Experiences of Using a Dependence Profiler to Assist Parallelization for Multi-cores

Dibyendu Das
IBM India
dibyendu.das@in.ibm.com

Peng Wu
IBM T. J. Watson Research Center
pengwu@us.ibm.com

Abstract— **In this work we show how to use a data-dependence profiling tool called DProf, which can be utilized to assist parallelization for multi-core systems. DProf is based on an optimizing compiler and uses reference runs to emit information on runtime dependences between various memory accesses within a loop. The profiler not only marks the dependent statements and the accesses but also emits details regarding the percentage of time the dependences are encountered – the percentage being taken over the loop iteration size. Though DProf has been primarily built to capture opportunities for speculative thread-level parallelism, it has been found that the report generated by the DProf can be utilized very effectively in detecting and parallelizing complex code. To demonstrate this, we have taken two complex benchmarks – 435.gromacs and 437.leslie3d from the SPECfp CPU2006 suite and show how they can be parallelized effectively using DProf as an assist. To the best of our knowledge none of the existing parallelizing compilers can detect and parallelize all the instances reported in this work. We find that by using DProf we are able to parallelize these benchmarks very effectively for IBM P5+ and P6 multi-core systems. We have parallelized the benchmarks using OpenMP leading to speedups up to 2.6x. Also, due to the detailed reporting by DProf, we could cut down on our parallelization development effort significantly by concentrating on portions of the code that require attention. DProf can also be used to identify applications, where applying parallelization may lead to regression in performance. This allows the developers to discard applications or parts of it quickly, which may not lead to performance improvements when deployed on multi-cores. Thus, a data dependence profiler like DProf can act as an excellent assist mechanism to move applications to multi-cores in an effective way.**

Keywords- Data Dependence, Profiler, Multi-core, Parallelization, Compiler.

I. INTRODUCTION

Unassisted manual parallelization of complex code is cumbersome and error-prone requiring a huge amount of human effort. In this work we show how to use a data dependence profiling tool called **DProf** [4] which helps in cutting down this effort drastically. Thus, DProf acts as a parellelization assist. As multi-core processors proliferate and become ubiquitous, it has become increasingly important to parallelize existing applications with as less effort as possible. Programmers have to rely on automated tools for a quick turnaround time while parallelizing applications. Parallelizing compilers have existed to aid this process for a long time, but, unfortunately, they fall short of expectation when it comes to parallelizing complex codes. Examples of this are in programs which use irregular array accesses among others. Hence, tools which can quickly point to the parallelization bottlenecks of serial to parallel conversion are specifically useful. One of the major reasons why compute-intensive loops in serial programs cannot be parallelized is the existence of memory dependences that arise across iterations of the loops. However, it is not possible to analyze all such loop dependences statically at compile time though there is a small class of programs which may fall into this category. Parallelizing loops may thus require knowledge and characterization of runtime dependences to correctly parallelize them. In order to do this a prototype tool called **DProf** has been used.

DProf is a profiling tool that provides run-time information about may-dependences and map dynamic dependences to the source code. The main aim of DProf is to provide dependence related information and feed it back to the compiler to aid Thread Level Speculative (TLS) execution of loops [4,7,8]. The information is also made available to the programmer to assist in code rewriting and algorithm redesign. DProf is built on top of an optimizing compiler and utilizes its optimization passes and analyses to instrument the code required to generate the runtime may-dependence information. In this work we present case studies to illustrate how DProf can be used as an assist to parallelize some complex SPEC CPU2006 [20] benchmarks effectively – 435.gromacs and 417.leslie3d. These benchmarks contain loops which are hard to parallelize automatically. However, by utilizing the knowledge attained from DProf, we could parallelize these loops manually with very low effort. The resultant code shows significant speedup when run on IBM's Power5+/Power6 multi-cored, simultaneously multithreaded machines. We achieve up to 1.8x improvement for gromacs when run on a 4-core, 8-thread, P6 system running at 5GHz, while leslie3d shows performance improvement to the tune of 2.6x. For all these benchmarks, even production parallelizing compilers fail to detect the opportunities due to several issues including irregular access patterns and complex array access which cannot be easily privatized, among others.

The paper is organized as follows. Section II discusses about the data dependence profiler, DProf, and its design. Section III details our parallelization strategy. Section IV discusses the SPEC CPU2006 cases for gromacs and leslie3d. Section V is on related work. We end with conclusions and future work in Section VI.

978-1-4244-6534-7/10/$26.00 ©2010 IEEE

II. DPROF – DATA DEPENDENCE PROFILER

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.

A. DProf Instrumenter

The instrumenter is based on an existing optimizing XL compiler [31]. The compiler first analyzes candidate loops to decide whether they are parallel. Parallel loops are not profiled. The rest of the 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 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 loop by calls to `__profile_boundary`. The compiler does not instrument references to induction, reduction and private variables. To reduce profiling time, the compiler may not instrument references whose loop-carried dependences can be represented by dependence distances that can be computed statically. In this case, the dependence information is recorded by the instrumenter and is later combined with profiled dependences to produce a complete dependence report.

B. DProf Runtime Library

The runtime library profiles a set of properties of loops such as whether a given loop is parallel, and dependence properties such as source (the producer of a dependence), sink (the consumer of the dependence) and the frequency of dependences or dependence distances. The profiler also maintains an *independence window* – which is the set of consecutive iterations that are independent of each other. Iterations in the independence window can be executed in parallel. The profiler also 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 maintained in a stack so that nested loops can be profiled in one pass. For each `_profile_access` call, which is introduced for every memory operation in a loop, 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 the current iteration). For each `__profile_boundary` call, which is introduced for 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 an empty 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.

C. 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 those dependences by making source code changes. The source-level mapping, maintained by the instrumenter, associates each call site of `__profile_access` and `__profile_boundary` with a unique identifier. The instrumenter also generates a file, which is later used by the profiler, to map identifiers to its associated source-level information and other properties obtained from the compiler.

D. DProf Usage

DProf uses an internal compiler pragma to mark loops for profiling. The compiler, at a sufficiently high optimization level, instruments memory references in the marked loop into calls to DProf library functions so that memory references are tracked by the runtime. The compiler does not instrument reduction or induction variables. It also instruments loop boundaries and back-edges so that only cross-iteration dependences are detected.

For user-defined functions called inside the profiling loop and not defined in the same compilation unit, the file(s) containing the function definitions needs to compiled with an additional flag so that the function bodies are also instrumented by the compiler. The compiler also instruments a call to initialize and a call to exit the runtime at program entry and exit points.

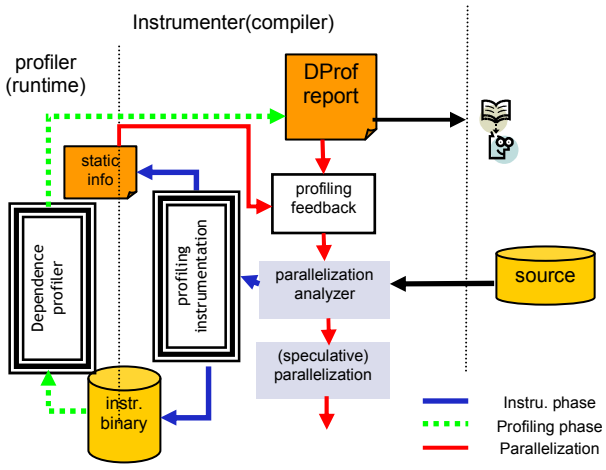

Figure 1. DProf Environment

Fig. 1 depicts the general DProf environment. Starting from the application source, it builds on the static compiler parallelization analyses to create the instrumented binary that tracks the runtime dependences. On running the instrumented binary, the report generated can be either used to as a parallelization assist (the focus of this paper) or to drive advanced parallelization techniques like thread-level speculative parallelization [8,9,10].

During the instrumentation phase, the compiler also generates `_dprof_xxx` files which contain information to map instrumented references to their respective locations. These files are used by DProf during profiling runs. During profiling phase, the profiler emits profiling information in a file called DPROF.REPORT.

III. ASSISTED PARALLELIZATION METHODOLOGY

In this section we briefly visit the generic sequence of steps that need to be applied to parallelize an application (or evaluate the parallelism potential) for multi-cores using DProf.

The first step is to run a profiler (ex: IBM's tprof [28]) on the application to capture the hotspots of the program. At present this step has to be manually carried out. The next step is to instrument the hot loops using the DProf supported pragma and run the application again. DProf generates a report in DPROF.REPORT at the end of the run listing out the possible dependences and their frequencies. If the loop is runtime-parallel DProf also highlights that information. Once the dependences and their locations are identified, it remains to be analyzed whether the dependences can be eliminated by an existing mechanism - as in irregular array accesses with reduction patterns or simpler loop-carried dependences which can be eliminated by using induction expressions. Once the possible sources of dependences are eliminated, DProf needs to be run again to understand the parallelization potential of the transformed loop. This is an iterative process which may need to be carried out several times.

If a loop is profiled to be runtime-parallel several additional steps may need to be carried out. One of them is to blindly parallelize the loop to evaluate the speedup potential on multi-core systems the application/loop is targeted at. This step may provide a good indicator of the performance potential of the application for multi-core systems. Based on the outcome of this experiment one may decide to either discard the loop/application or proceed to the next steps.

Once the performance potential looks promising, one may factor in the cost of correctness-checking code to enable parallelism of the loop under all possible circumstances. Correctness checks for the loop usually take the form of loop-versioning where the parallel version is invoked only if certain conditions are met. But there may be cases as in leslie3d, where the loop can be proved to be parallel at compile-time, with some additional work and analyses. Such analyses may frequently require the knowledge of advanced compiler algorithms and techniques.

If a loop can be proved to be parallel at compile-time, no additional correctness checking code needs to be inserted for correct parallelization leading to zero additional cost. During the parallelization process, loops which exhibit very high dependence frequency and which are not easily amenable to parallelization via known transformations can be quickly discarded allowing more time to the developers to concentrate on promising loops/applications.

IV. SPEC CPU 2006 CASE STUDIES

In this section we outline our experience with DProf-driven assisted parallelization of two benchmarks from SPEC CPU2006 [18]. The benchmarks are – 435.gromacs and 417.leslie3d. We have chosen these benchmarks as they are not parallelized effectively by production compilers resulting in lost opportunities when run on multi-core systems.

A. Parallelization of 435.gromacs

435.gromacs (Fig. 2) is one of the floating point benchmarks in the SPEC CPU2006 suite. The hottest loop in gromacs is in a file named innerf.f in the subroutine `inl1130`. The loop is doubly nested and covers around 58% of the total execution time of the benchmark. The inner loop nest contains many array references through subscript arrays of the form `faction(3*jjnr(k)…)`. Thus dependences on this loop nest cannot be detected statically. With the training set, the inner loop has an average trip count of 28 iterations, ranging from 1 to 173. The loop is profiled to be parallel by **DProf**. This implies that even though irregular accesses exist within the inner loop, at runtime these accesses do not result in loop-carried dependences.

The outer loop has an average trip count of 1250 iterations, ranging from 4 to 13891. **DProf** reports many pairs of true data dependences for this loop. The outer loop nest has many irregular accesses of the form `faction(3*iinr(n) + …)`. The frequencies of data dependences range from 0.4% to 100%. Fig. 2 shows fragment of the loop, where all high frequency dependences occur at the bottom of the outer loop. All high frequency dependences occur among statements with array element reduction patterns of the form `a[x] +=` . Of them, updates to elements of arrays `Vc`, `Vnb` and `fshift` are true reductions. The dependence percentage between the `faction` updates in the inner loops and their accesses in the outer loop has a low dependency of 0.4%. Here's a sample from the DPROF.REPORT generated for the loop at line 3932 which corresponds to the outer loop nest of `inl1130`. Some of the messages are presented in a simplified manner here for better clarity and understanding.

```
   Loop at line 3932 has :

     - Detected a true dependence with frequency of
0.40636% between "faction(j3)" (line# 4110) and "
faction(ii3)" (line# 4140) in "innerf.f"

   - Detected a true dependence with frequency of
50.1161% between "faction(ii3)" (line# 4140) and "
faction(ii3)" (line# 4140) in "innerf.f"
```

Updates to `faction` exhibit a complex pattern. Besides reduction updates to elements of `faction` in the outer loop

there are additional reads and writes to `faction` in the inner `k`-loop that is not in a straightforward reduction pattern. However, closer examination reveals that these accesses also follow the reduction pattern as pointed out in Fig. 2. This happens due to the read of `faction`(…) writing onto a scalar variable (ex: `fjx1`), which is subsequently read. After `fjx1` is read, the same index of the `faction` array (which was read to write onto `fjx1`) is updated. In such cases, one can replace the use of `fjx1` via copy propagation. This results in direct reduction of the `faction` access.

```
… DProf_pragma …
  // Reports all the runtime dependences for the loop-nest
     do n=1,nri
       …
         ii = iinr(n) + 1
         ii3 = 3*ii -2
         ...
         nj0 = jindex(n)+1
         nj1 = jindex(n+1)
         ...
         do k=nj0,nj1
            jnr = jjnr(k)+1
            j3 = 3*jnr-2
            …
            vnbtot = vnbtot + vnb12-vnb6
            vctot = vctot + vcoul
            …
            fjx1 = faction(j3)-tx11
            …  Indirect reduction
            faction(j3) = fjx1-tx31
                                          Low Dep freq 0.4%
            …
       end do
       …
       faction(ii3) = faction(ii3) + fix1          // Dep fr = 50%
       faction(ii3+1) = faction(ii3+1) + fiy1      // Dep fr = 50%
       …

       fshift(is3)    = fshift(is3) + fix1+fix2+fix3   // Dep fr = 18%
       …
       ggid = gid(n)+1
       Vc(ggid) = Vc(ggid) + vctot                 // Dep fr = 100%
       Vnb(ggid) = Vnb(ggid) + vnbtot              // Dep fr = 100%
    end do
```

Figure 2.

Irregular reduction is a well-known parallelization pattern that can possibly be parallelized using one of several techniques as listed in [1,2]. One such technique is array privatization using replicated buffers [1]. Under this, each thread computes a portion of the reduction, storing results in a privatized (locally replicated) buffer. Results from all buffers are then accumulated into the shared array. Large replicated buffers can be combined faster by running the accumulation step in parallel. While array privatization is simple to implement it can waste processing time on unused elements for sparse reductions. Sparse reductions are indicated by low dependency frequency numbers in DProf reports. Overhead is due to each thread needing to initialize all elements of the replicated buffers and merging them again into the shared global array after the parallel reduction is complete. To avoid the replication overhead of the buffers several alterative mechanisms have been proposed like Selective Privatization, Local Write and Synchronized Write [1,2,3,12,13,15,16]. However, most of these techniques are suited for situations where the dependence frequency of the irregular accesses is low. In addition, these techniques are difficult to implement in a straightforward fashion. As there are several array dependences which have a high frequency in the gromacs code, we decided to implement the simple replicated buffer algorithm in gromacs. In addition, we were looking at a faster turnaround time for the implementation of the parallelized code. Once the parallelized application demonstrates performance improvement, other techniques can be applied to improve the performance further.

1) Parallelized gromacs

```
//calculate sizes of faction array(fsize), fshift(ssize) etc.
!$omp parallel
    tnum = omp_get_thread_num() + 1
    if ( tnum .eq. 1 ) then
       numthr = omp_get_num_threads()
        allocate(redfaction(fsize,numthr))
        // allocate redVc, redVnb, redfshift
    endif
    !$omp barrier
    // In each thread sets up the private copies of arrays
    !$omp do
    do n=1,nri
    ...
       nj0 = jindex(n)+1
       nj1 = jindex(n+1)
    ...
       do k=nj0,nj1
          jnr = jjnr(k)+1
          j3 = 3*jnr-2
          …
          fjx1 = redfaction(j3,tnum)-tx11
          …
          redfaction(j3,tnum) = fjx1-tx31
          …
        end do
        redfaction(ii3,tnum) = redfaction(ii3,tnum) + fix1
        …
        // similar code for redfshift, redVc and redVnb
    end do
    !$omp end do
    //Accumulate result back, do the same for Vc,Vnb,fshift
    !$omp do
    do fi = 1,fi
       do tn = 1,tnum
          faction(fi) = faction(fi) + redfaction(fi,tn)
       end do
    end do
    !$omp end do
!$omp end parallel
```

Figure 3.

We used OpenMP [14] to parallelize the gromacs code for multi-cores (Fig. 3). OpenMP is a well-known shared-memory parallel programming paradigm that is supported by modern compilers. However, OpenMP 3.0 does not support array reduction yet. To circumvent this problem we had to

hand-code the array reduction as shown in Fig. 3. **DProf** reports the inner loop of gromacs to contain no dependences. But due to the low average trip count of this loop, applying OpenMP parallel pragmas on this loop results in a slowdown in performance even when array reduction need not be implemented. As opposed to the parallel inner loop, **DProf** reports several dependences in the outer loop – all involving irregular array accesses. There are also dependences between the irregular accesses of the inner loop with those of the outer loop for the `faction` array. We parallelized the irregular array access pattern of the gromacs loop with privatized copies of these arrays – one for each thread. The temporary array created for `faction` (which is one dimensional) is called `redfaction` – which is a two dimensional array, the second dimension being used by the threads. Reduction arrays are also created for `Vc`, `Vnb` and `fshift` in a similar fashion. Additional code has been added to setup these arrays as well as merge them back at the end of the reduction process. Extra code is added to compute the sizes of the original arrays that are involved in irregular accesses as they are not available easily in the subroutine. Knowing the sizes of the arrays is important for allocating the private copies of the arrays – one for each thread.

While the method of replicated buffers/arrays has been known for some time to deal with irregular reductions, computing the sizes of the array temporaries may remain an issue. In many programs, as in gromacs, the arrays involved in irregular accesses are passed as parameters to the functions. While Fortran90 provides an in-built function called `size()` to compute the sizes of arrays (even when passed as parameters), the arrays must not be of assumed-size type for `size` to work correctly. When assumed-size arrays are present in Fortran90 or arrays are passed as parameters in C/C++, one way to compute the sizes of the parameter arrays and use them for temporary array allocation, is to create a new copy of the loop/s in which irregular accesses exist. Using the access indices of the arrays, removing all the other irrelevant code, and executing the newly created size-generating loop before the original computation loop, one can determine the sizes of the arrays which can be used for subsequent allocation.

2) Results

The results in Table I show how the speedup of gromacs varies when run with different configurations. The benchmark has been compiled at the highest optimization level of `-O5` of the IBM XL compiler Ver. 10/12 [31], along with a bunch of other flags. As all the cores are 2-way simultaneously multithreaded in P5+ and P6, the cores can run threads double its number. We notice a substantial speedup for gromacs when the OpenMP version of the benchmark is run on a 4-core configuration using 8 simultaneous threads for both P5+ as well as P6. For P6, the speedup varies from 1.3x for a 2-thread run to 1.7x for a 4-thread and 1.83x for an 8-thread run. We use a maximum of 4 cores. Similarly, the speedup numbers are 1.25, 1.45 and 1.65 for P5+. The results demonstrate that the replicated buffer method provides good speedup even when code is added for allocation, merging as well as determining array sizes.

TABLE I. MULTI-CORE PERFORMANCE RESULT FOR GROMACS

Processor	Serial	2-Thread	4-Thread	8-Thread
P6 5Ghz	556	426	329	305
P5+ 2.1 Ghz	701	558	483	424

B. Parallelization of 417.leslie3d

417.leslie3d is one of the SPEC CPU2006 floating point benchmarks. Profile data of a serial run shows that the top three subroutines of leslie3d, called `fluxi`(25%), `fluxk`(20%) and `fluxj`(15%) consume around 60% of the total execution time. All the three routines are coded in a very similar manner. Though there are no indirect accesses, the loops in these routines contain a significant number of conditional statements. Most of the array accesses happen inside these conditional statements. This adds to the complexity of the static analysis of this loop. Manual inspection of the code is daunting as the loop span is big and consists of many array accesses.

To evaluate whether the outer loops of the `flux*` routines are parallelizable we have utilized the DProf tool. Each of the outer loops of these routines is marked by a pragma for the DProf to test and figure out the dependences (Fig. 4 shows the `fluxi` code and the associated pragma). On running the tool DProf reports that there is **no runtime dependence** that exists in these loops at the outermost levels of the `flux*` subroutines. This clearly indicates that the loops are **parallel at runtime**. The simplified DPROF.REPORT looks like this:

```
Loop at line 515 :
  - The loop is parallel
Loop at line 888 :
  -   The loop is parallel
Loop at line 1268 :
  -   The loop is parallel
```

We can parallelize the `flux*` outer loops by using OpenMP pragmas. However it is not guaranteed that for a different input dataset these loops will still be parallel. Hence, blindly parallelizing these loops is incorrect. In order to circumvent this problem we pursued the policy of inspecting the loop of the `fluxi` routine to check whether it is compile-time parallel. As the subroutines `flux*` are very similar to `fluxi` the effort of checking for one can be used effectively to check the others.

In `fluxi` a number of arrays are read and written in conditional statements. The crux of the problem is to figure out whether all these arrays are privatizable. This would imply that they are written before they are read in every iteration of the outer `K`-loop. Though there are compiler-based algorithms to detect whether an array is privatizable [21,22,23], such algorithms are (usually) either not fully implemented due to their cost complexities or are implemented such that only simple cases can be tackled. In

leslie3d array privatization is non-trivial due to the presence of conditional code.

1) Static Analysis of fluxi

Fig. 5 highlights the loop structure of `fluxi`. We explore the parallelization possibility of the K-loop for all datasets, which is reported to be runtime-parallel by DProf. Our approach is demonstrated using one of the arrays called DUDX which is both read and written in the loop nest. The array accesses in `fluxi` can be privatized as none of them have an upward-exposed use and all reads of arrays are covered by the writes placed earlier. Arrays accessed within conditions make it necessary that analysis of guarded array regions (GARs) [22] is applied extensively in order to detect all these privatizable arrays. To do this manually is a non-trivial task indeed, especially when the loop is big as in `fluxi`, but we carried out this task to find that the loop is indeed static time parallelizable. There is only one array called DU which is written and read in each iteration of the outer loop. DU has no loop-carried dependences and hence can be shared by all the parallel iterations without correctness issues. The scalars used in the loops can also be privatized as they do not have upward-exposed uses.

```
... DProf-pragma ...
// DPROF states ZERO DEPENDENCE in the outer loop
   DO K = K1,K2
     DO J = J1,J2
       ...
       IF(NSCHEME .EQ. 2) THEN
         DO I = I1,I2
           DUDX(I) = ...
           ...
         END DO

       ELSE
         DO I = I1,I2
           DUDX(I) = ...
           ...
         END DO
       END IF
       ...
       DO I = I1,I2
         ...   = DUDX(I) ...
       END DO
       ...
       IF(ISGSK .EQ. 1) THEN
         DO I = I1,I2
           ...
           ... = DUDX(I)
         END DO
         ...
       END IF
       ...
     END DO
   END DO
```

WRITEs of DUDX

READs of DUDX

Figure 4.

Fig. 5 shows the GARs of the DUDX array. `DUDX[i1:i2]` notation in the figure signifies that the DUDX accesses the linear region `DUDX[i1]` to `DUDX[i2]` using a stride of 1. The GAR notation has been simplified in Fig. 5 by using an annotated CFG with AR(array region)s, instead of GARs. DUDX is written in both the true and false paths of the condition `if (NSCHEME .EQ. 2)`. It is read later in a couple of instances, once unconditionally and once under `if (ISGSK .EQ. 1)`. It is clear that both the reads of `DUDX[i1:i2]` are dominated by an earlier instance of write to the same array region (AR) of `DUDX[i1:i2]`. This implies that no section of array DUDX is read-exposed which makes privatization of DUDX possible. Similarly DUDY, DUDZ, etc. are also privatizable. Thus, both the DProf and the static analyses suggest that the outer loops of the `flux*` calls can be parallelized directly using OpenMP once the requisite arrays are privatized.

2) Parallelized leslie3d

`fluxi/j/k` can be parallelized in a straightforward manner (Fig. 6) since we have been able to figure out that all the concerned arrays can be privatized safely. Once the privatization list is set up, and the OpenMP pragmas are inserted, leslie3d can be run in parallel with significant speedup. There is no extra code transformation required in the body of the loop nest. The parallelized loops do not incur any additional overhead over their serial counterparts.

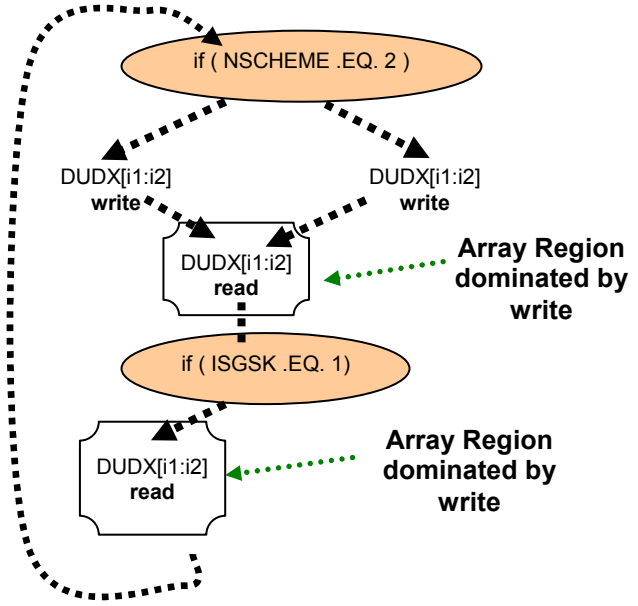

Figure 5. Array regions for a snippet of leslie3d

```
SUBROUTINE FLUXI()

!$OMP PARALLEL DO
!$OMP& PRIVATE(…)
!$OMP& PRIVATE(…)
…
      DO K = K1,K2
         DO J = J1,J2
            … BODY OF FLUXI UNCHANGED …
         END DO
      END DO
!$OMP END PARALLEL DO

      RETURN
      END
```

Figure 6.

3) Results

The results for leslie3d that were noted when executed on a 4-core, 8-way simultaneously multithreaded P5+ and P6 machine are shown in Table. II. The benchmark was compiled at the highest optimization level of the XL compiler which is `-O5` along with a bunch of other flags. We can observe that leslie3d speeds up quite well when the number of threads and cores are increased. However the speedup becomes flatter when the parallelized version is deployed using 8 threads instead of 4. For P6 the speedup for 4-core, 8-threads is around 2.4x while a similar configuration in P5+ provides a speedup of around around 2.6x.

TABLE II. MULTI-CORE PERFORMANCE RESULTS FOR LESLIE3D

Machine	Serial	2-Thread	4-Thread	8-Thread
P6 5Ghz	290	199	137	119
P5+ 2.1 Ghz	604	346	242	236

V. RELATED WORK

Parallelizing applications via external aids like parallelizing or analyses tools is not new. Several works in the literature like [5,6] have incorporated semi-automatic mechanisms for general as well as domain-specific computations. However, these tools are based on static analyses and knowledge rather than runtime profiles of dependences in loops. DProf adopts a source instrumentation approach for better reporting of dependences. However, tools have been developed that can instrument binaries [26,29] for dependence tracking based on PIN [27] and Valgrind [30].

There is a rich body of literature on automatic parallelization. [20] studies the effectiveness of such parallelizing compilers for a set of benchmarks and concludes privatization and reduction to be two major blockers for application parallelization. Work in the area of automatic parallelization had introduced paradigms like the inspector-executor model [17]. This kind of paradigm is primarily used to handle irregular array accesses. However, such inspector-executor methods do not provide any insight into the frequency of the dependences or the absence of such dependences for sections of the code with irregular access patterns. Runtime parallelization via LPD/LRPD (Lazy Parallel DoAll) tests is described in [21]. In this work the authors parallelize the loops requiring privatization and reduction and provide tests to check for the safety and correctness of their transformations. If at runtime, dependence is found to exist, the compiler switches over to the serial mode of execution. However, this work cannot parallelize loops with function calls. In [19,25] the authors propose advanced techniques for combining compile- and run- time information in order to parallelize more effectively. Their work improves upon the LPD (Lazy Privatizing Doall) test, which checks for runtime data dependences in non-parallelized loops and decide whether such dependences can be eliminated by privatization techniques. For implementing this they use predicated array data flow analysis.

Use of profile-based cost model for task decomposition and selection for speculative multithreading have been in use for some time. Probabilistic points-to analysis is reported in [7]. In [8], the compiler uses dependence profile for task selection and for partitioning 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 partition of loop bodies into a serial and parallel portion. In [10] the POSH compiler uses profiling for task selection. In [24], for the Mitosis compiler, the dependence as well as edge-profiles are used to generate speculative pre-computation slices and selecting spawning pairs.

VI. CONCLUSIONS AND FUTURE WORK

Automatic parallelizing tools usually falls short when it comes to parallelizing complex serial code. In spite of a rich body of research that has gone into automatic parallelizing technologies it is frequently found that such tools are unable to extract parallel regions from serial code. With the advent of multi-core this may result in a significant loss of performance. We try to bridge this gap in this work by proposing the use of a profiling tool called DProf which can be effectively utilized to parallelize the code using the information garnered from the tool.

DProf was originally designed for automatic detection of thread level speculation, but the tool has been used here for detecting parallelization opportunities with encouraging results. DProf flags dependent computations and reports their frequencies. This has implications beyond what some of the research automatic parallelizers have been able to achieve. Dependence frequencies can be used to drive the parallelization strategy as well as measure the parallelization potential. DProf, thus, helps us target the promising loops, discarding those which are either not parallelizable or do not have performance potential even when parallelized. Hence, using DProf for hand-parallelizing code increases programmer productivity and shows good speedup.

Tools like DProf induce overheads in the runtime of an application due to instrumentation. Hence, for loops with high iteration counts, the output report generated by DProf may take a substantial amount of time to complete. In order to alleviate the problem, we have provided special environment variables to control the report generation. Such environment variables update the report at regular intervals so that the user can examine it at intervals. We have observed in several cases that most of the dependences can be extracted during the initial iterations. In such cases we can stop or continue with the report generation process and move on to the next steps. However, for cases like leslie3d where runtime dependences are absent, we need to allow the entire report generation phase to be completed before further analysis.

Parts of the assisted parallelization methodology can be further automated, leading to higher productivity. This involves automatically detecting hot loops and tracking them via DProf. The application can be launched subsequently resulting in the report file being generated. However, we feel that analyses of the report file and the final parallelization

will require manual intervention, as these stages require expertise that is difficult to build into an automated tool.

ACKNOWLEDGMENT

The authors would like to thank the anonymous referees for useful suggestions to improve the focus, quality and presentation of the paper.

REFERENCES

[1] H. Han and C.W. Tseng, "A Comparison of Parallelization Techniques for Irregular Reductions," Proc. of 15th International Parallel and Distributed Processing Symposium, IPDPS, 2001.

[2] E. Gutierrez, O. Plata, and E.L. Zapata, "A Compiler Method for the Parallel Execution of Irregular Reductions in Scalable Shared Memory Multiprocessors," Proc. of the 14th International Conf. on Supercomputing, 2000, pp. 78-87.

[3] M. Erez, J.H. Ahn, J. Gummaraju, M. Rosenblum, and W.J. Daly, "Executing Irregular Scientific Applications on Stream Architectures," Proc. of the 21st International Conf. on Supercomputing, 2007, pp. 93-104.

[4] W. Peng, A. Kejariwal, and C. Cascaval, "Compiler-driven Dependence Profiling to Guide Program Parallelization," Proc. of the Languages and Compilers for Parallel Computing (LCPC), 2008, pp. 232-248.

[5] M. Ishihara, H. Honda, T. Yuba, and M. Sato, "Interactive Parallelization Assistance Tool for OpenMP:iPat/OMP," In Fifth European Workshop on OpenMP, EWOMP, 2003, pp. 93-100.

[6] S. Mitra, S. C. Kothari, J. Cho, and A. Krishnaswamy, "Paragent: A Domain-Specific Semi-automatic parallelization tool," Proc. of High Performance Computing, HiPC, 2000, pp. 141-148.

[7] 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," Proc. of the 9th ACM SIGPLAN Symposium on Principle and Practice of Parallel Programming (PPoPP), 2003, pp. 25-36.

[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," Proc. of the Conference on Programming Language Design and Implementation (PLDI), 2004, pp. 71-81.

[9] 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," Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2007, pp. 215-225.

[10] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "POSH: A TLS compiler that exploits program structure," Proc. of the 11th ACM SIGPLAN Symposium on Principle and Practice of Parallel Programming(PPoPP), 2006, pp. 158-167.

[11] B. Pottenger and R. Eigenmann, "Parallelization in the presence of generalized induction and reduction variables," Proc. of the 9th International Conf. on Supercomputing, 1995.

[12] H. Yu, and L. Rauchwerger, "Adaptive reduction parallelization techniques," Proc. of the 14th International Conference on Supercomputing, May 2000, pp. 66-77.

[13] H. Han and C.W Tseng, "Improving compiler and run-time support for irregular reductions," Proc. of the 11th Workshop on Languages and Workshops for Parallel Computing, Aug 1998, pp. 181-196.

[14] OpenMP. http://www.openmp.org

[15] Y. Lin and D. Padua, "On the automatic parallelization of sparse and irregular Fortran programs," Proc. of the 4th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers, May 1998, pp. 231-246.

[16] M. Guo, "Automatic parallelization and optimization for irregular scientific applications," Proc. of the 18th International Parallel and Distributed Processing Symposium, IPDPS, 2005, pp. 228-237.

[17] Y.S.Hwang, B. Moon, S.D. Sharma, R. Ponnusamy, R. Das, and J. H. Saltz, "Runtime and language support for compiling adaptive irregular programs on distributed-memory machines," Software Practice and Experience, 25(6), 1995 pp. 597-621.

[18] SPEC CPU2006 benchmarks. http://www.spec.org

[19] S. Moon, B. So, M.W. Hall, and B. Murphy, "A Case for Combining Compile-Time and Run-Time Parallelization," Proc. of the 4th Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers, May 1998, pp. 91-106.

[20] W. Blume and R. Eigenmann, "Performance Analysis of parallelizing compilers on the Perfect Benchmark Programs," IEEE Transactions on Parallel and Distributed Systems, 3(6), Nov 1999, pp. 643-656.

[21] L. Rauchwerger and D. Padua, "The LRPD Test: Speculative run-time parallelization of loops with privatization and reduction parallelization," Proc. of the Conference on Programming Language Design and Implementation (PLDI), June 1995, pp. 218-232.

[22] Z. Li, J. Gu, and G. Lee, "Interprocedural Analysis Based on Guarded Array Regions," Proc. of Compiler Optimizations for Scalable Parallel Systems: Langauges, Compilation Techniques and Run-Time Systems, 2001, pp. 221-246.

[23] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, "Array Data-Flow Analysis and its Use in Array Privatization," Proc. of the 20th ACM Symposium on Principle of Programming Languages (POPL), 1993, pp. 2-15.

[24] C. G. Quinones, C. Madriles, J. Sanchez, P. Marcuello, A. Gonzalez, and D. M. Tullsen, "Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices," Proc. of the Conference on Programming Language Design and Implementation (PLDI), 2005, pp. 269-279.

[25] S. Moon and M. W. Hall, "Evaluation of predicated array data-flow analysis for automatic parallelization," In Proceedings of the 7th ACM SIGPLAN Symposium on Principle and Practice of Parallel Programming(PPoPP), 1999, pp. 84-95.

[26] C. V. Praun, R. Bordawekar, and C. Cascaval, "Modeling optimistic concurrency using quantitative dependence analysis," Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2008, pp. 185-196.

[27] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," Proc. of the Conference on Programming Language Design and Implementation (PLDI), 2005, pp. 190 – 200.

[28] tprof.http://publib.boulder.ibm.com/infocenter/aix/v6r1/index.jsp?topic=/com.ibm.aix.cmds/doc/aixcmds5/tprof.htm.

[29] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A Transparent Dependence Distance Profiling Infrastructure," In Proceedings of of Code Generation and Optimization (CGO), 2009, pp. 47 – 58.

[30] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," Proc. of the Conference on Programming Language Design and Implementation (PLDI), 2007, pp. 89-100.

[31] IBM XL Compilers and White Papers, http://www.ibm.com/software/awdtools/ccompiler.