory performance model to capture the performance degradation caused by the memory system in parallelized code.

II. Related Work

A. Analytical Models in Predicting Speedups

Presumably, Amdahl's law [5] is the first analytical model to estimate parallel speedup. Based on Amdahl's law, a number of researchers proposed extended analytical models: Gustafson's law [12], Karp and Flat metric [19], a model for multicore chips [26], a model for asymmetric multiprocessors [14], and a recent model for critical sections [10]. However, these models are designed to provide insights for programmers and architects, not to predict speedups from realistic serial programs.

A number of researchers have proposed more sophisticated analytical models that attempted to practically predict speedups. Among them, Adve and Vernon's work [4] may be compared to our fast-forward emulator (Section IV-C). Given inputs including task graph and task scheduling function, Adve and Vernon's analytical model tracks the execution states of processors by traversing the task graph and evaluating the scheduling function. Although this analysis itself is an analytical model, the model requires input variables that mostly need dynamic information.

B. Dynamic Approaches to Predict Speedup

Kismet [17] is a profiler that provides estimated speedups for a given unmodified serial program. Kismet is different from Parallel Prophet in that no annotation is needed. Kismet performs an extended version of hierarchical critical path analysis [11] that calculates self-parallelism for each dynamic region and then finally reports estimated overall speedups by summarizing all region profiles. However, the overhead of Kismet typically shows 100+ slowdowns because memory instructions are instrumented.

Intel Parallel Advisor (specifically, Suitability analysis) [16] tries to estimate the speedups of a program for various threading paradigms and CPU numbers. It does so by collecting timing information from an instrumented serial version of the program, using that information to build a model of the program's dynamic parallel-region tree, and then running that model with an interpreter. The emulation of the model is done by an interpreter that uses a priority queue to fast forward a pseudo-clock to the next event. The emulation includes some details specific to the threading paradigm - for example, how long it takes to acquire a lock (including contention) - but does not consider memory interactions or exact choices of the task to run. Cilkview [13] is different from Kismet, Suitability, and Parallel Prophet because an input program needs to be parallelized already. The purpose of Cilkview is not to predict speedups from a serial code. Rather, Cilkview is a tool that visualizes the scalability of a parallelized program by Cilk Plus.

Suitability uses an approach that is close to ours. However, our experimentation shows that Suitability currently supports limited parallelism models. In the following section, we present the limitations of Suitability in more detail and provide the motivations for our work.

III. Background and Motivation

A variety of parallel program patterns exist, but the state-of-the-art profiling tools can only support a subset of program patterns. The two most basic patterns commonly supported are (1) single-level parallel loops and (2) multiple critical sections. However, based on our survey of OmpSCR (OpenMP Source Code Repository) [1] and NPB [18], we found that the following four cases are not fully supported in the previous approaches.

```
1: for (k = 0; k < size - 1; k++)
2:   #pragma omp parallel for schedule(static,1)
3:   for (i = k + 1; i < size; i++) {
4:     L[i][k] = M[i][k] / M[k][k];
5:     for (j = k + 1; j < size; j++)
6:       M[i][j] = M[i][j] - L[i][k]*M[k][j];
7:   }
```

(a) Workload imbalance and inner loop parallelism in *LUreduction*: Note that the trip count of inner loops varies.

```
 1: void FFT(...) {
...
11:   cilk_spawn FFT(D,   a,      W, n, strd/2, A);
12:              FFT(D+n, a+strd, W, n, strd/2, A+n);
13:   cilk_sync;
...
17:   cilk_for (i = 0; i <= n - 1; i++) {
...
27: }
```

(b) Recursive and nested parallelism in *FFT*: For better efficient execution, OpenMP 2.0 is replaced by Cilk Plus.

Figure 1. Parallel program behaviors in OmpSCR.

- *Workload imbalance*: This is a common cause for low speedups. In Figure 1(a), the amount of work of each iteration is proportional to the `for-i` loop variations. In particular, the shape of work for threads is regular diagonal. Scheduling policies can affect speedups significantly in these cases. Hence, speedup prediction should consider such policies.

Name	Input to the Profiler	Parallel Program Patterns					Overhead
		Simple loops/locks	Imbalance	Inner-loop	Recursive	Memory limited	
Cilkview [13]	*Parallelized* code	◯	◯	◯	◯	×	Moderate
Kismet [17]	Unmodified *serial* code	◯	△	△	△	△ (only super-linear)	Huge
Suitability [16]	Annotated *serial* code	◯	△	△	△	×	Small
Parallel Prophet	Annotated *serial* code	◯	◯	◯	◯	△ (Mem. contention only)	Small

Table I
COMPARISONS OF RECENT DYNAMIC TOOLS TO PREDICT PARALLEL SPEEDUPS. ◯: PREDICTS WELL FOR THE EXPERIMENTATION IN THE PAPER; △: PREDICTIONS ARE LIMITED; ×: NOT EXPLICITLY MODELED.

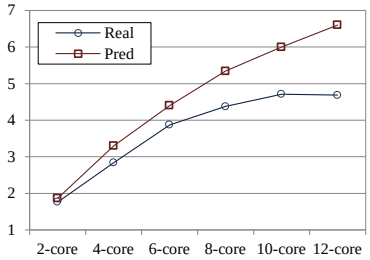

Figure 2. *FT* in NPB. Input set is 'B' of 850 MB memory footprint. Kismet and Suitability overestimate speedups. Speedups are saturated due to increased memory traffics.

- *Inner loop parallelism*: In general, parallelizing an outer loop yields better speedups by minimizing spawning and joining overhead. However, as shown in Figure 1(a), programmers are often forced to parallelize inner loops if an outer loop is not effectively parallelizable. Predicting the cost and benefit of inner loop parallelism would be very informative for programmers.
- *Nested and recursive parallelism*: Speedups of nested or recursive parallel programs are not easy to predict because their runtime behavior heavily depends on parallel libraries. Figure 1(b) shows recursive parallelism, which is also a form of nested parallelism. However, a naive implementation by OpenMP's nested parallelism mostly yields poor speedups in these patterns because of too many spawned physical threads. For such recursive parallelism, TBB, Cilk Plus, and OpenMP 3.0's `task` are much more effective. Hence, speedup prediction should consider these characteristics.
- *Memory-limited Behavior*: Figure 2 shows an example in which the speedup does not scale due to poor memory performance in a parallelized code. This motivates us to build a memory performance model that predicts this behavior. Without any memory models, the speedup could be significantly mispredicted.

Table I summarizes the previous approaches for the above four cases. The reasons for the limitations of these approaches are as follows.

- Modeling realistic parallel behavior: Speedups can be substantially varied by (1) scheduling policies (both parallel libraries and operating systems) and (2) parallel overhead. Kismet does not explicitly model scheduling policies yet. Suitability considers this to some extent. However, it cannot effectively and fully differentiate various schedulings and paradigms.
- Modeling memory performance: Kismet provides a cache-aware model, but it can only predict super-linear effects. The cache model also incurs huge overhead because a cache simulator is used for prediction. Suitability currently implements no such model.

Parallel Prophet overcomes these limitations by introducing a *program synthesis-based* emulation algorithm and a low-overhead *memory performance model*.

IV. DETAILED DESCRIPTIONS OF PARALLEL PROPHET

An overview of Parallel Prophet is shown in Figure 3. First, programmers insert annotations on a serial program and then recompile the annotated program. Second, Parallel Prophet performs *interval profiling* with a representative input. Simple memory profiling that collects hardware performance counters is also conducted. The interval profiling generates a *program tree* for the emulations, and the memory profiling results are augmented on a program tree. Parallel Prophet emulates the parallel behavior using two methods along with a memory performance model. Finally, speedups are reported against different parallelization parameters such as scheduling policies, threading models, and CPU numbers.

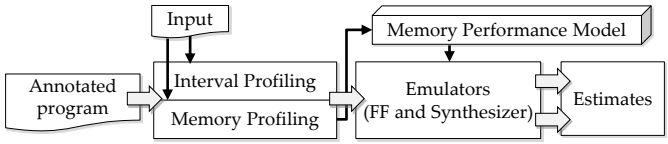

Figure 3. Workflow of Parallel Prophet.

A. Annotating Serial Code

Parallel Prophet takes an annotated serial program that specifies potentially parallel and protected regions. Table II enumerates our annotations, which are similar to those of Suitability except for supporting multiple locks and `nowait` in OpenMP. A pair of `PAR_TASK_*` defines a parallel *task* that may be executed in parallel. A pair of `PAR_SEC_*` defines a parallel *section*, a container in which parallel tasks within the section are executed in parallel. A parallel section defines an implicit barrier at the end unless a programmer

```
 1: PAR_SEC_BEGIN("loop1");
 2: for (i = 0; i < N; ++i) {      // parallel
 3:   PAR_TASK_BEGIN("t1");
 4:   Compute(p1);
 5:   LOCK_BEGIN(lock1);
 6:   Compute(p2); // To be protected
 7:   LOCK_END(lock1);
 8:   if (p3) {
 9:     PAR_SEC_BEGIN("loop2");
10:     for (j = 0; j < M; ++j) { // parallel
11:       PAR_TASK_BEGIN("t2");
12:       Compute(p4);
13:       PAR_TASK_END();
14:     }
15:     PAR_SEC_END(true /*implicit barrier*/);
16:   }
17:   Compute(p5);
18:   PAR_TASK_END();
19: }
20: PAR_SEC_END(true);
```

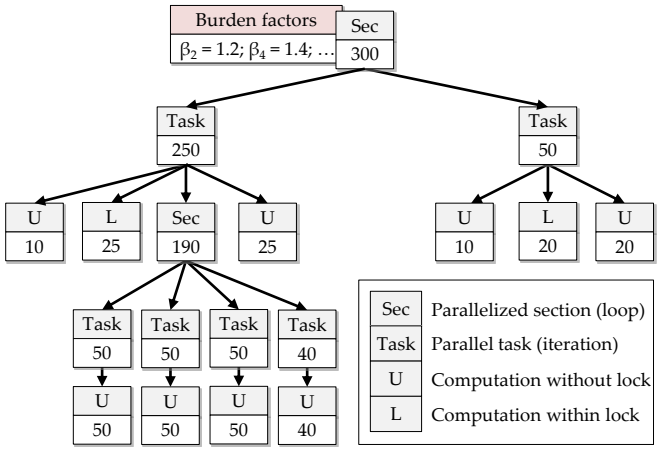

Figure 4. An example of a program tree: The numbers in nodes are the elapsed cycles within the node. Burden factors are computed by the memory performance model and will be multiplied with all terminal nodes in a section. Note that the root node is omitted.

Interface	Description
`PAR_TASK_BEGIN(string task_name)`	This task may be run in parallel.
`PAR_TASK_END()`	The end of a task.
`PAR_SEC_BEGIN(string sec_name)`	This parallel section begins.
`PAR_SEC_END(bool nowait)`	The end of the current section.
`LOCK_BEGIN(int lock_id)`	Try to acquire the given lock.
`LOCK_END(int lock_id)`	Release the owned lock.

Table II
ANNOTATIONS IN PARALLEL PROPHET.

overrides the behavior by, for example, OpenMP's `nowait`. These annotations can describe multiple critical sections and the parallel programming patterns in Figure 1.

Our experience shows that the annotation is very effective and requires little effort compared to actual parallelization efforts, even if a program is embarrassingly parallel. The annotation is currently a manual process. However, this step can be made fully or semi-automatic by several techniques: (1) traditional static analyses from compilers, (2) dynamic dependence analyses [20, 21, 24, 25, 27], and (3) an extended critical path analysis [11].

B. Interval Profiling to Build a Program Tree

Next, Parallel Prophet performs low-overhead *interval profiling* to collect the lengths of all annotation pairs. A length of an annotation pair is defined as either the elapsed time or the number of dynamically executed instructions *between* a pair of annotations. In this implementation, we use the elapsed time (cycles) as the unit of interval profiling.

Meanwhile, hardware performance counters, including cache misses and instruction counts, are collected for each top-level parallel section. The memory performance model will calculate a *burden factor* for each section. Section V discusses the detail of the memory performance model.

The procedure for interval profiling is as follows:

- When an `*_BEGIN` annotation is encountered, the current cycle stamp is pushed on a stack. If the observed annotation is the beginning of a top-level section, we start to collect hardware performance counters.
- On an `*_END` annotation, we match the kinds of current `END` (section, task, or lock) and the top of the stack. If they do not match, an error is reported. If they match, the elapsed cycles between the two annotations are calculated by subtracting the top of the stack from the current cycle stamp. However, we must exclude the profiling overhead itself for accurate results. This important issue is discussed in Section VI-A. If a top-level section finishes, the collection of performance counters is halted. Finally, the stack is popped.

While collecting lengths and counters, Parallel Prophet concurrently builds a *program tree* that records dynamic execution traces of parallel sections from an annotated serial program. The kinds of nodes are (1) `section`, (2) `task`, (3) `L` (computations in a lock), (4) `U` (computations without a lock), and (5) `root` (It has a list of top-level parallel sections and serial sections), as illustrated in Figure 4.

The code of Figure 4 has a critical section and two nested parallel loops. The inner loop may be executed on the condition of parameter `p3`. The inner loop has four iterations, and the length of each iteration is either 40 or 50 cycles. As this figure shows, each iteration (task) is recorded as a separate node. Therefore, the size of the tree could be extremely huge when the trip count of a loop is large and deeply nested parallel loops exist. We solve this problem by compression, which is discussed in Section VI-B.

Figure 4 also shows that the top-level section node has the corresponding burden factors. For example, $\beta_2 = 1.2$ indicates that our memory performance model predicts that if the code were to be parallelized on two threads, the durations of the nodes in this parallel section would be penalized by 20%, due to increased memory contention.

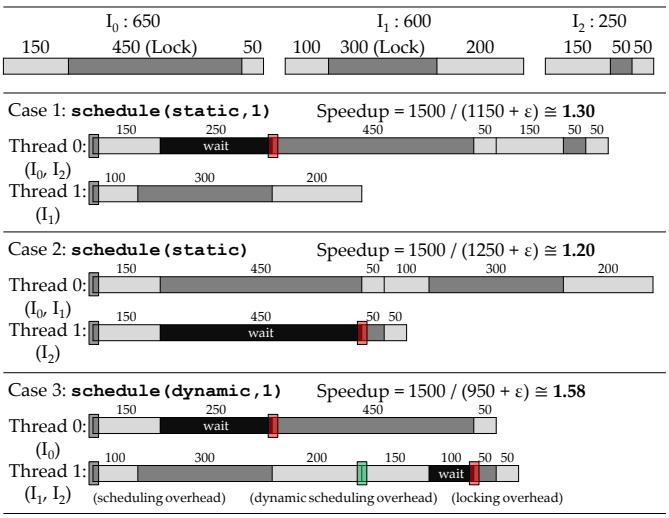

Figure 5. An example of the fast-forwarding method, where a loop with three unequal iterations and a lock is to be parallelized on a dual-core. The numbers above the boxes are the elapsed cycles. ϵ is the parallel overhead.

C. Emulating Parallel Execution: (1) The Fast-Forwarding Emulation Algorithm (The FF)

The final step of the prediction is *emulation*, which emulates parallel execution to compute the projected parallel execution time. Parallel Prophet provides two complementary emulators: (1) fast-forward emulation (*the FF*) and (2) program synthesis-based emulation (*the synthesizer*). The basic idea of both emulations is to emulate parallel behavior by traversing a program tree. The FF emulates in an analytical form that does not require measurement on multicore machines. In contrast, the synthesizer measures the actual speedup of an automatically generated parallel program. We first discuss the FF and its challenges and then introduce the synthesizer in the following section.

The FF traverses a program tree and calculates the expected elapsed time for each ideal processor data structure using a priority heap. The FF predicts speedups for an arbitrary CPU number while considering a specific parallelism paradigm (this paper implements an OpenMP emulator), scheduling policies (static and dynamic schedulings), and parallel overhead (e.g., barriers and locking). Due to space constraints, we cannot present the details of the FF. However, we show two intuitive illustrations.

Figure 5 demonstrates a simple example of (1) workload imbalance, (2) a critical section, and (3) scheduling policies of OpenMP. As shown in the figure, precise modeling of scheduling policies is important for accurate prediction. We also model the overhead of the OpenMP's parallel constructs. We use the benchmarks [8] to obtain the overhead factors. We then add the factors in the FF emulator when (1) a parallel loop is started and terminated, (2) an iteration is started, and (3) a critical section is acquired and released. The overhead is also shown in the illustration as thin colored rectangles.

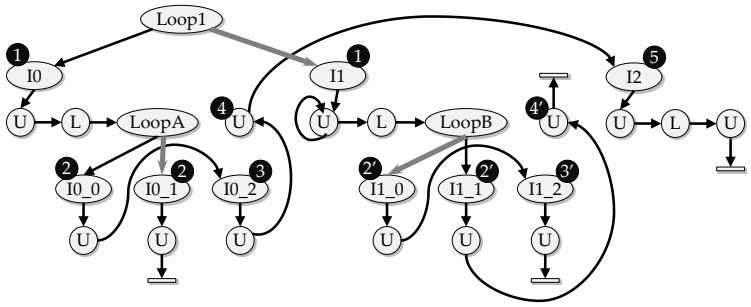

Figure 6. An example of nested parallel loops in the FF: Loop1, LoopA, and LoopB are parallel loops (sections), while LoopA and LoopB are nested parallel loops. I*n* and I*n_k* are loop iterations (tasks). Black circles and arrows indicate an order of the tree traverse, assuming dual-core and OpenMP's `(static,1)`. The same numbers in the black circles (see the gray arrows) mean parallel executions.

Figure 6 illustrates how the FF handles a nested parallel loop. The upper loop (Loop1) has three iterations, and two iterations invoke a nested loop (LoopA and LoopB). All loops are to be parallelized. The order of the tree traverse is depicted in this figure. The exact total order will be determined by the lengths of all U and L nodes. We explain a few cases:

- The first and second iteration of Loop1 (I0, I1) are executed in parallel (the gray arrows at ❶) by the scheduling policy, OpenMP's `(static,1)`. The execution order of the U and L nodes of I0 and I1 are determined by their lengths. The FF uses a priority heap that serializes and prioritizes competing tasks.
- (2) After finishing I0, the FF fetches I2 (❹ → ❺) on the same core, using the `(static,1)` policy.
- (3) For a nested loop, the FF allocates a separate data structure for the scheduling context of the nested loop and processes nested parallel tasks as usual. The priority heap resolves the ordering of parallel tasks, too.

D. Challenges and Limitations in Fast-Forwarding Method

The FF shows highly accurate prediction ability in many cases, but we have observed cases where the FF produces significant errors. First, if the parallel overhead is too high, such as frequent lock contention or frequent fork/join, the predictions are not precise. Second, the FF does not model the details of dynamic scheduling algorithms and interactions with an operating system. These simplifications may often lead to incorrect predictions for complex cases, such as nested and recursive parallelism.

Figure 7 shows a simplified program tree of a two-level nested parallel loop. If the code is parallelized on a dual-core, the speedup is 2 (=30/15). However, our initial FF implementation as well as Suitability give a speedup of 1.5 (=30/20). This is because neither of them models the thread scheduling of the operating system, such as preemptive scheduling and oversubscription (i.e., the number of worker

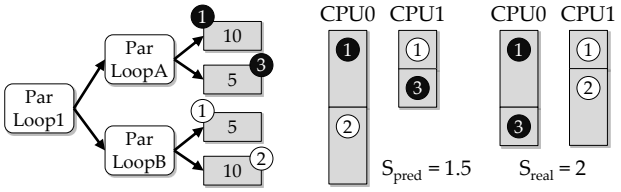

Figure 7. A two-level nested parallel loop where the FF and Suitability cannot predict precisely. The numbers in circles are the scheduling order. The numbers of the nodes are computation lengths. The prediction was 1.5, while the real speedup is 2.0.

threads is greater than the number of processors).

We detail the reason for such inaccuracy in Figure 7. After finishing ①, the FF needs to assign the second parallel task of ParLoopB, ②, to a CPU. However, the FF simply maps ② to CPU0 in a round-robin fashion. Furthermore, the work is assigned to a CPU in a non-preemptive way: A whole `U` (or `L`) node is assigned to a logical processor.

Of course, we can modify this behavior, but a simple ad-hoc modification cannot solve the problem entirely. We must model the thread scheduling of the operating system as well, which is very hard to implement and also takes a long time to emulate. Furthermore, neither FF nor Suitability can predict the case in Figure 1(b): the recursive (or deeply nested) parallelism. We also need a sophisticated emulator that implements logical parallel tasks and a work-stealing scheduler, as in Cilk Plus.

To address these challenges in FF and Suitability, we provide an alternative and complementary way to emulate the parallel behavior, called program synthesis-based emulation.

E. Emulating Parallel Execution: (2) Program Synthesis-Based Emulation Algorithm (The Synthesizer)

The synthesizer predicts speedups by measuring the *actual* speedups of automatically generated parallel code on a *real* machine. A parallel code does not contain any real computation from what the original code has. Instead, it contains intended parallel behaviors, task lengths, and synchronizations. The synthesizer has a template code that generates a parallel code for each parallel section based on a program tree. It then measures its real speedup. As the synthesizer traverses a program tree, all speedups from parallel sections are summed to give an overall speedup. Hence, the overall speedup for a given number of threads, t, is calculated as follows:

$$S = \frac{\sum_i^N \textbf{EmulTopLevelParSec}(sec_i, t) + \sum_i^M \textbf{Length}(U_i)}{\sum_i^N \textbf{Length}(sec_i) + \sum_i^M \textbf{Length}(U_i)},$$

where sec_i is a top-level parallel section node, U_i is a top-level serial computation node (i.e., a `U` node without a `Sec`), N is the total number of top-level parallel sections, and M is the total number of top-level serial nodes. **EmulTopLevelParSec**, defined in Figure 8, returns an estimated parallel execution time for a given (1) top-level parallel section, (2) a number of threads, and (3) a parallelism paradigm.

```
1: void EmulWorkerCilk(NodeSec& sec, OverheadManager&)
2: {
3:   cilk_for (int i = 0; i < sec.trip_count; ++i) {
4:     foreach (Node node in sec.par_tasks[i]) {
5:       overhead += OVERHEAD_ACCESS_NODE;
6:       if (node.kind == 'U')
7:         FakeDelay(node.length* sec.burden_factor);
8:       else if (node.kind == 'L') {
9:         locks[node.lock_id]->lock();
10:        FakeDelay(node.length * sec.burden_factor);
11:        locks[node.lock_id]->unlock();
12:      } else if (node.kind == 'Sec') {
13:        overhead += OVERHEAD_RECURSIVE_CALL;
14:        EmulWorkerCilk(node, overhead);
15:      }
16:    }
17:  }
18: }
19:
20: int64_t EmulTopLevelParSec(NodeParSec& sec, int nt)
21: {
22:   __cilkrts_set_param("nworkers", nt);
23:   OverheadManager overhead;
23:   int64_t gross_time = rdtsc();
24:   EmulWorkerCilk(sec, overhead);
25:   gross_time = rdtsc() - gross_time;
26:   return gross_time -overhead.GetLongestOverhead();
27: }
```

Figure 8. Pseudo code of the synthesizer for Cilk Plus.

Figure 8 has a synthesizer code for Cilk Plus. It first sets the number of threads to be estimated (Line 22). Then, it starts to measure the elapsed cycle by using `rdtsc()` for a given top-level parallel section, which is implemented in the worker function, `EmulWorkerCilk`.

Recall the program tree in Figure 4: a section has a list of children-parallel tasks. These children tasks are to be run concurrently in the parent section. To execute all the children tasks concurrently, we apply a parallel construct, `cilk_for`, on a section node (Line 3). Because we actually use a real parallel construct, unlike the FF, all the details of schedulings and overhead are automatically and silently modeled. For example, we do not need to consider the order of traversing parallel tasks; the parallel library and operating system will automatically handle them. Hence, the problem of Figure 7 does not occur in the synthesizer.

For each concurrently running task, we now iterate all computation nodes (Line 4). The computations in `U` and `L` nodes are emulated by `FakeDeay` in which a simple busy loop spins for a given time without affecting any caches and memory. The computation lengths are adjusted by the burden factors (Lines 7 and 10), which are given by the memory performance model. Note that the computation cannot be simply fast-forwarded like in FF. Doing so may repeat the same mistake of Figure 7. To fully observe the actual effects from an operating system, the actual computation time should be executed on a real machine.

Emulation Method	Fast-Forward	Program Synthesis
Kin	Analytical model	Experimental (dynamic) method
Time Overhead (per estimate)	Mostly 1.1-3× slowdown; Worst case: 30+× slowdown	Mostly 1.1-2× slowdown (See Section VII-D)
Memory overhead	Moderate (by compression)	Moderate (by compression)
Accuracy	Accurate, except for some cases	Very accurate
Target parallel machine	An abstracted parallel machine	A real parallel machine
Supported paradigms	Only implemented parallelism paradigms	Easily supports many parallelism paradigms
Ideal for	To see inherent scalability and diagnose bottleneck. For a small program tree and 1-level parallel loops.	To see more realistic predictions. For a large program tree and nested parallel loops.

Table III
COMPARISONS OF THE FAST-FORWARDING EMULATION AND THE PROGRAM SYNTHESIS-BASED EMULATION.

For an `L` node, a real mutex is acquired and released, thereby emulating precise behavior and overhead of lock contention (Line 9 and 11). Finally, a `Sec` node means nested and recursive parallelism, which is extremely hard to emulate in the FF. Now, it is supported by a simple recursive call (Line 14).

Writing a synthesizer for a different specific parallel paradigm is surprisingly easy. If we replace `cilk_for` with `#pragma omp parallel for`, we will have an OpenMP synthesizer. In sum, the synthesizer easily and precisely models factors related to parallel libraries, operating systems, and target machines.

Nonetheless, implementing a precise synthesizer presents a major challenge: The tree-traversing overhead must be subtracted. If the number of nodes is small, the traversing overhead is negligible. However, a program tree could be huge (an order of GB), for example, if nested loops are parallelized and its nested loops are frequently invoked. In this case, the traversing overhead itself takes considerable time, resulting in significantly deviated estimations. To attack this problem, we measure the overhead of two units in the tree traversing via a simple profiling on a given real machine: `OVERHEAD_ACCESS_NODE` and `OVERHEAD_RECURSIVE_CALL`, as shown in Figure 8. This approach is based on the observation that the unit overhead tends to be stable across various program trees. For example, these two units of overhead on our machine are both approximately 50 cycles. We count the traversing overhead per each work thread (Line 5 and 13), and take the longest one as the final tree-traversing overhead (Line 26).

Unlike the FF, which predicts speedups on arbitrary CPU numbers, the synthesizer can only predict performance for a given real machine. Programmers should run Parallel Prophet where they will run a parallelized code. Table III compares these two approaches.

V. LIGHTWEIGHT MEMORY OVERHEAD PREDICTION

We introduce *burden factors* to model the speedup slowdown due to increased memory traffic. The model is built on an analytical model with coefficients measured on the profiling machine. Runtime profiling information for a given program is obtained from hardware performance counters, as this paper particularly focuses on low overhead up to a 10 times slowdown.

Variation of LLC miss/instr	Observed memory traffic from serial code		
	Low	Moderate	Heavy
Par $\gg$ Ser	Likely scalable	Slowdown+	Slowdown++
Par $\cong$ Ser	Scalable	Slowdown	Slowdown++
Par $\ll$ Ser	Scalable or superlinear	-	-

Table IV
EXPECTED SPEEDUP CLASSIFICATIONS BASED ON MEMORY BEHAVIOR. This paper only considers the second row cases for lightweight predictions.

Note that a burden factor is estimated for each top-level parallel section. If a top-level parallel section is executed multiple times, we take an average. This burden factor is then multiplied to all computation nodes (`U` and `L` nodes) within the corresponding section. It adds overhead to the original execution time and in turn simulates speedup slowdowns.

A. Assumptions

We discuss and justify the assumptions for our low-overhead memory performance model.

• **Assumption 1**: We can separate the execution time of a program into two *disjoint* parts: (a) computation cost and (b) memory cost. Our model only predicts additional overhead on the memory cost due to parallelization. Assumption 2 discusses the computation cost.

• **Assumption 2**: We assume that the work is ideally divided among threads, similar to an SPMD (Single Program Multiple Data) style. Hence, we assume that each thread has n-th of the total parallelizable work, where n is the number of threads. We also assume that no significant differences exist in branches, caches, and computations among threads. The tested subset of NPB benchmarks satisfies this assumption.

• **Assumption 3**: A simplified memory system is assumed. (a) We only explicitly consider LLC(last-level cache), and only the number of LLC misses is used; (b) the latencies of memory read and write are the same; (c) SMT or hardware multi-threading is not considered; and (d) hardware prefetchers are disabled.

• **Assumption 4**: We only consider the case of when the LLC misses per instruction does not significantly vary from serial to parallel. Table IV shows the classifications of applications based on the trend of the LLC misses per instruction from serial to parallel: (1) increases, (2) does not vary significantly, and (3) decreases. However, in order to estimate LLC changes, an expensive memory profiling or

cache simulation is necessary. The main goal of this paper is to provide a lightweight profiling tool. The cases of the first and third rows in Table IV will be investigated in our future work.

• **Assumption 5**: The super-linear case is not considered. When LLC misses per instruction is less than 0.00l for a given top-level parallel section, the burden factor is 1, which is the minimum value in our model.

B. The Performance Model

Based on these assumptions, the execution time (in cycles) of an application may be represented as follows:

$$T = CPI \cdot N = CPI_{\$} \cdot N + \omega \cdot D, \tag{1}$$

where N is the number of all instructions, D is the number of DRAM accesses, $CPI_{\$}$ is the average CPI for all instructions if there are *no* DRAM accesses (i.e., all data are fit into the CPU caches), and ω is the average CPU stall cycles for one DRAM access. For example, if a CPU is stalled for 100 cycles because of 10 DRAM requests, ω is 10. Note that $CPI_{\$}$ and ω represent the disjointed computation cost and memory cost, respectively, as in the first assumption.

C. The Burden Factor

The burden factor for a given thread number t (β_t) represents the performance degradation only due to the memory performance when a program is parallelized. We define β_t as the ratio of T^t and T_i^t, where T^t is the execution time of a parallelized program on a real target machine with t cores, and T_i^t is the execution time on an *ideal* machine where the memory system is perfectly scalable:[1]

$$\beta_t = \frac{T^t}{T_i^t} = \frac{CPI^t \cdot N^t}{CPI_i^t \cdot N_i^t} = \frac{CPI_{\$}^t \cdot N^t + \omega^t \cdot D^t}{CPI_{\$,i}^t \cdot N_i^t + \omega_i^t \cdot D_i^t}. \tag{2}$$

However, our assumptions further simplify Eq. (2):

- N^t, the number of instructions in parallel code for a thread, is obviously the same as N_i^t.
- In our SPMD parallelization assumption, we may safely consider that $N/N^t \simeq t$ and $D/D^t \simeq t$.
- Recall assumption 4: the LLC misses per instruction do not vary from serial to parallel. Hence, MPI (or D/N, LLC misses per instruction from a serial program) will be identical to MPI^t and MPI_i^t.
- We assumed that the computation cost, $CPI_{\$}$ will not change in parallel code. Hence, $CPI_{\$}$, $CPI_{\$}^t$, and $CPI_{\$,i}^t$ are all identical.
- Finally, the DRAM access penalty on a perfectly scalable parallel machine equals the penalty on a single core. Hence, ω_i^t and ω (= ω^1) should be the same.

[1]The superscript t indicates a value from one thread when a program is parallelized on t cores. Note that profiling values from different threads will not vary significantly by the assumption 2. For simplicity, the superscript t can be omitted if t is 1 (or numbers from a serial program).

As a result, the burden factor is finally expressed as:

$$\beta_t = \frac{CPI_{\$} + MPI \cdot \omega^t}{CPI_{\$} + MPI \cdot \omega}. \tag{3}$$

Our goal is to calculate β_t by only analyzing an annotated serial program. We obtain the T, N, D, and MPI of a serial program from hardware performance counters. The remaining terms are now ω, ω^t, and $CPI_{\$}$. However, once we predict ω, Eq. (1) computes $CPI_{\$}$. Therefore, the calculation of the burden factor is reduced to the estimation of ω and ω^t, which equals predicting ω^t.

D. DRAM Access Overhead, w^t Prediction

Recall that ω^t is the additional cycles purely due to DRAM accesses. Our insight for this model is that ω^t would depend on the DRAM access traffic. Hence, the prediction of ω^t can be done by two steps: (1) predict the total DRAM access traffic of t threads by using a serial profiling result only; and (2) predict ω^t under the predicted DRAM traffic. This step is formulated as follows:

$$\delta^t = \Psi(\delta), \tag{4}$$

$$\omega^t = \Phi(\delta^t), \tag{5}$$

where δ is a measured DRAM traffic (in MB/s) by using D, LLC line size, and CPU frequency; δ^t is an estimated DRAM traffic from a *single* thread when t threads are running simultaneously.

Based on this modeling, we find two empirical formulas, Ψ and Φ, via a specially designed microbenchmark. Basically, the microbenchmark measures the elapsed cycles for arbitrary DRAM traffic by making various degrees of DRAM traffic. It also controls the number of threads to measure the total DRAM traffic of multiple threads.

To determine Ψ, we first measure the traffic of the single thread execution from the microbenchmark. Then, we measure the total traffic when multiple threads are running together. Obviously, we would observe both scalable and saturated cases. We obtain Ψ per each thread number on our machine. The machine specification is described in Section VII-A. The formula is (only when $\delta \geq 2000$ MB/s):

$$\begin{aligned} \delta^2 &= (1.35 \cdot \delta + 1758) \ / \ 2, \\ \delta^4 &= (5756 \cdot \ln(\delta) - 38805) \ / \ 4, \\ \delta^8 &= (6143 \cdot \ln(\delta) - 39657) \ / \ 8, \\ \delta^{12} &= (6314 \cdot \ln(\delta) - 39621) \ /12. \end{aligned} \tag{6}$$

Regarding Φ, we also use the same microbenchmark. We first measure total execution cycles by *only* changing LLC misses while L1 and L2 cache misses are *fixed*.[2] However, we must subtract the computation cycles from the total cycles to obtain ω^t. This is done by measuring the total

[2]In practice, we manipulate memory access patterns in our microbenchmark so that *all* memory instructions miss L1 and L2 caches, but while the LLC miss ratio is controlled.

cycles when there are no LLC misses; however, again, L1 and L2 misses must be the same. Then, we can compute the pure overhead due to DRAM accesses for an arbitrary DRAM traffic:

$$w^t = 101481 \cdot (\delta^t)^{-0.964}, \ (\delta^t \geq 2000 \text{ MB/s}). \quad (7)$$

Φ and Ψ may return nonsensical numbers when δ^t is small. Recall assumption 5. In such a small δ range where the LLC misses per instruction is very small, β_t will be 1.

To summarize, obtain the N, T, D, MPI, and δ of a serial program via profiling; estimate δ^t from Eq. (4); estimate ω^t from Eq. (5); finally, Eq. (3) yields the burden factor.

VI. Implementation

We implement Parallel Prophet as a tracer that performs profiling and an analyzer that emulates a program tree. The tracer is based on Pin [22] and PAPI [3], but uses Pin's probe mode to minimize overhead. Our annotations are implemented as C/C++ macros calling corresponding stub functions, which will be detected by Pin.

A. Implementation of Interval Profiling

During profiling, collecting length information as precisely as possible is critical. We use `rdtsc()`, which reads cycle counts for extremely high resolution. However, there are a couple of known problems in using `rdtsc()` on multicore processors, such as interference with other cores. We fix the processor affinity of the tracer (where an input program is running) to a specific processor.

Likewise, in the tree-traversing overhead issue in Section IV-E, we also need to exclude the profiling overhead itself from the length values. If we use the unit of length as the number of executed instructions, the problem is easy to solve. However, we observed that different instruction mixes cause a lot of prediction errors. Also, obtaining instruction counts and mixes incurs additional overhead (need to instrument every basic block). Instead, we use time as the unit. We tried our best to calculate the *net* length of each node, but every detail cannot be described here.

B. Profiler Memory Overhead Problem and Solutions

A program tree may consume huge memory because all intervals of the loop iterations are recorded. Many benchmarks in our experiments use moderate memory consumption (less than 500 MB); however, IS in the NPB benchmark consumes 10 GB to build a program tree. To solve this memory overhead problem, we apply a compression technique. When intervals of loop iterations do not vary significantly, we perform *lossless* compression using a very simple RLE(Run-length encoding) and a simple dictionary-based algorithm. Such simple compression is quite effective. For example, the program tree of CG in NPB (with 'B' input) can be compressed into 950 MB from 13.5 GB (a 93% reduction). In our implementation, we allow 5% of variation

```
1: // To be parallelize by OpenMP.
2: for (int64_t i = 0; i < i_max; ++i) {
3:   overhead = ComputeOverhead(i, i_max, M, m, s);
4:   FakeDelay(overhead * ratio_delay_1);
5:   if (do_lock1) {
6:     // To be protected.
7:     FakeDelay(overhead * ratio_delay_lock_1);
8:   }
9:   FakeDelay(overhead * ratio_delay_2);
10:  if (do_lock2) {
11:    // To be protected.
12:    FakeDelay(overhead * ratio_delay_lock_2);
13:  }
14:  FakeDelay(overhead * ratio_delay_3);
15: }
```

Figure 9. `Test1`: workload imbalance + locks + high lock contention.

```
1: // To be parallelize by OpenMP.
2: for (int64_t k = 0; k < k_max; ++i) {
3:   overhead = ComputeOverhead(k, k_max, M, m, s);
4:   FakeDelay(overhead * ratio_delay_A);
5:   if (do_nested_parallelism)
6:     Test1(k, k_max);
7:   FakeDelay(overhead * ratio_delay_B);
8: }
```

Figure 10. `Test2`: `Test1` + inner-loop parallelism + nested parallelism.

to be considered as the same length. However, simple compressions may not be feasible if the iteration lengths of a loop are extremely hard to compress in a lossless way. As a last resort, we may use lossy compression. Fortunately, lossy compression was not needed in our experimentations. With lossless compression, 3 GB of memory is sufficient for all evaluated benchmarks in this paper.

VII. Experimentation Results

A. Experimentation Methodologies

All experiments are done on a 12-core machine with 24 GB memory and two sockets of an Intel Xeon processor (Westmere architecture). Each CPU has six cores and 12 MB L3 cache. Hyper-Threading, Turbo Boost, SpeedStep, and hardware prefetchers are all disabled. We use OpenMP and Cilk Plus of Intel C/C++ Compiler 12.0. Finally, eight benchmarks in OmpSCR [1] and NPB [18] are evaluated.

B. Validation of the Prediction Model

We first conduct validation experiments to verify the prediction accuracy of Parallel Prophet without considering memory and caches. This step is important to quantify the correctness and diagnose the problems of Parallel Prophet before predicting realistic programs with our memory performance model.

We use randomly generated samples from two serial program patterns: `Test1` and `Test2`, shown in Figure 9 and Figure 10, respectively. `Test1` exhibits (1) load imbalance, (2) critical sections whose contentions and lengths are arbitrary, and (3) a high parallel overhead case: high lock contention. `Test2` shows all of the characteristics of Test1 as well as (1) a high parallel overhead case:

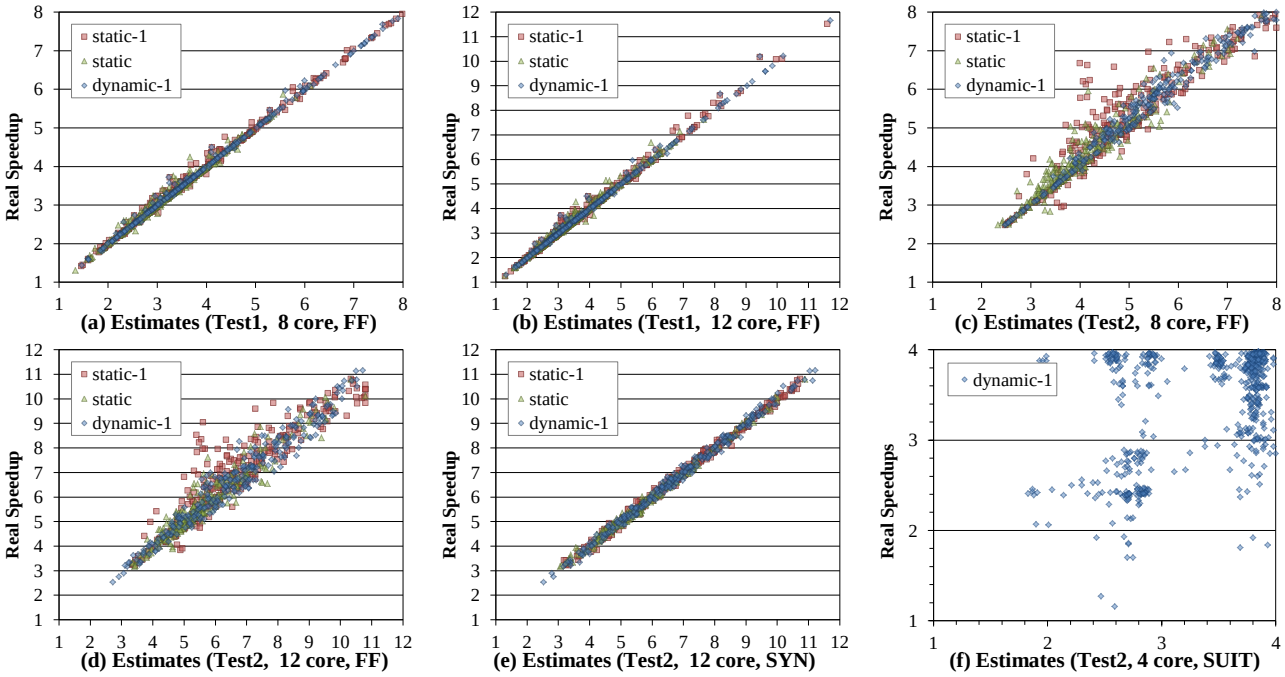

Figure 11. Prediction accuracy results of `Test1` and `Test2`, targeting eight and 12 cores, and OpenMP: FF and SYN mean the FF and the synthesizer emulations, respectively. SUIT is the Suitability in Parallel Advisor.

frequent inner loop parallelism and (2) nested parallelism, as well as all the characteristics of `Test1`. The function `ComputeOverhead` in the code generates various workload patterns, from a randomly distributed workload to a regular form of workload, or a mix of several cases.

We generate 300 samples per each test case by randomly selecting the arguments. These randomly generated programs are annotated and predicted by Parallel Prophet. To verify the correctness, we also parallelize samples using OpenMP and measure real speedups on 2, 4, 6, 8, 10, and 12 cores. OpenMP samples use three scheduling policies: `(static,1)`, `(static)`, and `(dynamic,1)`. We finally plot the predicted speedups versus the real speedups to visualize the accuracy. We only show the results of 8 and 12 cores in Figure 11, but the rest of the results show the same trend.

Fast-forwarding emulation achieves very high accuracy for `Test1`, shown in Figure 11 (a) and (b); the average error ratio is less than 4% and the maximum error ratio is 23% for a case of predicting 12 cores. Figure 11 (c) and (d) show the results of `Test2` with the FF. The average error ratio is 7% and the maximum was 68%, which are higher than the results of `Test1`. Among the three schedulings, `(static)` shows more severe errors. Our investigation of such errors tentatively concludes that operating system-level thread scheduling can affect the speedups, which the FF currently does not precisely model. An example was discussed in Figure 7. We also discover that the overhead of OpenMP is not always constant, unlike the results from the previous work [6, 8]. The previous work claims that the overhead of OpenMP constructs can be measured as constant delays, but our experience shows that the overhead is also dependent on the trip count of a parallelized loop and the degree of workload imbalance. To address such difficulties, we have introduced the synthesizer. Fortunately, the synthesizer can provide highly accurate predictions for `Test2`, showing a 3% average error ratio and 19% at the maximum, shown in Figure 11(e). Please note that such a 20% deviation in speedups is often observed in multiple socket machines; thus we consider a 20% error as a boundary for reasonably precise results.

We finally conduct validation tests with Suitability analysis. Due to constraints of the out-of-the-box tool, we are only able to predict `Test1` and `Test2` up to quad cores. The results of `Test1` (not shown in the paper) are as accurate as ours. However, Figure 11(f) shows that Suitability does not predict `Test2` well. Note that Suitability does not provide speedup predictions for a specific scheduling. Our experience shows that the emulator of Suitability is close to the OpenMP's `(dynamic,1)`. We speculate that the reasons for such inaccuracy would be (1) the limitations of the emulator discussed in Section IV-D, and (2) the lack of precise modeling of OpenMP's scheduling policies.

C. OmpSCR/NPB Results with Memory Performance Model

We present the prediction results on four OmpSCR (MD, LU, FFT, QSort) and four NPB benchmarks (EP, FT, MG, CG) in Figure 12. Each benchmark is estimated by (1) the synthesizer *without* the memory model ('Pred'), (2) the synthesizer *with* the memory model ('PredM'), and (3) Suitability ('Suit').

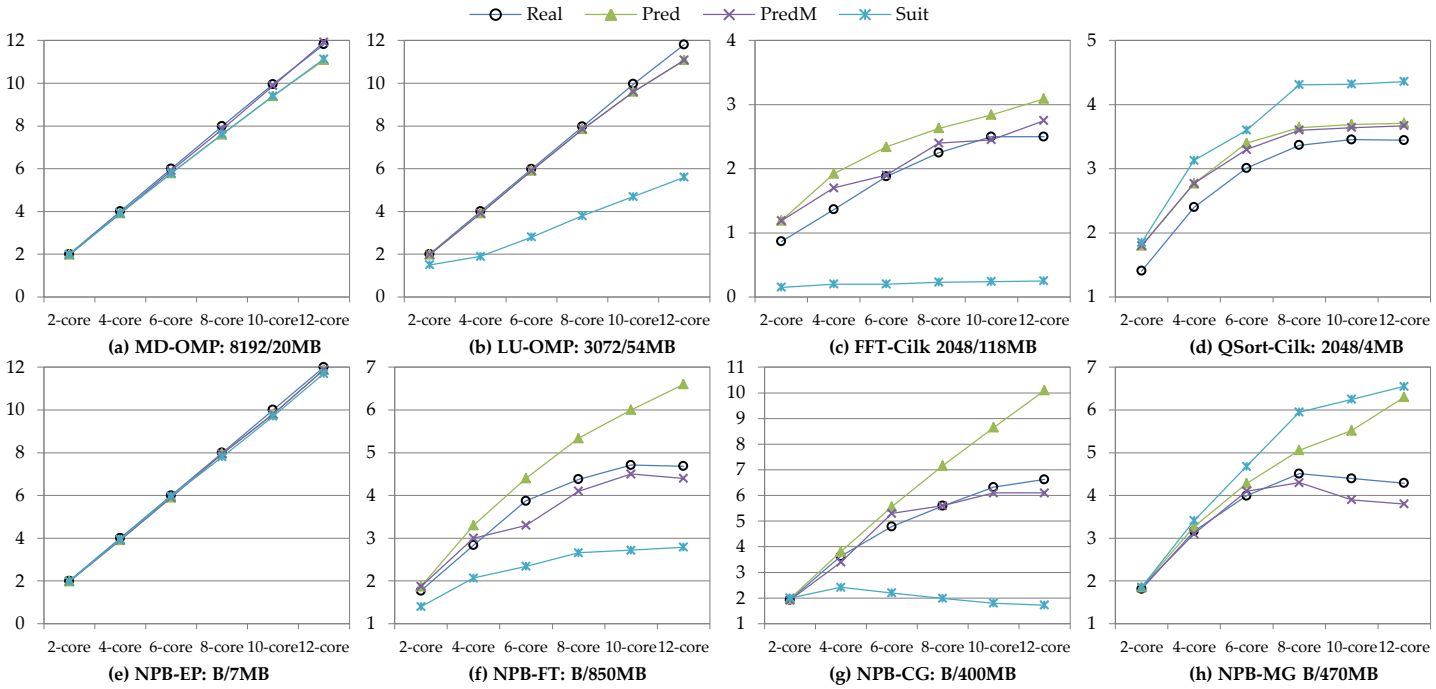

Figure 12. Predictions of OmpSCR and NPB benchmarks: 'Pred' and 'PredM' are the results without/with the memory performance model, respectively. 'Suit' is the results of Suitability. Note that Suitability does not have a memory performance model, and only provides speedups for 2^N CPU numbers. The predictions of Suitability for 6/10/12 cores are interpolated. The captions show the input set and its maximum memory footprint.

Even without considering memory effects, four benchmarks, MD-OMP, LU-OMP, QSort-Cilk, and NPB-EP, show very good prediction results. However, we underestimate the speedups of MD-OMP and LU-OMP on 6-12 cores. This could be the super-linear effects due to increased effective cache sizes. We do not currently consider such an optimistic case. However, our memory model gives burden factors of 1, meaning they are not limited by the memory performance. Suitability was not effective to predict LU-OMP. A reason would be the fact that LU-OMP has a frequent parallelized inner loop, overestimating the parallel overhead.

The burden factors of the other benchmarks (FFT-Cilk, NPB-FT, NPB-CG, NPB-MG) are greater than 1. For example, the burden factors of NPB-FT show the range of 1.0 to 1.45 for two to 12 cores. Some burden factors (e.g., NPB-FT and NPB-MG) tend to be predicted conservatively. FFT-Cilk shows more deviations on the burden factors. However, we believe that the results show the outstanding prediction ability. We also verified the burden factor prediction by using the microbenchmark used in Eqs. (6) and (7). In more than 300 samples that show speedup saturation, we were able to predict the speedups mostly within a 30% error bound.

FFT-Cilk and QSort-Cilk use recursive parallelism that cannot be efficiently implemented by OpenMP 2.0. They are parallelized by Cilk Plus and predicted. As our synthesizer can easily model a different threading paradigm, the results are also pretty close. For the case of FFT-Cilk, Suitability was unable to provide meaningful predictions. However, we found that the tree-traversing overhead did show some variations for these benchmarks. Because the synthesizer actually performs recursive calls, this behavior may introduce hard-to-predict cache effects on the tree-traversing overhead.

D. The overhead of Parallel Prophet

The worst memory overhead in all experimentations is 3 GB when only the loseless compression is used. Note that NPB-FT only requires 5 MB for storing its program tree. The time overhead is generally a 1.1× to 3.5× slowdown *per each estimate*, except for the prediction of FFT with the FF emulator, where the slowdown is more than 30×. Suitability shows 200× slowdowns for FFT. Such high overhead comes from the significant overhead of tree traversing and intensive computation using the priority heap. The synthesizer only shows approximately 3.5× slowdowns for FFT.

Quantifying the time overhead is somewhat complicated. The total time overhead is dependent on how many estimates programmers want. The more estimates desired, the more time needed. In addition, the synthesizer runs a parallelized program. Hence, its time overhead per estimate is dependent on its estimated speedup as well. An estimated speedup of 2.5 would have at least a 1.4× slowdown (=1+1/2.5).

The total time of the synthesizer can be expressed as:

$$T_{SYN} \approx T_P + \sum_{i=1}^{K}(T_T + \frac{T}{S_i}),$$

where T is the serial execution time, T_P is the interval and memory profiling overhead, T_T is the tree-traversing overhead in the emulators, K is the number of estimates to obtain, and S is the measured speedup. Note that T_P and T_T are heavily dependent on the frequency of annotations.

E. Limitations of Parallel Prophet

Parallel Prophet has the following limitations:

- Obviously, a profiling result is dependent on an input.
- We assume no I/O operations in annotated regions.
- Current annotations only support task-level parallelism and mutex. However, supporting other types of synchronization and parallelism patterns is not challenging. For example, pipelining can be easily supported by extending annotations [23] and the emulation algorithm.
- Section V-A discussed the assumptions of our memory model. The assumptions include the current limitations.

VIII. Conclusions and Future Work

We presented Parallel Prophet, which predicts the potential speedup of a parallel program by dynamically profiling and emulating a serial program. It takes an input as annotated serial programs, where annotations describe parallel loops and critical sections. We provided detailed descriptions of two emulation algorithms: fast-forwarding emulation and the synthesizer. In particular, the synthesizer is a novel method to model various parallel patterns easily and precisely. We also proposed a simple memory model to predict the slowdown in parallel applications resulting from memory resource contention.

We evaluated Parallel Prophet with a series of microbenchmarks that show highly accurate prediction results (less than 5% errors in most cases). We also demonstrate how Parallel Prophet executes the prediction with low overhead. Finally, we evaluated Parallel Prophet with a subset of OmpSCR benchmarks and NPB benchmarks, showing good prediction accuracies as well. Our memory model which uses few hardware performance counters predicts performance degradations after parallelization in memory intensive benchmarks.

In our future work, we will improve our memory model to include more complex parallel programming patterns. Combining a static memory address pattern analysis and dynamic profiling could be an option to develop a low-overhead memory profiler.

IX. Acknowledgements

Many thanks to Chi-Keung Luk, John Pieper, Paul Petersen, Neil Faiman, Intel Parallel Advisor Team, Sunpyo Hong, Joo Hwan Lee, other HPArch members, and anonymous reviewers, for suggestions and feedback. We gratefully acknowledge the support of the National Science Foundation (NSF) CAREER award 1139083; Intel Corporation; Microsoft Research and Samsung. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF, Microsoft, Intel, or Samsung.

References

[1] OmpSCR: OpenMP source code repository. http://sourceforge.net/projects/ompscr/.

[2] OpenMP. http://openmp.org/wp/.

[3] Performance Application Programming Interface. http://icl.cs.utk.edu/papi.

[4] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.*, (1):94–136, 2004.

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), 1967.

[6] J. M. Bull and D. O'Neill. A microbenchmark suite for openmp 2.0. In *Proceedings of the 3rd European Workshop on OpenMP*, 2001.

[7] H.-Y. Cheng, C.-H. Lin, J. Li, and C.-L. Yang. Memory latency reduction via thread throttling. In *MICRO-32*, 2010.

[8] V. V. Dimakopoulos, P. E. Hadjidoukas, and G. C. Philos. A microbenchmark study of OpenMP overheads under nested parallelism. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08.

[9] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS*, 2010.

[10] S. Eyerman and L. Eeckhout. Modeling critical sections in Amdahl's law and its implications for multicore design. In *ISCA*, 2010.

[11] S. Garcia, D. Jeon, C. Louie, and M. B. Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *PLDI*, 2011.

[12] J. L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31, May 1988.

[13] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, 2010.

[14] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41, July 2008.

[15] Intel Corporation. Intel cilk plus. http://software.intel.com/en-us/articles/intel-cilk-plus/.

[16] Intel Corporation. *Intel Parallel Advisor*. http://software.intel.com/en-us/articles/intel-parallel-advisor/.

[17] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor. Kismet: Parallel speedup estimates for serial programs. In *OOPSLA*, 2011.

[18] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, NASA, 1998.

[19] A. H. Karp and H. P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33, May 1990.

[20] M. Kim, H. Kim, and C.-K. Luk. SD^3: A scalable approach to dynamic data-dependence profiling. In *MICRO-43*, 2010.

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

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[23] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *MICRO '07*, 2007.

[24] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP '08*, 2008.

[25] P. Wu, A. Kejariwal, and C. Caşcaval. Languages and compilers for parallel computing. chapter Compiler-Driven Dependence Profiling to Guide Program Parallelization, pages 232–248. Springer-Verlag, Berlin, Heidelberg, 2008.

[26] E. Yao, Y. Bao, G. Tan, and M. Chen. Extending amdahl's law in the multicore era. *SIGMETRICS Perform. Eval. Rev.*, 37, October 2009.

[27] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO '09*.