Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs

Feng Li, Antoniu Pop, Albert Cohen

Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs

Feng Li
INRIA and École Normale Supérieure
feng.li@inria.fr

Antoniu Pop
INRIA and École Normale Supérieure
antoniu.pop@inria.fr

Albert Cohen
INRIA and École Normale Supérieure
albert.cohen@inria.fr

Abstract

This article presents a general algorithm for transforming sequential imperative programs into parallel data-flow programs. Our algorithm operates on a program dependence graph in SSA form, extracting task, pipeline, and data parallelism from arbitrary control flow, and coarsening its granularity using a generalized form of typed fusion. A GCC-based prototype is applied to the automatic parallelization of recursive C programs.

1. Introduction

The evolution of current processor architectures follows a path of exponential growth of the number of cores per chip, while little improvements for single-core performance are expected. Higher performance returns on newer architectures are therefore contingent on the amount of parallelism that can be efficiently exploited in applications. However, not all expressions of parallelism are compatible with a power- and bandwidth-efficient execution. The limited power envelope and off-chip memory bandwidth of many-core processors push for locality-friendly parallelization schemes. By mitigating the memory wall and reducing power consumption through a more efficient management of data communications and control, the data-flow model of computation provides an important direction for exploiting upcoming architectures.

The paper discusses a different way of looking at imperative programs—as data-flow threads. These data-flow threads can be extracted from a conventional static single assignment (SSA) representation. Starting from the extraction of fine-grained parallelism, grain-coarsening transformations reduce synchronization overhead while avoiding significant loss of parallelism. As an important contribution, we lift all restriction on data dependence and control flow patterns, allowing for irreducible control flow and recursive calls. Our compilation algorithm also emphasizes modularity (separate compilation) and integration with existing development practices and binary interfaces.

This paper is limited to the exploitation of scalar data dependences, expecting programmer intervention to expose producer-consumer relations from array and pointer code by means of explicit scalar dependences. Our approach is complementary to programming language efforts to express inter-task dependences as pragma annotations [19–21, 24]; it contributes to reducing the verbosity of such annotations.

2. Related Work

The principal motivation for research into dataflow models comes from the limitations of *von Neumann* machines to exploit massive amounts of fine grained parallelism. The early dataflow architectures proposed by Dennis and Misunas [8] or Davis [7] avoid the von Neumann bottlenecks by only relying on local memory and replacing the global program counter by a purely data-driven execution model, executing instructions as soon as their operands become available. More recent dataflow architectures rely on the same principles, albeit at a coarser grain, executing sequences of instructions, or dataflow threads, instead of single instructions.

While many dataflow programming languages have been proposed [11], our objective here is to automatically extract dataflow threads from imperative programs.

Compiling imperative programs to data-flow threads. The problem of compiling imperative programs for data-flow execution has been widely studied. Beck et al. [3] propose a method for translating control flow to data flow, and show that data-flow graphs can serve as an executable intermediate representation in parallelizing compilers. Ottenstein et al. [17] study such a translation using the Program Dependence Web, an intermediate representation based on gated-SSA [25] that can directly be interpreted in either control-, data-, or demand-driven models of execution. Programs are transformed to the MIT dataflow program graph [2], targetting the Monsoon architecture.

Najjar et al. evaluated multiple techniques for extracting thread-level data-flow [16]. These papers target a token-based, instruction-level data-flow model, analogous to the simulation of hardware circuits. In contrast, our data-flow model does not require tokens or associative maps, shifting the effort of expliciting the consumer threads to their producers to the compiler. The comparison between our approach and the token-based solution is further discussed in Section 3. In addition, thread-level data-flow requires additional efforts to coarsen the grain of concurrency, handling the dynamic creation of threads, and managing their activation records (data-flow frames).

SSA as an intermediate representation for data-flow compilation. The *static single assignment* form (SSA) is formally equivalent to a well-behaved subset of the *continuation-passing style* (CPS) [1, 13] model, which is used in compilers for functional languages such as Scheme, ML and Haskell. The data-flow model has been tied closely to functional languages, since the edges in a data-flow graph can be seen both as encoding dependence information as well as continuation in a parallel model of execution. The SSA representation builds a bridge between imperative languages and the data-flow execution model. Our algorithm uses the properties of the SSA to streamline the conversion of general control flow into thread-level data-flow.

Decoupled software pipelining. Closely related to our work, and in particular to our analysis framework, is the *decoupled software pipelining* (DSWP) technique [18]. It partitions loops into long-running threads that communicate via inter-core queues, following the execution model of Kahn process networks [12]. DSWP builds a Program Dependence Graph (PDG) [9], combining control and data dependences (scalar and memory). 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, it handles uncounted loops, complex control flow and irregular pointer-based memory accesses. Parallel-Stage Decoupled Software Pipelining (PS-DSWP) [23] is an extension to combine pipeline parallelism with some stages executed in a DOALL, data-parallel fashion.

These techniques have a few caveats however. They offer limited support for decoupling along backward control and data dependences. They provide a complex yet somewhat conservative code generation method to decouple dependences between source and target statements governed by different control flow.

Outline of the paper. The remainder of this paper is organized as follows. Section 3 describes the data-flow execution model. Section 4 presents our algorithm for the modular translation of imperative programs to coarse grain data-flow. Section 5 briefly describes our implementation. Section 6 presents experimental results, before we conclude in Section 7.

3. Thread-Level Data-Flow Execution Model

To express producer-consumer and control dependences, we define an abstract data-flow interface suitable for parallelization passes in compilers as well as expert programmers developing low-level data-flow code. This interface is designed after the DTA data-driven execution model and the T* ISA [10, 22].

The interface defines two main components: *data-flow threads*, or simply *threads* when clear from the context, together with their associated *data-flow frames*, or simply *frames*.

The frame of a data-flow thread stores its input values, and optionally some local variables or thread metadata. The frame's address also serves as an identifier for the thread itself, to synchronize producers with consumers. Communications between threads are single-sided: the producer thread knows the address of the data-flow frames of its dependent, consumer threads. A thread writes its output data directly into the data-flow frames of its consumers.

Each thread is associated with a Synchronization Counter (SC) to track the satisfaction of producer-consumer dependences: upon termination of a thread, the SC of its dependent threads is decremented. A thread may execute as soon as its SC reaches 0, which may be determined immediately when the producer decrements the SC. The initial value of the SC is derived from the dependence graph: it is equal to the number of arguments of the thread, each one corresponding to an externally defined use.

In contrast, token-based approaches require checking the presence of the necessary tokens on incoming edges. This means that either (1) a scanner must periodically check the schedulability of data flow threads, or (2) data flow threads are suspendable. The former poses performance and scalability issues, while the latter requires execution under complex stack systems (e.g., cactus stacks) that may introduce artificial constraints on the schedule. The SC aggregate the information on the present and missing tokens for a thread's execution, allowing producer threads to decide when a given consumer is fireable.

We introduce four functions to manage threads and frames. They are implemented as compiler builtins, recognized as primitive operations of the compiler's intermediate representation. They can be implemented efficiently in software or hardware [10, 22].

`void *tcreate(void (*func)(), int sc, int size);`

Creates a new data-flow thread and allocates its associated frame. `func` is a pointer to the argument-less function to be executed by the data-flow thread, `sc` is the initial value of the thread's synchronization counter, and `size` is the size of the data-flow frame to be allocated. It returns a pointer to the allocated data-flow frame.

`void tdecrease(void *fp);`

Marks the thread designated by frame pointer `fp` to be decremented upon termination of the current thread.

`void tend();`

Terminates the current thread and deallocates its frame.

`void *tget_cfp();`

Returns the frame pointer of the current thread.

4. Conversion to Thread-Level Data-Flow

The general approach for transforming sequential imperative programs into parallel data-flow programs extracts the finest grain of thread-level parallelism, splitting basic blocks at the statement level, which is then coarsened through typed fusion to reduce communication overhead. Strongly Connected Components (SCC) of the program dependence graph, where no parallelism can be exploited, are also coalesced.

Our algorithm operates on a low-level program representation in SSA form [5, 6] form, common in modern production compilers like GCC and LLVM. Our algorithm is implemented as a new parallelization pass of GCC's middle-end.

We only consider scalar data dependences and control dependences. As stated in the introduction, arrays and pointers are currently ignored from the dependence analysis. The correctness of program transformations requires programmers to expose dependences as scalar dependences in the source program.

4.1 Algorithm for Generating Data Flow Threads

We first build the Program Dependence Graph (PDG) [9] under SSA form (SSA-PDG) from the serial program, then coarsen the granularity by merging SCCs in the graph and applying typed fusion. To align the flow of values and data-flow frames with control dependences, we define the Data-Flow Program Dependence Graph (DF-PDG), built from the SSA-PDG. The DF-PDG allows to generate target data-flow code. The typed fusion and DF-PDG conversion algorithms are the main contributions of the paper.

We illustrate our algorithm on the example in Figure 1 (left), where we assume that all functions are pure (no side effects, no state). In the loop body, `bar(i)` is evaluated at each iteration, but only the last computed value will be used outside of the loop, along with the last value of `i`.

4.1.1 Loop Unswitching

The SSA form uses a unique name for each assignment to a variable. In this way, each use of a variable has a unique reaching definition. A merging Φ node is introduced in the SSA form at points where multiple control flow paths converge and a given variable is defined on more than one path. For the example on the left side of Figure 1, the Φ nodes for variables `i`, `a` and `b` will be placed before S9, in the loop header, as shown in Figure 2 (left).

Some of these Φ nodes in the header carry redundant data flow: the Φ node for variable `i` defines a value used inside the loop body, while the Φ nodes for `a` and `b` only define values used outside of the loop. To differentiate the type of Φ node, we apply loop unswitching so that the inductive Φ node for variable `i` will remain at the header while the Φ nodes capturing the last values of `a` and `b` will be placed at loop exit, before their respective uses, as in Figure 2 (right).

```
S1  a = 0;
S2  i = 0;
S3  b = 0;

S9  while (i < 100) {
S6    a = i;
S7    b = bar (i);
S8    i = next (i);
    }

S12 if (a > b)
S13   ret = a;
    else
S14   ret = b;
S16 return ret;
```

```
S1  a = 0;
S2  i = 0;
S3  b = 0;
S4  if (i < 100) {
      do {
S6      a = i;
S7      b = bar (i);
S8      i = next (i);
S9    } while (i < 100);
    }
S12 if (a > b)
S13   ret = a;
    else
S14   ret = b;
S16 return ret;
```

Figure 1. Running example (left) and after unswitching (right).

4.1.2 Build Program Dependence Graph under SSA

There are two main reasons for relying on a SSA-based representation of the PDG:

```
   # i1 = Φ (i0, i2);
   # a1 = Φ (a0, a2);
   # b1 = Φ (b0, b2);
S9 while (i1 < 100) {
S6   a2 = i1;
S7   b2 = bar (i1);
S8   i2 = next (i1);
   }
   /*use of a1, b1. */
```

```
if (i1 < 100) {
  do {
    # i1 = Φ (i0, i2);
S6    a2 = i1;
S7    b2 = bar (i1);
S8    i2 = next (i1);
S9  } while (i1 < 100);
  }
  # a1 = Φ (a0, a2);
  # b1 = Φ (b0, b2);
  /* use of a1, b1. */
```

Figure 2. SSA form before unswitching (left) and after (right).

1. making the reaching definitions unique for each use, effectively converting the scalar flow into a functional program;
2. reducing the complexity from $O(N^2)$ to $O(N)$, as the number of def-use edges can become very large, sometimes quadratic in the number of nodes [14].

In SSA form, multiple reaching definitions for a use are factored through Φ nodes, which ensures that the number of def-use chains is bounded by the number of chains in the control flow graph.

Each node in the SSA-PDG represents a statement in SSA form and edges in the graph represent either control or data dependences. Control dependences can form several weakly connected graphs. We add control dependences from the entry node of the function to the root node of each weakly connected graph. The root node of each weakly connected graph is defined as the first node in the graph reached by the control flow.

Figure 3 shows the SSA representation, after loop unswitching, for the code on Figure 1 (right), and the corresponding SSA-PDG is presented on Figure 4, where dashed edges represent control dependences and solid edges represent data dependences. We add control dependences from the function entry point `foo` to `S2, S4, S1, S3, S10, S11, S12, S15, S16`.

```
int foo () {
  S1  a0 = 0;
  S2  i0 = 0;
  S3  b0 = 0;
  S4  if (i0 <= 99)
        goto S5;
      else
        goto S10;

  S5  # i1 = Φ(i0, i2);
  S6  a1 = i1;
  S7  b1 = bar (i1);
  S8  i2 = next (i1);
  S9  if (i2 <= 99)
        goto S5;
      else
        goto S10;

 S10 # a2 = Φ(a0, a1)
 S11 # b2 = Φ(b0, b1)
 S12 if (a2 > b2)
       goto S13;
     else
       goto S14;

 S13 ret0 = a2;
     goto S15;

 S14 ret1 = b2;

 S15 # ret2 = Φ(ret0, ret1)
 S16 return ret;
}
```

Figure 3. SSA representation after loop unswitching.

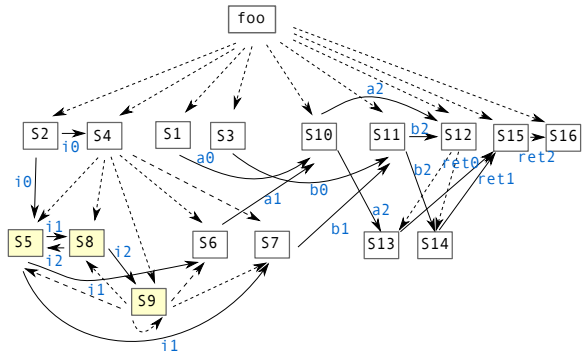

Figure 4. SSA-PDG for the code example in Figure 3.

4.1.3 Merging Strongly Connected Components

In the SSA-PDG, SCCs present no parallelization opportunities. Their execution is sequential, so generating data flow threads would mostly incur parallelization overhead.[1] We coalesce SCCs, as shown on Figure 5 where the new node `SCC1` replaces nodes `S5`, `S8` and `S9` from Figure 4. This SCC corresponds to the induction on variable `i` in the loop.

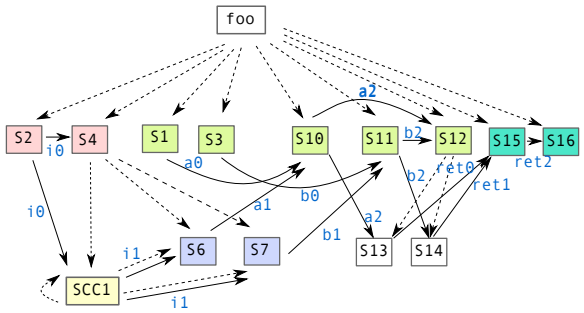

Figure 5. SSA PDG after merging the SCC.

4.1.4 Typed Fusion

Before partitioning, we coarsen the granularity of each data-flow thread using a generalized form of the typed fusion algorithm of McKinley and Kennedy [15]. In SSA-PDG, we assign a *type* to each node according to its control dependences: all nodes sharing identical control dependences are assigned the same type. Nodes of the same type are candidates for typed fusion. On Figure 5, the nodes `S6` and `S7` have the same type, as both are control dependent on `S4`, and can potentially be fused. Similarly, nodes `S2, S4, S1, S3, S10, S11, S12, S15` and `S16` could be fused, but this would lead to adverse side effects, in particular reducing parallelism by creating artificial SCCs. In our example, such a fused node would lead to SCC with nodes `S13` and `S14` because of the dependence chains `S10-S13-S15` and `S11-S14-S15`.

For this reason, we limit the fusion algorithm to avoid introducing new SCCs or increasing the size of existing SCCs, which are on the critical path of the program's execution.

Our approach for typed fusion follows a simple greedy algorithm, which starts from a random node in each typed set and adds new nodes of the same type to the fusion set as long as: (1) the new additions do not lead to creating a new SCC in the SSA-PDG after fusion; and (2) the fusion set does not contain SCCs itself. As this algorithm is applied after fusing the existing SCCs, the latter condition simply means that nodes that have self-dependences are not considered candidates even if their type matches.

Figures 5 and 6 (A) present one possible outcome of typed fusion applied to the running example. There are five different types in the graph on Figure 5: `(foo)`, `(S2, S4, S1, S3, S10, S11, S12, S15, S16)`, `(SCC1, S6, S7)`, `(S13)` and `(S14)`. Note that `S13` and `S14` have distinct types because their control dependences correspond to different truth values for `S12`.

For the typed set `(SCC1, S6, S7)`, `SCC1` cannot be fused with any other node as it would increase the size of an existing SCC (restriction (2)). This only leaves `S6` and `S7`, which can be fused and yield the fused node `F2` on Figure 6 (A). For the typed set `(S2, S4, S1, S3, S10, S11, S12, S15, S16)`, there are many possible outcomes, depending on the traversal order of typed sets. In this case and as we discussed above, restriction (1) does not allow, for example, nodes `S4` and `S10` to be in the same fusion set because of the dependence chain `S4-S6-S10`. The algorithm will always lead to at least three fusion sets for this type due to the long dependence chain `S4-S6-S10-S13-S15`.

This technique coarsens the granularity of data flow threads, without inserting redundant computations and without increasing

[1] Some latency-hiding benefits exist, but we concentrate on parallelism extraction in this paper.

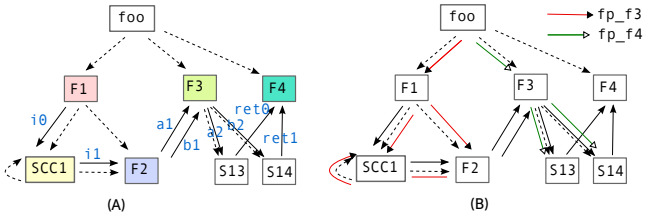

Figure 6. (A) SSA-PDG after typed fusion. (B) Data Flow Program Dependence Graph.

the number of instructions belonging to SCCs (which usually happens when relying on basic blocks to partition the computation).

4.1.5 Data Flow Program Dependence Graph

As data flow threads communicate by writing directly in the data flow frame of their consumers, it is necessary that, along all data dependence edges of the SSA-PDG, the producer nodes know the data flow frame of the consumer nodes. We transform the SSA-PDG graph to reflect the communication required to this effect.

Thread creation point Thread creation occurs, conditionally, along each control dependence edge of the SSA-PDG. The thread creation points for a given node are its predecessor nodes in the control dependence graph. On Figure 6 (B), where dashed lines represent control dependences, the nodes `foo, F1, F3` and `SCC1` are thread creation points as they have outgoing control dependence edges. Control dependences can be conditional, like in the case of `F3` which creates either `S13` or `S14` depending on the conditional statement `S12`, or unconditional in the case of the function entry point `foo`.

At a thread creation point, the data-flow frame for the newly created thread is known and it needs to be passed to all threads producing data for this new thread.

Passing data-flow frame information The DF-PDG for our running example is shown on Figure 6 (B). We add data dependences to the SSA-PDG for passing the data-flow frame information of consumer threads to producer threads. The edges with white triangular arrow communicate the data-flow frame of `F4` (consumer) to the data-flow threads `S13, S14` (producers). The values of `ret0` and `ret1`, produced in `S13` and `S14` are consumed by `F4`. As the data is stored directly in the frame of the consumer, the producer must get a pointer to the frame of `F4`. The thread creation point for `F4` is the function entry point, `foo`, and it needs to forward this frame pointer to all producers, which involves thread `F3` in our example.

We rely on the following algorithm to pass the data flow frame pointers of consumers to producers. For each data dependence from a node `A` to a node `B` in the SSA-PDG, we visit the predecessors of each node along control dependences until we reach a common predecessor `P`.

- If `P` is an immediate predecessor of the consumer node `B`, then `P` is the thread creation point for `B` and therefore knows the data-flow frame of `B`. We add data dependences for the frame pointer of `B` along all control dependence paths in the graph to `A`.
- If `P` is not an immediate predecessor of `B`, we need to split the data dependence as the frame of the consumer cannot be known due to diverging control flow paths, as illustrated on Figure 7. We remove the original data dependence from `A` to `B` and we add data dependences, for the same variable, from `A` to all successors `D` of `P` in the SSA-PDG such that there is a control dependence path from `D` to `B`. We further add data dependences for the data flow pointer of `D` from `P` to `A` and also for the data itself, from `D` to `B`.

This second case is illustrated on Figure 7, where data dependences from `S1` to `S2` and `S3` need to be split. The common predecessor is the function entry node `foo`, which is not an immediate predecessor of either `S2` or `S3`. The successor of `foo` that is a predecessor of `S2` and `S3` is `C1`, which is used to forward the data produced by `S1`. The frame pointer of `C1` is sent to `S1` from its thread creation point.

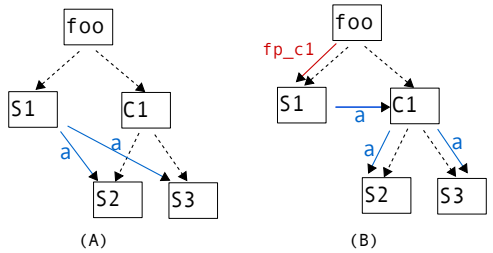

Figure 7. Splitting data dependences: (A) the original SSA-PDG and (B) the generated DF-PDG.

Strongly connected components in the DF-PDG The algorithm for building the DF-PDG presented above introduces additional data dependences that can lead to new SCCs in the graph. These new constraints need to be enforced, which serializes the execution. For this reason, we perform one additional pass of fusion of SCCs once the DF-PDG is constructed.

The example on Figure 8 (left) and corresponding SSA-PDG (center) illustrate this issue. There are two data dependences: `S1` to `S2` and the loop-carried dependence `S3` to `S1`. For the latter, the DF-PDG construction algorithm explores every control dependence paths linking `S3` and `S1` from a common predecessor, namely `C0`→`S3` and `C0`→`C1`→`S1`. As `S1` is only reached through `C1`, the data dependence is split and the data forwarded to `S1` through `C1`, as shown by the extra data dependence edges on Figure 8 (right). There are four different combinations for each data dependence and we only show one on Figure 8, yet it already results in a SCC involving nodes `C1` and `S3`.

```
C0   if (c) goto S1;
     else goto D1;
S1   a2 = Φ (a0, a1);
S2   ... = a2;
S3   a1 = ...;
C1   if (c) goto S1
     else goto D1
D1   ...
```

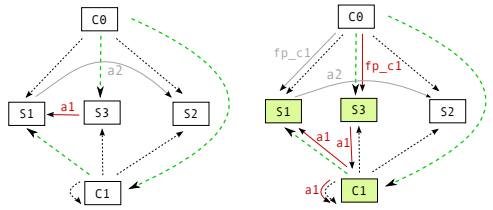

Figure 8. SSA representation for a simple loop carried dependence (left) and the corresponding SSA-PDG (center) and a partial DF-PDG (right).

4.2 Modular Code Generation

We put a strong emphasis on generality and flexible integration of data-flow compilation tools in a state-of-the-art development process. Modularity has not been considered a first-class objective in previous thread-level data-flow algorithms. We show that modular code generation is possible, provided that each processor core (or instruction fetch unit) is associated with a private user-level stack. The stack only needs to be accessed by this particular core. Data-flow threads themselves do not need any internal stack; they are non-suspendable and run sequentially w.r.t. other data-flow threads scheduled on the same core. Any data-flow thread scheduled on a given core is free to use the core's private stack. This stack streamlines the implementation of classical (blocking) function calls. It

may also be used to spill registers within a thread, although data-flow frames may accommodate free space for this purpose.

For modular compilation purposes, externally visible functions in a compilation unit should be cloned, to preserve the original control flow interface, while the clone is compiled into data-flow threads. The original function can be used when calling the function from outside of a parallel data-flow region or to avoid saturating the system with threads. The clone must be exported among the module's symbols to seamlessly compose threaded code over separately compiled modules.

Within a parallel region, all functions are called asynchronously. Internal functions, within the scope of the compilation unit, are directly compiled into data-flow threads. External functions, linked from separately compiled modules, and builtin functions from the compiler are wrapped into a data-flow thread in which they are called synchronously.

Every threaded clone of a function is split into three stages: the entry thread, multiple compute threads, and the return thread.

- The entry thread implements the entry block of the control flow graph, creating all the threads for the blocks that are unconditionally executed upon entering the function.
- The compute threads are systematically created by the immediate predecessors in the control dependence graph, as described in the previous section. Thread creation is conditional on the predicate at the source of the control dependence. Input arguments and pointers to the frames of the dependent threads are handled according to the DF-PDG.
- The return thread propagates the return value to the continuation threads at the call site of the function.

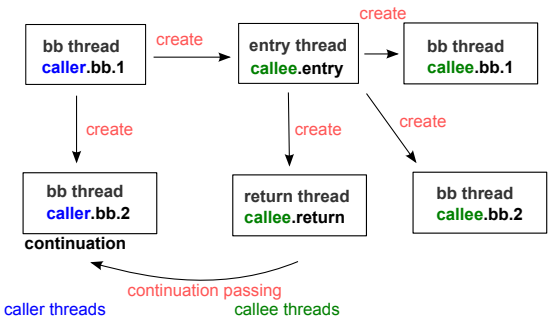

Figure 9. Caller and callee, threaded version.

Figure 9 illustrates the calling convention. Calling a data-flow function creates the entry thread (`callee.entry`) of the callee and the caller's continuation thread (`caller.bb.2`); the latter will wait for the value of the callee's return thread (`callee.return`).

5. Implementation of the Data-Flow Interface

We target data-flow execution on a shared-memory multiprocessor with hardware coherence.

Our prototype has been developed within GCC 4.7.0 20110426. It replaces the `libgomp` OpenMP runtime of GCC. The scheduler takes data-flow threads whose SC reached 0 and moves them to the ready queue.

We currently assume there are enough ready threads to occupy the processor cores and hide latency. This hypothesis eliminates the need for a waiting queue collecting threads whose SC has not reach 0, since we do not need to start prefetching data or code for these threads. Revisiting this hypothesis may be necessary when studying applications with limited parallelism degree where scheduling and memory latencies are harder to hide.

Data-flow frames are allocated from a dedicated memory pool. This pool internally uses slab allocation to accelerate the allocation and deallocation of frames of predefined sizes. The frame structure itself is laid out as follows:

- a thread template pointer referring to invariant meta-data shared by all thread instances of this template, including the function (code) pointer and the size of the frame;
- the thread's synchronization counter (SC);
- pointers to frames of data- and control-dependent threads;
- the thread's arguments.

The last two items correspond to the frame structure exposed in the abstract data-flow interface and generated by the compiler.

6. Experimental Validation

We validate our approach on two universal examples of tree recursion, Fibonacci and merge sort. The objectives are:

1. checking the method on diverse data and control flow, including loops over arrays, divide and conquer recursion, and data-dependent conditions;
2. Fibonacci exhibits the finest-grain threads possible, which gives a precise reference on the break even point and scalability for thread-level data flow compared to fine-grain data flow;
3. merge sort is more realistic and allows to illustrate typed fusion for grain coarsening.

We target an Intel Core i7-2720QM 4-core laptop (Sandy Bridge chip) and an AMD Opteron 6164 HE 24-core blade server (two Magny Cours chips). Both benchmarks are recursive, sequential C programs, and automatically parallelized.[2]

To assess the effect of thread granularity, we set a threshold for parallel recursive calls. Below this threshold, the serial version is executed. This programmer-controlled granularity complements the effects of the automatic typed fusion algorithm. Modular compilation allows the serial version to be called seamlessly as an external function.

As an illustration, in the Fibonacci implementation below, `fib.threaded` will be transformed to data-flow threads, and `fib.serial` will not since it is declared as an external function.

```
extern int fib.serial (int);

int fib.threaded (int n) {
  if (n < THRESH)
    return fib.serial (n);
  else
    return fib.threaded (n-1) + fib.threaded (n-2);
}
```

Figure 10 reports the performance of merge sort on $200,000$ random integers between 0 and $10,000$. The compiler automatically partitions the function into data-flow threads, then converts the data and control dependences into the proper frame operations. The algorithms not only parallelize the recursive division of the array, but also the merge operation. The latter is a good candidate for typed fusion: the array comparisons and assignments are dominated by the same loop header and can be fused into a coarser-grain thread.

The grain threshold ranges from 2^0 to 2^{18} ($262,144$, effectively sequentializing the execution). The figures show the speedup as a function of the granularity of the parallel threads. As the grain threshold increases, speedup gained from parallel execution on multiple cores exceeds the overhead of thread creation. The generated code breaks even at the threshold of 2^4. This low break-even threshold is a benefit of the applicability of typed fusion on the merge operation. As a divide and conquer algorithm, the problem size is reduced in each division, the array eventually fitting into the cache; it reaches a maximal speedup of 2.82.

[2] The array dependences in merge sort are covered by scalar dependences on the indexes, and can safely be ignored.

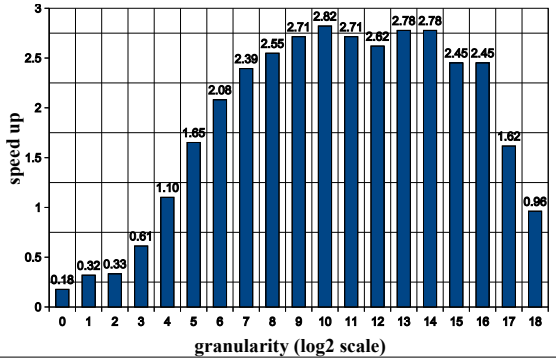

Figure 10. Merge Sort running on 4 cores.

Figure 11 shows the performance results for `fib(42)`. Fibonacci is an extreme case where typed fusion is ineffective, since no pairs of instructions share the same control dependence. The generated code breaks even when setting the threshold at `fib(15)`, where 2^{27} threads are created. But as granularity increases, the overhead of thread synchronization decreases. Our results on 24 cores reach a speedup of 11.86, which validates our algorithm's ability to exploit parallelism effectively.

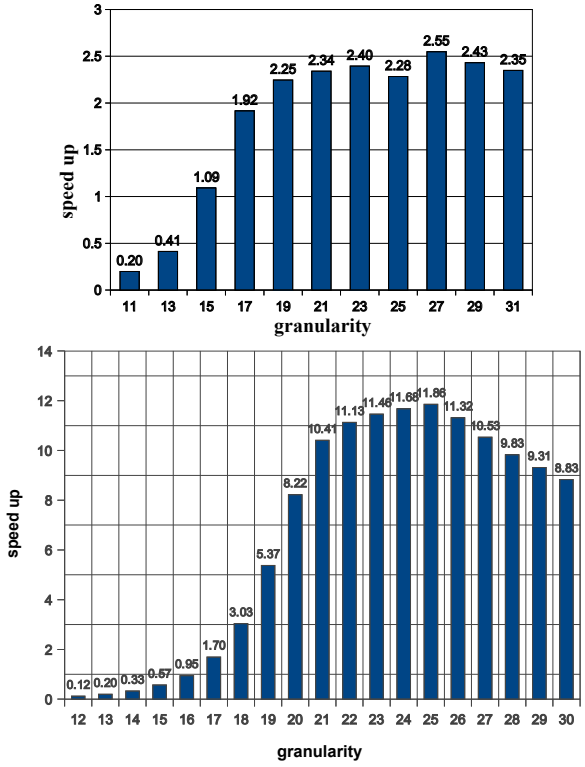

Figure 11. Computing the 42$^{\text{th}}$ Fibonacci number on 4 cores (above) and 24 cores (below).

7. Conclusion and Perspectives

We presented an automatic parallelization algorithm to compile arbitrary imperative control flow to a thread-level data-flow model. The algorithm operates on an SSA form PDG, extracting task, pipeline and data parallelism, then applying a generalized form of typed fusion to coarsen the synchronization grain, and finally expressing the communications in a suitable way for tokenless threaded data-flow execution. Our prototype is implemented in a production compiler; it currently supports scalar dependences only.

Of course, the algorithm and implementation should be extended to handle pointer and array dependences. Language support will be needed in general. We see our automatic parallelization algorithm as a means to reduce the verbosity of dependence annotations in a task-parallel language.

Acknowledgments This work was partly funded by the European FP7 project TERAFLUX id. 249013.

References

[1] A. W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33:17–20, April 1998.

[2] K. Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Computers*, 39:300–318, March 1990.

[3] M. Beck, R. Johnson, and K. Pingali. From control flow to dataflow. Technical report, Cornell University, 1989.

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

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13:451–490, October 1991.

[6] R. Cytron, J. Ferrante, and V. Sarkar. Experiences using control dependence in PTRAN. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 186–212, London, UK, UK, 1990. Pitman Publishing.

[7] A. L. Davis. The architecture and system method of DDM1: A recursively structured data driven machine. In *Proceedings of the 5th annual symposium on Computer architecture*, ISCA '78, pages 210–215, New York, NY, USA, 1978. ACM.

[8] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd annual symposium on Computer architecture*, ISCA '75, pages 126–132, New York, NY, USA, 1975. ACM.

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

[10] R. Giorgi, Z. Popovic, and N. Puzovic. DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems. In *SBAC-PAD*, pages 263–270, 2007.

[11] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36:1–34, March 2004.

[12] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug. 1974. North Holland, Amsterdam.

[13] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, IR '95, pages 13–22, New York, NY, USA, 1995. ACM.

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

[15] K. Kennedy and K. S. Mckinley. Typed fusion with applications to parallel and sequential code generation. Technical report, Rice University, 1993.

[16] W. Najjar, L. Roh, and A. Wim Böhm. An evaluation of medium-grain dataflow code. *Intl. J. of Parallel Programming*, 22:209–242, 1994. 10.1007/BF02577733.

[17] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. of the ACM SIGPLAN 1990 Conf. on Programming Language Design and Implementation*, PLDI '90, pages 257–271, New York, NY, USA, 1990. ACM.

[18] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *IEEE/ACM Intl. Symp. on Microarchitecture*, volume 0, pages 105–118, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[19] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *Intl. J. on High Performance Computing Architecture*, 23(3):284–299, 2009.

[20] 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.

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

[22] A. Portero, Z. Yu, and R. Giorgi. T-Star (T*): An x86-64 ISA extension to support thread execution on many cores. In *HiPEAC ACACES-2011*, pages 277–280, Fiuggi, Italy, July 2011.

[23] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proc. of the 6th annual IEEE/ACM Intl. Symp. on Code Generation and Optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.

[24] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso. TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems. In *Intl. Conf. on Parallel Processing (ICPP'08)*, pages 25–34, Portland, Oregon, Sept. 2008.

[25] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proc. of the 9th Intl. Conf. on Supercomputing*, ICS '95, pages 414–423, New York, NY, USA, 1995. ACM.