Assessing Safe Task Parallelism in SPEC 2006 INT

Tongxin Bai, Chen Ding, Pengcheng Li
Department of Computer Science
University of Rochester
Rochester, NY, USA
{bai, cding, pli}@cs.rochester.edu

***Abstract*—To migrate complex sequential code to multicore, profiling is often used on sequential executions to find opportunities for parallelization. In non-scientific code, the potential parallelism often resides in while-loops rather than for-loops. The do-all model used in the past by many studies cannot detect this type of parallelism. A new, task-based model has been used by a number of recent studies and shown safe for general loops and functions.**

This paper presents a feedback-based compiler that measures the amount of safe task parallelism in a program and ranks the potential candidates. It solves two problems unique for task analysis. The first is the relation between loop parallelism and function parallelism. The second is the effect of the calling context. The new tool is built in the GCC compiler and used to analyze the entire suite of SPEC 2006 integer benchmarks.

I. Introduction

The proliferation of multithreaded and multi-core systems has propelled research on extraction and utilizing task-level parallelism in general purpose applications. Because of the difficulty of precisely analyzing dependences at compile time, researchers have resorted to exploiting optimistic parallelism through speculation at run time.

Recent advances have enabled speculative parallelization of general-purpose code on conventional machines and made speculation programmable by direct programmer control. The new systems, for examples [9, 16, 2], can safely parallelize programs with tens of thousands of lines of legacy C/C++ code, suggesting that even large programs may be parallelized fully automatically or with minimal programmer involvement. These studies, however, have handpicked test programs. While the results are promising, it is unclear how much benefits there are in other programs.

This paper proposes and evaluates parallelism potential assessment. It models safe task parallelism in the form of *possibly parallel routines*PPRs, supported in software by a system called software behavior-oriented parallelization (BOP) [4]. In this paper, safety means sequential equivalence, that is, a program with safe task parallelism always produces the same output as the program without task parallelism. Syntactically we mark a task using the following hint:

- possibly_parallel{ code } marks a block of code as a possibly parallel routine (PPR) and suggests task parallelism—the PPR block may be parallel with the code after the PPR block.

As an interface, PPR is equivalent to safe future [19] and ordered transactions [18]. It is a safe variant of the future primitive, originated in Multilisp [7] and evolved into spawn in Cilk, future in Java, async in X10 and the newest version of C++, among others. As a safe task, a PPR differs from a parallel loop and a conventional task in several ways:

- *Loop and function parallelism.* A PPR can bracket the body of a loop (a loop task), the body of a function (a function task), a loop as a whole, a call site, or any block of code anywhere in a program.
- *Continuous parallelism.* A PPR is a fork. There is no need to specify when the spawned task has to end. With available processors, tasks may be started continuously [21, 9].
- *Speculative synchronization.* Tasks may have dependences and can be synchronized (by dependence hints [9]) to avoid speculation failure. Rollbacks only happen on unexpected conflicts. Synchronization enables safe pipelining and other types of do-across parallelism.

To find potentially parallel tasks, we analyze the code using a compiler and then analyze its executions using a profiler. To refer to them, we use a acronym, STAPLE, for *safe task parallelism*.

The STAPLE compiler instruments a program to capture all data accesses and the start and the end of all loops function calls when the program executes. The STAPLE profiler uses all-construct profiling, pioneered by Zhang et al. in Alchemist [22]. Alchemist uses the full call tree to keep track of nested loops and function calls. To improve efficiency, it adaptively erases previous execution history. The analysis is independent of the number of processors. Unlike Alchemist, STAPLE uses the calling-context tree (CCT [1]) and maintains the entire tree. The analysis considers all contexts in addition to all constructs. Using all contexts, STAPLE ranks all program constructs based on the potential improvement for the overall program speed.

Trace analysis is costly. Compared to locality profiling, which needs to capture only memory loads and stores, parallelism profiling must track all accesses to all data so not to miss a single dependence. Previously, loop profilers analyze one loop at a time and ignore dependences outside the loop. In task profiling, STAPLE has to consider all dependences in complete executions, including the effect of abnormal control flow such as exceptions, which complicates context tracking. To evaluate the cost and verify our design, we have implemented STAPLE using GCC and tested it on the entire set of SPEC 2006 integer benchmarks (using the train input). The source code may have thousands of nested loops and

recursive functions (as in *GCC* itself) , and the unmodified run time can be over 4 minutes.

Finding parallelism does not mean automatic parallelization. Once candidate code is identified, a programmer may still need to use loop blocking to increase task granularity, add dependence hints to synchronize on known dependences, and protect shared data and monitor their access to guard against unknown conflicts. Efficient dependence marking and program monitoring require one to understand the code being parallelized. The purpose of `STAPLE` is to help with the manual effort. First, a user needs not spend time on code that has little parallelism. In addition, a user can start with the most profitable loops or functions, so to prioritize the use of programming time. Furthermore, a user can use the dependence profile to arrange synchronization between safe tasks to avert unnecessary rollbacks.

The rest of the paper is organized as follows. Section II defines possibly parallel loops and functions and discusses their relation. The next section describes the all-context, all-construct analysis. Section IV describes an empirical study of the analysis cost and outcome. Finally the last two sections review related work and recap.

II. Safe Loop and Function Parallelism

Safe task parallelism comes from two sources: possibly parallel loop tasks and possibly parallel function tasks.

A. Possibly Parallel Loops with Function Calls

A do-all loop is the most common parallel construct, supported in practice by OpenMP and CUDA and in array languages such as SaC. It is the default loop in Fortress. If the iterations have dependences between them, the loop is called do-across. The number of iterations is known before a do-all loop executes.

A possibly do-all loop is created by placing the body of a sequential loop a `PPR`. `PPR` extends do-all in three ways. First, it can parallelize a loop that has unpreditable conflicts and exploit parallelism in loops that are mostly parallel and loops that are parallel only for certain program inputs. Second, it can parallelize a while-loop or a for-loop with an early exit, where the iteration count is unknown until the loop ends, e.g. a search loop that exits upon the first match. Finally, the possibly do-all loop does not require a barrier at the end as a do-all loops does. It permits parallelism between the end of the loop and the post-loop code. Listing 1 shows an example of safe do-all.

Possibly do-all loops are supported by speculative execution, which may be implemented either in hardware or software. Hardware speculation, known as task-level speculation (TLS), is both efficient and transparent to the user. However, since it has to buffer the intermediate states in the limited on-chip memory, the loop body must not access too much data.

A software solution has the opposite constraints. It has to instrument the code and replicate data explicitly, which is costly. But it does not have a size limit. In fact, it needs large loop tasks to amortize the cost of speculation.

```
1 for (i=0; i<N; i++)
2   possibly_parallel { // safe loop parallelism
3    if (a[i] < 0)
4      break          // an early exit requiring speculation
5    else if (a[i] == 0)
6      b[a[i]] = 0              // good for hardware speculation
7    else
8      b[a[i]] = lots_of_work(a[i]) // good for software speculation
9  }
```

Listing 1: A possibly do-all loop. A short-running loop body is required by hardware speculation and a long-running loop body by software speculation.

A consequence is that hardware speculation (TLS) prefers innermost loops while software speculation prefers outermost loops. The implication on parallelism profiling is the role of function calls in a loop. For TLS, if a loop body contains a function call cannot be completely inlined, the loop is probably too large. Most proposed TLS hardware terminates a thread at a function call. For SPEC 2006, the TLS potential is reported to be limited—1% [10] and 60% [13].

We target coarse-grained tasks beyond the limit of TLS in hope of finding additional parallelism. For this we have to analyze loops with function calls.

B. Possibly Parallel Functions in Loops

Function parallelism has many variants, known also by names such as module, procedure, or method-level parallelism. A basic abstraction comes from Multilisp, where a function is executed by spawning a new task (called the *future*) and running in parallel with the code after the future (called the *continuation*) [7]. Additional tasks may be spawned in the future or in the continuation.

A possibly parallel function is a function whose body is folded inside a `PPR` block. Speculation extends future in two ways. Semantically, a `PPR` function is safe. Syntactically, a `PPR` is a fork without a join, unlike the future-type primitives in Cilk, Java, and X10. The join for a `PPR` happens implicitly when the `PPR` finishes. The speculation mechanism ensures the correctness[1].

Listing 2 shows how safe function parallelism helps with loop parallelization. Two loops are shown. In the `main` function, the first loop traverses an array of lists and for each one, calls two functions to process, `sub1`, `sub2`. In `sub1`, the second loop traverses the incoming list. The code also calls `sub2` in the first loop and `sub1` outside it. Their effects will become clear when we later explain mixed function and loop parallelism in Section II-D.

Neither loop is do-all or possibly do-all. In the body of the first loop, the array index is tested at the start and incremented at the end. As a result, the next iteration cannot start until the previous one finishes. A similar loop-carried dependence is caused in the second loop by moving the next pointer. For

[1]by a rollback in the worst case but one can suggest a join using a dependence hint to forestall the rollback [9]

```
1  main( ) {
2      i = 0
3      while ( lists[i] ≠ nil ) {     // not a do-all loop
4          sub1( lists[i]->list )
5          sub2( lists[i]->aux )
6          i++        // because of end-to-start dependence on i
7      }
8      sub1( extra_list )
9  }
10
11 sub1( head ) {
12   possibly_parallel {      // safe function parallelism
13     node = head
14     while ( node ≠ nil ) {
15         sub3( node )
16         node = node->next
17     }
18   }
19 }
20 ...
```

Listing 2: An example program with two loops and one function shown. Neither loop is *do-all* because of the loop-carried end-to-start dependences on `i,node`. To parallelize, `sub1` is made into a possibly parallel function.

ease of reference, we term it *an end-to-start dependence*.[2]

The functions may be parallel. In particular, the calls to `sub1` are parallel if the `lists` array does not contain duplicate entries. The function body is marked as a `PPR`. Such function parallelism enables a loop to run in parallel even if the loop is not do-all or possibly do-all.

C. Alternative methods of loop parallelization

There are other ways of dealing with loop-carried end-to-start dependences such as the ones in Listing 2. A compiler can statically parallelize this and other types of loops in scientific code. It can perform interprocedural analysis to extract parallelism regardless of the number of functions being called in the loop [6]. Similar analysis can select the best loops for parallelization [11]. Compiler analysis cannot parallelize as effectively for general-purpose code written in C/C++ with most of data dynamically allocated in heaps and accessed through pointers. Our method targets programs that cannot be statically parallelized.

Another solution is to combine compiler analysis with TLS run time. A compiler lifts the source of the dependence from the end to the start of the loop body, as done by a number of TLS compilers including SPT [5] and POSH [12] (see Section V for a more complete list). TLS targets innermost loops with no function calls. The full view of the loop body allows a compiler to perform the transformation safely. `STAPLE` targets much longer tasks. A static approach would amount to automatic parallelization of C/C++ programs. Safe task parallelism described here may parallelize loops when static analysis is unable or unavailable.

[2]The exact boundary between loop iterations depends on code generation. GCC generates the test at the end of the loop body. In this case, the loop-carried end-to-start dependence happens between the increment and the function call (parameter passing).

D. Implications for Parallelism Analysis

a) Coarse-grained tasks: `PPR` is supported by a system called `BOP`, which uses processes to implement (speculative) tasks. The forking overhead can be reduced by copy-on-write in threads implemented by a compiler [16] or by reusing a pool of processes (rather than forking one for each `PPR`) [9]. In the latter work, Ke et al. reported large speedup numbers for millisecond-short `PPR` tasks. At this size, it still requires a task to possible execute many functions and loops.

b) Task selection: Both loop and function parallelism may be nested. Nesting is favored for reasons of granularity but makes the analysis more complex. More importantly, it raises the problem of task selection. To parallelize a program with many loops and functions, we have to choose whether to parallelize loops and functions and in each case, choose which loops or functions to parallelize. Speculative systems do not support nested parallelism, so the problem is finding the best single-level parallelism. In the example in List 2, the two loops are nested, so only one can be parallelized. The choice depends on the number of iterations in each loop. In the algorithm we will describe, the one with more iterations will be selected because it utilizes more processors.

c) Mixed loop and function parallelism: A loop may call multiple functions, e.g. `sub1,sub2` in Listing 2, that may be parallel. Function parallelism enables the concurrent execution of different functions rather than loop iterations. In addition, the same function may be called in different contexts, e.g. `sub1` called inside the loop and then outside the loop. The `PPR` designation means that not only loop iterations are parallelized but also the loop is parallel with the `sub1` call after the loop. Mixing of function and loop parallelism is a natural consequence of function parallelism. It has not been considered by past studies on loop parallelization (see the next paragraph), and it may improve parallelism significantly if multiple functions are mixed-in with a parallel loop with a small trip count.

d) All-construct task parallelism: The `PPR` hint can mark any program construct not just a function, e.g. the body of `sub1`. It can mark a loop, a branch, or a statement (including a call site). The effect is the same. Consider the example in Listing 2 one last time. If `sub1` is in-lined, the analysis still considers the second while loop as a candidate `PPR`. As a construct, it does not make a difference whether it is a loop or a function call. For task parallelism, we consider three constructs: the body of each loop, the loop as a whole, and each function call.

III. All-context and All-construct Parallelism

A. Basic Data Structure

The profiling algorithm has three basic tasks: recording the time range of all instances of all program contructs, tracking all dependences across all contexts, and updating the ideal parallel speedup for each construct.

For the first task, the algorithm keeps a progress map and updates it whenever entering and exiting a program construct, e.g. a loop, a loop iteration or a function. It answers the question whether a time point is inside or outside an execution

instance of a construct. A hash table is used to detect dependences. It contains an entry for each memory location used by a program. The entry is updated at each write to reflect the latest memory state. Later reads and writes would trigger a dependence that affects parallelism.

Figure 1 shows the major components and their relation.

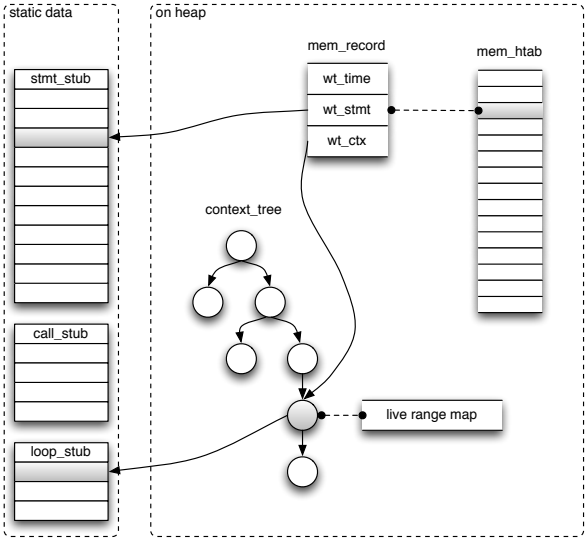


Fig. 1: The components of the profiling framework.

- *Memory record table*, the hash table that maintains the latest states of all the active memory locations. Each memory record contains the latest write time, along with a pointer to the definition context and a pointer to a structure describing the definition statement.
- *Context tree*, a variant of CCT [1] we call *Extended Calling Context Tree* (ECCT, or CCT for brief), in which a node represents either a function call or a loop. During execution, each calling sequence (or loop nest) is uniquely mapped to a tree path starting from the root. To identify dependence, each tree node holds a range map which contains complete or partial time range information for the calling context.
- *Profiling stubs*, compiler allocated space for program constructs. Each stub is dedicated to a single construct. The contents of the stubs are serialized and stored to permanent storage after profiling. When the program is compiled again, the compiler knows how to read back and associate the profile data with each construct.

Consider an example (figure omitted for space), where the j-loop calls function foo, which runs the i-loop, which calls bar. If there is a dependence spanning two instances of bar, the memory record table would detect it and return the time pair for the source and the sink. The context tree, in particular, the range map, differentiates whether the dependence happens inside a j-iteration or across j-iterations.

B. Procedures of CCT-based Profiling

As a program executes, the profiler for the most time handles four types of events: read references, write references, construct enters and construct exits. In this section we describe the procedures and variations when analyzing different types of safe parallel tasks.

Algorithm 1: Procedures taken at construct boundaries

```
   Input: stub_C, pointer to the construct's profiling stub.
   Data: δ_r, the depth of recursion in the current context
 1 enter_construct(stub_C):
 2 begin
 3     if δ_r > 0 then
 4         Increment δ_r, then return
 5     end
 6     if stub_C.active then
 7         Set δ_r to 1, then return
 8     end
 9     if a fresh new context is encountered then
10         Create a new context node and update the CCT
11     end
12     Update the current context pointer
13     Update the schedule table of the current context or the current
       construct
14     Set stub_loop.active
15 end
16
17 exit_construct():
18 begin
19     if δ_r > 0 then
20         Decrement δ_r, then return
21     end
22     Update the schedule table of the current context or the current
       construct
23     Update the critical path of the current context or the current
       construct
24     Unset stub_C.active
25     Change current context to its parent
26 end
27
28 at_loop_latch():
29 begin
30     if δ_r > 0 then
31         return
32     end
33     Update the schedule table of the current context or the current
       construct
34     Update the critical path of the current context or the current
       construct
35 end
```

1) Capturing and Maintaining Contexts: Context tracking has two difficulties: one is abnormal control flow, and the other is recursion. Usually a context change happens when control flow crosses a construct boundary at the end of the scope of program construct. Unusual context changes can happen due to exception handlings and other irregular control flows such as co-routines. In C, they are implemented using `setjmp` and `longjmp`. In our profiling framework, we intercept after each call of `setjmp` and enforce context state recording and recovering according to the return value.

Algorithm 1 outlines the procedures taken at contruct boundaries for maintaining the usual context changes. When control enters a construct, the first step is recursion check. Deep recursion can dramatically increase the size of the context tree without adding a proportional amount of new information. We handle recursion by ignoring all context changes inside a recursive call. To that end we use a global variable δ_r to store the recursion depth, with zero indicating not recursive in the current context. When entering a construct,

Algorithm 2: Procedures at memory read and write

```
   Input: addr, memory access location
   Input: stmt_use, pointer to the statement's profile stub
1  handle_memory_read(addr, stmt_use):
2  begin
3      Get memory record ⟨time_def, stmt_def, ctx_def⟩ by addr
4      ctx ← get lowest common ancestor of ctx_def and ctx_cur
5      while time_def < ctx.range_0.start do
6          ctx ← get parent context of ctx
7      end
8      Update ctx's task schedule
9      Record this dependence information
10 end
11 handle_memory_write(addr, stmt_def):
12 begin
13     Update memory record of addr with ctx_cur and stmt_def
14 end
```

if δ_r is zero, the context is not recursive. If δ_r is positive, the context is recursive. Algorithm 1 simply increments the counter and returns. Symmetrically, leaving a construct inside a recursive context only involves decrementing the recursion level. Detecting recursion is done simply by checking the construct stub's active bit, with 1 indicating the construct has been entered but not yet fully exited. Therefore, entering a construct whose active bit is 1 implies entering a recursive context.

The previous system, Alchemist, used a run-time call tree to store contexts and dynamically trimmed the tree to save space [22]. In comparison to the call tree, CCT has less memory consumption and a lower overhead in memory allocation and reclamation. When a context is repeated, its change involves only a pointer update. In addition, Alchemist used binary rewriting. `STAPLE` uses a compiler, which can more precisely track program constructs especially in the presence of exceptions.

2) Computing the Parallel Speedup: `STAPLE` quantifies parallelism by counting the number of run-time instructions in a task. The model is efficient and architecture neutral, and the result easy to interpret. It does not include the overhead of spawning and monitoring and the complexity of modern processors and memory hierarchy, needed when targetting `STAPLE` for a specific machine.

To start, let's consider a pair of consecutive run-time instances of a construct. The maximal overlap is given by the longest cross-instance dependence, as characterized by Alchemist. The Alchemist distance, when cumulated across all instances, give the shortest time to execute a construct. We use a more elaborate method. For each instance, we schedule it at the earliest possible time based not just on the Alchemist distance but also the next available processor. In this way we can compute speedups for all processor counts (which Alchemist does not do).

To determine scheduling delays caused by a inter-task data dependence, we need to quickly find all the constructs to which the dependence may affect. We will evaluate three task models. In addition to parallel loops and functions, we also consider parallel loops that have a barrier at the end as do-all loops do. With the barrier, we just need to consider a single context—the loop. In continuous loop parallelism, we also consider all related nesting constructs and compute the effect of a dependence on all of them. Algorithm 2 outlines how to locate related contexts for a data dependence by finding the *Minimum Covering Context*.

Definition 1: Minimum Covering Context. For a given time range, a covering context is one that has a live range that covers it. A minimum covering context of a time range is the covering context with the smallest covering live range.

In the example illustrated in Figure 2, the dependent read and write are in context $c5$ and $c6$ respectively. The write time is 20751. For any context that covers both the read and the write, it must cover both the read context and the write context. According to Algorithm 2 the common ancestor $c4$ is reached. Since $c4$'s late live range is [21000, -) which doesn't cover the dependence range, $c4$'s parent $c3$ is then checked. $c3$'s late live range is [20000, -) which covers the dependence so the algoritm determines that $c3$ is the minimum covering context of the dependence.

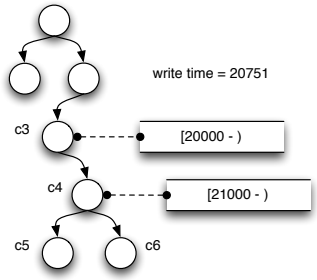


Fig. 2: Illustration of the minimum covering context

When modeling function tasks and continuous loops, we need to traverse up the CCT from the defining context, updating all the constructs along the path until an active construct is found (algorithm omitted for lack of space). Intuitively, this type of parallelism for all constructs should take more time to profile than loop parallelism. If k is the average number of steps that each dependence seeks from the defining context to the minimum overing context, then the function profiler takes k times as many steps as its loop counterpart, not to mention that each record searching may take longer if there are more function contexts than loop contexts. For a clean comparison, we have implemented the function and loop profilers seperatedly to compare their speed. The results are not as simple as the intuition predicts, as we will show in Section IV-B.

The profiler produces two types of outputs: the context level results and the construct level results. Constructs are ranked by the aggregate speedup for use as a programming aid, which are results we will show in the evaluation. Context profiles are displayed on the CCT, which are used by both the compiler and users in context-aware transformation and in the selection of possible parallel loops and functions. A simple bottom up analysis on the CCT to compute the aggregated speedup (algorithm omitted for lack of space).

IV. Evaluation

A. Implementation and Experimental Setup

The STAPLE compiler is a customized version of GCC (version 4.3.2). The instrumentation is implemented as a GIMPLE pass after all the GIMPLE optimization passes. Since the STAPLE instrumentation precedes code generation, there's no need to track register dependences as required if instrumentating at the binary level (e.g. using Pin or Valgrind). Besides instrumenting references and regular control flow edges, the STAPLE compiler also inserts special function calls to handle the side effects of frequently used standard library functions. These library functions include common IO API's (read, write, fread, fwrite, etc), common string operations (strcpy, strncpy, strcat, memcpy, memset, etc), memory allocation (calloc), and irregular control flows (setjmp, sigsetjmp, longjmp, siglongjmp). Memory references by the library calls are treated as a sequence of memroy references. Each IO operation is treated as a memory read followed by a write to virtual address 1. Hence, all IO operations are serialized.

The profiling is a C library. To reduce false sharing in dependence detection, the STAPLEś memory hash table differentiates memory locations at byte granularity. To balance between efficiency and space, STAPLE allocates a 1M-entry pool at a time, and can support up to 512 pools for the memory hash table.

SPEC2006 was created for testing the performance of modern computer systems with ever growing computation (up to tens of minutes long execution) and resource demands (hundreds of megabytes data on average) . The full suite includes 12 integer applications and 17 floating point applications. We choose the integer applications for evaluation because they are difficult to parallelize by a compiler and hence a major target for safe speculative parallelization.

We profile all 12 integer benchmarks in SPEC 2006, using machines with identical hardware and system software configuration, each equipped with an Intel Core i5 processor running at 3.3GHz and 4G DRAM and running Fedora Core 15, with Linux kernel version 2.6.40.3 and configured to have a 32bit address space. Exceptions are with the benchmarks 458.sjeng and 473.astar, for which we have to use an Intel x86_64 machine with 12G memory. To correctly locate source information and avoid parallelism loss due to compiler optimization, we compile the benchmarks with no optimization (-O0) then link with the STAPLE profiler compiled with -O2. All profiling results are shown for the train input.

B. Profiling Costs

Table I shows the loop and function counts. Statically, the programs have up to 2,243 loops (*gcc*) and 9,100 functions (*xalancbmk*). The profiling runs have up to 2.1 billion loop instances (*h264ref*) and 6.1 billion function calls (*omnetpp*). The programs do not uniformly favor loops or functions. *bzip* has 3 times as many loops as function calls, while *xalancbmk* has 15 times more functions than loops.

The execution and profiling times are shown in Table II. The base run times, shown in the second column, range from 1.52 seconds in *gcc* to nearly 5 minutes in *xalancbmk*. The average slowdowns for the 3 analysis— loops with the loop-end barrier (LOOP-B), loops without the barrier (LOOP), and functions—are 50, 102 and 53 respectively. LOOP-B represents the traditional do-all loop, with a barrier at the end. PPR loops do not use a barrier and naturally exploits nested loop parallelism. Profiling nested parallelism is twice as costly as no nesting. Overall, the highest slowdown is 173 in *mcf* and the lowest is 18 in *xalancbmk*.

BENCHMARK	LOOP COUNT		FUNCTION COUNT	
	STATIC	DYNAMIC	STATIC	DYNAMIC
400.perlbench	362	99177840	358	318066421
401.bzip2	184	171995584	62	56444929
403.gcc	2243	29156895	2234	44610523
429.mcf	47	366863062	22	250027726
445.gobmk	867	36372738	1061	13828145
456.hmmer	83	121326995	70	121760365
458.sjeng	113	1131329	68	288484
462.libquantum	44	9493514	61	13902621
464.h264ref	574	2060349291	319	6113116078
471.omnetpp	131	280040303	504	6098676787
473.astar	104	641930138	128	6017477219
483.xalancbmk	658	89624773	9100	1425328795

TABLE I: Benchmark statistics

BENCHMARK	BASE	LOOP-B	LOOP	FUNCTION
400.perlbench	7.80	406.43	812.55	329.70
401.bzip2	12.55	792.22	1813.91	680.16
403.gcc	1.52	51.22	122.25	72.14
429.mcf	20.94	3356.65	4346.48	3163.92
445.gobmk	8.94	349.10	908.20	391.72
456.hmmer	204.71	8027.94	21612.74	3895.29
458.sjeng	249.82	17543.57	16214.98	16668.49
462.libquantum	6.22	151.06	601.48	260.41
464.h264ref	239.68	12812.81	50053.03	13002.02
471.omnetpp	139.97	3224.81	7020.68	5957.68
473.astar	93.45	2568.55	4305.27	4453.88
483.xalancbmk	290.34	2759.39	6324.71	6842.56

TABLE II: Benchmarks' profiling running times (in seconds)

The statistics of the calling context tree is shown in Table III. The largest data structure in the profiler is the memory hash table (column 2) (two programs had over 6GB tables and had to run on 64-bit machines). The number of CCT nodes (columns 4 and 6 for loops and functions) ranges from tens to millions of nodes. Most benchmarks have fewer than 1000. The programs are roughly balanced: seven have more function contexts than loop contexts, while the other five have the reverse. This again shows that integer code is not always function intensive.

Column 3 shows the numbers of run-time dependences, ranging from 650 million to 223 billion. Column 5 and 7 shows the average number of CCT walk-up steps needed for each dependence. Interestingly, for most cases, there's no need to lookup parent context in loop profiling, which implies that the majority of dependences do not cross loop boundaries. In contrast, function dependences have on average 2 to 4 CCT walk-ups, indicating that most data uses in a function are for variable defined outside the function. It is interesting that function profiling, although performing more CCT operations, is on average twice as fast as loop profiling (53 vs. 102) as shown in Table II.

Benchmark	TabSz (MB)	Deps (billion)	CCT(L)	Seeks(L) per dep	#CCT(F)	Seeks(F) per dep
perlbench	456	5.35	965	0.25	7067	4.09
bzip2	480	6.93	300	0.58	102	1.91
gcc	456	0.65	13987	0.16	32137	3.36
mcf	1440	6.17	53	0.94	24	2.14
gobmk	96	6.14	1211338	0.17	280120	2.78
hmmer	24	125.46	123	0.16	92	1.50
sjeng	6904	164.90	380	0.07	431	1.30
libquantum	72	4.60	93	0.08	336	2.81
h264ref	408	223.81	870	0.09	815	2.60
omnetpp	624	54.72	871	0.54	7753	3.75
astar	6744	36.65	145	0.89	505	2.68
xalancbmk	840	77.46	1501	0.23	70261	3.02

TABLE III: Calling context tree statistics

Benchmark	Effective Amount						Speedups					
	Loops				Functions		Loops				# Functions	
	sgl-w/	sgl-w/o	nest-w/	nest-w/o	w/	w/o	sgl-w/	sgl-w/o	nest-w/	nest-w/o	w/	w/o
perlbench	2	2	4	4	8	8	1.07	1.08	1.07	1.07	1.08	1.09
bzip2	3	3	4	4	5	5	1.11	1.11	1.10	1.10	1.16	1.18
gcc	0	3	0	5	6	7	0	1.07	0	1.08	1.07	1.09
mcf	7	7	9	10	3	4	1.28	1.31	1.31	1.28	1.23	1.26
gobmk	0	0	0	0	7	10	0	0	0	0	1.14	1.11
libquantum	5	5	5	5	11	11	1.35	1.36	1.36	1.36	2.55	2.55
h264f	4	5	6	11	5	6	1.35	1.54	1.34	1.47	1.08	1.43
omnetpp	1	1	1	1	2	2	1.17	1.17	1.17	1.17	1.10	1.10
astar	6	8	6	8	4	8	1.20	1.17	1.29	1.25	1.21	1.17
hmmr	2	3	2	3	0	1	2.88	2.93	3.69	3.51	0	52.48

TABLE IV: Average whole-program speedups from parallelizing single loops and functions. The effective amount shows the number of loops and functions whose overage is over 5% and speedups are over 1.05. Each loop has 4 types: single-level parallelism with and without false dependences (denoted by **sgl-w/** and **sgl-w/o**) and nested parallelism with and without false dependences(denoted by **nest-w/** and **nest-w/o**). Each function has two types to show just the effect of false dependences(denoted by **w/** and **w/o**).

C. Loop and Function Parallelism

We show the results of STAPLE for two types of constructs: loops and functions. The coverage of a loop is the total length of all its instances. The coverage of a function is the portion of the smallest continuous execution window that contains all its run-time instances. The difference is that loop coverage does not include the gap between loop instances, but the function coverage does. The parallel time is measured by running loop iterations and function instances in parallel assuming no parallel or speculation overhead. The speedup is the sequential (coverage) time divided by the parallel time. In this section we use results from 256 processors and will show other results in Section IV-E. We show the overall speedup—the improvement of each loop or function on the whole-program finish time.

Table IV show all loops and functions with coverage over 5% and a minimal (overall) speedup of 1.05, where the left part shows the total amount and the right part shows the average whole-program speedups. *gobmk* has no qualifying loop so it has only data for function parallelism. *sjeng* and *xalancbmk* have little parallelism since nothing qualifies, not shown in Table IV. Even in average, the speedups are obvious, up to 3.69 times in *hmmr*, over or near 1.5 times in {*h264ref, libquantum, mcf, astar*} and around 1.1 times in the remaining.

Figure 3 shows maximum whole-program speedups of parallelizing loops and functions. The overall speed of 7 out of 12 tests can be improved significantly by single-loop parallelization, in particular, over 2 times in {*hmmer, libquantum, mcf*}, over or near 1.5 times in {*h264ref, astar*}, around 1.2 times in {*bzip2, omnetpp*}.

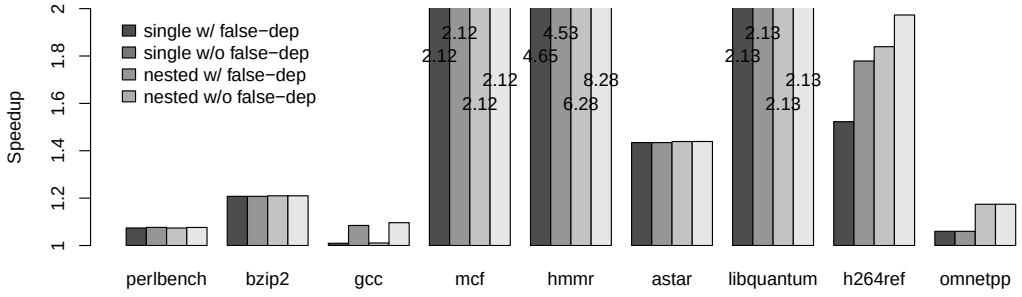
The overall speed of 10 out of 12 tests can be improved significantly by single-function parallelization, more specifically, over 50 times in *hmmer*, 12 times in *libquantum*, near 2 times in *h264ref*, over or near 1.5 times in {*mcf, astar*}, over 20% in {*gcc, bzip2, gobmk*}, and over 10% in {*perlbench, omnetpp*}.

Assuming our test set is representative of integer code, we observe that both loop and function parallelism are important. Neither one dominates the other. Function parallelism improves over do-all loops (as in *hmmer* and *libquantum*) and enables parallelization when loops cannot be parallelized (most notably *gobmk*).

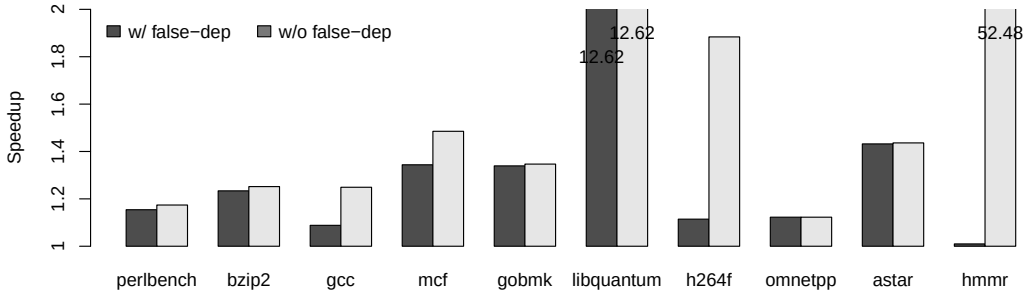
D. Continuous Speculation and False Sharing

The loop has four corresponding types of data in Table IV and four bars in Figure 3. The single-level parallelism is for loops with an end barrier, and the nested parallelism no barrier. A barrier permits parallelism for this loop alone. Otherwise, the parallelism includes those of its outer loops, named nested-loop parallelism. Single-loop parallelism subsumes innermost loop parallelism. For the loop, the first type shows parallelism limited by all dependences, while the second type only flow (read-after-write) dependences. The second case is more than a theoretical interest. Many current speculation systems can already remove all false dependences via data copy-on-write in each task.

From Table IV, of the 9 programs that have significant loop parallelism, 7 benefit from removing the loop barrier, nearly doubling the parallelism for the highest performing loop in *hmmer*. Removing false dependences is no less dramatic,



(a) The maximum speedup (for the entire prorgram) from parallelizing a single loop



(b) The maximum speedup (for the entire program) from parallelizing a single function

Fig. 3: Maximum whole-program speedups from parallelizing single loops and functions. Each loop has 4 bars: single-level parallelism with and without false dependences and nested parallelism with and without false dependences. Each function has two bars to show just the effect of false dependences. The numbers beyond scale are marked in bars.

enabling parallelization for all 5 loops in *gcc* (small improvements), the largest loop in *hmmer* and 3 of the 11 loops in *h264ref*. For function parallelism, removing false dependences is essential for *hmmer* and also enables the highest single-function speedup in *gcc*, *mcf* and *h254ref*.

Interestingly, only 2 loops in *h264ref* benefit non-trivially from removing both the barrier and false dependences. We conjecture that outermost loops (no benefit from nested parallelism) are often the ones to have false dependences, while inner loops often do not.

E. Scalability

Using all context information, `STAPLE` can measure parallel speedup for different number of processors (compared to Alchemist, which measures only the maximal speedup [22]). For a subset of programs, Figure 4 shows two graphs for each program: one for loop scaling and the other for function scaling. Instead of the overall speedup in bar graphs as in Figure 3, here we show self speedups in line graphs.

First, we see that in *libquantum* loops are more scalable than functions, but in *hmmer*, the reverse is true, consistent with our previous observation that loops are not always better than functions for parallelization. Also clear is that the scaling results help to direct parallelization for a specific processor count. For example in *libquantum*, the previous bar graph shows that the function muin can give a factor of 12 overall speedup with 256 processors. From the line graph, we see that half of the improvement can be obtained with 8 processors, making it a good candidate for current multicore systems. The self speedup should be read in correlation with the overall speedup. For example in *hmmer*, the most scalable loop, loop 55, has a maximal speedup of over 100. The overall effect is small due to its 10% coverage. While 10% is good benefit, it is far more effective to parallelize function P7Viterbi, which can improve overall performance by a factor of 50 (shown in the bar graph) and has near perfect scaling (shown by the line graph). We are also measuring the maximal speedup and plan to quantify the relation between program parallelism and the available processors using a concept called smoothability [15].

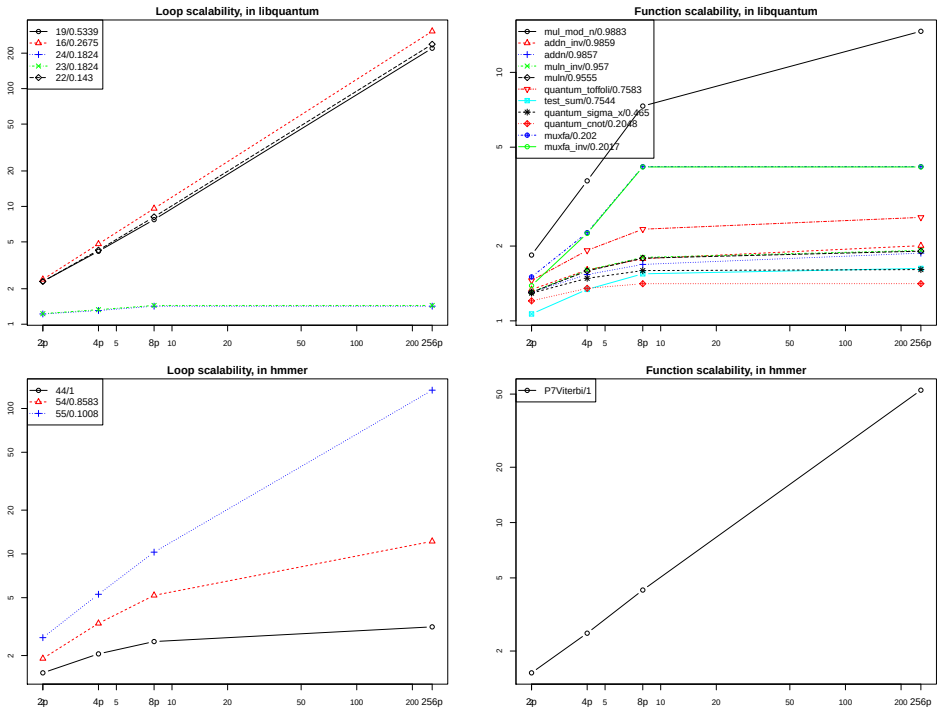


Fig. 4: Scalability (i.e. smoothability) shown by *self* speedups for individual loops and functions in 2 benchmarks.

V. Related Work

a) All-construct profiling: To find parallelism, most of profiling techniques analyze innermost loops (discussed next) or selected units such as phases [4]. Alchemist analyzes all program constructs. It defines what we call the *Alchemist distance* to quantify the amount of parallelism between a program construct and its continuation [22]. It ranked all program constructs to find the most profitable and enabled a set of programs to be hand parallelized for the first time.

Alchemist uses the call tree to track all program constructs. While the full size of the call tree is proportional to the length of the execution, Alchemist keeps only the part enough to fully overlap the execution of the construct being analyzed and its continuation and produces the maximal speedup as well as the statistical average of the Alchemist distance. Instead of taking the call tree and trimming it as in Alchemist, `STAPLE` uses the calling-context tree. It keeps either full records or a fixed number of records so to estimate the parallel speedup more accurately. It can simulate the scheduling of k-processor runs with at most k records per tree node and without tree trimming. Like Alchemist, our tool ranks all program constructs but on the complete parallelism information. Perhaps more significant especially for SPEC 2006 is the speed. Calling-context tree is smaller and less dynamic. `STAPLE` uses GCC instead of a binary rewriter to track data accesses more efficiently and execution contexts more robustly. It may be extended to insert `PPR`s atomatically into a program.

b) Loop profiling by a compiler: Du et al. used dependence profile in the SPT tool in the ORC compiler to select loops for speculative parallelization [5]. Dependences were used to divide a loop body into a serial region and a parallel region. Based on a cost model, the partitioning algorithm minimizes the misspeculation overhead for a given parallel-region size. The problem of loop partitioning was subsequently addressed by Quinones et al. in the Mitosis compiler (based on ORC) using a value prediction technique called p-slices to improve parallelism [14], by Liu et al. in the POSH compiler using static information [12], and by Vachharajani et al. in the SpecDSWP compiler to partition a loop for pipelining [17]. A complementary technique was developed by Wu et al. in the IBM XL compiler to select the most profitable loops for speculation [20]. A similar compiler framework was used by Chen et al. in the ORC compiler to guide speculative optimization [3].

These techniques consider only loops, and with the exception of the IBM XL compiler, only innermost loops, excluding those that are too large or too small. The execution before and after the loop is excluded from analysis. As a result, the analysis does not consider all dependences in the execution or the context beyond a single loop.

Profile-driven analysis requires high efficiency and robustness to be able to handle large benchmark code. Most of the compiler techniques predates the release of the SPEC 2006 benchmarks. The most capable technique, the Mitosis p-slice insertion, analyzed the smallest tests, the Olden programs [14]. The ORC compiler analyzed all SPEC 2000 benchmarks except for two (for lack of C++ library and system call supports) [5]. The IBM study evaluated a subset of SPEC 2006 programs and did not consider while-loops (due a limitation in the implementation) [20]. A recent tool called Kismet accurately estimates the potential parallel speedup in sequential code but does not consider speculative parallelism (so finding little potential in SPEC 2000INT) [8]. There had not been a characterization for both loop and function parallelism in

SPEC 2006.

c) Loop profiling by a simulator: We consider it a simulation if the analysis does not record a dependence graph among program statements. Indeed, most techniques focus on loops that are do-all loops or loops that can made into do-all by hardware-based techniques in particular speculation and value prediction. Kejariwal et al. measured the potential of control speculation, data dependence speculation, and data value speculation and found that the maximal effect computed by the geometric mean is less than 1% for SPEC 2006 [10]. The modest improvement has two reasons. Many of the do-all loops can be parallelized by the Intel compiler without the need of speculation. In addition, they observed that "the total loop coverage in many integer applications is quite low." Their study considered only innermost loops. A compiler can enable additional parallelism by partitioning a loop into a serial and a parallel part. Equipped with such a compiler, Packirisamy found that SPEC 2006 could benefit more from speculative parallelization and obtain an average speedup of 60% with four cores [13].

VI. Summary

We have developed all-context, all-construct parallelism analysis and used it to identify safe loop and function parallelism in SPEC 2006 integer applications. The results show mixed characteristics in integer programs—they do not always have more functions than loops nor have they more function contexts than loop contexts. All programs have a significant amount of parallelism in loops and functions. Loops are no more scalable to parallelize than functions. In the absence of speculation and scheduling overheads, we found that single-loop parallelization can improve overall performance by 20% or more in 7 programs, and single-function parallelization by 10% or more in 10 out of 12 programs. The new tool can help programmers to parallel large sequential code to utilize the spare cores on today's multicore machines.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of PLDI*, pages 85–96, 1997.

[2] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *Proceedings of OOPSLA*, pages 81–96, 2009.

[3] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C. Yew. Data dependence profiling for speculative optimizations. In *Proceedings of CC*, pages 57–72, 2004.

[4] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of PLDI*, pages 223–234, 2007.

[5] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of PLDI*, pages 71–81, 2004.

[6] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in SUIF. *ACM TOPLAS*, 27(4):662–731, 2005.

[7] R. H. Halstead. MULTILISP: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, Oct. 1985.

[8] D. Jeon, S. Garcia, C. M. Louie, and M. B. Taylor. Kismet: parallel speedup estimates for serial programs. In *Proceedings of OOPSLA*, pages 519–536, 2011.

[9] C. Ke, L. Liu, C. Zhang, T. Bai, B. Jacobs, and C. Ding. Safe parallel programming using dynamic dependence hints. In *Proceedings of OOPSLA*, pages 243–258, 2011.

[10] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using SPEC CPU 2006. In *Proceedings of PPoPP*, pages 215–225, 2007.

[11] S.-W. Liao, A. Diwan, R. P. B. Jr., A. M. Ghuloum, and M. S. Lam. SUIF Explorer: An interactive and interprocedural parallelizer. In *Proceedings of PPoPP*, pages 37–48, 1999.

[12] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a tls compiler that exploits program structure. In *Proceedings of PPoPP*, pages 158–167, 2006.

[13] V. Packirisamy, A. Zhai, W.-C. Hsu, P.-C. Yew, and T.-F. Ngai. Exploring speculative parallelism in spec2006. In *Proceedings of ISPASS*, pages 77–88, 2009.

[14] C. G. Quiñones, C. Madriles, F. J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of PLDI*, pages 269–279, 2005.

[15] K. B. Theobald, G. R. Gao, and L. J. Hendren. On the limits of program parallelism and its smoothability. In *Proceedings of MICRO*, pages 10–19, 1992.

[16] C. Tian, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of PLDI*, pages 62–73, 2010.

[17] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of PACT*, pages 49–59, 2007.

[18] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *Proceedings of PPoPP*, pages 79–89, Mar. 2007.

[19] A. Welc, S. Jagannathan, and A. L. Hosking. Safe futures for Java. In *Proceedings of OOPSLA*, pages 439–453, 2005.

[20] P. Wu, A. Kejariwal, and C. Cascaval. Compiler-driven dependence profiling to guide program parallelization. In *Proceedings of the LCPC Workshop*, pages 232–248, 2008.

[21] C. Zhang, C. Ding, X. Gu, K. Kelsey, T. Bai, and X. Feng. Continuous speculative program parallelization in software. In *Proceedings of PPoPP*, pages 335–336, 2010. *poster paper*.

[22] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of CGO*, pages 47–58, Washington, DC, USA, 2009. IEEE Computer Society.