Decoupled Software Pipelining Creates Parallelization Opportunities

Jialu Huang Arun Raman Thomas B. Jablin Yun Zhang Tzu-Han Hung David I. August

Departments of Computer Science and Electrical Engineering
Princeton University
{jialuh,rarun,tjablin,yunzhang,thhung,august}@princeton.edu

Abstract

Decoupled Software Pipelining (DSWP) is one approach to automatically extract threads from loops. It partitions loops into long-running threads that communicate in a pipelined manner via inter-core queues. This work recognizes that DSWP can also be an *enabling transformation* for other loop parallelization techniques. This use of DSWP, called DSWP+, splits a loop into new loops with dependence patterns amenable to parallelization using techniques that were originally either inapplicable or poorly-performing. By parallelizing each stage of the DSWP+ pipeline using (potentially) different techniques, not only is the benefit of DSWP increased, but the applicability *and* performance of other parallelization techniques are enhanced. This paper evaluates DSWP+ as an enabling framework for other transformations by applying it in conjunction with DOALL, LOCALWRITE, and SpecDOALL to individual stages of the pipeline. This paper demonstrates significant performance gains on a commodity 8-core multicore machine running a variety of codes transformed with DSWP+.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization; C.1.4 [*Processor Architectures*]: Parallel Architectures

General Terms Performance

Keywords multicore, parallelization, DSWP, speculation, enabling transformation

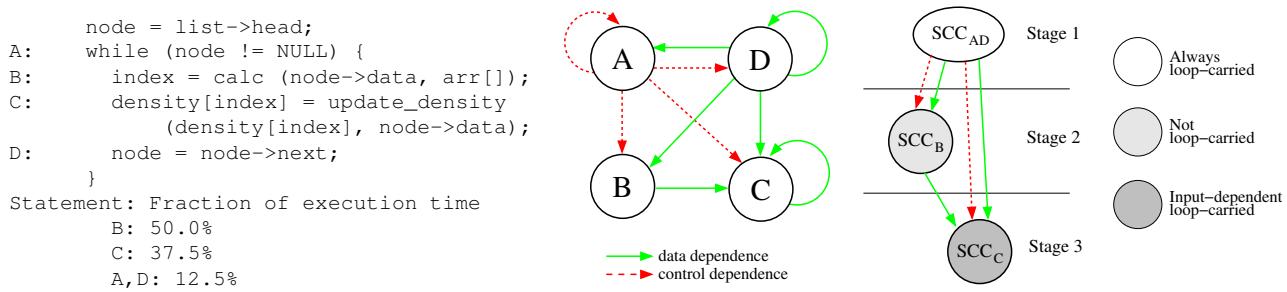
1. Introduction

Chip manufacturers have shifted to multicore processors to harness the raw transistor count that Moore's Law continues to provide. Unfortunately, putting multiple cores on a chip does not directly translate into performance. The trend toward simpler cores and the increasing disparity between the growth rates of core count and off-chip bandwidth means performance may even degrade. Not only do sequential codes suffer, multi-threaded programs may too deteriorate due to smaller caches per core and lower single-threaded performance. Consequently, producing well-formulated, scalable parallel code for multicore is the biggest challenge facing the industry.

Concurrency and non-determinism pose fundamental problems for programmers looking to (re-)write parallel code, as is evident in active research in automatic tools for the identification of deadlocks, livelocks, and race conditions. Conceptually, tools to automatically identify and extract parallelism appear more attractive. Until recently, success in automatic loop parallelization was restricted to the scientific domain. If all the iterations of a loop can be executed concurrently, then the DOALL transformation can be applied to extract the parallelism [1]. If there are *inter-iteration* (or *loop-carried*) dependences in the loop, then techniques such as DOACROSS and DOPIPE may be applicable [4, 6]. The applicability of these techniques is generally limited to codes with regular, array-based memory access patterns, and regular control flow, hence their success in scientific codes.

Unfortunately, the vast majority of general-purpose applications have irregular control flow and complex memory access patterns using pointers. To parallelize such codes, *Decoupled Software Pipelining* (DSWP) was proposed [15]. An automatic parallelization technique, DSWP splits the loop body into several stages distributed across multiple threads, and executes them in a pipeline. To generate the pipeline organization, DSWP segregates dependence recurrences into separate stages. Compared to DOALL, DSWP enjoys wider applicability. Compared to DOACROSS, DSWP enjoys better performance characteristics. By keeping critical path dependences thread-local, DSWP is tolerant of long communication latencies between threads. By decoupling the execution of the threads using inter-core queues, DSWP is also tolerant of variable latency within each thread.

CGO'10, April 24–28, 2010, Toronto, Ontario, Canada.



(a) Example code (b) Build PDG (c) Construct and partition DAG_{SCC}

Figure 1. Steps in the transformation of a sequential program (a) into a pipeline parallel program using DSWP. In (c), each SCC has a different loop-carried dependence pattern.

This paper looks beyond the performance benefits of DSWP, and explores DSWP as an *enabling transformation* for various loop parallelization techniques. At its core, DSWP identifies dependence cycles within a loop and isolates said cycles in stages of a pipeline. As a result, the dependence patterns of each stage tend to be much simpler. A key insight of this work is that other parallelization techniques may often be applied or applied more effectively to these simplified stages but cannot be applied to the original code.

This paper introduces DSWP+, an *enabling transformation* for other loop parallelization techniques, and evaluates DSWP+ in conjunction with several techniques such as DOALL, LOCALWRITE, and SpecDOALL [1, 10, 17]. Unlike DSWP, which tries to maximize performance by balancing the stages of the pipeline, DSWP+ tries to assign work to stages so that they can be further parallelized by other techniques. After partitioning, DSWP+ allocates enough threads to the parallelizable stages to create a balanced pipeline. DSWP+ yields more performance than DSWP and other parallelization techniques when applied separately. This paper shows how DSWP+ code partitioning can mitigate problems inherent to some parallelization techniques, such as redundant computation in LOCALWRITE.

Through manual parallelization of a variety of application loops, this paper demonstrates significant performance gains on *existing* multicore hardware. By virtue of its use of fully automatic techniques, the proposed methodology can be made fully automatic in future work.

Section 2 provides a detailed background of the DSWP transformation and then illustrates how DSWP+ uses it to create other parallelization opportunities. Section 3 discusses how various parallelization techniques are chosen and integrated with DSWP+. Section 4 provides results of experiments and analyzes factors that affect performance. Section 5 discusses related work, while Section 6 concludes the paper.

2. Motivation

The C code example in Figure 1(a) illustrates that in many applications' loops, the dependence pattern renders conventional parallelization techniques either inapplicable or poorly-performing. In the example, a program traverses a linked-list `list` (statements A and D); the data in each node of the list indexes into an array `arr` in order to calculate the `index` (statement B), and updates the `density` array at `index` (statement C). By applying DSWP+ to this loop, the very techniques that were originally inappropriate can become applicable and well-performing. For illustrative purposes, assume that statement B accounts for 50%, statement C accounts for 37.5%, and statements A and D account for 12.5% of the total execution time of the loop.

Figure 1(b) shows the Program Dependence Graph (PDG) of the loop [7]. There are many inter-iteration dependences. The inter-iteration self-dependence in statement C arises because the `index` may not always be different on different iterations. It depends on the data inside each node, the contents of array `arr`, and the `calc` function.

Inter-iteration dependences prevent executing each iteration of the loop independently (as in DOALL) without costly synchronization. DOACROSS schedules entire loop iterations on different threads, while synchronizing dependences to maintain correctness. Although DOACROSS is applicable, the large amount of synchronization will severely limit the benefits of parallelization. DOACROSS communicates dependence recurrences (cycles in the PDG) from core to core, putting inter-core communication latency on the critical path [22].

Thread-Level Speculation (TLS) may speculatively remove dependences that prevent DOALL or DOACROSS from being applicable [14, 17, 20, 24]. In the example shown, speculating all loop-carried dependences to make DOALL applicable could cause excessive misspeculation since the linked-list traversal dependence manifests on every loop iteration. A better option is to synchronize these dependences as in DOACROSS and speculate only the input-dependent self-dependence in statement C. However, the problem of inter-core communication latency on the critical path persists.

In contrast to DOALL and DOACROSS which partition the iteration space across threads, DSWP partitions the loop

body into stages of a pipeline, with each stage executing within a thread. DSWP's pipeline organization keeps dependence recurrences local to a thread, avoiding communication latency on the critical path. DSWP operates in four steps [15].

- First, DSWP builds the PDG of the loop [1]. The program dependence graph contains all data (both register and memory) and control dependences, both intra- and inter-iteration. Figure 1(b) shows the PDG for the loop in Figure 1(a).
- Second, DSWP finds the loop recurrences, instructions participating in a dependence cycle. DSWP groups dependence cycles into strongly-connected components (SCCs) that form an acyclic graph. Figure 1(c) shows the DAG_{SCC}. Each node in the DAG_{SCC} is a single SCC. These SCCs form the minimum scheduling units so that there are no cross-thread cyclic dependences.
- Third, DSWP allocates each SCC to a thread. Since the slowest stage in the pipeline limits overall performance, DSWP tries to balance the load on each stage when assigning SCCs to threads. This partitioning operation is shown using horizontal lines in Figure 1(c).
- Finally, DSWP inserts *produce* and *consume* operations to transmit data values in case of data dependences and branch conditions for control dependences.

While DSWP often provides noticeable performance improvement, it is limited by the number and size of the SCCs. DSWP extracts at most one thread per SCC. Further, the pipelined organization means that the throughput is limited by the slowest stage (Speedup = $1/T_{\text{slowest stage}}$). In the example loop, DSWP performance is limited to 2.0x (= $1/T_{\text{Stage 2}}$). However, as Figure 1(c) shows, the loop-carried dependence pattern in each stage is different. While the loop-carried dependences in Stage 1 *always* manifest, there are no loop-carried dependences in Stage 2 and the manifestation of the loop-carried dependence in Stage 3 depends on the input. By choosing suitable parallelization strategies for Stages 2 and 3, the execution times of SCC_B and SCC_C can be reduced to that of SCC_AD. The resulting pipeline is potentially balanced.

While there may be dependences inside `calc` that are carried across an inner loop, note that B does not contain any loop-carried dependences across the loop being parallelized. Consequently, DOALL may be applied to the second stage which can now be replicated across multiple threads. Stage 1 distributes the values it produces in a round-robin fashion across iterations to the multiple Stage 2 threads. By extracting a new loop with no loop-carried dependences out of the original, DSWP+ makes DOALL applicable.

After the DSWP+DOALL transformation, the obtainable speedup is limited to 2.67x (= $1/T_{\text{Stage 3}}$). The self-dependence in statement C inhibits parallelization. Since the `index` depends on the contents of array `arr`, `node->data`, and the `calc` function, it is not guaranteed to be different on different iterations; in other words, the manifestation of the dependence from statement C in one iteration to another is input-dependent. Consequently, the compiler must conservatively insert a loop-carried dependence in the PDG preventing DOALL from being applicable. However, a technique known as LOCALWRITE can be applied [10]. LOCALWRITE partitions the array into blocks and assigns ownership of each block to a different thread. Each thread only updates the array elements belonging to its block. Figure 2 shows the code for each thread: updates to the array are guarded by ownership checks. While LOCALWRITE is applicable to the original loop in Figure 1(a), it will perform poorly due to the problem of redundant computation (see Section 4.2.2 for details).

```
Thread i:
  if (i == owner (density[index])) {
    density[index] = update_density
      (density[index], node->data);
  }
```

Figure 2. Memory ownership checking in LOCALWRITE

An alternative parallelization of Stage 3 is possible. Speculation can be used to remove the loop-carried dependence from statement C to itself. Hardware or software TLS memory systems can be used to detect whether the dependence manifests; if it does, then the violating iteration(s) will be rolled back and re-executed [14, 17, 20, 23]. Blindly speculating all loop-carried dependences in the original loop will cause excessive misspeculation because other loop-carried dependences (self-dependence in statement D) manifest on every iteration. The overhead of misspeculation recovery negates any benefits of parallelization. By separating out the problematic dependences, DSWP+ creates a loop with dependences that can be speculated with a much higher degree of confidence, thus getting good performance.

In summary, the example shows that DSWP+ can transform a loop with an unworkable dependence pattern into multiple loops each of which can be parallelized in (potentially) different ways using the very techniques that failed on the original loop. The following section describes the integration of specific loop parallelization techniques with DSWP+.

3. Code Transformation With DSWP+

The code example in Figure 1(a) is used to illustrate the transformations done with DSWP+. For the sake of clarity, transformations are shown at the source-code level. Although these transformations were performed manually, we emphasize that each of these techniques has been automated in parallelizing compilers. Details of the algorithms can be found in [3, 10, 15, 16]. Automation of the ensemble technique involves bringing them together into one framework and changing the optimization goal of DSWP.

```
node = list->head;
while (node != NULL) {
  produce (Q[1,2], node);
  produce (Q[1,3], node);
  node = node->next;
}
```

(a) Stage 1: SCC_{AD} in Figure 1(c)

```
while (TRUE) {
  node = consume (Q[1,2]);
  if (!node) break;
  index = calc (node->data, arr[]);
  produce (Q[2,3], index);
}
```

(b) Stage 2: SCC_{B} in Figure 1(c)

```
while (TRUE) {
  node = consume (Q[1,3]);
  if (!node) break;
  index = consume (Q[2,3]);
  density[index] = update_density
    (density[index], node->data);
}
```

(c) Stage 3: SCC_{C} in Figure 1(c)

Figure 3. DSWP applied to the loop in Figure 1(a) extracts three stages which communicate using *produce* and *consume* primitives.

3.1 DSWP+

Figure 1 shows the steps in the DSWP transformation. After building the PDG of the loop, dependence recurrences are identified, and the loop is partitioned into stages at the granularity of SCCs. DSWP+ follows the first two steps of the DSWP algorithm. In the third step, DSWP optimizes for pipeline balance. In contrast, DSWP+ tries to put as much work as possible in stages which can be subsequently parallelized by other techniques. Finally, the other techniques are performed on each stage, and threads are allocated to each stage as to achieve pipeline balance. Figure 3 shows the code for each stage. Data values and control conditions are communicated across threads using *produce* and *consume* primitives. These primitives are implemented in software using cache-aware, concurrent lock-free queues [8].

3.2 DSWP+*PAR_OPTI*

The DSWP+ transformation works in conjunction with other loop parallelization techniques. For a given parallelization technique *PAR_OPTI*, let DSWP+*PAR_OPTI* be the pipeline parallelization strategy that chooses the largest stage(s) to which *PAR_OPTI* is applicable. In this paper, we investigate DSWP+DOALL, DSWP+LOCALWRITE, and DSWP+SpecDOALL.

3.2.1 DSWP+DOALL

From Figure 1(c), the second stage is free of loop-carried dependences. DOALL may be applied to this stage. Figure 4 shows the replication of the second stage. All the threads executing the second stage share the same code. However, a logical queue between two stages is implemented as several physical queues between threads, and so the code for each stage must be parameterized to support replication (hence the thread id *i* in Figure 4). Stage 1 produces values to the Stage 2 threads in a round-robin fashion. For example, if there are two Stage 2 threads, then the values on even iterations are sent from Stage 1 to (Stage 2, Thread 1), while values on odd iterations are sent from Stage 1 to (Stage 2, Thread 2). Stage 3 consumes from (Stage 2, Thread 1) and (Stage 2, Thread 2) on alternate iterations in a corresponding fashion.

3.2.2 DSWP+LOCALWRITE

LOCALWRITE partitions an array into blocks and assigns ownership of each block to a one of several threads. Each

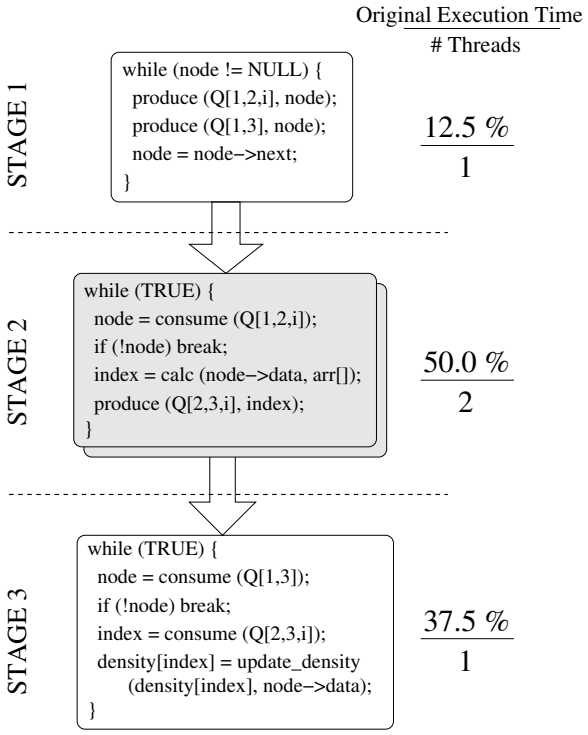


Figure 4. DSWP+ with DOALL applied to the second stage. With 2 threads assigned to the second stage, speedup can be improved to 2.67x (= $1/T_{\text{Stage 3}}$). Assigning more threads is useless because Stage 3 has become the bottleneck.

thread may only update array elements that it owns. Under this ownership discipline, array updates do not need to be synchronized. Figure 5 shows the application of LOCALWRITE to Stage 3 of the pipeline. Stage 1 must produce the value of `node` to all the Stage 3 threads. (In Figure 5, the number of Stage 3 threads is `n_lw_threads`.) As in DSWP+DOALL, the thread id (`j` in Figure 5) is used to select the physical queue between threads. Updates to global state (the `density` array) are guarded by the ownership check highlighted in Stage 3 in Figure 5. Typically, each thread is given ownership of a contiguous block of array elements. Figure 6 shows an example of ownership functions. The scalability of LOCALWRITE is limited by the distribution of the values of `index` across the array blocks; if the

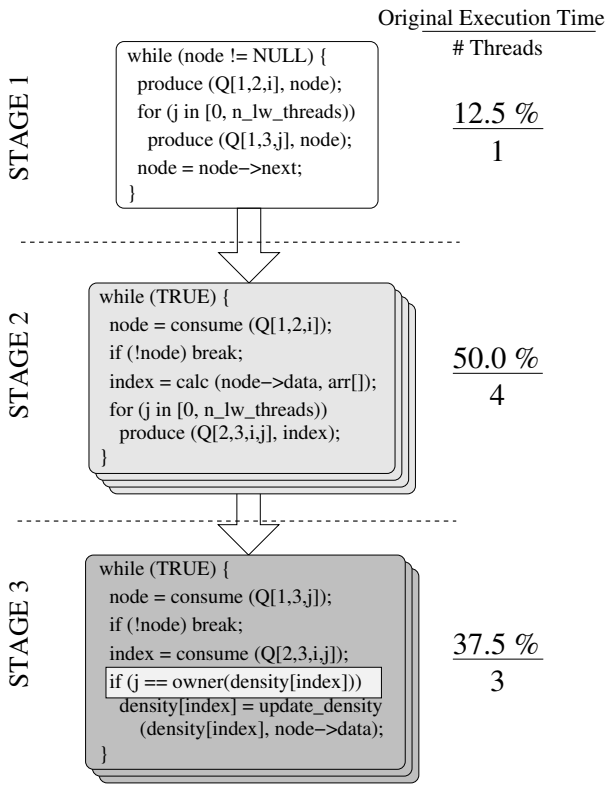


Figure 5. DSWP+ with DOALL applied to the second stage and LOCALWRITE applied to the third stage. The ownership check for LOCALWRITE in Stage 3 is highlighted. With 4 threads assigned to Stage 2 and 3 threads to Stage 3, speedup can be improved to 8x (= $1/T_{\text{Stage 1}}$).

distribution is uniform, performance scales linearly with additional threads.

Referring to Figure 5, Stage 2 produces the `index` values to Stage 3. The ownership check is inserted in the consumer (Stage 3). This means that the producer (Stage 2) must produce the value of `index` on every iteration to *all* the consumer threads. Each consumer thread must perform the ownership check for every `index` value. Figure 7 shows that this redundancy can be eliminated by moving the ownership check to the producer. On each iteration, the producer uses the ownership check to determine the consumer thread that is the owner of the array element `density[index]`, and communicates the value only to that owner.

```
owner (A[i]) {
  n = number of elements in A
  block_size = n / n_lw_threads
  //n_lw_threads is the number of LOCALWRITE threads
  //Assume n is evenly divisible by n_lw_threads
  return (i / block_size)
}
```

Figure 6. Ownership function that assigns ownership of blocks of contiguous array elements to different threads

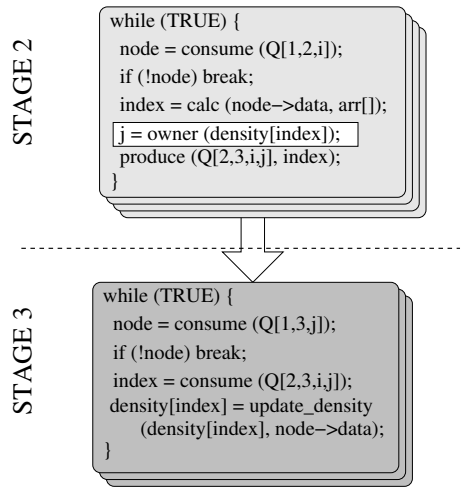


Figure 7. Moving the ownership check to Stage 2 eliminates redundancy in DSWP+LOCALWRITE.

3.2.3 DSWP+SpecDOALL

From Figures 1(a) and (c), statement C is conditionally self-dependent with respect to the contents of each `node`, the array `arr`, and the `calc` function; in other words, the probability of this dependence is a function of the input and is significantly smaller than 1. This stage can be parallelized using Speculative-DOALL. However, for the reasons mentioned in Section 2, the simplified dependence structure of the stage gives a much higher degree of confidence than the original loop.

Figure 8 shows the code transformation. Iterations of the loop in Stage 3 are executed concurrently on multiple threads. Original loads and stores become speculative loads and stores (`tx_load` and `tx_store` in Figure 8). As in DSWP+DOALL, Stage 2 is modified to produce values to the Stage 3 threads in a round-robin fashion.

DSWP+SpecDOALL allows the system to restrict speculation (and also the risk of misspeculation) to a fraction of the loop. By allowing stages to `peek` at queue entries without immediately dequeueing them, misspeculation recovery of inter-core queues has almost no cost. The queue entries that are peeked are dequeued when the iteration commits. Loop termination causes the `misspec` handler to be invoked. Conventional hardware or software TLS memory systems can be used to provide the transactional support [14, 17, 20, 24].

4. Evaluation

We evaluate DSWP+ on a dual quad-core (total of 8 cores) x86 machine. Table 1 gives the details of the evaluation platform. The results are obtained by manual application of DSWP+. The manual transformations proceeded systematically as a modern compiler would do, taking care to avoid exploiting human-level knowledge of the application's overall structure and purpose. GCC, LLVM-GCC, and LLVM

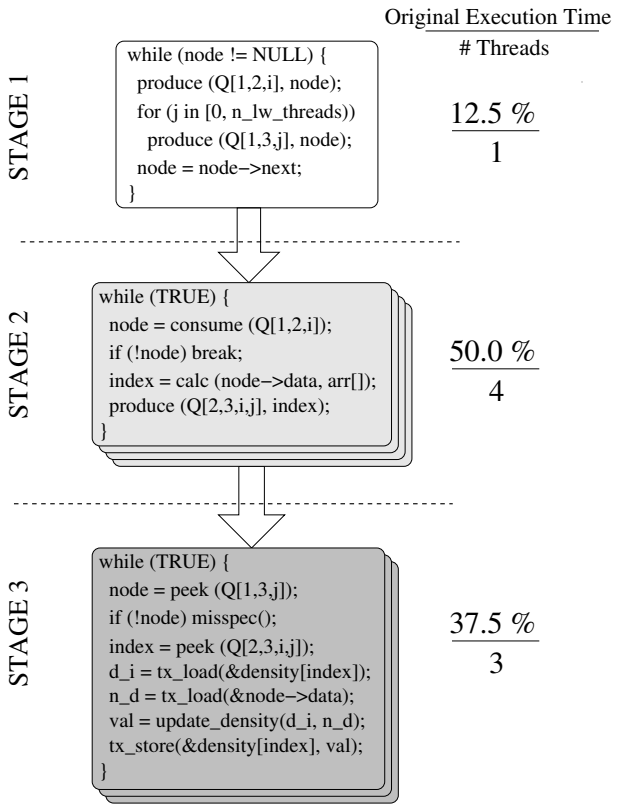


Figure 8. DSWP+ with DOALL applied to the second stage and SpecDOALL applied to the third stage. With 4 threads assigned to Stage 2 and 3 threads to Stage 3, speedup can be improved to 8x (= $1/T_{\text{Stage 1}}$).

Processor	Intel Xeon®E5310
Processor Speed	1.60GHz
Processor Configuration	2 processors X 4 cores
L1 Cache size	32KB (per core)
L2 Cache size	4096KB (per 2 cores)
RAM	8GB
Operating System	Linux 2.6.24
Compiler	GCC and LLVM

Table 1. Platform details

with DSWP and DSWP+DOALL implementations assisted the parallelizations. With the exception of `CG`, DSWP+ parallelized the hottest loop in each benchmark. Like many scientific applications, most of the loops in `CG` (inside function `conj_grad`) are amenable to DOALL. As a result, parallelizing the remaining loop that has irregular dependence patterns becomes critical.

Table 2 gives detailed information about each benchmark including its source [2, 13, 19, 21], name of the function containing the parallelized loop, fraction of benchmark execution time constituted by the loop, and the parallelization techniques applied in conjunction with DSWP+.

4.1 Results

In Figure 9, each bar represents the *best* performance of a technique on *up to* 8 threads. For each benchmark, the first bar indicates the speedup obtained with whichever of DOALL, LOCALWRITE, and SpecDOALL is applicable. If the parallel optimization (*PAR_OPTI*) is not applicable to the unmodified code, then the speedup is shown as 1x. The second bar indicates the speedup with DSWP which tries to balance the work done by each stage. The third bar indicates the speedup with DSWP+; recall that DSWP+ creates unbalanced stages with the intent to parallelize the larger ones. By itself, the speedup with DSWP+ is worse than with DSWP. However, applying DSWP+ with one or more of DOALL, LOCALWRITE, and SpecDOALL results in significantly greater performance as indicated by the fourth bar.

Benchmark	Source Suite	Function	% of Runtime	PAR_OPTI with DSWP+
`ks`	Ref. Impl.	`FindMaxGp-AndSwap`	99.4	DOALL
`otter`	Ref. Impl.	`find_lightest_geo_child`	13.8	DOALL
`052.alvinn`	SPEC CFP	`main`	96.7	DOALL
`filterbank`	StreamIt	`FBCore`	45.6	DOALL
`456.hmmer`	SPEC CINT	`main_loop_serial`	100.0	DOALL
`GTC`	Ref. Impl.	`chargei`	58.8	DOALL, LOCALWRITE
`470.lbm`	SPEC CFP	`LBM_performStreamCollide`	92.4	DOALL, LOCALWRITE, IARD
`CG`	NPB3.2-SER	`sparse`	12.2	LOCALWRITE
`ECLAT`	MineBench	`process_invert`	24.5	LOCALWRITE
`197.parser`	SPEC CINT	`batch_process`	100.0	Spec-DOALL
`256.bzip2`	SPEC CINT	`compressStream`	98.5	Spec-DOALL

Table 2. Benchmark details

4.2 Case Studies

Factors affecting the performance of each benchmark are discussed below.

4.2.1 DSWP+DOALL

- **`ks`** is a graph partitioning algorithm. `FindMaxGp-AndSwap`'s outer loop traverses a linked-list and the inner loop traverses the internal linked-lists, finding the internal linked-lists' minimum value. The linked-list traversal's loop carried dependence prevents a DOALL parallelization. DSWP+ splits the loop into two stages: the first stage traverses the outer linked-list, while the second stage traverses the internal linked-list. After min-reduction is applied, multiple traversals on the inner loop may proceed simultaneously in a DOALL-style parallelization. By spawning multiple copies of the second stage, which significantly outweighs the first, DSWP+DOALL gets much better speedup than DSWP.
- **`otter`** is an automated theorem prover for first-order and equational logic. The parallelized loop is similar to the one in `ks`. While a loop iteration in `ks` takes about 5 µs on average, it takes only $0.03 - 0.22$ µs in `otter`.

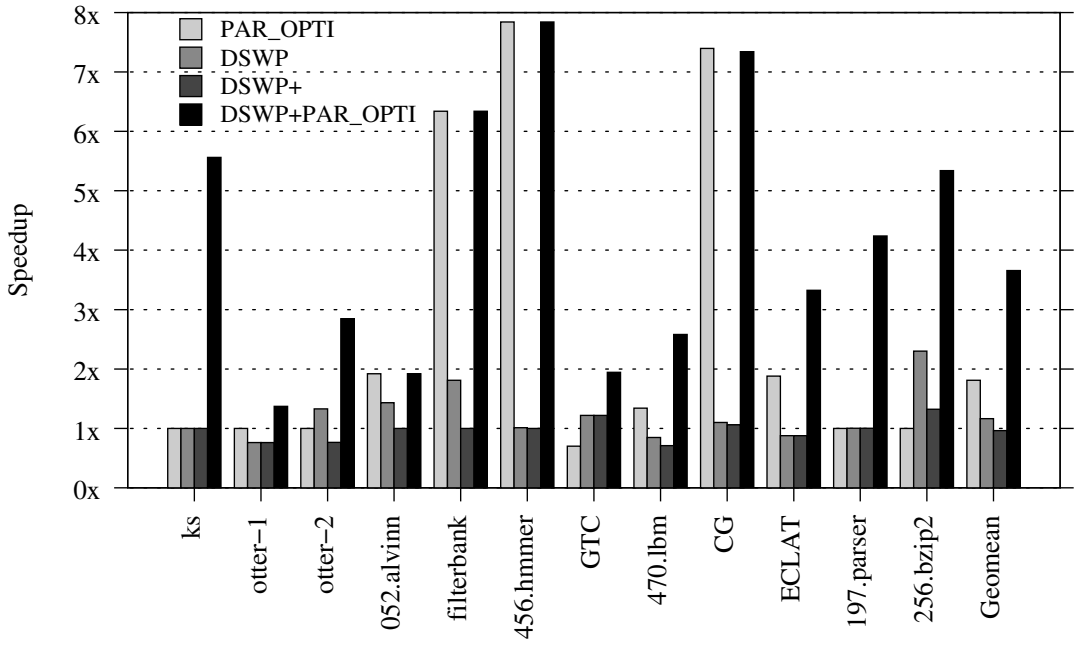


Figure 9. Loop speedup on up to 8 threads using different parallelization techniques

Consequently, the communication instructions for synchronization in the sequential stage constitute a much larger fraction of the loop iteration time, limiting the speedup. The balance between the two stages depends on the input: Figure 9 shows that DSWP achieves a much better balance and speedup on input2 (`otter-2`) compared to input1 (`otter-1`).

- **`052.alvinn`** is a backward-propagation-based artificial neural network. The parallelized loop is the second level loop in the loop hierarchy in `main`. DOALL is not directly applicable because there are loop-carried dependences on array updates. However, when combined with accumulator-expansion, DOALL yields 1.9x speedup. DSWP partitions several loops inside the parallelized loop onto different stages. By using communication queues between stages as buffers for intermediate results, DSWP performs dynamic privatization of the arrays, yielding a speedup of 1.4x with two threads. DSWP+DOALL recognizes that the loop-carried dependence in both DSWP stages can be removed using accumulator-expansion, and merges them into a single parallel stage followed by a sequential stage to perform reduction on the expanded arrays. This generates code that is equivalent to DOALL with accumulator-expansion. DSWP+DOALL also yields a speedup of 1.9x, with the sequential reduction stage limiting the performance.
- **`filterbank`** applies a set of filters for multirate signal processing [21]. As in `052.alvinn`, the outer loop in `FBCore` has two major inner loops and is not amenable to DOALL because of inter-iteration dependences on array updates. DSWP generates a pipeline with two stages, with each inner loop in one stage, yielding a speedup of 1.8x. DSWP+DOALL applies accumulator-expansion, merges the two stages into a parallel stage and performs the reduction in a sequential stage. This yields a speedup of 6.3x.

  Although the structure of `052.alvinn` and `filterbank`'s parallelizations are very similar, the resulting performance results are not. The difference in the speedup of `052.alvinn` and `filterbank` is due to the size of the reduction. In `052.alvinn`, there are two arrays with a combined size of 150K bytes that need to be privatized; in `filterbank`, only one array of size 16K bytes needs to be privatized. With more DOALL threads, the reduction overhead soon becomes the performance bottleneck. This limits the scaling of speedup in `052.alvinn` to six threads.
- **`456.hmmer`** is a computational biology application that searches for patterns in DNA sequences using Profile Hidden Markov Models. Scores are calculated in parallel on sequences which are randomly selected. The Commutative annotation is used to break the dependence inside the random number generator [12]. While the scores are calculated in parallel, a sequential stage computes a histogram of the scores and selects the maximum score. With the Commutative annotation and max-reduction, the loop can be parallelized using DOALL. DSWP+DOALL generates essentially the same code, and performance is

```
for (m=0; m<mi; m++) {
  for (n=0; n<4; n++) {
    v1 = cal_v1 (m, n);
    ... ...
    v8 = cal_v8(m, n);
    i1 = cal_i1 (m, n, A1[ ]);
    ... ...
    i8 = cal_i8 (m, n, A1[ ]);
    densityi[i1] = densityi[i1] + v1;
    ... ...
    densityi[i8] = densityi[i8] + v8;
  }
}
```

Figure 10. Example loop from `GTC`: Main dependence pattern is $A[i] = A[i] + B$

also the same. Speedup is limited by the sequential reduction phase.

4.2.2 DSWP+LOCALWRITE(+DOALL)

While LOCALWRITE is applicable to most irregular reductions, it suffers from the problem of redundant computation that significantly affects its performance potential. Referring back to Figure 1(a), applying just the LOCALWRITE transformation would result in statement C being guarded by an ownership check, and the entire loop being replicated across threads. Since the linked-list traversal and calculation of the index on each iteration is performed by every thread, performance scaling is impeded. By extracting out the common code executed by each LOCALWRITE thread into a separate stage, DSWP+ alleviates the problem of redundant computation and makes LOCALWRITE better-performing.

- **`GTC`** is a 3D particle-in-cell simulator that studies microturbulence in magnetically confined fusion plasma. Figure 10 shows a simplified form of the `GTC` loop. As in the example code in Figure 1, there are loop-carried dependences because of the $A[i] = A[i] + B$ irregular reduction pattern. LOCALWRITE is applicable to the loop; however, the rest of the computation in the loop is replicated in each thread. The result is a slowdown over sequential execution. DSWP+ partitions the loop into a producer stage that calculates the index and values, and a consumer stage that updates the array elements. DSWP+DOALL is applied to the producer since it does not have loop-carried dependences, while DSWP+LOCALWRITE is applied to the consumer since it has irregular reductions. This hybrid parallelization yields close to 2x speedup over sequential execution.

  `GTC`'s performance is limited by several factors. First, the producer communicates 64 values per iteration. While DSWP is tolerant of long communication latencies, the instructions executed to produce a value constitute an overhead. Second, since each consumer thread consumes values from all the producer threads, a single slow producer thread may cause a consumer thread to wait for it rather than process the values produced by the other faster producer threads.The overhead caused by the above two factors can be mitigated if each loop iteration executes long enough. However, since each iteration takes only 0.55 µs, the performance improvement is limited.

```
for (i=START, i<END; i=i+ISTEP) {
  if (OBSTACLE (i)) {
    dstGrid[cal_i1(i, A[ ])] =
      srcGrid[cal_i1'(i, A[ ])];
    ... ...
    dstGrid[cal_i19(i, A[ ])] =
      srcGrid[cal_i19'(i, A[ ])];
    continue;
  }
  cal_rho (i, srcGrid, A[ ]);
  cal_ux (i, srcGrid, A[ ]);
  dstGrid[cal_j1(i, A[ ])] =
    srcGrid[cal_j1'(i, A[ ])] + cal_v1(rho, ux);
  ... ...
  dstGrid[cal_j19(i, A[ ])] =
    srcGrid[cal_j19'(i, A[ ])] + cal_v19(rho, ux);
}
```

Figure 11. Loop in `470.lbm`: Main dependence pattern is $A[i] = B$

- **`470.lbm`** implements the "Lattice Boltzmann Method" to simulate incompressible fluids in 3D. Figure 11 shows a simplified form of the parallelized loop. The main difference from `GTC` is the array update pattern which is $A[i] = B$ compared to $A[i] = A[i] + B$ in `GTC`. While the latter update to the same element can be done in any order, the former needs to respect the original sequential ordering. The iteration space is divided into chunks that are assigned to each producer in a round-robin manner. When a producer finishes an iteration chunk, it produces an "end" token to all of its queues. Each consumer thread starts from the queue holding values from the earliest iteration chunk. When it sees an "end" token, it switches to the next queue. The sequential processing of the iteration space using the "end" tokens guarantees the correct order of update of each array element. This technique yields 1.3x speedup on 8 threads, with the extra "end" token based synchronization limiting the amount of work that is done in parallel in the consumer.

  `470.lbm` shows an interesting array (dstGrid) access pattern that can be used to improve performance (see Figure 12). By using the IARD technique proposed in [18], the iterations can be partitioned into private regions and shared regions. Iterations in different private regions access non-overlapping array elements and thus can be executed concurrently, while iterations in shared regions might access the same array element and thus need synchronization. Profiling shows that the access pattern is very stable. With this information, DOALL is applied

```
Min(a): minimum updated array element index in iteration a
Max(a): maximum updated array element index in iteration a
for any two iterations x and y:
  if x is before y,
  then Min(x) <= Min(y) and Max(x) <= Max(y)
```

Figure 12. Array access pattern in `470.lbm`

to the private regions and DSWP+LOCALWRITE is applied to the shared region. This improves the speedup to 2.6x.

- **CG** from the NPB3.2-SER benchmark suite solves an unstructured sparse linear system by the conjugate gradient method [2]. The loop in `CG` contains the $A[i] = A[i] + B$ and $A[i] = B$ dependence patterns seen in `GTC` and `470.lbm` respectively. Compared to the parallelization model used in those programs, the index and value computation stage in `CG` is very small and is executed sequentially.

  Both LOCALWRITE and DSWP+LOCALWRITE are able to extract scalable speedup because there is hardly any redundant computation. LOCALWRITE is slightly better performing than DSWP+LOCALWRITE on 8 threads because DSWP+LOCALWRITE allocates one thread to the small sequential producer stage leaving 7 threads for parallel execution, whereas LOCALWRITE has all 8 threads available for parallel execution. The problem with DSWP+LOCALWRITE can be overcome by re-using the producer thread for parallel work after completing the sequential work. Speedup is limited primarily by the input size.

- **ECLAT** from MineBench is a data mining benchmark that uses a vertical database format [13]. The parallelized loop traverses a list of items and appends each item to corresponding list(s) in the database based on the item's transaction number. The loop is partitioned into two stages with the first stage calculating the item's transaction number and the second stage appending the transaction to the corresponding list(s). Transactions that do not share the same transaction number can be inserted into the database concurrently. Applying DSWP+LOCALWRITE to the second stage yields 3.32x speedup. As with `GTC`, LOCALWRITE is limited by the redundant computation of each item's transaction number and achieves only 1.87x speedup.

4.2.3 DSWP+SpecDOALL

- **197.parser** is a syntactic parser of the English language based on link grammar. The parsing of a sentence is grammatically independent of the parsing of others. The loop is split up into a sequential stage that reads in the sentences and determines whether it is a command or an actual sentence, and a speculatively DOALL stage that does the parsing of the sentence. Values of several global data structures need to be speculated to parse sentences in parallel. While these structures are modified inside an iteration, they are reset at the end of each iteration to the same values that they had at the beginning of that iteration. Branches that taken under special circumstances are speculated to not be taken. Loop speedup is affected primarily by the number of sentences to parse and the variability in sentence length.

- **256.bzip2** performs data compression using the Burrows-Wheeler transform. The loop is split up into three stages: The first one reads in the input, performs an initial compression, and then outputs blocks; the second stage compresses the blocks in parallel; the third serializes the blocks and outputs a bit-stream. The second stage requires privatization of the block data structure and speculation to handle error conditions while compressing the blocks. Speedup is limited by the input file's size and the level of compression.

5. Related Work

Many techniques have been proposed to extract thread-level parallelism from scientific and general-purpose applications. This paper integrates many of these techniques such as DOALL, LOCALWRITE, and SpecDOALL with DSWP [1, 10, 15].

Data Write Affinity with Loop Index Prefetching (DWA-LIP) is an optimization based on LOCALWRITE [9]. Like DSWP+LOCALWRITE, DWA-LIP also eliminates redundant computation in LOCALWRITE. It does this by prefetching loop indices, but since DWA-LIP takes the whole iteration as a parallel unit, it misses parallelism in loops that contain multiple array element updates. DSWP+LOCALWRITE can split the updates across multiple stages and provide more scalable and finer-grained parallelization.

Parallel Stage DSWP (PS-DSWP) is an automatic parallelization technique proposed by Raman et al. to improve the scalability of DSWP by applying DOALL to some stages of the DSWP pipeline[16]. DSWP+ derives its insight from this extension, and generalizes PS-DSWP by creating pipeline stages optimized for arbitrary parallelization techniques. The code transformation done by DSWP+DOALL is the same as PS-DSWP. By applying techniques like LOCALWRITE and SpecDOALL to stages with loop-carried dependences, DSWP+ can extract more parallelism than PS-DSWP.

Loop distribution isolates parts of a loop with loop-carried dependences from the the parts without these dependences [11]. Like PS-DSWP, loop distribution is used to extract a loop to which DOALL can be applied. The technique proposed in [23] is the speculative counterpart of loop distribution + DOALL, and targets loops that are almost DOALL. Other techniques such as LRPD, R-LRPD, and master/slave speculative parallelization have also been proposed to parallelize loops [5, 17, 24]. In contrast to these approaches, DSWP+ not only extracts the parts without loop-carried dependences, but also extracts parts with dependence patterns that are amenable to parallelization techniques other than (speculative) DOALL. This significantly improves both applicability and performance scalability. While loop distribution executes the sequential part of the loop *followed* by the parallel part, DSWP+ overlaps the execution of different parts of the original loop through pipeline parallelism.

6. Conclusion

This paper introduces the idea that DSWP is an *enabling transformation* that creates opportunities for various parallelization techniques to become applicable and wellperforming. By splitting up a loop with complex dependence patterns into new loops each with a dependence pattern amenable to other parallelization techniques, DSWP uncovers opportunities to extract scalable parallelism from apparently sequential code. This paper describes in detail the code transformations that occur when DSWP+ is applied in conjunction with DOALL, LOCALWRITE, and SpecDOALL. Since it leverages automatic compiler techniques, the proposed parallelization framework can be automated in future work. An evaluation of DSWP+ on a set of codes with complex dependence patterns yielded a geomean speedup of 3.69x on up to 8 threads. This surpasses the geomean speedups of DSWP (1.16x) or other parallel optimizations (1.81x) acting on their own.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. Additionally, we thank the anonymous reviewers for their insightful comments. The authors acknowledge the support of the GSRC Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This material is based upon work supported by the National Science Foundation under Grant No. CCF-0811580. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] M. J. Bridges. *The VELOCITY Compiler: Extracting Efficient Multicore Execution from Legacy Sequential Codes*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[4] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–884, August 1986.

[5] F. H. Dang, H. Yu, and L. Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 318, 2002.

[6] J. R. B. Davies. Parallel loop constructs for multiprocessors. Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, May 1981.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[8] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 43–52, New York, NY, USA, February 2008.

[9] E. Gutiérrez, O. Plata, and E. L. Zapata. Improving parallel irregular reductions using partial array expansion. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 38–38, New York, NY, USA, 2001. ACM.

[10] H. Han and C.-W. Tseng. Improving compiler and run-time support for irregular reductions using local writes. In *LCPC '98: Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, pages 181–196, London, UK, 1999. Springer-Verlag.

[11] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing*, pages 407–416, November 1990.

[12] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[13] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. *IEEE Workload Characterization Symposium*, 0:182–188, 2006.

[14] C. E. Oancea and A. Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE '08: Proceedings of the 1st International Workshop on Multicore Software Engineering*, pages 23–32, New York, NY, USA, 2008. ACM.

[15] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–116, November 2005.

[16] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, April 2008.

[17] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[18] D. E. Singh, M. J. Martin, and F. F. Rivera. Runtime characterisation of irregular accesses applied to parallelisation of irregular reductions. *Int. J. Comput. Sci. Eng.*, 1(1):1–14, 2005.

[19] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

[20] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, February 2005.

[21] StreamIt benchmarks. http://compiler.lcs.mit.edu/streamit.

[22] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, September 2007.

[23] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proc. of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

[24] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 85–96, November 2002.