Tight Analysis of the Performance Potential of Thread Speculation using SPEC CPU2006

Arun Kejariwal[a], Xinmin Tian[b], Milind Girkar[b], Wei Li[b], Hideki Saito[b], Utpal Banerjee[b]
Alexandru Nicolau[a], Alexander V. Veidenbaum[a], Constantine D. Polychronopoulos[c]

[a]University of California, Irvine [b]Intel Corporation [c]University of Illinois at Urbana-Champaign

Abstract

Multi-cores such as the Intel®[1] Core™2 Duo processor, facilitate efficient thread-level parallel execution of ordinary programs, wherein the different threads-of-execution are map-ped onto different physical processors. In this context, several techniques have been proposed for auto-parallelization of programs. Recently, thread-level speculation (TLS) has been proposed as a means to parallelize difficult-to-analyze serial codes. In general, more than one technique can be employed for parallelizing a given program. The overlapping nature of the applicability of the various techniques makes it hard to assess the intrinsic performance potential of each. In this paper, we present a tight analysis of the (unique) performance potential of both: (a) TLS in general and (b) specific types of thread-level speculation, viz., control speculation, data dependence speculation and data value speculation, for the SPEC[2] CPU2006 benchmark suite in light of the various limiting factors such as the threading overhead and misspeculation penalty. To the best of our knowledge, this is the first evaluation of TLS based on SPEC CPU2006 and accounts for the aforementioned real-life constraints. Our analysis shows that, at the innermost loop level, the upper bound on the speedup uniquely achievable via TLS with the state-of-the-art thread implementations for both SPEC CINT2006 and CFP2006 is of the order of 1%.

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, threading overhead, conflict probability, misspeculation penalty

[1] Intel is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[2] Other names and brands may be claimed as the property of others.

PPoPP'07 March 14–17, 2007, San Jose, California, USA.

1. Introduction

Several techniques have been proposed[3] for thread-level program parallelization. These techniques can be primarily classified into two categories: non-speculative or speculative. The former aims at exploitation of true (non-speculative) thread-level parallelism (TLP) from inherently parallel program regions (such as a `DOALL` loop [2]) whereas the latter aims at exploiting speculative thread-level parallelism (sTLP) from difficult-to-analyze (potentially parallel) program regions. Recently, there has been a large amount of work done in the context of thread-level speculation (TLS) to exploit sTLP [1]. In prior work it has been shown that TLS can yield high speedups. However, in practice, the profitability of TLS is constrained by a variety of factors such as the threading overhead and the misspeculation penalty. As a consequence, TLS-driven parallelization is beneficial for only a subset of all the program regions that cannot be parallelized using the existing compiler techniques. This makes it difficult to assess the true performance potential of TLS.

In this work, we obtain *tight* upper bounds[4] on the performance potential of TLS by filtering out *both* (a) inherently parallel program regions and (b) non-profitable candidates for TLS. The analysis presented in [3] does not account for the latter; hence, the upper bounds (on the performance potential of TLS) presented in [3] are loose. Due to space limitations, we present our analysis for loops only. Nonetheless, the evaluation methodology presented in this paper can be applied to program regions in general.

Arguably, TLS can potentially yield higher performance gain in the absence of a parallelizing and optimizing compiler. TLS may also serve as an assist in achieving higher performance by warming up the caches. However, this is subject to various practical constraints such as the threading overhead incurred and misspeculation penalty. In this work we focus on the parallelism that cannot be exploited with the state-of-the-art compiler technology but can be (uniquely and) efficiently exploited via TLS. The analysis presented in this paper should not be construed as an argument in favor of any particular technique.

The main contributions of the paper are as follows:

- First, we outline a baseline for evaluation of TLS. For this, we cleanly separate the performance gain that can be achieved via the state-of-the-art ILP and TLP-based approaches. We *illustrate* the above differentiation with the help of code snippets from SPEC CPU2006. We believe that outlining such a baseline is critical for assessing the speedup uniquely achievable

[3] See [1] for an extensive listing of prior work in multithreading and program parallelization.

[4] Such bounds are empirical in nature and should not be confused with theoretical bounds.

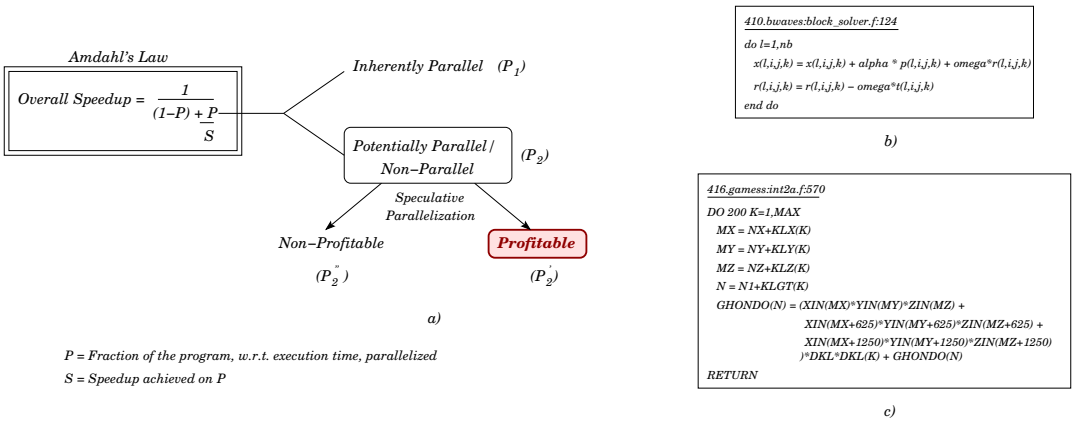


Figure 1. Revisiting Amdahl's Law. a) How to measure the performance potential of TLS; b) An example of inherently parallel loop; c) An example of a possibly parallel loop, subject to the uniqueness of the values in the array `KLGT`, loop.

via TLS and will help in the development of tighter evaluation methodologies for future research in TLS.

- Second, using the Intel® compiler and manual analysis, we present a realistic upper bound on the amount of speedup achievable via TLS while accounting for the threading overhead incurred with state-of-the-art thread implementations. The evaluation is carried for the applications present in the industry-standard SPEC CPU2006 benchmark suite [4].
- Third, we present a thorough analysis of the sensitivity of the performance potential of TLS w.r.t. the threading overhead. This is particularly useful from a futuristic perspective as the analysis presented in this paper can be used to reassess the speedup uniquely achievable via TLS if and when better thread implementations are available in future.
- Fourth, we determine an upper bound on the conflict[5] probability for TLS to be profitable. The bound can be interpreted as a *tolerance factor* while deciding whether to apply TLS to a given program region or not. Further, based on our analysis we find that the conflict probability decreases with increasing threading overhead.

The rest of the paper is organized as follows: Section 2 overviews the evaluation methodology and briefs about the applications in the newly released SPEC CPU2006 benchmark suite. The baseline for evaluation of the performance potential thread-level speculation model is discussed in Section 3. Performance evaluation results are presented in Section 4. Finally, in Section 5 we conclude with directions for future work.

2. Preliminaries

Amdahl's Law [5] governs the overall speedup achievable via program parallelization (see Figure 1). Let P represent the portion of the program parallelized. In the context of TLP and TLS, P can be divided into two classes as follows:

$$P = P_1 + P_2$$

where P_1 represents the portion of P which is inherently parallel (from dependence perspective), as exemplified by Figure 1(b), and can be executed non-speculatively; and P_2 represents the portion of P which is potentially parallel or is non-parallel and can be parallelized speculatively, as exemplified by Figure 1(c). As discussed in [3], the coverage, defined as the percentage of the total execution time, of P_2 constitutes an upper bound on the speedup that can be achieved via TLS. P_2 can be classified into the following two categories:

$$P_2 = P''_2 + P'_2$$

where P''_2 and P'_2 represent the portions for which TLS is non-profitable and profitable respectively. The cost efficiency of TLS is governed by a variety of factors such as the threading overhead and the misspeculation probability. The impact of these parameters on TLS in general and specific types of TLS, viz., control speculation (CS), data dependence speculation (DDS) and data value speculation (DVS) has not been studied so far.

The speculation space can be partitioned into three categories. The first category — CS-Only, DDS-Only, DVS-Only — corresponds to the cases in which a program region can be speculatively parallelized using a TLS approach of the corresponding speculation type in a standalone fashion. The second category — CS+DDS, CS+DVS, DDS+DVS — corresponds the cases in which a program region can be speculatively parallelized *iff* TLS approaches of both types (as applicable) are applied. Lastly, the third category — CS+DDS+DVS — corresponds to the cases in which a program region can be speculatively parallelized *iff* TLS approaches of all the three types are applied [3].

We now present a brief overview of the applications in SPEC CPU2006. It comprises of 12 integer and 17 floating point benchmarks. The size (in lines of code) of each application and their brief description are given in Table 1. The SPEC CPU benchmarks are widely used and considered to be representative of a wide spectrum of application domains. This provides a good platform for evaluating the performance potential of TLS on applications with different characteristics.

3. The Baseline

We employ a two-step approach to determine the speedup achievable via TLS only at the innermost loop level. First, we separate the loops which can be parallelized via state-of-the-art compiler techniques for dependence analysis, pointer analysis, inter-procedural analysis et cetera from the loops which can be parallelized only speculatively. Second, we filter out the loops for which it is more beneficial (from performance perspective) to exploit ILP instead of sTLP. This can, in part, be attributed to factors such as the threading overhead which limit exploitation of sTLP. The set of remaining loops constitute the baseline for evaluation of the true performance potential of TLS at the innermost loop level. The filtering was done using the Intel® compiler and manual analysis. However, less than 10% of the total number of loops were parallelized manually; more importantly, the coverage of such loops is negligible. The techniques considered during the manual analysis have been implemented either in production compilers or in research compilers such as ORC [6] or in public domain compilers such as gcc [7]. In the rest of this section, we illustrate our approach for a few

[5] A conflict is said to occur when a dependence speculation "misfires ", i.e., the dependence is wrongly speculated.

cases with the help of code snippets from applications in SPEC CPU2006.

3.1 Auto-parallelization

As mentioned above, we first separate out DOALL loops from non-DOALL loops. For example, consider the loop shown in Figure 2. On careful examination one would observe that the writes to the array img->m7 in the different iterations can potentially alias with each other. On the other hand, it is critical, from parallelization perspective, to note that the same constant 0 is written in each iteration of the loop. This renders the partial ordering between the writes redundant. In other words, the iterations of the loop can be executed in parallel without any speculation[6] or any explicit thread synchronization. Thus, the loop shown in Figure 2 is in fact a DOALL loop. We exclude such loops while evaluating the performance potential of TLS.

```
464.h264avc: block.c: 1621
for (coeff_ctr =1; coeff_ctr<12; coeff_ctr++) //ac coeff
{
   if (img->field_picture || (img->MbaffFrameFlag && currMB->mb_field))
   { // Alternate scan for field coding
     i = FIELD_SCAN[coeff_ctr][0];
     j = FIELD_SCAN[coeff_ctr][1];
   }
   else {
     i = SNGL_SCAN[coeff_ctr][0];
     j = SGNL_SCAN[coeff_ctr][1];
   }
   img->m7[n1+i][n2+j] = 0;
   ACLevel[coeff_ctr] = 0;
}
```

Figure 2. A candidate loop for semantic-driven parallelization

In a similar fashion, other techniques as proposed in [8, 9, 10] can be used for semantic-driven parallelization. There has been a large amount of work in the context of automatic program parallelization [11]. A large set of loops can be parallelized using the existing techniques.

3.2 ILP/sTLP Trade-off

In the context of SMT (simultaneous multithreading) Mitchell et al. [12] showed that in certain cases it is more profitable (from performance perspective) to exploit ILP instead of TLP. In a similar vein, on analysis we find that there is a similar trade-off between ILP and sTLP. In this subsection we illustrate a couple of instances of the same.

3.2.1 DVS vs. Software Pipelining

Loops with recurrences serve as potential candidates for DVS. However, in many cases application of DVS is either unnecessary or is not profitable. For example, consider the loop shown in Figure 3.

```
453.povray:splines.cpp:120
for (i=2; i <= sp->Number_Of_Entries - 2; i++) {
   u[i] = 2*(h[i]+h[i-1]) - (h[i-1]*h[i-1])/u[i-1];
   v[i] = 6*(b[i]-b[i-1]) - (h[i-1]*v[i-1])/u[i-1];
}
```

Figure 3. A candidate loop for DVS

On analysis we find that the amount of computation contained in the loop is much smaller than the threading overhead ($\approx$ 1000 cycles). This makes exploitation of sTLP non-profitable. Instead, performance of such loops can still be improved by exploiting ILP. In this case, the loop can be modulo scheduled [13] to achieve higher level of ILP. In general, percolation scheduling [14] in conjunction with advanced dependence analysis [15] can be employed for exploiting ILP.

3.2.2 CS vs. Perfect Pipelining

Consider the loop shown in Figure 4. The loop has a coverage of less than 1%. More importantly, the coverage per iterations is very small. This makes CS+DVS-based threaded execution[1] of the loop non-profitable (see Section 4 for further discussion). On the other hand, the loop can be *perfect pipelined* [16] to exploit ILP in order to achieve better performance.

```
450.soplex:ssvector.cc:863
for (int i=0; i< rhs.size(); ++i) {
   int k = rhs.index(i);
   Real v = rhs.value(i)
   if (isZero(v, epsilon))
     val[k] = 0.0;
   else {
     val[k] = v;
     idx[num++] = k;
   }
}
inline bool isZero(Real a, Real eps = Param::epsilon()) {
    return fabs(a) <= eps;
}
```

Figure 4. A candidate loop for perfect pipelining

The above example illustrates that not all the non-DOALL loops with control flow and a potential dependence serve as candidates for speculative execution. Such loops should not be considered while evaluating the speedup achievable via TLS.

In general, the speedup achievable via TLS should be evaluated beyond what can be achieved via any of the existing techniques for ILP and TLP with explicit synchronization.

3.3 Symbolic Analysis

Consider the example loop shown in Figure 5. From the figure we note that there exists a recurrence between consecutive iterations of the loop. The loop can be speculatively parallelized via DVS. However, on symbolic analysis, we observe that the loop can be transformed into a DOALL loop. This obviate the need for speculative execution. This is indeed desirable as it eliminates any performance penalty due to misspeculation.

```
447.dealII:fe_q.c:663
for (unsigned int i=1; i<dpo.size(); ++i)
   dpo[i] = dpo[i-1]*(deg-1);
```

Figure 5. A candidate loop for symbolic differencing-based parallelization

The transformed loop obtained after applying symbolic differencing is given below:

```
for (unsigned int i=1; i<dpo.size(); ++i)
   dpo[i] = dpo[0]*((deg-1)**i);
```

From above we note the transformed loop is a DOALL loop. Consequently, speculative parallelization of the loop is unnecessary in such cases. We argue that such loops should be excluded while evaluating the speedup uniquely achievable via TLS.

[6] However, speculation may be still be useful to boost the intra-thread ILP. A discussion of this is beyond the scope of the paper, for related work see [CM1]–[CM11] in [1].

Benchmark	Lines of Code	Language	Description
			Integer Benchmarks
400.perlbench	155432	C	Cut-down version of Perl v5.8.7
401.bzip2	8293	C	Lossless, block-sorting data compression
403.gcc	518781	C	Based on gcc Version 3.2, generates code for Opteron
429.mcf	2685	C	Single-depot scheduling in public mass transportation
445.gobmk	197215	C	The program plays the game Go and executes a set of commands to analyze Go positions
456.hmmer	35992	C	Sensitive database searching
458.sjeng	13487	C	A highly-ranked chess program that also plays several chess variants
462.libquantum	4805	C	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm
464.h264avc	51578	C	Video compression standard
471.omnetpp	48159	C++	Discrete event simulation of a Ethernet network
473.astar	5842	C++	2D path finding library used in game AI
483.xalancbmk	326504	C, C++	A modified version of Xalan-C++, which transforms XML documents to other document types
			Floating Point Benchmarks
410.bwaves	918	Fortran	Computation of 3-dimensional transonic transient laminar viscous flow
416.gamess	932818	Fortran	General Atomic and Molecular Electronic Structure System
433.milc	15042	C	Generation of gauge filed for lattice gauge theory with dynamical quarks
434.zeusmp	37326	Fortran	Simulation of astrophysical phenomena
435.gromacs	87736	C	Simulation of Newtonian equations of motion for systems with hundreds to millions of particles
436.cactusADM	104047	Fortran, C	Solving Einstein evolution equations in their standard ADM 3+1 formualtion
437.leslie3d	3807	Fortran	Large-Eddy Simulations with Linear-Eddy Model in 3D
444.namd	5315	C	Simulation of large biomolecular systems
447.dealII	199654	C++	Adaptive finite elements and error estimation
450.soplex	41417	C++	Solves a linear program using the Simplex algorithm
453.povray	157825	C++	Ray-tracer
454.calculix	49927	Fortran, C	Finite element code for linear and non-linear three-dimensioanl structural applications
459.GemsFDTD	11580	Fortran90	Solving Maxwell equations in 3D in the time domain
465.tonto	143152	Fortran90	Quantum chemistry package
470.lbm	1176	C	Simulate incompressible fluids in 3D
481.wrf2	217896	Fortran90, C	Weather Research and Forcasting modeling system
482.sphinx	207732	C	Speech recognition system

Table 1. Description of benchmarks in SPEC CPU2006

3.4 Procedural-level Speculation

Several works [17, 18, 19] have proposed procedural-level TLS for exploiting higher levels of sTLP, as exemplified in Figure 6. From the figure we see that the iterations corresponding to $i = 1, 2$ are speculatively mapped onto threads T_3 and T_5. On encountering a procedure call, thread T_1 spawns a speculative thread T_2 which executes the continuation of the function `foo`. Threads T_3 and T_5 follow suit (though not shown in the figure for clarity).

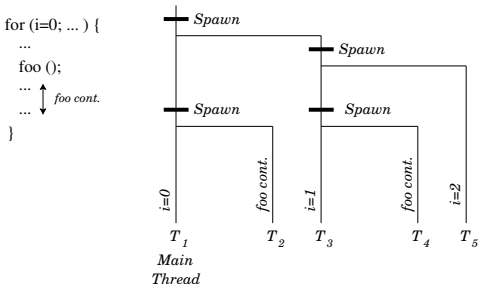


Figure 6. An illustration of procedural-level TLS (`foo cont.` refers to the continuation of the function `foo`)

Subject to overhead incurred due to spawning of multiple speculative threads and the performance penalty incurred due to misspeculation, procedure-level TLS can potentially yield higher levels of parallelism.

However, in quite a few cases we observe that loops with procedure calls are inherently parallel. Such loops can be detected via state-of-the-art data and control dependence analysis techniques. In addition, run-time disambiguation [20] and multi-versioning [21] can be used to assist detection of such loops. For instance, let us consider the example loop shown in Figure 7 (taken from SPEC CFP2000). The loop has a coverage of 20.1%. In absence of inter-procedural analysis, the loop serves as a good candidate for procedural-level TLS. However, on careful examination, we observe that the procedure call `phi2, phi1` and `phi0` are infact loop invariant. The resulting loop, obtained after moving the procedure calls outside the loop, is a `DOALL` loop.

In addition, on analysis we note that in many cases the procedure calls are too small w.r.t. amount of computation contained in them. This makes use of TLS non-profitable due to the high threading overhead (the impact of the latter is further discussed in Section 4). In such cases, techniques such as procedure inlining [22] can be used for exploiting ILP to achieve better performance. We filter such loops while evaluating the performance potential of TLS.

```
183.equake:quake.c:462
for (i=0; i<ARCHnodes; i++)
  for (j=0; i<3; j++)
    disp[distplus][i][j] += 2.0 * M[i][j] * disp[dispt][i][j] –
                            (M[i][j] – Exc.dt / 2.0 * C[i][j]) * disp[disptminus][i][j] –
                            Exc.dt * Exc.dt * (M23[i][j] * phi2(time) / 2.0 +
                            C23[i][j] * phi1(time) / 2.0 + V23[i][j] * phi0(time) / 2.0);
```

```
double phi2(t)
double t;
{
  double value;
  if (t <= Exc.t0) {
    value = 2.0 * PI / Exc.to / Exc.t0 * sin(2.0 * PI * t /Exc.t0);
    return value;
  }
  else
    return 0.0;
}
```

Figure 7. A `DOALL` loop with procedure calls

4. TLS Evaluation

In the previous section, we outlined an approach for TLS evaluation. The baseline can be interpreted as a *threshold* for profitability of TLS on the candidate `non-DOALL` loops. In this section we evaluate the above. First, we present sensitivity analysis of the efficacy of TLS in general and different types of TLS in light of the threading overhead using the state-of-the-art thread implementation. Then we analyze the impact of loop unrolling and usage of large number of cores on the speedup achievable via TLS. Lastly, we present an analytical model to determine an upper bound on the conflict probability for which TLS is beneficial (from performance point of view). For simplicity of exposition, we present the analysis for a two processor case unless mentioned otherwise explicitly. 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.

We compiled the applications listed in Table 1 using the Intel® compiler and ran them on a Intel® Core™2 processor (see Table 2 for the system configuration). Each application was run with the reference data set(s).

Processor	Intel® Core™2 Processor, 2.4 GHz
Memory	2 GB
L1 D-Cache	32 KB
L1 I-Cache	32 KB
L2 Cache	4 MB
Compiler Flags	-O3 -Qansi_alias -Q_loop_prof=1 -QxT -Qipo -Fa
OS	Windows Server 2003 Enterprise Edition (Service pack 1), 32-bit

Table 2. Experimental Setup

4.1 Impact of Threading Overhead

It has been shown in prior work that TLS can potentially yield large speedups. However, in each case,[7] the evaluation was carried via simulation which either did not account for the threading overhead or an unrealistic (compared to state-of-the-art) value of threading overhead was assumed, e.g., a 8 cycle threading overhead was assumed in [25]. The overhead is incurred when an iteration is mapped to a thread during the scheduling process. In light of this, it is difficult to assess the actual speedup that can be achieved via TLS.

In this subsection, we address the above by analyzing the sensitivity of the speedup achievable via TLS w.r.t. the threading overhead. For this, as a first step, we determined the threading overhead using Intel's VTune Performance Analyzer [26]. Our measurements reveal that the real threading overhead is over 1000 cycles. One could argue that this may be an artifact of the specific hardware and/or implementation. However, this is indeed not the case as other implementations incur much higher overhead, e.g., the Linux NPTL implementation incurs an overhead of more than 10000 cycles [27]. As a matter of fact, based on our evaluation using the EPCC microbenchmarks [28] we find that the Intel® OpenMP runtime library is $2\times$–$10\times$ faster than other run-time libraries. The high threading overhead can in part be attributed to thread creation, thread management and synchronization. The latter in turn involves (but is not limited to) allocation of local variables, guaranteeing mutually exclusive access to the shared data structure to the different threads. The results presented in this subsection correspond to dynamic scheduling wherein iterations are mapped to an idle thread one at a time. Arguably, one can amortize the threading overhead by allocating a chunk of iterations to thread. However, this has its own downsides as discussed in the next subsection.

Figure 8 presents the sensitivity analysis of the performance potential of TLS at the innermost loop level[8] w.r.t. the threading overhead. Note that the scale of the different graphs is not the same; the scales were chosen so as to ensure readability. The coverage of `non-DOALL` loops (which corresponds to an upper bound on the speedup that can be achieved via loop-level TLS) for different threading overheads was obtained in the following way:

i) First, we determine the coverage of all the `non-DOALL` loops using the methodology described in [3].
ii) Then we determine the coverage per iteration (on an average) for all the `non-DOALL` loops. An average measure is used due to the following: in order to achieve best performance, the Intel® compiler employs multi-versioning [21] wherein different set of optimizations may be applied during different invocations of the same loop. As a consequence, the number of iterations executed may vary from one invocation to another of the same loop. For the sake of simplicity, we compute the average number of iterations executed for each loop under consideration.
iii) Third, for a given threading overhead, we filter out the loops whose coverage per iteration is less than the threading overhead because speculative parallelization of such loops is not profitable. The sum of the coverages of the remaining loops constitutes the performance potential of TLS corresponding to the given threading overhead.

Due to space limitations we present results only for SPEC CINT2006. The results for `462.libquantum` are not shown here as the coverage of `non-DOALL` loops in the `462.libquantum` is negligible. From the figure we observe that in applications such as `456.hmmer` and `464.h264avc` the performance potential is very small even when the threading overhead is zero! Further, we note that in the benchmarks `429.mcf`, `456.hmmer` and `458.sjeng`, the dip in coverage is akin to a step function. This behavior is reflective of their high sensitivity to the threading overhead. Unlike other benchmarks where the coverage is negligible for high values of the threading overhead, `473.astar` has a coverage of 13.82% even for high threading overheads. This corresponds to the `non-DOALL` loop at `Way_.cpp:57`. Subject to other practical constraints, application of TLS on such loops can potentially yield performance gains. As evident from Figure 8, the above phenomenon occurs very seldom across a variety of applications.

More interestingly, we note that there is an exponential decrease in the performance potential of TLS with increasing threading overhead. This highlights the importance of accounting for the threading overhead while evaluating the efficacy of any TLS mechanism.

[7] Note that the results presented in [23, 24] are presented for single threaded execution.

[8] Nevertheless, the sensitivity "pattern" at the outermost loop level is similar.

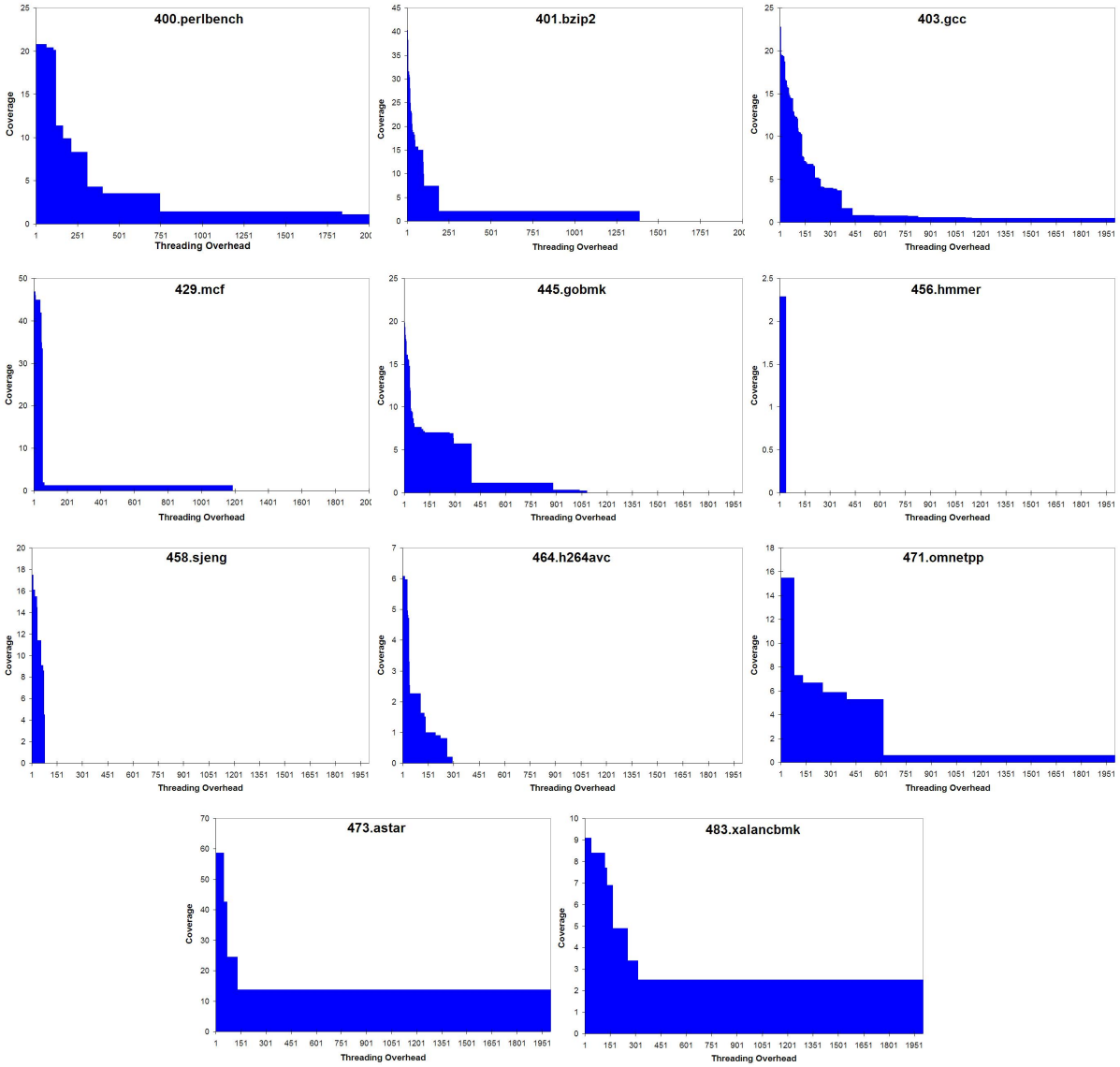


Figure 8. Senstivity of the performance potential of TLS (at the innermost loop level) w.r.t. to the threading overhead for applications in SPEC CINT2006

To much of our surprise, this has been largely overlooked in prior work. The high sensitivity of the TLS execution model w.r.t. threading overhead can provide a valuable guidance for developing high performance thread libraries.

The sensitivity results presented in Figure 8 correspond to unbounded number of processors and do not account for effects of other parameters such as misspeculation penalty et cetera. Consequently, in practice, we expect that the actual speedup achievable via TLS would be much lower than presented in Figure 8.

4.1.1 Impact of Loop Unrolling

From the discussion so far it is evident that the speedup achievable via TLS is limited to a great extent by the threading overhead. One of the ways to alleviate the adverse impact of threading overhead is to unroll the loop. Loop unrolling would increase the amount of computation contained in each iteration, thereby improving the computation/threading overhead ratio for each fork of a speculative thread. In a similar fashion, one could map more than one iteration during dynamic scheduling, referred to as *block scheduling* [29], to a speculative thread. This has the same "benefit" mentioned above. However, this has the following downsides:

❶ First, the increased computation per thread may result in higher destructive interference in the D-cache between the different threads. Furthermore, a speculative thread (which is aborted later on) may cause D-cache misses on the main thread which may lead to performance degradation!

❷ Second, more importantly, loop unrolling would result in a higher misspeculation rate which can potentially have an adverse affect on performance. This is due to the fact that the number of potential dependences between any two iterations of the unrolled loop increases with unrolling as explained below(note that the number of potential dependences corresponding to all the iterations in the unrolled loop is the same as in the loop before unrolling).

Let us consider the loop (taken from `401.bzip2:blocksort.c:314`) shown below, where `ISSET_BH(zz)` is defined as `(bhtab[(zz) >> 5] & (1 << ((zz) & 31))`. The correspo-nding data dependence graph is shown below. The dashed arrows rep-

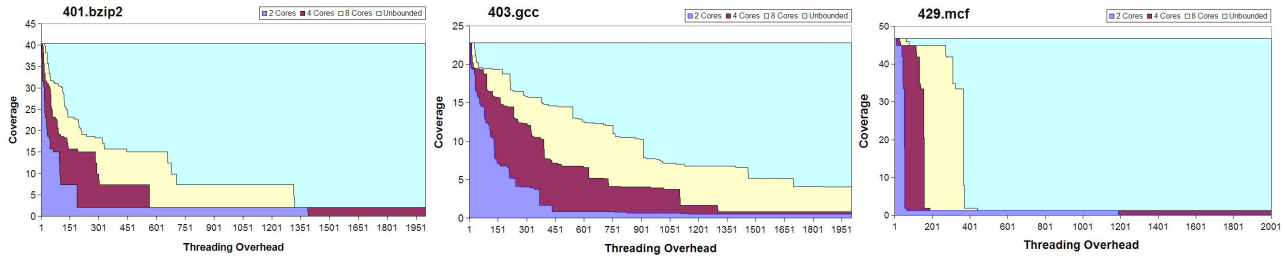
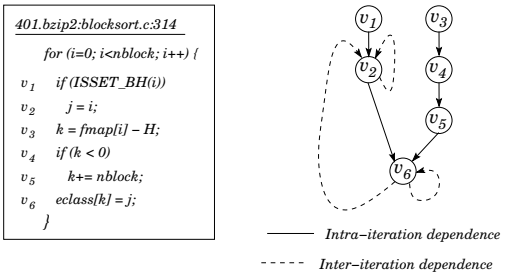


Figure 9. Variation in the performance potential of TLS with the use of large number of processors

resent the cross-iteration dependences. The dependence distance [30] for the different cross-iteration dependences is not shown as, in this case, the may or may not exist a dependence between any two iterations.



Let v_{i_k}, for $1 \leq i \leq 6$, denote the i-th operation of the k^{th} iteration of the loop, where $0 \leq k <$ *nblock*. On analysis, assuming privatization of the variable `k`, we observe that there exist the following potential cross-iteration dependences:

$$v_{2_l} \rightarrow v_{2_m}, \quad v_{6_l} \rightarrow v_{2_m} \quad \text{and} \quad v_{6_l} \rightarrow v_{6_m}$$

where $l < m$. Let us unroll the loop twice. The set of operations v_{i_l}, v'_{i_l}, for $1 \leq k \leq l$ and $0 \leq k <$ *nblock*, constitute the loop body of the unrolled loop. There exist the following, in addition to the above, potential cross-iteration dependences:

$$v_{2_l} \rightarrow v'_{2_m}, \quad v_{6_l} \rightarrow v'_{2_m}, \quad v_{6_l} \rightarrow v'_{6_m}$$

$$v'_{2_l} \rightarrow v_{2_m}, \quad v'_{6_l} \rightarrow v_{2_m}, \quad v'_{6_l} \rightarrow v_{6_m} \quad \text{and}$$

$$v'_{2_l} \rightarrow v'_{2_m}, \quad v'_{6_l} \rightarrow v'_{2_m} \quad v'_{6_l} \rightarrow v'_{6_m}$$

From above we note that speculative execution of the iterations of the unrolled loop entails higher degree of speculation owing to larger number of (potential) dependences between the different iterations of the unrolled loop. In general, unrolling the loop u times increases the number of potential dependences between any two iterations by u times.[9]

Let $p(v_{k_l}, v_{k_m})$ denote the conflict probability corresponding to the dependence $v_{k_l} \rightarrow v_{k_m}$. If $v_{k_l} \nrightarrow v_{k_m}$, then $p(v_{k_l}, v_{k_m}) = 0$. For instance, in the current example, $p(v_{1_l}, v_{1_m}) = 0$. The probability of *perfect* speculative execution, i.e., when there is no conflict for each dependence, of an iteration m is given by:

$$\prod_{\forall k} \prod_{\forall l,m} (1 - p(v_{k_l}, v_{k_m}))$$

Alternatively, the misspeculation probability can be computed as follows:

$$1 - \prod_{\forall k} \prod_{\forall l,m} (1 - p(v_{k_l}, v_{k_m}))$$

[9] Although some of the cross-iteration dependences in the unrolled loop may be redundant and be eliminated [31, 32].

Clearly, the misspeculation probability increases with unrolling. Therefore, we argue that loop unrolling does not necessarily improve the performance potential of TLS owing to an increase in misspeculation penalty. However, loop unrolling may be used to enable exploitation of higher levels of ILP [33]. Further discussion of this is beyond the scope of the paper.

Arguably, loop unrolling may be beneficial for `non-DOALL` loops with non-unit dependence distance. In such cases, it is critical to filter out the speedup achievable via true TLP while evaluating the unique performance potential of TLS. For this, the candidate loop is transformed into a doubly nested loop wherein the inner loop is a `DOALL` loop and the outer loop is a `non-DOALL` loop. The number of iterations in the inner loop is equal to the minimum dependence distance [30] of the loop. The analysis presented earlier in this subsection can be employed to determine the profitability of unrolling the outer loop during speculative parallelization,

4.1.2 Impact of Large Number of Processors

Another way to minimize the impact of threading overhead is to use a large number of cores. This helps to amortize the impact of the threading overhead. However, similar to loop unrolling, use of a large number of cores increases the misspeculation probability. Assuming an oracle TLS mechanism, we analyze this in the rest of subsection.

Let N_p denote the number of processors and $T_{\text{ovhd}}, T_{\text{iter}}, T_{\text{serial}}, T_{\text{parallel}}$ and N_{iter} denote the threading overhead, the average execution time of one iteration of the loop, the execution time of the loop when executed serially, the execution time of the loop when executed in parallel on N_p processors and the number of iterations respectively. Let us first consider the 2 processor case. The value of T_{serial} and T_{parallel} is given by:

$$T_{\text{serial}} = T_{\text{iter}} \times N_{\text{iter}}$$

$$T_{\text{parallel}} < (T_{\text{iter}} + T_{\text{ovhd}}) \times \frac{N_{\text{iter}}}{2}$$

Given the aforementioned assumption, speculative execution of 2 iterations is profitable subject to the following:

$$T_{\text{parallel}} < T_{\text{serial}}$$

On simplification, we get:

$$T_{\text{ovhd}} < T_{\text{iter}}$$

The above condition serves as the basis for determining the profitability of executing a `non-DOALL` loop speculatively. All the `non-DOALL` loops which do not meet this criteria are executed serially. Likewise, in case of 4 processors the profitability condition is given by:

$$T_{\text{ovhd}} < 3 \times T_{\text{iter}}$$

In general, given N_p number of processors, the profitability condition is given by:

$$T_{\text{ovhd}} < (N_p - 1) \times T_{\text{iter}}$$

Benchmark	CS-Only (%)		DDS-Only (%)		DVS-Only (%)		CS+DDS (%)		CS+DVS (%)		DDS+DVS (%)		CS+DDS+DVS (%)	
Integer Benchmarks														
400.perlbench	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	14.3	3.61	0.1	1.42	6.47
401.bzip2	0.1	0.1	0.1	0.1	0.1	2.55	0.1	0.7	0.1	12.4	0.1	12.36	0.1	12.31
403.gcc	0.1	0.92	0.1	0.1	0.1	0.1	0.1	0.1	0.1	3.96	0.1	0.1	0.1	14.14
429.mcf	0.1	0.6	0.1	0.1	0.1	0.1	0.1	0.1	0.1	28.6	0.1	0.1	0.1	17.7
445.gobmk	0.1	0.26	0.1	0.33	0.1	0.1	0.1	0.1	0.1	4.02	0.1	0.1	0.36	12.78
456.hmmer	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	2.29	0.1	0.1	0.1	0.1
458.sjeng	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	1.6	0.1	0.1	0.1	16.1
462.libquantum	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
464.h264avc	0.1	0.1	0.1	1.13	0.1	0.51	0.1	0.1	0.1	1.1	0.1	0.1	0.1	3.34
471.omnetpp	0.1	0.1	0.6	0.6	0.1	0.1	0.1	0.6	0.1	0.1	0.1	0.1	0.1	15.5
473.astar	0.1	0.1	0.1	18.11	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	13.82	40.62
483.xalancbmk	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	2.1	0.1	0.1	0.1	2.5	9.1
Geometric Mean	0.1	0.15	0.12	0.24	0.1	0.15	0.1	0.12	0.1	1.33	0.1	0.13	0.27	5.51
Floating Point Benchmarks														
410.bwaves	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
416.gamess	0.1	1.69	0.1	18.83	0.1	1.4	0.1	0.17	0.1	0.1	0.1	0.1	0.1	3.43
433.milc	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	1.4
434.zeusmp	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
435.gromacs	0.1	0.1	0.1	21.7	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	9.5
436.cactusADM	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
437.leslie3d	0.1	0.1	0.1	0.1	0.1	1.8	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
444.namd	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
447.dealII	0.4	0.7	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.6
450.soplex	0.1	1.67	0.1	3.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	4.37	6.54
453.povray	0.1	2.9	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	7.5
454.calculix	0.1	0.1	0.1	11.7	0.1	0.8	1.1	0.1	0.1	5.9	0.1	0.1	0.1	2.3
459.GemsFDTD	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	1.1	1.1
465.tonto	0.1	0.1	0.1	0.1	0.1	1.6	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.6
470.lbm	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
481.wrf2	0.1	0.6	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0.8
482.sphinx	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	1.3	0.1	0.1	0.1	0.1
Geometric Mean	0.11	0.19	0.1	0.34	0.1	0.18	0.1	0.1	0.12	0.21	0.1	0.1	0.17	0.62

Table 3. Performance dissection of different types of speculation for SPEC CINT2006 and SPEC CFP2006, assuming a threading overhead of 1000 cycles (gray columns) and 10 cycles

In case of unbounded number of processors, the profitability condition is given by:

$$T_{\text{ovhd}} < (N_{\text{iter}} - 1) \times T_{\text{iter}}$$

For 4 different scenarios, viz., $2, 4, 8$ and *unbounded* number of processors, we filtered out the non-profitable `non-DOALLs` to determine the performance potential of TLS at the innermost loop level. However, the analysis presented here can be easily extended to outermost loops. The same is shown in Figure 9. Due to space limitations, we show the variation in the performance potential of only three applications in SPEC CINT2006. From the figure we see that TLS performance potential increases with higher number of cores. However, the increase is rather small, being limited by the threading overhead. Assuming a threading overhead of 1000 cycles, the increase for 2, 4, 8 and unbounded number of cores is shown in Table 4. Recall that the results presented in both Figure 9 and Table 4 assume an oracle TLS mechanism and do not account for other limiting factors such as increased memory bus contention. Therefore, in practice, the performance potential of TLS would be even lower.

	2 cores (%)	4 cores (%)	8 cores (%)	Unbounded (%)
401.bzip2	2.08	2.08	7.38	40.32
403.gcc	0.6	3.89	7.5	22.83
429.mcf	1.3	1.3	1.3	46.9

Table 4. Variation in the performance potential of TLS (at the innermost loop-level) assuming a threading overhead of 1000 cycles

4.2 TLS Dissection

In this subsection, we present a dissection of the performance potential of the different types of TLS. 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 3 we present the dissection of the performance potential of each type of TLS subject to a threading overhead of 1000 cycles (see columns in gray) and 10 cycles. The columns in Table 3 correspond to the different types of TLS, viz., CS-Only, DDS-Only, DVS-Only, CS+DDS, CS+DVS, DDS+DVS and CS+DDS+DVS (see [3] for a detailed discussion of the above types of TLS).

An entry in a given cell of the table is equal to the coverage of the non-DOALL loops that can be (speculatively) 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 at the innermost loop level. For example, 2.1% of the total execution time of 483.xalancbmk can be profitably parallelized, subject to a threading overhead of 1000 cycles, with control and data value speculation when applied to the innermost loops. Note that the number 0.1 only signifies that the performance potential of the corresponding TLS type for the given application is negligible.

Conterintuitively, we observe that in some cases CS-Only, DDS-Only and DVS-Only have higher potential than CS+D-DS+DVS. For example, DDS-Only has the highest performance potential in the 416.gamess and 454.calculix benchmarks, for a threading overhead of 10 cycles. However, from the table we note that CS-Only, DDS-Only and DVS-Only have a marginal performance potential across the entire suite, see the row corresponding to the *geometric mean* speedup. Moreover, the performance potential of the different TLS types is much smaller with a realistic threading overhead of 1000 cycles.

Amongst the different combinations, CS+DDS+DVS seems to bear the most benefit, which in itself is limited to 0.27% and 0.15% for SPEC CINT2006 and CFP2006 respectively (see the last column corresponding to the geometric mean), for a realistic threading overhead of 1000 cycles (assuming an oracle TLS mechanism). On the other hand, for a threading overhead of 10 cycles, the geometric performance potential of CS+DDS+DVS is 5.51% and 0.62% for SPEC CINT2006 and CFP2006 respectively, for a threading overhead. From our analysis, we conclude that the uniquely achievable gain via TLS for applications in SPEC CINT2006 and CFP2006 is rather small.

4.3 Bounds on Conflict Probability

So far we analyzed the performance potential of TLS assuming perfect speculation. However, perfect speculation cannot be achieved in practice. In this subsection, we study the impact of misspeculation on the performance potential of TLS. This is very important as it facilitates to assess the applicability of TLS in presence of hard-to-predict dependence(s) and other practical constraints.

In general, the performance penalty incurred varies from one instance of misspeculation to another. This can be attributed to a variety of run-time factors such as the number of outstanding cache misses, non-deterministic interaction between the different threads. However, we show that there exists a relation between the threading overhead and the conflict probability for TLS to be profitable. We developed an analytical model to capture the above. Based on our model, we determine an upper bound on the conflict probability for each candidate (innermost) non-DOALL loop, for applications in SPEC CINT2006. Nonetheless, we find that the bounds for the outermost loops have similar characteristics as that of the innermost loops.

As proposed in prior works (see [P90-1]–[P90-41] in [1]), we assume the following thread execution model: if any of the points-of-speculation "misfires", then the speculative thread is squashed and the iteration is executed serially on the main (non-speculative) thread (refer to Figure 11). Figure 11(a) corresponds to non-speculative execution of the loop (only two iterations are shown for clarity purposes). Figure 11(b) corresponds to speculative execution of the loop with no conflicts; the speculative execution time in this case is equal to $T_{\text{ovhd}} + T_{\text{non-spec}}$. Figure 11(c) corresponds to speculative execution of the loop with conflicts; for a two processor case, the lower bound on the speculative execution time in this case is equal to $T_{\text{ovhd}} + 2 \times T_{\text{non-spec}}$. In practice, the speculative execution time would be much higher due to higher number of conflicts between multiple iterations. Modeling and discussion of other techniques such as the one proposed in [34] is beyond the scope of this paper.

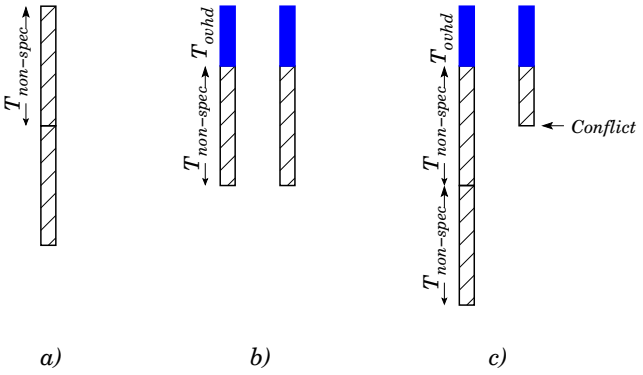


Figure 11. a) Serial execution b) Speculative execution with no conflict c) Speculative execution with conflict.

Let there be m *points-of-speculation*[10] on an iteration of a given non-DOALL loop. Let p_i denote the conflict probability of the i^{th} speculation point. Lastly, let $T_{\text{non-spec}}, T_{\text{spec}}$ denote the execution time of an iteration when executed non-speculatively and speculatively respectively. The lower bound on the speculative execution time T_{spec} can be modeled in the following way:

$$T_{\text{spec}} = (T_{\text{non-spec}} + T_{\text{ovhd}}) \prod_{1 \le i \le m} (1 - p_i) \; + \; (2 \times T_{\text{non-spec}} + T_{\text{ovhd}}) \left(1 - \prod_{1 \le i \le m} (1 - p_i) \right)$$

The first term on the right hand side corresponds to the case when none of the points-of-speculation "misfires". This is computed as the product of (i) the sum of non-speculative execution time and the threading overhead, (ii) the probability that none of the points-of-speculation "misfires". The second term on the right corresponds to the case when any one of the points-of-speculation "misfires". In such a scenario, the speculative thread is squashed and the iteration allocated to the speculative thread is executed serially by the non-speculative thread.

For TLS to be profitable, the following must hold:

$$T_{\text{spec}} < 2 \times T_{\text{non-spec}}$$

On simplification, we obtain

$$T_{\text{ovhd}} < T_{\text{non-spec}} \prod_{1 \le i \le m} (1 - p_i)$$

The above condition serves as a simple means to determine the profitability of speculative execution while accounting for both the

[10] A *point-of-speculation* corresponds to an incoming edge in the dependence graph.

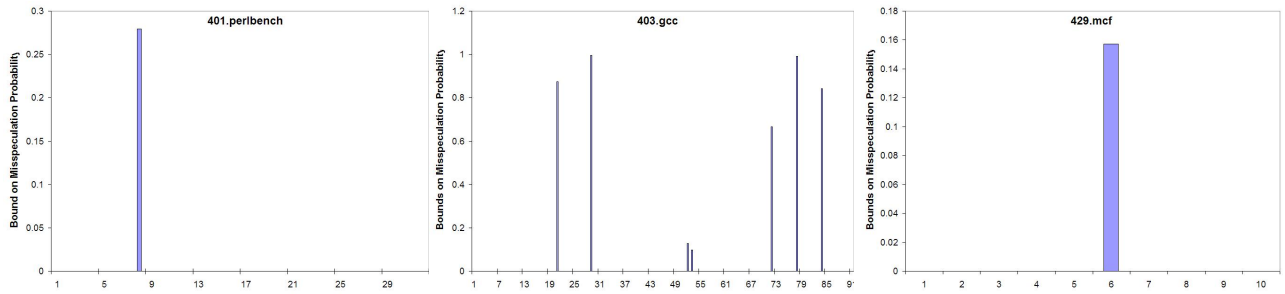


Figure 10. Bounds on misspeculation probability, assuming a threading overhead of 1000 cycles

threading overhead as well as the conflict probability. Clearly, a lower misspeculation conflict corresponds to a higher performance potential of TLS. When $p_i = 0$, for $1 \leq i \leq m$, the above condition reduces to the case of unbounded processors as discussed in subsection 4.1.

Assuming equal conflict probability for each dependence, i.e., $p_i = p_j = p$, for $1 \leq i, j \leq m$ and $i \neq j$, after simplification, we obtain the upper bound on p as follows:

$$p < 1 - \sqrt[m]{\frac{T_{\text{ovhd}}}{T_{\text{non-spec}}}}$$

For the simple case $m = 1$, characteristic of recurrence solvers, the upper bound on conflict probability is given by:

$$p < 1 - \frac{T_{\text{ovhd}}}{T_{\text{non-spec}}}$$

Based on the above condition, we obtained the upper bound on the conflict probability of all the `non-DOALL` loops with a coverage of more than 1% for all applications in SPEC CINT2006. The probability distribution is shown in Figure 10 (the x-axis corresponds to the loop numbers). From the figure we observe that for most of the loops the bound is zero. This implies that TLS is not profitable for such loops. Further, we observe that there do exist a few loops with non-zero probability. For instance, the upper bound on the conflict probability of loop #51 of `403.gcc` is 0.13. This signifies that speculative parallelization of this loop is profitable *iff* the conflict probability is less than 0.13. An estimate of the latter can be obtained via static[35]/dynamic profiling. On analysis we find that the coverage of such loops is rather low. As a consequence, the overall performance potential of TLS is rather limited, unlike what has been reported in prior work.

The model proposed in this subsection can be used in conjunction with profiling to determine whether a candidate `non-DOALL` loop should be speculatively parallelized or not. Note that the applicability of our model, by construction, is not limited to `non-DOALL` loop. The model can be easily extended and used for program regions in general.

5. Conclusion

Prior work has shown that speculative multithreading bears significant potential to boost performance. However, to much of our surprise, none of the prior works accounted for the impact of the threading overhead while evaluating the performance gain uniquely achieved via TLS. Our results show that in applications of SPEC CPU2006 the speedup achievable by means of TLS is highly sensitive to the same. Specifically, we find that the geometric mean performance potential of TLS (at the innermost loop level) is of the order of 6% for a threading overhead of 10 cycles, whereas it is of the order of 1% for a realistic threading overhead of 1000 cycles. Lastly, we presented an analytical model to determine upper bounds on the conflict probability for which TLS is beneficial. Even in this case, we find that the upper bound on conflict probability is very sensitive to the threading overhead. This provides a very valuable guidance to develop high performance thread libraries for efficient exploitation of sTLP. As future work, we plan to explore the trade-off between run-time and speculative parallelization.

6. Acknowledgments

The first author would like to thank Matteo Frigo, IBM, Austin for letting know about the work done in speculative computation in 1960s such as in IBM 7030 (a.k.a. STRETCH).

References

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

[2] S. F. Lundstrom and G. H. Barnes. A controllable MIMD architectures. In *Proceedings of the 1980 International Conference on Parallel Processing*, pages 19–27, St. Charles, IL, August 1980.

[3] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *Proceedings of the 20th ACM International Conference on Supercomputing*, pages 24–35, Cairns, Australia, 2006.

[4] SPEC CPU2006. http://www.spec.org/cpu2006.

[5] G. M. Amdahl. Validity of the single processor approachtoachieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[6] Open Research Compiler for Itanium™Processor Family. http://ipf-orc.sourceforge.net/.

[7] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[8] P. Jouvelot. Semantic parallelization: a practical exercise in abstract interpretation. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 39–48, Munich, West Germany, January 1987.

[9] P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Proceedings of the 3rd ACM International Conference on Supercomputing*, pages 186–194, Crete, Greece, June 1989.

[10] D. J. Quinlan, M. Schordan, Q. Yi, and B. R. de Supinski. Semantic-driven parallelization of loops operating on user-defined containers. pages 524–538, College Station, TX, October 2003.

[11] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[12] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. ILP versus TLP on SMT. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, page 37, Portland, OR, 1999.

[13] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance

scientific computing. In *Proceedings of the 14th annual workshop on Microprogramming*, pages 183–198, Chatham, MA, December 1981.

[14] A. Nicolau. Percolation scheduling. In *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[15] K. Kyriakopoulos and K. Psarris. Efficient techniques for advanced data dependence analysis. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 143–156, St. Louis, MO, 2005.

[16] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. Technical Report 87-873, Dept. of Computer Science, Cornell University, 1987.

[17] 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.

[18] 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.

[19] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in tls chip multiprocessors: Microarchitecture and compilation. In *Proceedings of the 19th ACM International Conference on Supercomputing*, pages 179–188, Cambridge, MA, 2005.

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

[21] R. Gerber, A. J. C. Bik, K. B. Smith, and X. Tian. *The Software Optimization Cookbook, Second Edition*. Intel Press, 2006.

[22] I. Piumarta and F. Riccardi. Optimizing direct-threaded code by selective inlining. *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[23] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan. A compiler framework for speculative analysis and optimizations. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 289–299, San Diego, CA, 2003.

[24] X. Dai, A. Zhai, W.-C. Hsu, and P.-C. Yew. A general compiler framework for speculative optimizations using data speculative code motion. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 280–290, San Jose, CA, 2005.

[25] P. Marcuello and A. González. Thread spawning schemes for speculative multithreading. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 55–64, Boston, MA, February 2002.

[26] Intel® VTune™ Performance Analyzer 8.0 for Windows. http://www.intel.com/cd/software/products/asmo-na/eng/vtune/219898.htm.

[27] U. Drepper. The Native POSIX Thread Library for Linux. http://people.redhat.com/drepper/nptl-design.pdf, February 2005.

[28] EPCC OpenMP Microbenchmarks. http://www.epcc.ed.ac.uk/research/openmpbench/openmp_index.html.

[29] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, 1985.

[30] U. Banerjee. *Dependence Analysis*. Kluwer Academic Publishers, Boston, MA, 1997.

[31] D.-K. Chen and P.-C. Yew. Redundant synchronization elimination for DOACROSS loops. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 477–481, Cancùn, Mexico, 1994.

[32] M. Philippsen and E. Heinz. Automatic synchronization elimination in synchronous FORALLs. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

[33] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[34] H. Akkary, S. T. Srinivasan, and K. Lai. Recycling waste: exploiting wrong-path execution to improve branch prediction. In *Proceedings of the 17th ACM International Conference on Supercomputing*, pages 12–21, San Francisco, CA, 2003.

[35] T. Ball and J. Larus. Branch prediction for free. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313, Albuquerque, NM, June 1993.