Automatic Thread Extraction with Decoupled Software Pipelining

Guilherme Ottoni   Ram Rangan   Adam Stoler   David I. August

Departments of Computer Science and Electrical Engineering
Princeton University
{ottoni, ram, astoler, august}@princeton.edu

Abstract

Until recently, a steadily rising clock rate and other uniprocessor microarchitectural improvements could be relied upon to consistently deliver increasing performance for a wide range of applications. Current difficulties in maintaining this trend have lead microprocessor manufacturers to add value by incorporating multiple processors on a chip. Unfortunately, since decades of compiler research have not succeeded in delivering automatic threading for prevalent code properties, this approach demonstrates no improvement for a large class of existing codes.

To find useful work for chip multiprocessors, we propose an automatic approach to thread extraction, called Decoupled Software Pipelining (DSWP). DSWP exploits the fine-grained pipeline parallelism lurking in most applications to extract long-running, concurrently executing threads. Use of the non-speculative and truly decoupled threads produced by DSWP can increase execution efficiency and provide significant latency tolerance, mitigating design complexity by reducing inter-core communication and per-core resource requirements. Using our initial fully automatic compiler implementation and a validated processor model, we prove the concept by demonstrating significant gains for dual-core chip multiprocessor models running a variety of codes. We then explore simple opportunities missed by our initial compiler implementation which suggest a promising future for this approach.

1 Introduction

For years, a steadily growing clock speed and other uniprocessor microarchitectural improvements could be relied upon to consistently deliver increased performance for a wide range of applications. Recently, however, this approach has faltered. Meanwhile, the exponential growth in transistor count remains strong, tempting major microprocessor manufacturers to add value by producing chips that incorporate multiple processors. Unfortunately, while chip multiprocessors (CMPs) increase throughput for multiprogrammed and multithreaded codes, many important applications are single threaded and thus do not benefit.

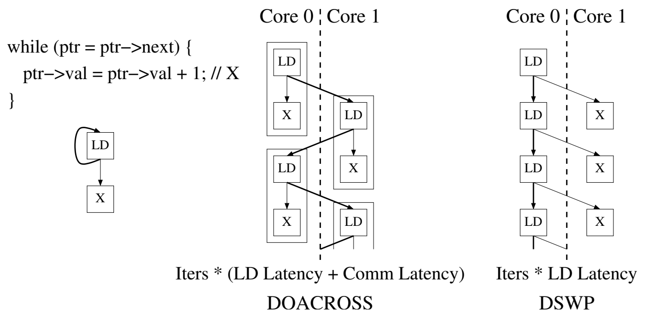
Despite the routine use of powerful instruction-level parallelism (ILP) compilation techniques on a wide variety of unmodified applications, compiler writers have been unable to repeat such success for thread-level parallelism (TLP) despite the pressing need. While success of this type has not been achieved, progress has been made. Techniques dedicated to parallelizing scientific and numerical applications are used routinely in such domains with good results [13]. Such techniques perform well on counted loops manipulating very regular, analyzable structures, consisting mostly of predictable array accesses. In many cases, sets of completely independent (DOALL) loop iterations occur naturally or are easily exposed by loop traversal transformations. Unfortunately, the prevalence of control flow, recursive data structures, and general pointer accesses in ordinary programs renders these techniques unsuitable.

Since automatic thread extraction has been hard for compiler writers to achieve, computer architects have turned to speculative [5, 12, 24, 25, 29, 32] and multiple-pass [14, 3] techniques to make use of additional hardware contexts. These techniques are promising, but generally require significant hardware support to handle recovery in the case of mis-speculation or to effect the warming of microarchitectural structures. These approaches are also limited by the increasing mis-speculation rates and penalties encountered as they become more aggressive. Even the best of these techniques do not replace the need for automatic, non-speculative thread extraction. Instead, they play an important and largely orthogonal role.

In this paper, we propose an effective, *fully automatic* approach to non-speculative thread extraction, called *Decoupled Software Pipelining* (DSWP). DSWP exploits the fine-grained *pipeline parallelism* lurking in most applications to extract long-running, concurrently executing threads. Since extracting fine-grained pipelined parallelism requires knowledge of microarchitectural properties, automating DSWP frees the programmer from the difficult and even counter-productive involvement at this level. DSWP also complements coarser-grained manual threading, speculative threading, and prefetch threading techniques. Operating at the instruction level also allows compiler writers to leverage decades of ILP compilation work and, as we demonstrate, allows thread extraction to be easily added to existing compiler back-ends.

Use of the non-speculative and truly decoupled threads produced by DSWP can increase execution efficiency and provide significant latency tolerance, mitigating design complexity by reducing inter-core communication and per-core resource requirements. Using our initial fully automatic compiler implementation and a validated processor model, we prove the concept by demonstrating significant gains for a dual-core chip multiprocessor running a variety of codes. We also explore simple opportunities missed by our initial compiler implementation, which suggest a very promising future for this approach.

This paper first presents the details of the DSWP algorithm in Section 2. Section 3 then describes the specifics of our implementation of DSWP on an aggressive ILP compiler. Section 4 presents an evaluation of DSWP, and Section 5 analyzes several case studies for added insight. Prior work is visited in Section 6. Finally, Section 7 concludes.



```
while (ptr = ptr->next) {
  ptr->val = ptr->val + 1; // X
}
```

Figure 1. A simple linked list traversal loop executed as DOACROSS and DSWP. For illustration purposes, the pointer-chasing load is labeled "LD" and the body of the loop is labeled "X".

2 Decoupled Software Pipelining

One way to understand why decoupled software pipelining (DSWP) is effective is to start with an examination of the salient properties of DOACROSS parallelism [13]. DOACROSS parallelism is interesting for non-scientific codes because loops in these codes often have dependences among the iterations of the loop. DOACROSS parallelism is characterized by the concurrent execution of parts of each loop iteration across multiple cores. Dependences are respected by forwarding values from core to core by some means, often through memory with synchronization.

Consider the linked list traversal of Figure 1. In the DOACROSS case, each iteration is assigned alternately to each core on a dual-core machine. The pointer chasing load dependence is forwarded from core to core on each iteration. While DOACROSS overlaps the execution of the body of the loop in the current iteration with the next field traversal load in the next iteration, communication costs may more than completely negate such gains. This is a consequence of routing the loop critical path (the pointer chasing load recurring dependence) between the cores on each iteration, extending the critical path (and hence the completion of the loop body) by at least the average communication latency multiplied by the number of iterations.

The simple, key insight of DSWP is that the loop critical path dependence need not even once be routed from core to core to achieve pipelined parallelism. This alternative is illustrated in the right side of Figure 1. In this case, rather than placing each iteration alternately on each core, DSWP breaks the loop iteration up, placing the first part, the pointer chasing load, on Core 0 and placing the second part, the body of the loop, on Core 1. As a consequence of this, the loop critical path dependence remains on Core 0 and is therefore not subject to delay by communication latency.

Unlike techniques exploiting DOACROSS parallelism and other prior non-speculative partitioning techniques [17, 21], DSWP demands that the flow of data among cores is acyclic. This implies that the instructions of each recurrence (there may be several) must be scheduled on the same core as all other instructions of that same recurrence. This acyclic flow creates an opportunity for decoupling when inter-core queues are used to buffer inter-core values. Different recurrences are often assigned to different cores in practice, since by definition the dependences among them are acyclic. Section 4 shows that this insight provides decoupling of up to thousands of instructions between cores even with relatively small inter-core queues. Overall, in addition to utilizing parallel resources (cores) better, DSWP allows for superior latency tolerance through decoupled execution.

Clearly, the DSWP requirement that all instructions in a recurrence remain within a thread may limit the loops amenable to DSWP. For example, one could construct a loop consisting of only a single cross-iteration dependence chain. In such a case, DSWP would not be applicable without help, but neither would any other non-speculative technique. Section 5 describes our experience regarding the applicability of DSWP on existing codes in the face of this limitation. That such cases are rare in codes after ILP optimizations have been performed is one key observation of this paper.

While DSWP does impose a restriction regarding recurrences, it does not have other limitations associated with DOACROSS techniques. The extraction of DOACROSS parallelism is often more restricted than implied in the prior discussion. In many cases, such transformations require loops to be counted, to operate solely on arrays, to have regular memory access patterns, or to have simple (or even no) control flow [6, 13]. Observe in subsequent discussions that DSWP as presented in this paper does not have any of these restrictions.

2.1 Communication and Synchronization Architectural Model

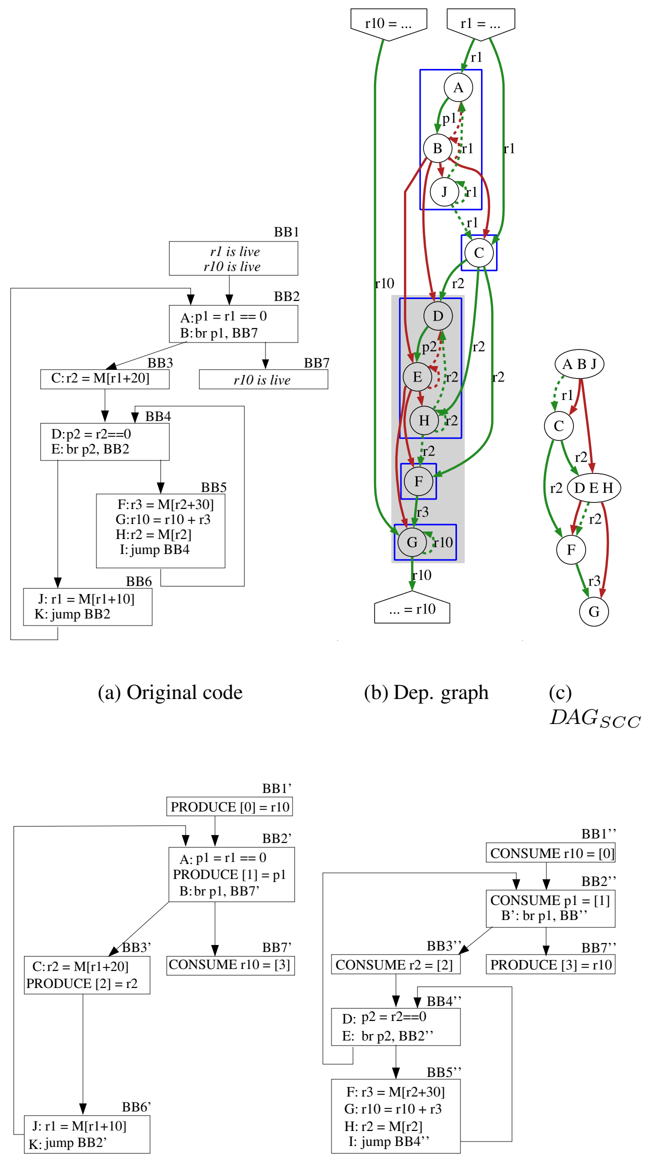
The DSWP architectural model assumed here has a simple message passing mechanism that can communicate one word of data per message, similar to what is available in scalar operand networks, and which can be implemented very efficiently in the hardware [28]. Two special instructions, `produce` and `consume`, are used to send and to receive values respectively. For clarity and ease of use, the `produce` and `consume` instructions take an operand that identifies a communication channel (queue) to operate upon. The `produce` and `consume` instructions are matched in order, and the compiler can rely on this property to correctly transform the code.

While queue latency is not important (as described above and measured later in Section 4), synchronization overhead, as it affects the forward progress of a thread individually, is very important because it may slow the critical path recurrence. To avoid the synchronization overhead associated with software implemented shared queues, the `produce` and `consume` instructions block only when enqueuing to a full queue and when dequeuing from an empty queue, but otherwise operate freely. Others describe how such inter-core queues can be implemented [20, 23].

2.2 The DSWP Algorithm

This section illustrates the DSWP algorithm as it operates on the code of Figure 2(a), which traverses a list of lists of integers and computes the sum of all the element values. After performing DSWP on the outer loop in Figure 2(a), it is transformed into two threads shown in Figures 2(d)-(e). In this example, the loop in Figure 2(d) is executed as part of the main thread of the program, the one which includes the un-optimized sequential portions of the code.

There are several important properties of the transformed code to be observed. First, the set of original instructions is partitioned between the two threads with one instruction in both (`B` as `B` and `B'`). Also notice that DSWP does *not* replicate the control-flow graph completely, but only the parts that are relevant to each thread. In order to respect dependences, `produce` and `consume` instructions are inserted as necessary. For example, instruction `C` writes a value into `r2` that is then used by instructions `D`, `F`, and `H` in the other thread. Queue 2 is used to communicate this value as indicated in the square brackets. Note that, within the loop, the dependences only go in one direction, from the producer to the consumer thread. This acyclic nature, along with the queue communication structures, provides the decoupling described earlier while executing the body of the loop. Outside the loop, this property need not be maintained; the main thread produces loop live-in values for the other thread and consumes loop live-out values after consumer loop termination.


```
BB1
  r1 is live
  r10 is live
BB2
  A:p1 = r1 == 0
  B:br p1, BB7
BB3
  C:r2 = M[r1+20]
BB7
  r10 is live
BB4
  D:p2 = r2==0
  E:br p2, BB2
BB5
  F:r3 = M[r2+30]
  G:r10 = r10 + r3
  H:r2 = M[r2]
  I:jump BB4
BB6
  J: r1 = M[r1+10]
  K: jump BB2
```

(a) Original code (b) Dep. graph (c) DAG_{SCC}

```
BB1'
  PRODUCE [0] = r10
BB2'
  A: p1 = r1 == 0
  PRODUCE [1] = p1
  B: br p1, BB7'
BB3'
  C: r2 = M[r1+20]
  PRODUCE [2] = r2
BB7'
  CONSUME r10 = [3]
BB6'
  J: r1 = M[r1+10]
  K: jump BB2'
```

```
BB1''
  CONSUME r10 = [0]
BB2''
  CONSUME p1 = [1]
  B': br p1, BB''
BB3''
  CONSUME r2 = [2]
BB7''
  PRODUCE [3] = r10
BB4''
  D: p2 = r2==0
  E: br p2, BB2''
BB5''
  F: r3 = M[r2+30]
  G: r10 = r10 + r3
  H: r2 = M[r2]
  I: jump BB4''
```

(d) Producer thread (e) Consumer thread

Figure 2. DSWP example.

Figure 3 shows the pseudo-code for the DSWP algorithm. It takes as input a loop L to be optimized in an intermediate representation, and modifies it as a side-effect. The following subsections describe each step of the algorithm in detail.

2.2.1 Step 1: Building the Dependence Graph – Line 1

The first step in the DSWP algorithm is to build the dependence graph G for loop L [13]. In this graph, each vertex corresponds to one instruction of L, and the arcs represent the dependences among the instructions ($u \rightarrow v$ indicates that u must execute before v). This dependence graph must

```
    DSWP (loop $L$)
(1)     $G \leftarrow$ build_dependence_graph($L$)
(2)     $SCCs \leftarrow$ find_strongly_connected_components($G$)
(3)     if $|SCCs|$ = 1 then return
(4)     $DAG_{SCC} \leftarrow$ coalesce_SCCs($G, SCCs$)
(5)     $\mathcal{P} \leftarrow$ TPP_algorithm($DAG_{SCC}, L$)
(6)     if $|\mathcal{P}|$ = 1 then return
(7)     split_code_into_loops($L, \mathcal{P}$)
(8)     insert_necessary_flows($L, \mathcal{P}$)
```

Figure 3. DSWP algorithm.

be complete in that it contains all data, control, and memory dependences, both intra-iteration and loop-carried, conservatively including a dependence when its absence cannot be proved. For register data dependences, the compiler needs only to account for true (flow) dependences. Output- and anti-dependences can be ignored since, when instructions related by such a dependence are put in different cores, they will run in different threads, naturally using a different set of registers. Additional control dependences are added just for the purposes of DSWP, for reasons described in Section 2.3.

Figure 2(b) illustrates the dependence graph for the loop in Figure 2(a). The arcs for intra-iteration dependences are represented with solid lines; inter-iteration (or loop-carried) dependences are represented with dashed lines. Data dependence arcs are annotated with the corresponding register holding the value. Control dependence arcs have no label. In this example, there are no memory dependences. Special nodes are included in the top (bottom) of the graph to represent loop live-in (live-out) registers.

2.2.2 Step 2: Thread Partitioning – Lines 2-6

The second step in the algorithm is to ensure an acyclic partitioning by finding the strongly connected components (SCCs) and creating the directed acyclic graph of them, the DAG_{SCC} [27]. The SCCs correspond to instructions collectively participating in a dependence cycle, the loop recurrences. As such, DSWP requires all instructions in the same SCC to remain in the same thread. Step (3) stops the transformation if G has a single SCC, since such a graph is not partitionable into multiple threads. Step (4) coalesces each SCC in G to a single node, obtaining the DAG_{SCC}. Figure 2(b) shows the SCCs delimited by rectangles, and Figure 2(c) shows the corresponding DAG_{SCC}.

Using the concepts above, we now define a *valid partitioning* of the DAG_{SCC}.

Definition 1 (Valid Partitioning) *A* valid partitioning $\mathcal{P}$ *of the DAG_{SCC} is a sequence $P_1, P_2, \ldots, P_n$ of sets of DAG_{SCC}'s vertices (i.e. P_is are sets of SCCs) satisfying the following conditions:*

1. *$1 \leq n \leq t$, where t is the number of threads that the target processor can execute simultaneously.*
2. *Each vertex in DAG_{SCC} belongs to exactly one partition in $\mathcal{P}$.*
3. *For each arc $(u \rightarrow v)$ in DAG_{SCC}, with $u \in P_i$ and $v \in P_j$, we have $i \leq j$.*

A valid partitioning guarantees that all members of partition $P_i \in \mathcal{P}$ can be assigned to a thread loop L_i, and that this loop L_i can be executed in its own context. Condition (3) in Definition 1 guarantees that each arc in the dependence graph G either flows forward to a loop L_j where $j > i$ or is internal to its partition. In other words, this condition guarantees an ordering between the partitions that permits the resulting loops to form a pipeline.

The *Thread-Partitioning Problem* (TPP) is the problem of choosing a *valid partitioning* that minimizes the total execution time of the resulting code. The optimal partitioning of the DAG_{SCC} that minimizes this cost is machine dependent, and can be demonstrated to be NP-complete through a reduction from the *bin packing* problem [8]. In practice, we use a heuristic to maximize the load balance among the threads. This is a commonly used criterion in scheduling and parallelization problems and, as experiments in Section 4 show, generally performs well here. As in a processor pipeline, the more balanced the DSWP stages are, the greater its efficiency. In other words, the thread pipeline is limited by the stage with the longest average latency.

Our heuristic computes the *estimated cycles* necessary to execute all the instructions in each SCC by considering the instruction latency and its execution profile weight. Ideally, function call latencies should include the average latency to execute the callee. The algorithm keeps a set of candidate nodes, whose predecessors have already been assigned to a partition, and proceeds by choosing the SCC node in this set with the largest estimated cycles. When the total estimated cycles assigned to the current partition (P_i) gets close to the overall estimated cycles divided by the desired number of threads, the algorithm finishes partition P_i and starts assigning SCC nodes to partition P_{i+1}. In order to minimize the cost of necessary flows between the threads, the heuristic breaks ties by choosing a candidate SCC that will reduce the number of outgoing dependences from the current partition. The partitioning chosen in Figure 2 puts the top two SCC nodes in P_1, and the remaining three in P_2.

After a partitioning is made, the algorithm estimates whether or not it will be profitable by considering the cost of the `produce` and `consume` instructions that need to be inserted. The TPP_algorithm may indicate that no partitioning is desirable by returning a single partition. In such cases, the algorithm in Figure 3 simply terminates in step (6). Otherwise, it continues by splitting the code of the original loop L according to the partitioning $\mathcal{P}$. In our splitting scheme, loop L_1, the one corresponding to the first partition P_1, remains part of the main program thread. The other threads are placed in new auxiliary threads. Section 3 describes this process in more detail.

2.2.3 Step 3: Splitting the Code – Line 7

Splitting the code involves the following steps:

1. Compute the set of relevant basic blocks (BBs) for each partition P_i. Naturally, this set includes all the BBs in the original loop that contain an instruction assigned to P_i. This set also contains BBs which contain an instruction upon which an instruction in P_i depends, to allow for the proper placement of `produce` and `consume` instructions at the point where dependent values are defined in the code. This preserves the condition under which the dependence occurs. This occurs in BB3" in Figure 2(e).
2. Create the BBs for P_i.
3. Place instructions assigned to P_i in the corresponding BB, maintaining their original relative order within the BB.
4. Fix branch targets. In cases where the original target does not have a corresponding BB in the same thread, the new target is set to be the BB corresponding to the closest relevant post-dominator BB of the original target. This is illustrated in the new loop in Figure 2(d) by the arc going from the BB3' to BB6'.

With the above steps, control flow will be respected because branch instructions were assigned to P_i directly (e.g. instruction `E` in Figure 2(e)), or they were duplicated to implement a control dependence entering P_i (e.g. instruction `B'` in Figure 2(e)). Additional jumps may be necessary, however, depending on the layout of the BBs in the new loop and subsequent code layout optimizations.

2.2.4 Step 4: Inserting the Flows – Line 8

The last step of the DSWP algorithm inserts the necessary `produce` and `consume` instruction pairs (called *flows*) to guarantee correctness of the transformed code. The flows created can be classified into three categories based upon the dependence type respected by them.

1. *Data Dependence*: a data value is transmitted.
2. *Control Dependence*: a flag indicating a branch direction is transmitted to a duplicated branch. This is illustrated by the control dependence emanating from instruction `B` in Figure 2(b), implemented using queue 1 in Figures 2(d)-(e).
3. *Memory/synchronization Dependence*: no value is transmitted. The flow itself is used as a token to enforce operation ordering constraints. This is useful for preserving necessary memory operation ordering and the ordering of system calls.

Flows can also orthogonally be classified by their position relative to the loop.

1. *Loop Flow*: when an instruction in loop L_j depends on an instruction in loop L_i, a pair of flow instructions are inserted inside loops L_i and L_j. As already mentioned, the necessary `produce` and `consume` instructions are inserted in the points corresponding to the source instruction for this dependence, so as to keep the correct condition under which this dependence occurs. This is illustrated in Figures 2(d)-(e).
2. *Initial Flow*: when an instruction in a loop L_i, $i > 1$, uses a value that is loop-invariant in the original loop L, a flow is inserted prior to the loops L_1 and L_i to deliver the loop-invariant values every time the transformed loop is invoked.
3. *Final Flow*: when an instruction in a loop L_i, $i > 1$, produces a value that is live out of the original loop L, a flow from L_i to the main thread delivers the value after the last iteration for subsequent use.

Redundant flow elimination can be used to avoid communicating a value more than once inside the loop. In addition, code motion can be performed to move initial (final) flow instructions as early (late) as possible to enhance parallelism by overlapping the fill (spill) portion of the DSWP'ed loop with other work.

2.3 Dependence Graph Details

As mentioned in Section 2.2.1, DSWP requires a few extensions to the traditional concept of control dependence. Each of the following subsections describes an extension necessary to make the DSWP transformation correct.

2.3.1 Loop-Iteration Control Dependences

In DSWP, the queues are reused every iteration and, depending on the control-flow path executed, the set of queues used can vary from one iteration to another. Therefore, in order to guarantee correctness, the compiler needs to make sure that values from different loop iterations are correctly delivered. For this purpose, the thread control flow is matched iteration by iteration. This requires some additional control dependences to be inserted which are not accounted for in standard control dependences [13]. We call such dependences *loop-iteration control dependences*. As an example, consider the code in Figure 4(a). Figure 4(b) shows the corresponding standard control dependence graph, in which no instructions are dependent on branch `F`. However, this branch determines whether or not the loop will skip to the next iteration. Additionally, there is no traditional control dependence from branch `B` to `F`. Yet, depending on the direction that `B` takes, `F` might be executed in this iteration.

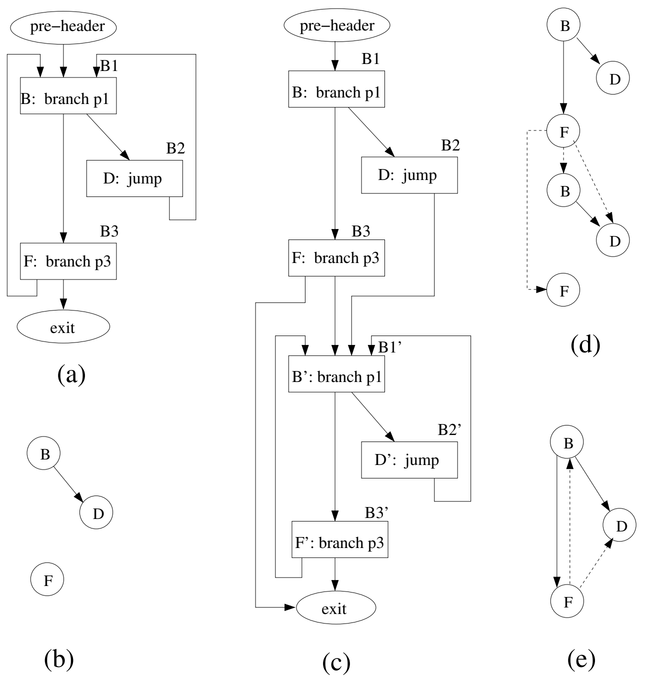


Figure 4. Example of loop-iteration control dependences.

In order to capture such loop-iteration control dependences, we conceptually peel the first iteration of the loop, so that each instruction is duplicated as shown in Figure 4(c). The algorithm then computes the standard control dependences for the peeled version of the code for use on the original code. Figure 4(d) shows the control dependence for Figure 4(c), with the dashed lines corresponding to dependences between different loop iterations. The control dependence graph used for DSWP is then obtained by coalescing corresponding pairs of nodes in the control dependence graph for the peeled code. This is illustrated in Figure 4(e). The resulting control dependence graph includes both standard and loop-iteration control dependences.

2.3.2 Conditional Control Dependences

For dependences that may or may not occur, it is not enough to simply communicate the dependence, but one must also communicate when the dependence occurs. To do this, additional flows are used to communicate the *condition* under which such dependences occur. Consider the example CFG in Figure 5(a), in which there is a data dependence flowing from instruction `D` to instruction `U`. In this example, clearly `D` is control dependent on branch `B`, while `U` is not. If `D` and `U` are assigned to different threads, keeping the semantics of the dependence the same as in the original code involves determining when to enact the flow of data from `D` to `U` through a inter-core flow. To indicate when `U` should

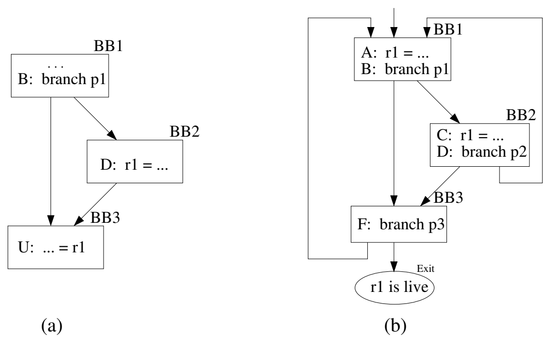


Figure 5. (a) CFG illustrating the need for conditional control dependence. (b) CFG showing the live output dependence problem.

get its value from `D`, the algorithm inserts a dependence arc from `B` to `U`. This dependence ensures that when `U` and `D` are put in different partitions, the dependence condition is communicated to the consuming thread indicating whether to use the current value or to consume a new value from `D` in the producing thread.

A similar problem occurs with the live-out values at the loop exit, if multiple definitions of a live-out value reaching the loop exit are assigned to different threads. This is illustrated by `r1` in Figure 5(b) if instructions `A` and `C` are put in different threads. Additional information would need to be maintained in order to know which definition would occur last in the sequential execution. Alternatively, we adopt a simple solution in this case, which is to not ignore the output dependences among these definitions. This effectively forces these instructions to be on the same SCC, thus executing in the same thread and making it trivial to determine which thread produces the final value. Although this solution can potentially reduce the number of SCCs, we did not observe this drawback in practice.

3 Compiler Implementation

To evaluate DSWP, we implemented it in the back-end of the IMPACT compiler [2]. The IMPACT compiler performs a large number of sophisticated ILP techniques (including Software Pipelining [15]) and delivers exceptional code quality when targeting Itanium 2, often matching or beating Intel's reference compiler on the SPEC-CPU2000 benchmark suite [22]. The compiler we created targets a dual-core Itanium 2 processor.

DSWP was added as a pass in the back-end, operating on ILP optimized predicated code at the assembly level. Memory analysis is the accurate but conservative memory analysis available in the IMPACT compiler [7]. Standard IMPACT profiling tools were used to obtain control flow arc weight used by the DSWP partitioning heuristics. However, function call latencies currently do not include an estimate of the cycles taken to execute the callee, what can lead to poor partitioning decisions for loops with function calls. No optimizations other than scheduling (which includes both traditional software pipelining and acyclic list scheduling) and register allocation are performed after DSWP.

Since we target a dual-core processor model, only two threads are created by the algorithm. These threads are the main thread and one auxiliary thread. To amortize the cost of thread creation, the auxiliary thread is created once, at the beginning of the program. A system call to create a new thread is used, which takes, among other arguments, the address of the function containing the new thread. This *auxiliary thread* function is created by the compiler.

For each optimized loop, the compiler creates a new function containing the corresponding code to be executed by the auxiliary thread. Before entering an optimized loop, the main thread sends to the auxiliary thread the address of the corresponding auxiliary function on a specific queue (the *master queue*). The auxiliary thread, blocked on a `consume` operation on this queue, wakes up and simply calls the function whose address it receives. Upon termination of a loop, the corresponding auxiliary function returns to the master auxiliary function, which loops back to the `consume` instruction. The auxiliary thread then blocks again on the master queue, waiting for the new request from the main thread. The auxiliary thread is terminated by a special terminate signal composed of a NULL function pointer.

4 Evaluation

In this section, we evaluate our implementation of the DSWP compilation technique targeting a dual-core chip multiprocessor. The selected benchmark set includes applications drawn from SPEC-CPU2000, Mediabench [16], and the Unix utility ‘wc’. Applications were discarded from this initial evaluation if they failed to compile in the unmodified development version of IMPACT upon which DSWP is based. They were also discarded if, even after aggressive inlining, no long running loops were visible to the compiler. The `164.gzip` benchmark was the only one in which DSWP was unable to find a multi-SCC DAG_{scc}. While not evaluated with the other benchmarks, it is described later in Section 5. For each application, DSWP is applied to the most important visible loop that executes at least 50 iterations on average each time it is entered.

4.1 Candidate Loop Statistics

Table 1 presents profile statistics for the chosen loops. These loops account for between 16% and 98% of the total benchmark execution time. IMPACT’s front-end preforms aggressive function inlining, which is the reason why most of the loops have no function calls before DSWP. Table 1 also presents the number of SCCs for each loop and gives the number of flows (produce/consume pairs) that were created by the automatic partitioning as per the heuristic in Section 2.2.2.

Benchmark	Loop Ex.%	Nest	BBs	Func. Calls	Instr.	SCCs	# Flows		
							Init.	Loop	Final
129.compress	16	1	1	0	20	18	2	2	1
179.art	21	1	1	0	9	7	3	3	2
181.mcf	36	2	13	0	71	23	2	20	2
183.equake	67	2	4	0	202	33	0	23	1
188.ammp	64	3	38	3	630	244	1	47	1
256.bzip2	17	3	161	18	917	127	2	116	5
adpcmdec	98	1	21	0	52	38	3	7	2
epicdec	29	1	2	0	28	4	3	4	1
jpegenc	20	1	1	0	15	13	5	4	2
wc	90	1	2	0	17	13	3	3	4

Table 1. Statistics for the selected loops in the benchmark suite.

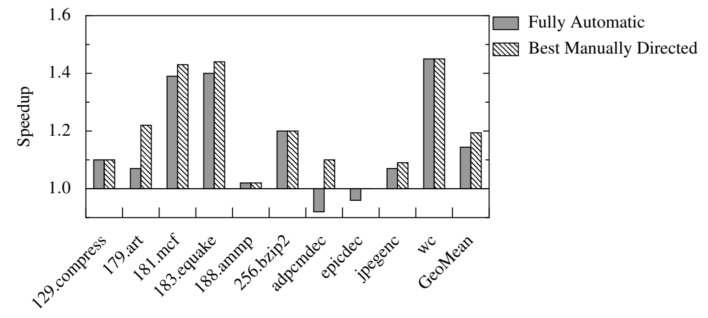
Note that three of the selected loops are actually DOALL loops, namely the ones from `129.compress`, `179.art`, and `jpegenc`. Although DSWP can be applied to these loops, as presented in this work, parallelizing them as independent threads is likely more efficient because it avoids all overhead of inter-thread communication during loop execution.

4.2 Performance Evaluation: Dual-Threaded vs. Baseline

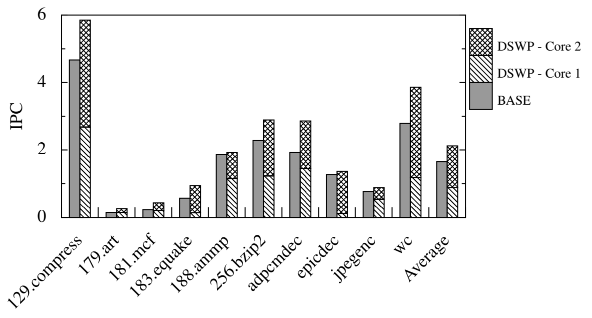
To evaluate the performance of DSWP, we used a validated cycle-accurate Itanium 2 processor [11] performance model (IPC accurate to within 6% of real hardware for benchmarks measured [19]) to build a dual-core CMP model comprising two Itanium 2 cores connected by the *synchronization array* communication mechanism proposed in [20]. The models were built using the Liberty Simulation Environment [30].

The synchronization array (SA) in the model works as a set of low-latency queues. In our implementation, there is a total of 256 queues, each one with 32 elements. The SA has a 1-cycle read access latency and has four request ports that are shared between the two cores. The IA-64 ISA was extended with `produce` and `consume` instructions for inter-thread communication. These instructions use the M pipeline, which is also used by memory instructions. This imposes the limit that only 4 of these instructions (minus any other memory instructions) can be issued per cycle on each core, since the Itanium 2 can issue only four M-type instructions in a given cycle. While the `consume` instructions can access the SA speculatively, the `produce` instructions write to the SA only on commit. As long as the SA queue is not empty, a `consume` and its dependent instructions can execute in back-to-back cycles.

The highly-detailed nature of the validated Itanium 2 model prevented whole program simulation. Instead, detailed simulations were restricted to the loops in question in each benchmark. We fast-forwarded through the remain-



(a) Speedup of DSWP over single-threaded.



(b) Baseline and DSWP IPC.

Figure 6. Performance summary: full-width Itanium 2 baseline.

ing sections of the program while keeping the caches and branch predictors warm.

The first comparison made was between the dual-threaded, DSWP version of the selected loops against their single-threaded, base versions. In this experiment, the latency to produce/consume a value to/from the synchronization array was set to 1 cycle in our simulator (minimum 2 cycles core-to-core). In order to evaluate both the effectiveness of our partitioning heuristic (from Section 2.2.2) and the potential of better heuristics, Figure 6(a) presents two speedup bars per benchmark loop. The first bar is the fully automated DSWP, using the heuristic. The second one corresponds to the best performing partitioning found by iteratively specifying the desired partitioning to the compiler and measuring its resulting performance. Figure 6(a) shows that in many cases the heuristic found the best partitioning we were able to find in our iterative search. The geometric mean across these benchmark loops is 14.4% and 19.4%, for the automatically created and manually specified partitions respectively. In terms of whole-program speedup, these geometric means translate into 6.6% and 9.2% respectively. The average baseline IPC is 1.65 and the IPC averages for the producer and consumer cores are 0.88 and 1.24 respectively as shown in Figure 6(b). Notice that these IPC numbers do not include the `produce` and `consume` instructions inserted by DSWP.

For simplicity, the simulator used did not model the cost of coherence protocol. To gauge the effect of this coherence cost on our results, we analyzed, for all benchmarks, the memory traces of both cores for false-sharings. Notice that true-sharings cannot occur inside the loop because of the memory dependence arcs in the dependence graph. If a load and a store may access the same memory address, there will be arcs in both directions between these instructions, because of the RAW and WAR dependences (one loop carried, and the other intra-iteration). This makes both instructions part of the same SCC, assigning them to the same core. Future work may relax this condition.

In order to analyze false-sharings, we replayed the memory accesses from the traces in an invalidation-based coherence model offline. Out of the nine applications, only three (`181.mcf`, `256.bzip2`, and `jpegenc`) exhibited false-sharing. In `181.mcf` and `jpegenc`, the false-sharing was always caused by writes in the consumer core to locations already present in the producer core's L1D cache. While in `181.mcf`, the miss-rate of the producer core's L1D went up by 0.01% to 98.62%, in `jpegenc`, there was *no* change in the miss-rate of both cores. The reason why the miss-rate is unaffected is because although there is false-sharing, the producer core always runs ahead and accesses any locations it needs *before* those locations are invalidated by writes to memory from the consumer's core. `256.bzip2` has a slightly more interesting behavior. We find that a particular store from the producer core causes a lot of false-sharing and hence invalidations in the consumer's L1 data cache. Since the consumer trails the producer, any extra latency arising out of such events could adversely affect the consumer thread and ought to have been modeled. However, on examining the code in detail, we found that *all* these coherence conflicts are caused by false-sharing due to a write to a global variable (`bsLive`) in the producer core. We promoted this global variable to a register and used the modified version of 256.bzip2 for all experiments. Thus, even without coherence modeling, our results are valid and not overstated.

Figure 7 illustrates the importance of balancing work across threads when partitioning loops. The figure shows the DAG_{SCC} for the target loop in `181.mcf`. Each SCC is labeled with the number of instructions it contains. Each left-to-right line crossing the DAG_{SCC} illustrates one possible way of partitioning it into two threads. For each possible partitioning, the figure also illustrates the resulting speedup, the corresponding synchronization array occupancy for a sample period and the cumulative cycle distribution at each possible occupancy level. An occupancy of negative one means the corresponding queue is empty and

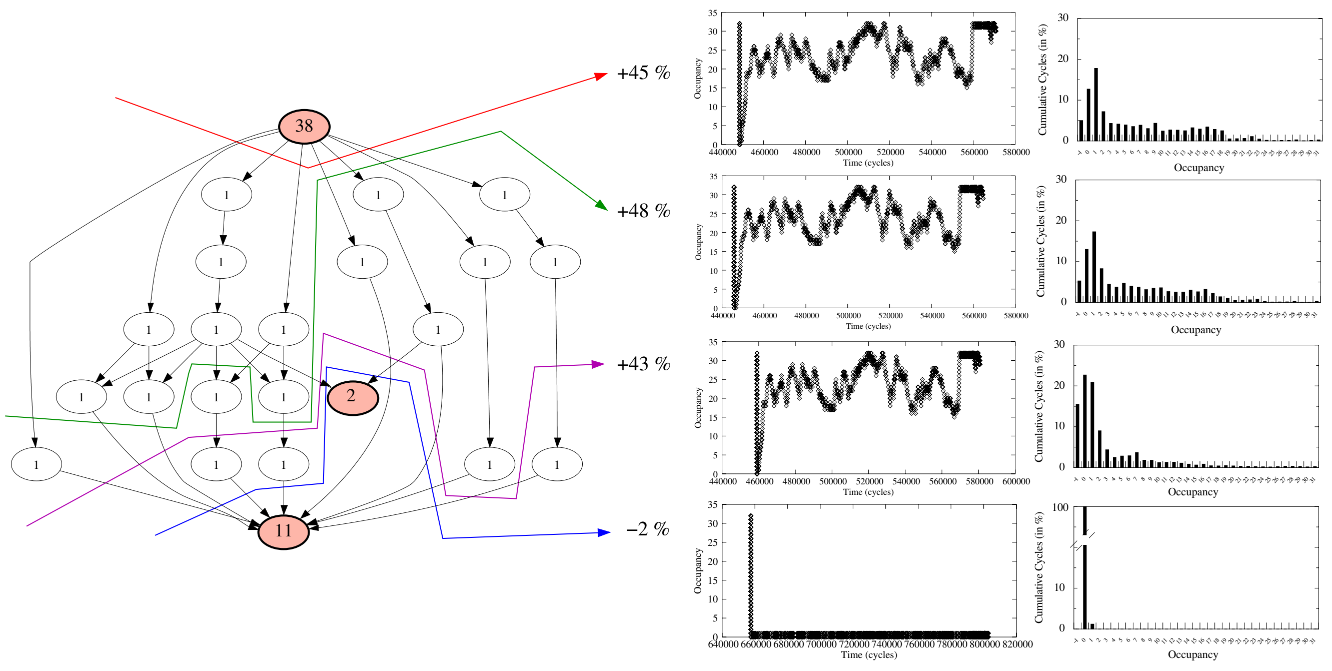


Figure 7. Importance of balancing: DAG_{SCC} for 181.mcf with different partitions. Numbers in each DAG_{SCC} node indicate the number of instructions in the SCC. Graphs on the right illustrate the occupancy of the synchronization array for a sample interval, and the distribution of the occupancy over the whole execution.

the consumer is stalled. The occupancy graphs illustrate the decoupling effect between the threads, and how they are able to make progress concurrently – a thread is only stalled by the other when the synchronization array is either full or empty. The partitioning chosen automatically by our heuristic is the one corresponding to 43% speedup. Note that all partitionings result in good speedups, except for the last one in which the threads are not well balanced. This imbalance can be seen by the fact that the synchronization array is usually empty, because too much work was assigned to the first thread, in particular the three load instructions difference between it and the compiler partitioning, causing the second thread to be blocked most of the time.

Figure 8 summarizes the occupancy histograms for all benchmarks. These give us an idea of how well the thread-balancing heuristic works in practice. In particular, these graphs provide vital feedback to the compiler designer, so that the heuristics can be designed to avoid stalls resulting from full or empty queues.

4.3 Performance Compatibility: Simpler Cores

Future CMP's are very likely to have simpler cores and so we evaluated the performance compatibility of the automatically-generated DSWP codes versus baseline codes across full-width and half-width models. We use a variant of the validated Itanium 2 model described earlier with half of the instruction fetch and dispersal width of the baseline Itanium 2 model.

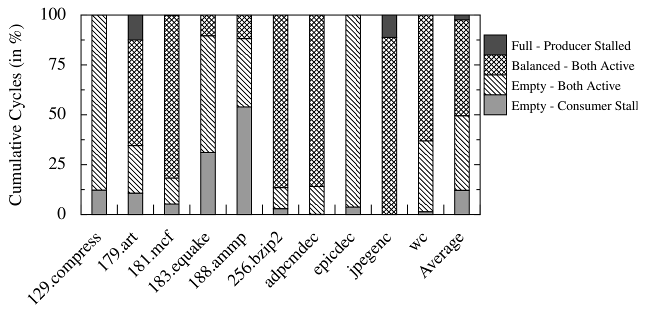
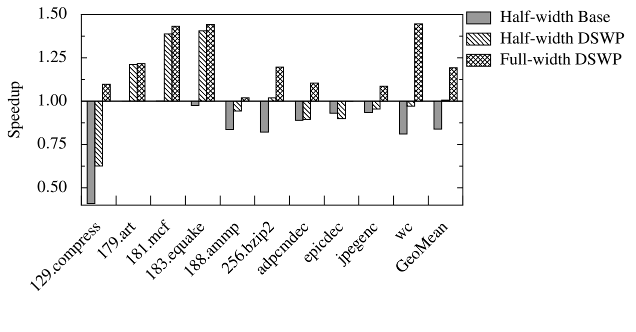
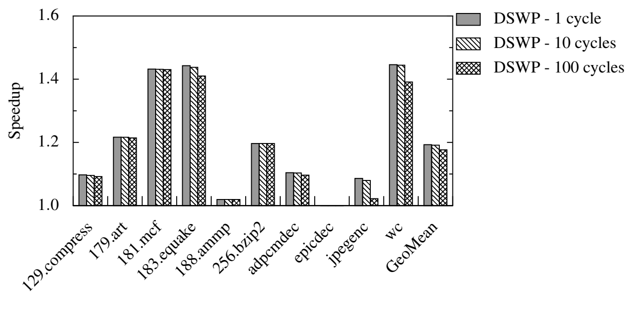


Figure 8. Cumulative cycle distribution at all occupancy levels.

Figure 9(a) presents the performance results, for both the single-threaded and the DSWP versions. On average, DSWP on a CMP with half-width cores performs *better* than a full-width core running the single-threaded (ST) version. The graph shows that DSWP-compiled codes have better performance compatibility than standard ILP-compiled codes across architectures with varying pipeline widths. Additionally, note that the speedup of half-width



(a) Varying issue-widths



(b) Varying communication latencies

Figure 9. Performance compatibility and sensitivity analyses: full-width Itanium 2 baseline.

DSWP over half-width ST is greater than the speedup of full-width DSWP over full-width ST. This occurs because DSWP sometimes trades ILP for TLP. Thus, the simpler and less powerful a core is, the more pronounced the benefits of DSWP are.

4.4 Sensitivity Analyses: Communication Latency and Queue Size

In order to quantify the importance of communication latency for DSWP, we ran our experiments again with the full-width CMP model, modified to have communication latencies of 10 and 100 cycles between the two cores. We modeled this pipelined delay in the `produce` instructions, while `consume` instructions continued to take one cycle (representing queue locality at the receiving side). The results are presented in Figure 9(b) and they show that DSWP is not very sensitive to the communication latency. In fact, this was expected due to the design of the DSWP transformation, as discussed in Section 2.

We also evaluated the impact of queue size on performance by varying the queue size to 8 and 128. We found that DSWP executions are fairly insensitive to queue size, with the mean slowdown with size 8 being 2% and the average speedup with size 128 being 1% compared to 32-element queues. The highest slowdown was 6% and the peak speedup was 7% respectively.

5 Detailed Analysis

As already mentioned, DSWP is not applicable in cases where the loop dependence graph has a single SCC. Additionally, even with multiple SCCs, DSWP does not have good partitioning opportunities if there exists large SCCs that create a thread imbalance. In our experiments, we found that DSWP is generally applicable in spite of this requirement. Nevertheless, a few cases were encountered where this restriction was limiting. This section describes these cases and highlights opportunities to mitigate or eliminate the problem.

```
(1)  for (i=0; i<x_size*y_size; i++)
     {
(2)    dtemp = result[i] / scale_factor;
(3)    if (dtemp < 0) result[i] = 0;
(4)    else if (dtemp > 255) result[i] = 255;
(5)    else result[i] = (int) (dtemp + 0.5);
     }
```

Figure 10. Important loop for `epicdec`.

5.1 Case Study: `epicdec` loop

Despite `epicdec`'s simplicity, it illustrates the potential of having a more accurate memory analysis for DSWP. The source code of the loop in question is shown on Figure 10. In the IMPACT compiler, this loop is unrolled once. During this process, we notice that both memory loads of the `result` array (line (2)) become memory dependent on all the stores. Because of this, the resulting dependence graph has only 4 SCCs, with all the loads and stores as part of a single SCC. Despite this, DSWP achieves a speedup. Closer inspection revealed that these were false memory dependences, conservatively inserted by earlier optimizations. A solution proposed in [10] addresses this problem by performing accurate memory analysis at the assembly level.

Better scheduling and partitioning resulting from removal of these false memory dependences gave the DSWP and the base codes a 98% and an 87% speedup over the original base code. We then measured the average instructions per cycle (IPC), for both new versions, and verified that the base had an average of 2.33, while the average for the DSWP threads were 1.26 and 1.37 respectively. This showed us that DSWP was trading ILP for TLP, and that additional ILP optimizations could potentially improve the results. We then applied more aggressive unrolling (8x), and recompiled both versions once again. The new DSWP'ed version resulted in a 45% speedup relative to the new base version by better utilizing the dual-core resources.

5.2 Case Study: `adpcmdec` loop

Lack of predicate-aware dependence analysis for DSWP caused spurious dependences to be inserted in our depen-

```
for (ti=0;ti<numf1s;ti++)
  Y[tj].y += f1_layer[ti].P * bus[ti][tj];
```

Figure 11. Important loop for `179.art`.

dence graphs. We manually identified many spurious dependences in `adpcmdec`. We circumvented this problem by applying DSWP to the same loop without hyperblock formation and this resulted in an increased number of SCCs (38, up from 4) with a better distribution of instructions (largest SCC had 15% of instructions, as opposed to 94% in the original). This allowed us to see a 10% speedup, as reported in Section 4. We suspect that predication may be hindering application of DSWP in other benchmarks too.

5.3 Case Study: `179.art` loop

The source code of the `179.art` loop is shown in Figure 11. Performing accumulator expansion [18] on the summing variable (Y[yj].y) allows the compiler to obtain 14 SCCs, 7 up from the original, giving a 60% speedup with an IPC of 0.17 and 0.12 for the two cores. Accumulator expansion also improves baseline scheduling resulting in a 56% speedup (over the original base) with an IPC of 0.21. The differences in the IPCs suggest that performing aggressive unrolling on the producer and consumer threads, can further improve DSWP performance, like in `epicdec`.

5.4 Case Study: `164.gzip`

In general, the producer must contain the loop termination condition. Sometimes, like in the `deflate_fast` loop of `164.gzip`, computation of this condition may be highly serialized resulting in one huge SCC, making it unfit for DSWP. A simple and likely profitable fix is to move loop termination detection to the consumer and provide support that will allow the latter to correctly reconcile all producer thread side-effects with the architectural state. Such speculation support will improve the applicability of DSWP.

6 Related Work

While statically-scheduled processors cannot deal very well with variable latencies, even dynamically-scheduled out-of-order machines exhibit poor in-order-like behavior in practice due to instruction window size limitations. Recognizing this, Rangan et al. proposed the idea of Decoupled Software Pipelining (DSWP) for thread-parallel architectures [20] and evaluated it with hand-modified codes of recursive data structure loops. They showed how DSWP is complimentary to dynamic out-of-order instruction scheduling and speculative techniques like prefetching (even perfect prefetching). This paper builds on [20], by establishing DSWP as a more general parallelization technique, proposing the first compiler algorithm for it, and evaluating it with a fully automated implementation across a diverse set of benchmarks.

While similar in name, Software Pipelining (SWP) [15] rearranges loop instructions to create an instruction pipeline, whereas DSWP partitions and schedules loop code to create a pipeline of threads. Although SWP is a very effective ILP technique, it performs poorly in the presence of variable latency instructions (e.g. loads) [22]. DSWP, on the other hand, is able to achieve better latency tolerance, thanks to the exploitation of coarse-grained parallelism and the decoupled execution of the thread pipeline. Both SWP and DSWP can be applied simultaneously and, in fact, are in the compiler used in this work.

DSWP was inspired in part by decoupled access-execute architectures (DAE) [23, 4, 31, 21], which tolerate latency by decoupling memory accesses from other work. Since dependences go both ways between the Access and the Execute cores, no single thread of execution can run ahead and exploit any coarse-grained parallelism [26]. DSWP avoids this problem at thread-pipeline creation time, by avoiding circular dependencies amongst threads.

Other general-purpose, non-speculative parallelization techniques exist, but these techniques often require special programming languages with parallel constructs [1]. Success has also been achieved for streaming applications, through the use of specialized programming languages [9], in effect requiring the programmer to rewrite the application to expose parallelism. These techniques ultimately rely on the programmer to identify thread-level parallelism.

Other means to unearth coarse-grained parallelism include a variety of thread-level speculation techniques [5, 12, 24, 25, 29, 32] which are orthogonal to DSWP and can compliment it.

7 Conclusion

This paper presented *Decoupled Software Pipelining* (DSWP), a new compilation technique to extract *non-speculative* thread-level parallelism from application loops. Contrary to traditional parallelization techniques, DSWP handles all kinds of dependences, effectively exploiting *pipeline parallelism* found in ordinary, general-purpose applications. The experimental results showed that DSWP is applicable to most application loops. Using a dual-core simulator built on top of validated Itanium 2 core models and an implementation in a high-quality optimizing compiler, DSWP achieves a mean speedup of 19.4% on important benchmark loops, translating to a mean of 9.2% over entire benchmarks. When executing on a reduced complexity core, DSWP turns a 17.1% slowdown for the loops in the original single threaded codes to a slight speedup, suggesting a decent performance compatibility for simpler cores.

In addition to the promising initial results achieved in this work, this paper showed that these results can be further improved by subsequent work. More accurate memory analysis, additional optimizations to break dependence cycles, more elaborate partitioning heuristics, and new optimizations to reduce the number of flows are among the directions for future work. We believe that DSWP can be an enabler for related TLP research in as much as instruction scheduling has been for ILP, with new optimizations being discovered as more is learned about DSWP in practice.

References

[1] J. N. Amaral, G. Gao, E. D. Kocalar, P. O'Neill, and X. Tang. Design and implementation of an efficient thread partitioning algorithm. In *Proceedings of the International Symposium on High Performance Computing*, pages 252–259, 2000.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predication and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227–237, June 1998.

[3] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. W. Hwu. Beating in-order stalls with 'flea-flicker' two-pass pipelining. In *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[4] M. E. Benitez and J. W. Davidson. Code generation for streaming: An access/execute mechanism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, United States, April 1991.

[5] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 99–108, 2002.

[6] D.-K. Chen. *Compiler Optimizations for Parallel Loops with Fine-grained Synchronization*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1994.

[7] B.-C. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W H Freeman & Co, New York, NY, 1979.

[9] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, 2002.

[10] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August. Practical and accurate low-level pointer analysis. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[11] Intel Corporation. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*. Santa Clara, CA, 2002.

[12] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 59–70, 2004.

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

[14] D. Kim and D. Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Trans. Comput. Syst.*, 22(3):326–379, 2004.

[15] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[16] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[17] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. P. Amarasinghe. Space-time scheduling of instruction-level parallelism on a Raw Machine. In *The Proceedings of the Eighth International Confrence on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, 1998.

[18] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara. Compiler code transformations for superscalar-based high-performance systems. In *Proceedings of Supercomputing '92*, pages 808–817, November 1992.

[19] D. A. Penry, M. Vachharajani, and D. I. August. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation (MOBS)*, June 2005.

[20] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

[21] K. Rich and M. Farrens. Code partitioning in decoupled compilers. In *Proceedings of the 6th European Conference on Parallel Processing*, pages 1008–1017, Munich, Germany, September 2000.

[22] J. W. Sias, S. zee Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. mei W. Hwu. Field-testing IMPACT EPIC research results in Itanium 2. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[23] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th International Symposium on Computer Architecture*, pages 112–119, April 1982.

[24] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, June 1995.

[25] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *The 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

[26] M. Sung, R. Krashinsky, and K. Asanovic. Multithreading decoupled architectures for complexity-effective general purpose computing. *SIGARCH Comput. Archit. News*, 29(5):56–61, 2001.

[27] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[28] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, February 2005.

[29] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[30] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, pages 271–282, November 2002.

[31] W. A. Wulf. Evaluation of the wm architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 382–390, Queensland, Australia, May 1992.

[32] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.