SEVENTH FRAMEWORK PROGRAMME
THEME
FET proactive 1: Concurrent Tera-Device Computing (ICT-2009.8.1)

PROJECT NUMBER: 249013

TERAFLUX

Exploiting dataflow parallelism in Teradevice Computing

D4.3 – First version of the compilation tools targeted to the TERAFLUX architecture

Due date of deliverable: 31/12/2010
Actual Submission: 31/12/2010

Start date of the project: January 1st, 2010 Duration: 48 months

Lead contractor for the deliverable: INRIA

Revision: See file name in document footer.

Project co-founded by the European Commission within the SEVENTH FRAMEWORK PROGRAMME (2007-2013)	
Dissemination Level: PU	
PU	Public
PP	Restricted to other programs participant (including the Commission Services)
RE	Restricted to a group specified by the consortium (including the Commission Services)
CO	Confidential, only for members of the consortium (including the Commission Services)

Change Control

Version#	Author	Organization	Change History
1.0	Albert Cohen	INRIA	
2.0	Albert Cohen	INRIA	Major revision
2.1	Albert Cohen	INRIA	Feedback and comments

Release Approval

Name	Role	Date
Albert Cohen	Originator	
Albert Cohen	WP Leader	
Roberto Giorgi	Project Coordinator for formal deliverable	31.12.2010

TABLE OF CONTENTS

The following list of authors will be updated to reflect the list of contributors to the writing of the document.

Albert Cohen, Feng Li

INRIA

Rosa Badia

BSC

Printed in Siena, Italy, Europe.

Part number: *please refer to the File name in the document footer*.

Deliverable number: D4.3 – **Dissemination Level: PU**

Deliverable name: **First version of the compilation tools targeted to the TERAFLUX architecture**

1. Glossary

Continuation.

In this deliverable, a continuation may either denote a function that has been outlined from the control flow region of a parent or caller function, and whose arguments include the live-in variables of the associated control flow region. Code generation for the TERAFLUX ISA builds explicit continuations when sequential control flow is decoupled for parallel, data-flow execution. Continuations can be built from the decoupling of both intraprocedural (conditionals) and interprocedural control flow (call and return points).

2. Executive Summary

This deliverable describes the form of the low level code that the TERAFLUX back-end compiler will aim to generate. The back-end compiler is currently being designed and implemented as a port of GCC. We report on two complementary code generation schemes, one based on the direct expansion of the efficiency layer, pragma-based language, and the second one relying on the automatic conversion of implicit parallelism into explicit parallelism, implementing the PS-DSWP technique described in D4.1. At the front-end level, we report on the design and implementation of a unified compiler infrastructure supporting the StarSs family of languages. This infrastructure is based on the source-to-source Mercurium compiler. Both compilers will communicate through the unified representation described in D4.2.

3.Introduction

This deliverable describes the compilation techniques and tools targeting the TERAFLUX architecture, and supporting the efficiency programming layer designed in WP3. It describes the first adaptations of the TERAFLUX compilation tools (planned and implemented) to generate code for the TERAFLUX architecture. In this deliverable, we focus on the ISA extensions for dataflow computing and on the lowering of the annotations of the efficiency programming layer to an executable abstraction of the machine.

3.1. Document structure

Section 4 describes the code generation scenarios targeting the TERAFLUX ISA extensions for dataflow threads. Section 5 describes the compilation infrastructure for the efficiency layer pragmas of the StarSs programming language.

3.2. Relation to other deliverables

This deliverable builds on the TERAFLUX Instruction Set Architecture (ISA) and deployment specifications defined in D7.1 and D7.2. It is closely related to the compilation methods described in D4.1.

3.3. Activities referred by this deliverable

This deliverable is associated with and concludes Task 4.1. Future work on code generation for the TERAFLUX architecture will be performed in the context of Task 4.2.

4. Code Generation for the TERAFLUX ISA

Due to some delay in the hiring and some logistic difficulty of integrating the work of three partners (INRIA, UCY, UNISI), the design and implementation of a compiler back-end dedicated to TERAFLUX has not finished in year 1 and needs to be completed in the second year of the project, in the context of Task 4.2. We believe this will not have consequences or other delays in the planning of the activities of the second year.

This deliverable describes the form of the low-level code targeted by the TERAFLUX back-end compiler, setting the objectives and general design choices for the implementation of this compiler. We report on two complementary code generation schemes that we decided to study and to support in the project. The first one relies on the direct expansion of the efficiency layer, pragma-based language. The second one complements this expansion with the automatic conversion of implicit parallelism into explicit parallelism, implementing the PS-DSWP technique described in D4.1. The second scheme is capable of exposing finer grain parallelism, converting all the control flow to dataflow threads, but it involves a more complex compilation infrastructure and will take more time to mature and to become applicable to full TERAFLUX applications. Both schemes are being implemented in GCC and leverage the static analysis and program transformation passes described in D4.1.

To illustrate the code generation schemes for the TERAFLUX ISA, we selected a small example representative of the problems arising when compiling complex (recursive) control flow to a dataflow architecture. This small example is the tree-recursive Fibonacci algorithm, in a C implementation called `fib` in the following.

We will show the original program, different annotated programs with OpenMP streaming extensions, and different versions of the generated code for the two code generation schemes we propose.

4.1. Methodology and definitions:

- We use a C syntax with compiler builtins; these builtins will ultimately be expanded into assembly instructions by the backend compiler.
- `void *TCreateDynamic(pred, function, counter, frame_size)` is a conditional, dynamic thread creation builtin; it (unconditionally) returns a fresh frame pointer to a frame structure of size `frame_size`. Note that a fresh frame is allocated and returned even if `pred` is false.
- `void TEnd()` terminates the current thread, publishing all data produced and written to consumer frames, decrementing the synchronization counters of these consumers accordingly, and marking the resources of the terminated thread for recycling.
- `void TUpdate(pred, frame_pointer)` marks the synchronization counter of the thread associated with `frame_pointer` to be decremented, if `pred` is true. The actual decrement and publication of the produced data does *not* happen until `TEnd()` is reached, terminating the execution of the current thread. This way, atomicity of the updates and productions is maintained even in the case of roll-backs. This is essential for future extensions with transactions in dataflow threads and for fault tolerance.
  Note that unlike the DTA architecture from which this syntax and semantics is partly inspired, we do *not* assume that stores to dataflow frames trigger an implicit update (decrement) of the synchronization counter. This is the main difference from the current ISA specification in D7.1 and D7.2, and it is intended to make the code generation more systematic.
  If possible, we will consider removing this explicit, distinct `TUpdate()` builtin and rely on auto-update store semantics in the future.
- void `TWait()` waits for the completion of all threads spawned by the current dataflow thread, and for the completion of threads transitively spawned by those. When the Thread Scheduling Unit (TSU – see D6.1) manages the threads, there will not be a need for this synchronization mechanism.

4.2. Original Program

```
int fib (int n) {
  if (n <= 1)
    return n;
  else {
    int s1 = fib(n-1);
    int s2 = fib(n-2);
    int s = s1 + s2;
    return s;
  }
}

void main () {
  printf("fib(7) = %d", fib(7));

  exit(0);
}
```

The original program computes the seventh Fibonacci number in a tree-recursion.

4.3. First parallel version

In the first scenario, the program is manually parallelized with the pragma-based efficiency programming model. We first consider a manually parallelized version where the recursive calls are executed asynchronously from the sum. We use the syntax and semantics of the streaming dataflow OpenMP extension described in D3.2.

```
int fib (int n) {
  #pragma omp parallel single
  {
    if (n <= 1)
      return n;
    else {
      #pragma omp task firstprivate(n) output(s1)
      {
        s1 = fib(n-1);
      }
      #pragma omp task firstprivate(n) output(s2)
      {
        s2 = fib(n-2);
      }
      #pragma omp task input(s1, s2) lastprivate(s)
      {
        s = s1 + s2;
      }
      // Wait for all tasks spawned in this parallel region
      #pragma taskwait
```

```
      return s;
    } // End else
  } // End parallel single
}

void main () {
  printf("fib(7) = %d", fib(7));

  exit(0);
}
```

This version uses nested parallel regions, with a lastprivate clause (one of our proposed extensions to OpenMP tasks) and a taskwait barrier before the return statement. This is probably the most incremental way to parallelize the code for a beginner, but it does not expose much pipeline parallelism. We will explore the code generation to dataflow threads nonetheless, before switching to a more aggressively parallelized version.

4.4. Direct lowering of the first parallel version

Code generation takes five main steps.

- Convert tasks into outlined functions; these functions will become the work units of the generated dataflow frames.
- Tag these functions with the proper thread type, as defined in D7.1 (DF1, DF1b, DF2, S, L).
- Declare frame data types for each outlined function, to capture the inputs of the associated dataflow frame.
- Capture the dependences between threads from the task input and output clauses, and convert them into continuation fields in the frame data types.
- Replace each task pragmas with a dataflow thread creation, with the associated allocation and initialization of its frame.
- Compile barriers (taskwait) into TWait(). This violates the basic principle that all synchronizations in a pure dataflow program should come from the availability of input data. But it makes the compilation of the efficiency layer's pragmas much more systematic. Note that TWait() is systematically eliminated by the second, finer grain compilation scheme that will be presented in a later section.

Here is the resulting code for the fib example.

```
#define CFP TGetCurrentFramePointer();

struct frame_fib1 {
  int n; // firstprivate(n)
  int *s1; // output(s1)
  void *cont;
};
```

```
struct frame_fib2 {
  int n; // firstprivate(n)
  int *s2; // output(s2)
  void *cont;
};

struct frame_fib_sum {
  int s1; // input(s1)
  int s2; // input(s2)
  int *s; // lastprivate(s)
  // fib_sum has no continuation/consumer
};

L_THREAD void main() {
  // main() is always a legacy POSIX thread, not a dataflow thread.

  printf("fib(7) = %d", fib(7));

  exit(0);
}

int fib (int n) {
  // fib is a plain function obeying procedural control flow.
  if (n <= 1)
    return n;
  else {
    int s;

    // Create threads in reverse topological order of the
    //   dataflow graph

    // Thread to compute the sum of fib(n-1) and fib(n-2).
    // Stalled until it gets the data on both sides
    //   (last argument is 2).
    //#pragma omp task input(n1, n2) lastprivate(n)
    struct frame_fib_sum* fp_fib_sum =
      TCreateDynamic(true, fib_sum, 2, sizeof(frame_fib_sum));
    // Store the address of the cont frame,
    //   used by fib_sum to forward the result.
    fp_fib_sum->s = &s;

    // Thread computing fib(n-1)
    // #pragma omp task firstprivate(n) output(n1)
    struct frame_fib1* fp_fib1 =
      TCreateDynamic(true, fib1, 1, sizeof(frame_fib));
    // Store the data needed for the thread computing fib(n-1).
    // fp_fib1 is the frame pointer where the data is stored to.
    // n-1 is the argument of function.
```

```
    // fp_fib_sum->arg1 is the field of the continuation
    //   thread where to store the result.
    fp_fib1->n = n;
    fp_fib1->cont = fp_fib_sum;
    fp_fib1->s1 = &fp_fib_sum->s1;
    TUpdate(true, fp_fib1);

    // Thread computing fib(n-2)
    // #pragma omp task firstprivate(n) output(n1)
    struct frame_fib2* fp_fib2 =
      TCreateDynamic(true, fib2, 1, sizeof(frame_fib));
    fp_fib2->n = n;
    fp_fib2->cont = fp_fib_sum;
    fp_fib2->s2 = &fp_fib_sum->s2;
    TUpdate(true, fp_fib2);

    TWait();

    return s;
  }
}

DF1_THREAD fib1() {
  int n = ((struct frame_fib1*)CFP)->n;
  *((struct frame_fib1*)CFP)->s1 = fib(n-1);
  TUpdate(true, ((struct frame_fib1*)CFP)->cont);

  TEnd();
}

DF1_THREAD fib2() {
  int n = ((struct frame_fib2*)CFP)->n;
  *((struct frame_fib2*)CFP)->s2 = fib(n-2);
  TUpdate(true, ((struct frame_fib2*)CFP)->cont);

  TEnd();
}

DF1_THREAD fib_sum() {
  int s1 = ((struct frame_fib_sum*)CFP)->s1;
  int s2 = ((struct frame_fib_sum*)CFP)->s2;
  int s = s1 + s2;
  (struct frame_fib_sum*)CFP)->s = s;

  TEnd();
}
```

The code generation scheme is very natural and systematic. Implementation in GCC is in progress and will serve as a basis for more aggressive, finer grain parallelization schemes.

However, because the original parallel code was using a synchronization barrier, the code does not expose as much parallelism as it could. In particular, it mostly exposes task parallelism and no pipeline parallelism across recursive calls. We will address this limitation in the next section.

4.5. Second parallel version

This is an illustration of task pragmas on functions, forming a nested pipeline and avoiding taskwait. We also decided not to decouple the execution in the control flow of the callee, but rather to make the fib function itself a dataflow thread using our extended OpenMP pragmas.

```
#pragma omp task input(n) output(*s)
int fib (int n, int *s) {
  {
    int s;
    if (n <= 1)
      *s = n;
    else {
      int s1, s2;
      fib(n-1, &s1);
      fib(n-2, &s2);
      *s = s1 + s2;
    }
  }
}

void main () {
  int s;

  #pragma omp parallel single
  {
    fib(7, &s);
  }

  printf("fib(7) = %d", s);

  exit(0);
}
```

This version seems to expose much more scalable parallelism. This will be confirmed when looking at the generated dataflow code in the next section.

4.6. Direct lowering of the second parallel version

To form a nested pipeline, we complete the previous code generation scheme by analyzing the inter-procedural data dependences, and passing a continuation to handle the value returned from the inner task(s). The continuation makes the program non-blocking, eliminating the TWait().

```
#define CFP TGetCurrentFramePointer();

struct frame_main_print {
  int arg;  // Implicit pipelining (usage of s in main) between
            // main and fib(n, &s).
};

struct frame_fib {
  int arg;    //#pragma omp task input(n) \
  int *ret;   //  output(*s)
  void *cont;
};

struct frame_fib_sum {
  int arg1;  // Implicit pipelining between fib(n-1, &s1)
  int arg2;  // fib(n-2, &s2)
  int *ret;  // and outer task fib(n, &s)
  void *cont;
};

L_THREAD void main() {
  // main() is always a legacy POSIX thread, not a dataflow thread.

  // The value returned(output) by fib(7, &s) will be used in the
  //   main thread. Build a continuation for the returned value.
  struct frame_main_print* fp_main_print =
    TCreateDynamic(true, main_print, 1, sizeof(frame_main_print));

  // Create an instance of the fib DF1b thread.
  // #pragma omp task input(n) output(*s)
  // the output(*s) here is connected to the outer main thread.
  struct frame_fib fp_fibn* =
    TCreateDynamic(true, fib, 1, sizeof(frame_fib));
  // Store the argument (to compute fib(7)).
  fp_fibn->arg = 7;
  // Store the continuation frame pointer.
  fp_fibn->cont = fp_main_print;
  // Store the address where the result of the continuation thread
  //   should be stored.
  fp_fibn->ret = &fp_main_print->arg;
  // Decrement the synchronization count of the fp_fibn instance.
  TUpdate(true, fp_fibn);

  exit(0); // Legacy threads are not managed by the TSU.
}

// Pipeline between main thread and fib.
// Create the continuation to handled the returned data by the inner task.
DFS_THREAD main_print() {
```

```
  // We assume a dataflow wrapper over printf is available,
  //   otherwise an S_THREAD should be used.
  printf("fib(7) = %d", ((struct frame_main_print*)CFP)->arg);

  TEnd();
}

DF1b_THREAD fib() {
  int n = CFP->arg;

  if (n <= 1) {
    // Return n to the continuation thread immediately.
    *((struct frame_fib*)CFP)->ret = n;
    TUpdate(True, ((struct frame_fib*)CFP)->cont);
  } else {
    // We create this continuation in the fib thread because there
    // is an implicit pipeline between task fib (n-1, &s1),
    // fib (n-2, &s2) and their outer task fib (n, &s).
    struct frame_fib_sum* fp_fib_sum =
      TCreateDynamic(True, fib_sum, 2, sizeof(frame_fib_sum));
    // Store the address of the cont frame,
    // used by fib_sum to forward the result.
    fp_fib_sum->cont = ((struct frame_fib*)CFP)->cont;
    fp_fib_sum->ret = ((struct frame_fib*)CFP)->ret;
    // Thread computing fib(n-1, &s1)
    struct frame_fib* fp_fib1 =
      TCreateDynamic(pred2, fib, 1, sizeof(frame_fib));
    // Store the data needed for the thread computing fib(n-1).
    // fp_fib1 is the frame pointer where the data is stored to.
    // n-1 is the argument of function.
    // fp_fib_sum->arg1 is the field of the continuation
    //   thread where to store the result.
    fp_fib1->arg = n-1;
    fp_fib1->cont = fp_fib_sum;
    fp_fib1->ret = &fp_fib_sum->arg1;
    TUpdate(True, fp_fib1);

    // Thread computing fib(n-2)
    // #pragma omp task firstprivate(n) output(s2)
    struct frame_fib* fp_fib2 =
      TCreateDynamic(pred2, fib, 1, sizeof(frame_fib));
    fp_fib2->arg = n-2;
    fp_fib2->cont = fp_fib_sum;
    fp_fib2->ret = &fp_fib_sum->arg2;
    TUpdate(True, fp_fib2);
  }

  TEnd();
}
```

```
//IMPLICIT pipelining between nested tasks.
//#pragma omp task input(s1, s2) output(s)
DF1_THREAD fib_sum() {
  int s1 = ((struct frame_fib_sum*)CFP)->arg1;
  int s2 = ((struct frame_fib_sum*)CFP)->arg2;
  int s = s1 + s2;
  *((struct frame_fib_sum*)CFP)->ret = s;
  TUpdate(true, ((struct frame_fib_sum*)CFP)->cont);

  TEnd();
}
```

4.7. Automatic parallelization of the source code, converting all procedural and conditional control flow to dataflow continuations

In the previous sections, we studied the lowering of the efficiency layer pragmas into TERAFLUX ISA extensions. In this section and the following ones, we go one step further and leverage the grain coarsening and implicit concurrency conversion techniques described in D4.1 to expose finer grain dataflow parallelism. In particular, we aim to generate as many DF1 threads as possible, rather than the more generic DF1b threads with internal control flow of the previous versions.

We do not detail the application of the parallel-stage decoupled software pipelining algorithm (see D4.1), but only show the generated code. Here, the code has been obtained manually, but the goal is to automate all of this in the second year of the project.

```
#define CFP TGetCurrentFramePointer();

struct frame_main_print {
  int arg;
};

struct frame_fib {
  int arg;
  int *ret;
  void *cont;
};

struct frame_fib_sum {
  int arg1;
  int arg2;
  int *ret;
  void *cont;
};

L_THREAD void main() {
  // main() is always a legacy POSIX thread, not a dataflow thread.

  // Create an instance of the main_print DFS thread.
```

```
  struct frame_main_print* fp_main_print =
    TCreateDynamic(true, main_print, 1, sizeof(frame_main_print));

  // Create an instance of the fib DF1 thread.
  struct frame_fib* fp_fibn =
    TCreateDynamic(true, fib, 1, sizeof(frame_fib));
  // Store the argument (to compute fib(7)).
  fp_fibn->arg = 7;
  // Store the continuation frame pointer.
  fp_fibn->cont = fp_main_print;
  // Store the address where the result of the continuation thread
  //   should be stored.
  fp_fibn->ret = &fp_main_print->arg;
  // Decrement the synchronization count of the fp_fibn instance.
  TUpdate(true, fp_fibn);

  exit(0); // Legacy threads are not managed by the TSU.
}

DFS_THREAD main_print() {
  // We assume a dataflow wrapper over printf is available,
  // otherwise an S_THREAD should be used
  printf("fib(7) = %d", ((struct frame_main_print*)CFP)->arg);

  TEnd();
}

DF1_THREAD fib() {
  int n = CFP->arg;
  bool pred1 = n <= 1;
  bool pred2 = !pred1;

  struct frame_fib* fp_fib_then =
    TCreateDynamic(pred1, fib_then, 1, sizeof(frame_fib));
  fp_fib_then->arg = n;
  fp_fib_then->ret = ((struct frame_fib*)CFP)->ret;
  fp_fib_then->cont = ((struct frame_fib*)CFP)->cont;
  TUpdate(pred1, fp_fib_then);

  struct frame_fib* fp_fib_else =
    TCreateDynamic(pred2, fib_else, 1, sizeof(frame_fib));
  fp_fib_else->arg = n;
  fp_fib_else->ret = ((struct frame_fib*)CFP)->ret;
  fp_fib_else->cont = ((struct frame_fib*)CFP)->cont;
  TUpdate(pred2, fp_fib_else);

  TEnd();
}
```

```
DF1_THREAD fib_then() {
  // Return n to the continuation thread immediately
  *((struct frame_fib*)CFP)->ret = n;
  TUpdate(true, ((struct frame_fib*)CFP)->cont);

  TEnd();
}

DF1_THREAD fib_else() {
  // Create the thread to compute the sum of fib(n-1) and fib(n-2).
  // Stalled until it gets data on both sides (last argument is 2).
  struct frame_fib_sum* fp_fib_sum =
    TCreateDynamic(true, fib_sum, 2, sizeof(frame_fib_sum));
  // Store the address of the cont frame,
  //   used by fib_sum to forward the result.
  fp_fib_sum->cont = ((struct frame_fib*)CFP)->cont;
  fp_fib_sum->ret = ((struct frame_fib*)CFP)->ret;

  // Thread computing fib(n-1)
  struct frame_fib* fp_fib1 =
    TCreateDynamic(true, fib, 1, sizeof(frame_fib));
  // Store the data needed for the thread computing fib(n-1).
  // fp_fib1 is the frame pointer where the data is stored to.
  // n-1 is the argument of function.
  // fp_fib_sum->arg1 is the field of the continuation thread where
  //   to store the result.
  fp_fib1->arg = n-1;
  fp_fib1->cont = fp_fib_sum;
  fp_fib1->ret = &fp_fib_sum->arg1;
  TUpdate(true, fp_fib1);

  // Thread computing fib(n-2)
  struct frame_fib* fp_fib2 =
    TCreateDynamic(true, fib, 1, sizeof(frame_fib));
  fp_fib2->arg = n-2;
  fp_fib2->cont = fp_fib_sum;
  fp_fib2->ret = &fp_fib_sum->arg2;
  TUpdate(true, fp_fib2);

  TEnd();
}

DF1_THREAD fib_sum() {
  int s1 = ((struct frame_fib_sum*)CFP)->arg1;
  int s2 = ((struct frame_fib_sum*)CFP)->arg2;
  int s = s1 + s2;
  *((struct frame_fib_sum*)CFP)->ret = s;
  TUpdate(true, ((struct frame_fib_sum*)CFP)->cont);
```

```
  TEnd();
}
```

Note that the code has been slightly optimized: the sum s is forwarded directly to the outer fib_sum thread or to the main thread, and not to an additional thread that would handle the return statement separately.

4.8. Automatic parallelization of the source code, converting all procedural control flow to dataflow continuations and if-converting conditional control flow

To further optimize the program and expose dataflow parallelism that may be easier to exploit on practical hardware (to mitigate synchronization and scheduling overheads), it is also possible to if-convert the innermost control flow rather than systematically converting it to dataflow continuations. The result of this transformation is depicted in the following version.

```
#define CFP TGetCurrentFramePointer();

struct frame_main_print {
  int arg;
};

struct frame_fib {
  int arg;
  int *ret;
  void *cont;
};

struct frame_fib_sum {
  int arg1;
  int arg2;
  int *ret;
  void *cont;
};

L_THREAD void main() {
  // main() is always a legacy POSIX thread, not a dataflow thread.

  // Create an instance of the main_print DFS thread.
  struct frame_main_print* fp_main_print =
    TCreateDynamic(true, main_print, 1, sizeof(frame_main_print));

  // Create an instance of the fib DF1 thread.
  struct frame_fib* fp_fibn =
    TCreateDynamic(true, fib, 1, sizeof(frame_fib));
  // Store the argument (to compute fib(7)).
  fp_fibn->arg = 7;
  // Store the continuation frame pointer.
```

```
  fp_fibn->cont = fp_main_print;
  // Store the address where the result of the continuation thread
  //   should be stored.
  fp_fibn->ret = &fp_main_print->arg;
  // Decrement the synchronization count of the fp_fibn instance.
  TUpdate(true, fp_fibn);

  exit(0); // Legacy threads are not managed by the TSU.
}

DFS_THREAD main_print() {
  // We assume a dataflow wrapper over printf is available,
  // otherwise an S_THREAD should be used.
  printf("fib(7) = %d", ((struct frame_main_print*)CFP)->arg);

  TEnd();
}

DF1_THREAD fib() {
  int n = CFP->arg;
  bool pred1 = n <= 1;
  bool pred2 = !pred1;

  // If pred1 is true, return n to the continuation
  //   thread immediately
  *((struct frame_fib*)CFP)->ret = n;
  TUpdate(pred1, ((struct frame_fib*)CFP)->cont);

  // The following operations are all guarded by pred2.

  // Create the thread to compute the sum of fib(n-1) and fib(n-2).
  // Stalled until it gets the data on both sides
  //   (last argument is 2).
  struct frame_fib_sum* fp_fib_sum =
    TCreateDynamic(pred2, fib_sum, 2, sizeof(frame_fib_sum));
  // Store the address of the cont frame,
  //   used by fib_sum to forward the result.
  fp_fib_sum->cont = ((struct frame_fib*)CFP)->cont;
  fp_fib_sum->ret = ((struct frame_fib*)CFP)->ret;
  // Thread computing fib(n-1)
  // #pragma omp task firstprivate(n) output(s1)
  struct frame_fib* fp_fib1 =
    TCreateDynamic(pred2, fib, 1, sizeof(frame_fib));
  // Store the data needed for the thread computing fib(n-1).
  // fp_fib1 is the frame pointer where the data is stored to.
  // n-1 is the argument of function.
  // fp_fib_sum->arg1 is the field of the continuation thread where
  //   to store the result.
  fp_fib1->arg = n-1;
```

```
  fp_fib1->cont = fp_fib_sum;
  fp_fib1->ret = &fp_fib_sum->arg1;
  TUpdate(pred2, fp_fib1);

  // Thread computing fib(n-2)
  // #pragma omp task firstprivate(n) output(s2)
  struct frame_fib* fp_fib2 =
    TCreateDynamic(pred2, fib, 1, sizeof(frame_fib));
  fp_fib2->arg = n-2;
  fp_fib2->cont = fp_fib_sum;
  fp_fib2->ret = &fp_fib_sum->arg2;
  TUpdate(pred2, fp_fib2);

  TEnd();
}

//#pragma omp task input(s1, s2) lastprivate(s)
DF1_THREAD fib_sum() {
  int s1 = ((struct frame_fib_sum*)CFP)->arg1;
  int s2 = ((struct frame_fib_sum*)CFP)->arg2;
  int s = s1 + s2;
  *((struct frame_fib_sum*)CFP)->ret = s;
  TUpdate(true, ((struct frame_fib_sum*)CFP)->cont);

  TEnd();
}
```

4.9. Status Report and Partial Conclusions

As announced in the introduction, the implementation of these two main scenarios, and their variants, is still underway. This delay has been an occasion to reinforce our automatic techniques to convert implicit concurrency into coarser grain, scalable parallelism (see D4.1) and to realize that a comparison of the different code generation styles for dataflow architectures will be necessary. We will complete the implementation in the second year of the project and perform the in-depth comparison together with the multi-level parallelization and locality optimizations conducted in Task 4.2.

The development branch of GCC in which these researches take place is not yet publicly available. The OpenMP extensions used in this deliverable are supported by the streamization branch of GCC and can be obtained from the GCC svn:

```
svn checkout svn://gcc.gnu.org/svn/gcc/branches/streamization
```

5. Compiling StarSs

While the previous section discussed code generation issues, we now address the front-end compilation challenges and infrastructure supporting the efficiency programming model (with pragmas) defined in WP3.

StarSs is a task-based programming model that enables the exploitation of the applications' inherent parallelism at the task level. To mark the tasks in a StarSs application, annotations (pragmas) similar to OpenMP are used. While this is already found in OpenMP 3.0, a unique feature of StarSs tasks is the input, output or inout clauses that applied to tasks' parameters enable the runtime to track tasks' data dependences. A task dependence graph is dynamically built and scheduled for execution in the different devices. Also the clause “target device” to specify that a given task code is tailored to a specific device type (e.g., a custom vector accelerator node) has been defined.

This section reviews the StarSs family of annotation languages, recalls the main syntactic and semantic features, and then describes the compilation infrastructure involved. The TERAFLUX project enabled us to conduct a necessary reengineering and a systematic merge of the different tools associated with StarSs into a common, coherent infrastructure. For this first deliverable, we report on this effort, discussing the techniques and compiler tools common to all flavors of StarSs, targeting different classes of parallel hardware. The specialization of these techniques and tools for the TERAFLUX architecture will be described in a future deliverable.

StarSs family

There are several instantiations of StarSs, specialized on different target architectures, like CellSs that targets Cell/B.E. architectures or SMPSs that target shared memory machines. OmpSs is another implementation of StarSs that also integrates OpenMP. It is a BSC project recently started with the aim of pushing the StarSs ideas in the OpenMP standard. While a prototype version of GPUSs was implemented based on the CellSs and SMPSs runtimes, BSC decided to continue the effort of this development integrated with the OmpSs infrastructure.

OmpSs, which can also include OpenCL or CUDA kernels, is a solution for easy programming of heterogeneous architectures. OmpSs can be used to run on plain SMP machines, and SMP machines with GPUs. There is also an implementation of the model for clusters under development. OmpSs leverages, from OpenMP, the possibility to inline loop parallelization and task definition pragmas (avoiding the need to manually outline tasks) and the possibility of nesting tasks. StarSs extensions allow runtime dependence analysis between tasks, and automatic data transfers. OpenCL and CUDA allow the programmer to easily write efficient and portable SIMD kernels to be exploited inside the tasks.

StarSs syntax

This section reviews the StarSs syntax, that it is based on a few directives that annotate the code, the main one being the task pragma. The input/output/inout clauses in StarSs are mandatory for all the arguments of the annotated functions to indicate their direction and are basic for the creation of the data-flow graph of the application at runtime.

```
#pragma css task[input ( parameters ) ] \
```

```
                [output ( parameters ) ] \
                [inout ( parameters )] \
                [target device( [cell, smp, cuda] ) ] \
                [implements ( task_name ) ] \
                [reduction ( parameters ) ] \
                [ highpriority ]

#pragma css wait on ( data_address )
#pragma css barrier
#pragma css mutex lock ( variable )
#pragma css mutex unlock( variable )
```

For example, the following pragma annotates a dgemm function, therefore denoting that all invocations of dgemm in the application will become a task.

```
#pragma css task input (A, B, BS) inout (C)
void dgemm (double A[BS][BS],
            double B[BS][BS],
          double C[BS][BS],
        int BS);
```

To support heterogeneity, two classes have been added: the `target device` clause specifies that the code of the task is specific for a given device (i.e., SPU for Cell, CUDA for NVIDIA GPUs or smp for general purpose processors). Another clause that has been added to support heterogeneity is the clause `implements` that specifies alternative implementations for a function. For example, for the same case of the *dgemm* function an alternative implementation that runs of NVIDIA GPUS can be given (in the absence of the `target device` clause, the default is `smp` that enables the task to be run on a general purpose processor):

```
#pragma css task input (A, B, BS) inout (C)
void dgemm (double A[BS][BS],
            double B[BS][BS],
            double C[BS][BS],
        int BS);
#pragma css task input (A, B, BS) inout (C)\
target device (cuda) implements (dgemm)
void dgemm_cuda (double A[BS][BS],
                 double B[BS][BS],
            double C[BS][BS], int BS);
```

With the objective of extending OpenMP with the data-flow ideas of StarSs, similar extensions have been defined on the OmpSs syntax for OpenMP. One of the differences with the StarSs syntax is that for OpenMP is not required to define in the `input/output/inout` clauses all the parameters of the functions although all those that will define the data dependences between tasks:

```
#pragma omp task inout (C)
void dgemm (double A[BS][BS],
            double B[BS][BS],
            double C[BS][BS], int BS);
```

```
#pragma omp target device (cuda) implements (dgemm)\
copy_in (A, B, BS)
void dgemm_cuda (double A[BS][BS],
                 double B[BS][BS],
            double C[BS][BS], int BS);
```

Another difference with StarSs syntax is that inlined pragmas are allowed. This saves the programmer from outlining the task code in functions, although the pragma should then be inserted in all the corresponding points in the code.

StarSs Compilation

StarSs is a generic programming model and several prototype implementations have been done so far: CellSs, SMPSs, MPI/SMPSs, etc. However, recently BSC started a task force to join all these ideas in a single compiler (Mercurium) and runtime infrastructure (Nanos++) called OmpSs. Since the future compiler efforts will be devoted to this infrastructure, we relate to it in this section.

Mercurium C/C++ is a source-to-source compiler, i.e., it does not generate object code (many other compilers do this very well). Mercurium modifies, rewrites, translates, changes, messes, and/or mixes the input source code into another source code that is then feed to a object-code generating compiler (the native compiler or backend compiler.

The compiler plays a relatively minor role on implementing the OmpSs model. The compiler recognizes the constructs and transforms them into calls to the Nanos++ runtime library. Most of OmpSs is supported by the current version of the compiler.

The dependences clauses are transformed by generating a set of expressions that will be evaluated when the application is executed. These expressions will generate addresses of memory that will be passed to the runtime library. In our current implementation, the target construct can only be applied to a task construct but we envision than in future revisions it will be possible to apply it to functions and OpenMP work sharing as well. The target devices that are currently supported are: SMP, CUDA and Cell. There is ongoing work to support OpenCL.

When the compiler is going to generate the code for a task construct it looks to see if it is immediately preceded with a target directive or if another target directive is linked to this task construct by means of an implement clause. If so, then the AST of the task is passed onto a device-specific provider for each non-SMP device.

This provider generates the device-dependent data that must be associated with the task. It also, if necessary, generate a specialized outline for the device which may need to be generated in a separate file. This additional file is reintroduced in the compiler pipeline following usually a different compilation profile that will invoke different backend tools (e.g., the NVidia compiler, nvcc, for CUDA devices).

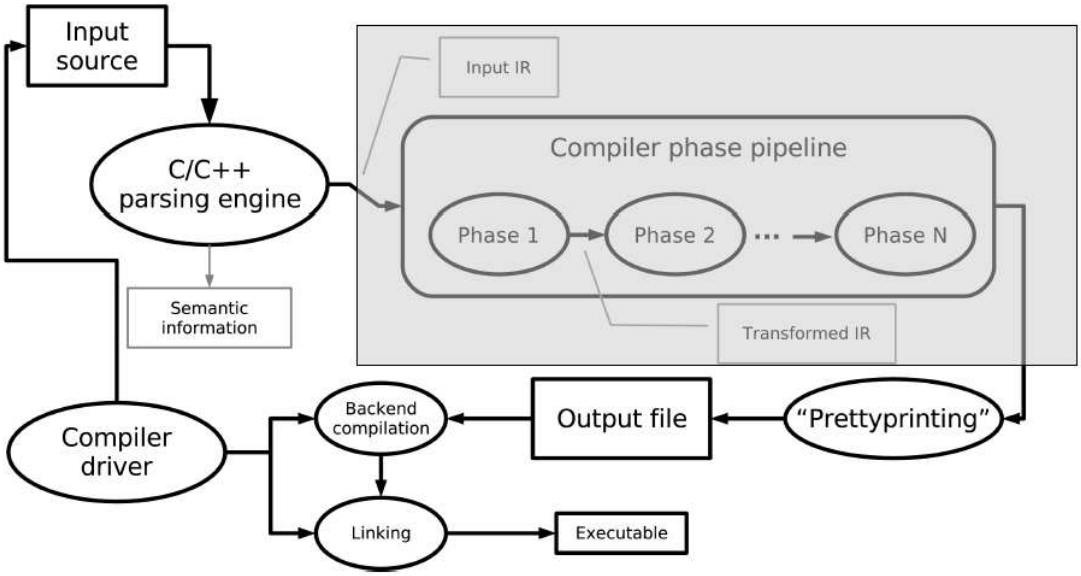
Internals of Mercurium

In Mercurium, new compiler phases are added as dynamic libraries to the compiler. These phases are written in C++ though scripting languages (like Python or Ruby) are under consideration in order to speed up prototyping. As said before, Mercurium is a source-to-source compiler, therefore it is based

on source generation and it does not spend too much time or use many strategies modifying syntax trees when transforming the code.

Mercurium supports single source compilation, therefore it can generate more than one file for each input source file. This is used for example to support heterogeneous architectures where two or more backend compilers are used.

Mercurium provides extensive C++ support, aiming at supporting all of C++2003. Also, many GCC extensions are supported.



The figure above shows the pipeline of the Mercurium compiler. The loop from the compiler driver back to the input source denotes that in some cases, for a given input file, the process is repeated one or more times.

The compiler is extended by means of compiler phases. Every compiler phase is a dynamically loadable library. Phases are written in C++ using an API called TL. Mercurium on start loads a sequence of compiler phases (a profile determines which phases and their order in each case). After parsing, these phases run one after the other. Phases receive a common very high-level internal representation (IR) but they may create a new IR that can be used by subsequent phases.

Mercurium has been designed to provide support to multiple programming models. For example, right now it gives support to StarSs and OpenMP pragmas (with the data-flow extensions). When building a new phase for Mercurium it is not always necessary to start from scratch. For example, if there is a need for handling new #pragma lines, PragmaCustomCompilerPhase can be subclassed. When extending OpenMP, can be subclassed. These classes simplify finding specific constructions.

Tools

The OmpSs infrastructure software is currently hosted at nanos.ac.upc.edu

The corresponding download URLs are:

- For the compiler: http://nanos.ac.upc.edu/projects/mcxx/downloads
- For the runtime: http://nanos.ac.upc.edu/content/nanos-environment-distribution

6. Conclusions

We have reported on the design and implementation of the compiler flow targeted to the TERAFLUX architecture.

Due to delays in the integration of the different ISA extension proposals, the back-end work was delayed and we describe the different compilation schemes and forms taken by the target code generation on one example, instead of reporting on the actual compiler tool (still being actively developed at this time).

On the front-end side, we also describe the infrastructure effort conducted to unify the different semantical and syntactic approaches in the StarSs family of pragma-based languages serving as the efficiency layer for TERAFLUX programming. This infrastructure is based on the Mercurium source-to-source compiler, and will be connected to the TERAFLUX port of the GCC back-end compiler through the unified intermediate representation described in D4.2.

Both front-end and back-end tools are accessible for public download and will be updated continuously in the course of the project.