Advances in Parallel-Stage Decoupled Software Pipelining

Leveraging Loop Distribution, Stream-Computing and the SSA Form

Feng Li
INRIA
feng.li@inria.fr

Antoniu Pop
Centre de recherche en informatique,
MINES ParisTech
antoniu.pop@mines-paristech.fr

Albert Cohen
INRIA
albert.cohen@inria.fr

Abstract

Decoupled Software Pipelining (DSWP) is a program partitioning method enabling compilers to extract pipeline parallelism from sequential programs. *Parallel Stage DSWP* (PS-DSWP) is an extension that also exploits the data parallelism within pipeline filters.

This paper presents the preliminary design of a new PS-DSWP method capable of handling arbitrary structured control flow, a slightly better algorithmic complexity, the natural exploitation of nested parallelism with communications across arbitrary levels, with a seamless integration with data-flow parallel programming environments. It is inspired by loop-distribution and supports nested/structured partitioning along with the hierarchy of control dependences. The method relies on a data-flow streaming extension of OpenMP.

These advances are made possible thanks to progresses in compiler intermediate representation. We describe our usage of the *Static Single Assignment* (SSA) form, how we extend it to the context of concurrent streaming tasks, and we discuss the benefits and challenges for PS-DSWP.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors-Compilers, Optimization

General Terms optimization

Keywords automatic parallelization, stream-computing, loop distribution

1. Introduction

In recent years, the CPU manufacturers have embraced chip multiprocessors because of technology, power consumption and thermal dissipation constraints, and because of diminishing returns in instruction-level parallelism. The amount of performance gained by the use of multicore processor depends highly on the software fraction that can be parallelized to run on multiple cores simultaneously. Multiprocessor programming leaves the burden to programmer who faces the extra complexity, heisenbugs, deadlocks and other problems associated with parallel programming. The situation is worse when dealing with the migration of legacy code.

Decoupled Software Pipelining (DSWP) is an automatic thread partitioning method which could partition a sequential program to run on multiple cores, and Parallel-Stage DSWP (PS-DSWP) exposes data parallelism into task pipelines extracted by DSWP. These automatic thread partitioning methods free the programmer from manual parallelization. They also promise much wider flexibility than data-parallelism-centric methods for processors, aiming for the effective parallelization of general-purpose applications.

In this paper, we provide another method to decouple control-flow regions of serial programs into concurrent tasks, exposing pipeline and data parallelism. The power and simplicity of the method rely on the restriction that all streams should retain a synchronous semantics [8]. It amounts to checking the sufficient condition that the source and target of any decoupled dependence are control-dependent on the same node in the control dependence tree (this assumes structured control flow). This restriction may appear as a severe one for experienced parallel programmers; but at the potential expense of adding extra levels of nested parallelism, it does not restrict the degree of pipeline parallelism. In fact, any pair of computational statements can be decoupled and assigned to different concurrent tasks. The partitioning algorithms also handle DOALL parallelization within task pipelines, and arbitrarily nested data-parallel pipelines following the control dependence tree of a structured control flow graph. Unlike existing DSWP algorithms, our method does not explicitly copy conditional expressions and can handle arbitrary backward data and control dependences.

We are using two intermediate representations.

- A conventional SSA-based representation, annotated with natural loop and control dependence trees (for structured control flow).
- And a streaming data-flow extension of the latter representation as a backend for our partitioning algorithm, still in SSA form but with explicit task boundaries (for single-entry single-exit regions) and multi-producer multi-consumer streams to communicate across tasks.

The backend representation streamlines the decoupling of multi-producer multi-consumer data flow through explicit, compiler-controlled sampling and merging stages. Multi-producer multi-consumer semantics is absolutely essential to handle general decoupling patterns where data-parallel stages feature an unbalance in the number of worker threads. Sampling is handled transparently by nesting tasks into enclosing control flow. Merging is captured by Φ functions at task boundaries, introducing a minor variant of the SSA form satisfying the so-called task-closed property that multiple incoming flows targeting the same use in a given task should be explicitly merged by a dedicated Φ function at the task entry point.

Relying on SSA avoids building the complete program dependence graph: with the exception of the array dependence graph, our method only processes linear-size data structures, as opposed to the worst-case quadratic program dependence graph in DSWP.

2. Related Work

The most closely related work to this paper is decoupled software pipelining and loop distribution. We recall the state-of-the-art in both and present the original finding at the source of this work: *by extending loop distribution with pipelining and asserting a synchronous concurrency hypothesis, arbitrary data and control dependences can be decoupled very naturally* with only minor changes to existing algorithms that have been proposed for loop distribution [10].

2.1 Decoupled software pipelining

Decoupled Software Pipelining (DSWP) [13] is one approach to automatically extract threads from loops. It partitions loops into long-running threads that communicate via inter-core queues. DSWP builds a Program Dependence Graph (PDG) [7], combining control and data dependences (scalar and memory). Then DSWP

introduces a load-balancing heuristic to partition the graph according to the number of cores, making sure no recurrence spans across multiple partitions. In contrast to DOALL and DOACROSS [4] methods which partition the iteration space into threads, DSWP partitions the loop body into several stages connected with pipelining to achieve parallelism. It exposes parallelism in cases where DOACROSS is limited by loop-carried dependences on the critical path. And generally speaking, DSWP partitioning algorithms handles uncounted loops, complex control flow and irregular pointer-based memory accesses.

Parallel-Stage Decoupled Software Pipelining [16] (PS-DSWP) is an extension to combine pipeline parallelism with some stages executed in a DOALL, data-parallel fashion. For example, when there are no dependences between loop iterations of a DSWP stage, the incoming data can be distributed over multiple data-parallel worker threads dedicated to this stage, while the outgoing data can be merged to proceed with downstream pipeline stages.

These techniques have a few caveats however. They offer limited support for decoupling along backward control and data dependences. They provide a complex code generation method to decouple dependences among source and target statements governed by different control flow, but despite its complexity, this method remains somewhat conservative.

By building the PDG, DSWP also incurs a higher algorithmic complexity than typical SSA-based optimizations. Indeed, although traditional loop pipelining for ILP focuses on innermost loops of limited size, DSWP is aimed at processing large control flow graphs after aggressive inter-procedural analysis optimization. In addition, the loops in DSWP are handled by the standard algorithm as ordinary control flow, missing potential benefits of treating them as a special case. To address these caveats, we turned our analysis to the state of the art in loop distribution.

2.2 Loop distribution

Loop distribution is a fundamental transformation in program restructuring systems designed to extract data parallelism for vector or SIMD architectures [10].

In its simplest form, loop distribution consists of breaking up a single loop into two or more consecutive loops. When aligning loop distribution to the strongly connected components of the data-dependence graph, one or more of the resulting loops expose iterations that can be run in parallel, exposing data parallelism. Barriers are inserted after the parallel loops to enforce precedence constraints with the rest of the program. An example is presented in Figure 1.

```
for (i = 1; i < N; i++) {
S1  A[i] = B[i] + 1;
S2  C[i] = A[i-1] + 1;
}
```

```
for (i = 1; i < N; i++)
S1  A[i] = B[i] + 1;

<barriers inserted here>

for (i = 1; i < N; i++)
S2  C[i] = A[i-1] + 1
```

Figure 1. Barriers inserted after loop distribution.

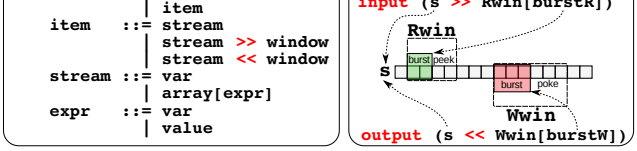
3. OpenMP Extension for Stream-Computing as a Code Generation Target

A recently proposed stream-computing extension to OpenMP [14] allows the expression of pipeline parallelism by making explicit the flow dependences, or producer-consumer patterns, between OpenMP tasks. It provides a simple way for explicitly building dynamic task graphs, where tasks are connected through streams that transparently privatize the data.

The extension consists of two additional clauses, `input` and `output` to the `task` construct, that define the producer-consumer relationships between tasks. The OpenMP language, with this extension, is a natural fit as a target for our code generation. It provides for dynamic task creation and connection in the task graph, it

```
input/output (list)
    list   ::= list, item
             | item
    item   ::= stream
             | stream >> window
             | stream << window
    stream ::= var
             | array[expr]
    expr   ::= var
             | value
```

```
int s, Rwin[Rhorizon];
int Wwin[Whorizon];
input (s >> Rwin[burstR])
```




```
output (s << Wwin[burstW])
```

Figure 2. Syntax for `input` and `output` clauses.

handles arbitrary nesting of pipelined tasks in control-flow, and it allows the hierarchical nesting of tasks.

The `task` construct is extended with `input` and `output` clauses as presented on Figure 2. Both clauses take a list of items, each of which describes a stream and its behavior w.r.t. the task to which the clause applies. In the abbreviated item form, `stream`, the stream can only be accessed one element a time through the same variable `s`. In the second form, `stream >> window`, the programmer uses the C++ flavoured `<< >>` stream operators to connect a sliding window to a stream, gaining access, within the body of the task, to `horizon` elements in the stream.

One of the main issues that needs to be addressed in order to distribute a PDG to the OpenMP stream-computing extension is that, in the latter, the data flow bypasses the control flow. In other words, when a task produces values on an output stream, these values will all reach the consumers of the stream, even if, in the serial semantics, the values would have been overwritten before reaching the consumers. This means that the only case where a direct annotation scheme will work is if all tasks are in the same control flow. There are multiple ways this issue can be handled, the most systematic one being to always ensure that every producer-consumer pair share the same control dependence. This is achieved by sinking all control flow surrounding the tasks, and not shared by both producer and consumer, in the tasks. To avoid the loss of parallelization opportunities, each task's body can be further partitioned into nested pipelines.

The GCC implementation of the OpenMP extension for stream-computing has been shown to be efficient to exploit mixed pipeline- and data-parallelism, even in dynamic task graphs [14]. It relies on compiler and runtime optimizations to improve cache locality and relies on a highly efficient lock-free and atomic operation-free synchronization algorithm for streams.

4. Observations

It is quite intuitive that the typical synchronization barriers in between distributed data-parallel loops can be weakened, resulting into data-parallel pipelines. We aim to provide a comprehensive treatment of this transformation, generalizing PS-DSWP in the process.

4.1 Replacing loops and barriers with a task pipeline

In the previous example, we could remove the barriers between two distributed loops with pipelining so that the two loops could run in parallel.

```
/* Initialize the stream,
   inserting a delay. */
void INIT_STREAM() {
    produce(stream, A[0]);
}
```

```
/* Decoupled producer and
   consumer. */
for (i = 1; i < N; i++) {
S1  A[i] = B[i] + 1;
    produce(stream, A[i]);
}
for (i = 1; i < N; i++) {
    tmp = consume(stream);
S2  C[i] = tmp + 1;
}
```

Figure 3. Pipelining inserted between distributed loops. Initialize the stream (left), producer and consumer thread (right).

Figure3 shows that pipelined execution is possible: the `INIT_STREAM` function inserts one delay into a communication stream; the

`produce`/`consume` primitives implement a FIFO, enforcing the precedence constraint of the data dependence on array `A` and communicating the value in case the hardware needs this information.

When distributing loops, scalar and array expansion (privatization) is generally required to eliminate memory-based dependences. The conversion to a task pipeline avoids this complication through the usage of communication streams. This transformation can be seen as an optimized version of scalar/array expansion in bounded memory and with improved locality [15].

4.2 Extending loop distribution to PS-DSWP

The similarity between DSWP and distributed loops with data-parallel pipelines is striking. First, both of them partition the loop into multiple threads. Second, both of them avoid partitioning the loop iteration space: they partition the instructions of the loop body instead. But four arguments push in favor of refining DSWP in terms of loop distribution.

1. Loop distribution leverages the natural loop structure, where the granularity of thread partitioning can be easily controlled. Moreover, it is useful to have a loop control node to which to attach information about the iteration of the loop, including closed forms of induction variables; this node can also be used to represent the loop in additional transformations.
2. Using a combination of loop distribution and fusion, then replacing barriers with pipelining leads to an incremental path in compiler construction. This path leverages existing intermediate representations and loop nest optimizers, while DSWP relies on new algorithms and a program dependence graph.
3. Considering the handling of control dependences, a robust and general algorithm already exists for loop distribution. McKinley and Kennedy's technique handles arbitrary control flow [10] and provides a comprehensive solution. The same methods could be applied for DSWP, transforming control dependences into data dependences, and storing boolean predicates into stream. After restructuring the code, updating the control dependence graph and data dependence graph, the code generation algorithm for PDGs [2, 5, 6] can be used to generate parallel code. This solution would handle all cases where the current DSWP algorithm fails to clone a control condition.
4. Since loop distribution does not partition the iteration space, it can also be applied to uncounted loops. Unfortunately, the termination condition needs to be propagated to downstream loops. This problem disappears through the usage of a conventional communication stream when building task pipelines.

From this high-level analysis, it appears possible to extend loop distribution with pipelining to implement PS-DSWP and handle arbitrary control dependences. Yet the method still seems rather complex, especially the if-conversion of control dependences and the code generation step from the PDG. We go one step further and propose a new algorithm adapted from loop distribution but avoiding these complexities.

4.3 Motivating example

Our method makes one more assumption to reduce complexity and limit risks of overhead. It amounts to enforcing the synchronous hypothesis on all communicating tasks in the partition [8]. A sufficient condition is to check if the source and target of any decoupled dependence is dependent on the same control node.

Consider the example in Figure 4. S1 and S7 implement the loop control condition and induction variable, respectively. S2, S3 and S6 are control dependent on S1. S3 is a conditional node, S4, S5 and L1 are control dependent on it. In the inner loop, L2 and L3 are control dependent on L1. When we apply DSWP to the outer loop, the control dependences originating from S1 must be if-converted by creating several streams (the number of streams depends on the number of partitions). When decoupling along the control dependence originating from S3, a copy of the conditional node must be created as well as another stream.

```
S1  while (p != NULL) {
S2    x = p->value;
S3    if(c1) {
S4      x = p->value/2;
S5      ip = p->inner_loop;
L1      while (ip) {
L2        do_something(ip);
L3        ip = ip->next;
        }
      }
S6    ... = x;
S7    p = p->next;
    }
```

Figure 4. Uncounted nested loop before partitioning.

```
S1  while (p1 = Φ^loop(p0,p2)) {
S2    x1 = p1->value;
S3    if(c1) {
S4      x2 = p1->value/2;
S5      ip1 = p1->inner_loop;
L1      while (ip2 = Φ^loop(ip1, ip3)) {
L2        do_something(ip2);
L3        ip3 = ip2->next;
        }
      }
      x3 = Φ^cond_c1(x1, x2);
S6    ... = x3;
S7    p2 = p1->next;
    }
```

Figure 5. Uncounted nested loop in SSA form.

```
//task0-0(main task)
S1  while (p1 = Φ^loop(p0, p2)) {
//persistent-task1-1
#pragma task firstprivate (p1) output(x1)
      {
S2      x1 = p1->value;
      }
//persistent-task1-2
#pragma task firstprivate (p1) output(c1, x2)
      {
S3      if(c1) {
//persistent-task2-1
#pragma task firstprivate (p1) output(ip1) lastprivate(x2)
          {
S4          x2 = p1->value/2;
S5          ip1 = p1->inner_loop;
          }
//persistent-task2-2
#pragma task input(ip1)
          {
L1          while (ip2 = Φ^loop(ip1, ip3)) {
//parallel - task3-1
#pragma omp task firstprivate (ip2)
              {
L2              do_something(ip2);
              }
L3            ip3 = ip2->next;
            }
          }
        }
      }
//persistent-task1-3
#pragma task input(c1, x1, x2)
      {
        x3 = Φ^cond_c1(x1, x2);
S6      ... = x3;
      }
S7    p2 = p1->next;
    }
```

Figure 6. Loops after partitioning and annotated with OpenMP stream extension.

Figure 5 shows the conversion to SSA form. Just like GCC, we use a loop-closed SSA form distinguishing between loop-Φ and cond-Φ nodes. The latter take an additional condition argument, appearing as a subscript, to explicit the selection condition. The

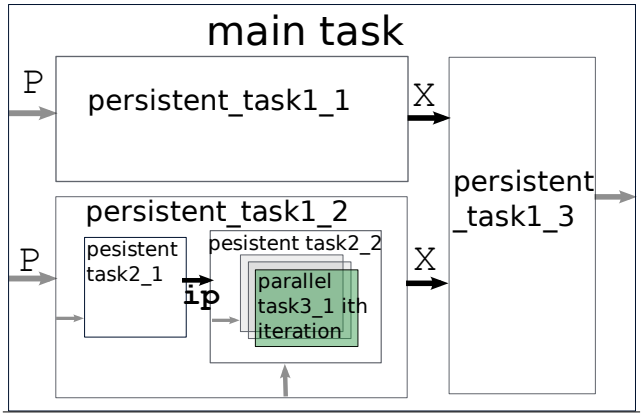


Figure 7. Pipelining and parallelization framework.

partitioning technique will build a stream to communicate this condition from its definition site to the cond-Φ node's task.

We build on the concept of *treegion*, a single-entry multiple-exit control-flow region induced by a sub-tree of the control dependence graph. In the following, we assume the control flow is structured, which guarantees that the control dependence graph forms a tree. Every sub-tree can be partitioned into concurrent tasks according to the control dependences originating from its root. Any data dependence connecting a pair of such tasks induces communication over a dedicated stream. We call `taskM_N` the `N`-th task at level `M` of the control flow tree.

In Figure 5, after building the control dependence tree, one may partition it into 3 tasks (`task1_1`, `task1_2` and `task1_3`) at the root level, and for `task1_2`, one may further partition this task into inner nested tasks `task2_1` and `task2_2`. One may then check for data parallelism in the inner loops; if they do not carry any dependence, one may isolate them in additional data-parallel tasks, such as `task3_1` in this example.

Figure 6 shows the task and stream-annotated code using an OpenMP syntax. Figure 7 shows the nested pipelining and data parallelization corresponding to the partitioned code. The main task will be executed first, and a pipeline will be created for the main task and its inner tasks three `task1_1`, `task1_2` and `task1_3`. Among these, the same variable `x` used to be defined in the control flow regions of both `task1_1` and `task1_2`, to be used in `task1_3`. This output dependence must be eliminated prior to partitioning into tasks, so that `task1_1` and `task1_2` could be decoupled, while `task1_3` may decide which value to use internally.

Nested tasks are introduced to provide fine grained parallelism. It is of course possible to adapt the partition and the number of nesting levels according to the load balancing and synchronization overhead. The generated code will be well structured, and simple top-down heuristics can be used.

In the execution model of OpenMP 3.0, a task instance is created whenever the execution flow of a thread encounters a task construct; no ordering of tasks can be assumed. Such an execution model is well suited for unbalanced loads, but the overhead of creating tasks is significantly more expensive than synchronizing persistent tasks. To improve performance, we use the persistent task model for pipelining, in which a single instance will handle the full iteration space, consuming data on the input stream and producing on the output stream [14]. In Figure 7, all the tasks except `task3_1` use the persistent model to reduce the overhead of task creation; `task3_1` is an ordinary task following the execution model of OpenMP 3.0 (instances will be spawned every time the control flow encounters the task directive). All these tasks will be scheduled by the OpenMP runtime.

One problem with the partitioning algorithms is the fact that the def-use edges (scalar dependences) can become very large, sometimes quadratic with respect to the number of nodes [9]. Figure 8 (left) presents an example that illustrates this problem, Statements S1, S2 define the variable `x`. These definitions all reach the uses in the statements S3, S4 by passing through S5. Because each definition could reach every use, the number of definition-use edges is proportional to the square of the number of statements. These dependences constitute the majority of the edges in a PDG. SSA provide a solution to this problem. In SSA form, each assignment creates a different variable name and at point where control flow joins, a special operation is inserted to merge different incarnations of the same variable. The merge nodes are inserted just at the place where control flow joins. Figure 8 (right) is the original program under SSA form. A merge node (Φ) is inserted at S5, and killed the definition of S1 and S2. We could see here, in the SSA form, we could reduce the definition-use edges from quadratic to linear.

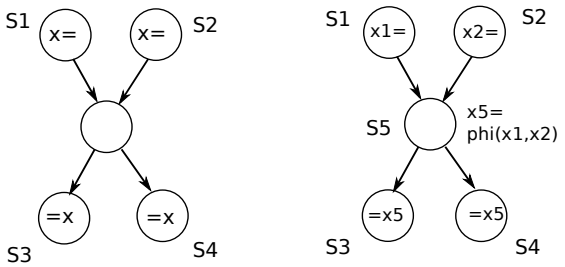


Figure 8. Definition and use edges in the presence of control flow.

The systematic elimination of output dependences is also facilitated by the SSA form, with a Φ node in `task3_1`. Notice that the conditional expression from which this Φ node selects one or another input also needs to be communicated through a data stream.

When modifying loop distribution to rely on tasks and pipelining rather than barriers, it is not necessary to distribute the loop control node and one may run it all in the master task, which in turn will activate tasks for the inner partitions. The statements inside each partition form a treegion whose root is the statement that is dependent on the loop control node. With pipelining inserted, distributed loops could be connected with pipelining when there are data dependences.

One concern here is that loop distribution with task pipelines may not provide expressiveness to extract pipeline parallelism. This is not a problem however, since we may apply the same method to every conditional statement rooted treegion, with some special care to the nested tasks, we could get fine grained parallelism without explicitly decoupling the control dependences. Considering again the example in Figure 4, its control dependence tree is given in Figure 9. The root treegion includes all the nodes in the control dependence graph, `treegion1_2` represents the treegion at conditional level 1 and its root is node 2, `treegion1_3` is at conditional level 1 and includes nodes (S3,S4,S5,L1,L2,L3). `treegion2_1` is in conditional level 2 and its root is node (L1), which is the loop control node of the inner loop.

So following our approach, we may start from the treegion at conditional level 0, which is the whole loop, an implicit task will be created as the master task. For the treegions at level 1, we could create them as sub-tasks running at the context of the main task. If there are data dependences between the treegions at the same level and without recurrence, we will connect them with communication streams. If there is a dependence from the master task to one inner task, the value from the enclosing context can be forwarded to the inner task like in a `firstprivate` clause of OpenMP. Dependences from an inner task to the master task are also supported, although `lastprivate` is not natively supported for OpenMP3.0 tasks, it is a necessary component of our streaming task representation. `lastprivate(x)` is associated with a synchronization point at the end of the task and makes the value of `x` available to the enclosing context. The same algorithms could be recursively applied to the treegion at the next inner level. e.g. For `treegion1_3` at level 1, the sub treegion at level 2 is

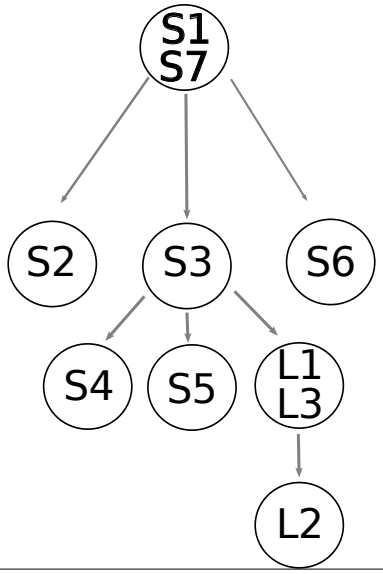


Figure 9. Control dependence graph of Figure 4. Express the definition of treegion.

`treegion2_4`, `treegion2_5` and `treegion2_1`, we could create sub-tasks by merging `treegion2_4` and `treegion2_5` as one sub-task and `treegion2_1` (which is also the inner loop) as one sub-task, or just for part of them. To reveal data parallelism, we can reuse the typed fusion algorithm introduced by McKinley and Kennedy [11]: it is possible to fuse communicating data-parallel nodes to increase the synchronization grain or improve the load balancing. In this example, the loop in node L2 does not carry any dependence, and we need to decouple it from its enclosing task to expose data-parallelism.

5. Partitioning Algorithm

In this section, we present our partitioning algorithm, based on the SSA and treegion representations. We define our model and the important constructs that will be used by our algorithm, then we present and describe our algorithm.

5.1 Definitions

In this work, we are only targeting natural structured loops [3]. Such loops are single-entry single-exit CFG sub-graphs with one entry block and possibly several back edges leading to the header from inside of the loop. `break` and `continue` statements can be preprocessed to comply with this restriction, but we plan to lift it altogether in the future.

Treegion The canonical definition of a treegion is a non-linear, single-entry multiple-exit region of code containing basic blocks that constitute a sub-graph of the CFG. We alter this definition to bear on the Control-Dependence Graph (CDG) instead, so we will be looking at single-entry multiple-exit sub-graphs of the CDG.

Loop Control Node In the representation we employ later, we will use the loop control node to represent the loop. The loop control node include statements which will evaluate the loop control expression and determines the next iteration.

Although control dependences in loops can be handled by the standard algorithm by converting them to a control flow graph, there are advantages in treating them as a special case with coalescing them in a single node (loop control node): not only the backward dependence is removed by building the loop control node so that the control dependence graph will form a tree, but also, this node can be used to represent the loop in all sort of transformations.

Conditional Level The control dependence graph of the structured code is a tree after building the loop control node. The root of the tree is the loop control node at the loop's outermost level. We define the conditional level for every node in the control dependence graph as the depth of the node in the tree. The root of the tree with depth 0 has conditional level 0.

We define the conditional level for the treegion is the conditional level of the root node of the treegion (subtree). We define `treegionN_M` to identify a treegion where `N` is the conditional level of the treegion and `M` is the root node number of the treegion.

5.2 The algorithm

The algorithm takes an SSA representation of a single function, and returns a concurrent representation annotated with tasks and communication streams.

Step 1: Transform Conditional Statements to Conditional Variables To achieve fine-grained pipelining, conditional statements are split to conditional variables. As showed in Figure 10. Full conversion to three-address SSA form is also possible (as it is performed in GCC or LLVM, for example).

```
if (condition(i))
//is transformed to
c1 = condition(i)
if (c1)
```

Figure 10. Split conditional statements to expose finer grained pipelining.

Step 2: Build the Program Dependence Graph under SSA By building the program dependence graph, the control dependence graph, data dependence graph (through memory) and scalar dependence graph (through registers) are built together.

The control dependence graph for the structured code is a tree, the root of the tree is the loop control node. The leaves of the tree are non-conditional statements and the other nodes inside the tree are the conditional statements or the loop control node of the inner loops. We start from building the control dependence graph, and evaluate the conditional level for each node in the graph. Every node inside the control dependence graph is an statement from the compiler's intermediate representation of the loop except for the loop control node. The loop control node will be built by searching the strongly connect component started from the loop header node (at each loop nest level) in the program dependence graph.

The data dependence graph could be built by the array dependence analysis [9] for the loop. We should analyse every pair of data dependences to mark the irreducible edges in a later step if there are recurrence.

Step 3: Marking the Irreducible Edges A partition can preserve all dependences if and only if there exists no dependence cycle spanning more than one output loop [1, 12]. In our case, for the treegion at the same conditional level, if there are dependences that form a cycle, we mark the edges in between as irreducible. If we have statements in different conditional level, we promote the inner one to its ancestor until both of them are in the same treegion, mark the promoted root node and the other root node as irreducible. The algorithms is presented in Figure 11.

Step 4: Structured Typed Fusion Before partitioning, to reveal data parallelism, we type every node in the dependences graph as `parallel` or `!parallel`. If there are loop-carried dependence inside this node, then it should be typed as `!parallel`, otherwise, typed as `parallel`.

The `parallel` type nodes are candidates for data parallelization. The goal is to merge this type of nodes to create the largest parallel loop, reducing synchronization overhead and (generally) improving data locality. Further partitioning can happen in the following step, starting from this maximally type-fused configuration. Given a DAG with edges representing dependences and the vertices representing statements in the loop body, we want to produce an equivalent program with minimal number of parallel loops. We want it to be as large as possible to balance the synchronization

```
// input: PDG Graph PDG(V,E) PDG--Program Dependence Graph
// input: CDG Graph CDG(V,E) CDG--Control Dependence Graph
// output: irreducible_edge_set Irreducible_edge_set
SCCS = find_SCCs(PDG)
For each SCC in SCCs:
  for each pair of node (Vx,Vy) in SCC:
    // CL represents for conditional level
    // in the Control dependence graph.

    // if they are in the same treegion, merge into one node.
    if Vx.CL == Vy.CL:
      merge_to_one_nodes(Vx, Vy)
      continue

    // if not in the same treegion, go up for n=|Vx.CL-Vy.CL|
    // levels. And mark the edge between the nodes as irreducible.

    max_CL = Vx.CL>Vy.CL?Vx.CL:Vy.CL
    Vx = up_n_level(CDG, max_CL - Vx.CL)
    Vy = up_n_level(CDG, max_CL - Vy.CL)
    //mark edge (Vx,Vy) irreducible
    Irreducible_edge_set.insert(edge(Vx,Vy))
```

Figure 11. Algorithm for marking the irreducible edges.

overhead. Even when we don't want that coarse grained parallel loops, we could also partition between iterations if possible.

In our case, we need a structured typed loop fusion algorithm. We revisit McKinley and Kennedy's *fast typed fusion* [11] into a recursive algorithm traversing the control dependence tree. Starting from the treegion at conditional level 0, which is the whole loop, we will check if there are loop carried dependences between iterations. If there are no loop carried dependence, we stop here by annotating the whole loop as parallel. If there are, we are going into each inner treegion, identifying those that have no loop carried dependences. If some of them carried no loop carried dependence, mark the nodes as parallel and try to merge them. There are some constraints when we fuse the nodes: (1) parallelization-inhibiting constraints; (2) ordering constraints. The parallelization-inhibiting fusion is that there are no loop-carried dependences before fusion, but will have the loop carried dependences after. So we should skip this kind of fusion which will degrades data parallelism. The ordering constraints describe that two loops cannot be validly fused if there exists a path of loop-independent dependences between them that contains a loop or statement that is not being fused with them.

The time complexity of the typed fusion algorithms is `O(E+V)` [11], and our structured extension has the same complexity.

```
void StructuredTypedFusion()
  Queue queue = new Queue()
  queue.push(treegions_at_level_0)
  while (not queue.empty()) {
    treegions = queue.pop()
    G = build_pdg_by_treegion(treegions)
    for each treegion in treegions:
      if loop_carried_no_dependence (treegion) {
        parallel_treegion.insert(treegion)
        update_typed_dependence_graph(G, treegion.num)
      }else{
        treegions_at_inner_level.insert(treegion)
      }
    queue.push (treegions_at_inner_level)
    B = Get_parallelization_inhabiting_edges
                    (parallel_treegion)
    t0 = 'parallel'
    TypedFusion (G, T, B, t0)
  }
procedure TypedFusion(G, T, B, t0)
  //G=(V,E) is the TYPED dependences graph,
  //including control,data,scalar dependences.
  //type(n) will return the type of a node.
  //B is the set of parallelization-inhabiting edges.
  //t0 is a specific type for which we will find a minimal fusion
end TypedFusion
```

Figure 12. Structured typed fusion algorithm.

Step 5: Structured Partitioning Algorithms Updating the CDG after typed fusion, start from the treegion which has conditional level 0 for our partitioning algorithms, and for all of its child treegions at conditional level 1, we should decide where to partition. The partition point could be any point between each of these treegions at the same level except the irreducible edges that we have created in step 3. The algorithm may decide at every step if it is desirable to further partition any given task into several sub-tasks.

Look at the example Figure 13:

```
for(i...)

  x = work(i)

  if (c1)
    y = x + i;

    if (c2)
      z = y*y;

    q = z - y;
```

```
for (i...)
  BEGIN task1_1
    x = work(i)
  END task1_1
  BEGIN task1_2
    if (c1)
      BEGIN task2_1
        y = x + i;
      END task2_1
      BEGIN task2_2
        if (c2)
          z = y*y;
      END task2_2
      BEGIN task2_3
        q = z - y;
      END task2_3
  END task1_2
```

Figure 13. Before partitioning (left), and After partitioning (right). Loop with control dependences.

The code in Figure 13 (left) is partitioned into 2 tasks, and one task (`task1_2`) is partitioned further into 3 sub-tasks.

6. Code Generation

After the partitioning algorithms, we have decided the partition point between the original treegions, with the support of the stream extension of OpenMP. We ought to generate the code by inserting the `input output` directives. With the support of nested tasks, relying on the downstream, extended OpenMP compilation algorithm (called OpenMP expansion). But some challenges remain, especially in presence of multiple producers and consumers. We are using SSA form as an intermediate representation and generating the streaming code.

6.1 Decoupling dependences across tasks belonging to different treegions

Clearly if we decouple a dependence between tasks in the same treegion, the appropriate `input` and `output` clauses can be naturally inserted. But what about the communication between tasks at different level?

Considering the example in Figure 14, if we decide to partition the loop to 3 main tasks: task1_1 with S1, task1_2 with (S2,S3), and task1_3 with S4, task1_2 is further divided to task2_1 with S3. If we insert the produce and consume directly into the loop, unmatched production and consumption will result.

```
for (...) {
S1  x = work(i)

S2  if (c1)
S3    y = x + i;
S4  ... = y;
}
```

```
for (i = 0; i < N; I++) {
S1  x = work(i)
    produce(stream_x, x)    //task1_1 end

    x = consume(stream_x)
S2  if (c1)                 //task1_2 start
      x = consume(?)        //task2_1 start
S3    y = x + i;
      produce(?, y)         //task2_1 end
    produce(stream_y, y)    //task1_2 end
    y = consume(stream_y)
S4  ... = y;                //task1_3 end
}
```

Figure 14. Normal form of code (left) and using streams (right).

The answer comes from following the synchronous hypothesis and slightly modifying the construction of the SSA form in presence of concurrent streaming tasks.

6.2 SSA representation

We are using the Static Single Assignment (SSA) form as an intermediate representation for the source code. A program in SSA form if every variable used in the program appears a single time in the left hand side of an assignment. We are using the SSA form to eliminate the output dependences in the code, and to disambiguate the flow of data across tasks over multiple producer configurations.

```
/* Normal form of the code. */
S1:  r1 = ...
S2:  if (condition)
S3:    r1 = ...
S4:  ... = r1
```

```
/* Code under SSA form. */
S1:  r1_1 = ...
S2:  if (condition)
S3:    r1_2 = ...
S4:  r1_3 = phi(r1_1, r1_2)
S5:  ... = r1_3
```

Figure 15. Normal form of code (left) and SSA form of the code (right).

Considering the example in Figure 15, if we partition the statements into (S1), (S2,S3), (S4), we need to implement precedence constraints for the output dependence between partition (S1) and (S2,S3), which decreases the degree of parallelism and induces synchronization overhead.

Eliminating the output dependences with the SSA form leads to the introduction of multiple streams in the partitioned code. In order to merge the information coming from different control flow branches, a Φ node is introduced in the SSA form. The Φ function is not normally implemented directly, after the optimizations are completed the SSA representation will be transformed back to ordinary one with additional copies inserted at incoming edges of (some) Φ functions. We need to handle the case where multiple producers in a given partition reach a single consumer in a different partition. When decoupling a dependence whose sink is a Φ node, the exact conditional control flow leading to the Φ node is not accessible for the out-of-SSA algorithm to generate ordinary code.

Task-closed* Φ *node In SSA loop optimization, there is a concept called loop-closed Φ node, which implements the additional property that no SSA name is used outside of loop where it is defined. When enforcing this property, Φ nodes must be inserted at the loop exit node to catch the variables that will be used outside of the loop. Here we give a similar definition for *task-closed* Φ node: if multiple SSA variables are defined in one partition and used in another, a phi node will be created at the end of the partition for this variable. This is the place where we join/split the stream. We need to make sure that different definitions of the variable will be merged in this partition before it continues to a downstream one. This node will be removed when converting back from SSA.

Task-closed stream Our partitioning algorithms generate nested pipelining code to guarantee that all communications follow the synchronous hypothesis. For each boundary, if there are one or more definitions of a variable coming through from different partitions, we insert a consumer at this boundary to merge the incoming data, and immediately insert a producer to forward the merged data at the rate of the downstream control flow.

1. When partitioning from a boundary, if inside the treegion, there are multiple definitions of a scalar and it will be used in other treegions which has the same conditional level, we create a Φ node at the end of this partition to merge all the definitions, and also update the SSA variable in later partitions.
2. If there is a Φ node at the end of a partition, insert a stream named with the left-hand side variable of the Φ node.
3. At the place where this variable is used, which is also a Φ node, add a special stream-Φ node to consume.
4. To generate code for the stream-Φ, use the boolean condition associated with the conditional phi node it originates from.

Let us consider the SSA-form example in Figure 15 where we partition the code into (S1,S2,S3) and (S4,S5). A Φ node will be inserted at the end of the first partition, $r1\_4 = phi(r1\_1, r1\_2)$, the Φ node in a later partition should be updated from $r1\_3 = \Phi(r1\_1, r1\_2)$ to $r1\_5 = \Phi(r1\_4)$. In the second step, we find out that in partition (S1,S2,S3), there is a Φ node at the end, so we insert a stream to produce there. And in partition (S4,S5), after the Φ node there is a use of the variable, so we insert a stream consume. The generated code will look like Figure 16.

```
/* Producer. */
S1: r1_1 = ...
S2: if (condition)
S3:   r1_2 = ...
    r1_4 = phi(r1_1, r1_2)
    produce(stream_r1_4, r1_4)
```

```
/* Consumer. */
S4: r1_5 = phi(r1_4)
    r1_5 = consume(stream_r1_4, i)
S5: ... = r1_5
```

Figure 16. Apply our algorithm to generate the parallel code. Producer thread (left) and consumer thread (right).

This example illustrates the generality of our method and shows how fine-grain pipelines can be built in presence of complex, multi-level control flow.

If we decide to partition the statements into (S1), (S2,S3), (S4,S5), which is the case for multiple producers, the generated code will look like in Figure 17.

```
/* Producer 1. */
S1: r1_1 = ...
    r1_2 = phi(r1_1)
    produce(stream_r1_2, r1_2)

/* Producer 2. */
S2: if (condition)
      r1_3 = ...
    r1_4 = phi(r1_3)
    produce(stream_r1_4, r1_4)
```

```
/* Consumer. */
S4: r1_5 = phi(r1_2, r1_4)
      if (condition)
        r1_5 = consume(stream_r1_4, i)
      else
        r1_5 = consume(stream_r1_2, i)
S5: ... = r1_5
      i++
```

Figure 17. Multiple producers with applied our algorithm, the generated code.

For multiple consumers, the stream extension of OpenMP will broadcast to its consumers, which is appropriate for our case.

7. Conclusion

In this paper, we propose a method to decouple independent tasks in serial programs, to extract scalable pipelining and data-parallelism. Our method leverages a recent proposition of a stream-processing extension of OpenMP, with a persistent task semantics to eliminate the overhead of scheduling task instances each time a pair of tasks need to communicate. Our method is inspired by the synchronous hypothesis: communicating concurrent tasks share the same control flow. This hypothesis simplifies the coordination of communicating tasks over nested levels of parallelism. Synchrony also facilitates the definition of generalized, structured typed fusion and partition algorithms preserving the loop structure information. These algorithms have been proven to be essential to the adaptation of the grain of parallelism to the target and to the effectiveness of compile-time load balancing. These partitioning algorithms also handle DOALL parallelization inside a task pipeline. We are using a combination of SSA, control dependence tree and (non-scalar) dependence graph as an IR. With the support of SSA, our method eliminates the nested multiple producer and multiple consumer problems of PS-DSWP. SSA also provides additional applicability, elegance and complexity benefits. This work is currently under development in a development branch of GCC, the partitioning algorithms is partially developed. For the code generation part, we first need to migrate the existing OpenMP expansion pass of GCC to work under SSA form, which has been a long-running challenge. When this work is complete, our method will leverage the array data-flow analysis of the Graphite polyhedral compilation pass of GCC to provide more precise data dependence information in loop nests with regular control flow.

Acknowledgments This work was partly funded by the European FP7 project TERAFLUX id. 249013, `http://www.teraflux.eu`

References

[1] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9:491–542, October 1987.

[2] W. Baxter and H. R. Bauer, III. The program dependence graph and vectorization. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 1–11, New York, NY, USA, 1989. ACM.

[3] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9:366–371, May 1966.

[4] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Intl. Conf. on Parallel Processing (ICPP)*, Saint Charles, IL, 1986.

[5] J. Ferrante and M. Mace. On linearizing parallel code. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '85, pages 179–190, New York, NY, USA, 1985. ACM.

[6] J. Ferrante, M. Mace, and B. Simons. Generating sequential code from parallel code. In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, pages 582–592, New York, NY, USA, 1988. ACM.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, July 1987.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

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

[10] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 407–416, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[11] K. Kennedy and K. S. Mckinley. Typed fusion with applications to parallel and sequential code generation. Technical report, Department of Computer Science Rice University, CITI, 1993.

[12] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[13] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. *Microarchitecture, IEEE/ACM International Symposium on*, 0:105–118, 2005.

[14] A. Pop and A. Cohen. A stream-comptuting extension to OpenMP. In *Proc. of the 4th Intl. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*, Jan. 2011.

[15] A. Pop, S. Pop, and J. Sjödin. Automatic streamization in GCC. In *GCC Developer's Summit*, Montreal, Quebec, June 2009.

[16] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.