Compiler-driven Dependence Profiling To Guide Program Parallelization

Peng Wu[1], Arun Kejariwal[2], Călin Caşcaval[1]

[1] Programming Models and Tools for Scalable Systems Lab
I.B.M. T.J. Watson Research Center
Yorktown Heights, NY 95098, USA
[2] Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697, USA

Abstract. As hardware systems move toward multicore and multi-threaded architectures, programmers increasingly rely on automated tools to help with both the parallelization of legacy codes and effective exploitation of all available hardware resources. Thread-level speculation (TLS) has been proposed as a technique to parallelize the execution of serial codes or serial sections of parallel codes. One of the key aspects of TLS is task selection for speculative execution.

In this paper we propose a cost model for compiler-driven task selection for TLS. The model employs profile-based analysis of *may*-dependences to estimate the probability of successful speculation. We discuss two techniques to eliminate potential inter-task dependences, thereby improving the rate of successful speculation. We also present a profiling tool, **DProf**, that is used to provide run-time information about *may*-dependences to the compiler and map dynamic dependences to the source code. This information is also made available to the programmer to assist in code rewriting and/or algorithm redesign.

We used **DProf** to quantify the potential of this approach and we present results on selected applications from the SPEC CPU2006 and SEQUOIA benchmarks.

1 Introduction

Thread-level speculation (TLS) [28,16,26,12] is one technique that has been proposed for parallelizing sequential codes to exploit parallel and multi-core architectures. Parallelization using TLS consists of selecting regions of code to execute in parallel, relying on the system to detect dependence violations and re-execute the conflicting sections such that sequential execution semantics is preserved. Typically the code regions are loop iterations and function continuations. A number of researchers made the case that automatically speculating on inner loops at the granularity of single iteration is not very effective for the applications in the SPEC CPU2006 benchmark suite, and gives little advantge over a state-of-the-art parallelizing compiler [15]. This highlights the importance of task selection towards the efficacy of TLS. Task selection can be done either automatically using a compiler, or manually through programmer annotations. Liu et al. [18] and Johnson et al. [14,13] propose mechanisms to automatically identify most profitable tasks for speculation through profiling information [14,18]

or empirical search [13]. On the other side of the spectrum, von Praun et al. [30] argue for user annotation of the speculative tasks and provide a tool that can classify program sections and recommend task placement directives.

It is obvious that programmers need tools to help them make parallelization decisions. This includes choosing a suitable parallelization strategy for a given application, e.g., speculative vs. non-speculative; task selection for speculation; or algorithm restructuring to expose parallelism. We propose a new model that uses compiler analysis and profiling to guide parallelization and task selection, in an attempt to reach a middle ground: the compiler and tools provide as much information as possible and prune the space, so that the user could focus on those parts of the application that may need rewriting and algorithm redesign.

In this paper, we present a compiler-driven approach for program dependence profiling and a cost model to identify loops suitable for TLS parallelism. One metric used in the cost model is the distance between consecutive dependent iterations [21], referred to as the *independence window*. If the independence window is larger than the speculation window (the number of tasks that are potentially executing in parallel at any point in time), the dependence does not affect TLS effectiveness. The Independence window is a dynamic property of a loop since it depends on the iteration schedule.

We developed a dependence profiler, referred to as **DProf**, to measure dependence probability and independence window. Profiling can be implemented using a dynamic binary instrumenter [29] or using compiler instrumentation. When using a binary instrumenter, program properties known to the compiler are lost or hard to obtain at binary level; "transferring" the dependence information from a trace to the compiler is quite involved: the binary instrumenter collects physical addresses which need to be mapped to the variables in the program. In Section 3, we present a compiler-based approach for dependence profiling that overcomes these limitations. We present the dependence and independence window profile obtained by **DProf** for selected programs from the SPEC CPU2006 and SEQUOIA benchmarks. Both the independence window and dependence data measured by the profiler provide useful feedback to the compiler to perform dependence tolerating transformations or to the programmer to restructure algorithms for parallelism.

The main contributions of this paper are:

- A static model for TLS profitability that is used by the compiler to select tasks for speculation;
- A compiler-driven approach for program dependence profiling;
- Two techniques – independence windows and dependence clustering – for increasing the profitability of TLS.

The rest of the paper is organized as follows: Section 2 describes our static model for TLS. The design of **DProf** is described in detail in Section 3 and its applications are discussed in Section 4. Previous work is discussed in Section 5. Finally, in Section 6 we summarize with directions for future work.

2 Static modeling of TLS profitability

In this section we present a cost model used by the compiler to determine profitable TLS code sections. There are three main factors that determine the profitability of speculative execution:

1. Conflict probability (C): defines what is the probability that two speculative tasks access the same data, and at least one is a write, such that they conflict and the speculative execution must be squashed. The conflict probability is a function of the data dependences in the task and of the task size. We discuss the conflict probability in detail in Section 2.1;
2. Speculative spawn and commit overheads (O) which are system dependent costs of speculation, for both successful and aborted execution;
3. Task sizes (S_i), which determine the the fraction of useful work, as well as influence conflict probability – the larger the size the larger the set of data accessed, and efficiency due to load balancing issues.

A TLS section of code is profitable if the time required for parallel execution (T_p), including the overheads, is less than the time required for serial execution (T_s), i.e., $T_p < T_s$. T_s and T_p for a set of N tasks running on P processors can be expressed as follows:

$$T_s = \sum_{i}^{N} S_i \tag{1}$$

$$T_p = \sum_{k} \frac{N \times (1-C)^k}{P} (\max_i(S_i) + O) \tag{2}$$

$$T_p = \frac{N \times (1-C)}{P} (\max_i(S_i) + O) + \sum_{i}^{N \times C} (S_i + O) \tag{3}$$

The serial execution time T_s (Eqn. 1) is the sum of all tasks. The parallel execution time T_p (Eqn. 2) takes into consideration the number of processors and the conflict probability. Eqn. 3 is a simplified version of Eqn. 2 that uses the following assumption: $N \times (1-C)$ tasks can all execute in parallel ($\max_i(S_i)$ determining the parallel execution time), with no other overhead in addition to the spawn and commit overhead O; the other $N \times C$ tasks are all serialized. Of course, this is a conservative assumption, because within the remaining tasks there may be independent sets of tasks, but between the overhead of spawning a task multiple times, and the diminishing returns of executing conflicting task versus just serializing execution, we select the latter.

The compiler may obtain the needed parameters for the model either statically using analysis, or through profiling, as follows:

- ❒ New data dependence analysis that takes into consideration dependence probabilities; or profiling information that estimates the conflict probability for a set of tasks;
- ❒ A cost model for task sizes; or profiling information that quantifies the sizes of tasks;

❒ System latencies and overheads for TLS support.

For this paper, we explore the effectiveness of the cost model, and thus we collect profiling information using **DProf** (see Section 3) that provides feedback information on data dependences and task sizes.

2.1 Conflict Probability

We use the metric of conflict probability to determine the likelihood of the speculation execution of a code region success. A conflict probability of 0 implies that the region is independent of all other regions with which it has the potential to run in parallel, while a conflict probability of 1 guarantees that a conflict will happen and the speculation will fail. Intuitively, the larger a code region, the higher the probability of conflict. However, there are many cases in which large code regions are independent, e.g., iterations of DOALL loops.

The conflict probability is computed using the data dependence density metric. In [29], von Praun et al. used the data dependence density metric to determine the available parallelism in an application. They argue that the amount of exploitable parallelism in the application is dependent on the scheduling of threads, and classified application phases into three categories: high-, medium-, and low-dependence density. The low-dependence density regions are the most profitable for speculation. Because we are focusing on loop iterations, and considering two dependent iterations t and s, we can simplify the data dependence density computation from [29] and use the following formula for conflict probability:

$$C(t) := \frac{\sum_{\forall s, has\_dep(t,s)} S_s}{\sum_{i}^{N} S_i} \tag{4}$$

In this paper, the conflict probability is used directly with the speculation overheads and task sizes (Eqn. 3) to select the profitable tasks.

In addition, the compiler can also increase the probability of successful speculation by taking advantage of patterns of dependences. We present two such techniques – the independence window and dependence clustering.

2.2 Independence Windows

An *independence window* is a set of consecutive iterations that are independent of each other. Iterations in the set – the independence window – can be executed in parallel. We call the cardinality of the set the width of an independence window.

The entire iteration space can be viewed as a partitioned set of independence windows. Consider the loop shown in Figure 1 (a) and the corresponding iteration space shown in Figure 1 (b). The loop has a loop-carried dependence with dependence distance of 4. Therefore, every 4 iterations, marked with dashed arrows in Figure 1 (b), can be executed in parallel. For loops with a constant

dependence distance of n, the iteration space is equally partitioned into independence windows of width n. For irregular loops, the independence window is a dynamic property determined by the iteration schedule.

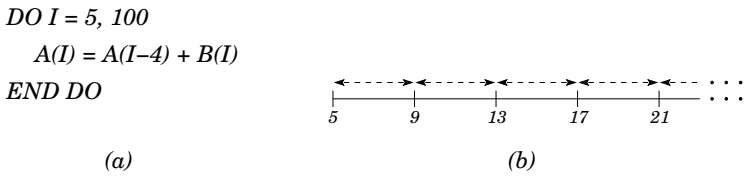

Fig. 1. An example illustrating the independence window

Assuming that tasks are scheduled and retired at a uniform rate at the granularity of one iteration, a loop with an independence window of width n can be parallelized with zero conflicts using no more than n speculative threads. In other words, the width of independence window gives the theoretical upper bound on the size of non-conflicting speculation.

We use the width of independence windows as a metric of dynamic parallelism, especially for loops that do not have a uniform dependence distance. For such loops, the width of the independence window varies across the iteration space, e.g., due to multiple dependences occurring at different intervals. We use the profiler to empirically determine the independence window width of such loops. In our profiler, we compute the maximum, minimum, and average widths of independence windows to capture dynamic widths of independence windows.

The compiler can exploit independence windows by throttling the speculation, such that dependences are naturally satisfied. This can be accomplished by either of the following methods:

- By the compiler inserting explicit synchronization, similar to the technique described in [32]; or
- Providing hints to the hardware for dynamic task merging [24]; or
- Inserting conditional spawn instructions [9] if the hardware supports it.

2.3 Dependence Clustering

Recently it was shown that the profitability of TLS is highly sensitive to the threading overhead [15]. The analysis assumed dynamically scheduling, wherein iterations of a loop are allocated one at a time. Arguably, one can unroll a loop or employ chunk scheduling, where multiple iterations are allocated either statically or dynamically to a processor, to tolerate the high threading overhead. On the other hand, this increases the probability that two speculative threads conflict with each other. However, this implicitly assumes a uniform distribution of dependences across the iteration space: if there exists 2 dependences between iterations i and $i+1$, then iterations j and $j+1$, where $i \neq j$ will be dependent. In other words, the width of the independence window is "fixed". This assumption may not hold for loops containing conditionals, subscripted subscripts and/or

function calls. For example, let us revisit the loop shown in Figure 1. The value of the variable `i` may be the same for the first 10 iterations and different in the remaining iterations; consequently, the first ten iterations have to be executed serially, whereas the remaining iterations can be executed in parallel. In such a case, we say that loop-carried dependences are *clustered* amongst the first ten iterations.

In general, the objective of *dependence clustering* is to determine, regions of the iteration space (of a given loop) with large independence windows. If such regions exist, then the iteration space is partitioned to isolate such regions and these regions are subsequently parallelized via TLS.

2.4 Summary

In Figure 2 we present a pictorial view of profitability analysis of TLS. We consider a hierarchy for non-`DOALL` loops. The first level filtering is done based on the threading overhead. Specifically, non-`DOALL` loops with small amount of computation in the loop body are filtered as these loops are *non-tolerant* (box **A** in the figure) to the threading overhead. The second level filtering is done based on conflict probability, using MinIWW (Minimum independence window width). Loops with MinIWW = 1 are classified as *non-profitable*, box $\boldsymbol{B_1}$ in the figure [15]. The rest of the loops are classified as *profitable*, box $\boldsymbol{B_2}$ in the figure.

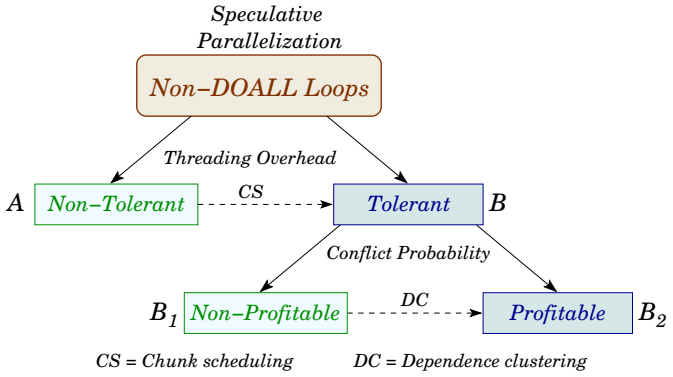

Fig. 2. Role of dependence clustering in facilitating speculative parallelization

Amongst the loops belonging to box **A**, we detect loops with high iteration counts. Such loops can be migrated to box **B** via chunk scheduling; it is important to note that these loops cannot be directly migrated to box $\boldsymbol{B_2}$ as chunk scheduling may result in an increase in conflict probability. Next, amongst the loops belonging to box $\boldsymbol{B_1}$, we detect loops with large MaxIWW (Maximum independence window width), exemplified by the loop in `429.mcf`, at `implicit.c:381` whose profile of independence windows is shown in Figure 8. Then, parts of the iteration space (of such loops) with large value of MaxIWW are peeled and classified as profitable for TLS. The percentage of execution time spent in the peeled portions of the iteration space correspond to the performance potential of TLS. Recall that, for given a loop, the existence of independence windows with large widths and their position in the iteration space may be dependent on the input data set. Therefore, independence windows (with large widths) detected using a training data set cannot be parallelized statically.

3 Design of DProf

DProf consists of two components: a compiler-driven instrumenter that selects loops and instruments memory references for dependence profiling, and a runtime library that logs references and profiles dynamic dependences and independence window.

3.1 DProf instrumenter

The instrumenter is based on the optimizing and parallelizing IBM XL (production) compiler. The compiler first analyzes candidate loops to decide whether they are parallel. Parallel loops are not profiled. The rest of candidate loops may be selected for profiling based on the likelihood of the dependences detected by the compiler and other characteristics of the loop. For instance, a loop with small dependence distances may not be selected for profiling if static information is sufficient to determine the loop as not a good candidate for parallelization.

For loops that are selected for profiling, the compiler transforms every memory reference that carries may-dependences into a call `__profile_access`, and marks the start, the end, and the backedge of a (possibly nested) loop by calls to `__profile_boundary`.

The compiler does not instrument references to induction, reduction, and private variables as dependences carried by these references can be eliminated via compiler transformations. To reduce profiling time, the compiler may not instrument references whose loop-carried dependences can be represented by dependence distances. In this case, the dependence information is recorded by the instrumenter and is later combined with profiled dependences to produce a complete dependence report.

3.2 DProf runtime library

The runtime library profiles a set of properties for loops such as whether a given loop is parallel, statistics of independence window size, and dependence properties such as the source, sink, and the frequency of dependences or dependence distances.

The profiler maintains a set of read- and write-logs for each loop: R_{curr}/W_{curr} for the current iteration being profiled, R_{indep}/W_{indep} for iterations in the current independence window, and R_{other}/W_{other} for iterations prior to those in the current independence window. Loop data structures are kept in a stack so that nested loops can be profiled in one pass.

For each `__profile_access` call, the profiler logs the reference to R_{curr} or W_{curr} accordingly, unless the access is a read and the memory address is already in W_{curr} (i.e., the read is private to current iteration).

For each `__profile_boundary` call that marks a loop backedge, the profiler detects loop-carried dependences by comparing R_{curr} against W_{indep}. If the intersection of the two sets is not a void set, then a true dependence is detected, and the current independence window is reset to start from the current iteration after the sets R_{indep}/W_{indep} are merged to R_{other}/W_{other} respectively. Otherwise, the current independence window is grown by one iteration. Note that, in this algorithm, only true dependences and dependences to references in the latest independence window are detected.

3.3 Source Mapping

The profiler reports source-level information corresponding to the source and sink of the dynamic dependences being profiled. Such information provides a valuable guidance to the programmer to make parallelization decisions and even eliminate these dependences by making source code changes.

The source-level mapping is maintained by the instrumenter which associates each call site of `__profile_access` and `__profile_boundary` with a unique id. The instrumenter also generates a file, which is later used by the profiler, to map ids to its associated source-level information and other properties obtained by the compiler. Since the instrumenter is a part of the compiler, the same mechanism can be used to map profiled properties back to the compiler.

4 Benchmark Evaluation

We implemented **DProf** on top of the development code base of the IBM XL compiler. Essentially, an instrumentation phase is added to an optimizing component of the XL compiler called **TPO** that performs high-level optimization including parallelization. The profiler is implemented as a runtime library that is linked with the instrumented binary.

4.1 Dependence Profiling

We used **DProf** to study the parallelization potential of selected applications from the SEQUOIA [1] and SPEC CPU2006 benchmarks. We profile only hot loops that are not parallelized by the compiler. For example, three applications from SPEC206, `bwaves`, `libquantum`, and `cactusADM`, are parallelized by the compiler, and thus excluded from the study.

We focus on four applications and provide a detailed discussion of their dependence profile characteristics and how they affect speculation profitability. A summary of the dependence characteristics obtained using **DProf** is given in Table 1. Other applications are not profiled due to a lack of hot for-loops (while-loops are not profiled due to limitation of the current implementation), or due to hot loops that are obviously non-parallel (e.g., containing I/O operations).

Benchmark	Hot Function	% exec	Profiling Summary
`lammps`	`pair_eam::compute()`	90%	parallel with array reduction
	`Neighbor::half_bin_newton()`	8%	serial due to scalar dependence
`gromacs`	`innerf.f:3932`	57%	inner loop parallel, outer loop serial
`hmmer`	`fast_algorithm.c:119`	90%	both inner/outer loops are serial
`mcf`	`implicit.c:265`	10%	both inner/outer loops are serial

Table 1. Summary of dependence characteristics obtained using **DProf**

SEQUOIA/lammps The hottest function, `pair_eam::compute()`, covers 90% of the execution time. There are two hot loops in this function that exhibit similar dependence patterns. Neither of them is parallelized by the compiler due to inadequate pointer aliasing information.

Figure 3 shows the fragment of the first hot loop, where the outer i-loop traverses a list of atoms and the inner k-loop traverses the neighbor list of each

```
164   for (i = 0; i < nlocal; i++) {
      ...
168     itype = type[i];
169     neighs = neighbor->firstneigh[i];
170     numneigh = neighbor->numneigh[i];
171
172     for (k = 0; k < numneigh; k++) {
173       j = neighs[k];
          ...
180       if (rsq < cutforcesq) {
            ...
188         rho[i] += ((coeff[3]*p + coeff[4])*p + coeff[5])*p+coeff[6];
189         if (newton_pair || j < nlocal) {
190           coeff = rhor_spline[type2rhor[itype][jtype]][m];
191           rho[j] += ((coeff[3]*p + coeff[4])*p +coeff[5])*p+coeff[6];
192         }
193       }
194     }
195   }
```

Fig. 3. Source code of 1st hot loop in `pair_eam::compute()`

item. Figure 4 gives the profiling report of the outer i-loop shown in Figure 3. The loop has an average independence window of 1 iteration. The narrow independence window is due to the tight dependence caused by read-modify-write to `rho[j]` at line 191, and by the conflict between `rho[i]+=` at line 188 and `rho[j] +=` at line 191. According to these stats, if consecutive iterations of the i-loop are scheduled as TLS tasks, then the conflict rate would be almost 100%.

```
Loop <1> at line 164 has 101 invocations, average 32000 iter/invoc (min=32000, max=32000)
  The loop has minIndep = 1 maxIndep = 5 avgIndep = 1
  - Detected a true dependence with frequency of 43.7679% between "this->rho[i]" (line# 188) and "this->rho[j]" (line# 191) in "pair_eam.cpp"
  - Detected a true dependence with frequency of 51.4117% between "this->rho[j]" (line# 191) and "this->rho[i]" (line# 188) in "pair_eam.cpp"
  - Detected a true dependence with frequency of 95.7514% between "this->rho[j]" (line# 191) and "this->rho[j]" (line# 191) in "pair_eam.cpp"
```

Fig. 4. Profiling report for hot loop in `pair_eam::compute()`

Note that all the loop-carried dependences are between the statements of the form `rho[x] +=` . In other words, elements of array `rho` are reductions. This loop can be parallelized by parallelizing the reduction [20], with the caveat that the compiler analysis needs to be extended to handle different subscripts.

CPU2006/gromacs The hottest loop in `gromacs` is in `innerf.f` at line 3932. The loop is a doubly nested and covers 57% of the total execution time. The loop nest contains many array references through subscript arrays (e.g., `faction(jjnr(k)-1)`), thus dependences on this loop nest cannot be detected statically. With the training input set, the inner loop has an average trip count of 28 iterations, ranging from 1 to 173. The loop is profiled to be parallel. Consequently, the loop is marked as profitable, with respect to conflict probability, for TLS.

The outer loop has an average trip count of 1250 iterations, ranging from 4 to 13891. **DProf** reports 30 pairs of true dependences for this loop and an independence window of 1 iteration. Half of the dependences have very low frequency (less than 1%). The rest have frequencies ranging from 18% to 100%. Figure 5 shows fragments of the loop, where all high frequency dependences occur between line 4140 and 4154 at the bottom of the outer loop.

All high frequency dependences occur among statements with array element reduction pattern (i.e., of the form of `a[x] +=`). Of them, updates to elements of arrays `Vc`, `Vnb`, and `fshift` are true reductions. Updates to `faction` exhibit

```
3932   do n=1,nri
         ...
         ii3 = 3*iinr(n)-1
         is3 = 3*shift(n)+1
3961     do k=nj0,nj1
         ...
4139     end do
4140     faction(ii3)     = faction(ii3) + fix1 /* dep freq 51% */
4141     faction(ii3+1)   = faction(ii3+1) + fiy1 /* dep freq 51% */
         ...
4148     faction(ii3+8)   = faction(ii3+8) + fiz3 /* dep freq 51% */
4149     fshift(is3)      = fshift(is3) + fix1+fix2+fix3 /* dep freq 18% */
4150     fshift(is3+1)    = fshift(is3+1) + fiy1+fiy2+fiy3 /* dep freq 18% */
4151     fshift(is3+2)    = fshift(is3+2) + fiz1+fiz2+fiz3 /* dep freq 18% */
4152     ggid             = gid(n)+1
4153     Vc(ggid)         = Vc(ggid) + vctot /* dep freq 100% */
4154     Vnb(ggid)        = Vnb(ggid) + vnbto /* dep freq 100% */
4155   end do
```

Fig. 5. Source code of the hot loop at line 3932 in innerf.c

a more complex pattern. Besides the reduction updates to elements of `faction` between line 4140 and 4148, there are additional reads and writes to `faction` in the inner k-loop that are not in the reduction form; however, these references to `faction` lead to very low frequency conflict ($\tilde{1}\%$) with those references to `faction` between line 4140 and 4148. This means the outer loop can not be easily parallelized by reduction handling and requires TLS support.

CPU2006/hmmer We now illustrate the dependence profiling of the hot loop in `hmmer` taken from `fast_algorithms.c`:119. The loop covers 90% of the execution time and is shown in Figure 6. In this loop, variables `mmx, dmx, xmx` and `imx` are declared as `int**`. Due to the lack of aliasing information, the compiler can not determine the precise dependence information for references in the loop.

```
120   for (i = 1; i <= L; i++) {
121     mc    = mmx[i];
122     dc    = dmx[i];
123     ic    = imx[i];
124     mpp   = mmx[i-1];
125     dpp   = dmx[i-1];
126     ip    = imx[i-1];
127     xmb   = xmx[i-1][XMB];
        ...
134     for (k = 1; k <= M; k++) {
135       mc[k] = mpp[k-1]   + tpmm[k-1];  /*flow to mc[k-1] line 143 */
136       if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
137       if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
138       if ((sc = xmb  + bp[k])         > mc[k])  mc[k] = sc;
139       mc[k] += ms[k];
140       if (mc[k] < -INFTY) mc[k] = -INFTY;
141
142       dc[k] = dc[k-1] + tpdd[k-1]; /* flow to dc[k-1] line 143 */
143       if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
144       if (dc[k] < -INFTY) dc[k] = -INFTY;
145
          ...
152     }

158     xmx[i][XMN] = -INFTY;  /* flows to xmx[i-1][XMN] at line 159 */
159     if ((sc = xmx[i-1][XMN] + hmm->xsc[XTN][LOOP]) > -INFTY)
160       xmx[i][XMN] = sc;
        ...
189   }
```

Fig. 6. Source code of the loop at `456.hmmer:fast_algorithm:119`

With the training input set, the inner loop is profiled to have a trip count of 100 iterations, and has an independence window of 1 iteration. Two pairs of dependences are detected for this loop on the `mc` and `dc` variables. The outer loop has an average trip count of 491 iterations, ranging from 7 and 1328 iterations. The loop also has an independence window of 1 iteration. Ten pairs of dependences are detected with 100% conflict rate. This is an example of a loop that is not a good candidate for speculative parallelization.

CPU2006/mcf The hottest loop in `429.mcf` (`psimplex.c:59`) is a while-loop thus is not profiled. The loop at `implicit.c:265` (see Figure 7) covers 10% of the execution time. The loop is profiled to have an independence window of 1 iteration. The profiler reports 18 pairs of dependences for this loop. Of them, two dependences have a conflict rate of 100%, one on the variable `first_of_sparse_list` at lines 269 and 270 and the other between the pointer accesses `arcout->head->firstout->head->arc_tmp` at line 270 and `tail->arc_tmp` at line 289.

```
265 for( ; i < trips; i++, arcout += 3 )
266 {
267     if( arcout[1].ident != FIXED )
268     {
269         arcout->head->firstout->head->arc_tmp = first_of_sparse_list;
270         first_of_sparse_list = arcout + 1;
271     }
272
273     if( arcout->ident == FIXED )
274         continue;
275
276     head = arcout->head;
277     latest = head->time - arcout->org_cost
278         + (long)bigM_minus_min_impl_duration;
279
280     head_potential = head->potential;
281
282     arcin = first_of_sparse_list->tail->arc_tmp;
283     while( arcin )
284     {
285         tail = arcin->tail;
286
287         if( tail->time + arcin->org_cost > latest )
288         {
289             arcin = tail->arc_tmp;
290             continue;
291         }
    ...
310 }
```

Fig. 7. Source code of the loop at `429.mcf:implicit.c:265`

4.2 Determining Independent Clusters

In this subsection, we present a case study to illustrate the use of **DProf** to determine independent clusters. Figure 8 shows the profile of independence windows for four different instances of the loop in `429.mcf`, at `implicit.c:381`, using the training and reference input data sets. All the references in the loop were instrumented, except for the variable`susp` – a reduction recognized by the compiler.

The x-axis in each subfigure of Figure 8 represents the number of independence windows and the y-axis represents the width of an independent window. From Figure 8 we see that parts of the iteration space have large independence window widths. For instance, let us consider the profile shown in the second row and right column. We note that the widths are very small towards the left of the x-axis and is more than 7500, on an average, on the right side of the x-axis. In such cases, we say that dependences occur in clusters. This behavior can be exploited for speculatively parallelization, as discussed earlier in Section 2. Interestingly, we note that the profile of independence windows varies from one instance to another and is significantly different for the training and reference data sets.

For the hot loops we studied, however, we find that the dependence window profile is quite regular: the loops are either fully parallel or have a constant independence window width of 1.

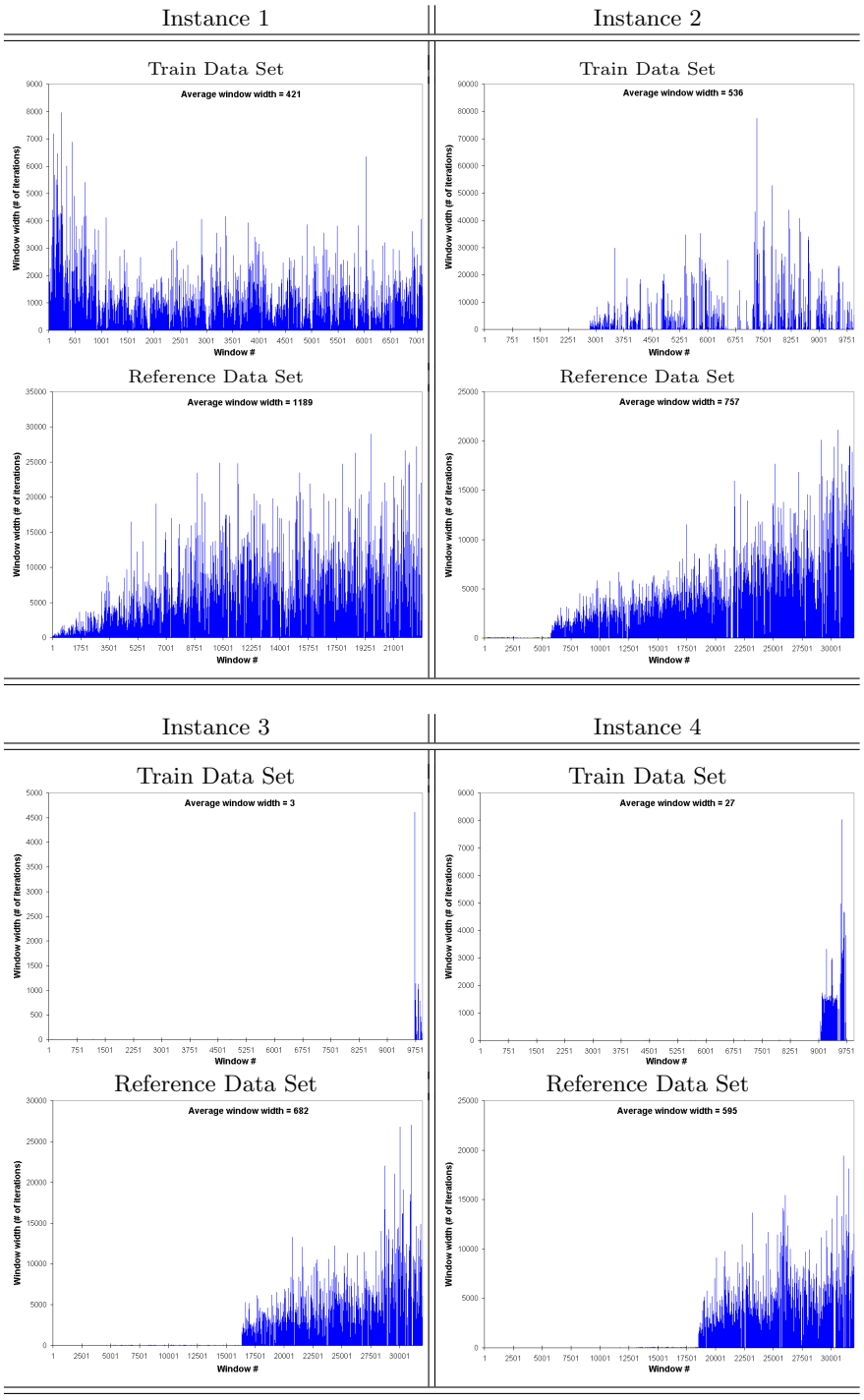

Fig. 8. Illustration of dependence clustering

5 Previous Work

Task-level speculative computation has been long proposed as a means for extracting higher levels of parallelism [3]. With the emergence of multithreaded processors [23], many researchers have proposed the use of threads for exploiting speculative parallelism in both hardware and software [11,25,2].

To ensure profitable speculation, most prior work use profile-based cost models for task decomposition and selection, while others rely on hardware mechanism [4,5,33,27], probabilistic static analysis (e.g., dependence probability [6]), compiler heuristics [28] for task generation. The rest of the section will focus on related work in software profile-based cost models, which our work belongs to.

In [8], the compiler uses dependence profile for task selection and for partitionning speculative loops into serial and parallel portions. The profiler tracks both intra- and carried- true dependences for speculative loops. Carried-dependences are used to guide the parition of loop bodies into a serial and a parallel portion. Since dependences originated from the serial portion do not trigger roll-back in the parallel portion, the key part of their framework is to move source computation of frequent dependences (called *violating candidate*) to the serial portion through instruction reording. A cost model is used to select the optimal loop partition, which is based on the size of serial portion and the misspeculation cost of the parallel portion. The latter is computed by combining re-execution cost of individual nodes weighted by probabilities of carried-dependences (for violating candidates) and intra-iteration dependences (for others).

In [19], the POSH compiler uses profiling for task selection. The profiler builds a rudimental timing model for TLS execution from the sequential execution. It assigns timestamps to each instruction as if the tasks were executed in parallel, and detect task squashes by comparing timestamps of conflicting memory accesses. The profiler does not collect individual dependence probabilities thus runs much faster than typical dependence profiler. The compiler also partition the loop into serial and parallel (called *hoisting distance*) portions. But unlike [8], the partitioning uses static information only. Tasks are pruned based on three independent thresholds for task size, hoisting distance, and squash benefit. The latter also factors in prefetching benefit of squashed tasks.

In [22], the Mitosis compiler uses both dependence and edge-profiles for 1) generating speculative precomputation slices (*p-slice*) and 2) selecting spawning pairs. P-slice predicts live-in values for speculative tasks and contributes to the serial portion of the speculative execution. To minimize p-slice overhead while maximizing the accuracy, the compiler uses dependence- and edge-profiles to prune instructions in p-slices. To select spawning pairs, another profile analyzes the sequential execution trace to model the speculative execution time of each candidate spawning pair without considering inter-task memory conflicts. Instead, in this execution model, task squashes is mostly determined by mispredication probability of p-slices.

In [14], Johnson et al. proposed an approach wherein speculative task decomposition is modelled as a balanced min-cut problem. In this framework, edge- and dependence profiles are used to assign weights to graph edges.

In [30], Praun et al. proposed a tool for speculative task head recommendation based on binary instrumentation. The cost model for task recommendation is based on *self length* that models task sizes, *dependence length* that models conflicts, and a parallelization speedup estimate.

Alias profiling has been proposed as an assist for memory disambiguation [31,10,17]. Chen et al. [7] proposed a dependence profiler for speculative optimizations.

6 Conclusions

This paper presents a cost model for speculative task selection and a compiler-based approach for program dependence profiling. The dependence profiler, **DProf**, measures width of independence windows to quantify dynamic parallelism in the program. We also propose dependence clustering as a technique to exploit TLS parallelism on segments of the iteration space.

In addition to its use in task selection, **DProf** also reports individual dependences being profiled including dependence probability and source mapping information. This information can be fed to the compiler or the programmer to assist code transformation or algorithmic optimizations.

We present the dependence and independence window profile obtained through **DProf** for selected programs in SEQUOIA and SPEC CPU2006 benchmark suites. We observe that:

- In addition to all the loops parallelized by the compiler, only one hot loop is profiled to be parallel in the programs being studied.
- There is little variability in independence window width in the hot loops we studied. Loops are either parallel or serial with an independence window width of 1.
- For loops with tight independence window, there are often a mixture of high- and low-frequency dependences. Eliminating high-frequency dependences is key to widening the independence window.
- Dependences due to complex reduction updates is one form of high-frequency dependence that can be potentially eliminated.

As future work, we plan to provide support for dependence profiling of `while` loops and enable the profiling of complex reduction patterns.

References

1. ASC Sequoia Benchmark Codes. http://www.llnl.gov/asc/sequoia/benchmarks/.
2. D. Bruening, S. Devabhaktuni, and S. Amarasinghe. Softspec: Software-based speculative parallelism. In *Proceedings of 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, 2000.
3. F. W. Burton. Speculative computation, parallelism, and functional programming. *IEEE Trans. Computers*, 34(12):1190–1193, 1985.
4. M. Chen and K. Olukotun. TEST: a tracer for extracting speculative threads. In *Proceedings of International Conference on Code Generation and Optimization*, pages 301–312, 2003.
5. M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 434–446, 2003.
6. P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, 2003.
7. T. Chen, J. L., X. Dai, W.-C. Hsu, and P.-C. Yew. Data dependence profiling for speculative optimizations. In *Proceedings of the 13th International Conference on Compiler Construction*, pages 57–72, Barcelona, Spain, 2004.

8. Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 71–81, Washington DC, USA, 2004.
9. P. Dubey, K. O'Brien, K. O'Brien, and C. Barton. Single-program speculative multithreading (SPSM) architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1995.
10. M. Fernández and R. Espasa. Speculative alias analysis for executable code. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 222–231, Charlottesville, VA, 2002.
11. M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelsim. *SIGARCH Comput. Archit. News*, 20(2):58–67, 1992.
12. L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS'98*, 1998.
13. T. Johnson, R. Eigenmann, and T. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.
14. T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 59–70, Washington DC, USA, 2004.
15. A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using SPEC CPU2006. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.
16. V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *ICS'98*, 1998.
17. 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, 2003.
18. 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*, 2006.
19. 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*, pages 158–167, 2006.
20. B. Pottenger and R. Eigenmann. Parallelization in the presence of generalized induction and reduction variables. In *(ICS'95)*, June 1995.
21. W. Pugh. The definition of dependence distance. Technical Report CS-TR-2292, Nov. 1992.
22. C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the SIGPLAN '05 Conference on Programming Language Design and Implementation*, pages 269–279, 2005.
23. J. R. H. Halstead and T. Fujita. Masa: a multithreaded processor architecture for parallel symbolic computing. *SIGARCH Comput. Archit. News*, 16(2):443–451, 1988.
24. 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 *ICS*, June 2005.
25. G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95*, pages 414–425, S. Margherita Ligure, Italy, 1995.
26. J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA*, 2000.
27. J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 14, 1998.
28. T. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *MICRO 98*, 1998.
29. C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.
30. C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.
31. Y. Wu and Y. Lee. Accurate invalidation profiling for effective data speculation on EPIC processors. In *Proceddings of the 13th International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, Aug. 2000.
32. A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *ASPLOS-X*, 2002.
33. C. B. Zilles and G. S. Sohi. A programmable co-processor for profiling. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 241–254, Nuevo Leon, NM, Jan. 2001.