On the Performance Potential of Different Types of Speculative Thread-Level Parallelism

Arun Kejariwal[†] Xinmin Tian[‡] Wei Li[‡] Milind Girkar[‡] Sergey Kozhukhov[*] Hideki Saito[‡]
Utpal Banerjee[‡] Alexandru Nicolau[†] Alexander V. Veidenbaum[†] Constantine D. Polychronopoulos[§]

[†]Center for Embedded Computer Systems
University of California at Irvine
Irvine, CA, USA

[‡]Intel Compiler Labs
Intel Corporation
Santa Clara, CA, USA

[*]Intel Compiler Labs
Intel Corporation
Novosibirsk, Russia

[§]Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, IL, USA

ABSTRACT

Recent research in thread-level speculation (TLS) has proposed several mechanisms for optimistic execution of difficult-to-analyze serial codes in parallel. Though it has been shown that TLS helps to achieve higher levels of parallelism, evaluation of the unique performance potential of TLS, i.e., performance gain that be achieved only through speculation, has not received much attention. In this paper, we evaluate this aspect, by separating the speedup achievable via true TLP (thread-level parallelism) and TLS, for the SPEC CPU2000 benchmark. Further, we dissect the performance potential of each type of speculation — control speculation, data dependence speculation and data value speculation. To the best of our knowledge, this is the first dissection study of its kind. Assuming an oracle TLS mechanism — which corresponds to perfect speculation and zero threading overhead — whereby the execution time of a candidate program region (for speculative execution) can be reduced to zero, our study shows that, at the loop-level, the upper bound on the arithmetic mean and geometric mean speedup achievable via TLS across SPEC CPU2000 is 39.16% *(standard deviation* = 31.23*) and* 18.18% *respectively.*

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques*; D.1.3 [**Software**]: Programming Techniques—*parallel programming*

General Terms

Performance, Measurement

Keywords

Performance evaluation, speculative execution, control dependence, data dependence, value dependence, DOALL loops

1. INTRODUCTION

Speculative execution model has been proposed as a means of achieving higher levels of parallelism. Its roots can be traced back to the early work in branch prediction [1] and its use for global microcode compaction [2].[1] Thereafter, based on the speculative execution model, several techniques have been proposed for extracting a higher degree of instruction-level parallelism (ILP) [4]. With the emergence of multithreaded processors, researchers have proposed to use the speculative execution model for extracting a higher degree of thread-level parallelism (TLP) [5]. Three different types of thread-level speculation (TLS) have been proposed to further enhance the performance gain achievable via multithreading, viz., (a) control speculation (CS) [1]; (b) data dependence speculation (DDS) [6]; and (c) data value speculation (DVS) [7] (these will be discussed further in Section 2).

While TLS has several advantages such as latency tolerance [8], it comes at the cost of misspeculation; furthermore, it also requires additional hardware resources.[2] Clearly, in case of parallel tasks such as iterations of a DOALL loop [9], it is more profitable to execute them on different threads non-speculatively (as there is nothing to speculate on). In such cases TLS is not applicable or is not necessary. This gives rise to a need to differentiate the performance gain achievable via true TLP (which corresponds to threaded execution of parallel tasks) from the one achievable via TLS. Lack of this differentiation in the existing literature makes it difficult to assess the true potential of TLS. Further, in prior works the different types of TLS are not evaluated *standalone*, e.g., while evaluating a DVS scheme, support for CS and DDS is implicitly assumed. In this paper, we remedy the above by carefully dissecting the performance potential of true TLP and TLS and of different types of TLS. Although TLS has been proposed at both the loop level and the procedural level, we shall restrict our evaluation of TLS to loops owing to space limitations.

The main contributions of the paper are as follows:

- ❏ First, we present a loop-level characterization of the entire SPEC CPU2000 benchmark suite. To our knowledge, this is the first characterization of this suite. Unlike previous attempts to characterize other applications which were simulation based, we obtain the loop coverage, defined as the percentage of the total execution time spent in the loops, by first instrumenting the code of each application with hardware performance

This work was supported in part by the National Science Foundation under grants NSF CCF-0311738 and CNS-0220069.

ICS'06, June 28-30, Cairns, Queensland, Australia.
Copyright (c) 2006 ACM 1-59593-282-8/06/0006 ...$5.00.

[1] A large amount of work has been done in the context of speculative computation and related fields. Due to space limitations, we shall only reference the early works of each in this paper. However, a detailed set of references of each is provided in [3]. Based on the context, pointers to the sections, containing the relevant references, in [3] shall be provided.

[2] The existing techniques for the three types of TLS require architectural/microarchitectural enhancements.

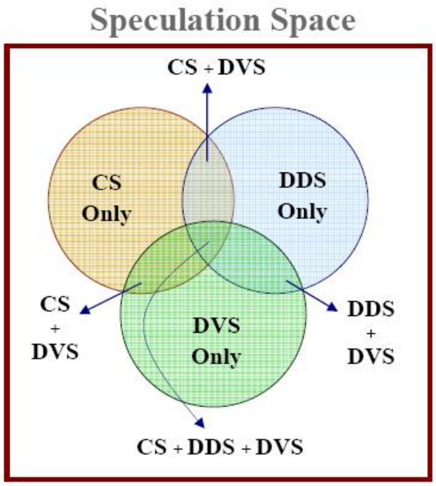


Figure 1: Thread-level Speculation Space

counters and then executing it on a real machine (2.8 GHz Intel$^{®}$ Prescott Processor) with the reference data sets.

- Second, we present a taxonomy of speculative execution wherein we cleanly separate different types of speculation, e.g., with the help of code snippets from SPEC CPU2000 we differentiate between DDS-Only cases (i.e., cases in which only DDS is required for exploiting speculative TLP) from DDS+DVS cases (i.e., cases in which both DDS and DVS are required for exploiting speculative TLP). We believe that this classification will help in the development of better evaluation methodologies for future research in TLS.

- Third, using the Intel compiler and manual analysis, we evaluate the extent of true TLP (at the loop-level) in each application of SPEC CPU2000. This is particularly useful as it yields an upper bound (obtained by filtering the above from the total loop coverage) on the amount of speedup achievable by TLS.

- Fourth, we present an evaluation of the performance potential of each type of speculation (independent of the schemes being used for the same) in two ways: standalone as well as in conjunction with other types of speculation. To the best of our knowledge, such a dissection is the first of its kind. Interestingly, in some cases a particular type of TLS has higher performance potential than CS+DDS+DVS. For example, in `189.lucas` DDS standalone has the highest performance potential.

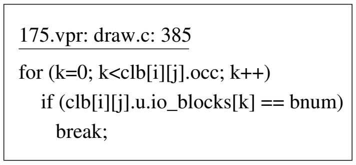
The rest of the paper is organized as follows: Section 2 presents the taxonomy of speculative execution. An overview of the applications in SPEC CPU2000 is given in Section 3. Evaluation of the performance potential of the different types of thread-level speculation models is presented in Section 4. Previous work is discussed in Section 5. Finally, in Section 6 we conclude with directions for future work.

2. TAXONOMY OF SPECULATIVE EXECUTION

In this section, with the help of code snippets from applications in SPEC CPU2000, we discuss the different types of TLS. The primary objective of this section is to distinguish between the instances where a given type of speculation, (say) CS, is useful standalone for speculative parallel execution vs. the instances where it is useful only when applied in conjunction with other types of speculation, (say) CS+DDS or CS+DVS or CS+DDS+DVS. For better understanding, let us consider Figure 1 wherein the the overall thread-level speculation space is represented by the bounding box. In the figure, the regions corresponding to the CS/DDS/DVS-Only labels represent the cases in which a program region can be speculatively parallelized using a TLS approach of the corresponding speculation type in a standalone fashion. In contrast, the regions corresponding to the X+Y label, where (X,Y) $\in$ {(CS,DDS),(CS,DVS),(DDS,DVS)}, represent the cases in which a program region can be speculatively parallelized *iff* TLS approaches of both types, X and Y, are applied. Similarly, the region corresponding to the label CS+DDS+DVS represents the cases in which a program region can be speculatively parallelized *iff* approaches of all the three types of TLS are applied.

2.1 Control Speculation

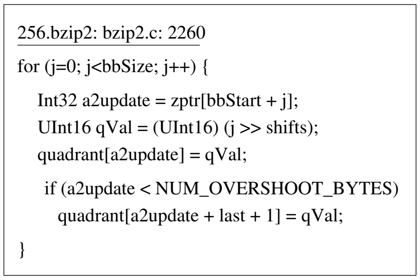
Control speculation (or branch prediction) has been proposed for long to exploit parallelism beyond basic blocks [1]. A large amount of work has been done in context of CS-driven exploitation of higher levels of ILP for both VLIW and superscalar processors (for related work, see [CM1]–[CM11], [BP1]–[BP62] in [3]). Similarly, several schemes have been proposed to use CS to achieve higher levels of TLP (see [CS1]–[CS5] in [3]). Given a loop with conditionals, the execution path is predicted for each iteration and then the iterations are mapped on to the different threads. A thread detects a control violation if the control flow of an earlier thread causes it to be not executed at all (due to an early exit) or on control misspeculation. For example, the iterations of the loop shown in Figure 2 can be executed in parallel on different threads with control speculation only, as there does not exist any loop-carried dependence [10]. We classify such cases under the CS-Only category, refer to Figure 1.



```
175.vpr: draw.c: 385
for (k=0; k<clb[i][j].occ; k++)
    if (clb[i][j].u.io_blocks[k] == bnum)
      break;
```

Figure 2: A candidate loop (with no loop-carried dependence) for CS

However, CS alone does not facilitate full speculative par-

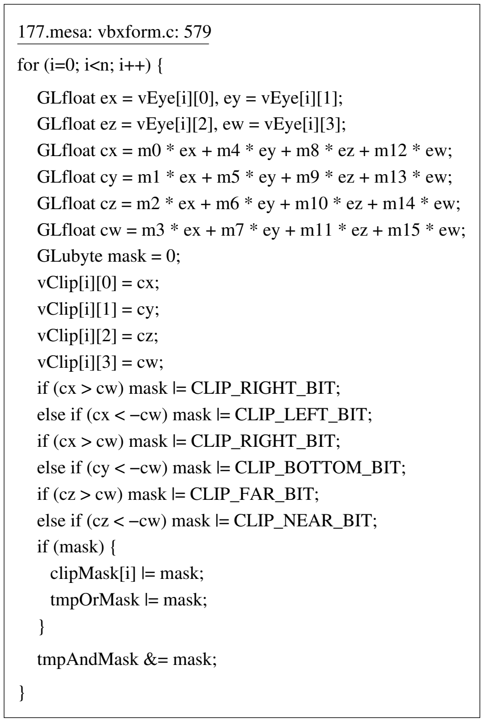


```
256.bzip2: bzip2.c: 2260
for (j=0; j<bbSize; j++) {
    Int32 a2update = zptr[bbStart + j];
    UInt16 qVal = (UInt16) (j >> shifts);
    quadrant[a2update] = qVal;
     if (a2update < NUM_OVERSHOOT_BYTES)
       quadrant[a2update + last + 1] = qVal;
}
```

Figure 3: A candidate loop for CS+DDS

allel execution. For example, consider the loop shown in Figure 3. CS is required in this case because the write to quadrant is conditional. Also, CS in Figure 3 is different from that of Figure 2 where it marks the loop exit condition. In Figure 3, the number of iterations is known but there is a conditional inside. Observe the ambiguous dependence on the write to the array `quadrant`. Clearly, the loop cannot be executed in parallel based on CS-Only. One way to parallelize the loop is via speculative synchronization, as proposed by Martinez and Torrellas in [11]. However, this would require additional hardware support and would incur (potentially high) overhead. Alternatively, one could parallelize the loop by speculating on the dependence in each iteration (DDS is discussed further later in subsection 2.2). This would enable full speculative parallel execution of the loop. We classify such cases under the CS+DDS category.

Although CS has advantages, the presence of a conditional in a loop does not necessitate the use of CS. Using the Intel® compiler we found loops with conditionals which can be executed in parallel (in a multithreaded fashion) without CS. For example, let us consider the loop shown in Figure 4. On careful examination, one would note that the loop obtained after applying reduction [12] on the variables `tmpOrMask` and `tmpAndMask` is a `DOALL` loop.[3] Clearly, there is no need for thread-level CS in this case. However, control speculation may be still useful to boost the intra-thread ILP (a discussion of this is beyond the scope of the paper, see [CM1]–[CM11] in [3] for related work). We separate such loops while evaluating the performance potential of CS-Only, CS+DDS and CS+DVS categories.

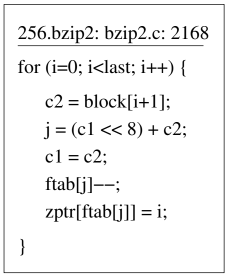


```
177.mesa: vbxform.c: 579
for (i=0; i<n; i++) {
   GLfloat ex = vEye[i][0], ey = vEye[i][1];
   GLfloat ez = vEye[i][2], ew = vEye[i][3];
   GLfloat cx = m0 * ex + m4 * ey + m8 * ez + m12 * ew;
   GLfloat cy = m1 * ex + m5 * ey + m9 * ez + m13 * ew;
   GLfloat cz = m2 * ex + m6 * ey + m10 * ez + m14 * ew;
   GLfloat cw = m3 * ex + m7 * ey + m11 * ez + m15 * ew;
   GLubyte mask = 0;
   vClip[i][0] = cx;
   vClip[i][1] = cy;
   vClip[i][2] = cz;
   vClip[i][3] = cw;
   if (cx > cw) mask |= CLIP_RIGHT_BIT;
   else if (cx < -cw) mask |= CLIP_LEFT_BIT;
   if (cx > cw) mask |= CLIP_RIGHT_BIT;
   else if (cy < -cw) mask |= CLIP_BOTTOM_BIT;
   if (cz > cw) mask |= CLIP_FAR_BIT;
   else if (cz < -cw) mask |= CLIP_NEAR_BIT;
   if (mask) {
     clipMask[i] |= mask;
     tmpOrMask |= mask;
   }
   tmpAndMask &= mask;
}
```

Figure 4: An example `DOALL` loop with control flow

2.2 Data Dependence Speculation

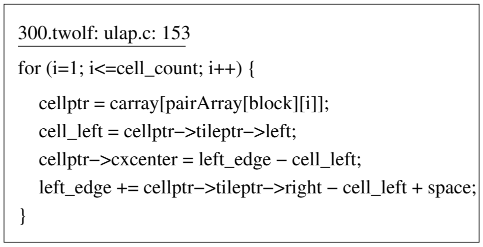
Several techniques have been proposed for code motion which involve operations that may come from widely separated places in the program (the early work in this context relates to techniques proposed for global microcode compaction [2]). These code motions are restricted by data dependencies, which have to be preserved to ensure the semantic correctness of the transformed program. These dependencies may be ambiguous when they involve indirect (array) references or pointer references. For example, it is unclear whether the iterations of the loop shown in Figure 5 can be executed in parallel because of potential aliasing between the writes to the array `zptr` in the different iterations. We classify such cases, where there does not exist any control dependence and there exist ambiguous data dependence(s), under the DDS-Only category.



```
256.bzip2: bzip2.c: 2168
for (i=0; i<last; i++) {
   c2 = block[i+1];
   j = (c1 << 8) + c2;
   c1 = c2;
   ftab[j]--;
   zptr[ftab[j]] = i;
}
```

Figure 5: A candidate loop for DDS

Data dependence speculation corresponds to speculative disambiguation of inter-thread memory dependences [6]. Many schemes such as address prediction have been proposed for the same [13] (also see [DDS1]–[DDS19] in [3]). This permits load instructions to proceed speculatively without waiting for their address operands. The predictability of data dependences can be attributed to the locality in address values [14]. Dependence collapsing [15], yet another approach for data dependence speculation, eliminates data dependences by combining a dependence among multiple instructions into a single instruction. In each approach, if a misspeculation is detected (by the hardware) then the offending speculative thread is rolled back to the point of conflict and the execution is resumed from there on the fly.

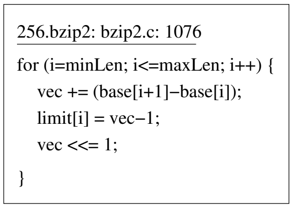


```
300.twolf: ulap.c: 153
for (i=1; i<=cell_count; i++) {
   cellptr = carray[pairArray[block][i]];
   cell_left = cellptr->tileptr->left;
   cellptr->cxcenter = left_edge - cell_left;
   left_edge += cellptr->tileptr->right - cell_left + space;
}
```

Figure 6: A candidate loop for DDS+DVS

However, even in the absence of a control dependence, DDS by itself does not guarantee maximal parallelization. For example, let us consider the example shown in Figure 6. On analysis one would observe there is an ambiguous dependence, write to the `cellptr->cxcenter`, between the iterations of the loop. Even if this is eliminated via DDS, the iterations of the loop cannot be executed in parallel as there exists a true data dependence, write and read to the

[3] Even though the variable `tmpOrMask` is updated inside an `if` block, reduction can still be applied owing to the nature of the OR operation.



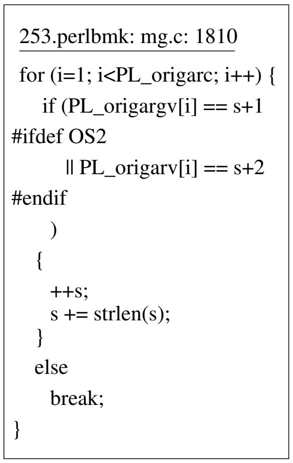
```
256.bzip2: bzip2.c: 1076
for (i=minLen; i<=maxLen; i++) {
    vec += (base[i+1]−base[i]);
    limit[i] = vec−1;
    vec <<= 1;
}
```

Figure 7: A candidate loop for DVS

variable `left_edge`, between the different iterations. One way to parallelize the loop is to speculate on the value of `left_edge` in each iteration. We classify such cases under the DDS+DVS category.

2.3 Data Value Speculation

Data value speculation has been proposed as a means to achieve parallelism beyond the dataflow limit determined by the data dependences. DVS collapses true data dependences [16] by predicting at run-time the result (or the outcome value) of instructions before they are executed and feeding the instructions that depend upon these values with the predicted values [7]. Thus, DVS facilitates the parallel execution of true data dependent instructions. Several reasons such as data redundancy, run-time program constants have been reported for the predictability of data values [17]. Based on the notion of value locality [17], DVS has been proposed in primarily three different forms: (a) load value prediction (this should not be confused with load address prediction); (b) computational value prediction; and (c) return value (of a procedure) prediction (this should not be confused with return address prediction). In [18], Sazeides and Smith classify the value predictors in two classes: computational such as the stride predictor and context-based predictors such as a repeated sequence predictor and a non-stride predictor. For example, let us consider the loop shown in Figure 7 which has a recurrence on the variable `vec`. As a consequence, the iterations of the loop cannot be executed in parallel. Since we know for sure that the recurrence exists, (in other words, there exists a definite data dependence) DDS is not applicable in this case. In absence of a conditional, CS is also not applicable. We classify such cases under the DVS-Only category. The loop can be parallelized by predicting the value of `vec` in iteration `i`, for each `i`.



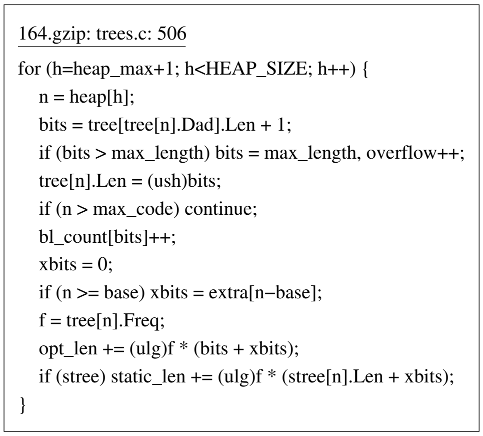
```
253.perlbmk: mg.c: 1810
 for (i=1; i<PL_origarc; i++) {
     if (PL_origargv[i] == s+1
#ifdef OS2
         || PL_origarv[i] == s+2
#endif
       )
     {
       ++s;
       s += strlen(s);
     }
     else
       break;
}
```

Figure 8: A candidate loop for CS+DVS

DVS can potentially yield higher levels of parallelism when used in conjunction with DDS (refer to Figure 6 for an example) and CS. Let us consider the example shown in Figure 8. From the figure we note that prediction of the value of `s`, for each `i`, in conjunction with CS enables parallel execution of the loop. Observe that there are no ambiguous dependence(s) in the loop; therefore, DDS is not applicable. We classify such cases under the CS+DVS category.

The existing literature does not clearly distinguish the DVS-Only and DDS+DVS categories of speculative execution. To clarify, we say that DVS is applicable *iff* there exists a definite data dependence between two instructions, as it does for the write to the variable `vec` in the different iterations of the loop shown in Figure 7. The dependence may be determined at either compile time or at run-time.

Lastly, our analysis shows that there exist program regions (loops in the current context) which require application of CS, DDS as well as DVS for speculative parallelization. One such example is shown in Figure 9. Clearly, there is a need for CS because of, for example, write to the recurrence variable `bits` is under a conditional. Indirect reference to the arrays `tree` and `bl_count` necessitates DDS. Lastly, the recurrence on the variables `opt_len` and `static_len` calls for DVS. Thus, in this case all three types of speculation are required. We classify such cases under the CS+DDS+DVS category.



```
164.gzip: trees.c: 506
for (h=heap_max+1; h<HEAP_SIZE; h++) {
    n = heap[h];
    bits = tree[tree[n].Dad].Len + 1;
    if (bits > max_length) bits = max_length, overflow++;
    tree[n].Len = (ush)bits;
    if (n > max_code) continue;
    bl_count[bits]++;
    xbits = 0;
    if (n >= base) xbits = extra[n−base];
    f = tree[n].Freq;
    opt_len += (ulg)f * (bits + xbits);
    if (stree) static_len += (ulg)f * (stree[n].Len + xbits);
}
```

Figure 9: A candidate loop for CS+DDS+DVS

3. AN OVERVIEW OF SPEC CPU2000

In this section, we present a brief overview of the applications in SPEC CPU2000 [19]. It consists of 12 integer and 14 floating point benchmarks. The size (in lines of code) of each application and their brief description is presented in Table 1. The benchmark suite is representative of a wide spectrum of application domains. It includes programs from the fields of chemistry, weather forecasting, mechanical engineering, physics. In addition, it also consists of general-purpose applications such as data compression and a C compiler. As shown in next section, the loop coverage and the coverage profile of the different applications vary significantly from each other. This behavior indeed provides a very good platform for evaluating the (loop-level) performance potential of TLS on applications with different performance characteristics.

Benchmark	Lines of Code	Language	Description
Integer Benchmarks			
164.gzip	8602	C	Compression
175.vpr	17733	C	FPGA Circuit Placement and Routing
176.gcc	239912	C	C Programming Language Compiler
181.mcf	2457	C	Combinatorial Optimization
186.crafty	21156	C	Game Playing: Chess
197.parser	11391	C	Word Processing
252.eon	41076	C++	Computer Visualization
253.perlbmk	85262	C	PERL Programming Language
254.gap	71390	C	Group Theory, Interpreter
255.vortex	67245	C	Object-oriented Database
256.bzip2	4649	C	Compression
300.twolf	20459	C	Place and Route Simulator
Floating Point Benchmarks			
168.wupwise	2184	Fortran 77	Physics / Quantum Chromodynamics
171.swim	435	Fortran 77	Shallow Water Modeling
172.mgrid	489	Fortran 77	Multi-grid Solver: 3D Potential Field
173.applu	3980	Fortran 77	Parabolic / Elliptic Partial Differential Equations
177.mesa	61323	C	3-D Graphics Library
178.galgel	15334	Fortran 90	Computational Fluid Dynamics
179.art	1270	C	Image Recognition / Neural Networks
183.equake	1513	C	Seismic Wave Propagation Simulation
187.facerec	2409	Fortran 90	Image Processing: Face Recognition
188.ammp	13483	C	Computational Chemistry
189.lucas	2999	Fortran 90	Number Theory / Primality Testing
191.fma3d	60122	Fortran 90	Finite-element Crash Simulation
200.sixtrack	47252	Fortran 77	High Energy Nuclear Physics Accelerator Design
301.apsi	7488	Fortran 77	Meteorology: Pollutant Distribution

Table 1: Description of benchmarks in SPEC CPU2000

4. PERFORMANCE CHARACTERIZATION

In this section, we first present a loop-level characterization of the SPEC CFP2000 and SPEC CINT2000 applications. Next, we present a thorough evaluation of the (loop-level) performance potential of TLS. Although several studies have been done in the past evaluating the benefit of TLS [20, 21, 22], they have the following limitations:

- First, previous limit studies correspond to a loose upper bound on the performance potential of TLS. This in part can be attributed to the inaccurate coverage analysis which in itself is an artifact of the limitations of the compiler used. Instead, we used the highly optimizing Intel® compiler to obtain a better loop coverage.
- Second, all the previous studies were simulation based. Thus, their results do not correspond to a real machine. In contrast, we carried our experiments on a real machine, see Table 2 for the system configuration. This is important due to fact that state-of-the-art processors such as the Intel® Prescott processor have several hardware accelerators, e.g., vector computation units. For example, intra-register vectorization of the following loop (taken from 164.gzip:deflate.c:545) yields a speedup of 8% on the entire application (which directly relates to a decrease in the coverage of the loop) on Pentium® (P4) Processor as compared to the default optimized version (O2) [23].

```
for (n=0; n<HASH_SIZE; n++){
   m = head[n];
   head[n] = (Pos) (m >= WSIZE? m-WSIZE : NIL);
}
```

A coverage analysis oblivious of such accelerators, as in prior works, skews the coverage of the loops (on the higher side) which in turn results in inaccurate assessment of the performance potential of TLS.

Processor	Intel® Prescott Processor, 2.8 GHz
Memory	2 GB
L2 Cache	1 MB
Compiler Flags	-O3 -Qansi_alias -Q_loop_prof=1 -QxP -Qipo -Fa
OS	Windows Server 2003 Enterprise Edition, 32-bit

Table 2: Experimental Setup

Integer Benchmarks

	164.gzip	175.vpr	176.gcc	181.mcf	186.crafty	197.parser	252.eon	253.perlbmk	254.gap	255.vortex	256.bzip2	300.twolf
# of loops	103	331	1984	40	364	579	80	680	846	132	176	565
% Execution Time	16.65	48.3	33.59	46.4	42.5	39.1	32.44	45.74	16.8	11.95	54.13	43.1

Floating Point Benchmarks

	168.wupwise	171.swim	172.mgrid	173.applu	177.mesa	178.galgel	179.art
# of loops	17	52	47	59	155	941	67
% Execution Time	98.5	98.9	95.9	98.4	44.4	89.1	78.1
	183.equake	187.facerec	188.ammp	189.lucas	191.fma3d	200.sixtrack	301.apsi
# of loops	47	123	122	92	643	587	287
% Execution Time	88.8	49.9	59.1	96.3	56.9	71.1	76

Table 3: Total number of innermost loops executed (after optimization) and their coverage, for each application in SPEC CPU2000

- Third, given that applications and architectures have evolved over time, it is difficult to assess the benefit of TLS based on the previous studies as they used SPEC CPU95 [24] benchmark which have been retired for quite some time and have data sets that are too small for today's systems. Instead, we carried our evaluation on SPEC CPU2000 benchmark which is considered to be representative of modern applications and on the Intel®Prescott processor.

4.1 Loop-Level Analysis

In this subsection, we present results about the loop coverage, defined as the percentage of the total execution time spent in the loops, and the coverage profile for each application in SPEC CPU2000. For above, the code generator of the Intel®compiler was modified for automatic insertion of hardware performance counters. The point of insertion of these counters (amongst the different phases of the compilation process) has a direct effect on the coverage analysis. This is due to the fact that insertion of these counters early in the compilation process can potentially disable some of the optimizations. Therefore, it is critical to make sure that these counters are inserted only during the code generation phase. In our experiments, we account for the overhead incurred due to the insertion of these counters. A detailed discussion of our instrumentation support is beyond the scope of the paper.

Further, unlike previous works [25, 26] wherein the instruction count was a measure to characterize the loops, we use the actual execution time spent in the loops as our coverage metric. This is important in order to assess the true performance gain achievable by parallelizing a loop via TLS.

4.1.1 Coverage

We compiled the applications listed in Table 1 using the Intel®compiler and ran them on a Intel®Prescott processor (see Table 2 for the system configuration). Each application was run with the reference data set(s). As listed in Table 4, some applications have multiple data sets. In such cases, the coverage of a loop was computed as a weighted average over all the data sets. Care should be taken when computing the above as the number of loops executed may differ from one data set to another. This is due to the fact that a loop may be executed for one data set and not for another. The total loop coverage is computed as follows: first, *tick%*, defined as the total time spent in a given loop, is determined for each loop. Next, the *self%*, defined as the execution time spent in a loop excluding the time spent in any function or any other loop that may be embedded in it, is determined for each loop. Finally, the *self%* of all the loops is summed to obtain the total loop coverage for a given application.

Table 3 presents the number of innermost loops and their coverage for the applications in SPEC CPU2000. From the table, we observe that only 1 out of 14 SPEC CFP2000 applications has a loop coverage less than 50%. On the other hand, only 1 out of 12 applications, viz., `256.bzip2`, in SPEC CINT2000 has a loop coverage over 50%. Based

Integer Benchmarks			
Benchmark	Number of data sets	Benchmark	Number of data sets
164.gzip	5	175.vpr	2
176.gcc	5	181.mcf	1
186.crafty	1	197.parser	1
252.eon	3	253.perlbmk	7
254.gap	1	255.vortex	3
256.bzip2	3	300.twolf	1
Floating Point Benchmarks			
Benchmark	Number of data sets	Benchmark	Number of data sets
168.wupwise	1	171.swim	1
172.mgrid	1	173.applu	1
177.mesa	1	178.galgel	1
179.art	2	183.equake	1
187.facerec	1	188.ammp	1
189.lucas	1	191.fma3d	1
200.sixtrack	1	301.apsi	1

Table 4: Number of input (reference) data sets per application in SPEC CPU2000

Integer Benchmarks

	164.gzip	175.vpr	176.gcc	181.mcf	186.crafty	197.parser	252.eon	253.perlbmk	254.gap	255.vortex	256.bzip2	300.twolf
# of loops	82	210	1513	30	301	350	72	613	733	128	100	384
% Execution Time	87.6	98.6	38.1	98.9	48	56.6	48.9	51.3	28.7	15.1	85.5	69.8

Floating Point Benchmarks

	168.wupwise	171.swim	172.mgrid	173.applu	177.mesa	178.galgel	179.art
# of loops	18	27	25	29	150	575	24
% Execution Time	98.5	99	99.7	99.8	52.5	99.3	99.9
	183.equake	187.facerec	188.ammp	189.lucas	191.fma3d	200.sixtrack	301.apsi
# of loops	28	57	109	67	504	285	161
% Execution Time	99.7	84.9	97.6	99.8	68.2	99.3	88.8

Table 5: Total number of outermost loops executed (after optimization) and their coverage, for each application in SPEC CPU2000

on our preliminary analysis using the Intel's VTune Performance Analyzer [27], the low loop coverage can be in part attributed to the time spent in library calls and recursion. For example, in `176.gcc`, approximately 30% of the total execution time is spent in the (unoptimized) `memcpy` library call. From above, we learn that for such applications it is critical to develop better approaches to handle recursion and to tune the libraries. A detailed analysis of the time spent in non-loop code is beyond the scope of this paper and is a subject of future work. Surprisingly, the loop coverage is far lower than what we had expected (based on what has been reported for the retired SPEC benchmarks). This is due to the aggressive loop optimizations performed by the Intel®compiler such as full and partial vectorization.

Note that the number of loops (listed in Table 3) for each application is not the same as the number of loops at the source level. This can be primarily attributed to the following reasons: (i) Prior to counting the loops at the code generation phase, the loops (at the source level) undergo a large set of transformations (using the Intel®compiler) such as loop blocking, loop fusion, distribution, et cetera. (ii) Also, the loop counts listed in Table 3 are their dynamic counts. In other words, for a given data set, only those loops are counted which are actually executed. For example, loops within a function which is never executed are not counted.

Another interesting thing to note is that a large number of loops does not imply a large loop coverage: compare the number of loops and their coverage for `176.gcc` and `197.parser`. Similarly, a large loop (in terms of lines of code) does not imply a large coverage. Furthermore, the coverage of a loop is subject to other factors such as the set of optimizations applied and the performance of the cache.

Table 5 presents the number of outermost loops and their coverage for the applications in SPEC CPU2000. The '`# of loops`' count in both Table 3 and Table 5 include the singly-nested loops. This is because a loop nest is not defined in case of singly-nested loops. From Table 5, we observe that only 2 applications in SPEC CFP2000 have a loop coverage less than 75%. In contrast, only 4 applications, viz., `164.gzip, 175.vpr, 181.mcf` and `256.bzip2`, in SPEC CINT2000 have a loop coverage over 75%. As expected, in most applications the coverage of outermost loops is larger than that of the innermost loops. For example, in `164.gzip` the coverage of the outermost loops is 87.6% whereas the coverage of the innermost loops is only 16.65%. Clearly, in such cases parallelization of outermost loops is a must in order to achieve higher levels of parallelism. On the other hand, in applications such as `186.crafty` the coverage of the outermost loops is almost the same as that of the innermost loops. In such cases of parallelization of the innermost loops might be preferable from compilation-time perspective. This is due to the fact that parallelization of non-perfect loop nests (if possible at all) calls for complex control and data dependence analysis which has a direct (adverse) impact on the compilation time. To our surprise, in applications such as `254.gap` and `255.vortex`, the coverage of even the outermost loops is very low. As mentioned earlier, this can be attributed in part to the time spent in recursion and library calls. This provides a valuable guidance to the researchers to focus on optimization of non-loop code.

4.1.2 Coverage Profile

Figure 10 shows the coverage profile of the innermost loops in the SPEC CINT2000 (due to space limitations, we do not include the coverage profile of the outermost loops in SPEC CINT2000 and of the loops in SPEC CFP2000). Only loops with a coverage of more than 0.1% are shown in each graph. Unlike the conventional wisdom, we observe that in many applications the loop coverage is well distributed over several loops. For example, in `176.gcc`, 90% of the total coverage of the innermost loops is spread over 100 loops. From this we learn that in order to achieve significant speedups, one would have to do well across the whole application. Likewise, we observe that in applications such as `255.vortex` the maximum coverage of a single loop is very limited, less than 4% in case of `255.vortex`. For such applications, the benefit of employing TLS it not obvious due to the (high) misspeculation overhead. This needs to be explored and is a subject of future work.

4.2 Auto-parallelization

As mentioned earlier, it is critical to determine the speedup

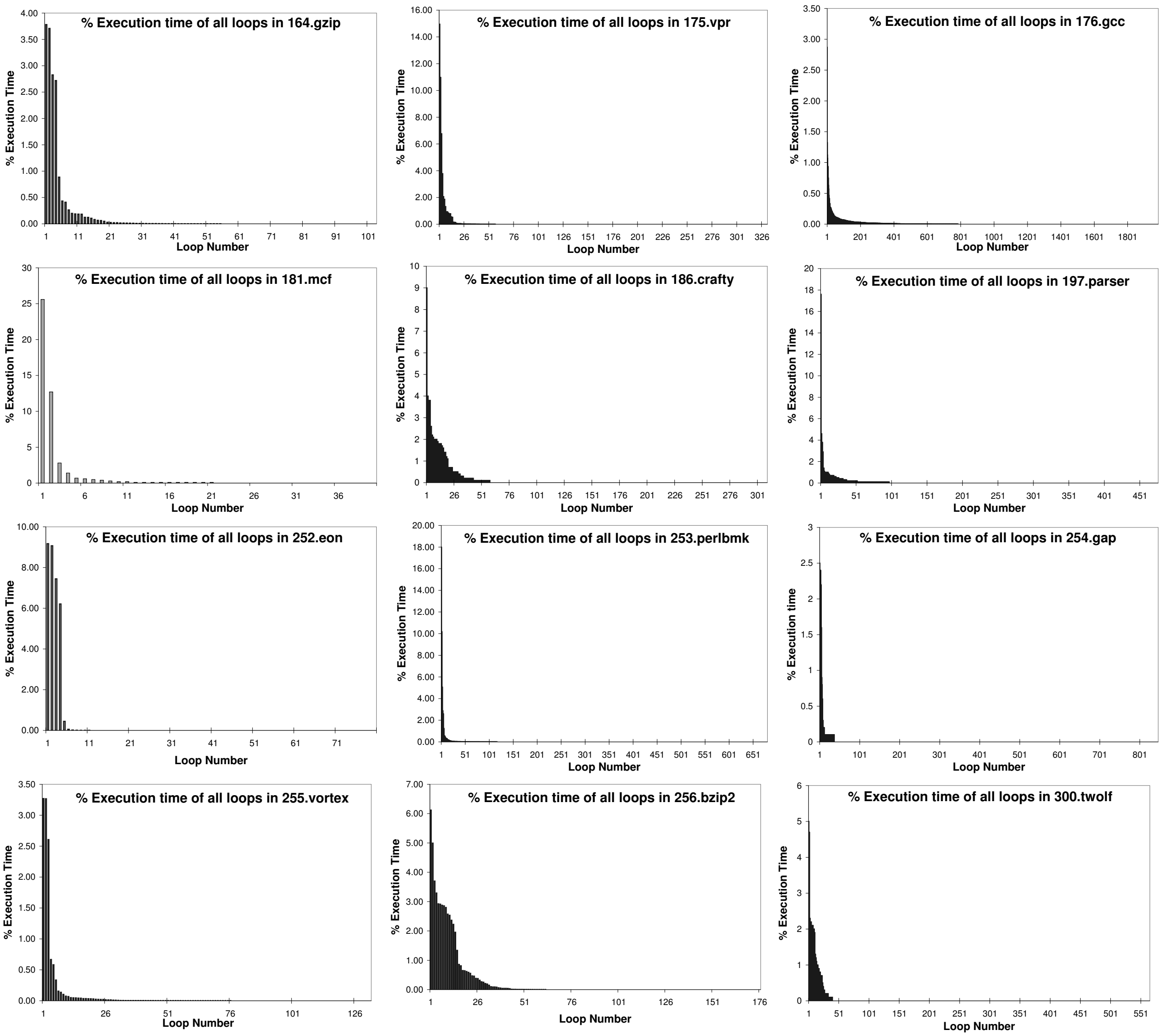


Figure 10: Coverage profiles of the innermost loops in the different applications of SPEC CINT2000

achievable from true TLP in order to assess an upper bound on the speedup achievable via TLS. For this, we breakdown the loop coverage into DOALL type and Non-DOALL type, as illustrated in Figure 11. The loops corresponding to the DOALL sector can be parallelized via simple multithreading, i.e., without any speculation. The sector corresponding to the Non-DOALL loops constitutes the TLS space. We analyze the latter in the next subsection to determine the performance potential of different types of speculation.

We used the Intel$^{®}$compiler to auto-parallelize the loops. The compiler performs a large set of optimizations via procedural inlining, advanced inter-procedural analysis for maximal parallelization. Further, we carried manual analysis (at the source level) of some of the loops which the compiler could not parallelize. For example, we found a few DOALL loops in 183.equake and 179.art which the compiler could not parallelize. Such loops can be easily parallelized via existing parallelizing compiler technology (not supported in the compiler as of now) or via, for example, OpenMP [12] pragmas such as — `#pragma omp parallel for` — which mandate the compiler to parallelize the corresponding loop, thereby alleviating the limitations of the compiler. Since these loops are inherently parallel and do not require any additional architectural or microarchitectural support for par-

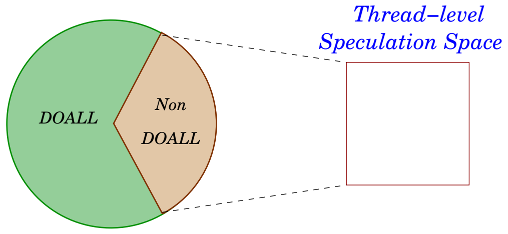


Figure 11: Breakdown of Loop Coverage

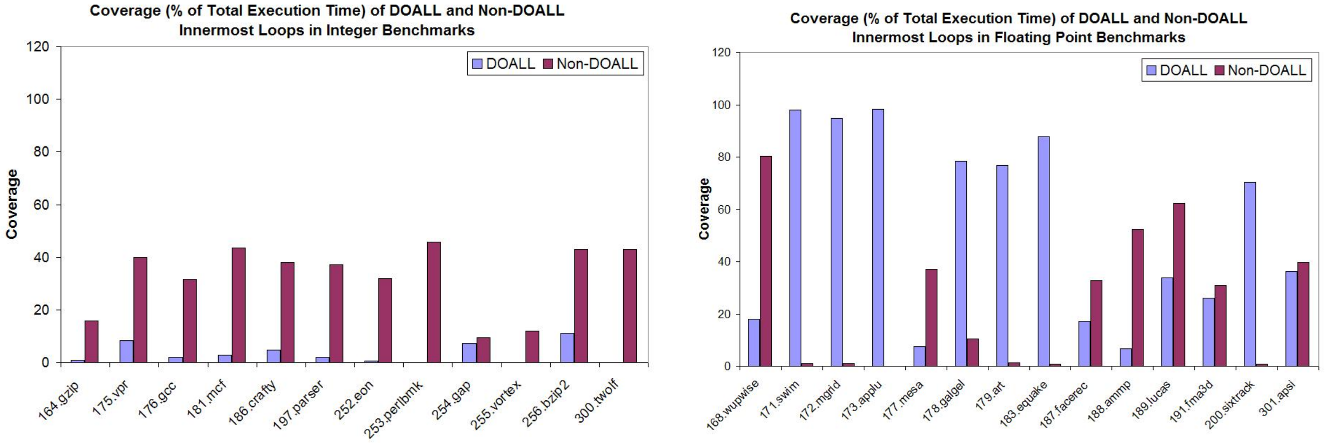


Figure 12: `DOALL`, Non-`DOALL` breakdown of the coverage of innermost loops in SPEC CPU2000

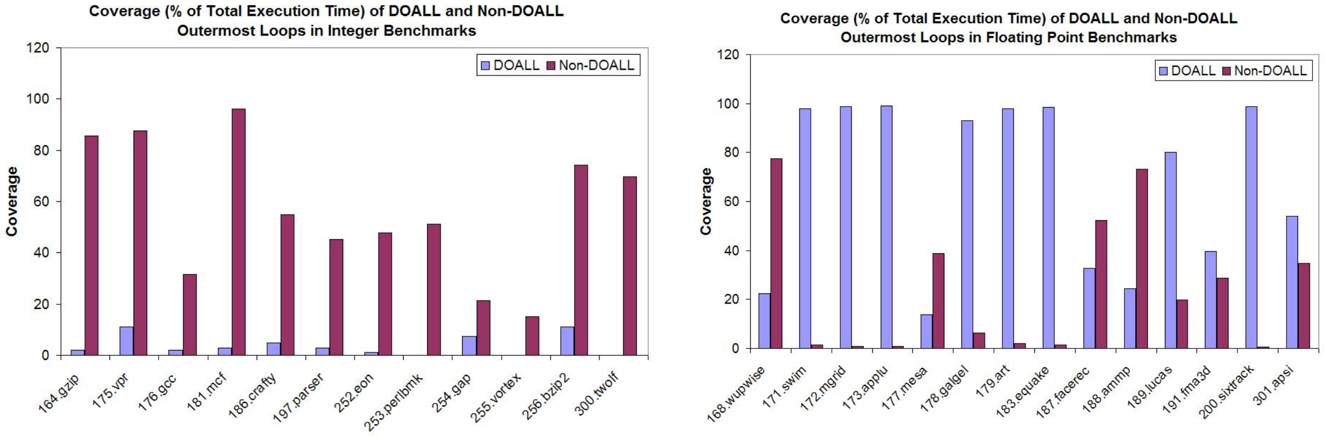


Figure 13: `DOALL`, Non-`DOALL` breakdown of the coverage of outermost loops in SPEC CPU2000

allelization, we include the coverage of such loops as part of the `DOALL` loop coverage. Note that no algorithmic changes were made during the manual analysis. Only (about) 10% of the total number of loops were parallelized manually.

Given a nested loop of depth n, where the depth of the outermost and innermost loop are 1 and n respectively, the coverage of `DOALL` and `Non-DOALL` loops is determined as follows:

i) First, if there exists a `DOALL` loop (say, $\mathcal{L}$) and a `Non-DOALL` loop with depths k and $k+1$, $1 \leq k < n-1$, respectively, then we include the *tick%* of $\mathcal{L}$ in the `DOALL` loop coverage. We ignore all the loops embedded inside $\mathcal{L}$ as there is no need for speculative parallelization of the inner (w.r.t. $\mathcal{L}$) loops. On the other hand, if the loop at depth $k+1$ is a `DOALL` loop, then we include the *self%* of $\mathcal{L}$ in the `DOALL` loop coverage.

ii) Second, if there exists a `Non-DOALL` loop (say, $\mathcal{L}$) which is not embedded inside a `DOALL` loop, then we include the *self%* of $\mathcal{L}$ in the `Non-DOALL` loop coverage. A `DOALL` loop embedded inside $\mathcal{L}$ contributes to the `DOALL` loop coverage.

The coverage of `DOALL` loops thus obtained is a tighter lower bound than achievable from only compiler-driven auto-parallelization. Therefore, the `Non-DOALL` sector in Figure 11 corresponds to an upper bound on the performance gain achievable (at the loop-level) via TLS. In fact, the upper bound is quite loose in nature as the current `DOALL` loop coverage does not capture the inherent parallelism of `Non-DOALL` loops. For instance, let us consider a `Non-DOALL` loop with n iterations, wherein there exists a true dependence between the i^{th} and the $(i+k)^{\text{th}}$ iteration, for each $1 \leq i \leq n-k$. The above loop can be transformed into a doubly nested loop wherein the inner loop is a `DOALL` loop and the outer loop is a `Non-DOALL` loop with true dependence between consecutive iterations. Clearly, the inner loop can be parallelized without any speculation. Such inherent parallelism must be accounted for to obtain a tight upper bound on the performance potential of TLS.

The aforementioned breakdown for the innermost and outermost loops in SPEC CINT2000 and SPEC CFP2000 is shown in Figures 12 and 13 respectively. Assuming an oracle TLP and TLS mechanisms whereby the execution time of the loops can be reduced to zero, the arithmetic mean and geometric mean speedup achievable is shown in Table 7.

	Arith. Mean	Std. Dev.	Geo. Mean
Innermost Loops	28.56%	21.53	13.32%
Outermost Loops	39.16%	31.23	18.18%

Table 7: Upper bound on the loop-level speedup achievable via TLS across SPEC CPU2000

Benchmark	CS-Only (%)	DDS-Only (%)	DVS-Only (%)	CS+DDS (%)	CS+DVS (%)	DDS+DVS (%)	CS+DDS+DVS (%)
Integer Benchmarks							
164.gzip	0.1	0.1	4.21	0.2	2.83	3.61	3.83
175.vpr	0.52	0.1	0.1	0.1	20.33	0.1	18.45
176.gcc	3.8	0.16	1.4	0.1	0.1	0.1	27.1
181.mcf	27	0.1	0.1	0.1	0.1	0.1	13.4
186.crafty	1.34	0.1	0.1	0.1	3.61	0.1	25.33
197.parser	1.59	0.1	0.1	0.1	9.41	0.1	16.89
252.eon	0.1	0.1	0.1	0.1	9.07	0.1	22.84
253.perlbmk	10.18	0.1	0.1	0.1	1.26	0.1	28.64
254.gap	0.1	0.1	0.1	0.1	4.9	0.1	1.6
255.vortex	0.1	0.1	0.1	0.1	3.27	0.1	6.55
256.bzip2	0.47	6.19	0.1	0.1	30.59	0.1	3.7
300.twolf	2.1	0.1	0.1	1.96	2.1	0.1	33.4
Geometric Mean	0.83	0.15	0.17	0.14	2.8	0.13	12.04
Floating Point Benchmarks							
168.wupwise	0.1	0.1	0.1	0.1	0.1	0.1	78.2
171.swim	0.1	0.1	0.1	0.1	0.1	0.1	0.1
172.mgrid	0.2	0.2	0.2	0.2	0.1	0.2	0.2
173.applu	0.1	0.1	0.1	0.1	0.1	0.1	0.1
177.mesa	0.1	0.1	0.1	0.1	0.1	0.1	36.5
178.galgel	0.1	0.1	0.1	0.1	0.1	0.1	8.6
179.art	0.1	0.1	0.1	0.1	0.1	0.1	0.1
183.equake	0.1	0.1	0.1	0.1	0.1	0.1	0.1
187.facerec	0.1	0.1	0.1	0.1	0.1	0.1	31.8
188.ammp	9.4	7.6	0.1	0.1	12.3	0.1	20.4
189.lucas	0.1	61.6	0.1	0.1	0.1	0.1	0.1
191.fma3d	0.1	0.1	0.1	0.1	0.1	0.1	30.3
200.sixtrack	0.1	0.1	0.1	0.1	0.1	0.1	0.1
301.apsi	0.1	14.5	17.8	0.1	0.1	0.1	3.6
Geometric Mean	0.16	0.32	0.15	0.1	0.15	0.11	1.52

Table 6: Performance dissection of different types of speculation for SPEC CINT2000 and SPEC CFP2000

Of course, in practice reducing the loop coverage to zero is not feasible. The actual speedup that can be achieved via TLP and TLS is much lower. Interestingly, in an ideal case speculative parallelization of outermost loops would yield a modest 4.86% higher (geometric mean) speedup than what can be achieved via speculative parallelization of innermost loops.

From Figures 12 and 13 we make the following observations:

a) In all the SPEC CINT2000 applications the coverage of `Non-DOALL` loops is more than the coverage of `DOALL` loops. The coverage of the `Non-DOALL` innermost loops varies from a minimum of 9.5% for `254.gap` to a maximum 45.64% for `186.crafty`. Whereas the coverage of `Non-DOALL` outermost loops varies from a minimum of 15% for `255.vortex` to a maximum of 96% for `181.mcf`. The low coverage of the `DOALL` loops can be attributed to the presence of control and data dependences and to the limitations of set of parallelizing loop transformations applied and the phase ordering of compiler optimizations. The upper bound on the geometric mean speedup achievable via speculative parallelization of `Non-DOALL` innermost and outermost loops is 29.13% and 49.73% respectively.

b) In SPEC CFP2000, we observe that in 7 out of 14 applications the coverage of `DOALL` innermost loops is more than that of `Non-DOALL` innermost loops. Interestingly, in these 7 applications the latter have a negligible coverage, which results in a modest upper bound on the geometric mean speedup (= 6.81%) achievable via TLS in SPEC CFP2000. In case of `DOALL` outermost loops, the upper bound on the geometric mean speedup achievable via TLS is 7.67%. On the other hand, we observe that `DOALL` innermost loops have a coverage of more than 15% for all applications except in `177.mesa` and `188.ammp`. This yields a high upper bound on the geometric mean speedup (= 38.66%) achievable via true TLP in SPEC CFP2000. This upper bound is even higher (= 56.77%) in case of outermost loops.

Overall, one has to also consider that the speedup achievable through TLS is limited by a number of factors. The `Non-DOALL` loop coverage of 29.13% can be sped up by a factor of 2 using two threads *iff* 100% parallel efficiency (defined as the ratio of achievable speedup and number of threads) can be achieved or using four threads with a 50% parallel efficiency. Such high efficiency is very difficult to achieve in practice. Therefore, an overall speedup of 1.17× that TLS is capable of in these two cases is very optimistic.

4.3 Thread-level Speculation

Lastly, we present a dissection of the performance potential of the different types of TLS. To evaluate the above, we analyzed the `Non-DOALL` innermost and outermost loops using the Intel® compiler and via manual analysis at the source level. Due to space limitations we present the dissection only for the innermost loops. Nonetheless, the dissection of the outermost loops has similar characteristics as that of the innermost loops.

Since the total number of `Non-DOALL` loops is very large, for each application we considered only the top (w.r.t. the individual coverages) loops accounting for a total of 90% of the coverage of the `Non-DOALL` innermost loops. In Table 6 we present the dissection of the performance potential of each type of TLS. The columns in Table 6 correspond to the different types of TLS, as illustrated in Figure 1. An entry in a given cell of the table signifies the percentage of the total execution time of the corresponding application that can be parallelized with the corresponding TLS type. Assuming an oracle mechanism for the TLS type, the entry in a given cell also corresponds to an upper bound on the performance potential of the TLS type. For example, 3.8% of the total execution time of `176.gcc` can be parallelized with only control speculation. Similarly, 6.19% of the total execution time of `256.bzip2` can be parallelized with only data dependence speculation; in other words, DDS-Only has a performance potential of 6.19% for `256.bzip2`. The number 0.1 only signifies that the performance potential of the corresponding TLS type for the given application is negligible. It should not be interpreted by its numerical value (as this does not correspond to all the `Non-DOALL` loops).

Contrary to our expectation that CS+DDS+DVS would have the highest performance potential, we observe that in some cases CS-Only, DDS-Only and DVS-Only have the highest potential as in `181.mcf, 189.lucas` and `301.apsi` respectively. However, from the perspective of the entire suite, we note that CS-Only, DDS-Only and DVS-Only have a marginal performance potential, see the row corresponding to the geometric mean in Table 6. Amongst the different combinations, only CS+DDS+DVS seems to have some benefit, which in itself is limited to 12.04% (assuming an oracle TLS mechanism). From our analysis, we conclude that although TLS does have benefits (as established by prior work), the uniquely achievable gain via TLS is somewhat small.

5. PREVIOUS WORK

Task-level speculative computation has been long proposed as a means for extracting higher levels of parallelism [28]. The origin of the task-level speculative computation can be traced back to the early works in functional programming (for a detailed reference of previous work in this context, see [EW1]–[EW9] in [3]). Speculative computation, as defined by Burton in [28], is a computation that is started before it is known to be required. The key idea is to speculatively execute tasks in parallel; on misspeculation, the tasks are aborted and a recovery mechanism is used to restore the machine state. With the emergence of multithreaded processors [29] (also see [MP1]–[MP3] in [3]), many researchers have proposed the use of threads for exploiting speculative parallelism in both hardware and software [30, 31, 32, 33]. This provides opportunities to find parallelism among larger, non-sequential regions [34] of a program's execution, unlike a superscalar processor in which parallelism extraction is restricted to a group of instructions fitting in a hardware instruction window. Unlike parallelization for traditional multiprocessors which requires conservative synchronization for preserving program semantics, TLS can potentially achieve higher levels of parallelism by exploiting dynamic parallelism. Furthermore, TLS is not limited to the programmer's (via use of explicit directives as in OpenMP [12]) or the compiler's ability [35, 36] to find parallel tasks in a given program. Several architectures have been proposed for exploiting speculative *thread-level parallelism (TLP)* [30, 31] (for a detailed listing of other speculative architectures see [MP4]–[MP13] in [3]). Such architectures provide support for multiple hardware contexts and execution roll-back in case of either a control or a data misspeculation. In addition, they provide mechanisms to forward values produced by one thread to another. They primarily differ in how programs are partitioned into different threads: some rely on the compiler to split the program into threads while the others rely on hardware-based techniques.

Several schemes have been proposed for CS, DDS and DVS (see [CS1]–[CS5], [DDS1]–[DDS19] and [DVS1]–[DVS35] in [3] for an extended list of references). In [37], Tubella and González proposed CS technique based on dynamic loop detection wherein the history of the past executed loops is used to speculate the future dynamic instruction sequence. In [38], Huang et al. proposed a compiler technique for dynamic memory disambiguation wherein the disambiguator creates an additional copy of the code that is dependent on the memory references. In one copy of the code, the addresses of the two references are assumed to be the same. In the other copy of the code, the two references are assumed to be different. In both copies, operations that do not have side effects are executed speculatively. Those that do are guarded by the result of comparing the two addresses. In [7], introduced the concept of data value prediction. While the principal objective of branch prediction is to increase the number of candidate instructions and help to efficiently fill the pipeline empty slots, data value prediction aims to enable the processor to execute instructions beyond the limit of true-dependence data dependences. Although the previous work present the benefit of the proposed techniques, the true impact of TLS, independent of the technique employed for a given type of speculation, has not been addressed so far.

Several limits of parallelism studies have been been done in the past, see ([LoP1]–[LoP14]) in [3]. In [39], Steffan and Mowry explored the potential for using thread-level data speculation to facilitate automatic parallelization. Amongst all the prior works, only [20, 21] address TLS in this regard. However, these have the following limitations: (a) these studies were simulation-based which makes it difficult to assess the benefit of TLS on real machines; (b) these studies used SPEC95 for their evaluation (which has been retired for quite some time); and (c) more importantly, they do not dissect the performance potential of different types of TLS.

6. CONCLUSION

Prior work in speculative multithreading has claimed that TLS bears significant potential to boost performance. However, to our surprise, our analysis and evaluation of TLS (excluding true TLP) on SPEC CPU2000 shows that TLS has a limited arithmetic mean speedup potential of 39.16% (standard deviation = 31.23) and a geometric mean speedup potential of 18.18% at the loop level. Most strikingly, none of the prior works filter out the performance gain uniquely achieved by *true-TLP* (which corresponds to threaded execution of truly parallel tasks such as the iterations of a DOALL loop while assessing the impact of TLS. Our results show that in many applications non-speculative TLP has a much larger potential than TLS. Lastly, we presented a dissection of the performance potential of different types of TLS, both standalone and in conjunction with other types of speculation. Our analysis shows that amongst the different combinations CS+DDS+DVS has the highest, albeit modest, geometric mean speedup potential of 12.04% at the loop level. Interestingly, we found that the total loop coverage in many integer applications is quite low. For such applications, this observation provides a valuable guidance to the researchers to focus on optimization of non-loop code.

Several techniques have been proposed based on branch prediction, predication et cetera to achieve higher levels of ILP. Likewise, many approaches have been proposed based on compiler-driven data and instruction prefetching [40, 41]. As future work, we plan to evaluate the performance potential of TLS beyond what is achievable via the techniques mentioned above. In essence, we would like to assess the ILP vs TLP/TLS trade-off.

7. ACKNOWLEDGMENTS

The first author would like to thank Ilya Cherny (Intel, Novosibirsk, Russia) for his help in providing instrumentation support in the Intel compiler used for obtaining the loop coverage of the applications in SPEC CPU2000. The first author will also like to thank Prof. Mario Furnari, Istituto di Cibernetica Eduardo Caianiello, Pozzuoli, Italy, for letting know about the work done in speculative computation in context of functional programming.

8. REFERENCES

[1] A. Liles Jr. and B. Wilner. Branch prediction mechanism. *IBM Technical Disclosure Bulletin*, 22(7):3013–3016, December 1979.

[2] J. A. Fisher. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling Beyond Basic Blocks*. PhD thesis, New York University, 1979.

[3] A. Kejariwal and A. Nicolau. Reading list of performance analysis, speculative execution. http://www.ics.uci.edu/~akejariw/SpeculativeExecutionReadingList.pdf.

[4] J. A. Fisher. Trace Scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[5] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 218–232, La Jolla, CA, 1995.

[6] A. Nicolau. Run-time disambiguation: coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):633–678, 1989.

[7] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Technical Report EE Department TR # 1080, Technion–Israel Institute of Technology, November 1996.

[8] H. Wang, P. Wang, R. D. Weldon, S. M. Ettinger, H. Saito, M. Girkar, S. S-W. Liao, and J. P. Shen. Speculative precomputation: Exploring the use of multithreading for latency. In *Intel Technology Journal*, February 2002.

[9] S. Lundstrom and G. Barnes. A controllable MIMD architectures. In *Proceedings of the 1980 International Conference on Parallel Processing*, St. Charles, IL, August 1980.

[10] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.

[11] J. Martinez and J. Torrellas. Speculative synchronization: Programmability and performance for parallel codes. *IEEE Micro*, 23(6):126–134, December 2003.

[12] OpenMP Specification, version 2.5. http://www.openmp.org/drupal/mp-documents/spec25.pdf.

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

[14] K. M. Lepak, G. B. Bell, and M. H. Lipasti. Silent stores and store value locality. *IEEE Transactions on Computers*, 50(11):1174–1190, 2001.

[15] R. R. Oehler and R. D. Groves. IBM RISC system/6000 system architecture. *IBM Journal of Research and Development*, 34(1):23–36, January 1990.

[16] G. Estrin and R. Turn. Automatic assignment of computations in a variable structure computer system. *IEEE Transactions on Electronic Computers*, EC-12(5):755–773, December 1963.

[17] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. *SIGPLAN Notices*, 31(9):138–147, 1996.

[18] Y. Sazeides and J. E. Smith. Limits of data value predictability. *International Journal of Parallel Programming*, 27(4):229–256, 1999.

[19] SPEC CPU2000. http://www.spec.org/cpu2000.

[20] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313, Newport Beach, CA, October 1999.

[21] P. Marcuello and A. González. A quantitative assessment of thread-level speculation techniques. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 595–604, Cancun, Mexico, May 2000.

[22] F. Warg and P. Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on cmp platforms. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 221–230, 2001.

[23] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic detection of saturation and clipping idioms. In *Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computing*, pages 61–74, 2002.

[24] SPEC CPU95. http://www.spec.org/cpu95/.

[25] Makoto Kobayashi. Dynamic characteristics of loops. *IEEE Transactions on Computers*, 33(2):125–132, 1984.

[26] J. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):812–826, 1993.

[27] Intel® VTune™ Performance Analyzer 8.0 for Linux. http://www.intel.com/cd/software/products/asmo-na/eng/vtune/vlin/index.htm.

[28] F. W. Burton. Speculative computation, parallelism and functional programming. *IEEE Transactions on Computers*, 34(12):1190–1193, 1985.

[29] R. H. Halstead Jr. and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 443–451, Honolulu, Hawaii, 1988.

[30] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th International Symposium on*

Computer Architecture, pages 58–67, Gold Coast, Australia, May 1992.

[31] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the* 22^{nd} *Annual International Symposium on Computer Architecture*, pages 414–425, Ligure, Italy, 1995.

[32] D. Bruening, S. Devabhaktuni, and S. Amarasinghe. Softspec: Software-based speculative parallelism. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[33] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen. Helper threads via virtual multithreading. *IEEE Micro*, 24(6):74–82, 2004.

[34] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, 1990.

[35] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 15(8):713–724, 2004.

[36] 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 the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[37] J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 14–23, Las Vegas, NV, February 1998.

[38] A. S. Huang, G. Slavenburg, and J. P. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21th International Symposium on Computer Architecture*, pages 200–210, Chicago, IL, 1994.

[39] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

[40] A. C. Klaiber and H. M. Levy. Architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–63, May 1991.

[41] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 40–52, Santa Clara, CA, April 1991.