Research Journal of Applied Sciences 11 (10): 900-909, 2016
ISSN: 1815-932X
© Medwell Journals, 2016

# Enhancing the Performance of Decoupled Software Pipeline Through Backward Slicing

[1]Esraa Alwan, [2]John Fitch and [2]Julian Padget
[1]Department of Computer Science, University of Babylon, Hillah, Iraq
[2]Department of Compute Science, Bath University, Bath, England

**Abstract:** The rapidly increasing number of cores available in multicore processors does not necessarily lead directly to a commensurate increase in performance: programs written in conventional languages such as C, need careful restructuring, preferably automatically, before the benefits can be observed in improved run-times. Even then, much depends upon the intrinsic capacity of the original program for concurrent execution. The subject of this study is the performance gains from the combined effect of the complementary techniques of the Decoupled Software Pipeline (DSWP) and (backward) slicing. DSWP extracts thread level parallelism from the body of a loop by breaking it into stages which are then executed pipeline style: in effect cutting across the control chain. Slicing, on the other hand, cuts the program along the control chain, teasing out finer threads that depend on different variables (or locations). parts that depend on different variables. The main contribution of this paper is to demonstrate that the application of DSWP, followed by slicing offers notable improvements over DSWP alone, especially when there is a loop-carried dependence that prevents the application of the simpler DOALL optimization. Experimental results show an improvement of a factor of ~1.6 for DSWP +slicing over DSWP alone and a factor of ~2.4 for DSWP + slicing over the original sequential code.

**Key words:** Decoupled software pipeline, slicing, multicore, thread-level parallelism, automatic restructuring

## INTRODUCTION

Multicore systems have become a dominant feature in computer architecture. Chips with 4, 8 and 16 cores are available now and higher core counts are promised. Unfortunately increasing the number of cores does not offer a direct path to better performance especially for single-threaded legacy applications. But using software techniques to parallelize the sequential application can raise the level of gain from multicore systems (Bridges, 2008). Parallel programming is not an easy job for the user, who has to deal with many issues such as dependencies, synchronization, load balancing and race conditions. For this reason the role of automatically parallelizing compilers and techniques for the extraction of several threads from single-threaded programs without programmer intervention is becoming more important and may help to deliver better utilization of modern hardware (Liao *et al*., 2010). Two traditional transformations, whose application typically delivers substantial gains on scientific and numerical codes are DOALL and DOACROSS. DOALL assigns eachiteration of the loop to a thread Fig. 1 which then may all execute in parallel because, there are no cross-dependencies between the iterations. Clearly, DOALL performance scales linearly with the number of available threads. The DOACROSS Fig. 2 technique is very similar to DOALL, in that each iteration isassigned to a thread, however there are cross-iteration dataand control dependencies. Thus to ensure the correct results,data dependencies have to be respected, typically through synchronization so that a later iteration receives the correctvalue from an earlier one as illustrated in Fig. 1 (Vachharajani, 2008).

DOALL and DOACROSS techniques depend onidentifying loops that have a regular pattern (Vachharajani *et al*., 2007) but manyapplications have irregular control flow and complex memoryaccess patterns, making their parallelization very challenging.The Decoupled Software Pipeline (DSWP) has been shown tobe an effective technique for the parallelization of applicationswith such characteristics. This transformation partitions theloop body into a set of stages, ensuring that critical pathdependencies are kept local to a stage as shown in Fig. 3. Each stage becomes a thread and data is passed between threads using inter-core communication (Huang *et al*., 2010). The success of DSWP depends on being able to extract the relatively fine grain parallelism that is present in many applications. Another technique which offers potential gains in parallelizing general purpose applications is slicing. Program slicing transforms

**Corresponding Author:** Esraa Alwan, Department of Computer Science, University of Babylon, Hillah, Iraq

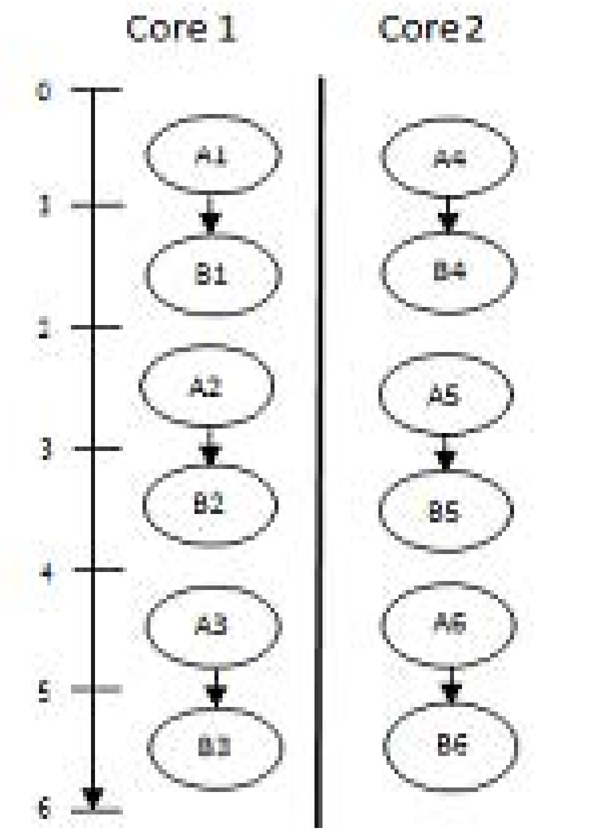

Fig. 1: DOALL technique

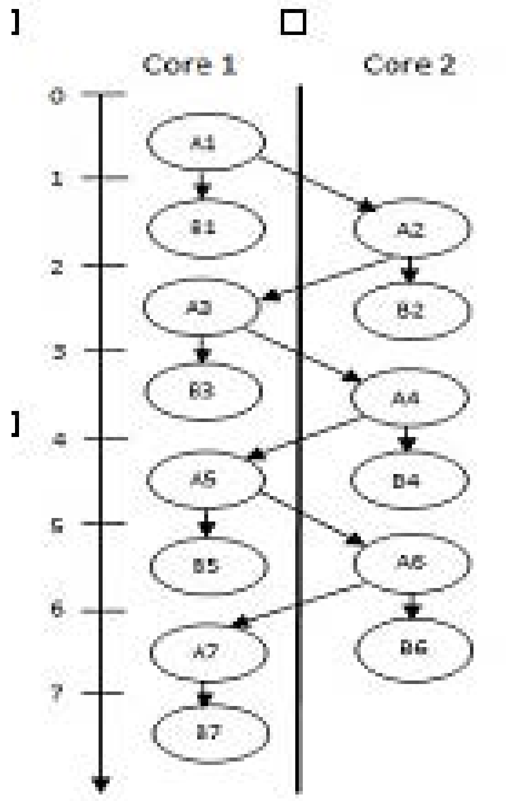

Fig. 2: DOACROSS technique

large programs into several smaller ones that execute independently, each consisting of only statements relevant to the computation of certain, so-called, (program) points. The slicing technique is appropriate for parallel execution on a multi-core processor because it has the ability to decomposethe application into independent slices that are executable in parallel (Wang *et al*., 2008; Weiser, 1983, 1984.). Many techniques have been proposed to extract thread level parallelism from the program. Wang *et al*. (2008) introduce a dynamic framework to parallelize a single threaded binary program using speculativeslicing (Rong *et al*., 2007) propose a method to construct a software pipeline from an arbitrarily deep loop nest, whereas the traditional one is applied to the innermost loop or from the innermost to outer loops. This approach is called the Single dimensional Software Pipeline (SSP). The (SSP) name came from the conversion of a multi-dimensional Data Dependency Graph (DDG) to 1-D DDG. Rangan *et al*. (2004) introduced a new technique to utilize a decoupled software pipeline for optimizing the performance of Recursive Data Structures (RDS) (e.g., linked lists, trees and graphs). Raman *et al*. (2008) introduce a Parallel Stage Decoupled Software Pipeline (PS-DSWP). This technique is positioned between the decoupled software pipeline and DOALL. The reason for this combination is that the slowest stage of DSWP bounds the speed of DSWP as we have noted so this work exploits the ability to execute some stages of DSWP using DOALL. Huang *et al*. (2010) show that DSWP can improve performance if it works with other techniques. This usage called DSWP+, divides the loop body into stages. These stages are open to parallelization with another techniques like DOALL, LOCALWRITE and Spec DOALL.

This rerches explores the possibility of performance benefits arising from a secondary transformation of DSWP stages byslicing. Our observation is that individual DSWP stages can be parallelized by slicing, leading to an improvement in performance of the longest duration DSWP stages. In particular, this approach can be applicable in cases where DOALL is not. The proposed method is implemented using the Low Level Virtual Machine (LLVM) compiler framework (Lattner and Adve, 2004). LLVM muses a combination of a low

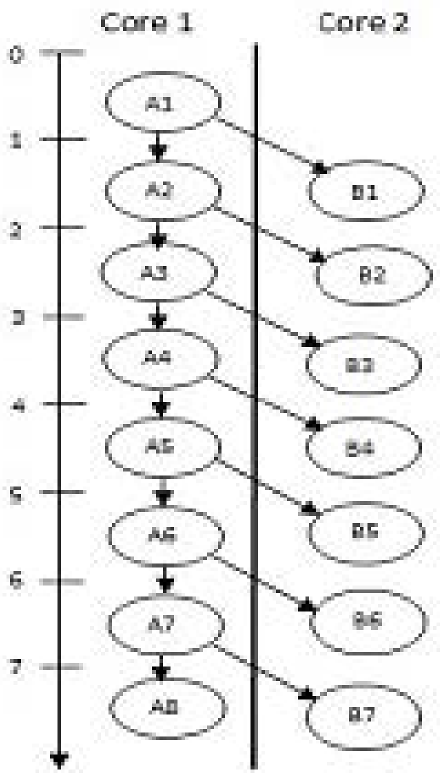

Fig. 3: DSWP technique adopted

level virtual instruction set combined with high level type information. An important partof the LLVM design is its Intermediate Representation (IR). This has been carefully designed to allow for many traditional analyses and optimizations to be applied to LLVM code and many of which are provided as part of the LLVM framework.

## MATERIALS AND METHODS

**DSWP+slicing transformation:** The performance of a DSWP-transformed program is limited by the slowest stage. Thus, any gains must come fromimproving the performance of that stage. The main feature of the proposed method is the application of backward slicingto the longest stage emerging from the DSWP transformation. This is particularly effective when that stage includes a function call. To illustrate the method, consider the example in. DSWP partitions the loop body into the parts labelled L and X, then we slice X to extract S1 and S2. Consequently, instead of giving the whole of stage X to one thread, it can be distributedacross n threads, depending on the number of slices extracted with in this case, one core running L (the first stage) and twomore running S1 and S2 (the slices from the second stage). However while there are potential gains from splitting theloop body into several concurrent threads, there is still the cost of synchronization and communication between threads to take into account. To minimize these overheads we use lock-free buffers (Giacomoni *et al*., 2008 Zhao and Hahnenberg, 2008). As a result, producer and consumer can access the queue concurrently, via the enqueue and dequeue operations. This makes it possible for the producer and consumer tooperate independently as long as there is at least one data element in the queue.

**Sliced loop body with recurrence dependency:**

```
X: Work(cur) {
S1: Slice1(cur);
S2: Slice2(cur);
}
List *cur = head;
L: for (; cur != NULL;
cur = cur->next)
        X: Work(cur);
```

**Source program:**

```
1...                                1Calc(int M     // Calc is the function to
be sliced
2double ss=0;                       2double da_in
3 int i;                                        3double* da_out) {
4 double a[0]=0;                    4int j;
5while( node != Null) {             5b[0]=0;
6Calc(node->data,a[i],&a[i+1]);     6 for(j=0;j<M;j++) {
  7 i++;                                        7 m+=da_in+seq(j);
  8 node=node->next;                            8(*da_out)+=da_in+cos(m);
// first slice variable da_out
 9}                         9  b[j]=b[j]+xx(m);// second slice variable b[j]
  10                                            10  }
```

**Implementation of DSWP+slicing:** We build on earlier work by Zhao and Hahnenberg (2008) who implement DSWP in LLVM. We have extended that code withbackward slicing and a decision procedure to determine when it is worth applying the transformation.

**Slice 1 on da_out:**

```
Slice_1(M,da_in){
 int j;
 for(j=0;j<M;j++) {
    m+=da_in+seq(j);
    (*da_out) += da_in+cos(m);
  }
}
```

**Slice 2 on b(j):**

```
Slice_2(M,da_in,da_out){
 int j;
 b[0]=0;
 for(j=0;j<M;j++) {
 m+=da_in+seq(j);
  b[j]=b[j]+xx(m);
 }}
```

The transformation procedure is based on the algorithm for DSWP proposed by Ottoni *et al*. (2005) It takes as input L, the loop to be optimized and modifies it as a side-effect. The details are as follows:

**Find candidate loop:** This step looks for the most profitable loop to apply DSWP + Slicing. We collect staticin formation about the program and then use a heuristic to estimate the number of cycles necessary to execute allinstructions in every loop in the program. The loop with the largest estimated cycle count and containing a functioncall is chosen.

**Build the Program Dependency Graph (PDG):** The subject is the loop to be parallelized. Figure 4 shows thatthe solid lines (red) denote data dependency and dashed lines (black) control dependency.

**Build strongly connected component (SCC) DAG:** In order to keep all the instructions that contribute toa dependency local to a thread, a Strongly Connected Component(SCC) is built, followed by the DAG for the SCCs. Consider the code in. The loop (lines 5–9) traverses a linked list and calls the procedure Calc. Figure 5 shows the DAGscc of the PDG of the programon the left had side of in the procedure Calc, there are loop-carried dependencies that make DOALLinapplicable. After DSWP partitioning, we extract two slices where function seq is side-effect-free.

**Assign SCCs to threads:** The previous step may result in more SCCs than available threads. In this case, we merge SCCs until there are as many as there are threads. In our example, we have a function call in the loop body. Weassign the SCCs that represent the outer loop body to the first thread and the n extracted slices to n threads.

**The compute all slice algorithm adopted from Jehad Al Dallal:**

Input: A PDG, set of empty list associated,one for each node identifier(variable in theslicing list).
Output: Slice for each node identifier(variable).

```
Algorithm:
Make all PDG nodes as not visited
ComputeASlice(exit node)
Compute A Slice ( node n){
    If node is not visited
Mark node n as visited
Add the instructions of n to the setassociated with node n
For each node m( instruction)in whichnode n depends ComputeASlice(m)
Add the content of the setassociated with node m to the setassociated with node n
}
```

**Extract slice:** In this part, a small slicing program is designed that has the ability to extract slices for the limited range of the case studies. The algorithm illustrated in is used to compute an intra-proceduralstatic slice. The N static slices from the function body areextracted as follows:In the first step, the PDG is built for the function bodyby drawing up the dependency table that has both controland data dependency (similar to the one above used todetermine thread assignment). Secondly, the entry blockfor the function body is examined so as to identify the variables to be sliced and then the names of these are collected being put on a slicing list. The compute A sliceis called to extract a slice for every listed variable. Then an attempt is made to isolate the control statement parts such as loop or if statement, into another table called the control table. After collecting the control part instructions, these are added to the extracted slice, if one of the slice instructions is contained in this control parts. For each filtered variable in the slicing identifiers list, first an empty list is associated with it and subsequently, all the PDG table entries are scanned to find which one matches the slicing identifier. If one is found, then allthe instructions that have data or control dependency are

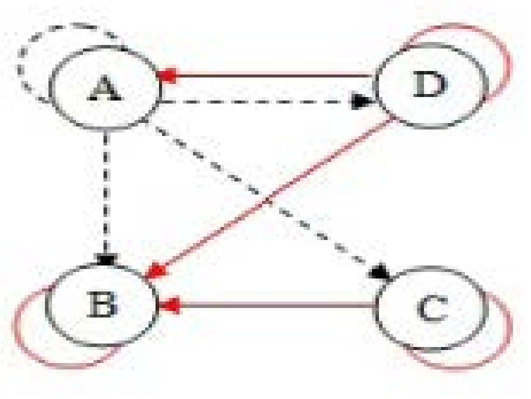

Fig. 4: Program dependency

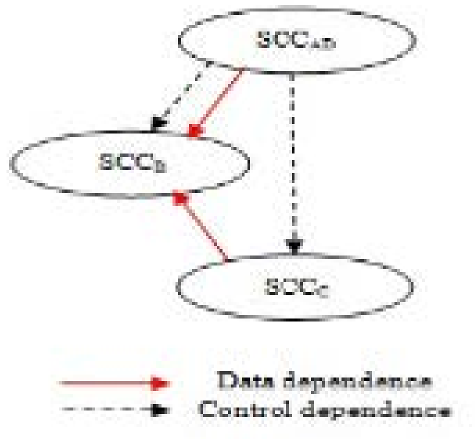

Fig. 5: DAG of SCCs

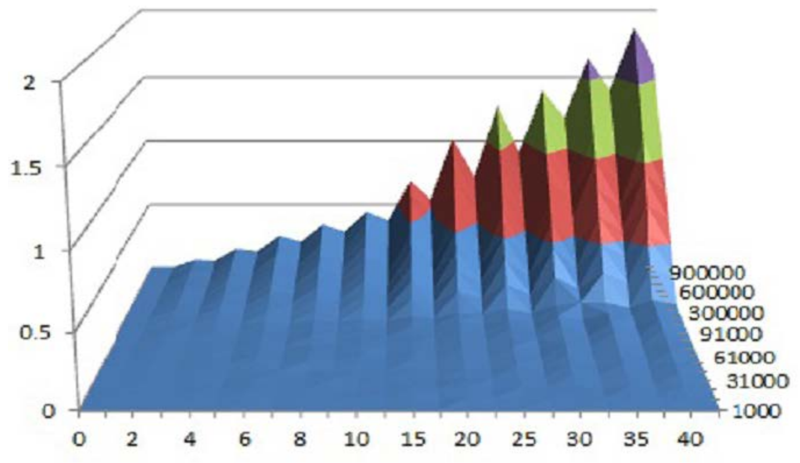

Fig. 6: Effect of N and M on DSWP

added to the associated list. This procedure is repeated toall the instructions in the associated list and their operands and is not stopped until all the instructions and their operands are contained in this list or all the variables that represent the loop induction variables have been reached. In the case of two slices will be retracted for two variables.

**Insert synchronization:** To ensure correct results, the dependence between threads must be respected and for pipeline parallelism to be effective, the overhead on core-to-core communication must be as low as possible. Hence, we use the fast forward circular lock-free queue algorithm (Giacomoni *et al*., 2008).

## RESULTS AND DISCUSSION

**Experimental results:** This study discusses the results obtained from the application of the automatic implementation of the proposed method. Several programs have been used as case studies. Some are artificial and others are taken from (Tao, 1997). The discussion examines two issues: the effect of lock-free buffers on the performance of DSWP and theresults from the application of DSWP+slicing, demonstrating how this method can improve the performance of long stage DSWP with different program patterns.

**Communication overhead:** This study examines the impact of communication costs on the performance of DSWP. It is important for us to be able to quantify this cost because it is a critical factor in the decision procedure for whether to carry out the DSWP + slicing transformation. We are also aware this cost will be platform dependent which is why we provide details of ourparticular platform. In a production deployment, this aspect would have to be measured as part of a calibration process.Consider the program in. We wish to execute this it by applying DSWP to the loop that takes the most executiontime of the program. Initially, we partition the program into two parts, give eachto a thread and execute the threads as a pipeline. The first thread handles lines 5–14 and the second, lines 15–24. Two parameters play a vital role in determining the benefit (or otherwise) of DSWP, namely M and N. M affects the amount ofwork inside each thread by controlling the number of iterations in the inner loops while N, in effect, determines the volumeof data transfer between threads, by controlling the number of outer loop iterations. Figure 6 shows how changing the value of N (1-40) and M (1000-1000000) affects the execution time of the DSWP version compared to the sequential program. From N = 6 and M = 51000 the performance of DSWP becomes better than the sequential one. Furthermore the effect of the buffer size on the performance of DSWP is examined, for which the same program as in was employed. However this time the value of N was fixed to 1,000 and M to 10,000 and the only parameter that was.

**Sequential version of program to evaluate DSWP overheads:**

```
1 main()
2 int N,M
3 .....
4 rows=N;
```

```
for(i1=1; i1 < rows; i1++) {
for(z=1;z<M;z++) {
sum = 0;
for(a=1; a<10; a++)
sum = sum + image[i1] *mask_1[a];
        if(sum > max) sum = max;
if(sum < 0) sum =10;
        if(sum < out_image[i1])
out_image[i1] = sum;
}
for(z1=1;z1<M;z1++) {
sum1 = 0;
        for(a1=1; a1<10; a1++) {
        sum1 = sum1 + image[i1] * mask_2[a1];
if(sum1 > max) sum1 = max;
        if(sum1 < 0) sum1 = 10;
        if(sum1 > out_image[i1])
        out_image[i1] = sum1;
}
        }
```

changed was the buffer size. That is was varied between 10 and 1000, with the execution time of the program being only slightly changed during the during the execution (2- 5 ms) which was because it was assumed that this was the amountof time needed to create the link list. As a result, it can be concluded that the effect of buffer size on DSWP is trivial.

**Combining DSWP + slicing:** We now examine the effect of combining DSWP + slicing by applying slicing to the long stage coming out of the DSWP transformation. The

sample programs that we study here all exhibit an imbalance between the two stages of the DSWP, i.e., the number of instructions in the outer loop is less than the number of instructions in the function body. The addition of slicing permits some degree of equilibration. Two of the sample programs are artificial (linked list 2c and linked list 3. c) while the remaining three (fftc, pro 2.4c and test 0697c) are genuine. For each of the case studies, we extract two slices from the function body, so that the maximum number of threads ingeneral were four depending on whether the extracted slice returns value to the original loop or not. The data transferred between DSWP stages corresponds to the arguments of a function which in our case studies are between one and four arguments. LLVM-gcc (the LLVM C front end, derived from gcc) and the LLVM compiler framework have been used to automate our method. In addition, manually transformed programs have been compiled using gcc in order to be able to compare manual and automatic results. (Table 1) summarises the technical details of the evaluation platform.Our automatic method uses two passes: 1) The first pass carries out static analysis of all the loops in a program. For each loop it adds up the static execution time for each instruction in the loop body and

Table 1: Platform details

| Processor | Intel(R) Core(TM) i7 CPU |
|---|---|
| Processor speed | 2.93 GHz |
| Processor configuration | 1 CPU, 4 Core, 2 threads per Core |
| L1d Cache size | 32 k |
| L1i Cache size | 32 k |
| L2 Cache size | 256 k |
| L3 Cache size | 8192 k |
| RAM | 4.GB |
| Operating system | SUSE |
| Compiler | GCC and LLVM |

Table 2: Execution times for program test0697.c

| Gcc-swp | Gcc-dswp-slice (Man.) | Gcc-seq | Llvm-dswp-slice (Auto.) | Llvm-seq | Iter. |
|---|---|---|---|---|---|
| 0.304 | 0.272 | 0.370 | 0.119 | 0.135 | 2 |
| 0.483 | 0.420 | 0.628 | 0.173 | 0.215 | 5 |
| 0.667 | 0.602 | 0.875 | 0.179 | 0.287 | 7 |
| 0.866 | 0.775 | 1.140 | 0.260 | 0.360 | 9 |
| 1.046 | 0.954 | 1.387 | 0.263 | 0.410 | 11 |
| 1.242 | 1.115 | 1.651 | 366 | 0.523 | 13 |

also accumulates the execution time for the function bodies and stores these results in a Table 2) The second pass chooses a loop to transform and constructthe software pipeline. This uses the data collected in the previous pass to identify the highest cost loop that also contain a function call. Next we look at the sample programs in more detail and at the results of the transformation process.

**The fft.c:** An implementation of the fast Fourier transform (Tao, 1997). The test program is a generalization of the program to make it work with N functions. We give the outer loop to the first thread and the fft function to the second thread. From the graph in it is clear how the unbalanced long stage DSWP can affect DSWP performance, where it only improvesslightly on the sequential program. We extract two slices from the loop body: The first is the computation of the real part and the second the imaginary part again shows loop speed up for DSWP+slicing in both manual and automatic forms.

**The pro-2.4.c:** This program (Tao, 1997) computes the derivative of N functions. F1 is the first derivative, F2 the second, D1 is the error in F1 and D2 the error in F2. Similar to the previous program we extract two slices from function body after giving the it to the second stage DSWP. As with the previous program we add some adaptations to the program and we generalize it to make it work for N functions. We set NMAX = 100000 and vary M from M = 5 to M = 30. Figure 7 shows the execution time for sequential, DSWP, DSWP+slicing (manual) and DSWP+slicing (automatic). Figure 8 and 9 shows loop speed up for Pro-2.4 using DSWP+slicing.

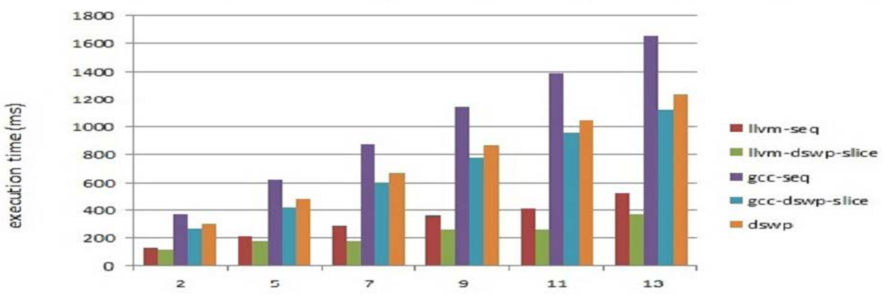

Fig. 7: Loop speed up with three threads for test 0697.c program

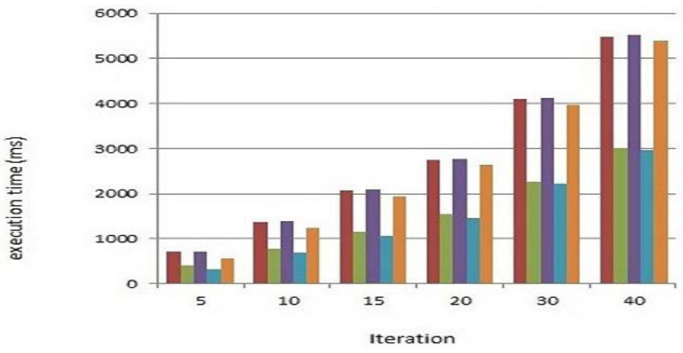

Fig. 8: Loop speed up with three threads for fft.c program

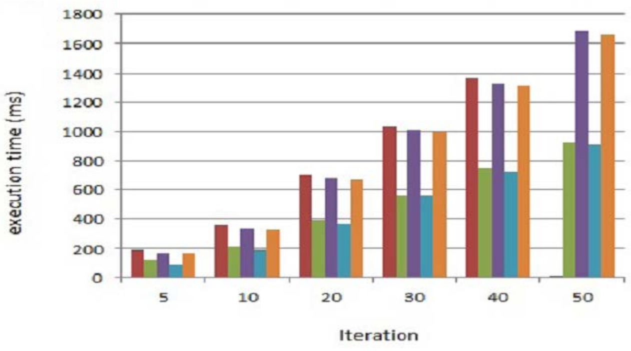

Fig. 9: Loop speed up with three threads for linked list 2.c program

**Test 0697.c:** This program computes the spherical harmonics function which is used in many physical problems ranging from the computation of atomic electron configuration to the representation of the gravitational and magnetic fields of planetary bodies. It has two function calls inside the loop body. The first, called the spherical-harmonic-value, gives the initial value to the second function argument with this functionbeing called the spherical-harmonic. The loop was divided into two parts, depending on the instruction latency execution time. The second function call which represents the spherical-harmonic was allocated to the second thread whilst the rest ofthe loop body containing the first function call was assignedto the first thread. Subsequently, two slices, c[] and s[] wereextracted from the second function call by applying slicingtechnique on this part alone. With high values (40000) of L and M the execution time of this combination was betterthan for the sequential program. The number of threads wasthree with two communication buffers and the number oftransferred function arguments was four. The results obtained by automatic and manual implementation for the sequential and DSWP slicing versions, show that the former methodgives ~1.4 speed up compared with the sequential programin the LLVM environment(see columns 2 and 3 in the table in14). Moreover, columns 4 and 5 under the GCC environmentshows that the speed up becomes ~1.5 after applying theslicing technique, while that for DSWP alone is only ≈ 1.3 (Table 3).

Table 3: Execution times for program fft.c

| Gcc-dswp | Gcc-dswp-slice (Man.) | Gcc-seq | Llvm-dswp-slice (Auto.) | Llvm-seq | Iter. |
|---|---|---|---|---|---|
| 0.558 | 0.310 | 0.700 | 0.406 | 0.702 | 5 |
| 1.244 | 0.690 | 1.391 | 0.780 | 1.375 | 10 |
| 1.934 | 1.069 | 2.078 | 1.155 | 2.058 | 15 |
| 2.625 | 1.453 | 2.770 | 1.532 | 2.750 | 20 |
| 3.972 | 2.214 | 4.130 | 2.272 | 4.106 | 30 |
| 5.390 | 2.954 | 5.530 | 3.013 | 5.474 | 40 |

Table 4: Execution times for linkedlist2.c program

| Gcc-dswp | Gcc-dswp-slice (Man.) | Gcc-seq | Llvm-dswp-slice (Auto.) | Llvm-seq | Iter. |
|---|---|---|---|---|---|
| 0.167 | 0.95 | 0.170 | 0.120 | 0.191 | 5 |
| 0.332 | 0.190 | 0.335 | 0.215 | 0.359 | 10 |
| 0.664 | 0.369 | 0.680 | 0.380 | 0.707 | 20 |
| 0.998 | 0.556 | 1.010 | 0.553 | 1.035 | 30 |
| 1.320 | 0.730 | 1.330 | 0.733 | 1.372 | 40 |
| 1.660 | 0.910 | 1.684 | 0.915 | 1.707 | 50 |

**linkedlist{2,3}.c:** The fourth program is anotherartificial program in two variants. The common feature is thetraversal of a linked list of linked lists (in contrast to the use ofarrays as in the other examples). The key difference betweenthe variants is that the function called from the loop body does not return a value in the first (linkedlist2.c) anddoes in the second (linkedlist3.c). This allows us todemonstrate the cost of adding a buffer to the program. Two parameters affect the workload, namely the length of the firstlevel list and the length of the second level list.In these test the length of the second level list is fixed at1000 elements, while the length of the first ranges between 10 and 70, giving rise to the results shown in Table 4 and theexecution times show in Fig. 9. The results for the secondversion of the program appear in Table 5 and 6. By comparing Fig. 10 and 11, we can see how

Table 5: Execution times for linkedlist3.c program

| Gcc-dswp | Gcc-dswp-slice (Man.) | Gcc-seq | Llvm-dswp-slice (Auto.) | Llvm-seq | Iter. |
|---|---|---|---|---|---|
| 0.167 | 0.95 | 0.170 | 0.122 | 0.160 | 5 |
| 0.332 | 0.190 | 0.335 | 0.214 | 0.344 | 10 |
| 0.664 | 0.369 | 0.680 | 0.387 | 0.694 | 20 |
| 0.998 | 0.556 | 1.010 | 0.557 | 1.058 | 30 |
| 1.320 | 0.730 | 1.330 | 0.927 | 1.726 | 50 |
| 1.660 | 0.910 | 1.684 | 1.286 | 2.440 | 70 |

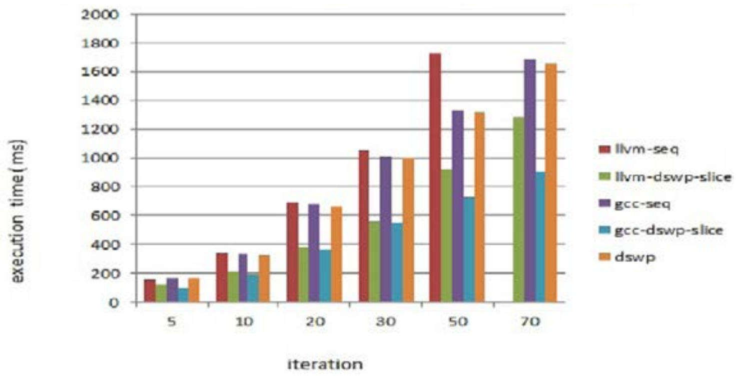

Fig. 10: Loop speed up with three threads for linked list 3.c program

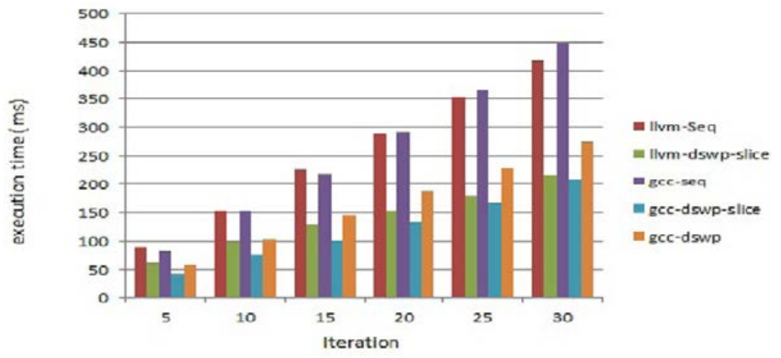

Fig. 11: Loop speed up with three threads for Pro 2.4 program

Table 6: Execution times for Pro 2.4 program

| Gcc-swp | Gcc-dswp-slice (Man.) | Gcc-seq | Llvm-dswp-slice (Auto.) | Llvm-seq | Iter. |
|---|---|---|---|---|---|
| 0.058 | 0.042 | 0.83 | 0.062 | 0.088 | 5 |
| 0.103 | 0.077 | 0.153 | 0.100 | 0.153 | 10 |
| 0.145 | 0.101 | 0.220 | 0.130 | 0.227 | 15 |
| 0.188 | 0.134 | 0.292 | 0.153 | 0.290 | 20 |
| 0.230 | 0.168 | 0.365 | 0.180 | 0.353 | 25 |
| 0.275 | 0.210 | 0.450 | 0.217 | 0.419 | 30 |

adding an additionalbuffer to communicate the return value from the one of these slices affects the execution time. This cost appears to have amarginally higher impact on the program using DSWP alone, making it slower than the original sequential program.

## CONCLUSION

This study introduces the idea of DSWP applied in conjunctionwith slicing, by splitting up loops into new loops that are amenable to slicing techniques. An evaluation of this technique on five program codes with a range of dependence patterns leads to considerable performance gains on a core-i7 870 machine with 4-core/ 8-threads. The results are obtained from an automatic implementation that shows the proposedmethod can give a factor of up to 2.4 speed up compared with the original sequential code. The contribution of this study is a proof of the concept that DSWP + slicing can offer useful benefits and, moreover that such transformation can be done automatically and under the control of an heuristic procedure that assesses the potential gains to be achieved. Consequently, there is much work toN be done in respect of improving the collection of data and the decision procedure as well as the integration of the technique into a non-experimental compiler environment. More specifically we aim to increase the potential parallelism that can be extracted from the long stage DSWP. One of major issues with backward slice is the longest critical path (slice) creates a limit on parallelism. Insights from (Wang *et al*., 2008) suggest we can increase parallelism (number of extracted slices) by combining loop unrolling with backward slice in the presence of loop carried dependencies.

## ACKNOWLEDGEMENTS

We gratefully acknowledge the Ministry of Higher Education and Scientific Research (MoHESR) in Iraq for theirfinancial support during the period of this research.

## REFERENCES

Bridges, M.J., 2008. The velocity compiler: Extracting efficient multicore execution from legacy sequential codes. Ph.D. Thesis, Princeton University, USA.

Giacomoni, J., T. Moseley and M. Vachharajani, 2008. Fast forward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 20-23, 2008, ACM, Salt Lake, Utah, ISBN:978-1-59593-795-7, pp: 43-52.

Huang, J., A. Raman, T.B. Jablin, Y. Zhang and T.H. Hung et al., 2010. Decoupled software pipelining creates parallelization opportunities. Proceedings of the 8th Annual IEEE-ACM International Symposium on Code Generation and Optimization, April 24-28, 2010, ACM, Toronto, Ontario, ISBN:978-1-60558-635-9, pp: 121-130.

Lattner, C. and V. Adve, 2004. LLVM: A compilation framework for lifelong program analysis and transformation. Proceedings of the International Symposium on Code Generation and Optimization, March 20-24, Chicago, IL, USA., pp: 75-86.

Liao, C., D.J. Quinlan, J.J. Willcock and T. Panas, 2010. Semantic-aware automatic parallelization of modern applications using high-level abstractions. Int. J. Parallel Program., 38: 361-378.

Ottoni, G., R. Rangan, A. Stoler and D.I. August, 2005. Automatic thread extraction with decoupled software pipelining. Proceedings of the 38th Annual IEEE-ACM International Symposium on Micro architecture, November 12-16, 2005, IEEE Computer Society, Washington, USA., ISBN:0-7695-2440-0, pp: 105-118.

Raman, E., G. Ottoni, A. Raman, M.J. Bridges and D.I. August, 2008. Parallel-stage decoupled software pipelining. Proceedings of the 6th Annual IEEE-ACM International Symposium on Code Generation and Optimization, April 06-09, 2008, ACM, Boston, Massachusetts,ISBN:978-1-59593-978-4,pp: 114-123.

Rangan, R., N. Vachharajani, M. Vachharajani and D.I. August, 2004. Decoupled software pipelining with the synchronization array. Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, September 29-October 3, 2004, Antibes Juan-les-Pins, France, pp: 177-188.

Rong, H., Z. Tang, R. Govindarajan, A. Douillet and G.R. Gao, 2007. Single-dimension software pipelining for multidimensional loops. ACM. Trans. Archit. Code Optim. (TACO.), 4: 163-174.

Tao, P., 1997. An Introduction to Computational Physics. Cambridge University Press, UK.

Vachharajani, N., R. Rangan, E. Raman, M.J. Bridges, G. Ottoni and D.I. August, 2007. Speculative decoupled software pipelining. Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, September 15-19, 2007, Brasov, Romania, pp: 49-59.

Vachharajani, N.A., 2008. Intelligent speculation for pipelined multithreading. Ph.D. Thesis, Princeton University, Princeton, New Jersey, USA.

Wang, C., Y. Wu, E. Borin, S. Hu and W. Liu et al., 2008. New slicing algorithms for parallelizing single-threaded programs. PESPMA., 2008: 20-27.

Weiser, M., 1983. Reconstructing sequential behavior from parallel behavior projections. Inf. Process. Lett., 17: 129-135.

Weiser, M., 1984. Program slicing. IEEE Trans. Software Eng., SE-10: 352-357.

Zhao, F. and M. Hahnenberg, 2008. Decoupled software pipelining in LLVM. MCS Thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania. https://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/fuyaoz/courses/15745/ report.pdf.