Automatic Task Generation on the SCMP architecture for data flow applications Progress report SCALOPES Project

Béatrice Creusillet
HPC Project

March 2011

1 Introduction

The SCMP architecture is a CEA proprietary MP-SoC architecture, which provides a dedicated environment for the execution of multiple tasks applications under the control of a specific task. Inter-tasks communications are performed through *buffers* allocated on a shared memory available through a specific API, and explicit synchronizations are also available. This report describes how we automatically transform streaming applications written in traditional C code into applications dedicated to this architecture and its simulator, Sesam.

We do not address the problem of identifying tasks, but we focus on the generation of code from an application in which task kernels are already identified by pragmas[1].

Our ambition of being able to handle any control flow, combined with the characteristics of the targeted architecture HAL, leads us to propose a client-server model to maintain data consistency. In this model, there is one *server task* (ST_s) for each bufferized datum. *Kernel tasks* communicate with them either to update the shared data values on the servers , or to retrieve them from the servers. This is examplified in Figure 1 where $T_0, \dots, T_4$ are original kernel tasks specified by the user. S_a, S_b, S_c are server tasks for arrays a, b and c. They communicate with each other through buffers modelled by rectangles. Tasks T_α and T_ω are additional start and end tasks required for synchronization purposes; these synchronizations are modelled by the green edges. The control task first lauches Task T_α and waits for its termination before lauching Tasks T_0 to T_4 as well as T_a, T_b and T_c. The synchronization between these tasks is ensured by the fact that they all execute the same code, only specialized by values describing how each task uses shared buffers. This is further described in Section 2. The control task waits for the termination of these tasks before lauching T_ω, and terminates at the end of the latter.

However, in an application where each datum is defined by a single task and used by only one other task, this model generates extra communications and data copying compared to what would be done manually, with direct communications between kernel tasks through a single buffer. In Section 4 we explore possible optimizations to get closer to hand generated code.

[1]We currently use labels because one of the PIPS phase our transformation relies on uses labels, but pragmas could also be used and automatically transformed into labels.

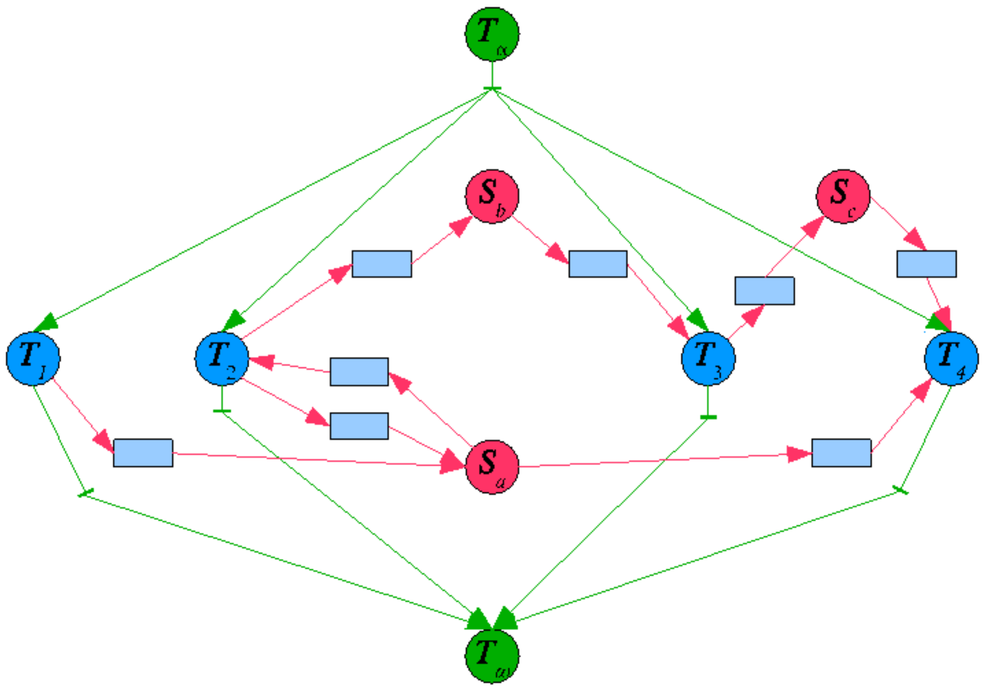

Figure 1: Communicating through server tasks

2 Automatic task generation

The existing variety of computer architectures and the rapid changes in this field make it necessary for compilers to reuse existing code transformations as much as possible. This may even lead to rely on a unique programming model target, which is then adapted to different architectures, possibly at run-time. The advantage is that all sub-compilers automatically benefit from improvements of the code transformations, and that adding a new target is only a matter of implementing new pre/post-processing phases and a new run-time.

This is the chosen approach for our SCMP compiler. We rely on an existing PIPS phase, *isolate statement* [9, 6] which for any statement, generates communications for input and output data. The original target is a SPMD computational model: the generated code is the same for the unique host and all the nodes, but the called communication functions are different at run-time.

However, simply using this model with kernel tasks playing the role of nodes and a host task playing the role of the host (and then of a unique centralized data server) would sequentialize the execution of the tasks. The idea is then to have as many data server tasks as original data to ensure the data consistency throughout the application execution without impeding task parallelism.

Another advantage in the specific case of the SCMP architecture, is that the model ensures that communications are performed on a one-to-one basis: for each produced buffer, there is a unique consumer. So there is no need to worry about the number of consumers of a buffer, even if several tasks consume the same original data: server tasks perform a specific communication for each consumer.

As this may sound mysterious to a new-comer in PIPS and PAR4ALL universe, we further explain this from the sample study case of Figure 2 in Section 2.1. Section 2.2 then presents

```
int main()
{
  int i, t, a[20], b[20];
  for (t=0; t < 100; t++)
  {
kernel_tasks_1:
    for(i=0; i <10; i++)
      a[i] = i+t;

kernel_tasks_2:
    for(i=10; i <20; i++)
      a[i] = 2*i+t;

kernel_tasks_3:
    for(i=10; i <20; i++)
      printf("a[%d] = %d\n", i, a[i]);
   }
   return(0);
}
```

Figure 2: Sample case

the specific run-time for the SCMP architecture using the SESAM simulator API.

2.1 Sample case study

This code is special becasue the input data is partially initialized by two different tasks (`kernel_tasks_1` and `kernel_tasks_2`), and is entirely used by a unique task (`kernel_tasks_3`).

Our SCMP compiler first applies Phase `isolate_statement` on each task statement, to obtain the code of Figure 3.

Notice that each task inner code now uses a local variable (prefixed by `P4A__`), which is either initialized from the original program variable if the task consumes the data, using the `P4A_copy_to_accel_1d` function, or which is used to update the original program variable if the task produces the data, using the `P4A_copy_from_accel_1d` function.

The compiler next phase generates a header file (see Figure 4) to be included by all tasks, and which gives the buffer identifiers (for instance `#define P4A__a__0_id 0`), and the way each task uses them. For instance, task `scmp_task_1` produces buffer `P4A__a__0`, which is expressed by `#define P4A__a__0_prod_p 1` if `scmp_task_1` is defined.

The code of Figure 3 is then modified to use the PAR4ALL run-time for the SCMP architecture, which ensures the synchronizations for the buffers productions and consumptions. Actually, the SESAM API uses integers to identify buffers instead of variable names. These identifiers must then be given as arguments to the PAR4ALL SCMP API. Moreover, as the executed code is the same for all tasks, additional arguments must be added to control whether a particular run-time function call is really executed, or how it is executed. Similarly, guards are added to control the execution of the original tasks codes. For instance "`if (scmp_task_1_p)`" is added to guard the execution of the first task original code, to ensure that only task `scmp_task_1` executes this code. This leads to the code of figure 5.

Next, for each task, either kernel or server, the code is replicated and specialized by adding a preprocessing directive, "`#define scmp_task_1 1`" in the case of Task 1, at the very beginning. Server tasks PAR4ALL run-time calls are also modified to use specific versions. Figure 6 gives the generated code for Task 1 while Figure 7 gives the generated code for the server task of array `a` . File `scmp_buffers` is the buffers description header file, and `data_flow03_event_val.h` is the synchronization event header file which is produced by SESAM control task compilation tools.

```
int main()
{
   int i, t, a[20], b[20];
   for(t = 0; t <= 99; t += 1) {
      {
         //PIPS generated variable
         int (*P4A__a__0)[10] = (int (*)[10]) 0;
         P4A_malloc((void **) &P4A__a__0, sizeof(int)*10);

kernel_task_1:
         for(i = 0; i <= 9; i += 1)
            (*P4A__a__0)[i-0] = i+t;
         P4A_copy_from_accel_1d(sizeof(int), 20, 10, 0, &a[0], *P4A__a__0);
         P4A_dealloc(P4A__a__0);
      }
      {
         //PIPS generated variable
         int (*P4A__a__1)[10] = (int (*)[10]) 0;
         P4A_malloc((void **) &P4A__a__1, sizeof(int)*10);

kernel_task_2:
         for(i = 10; i <= 19; i += 1)
            (*P4A__a__1)[i-10] = 2*i+t;
         P4A_copy_from_accel_1d(sizeof(int), 20, 10, 10, &a[0], *P4A__a__1);
         P4A_dealloc(P4A__a__1);
      }
      {
         //PIPS generated variable
         int (*P4A__a__2)[10] = (int (*)[10]) 0;
         P4A_malloc((void **) &P4A__a__2, sizeof(int)*10);
         P4A_copy_to_accel_1d(sizeof(int), 20, 10, 10, &a[0], *P4A__a__2);

kernel_task_3:
         for(i = 10; i <= 19; i += 1)
            printf("a[%d] = %d\n", i, (*P4A__a__2)[i-10]);
         P4A_dealloc(P4A__a__2);
      }
   }
   return(0);
}
```

Figure 3: Code after applying `isolate_statement`

```
#define NB_BUFFERS 3
#define P4A__a__0_id 0
#define P4A__a__1_id 1
#define P4A__a__2_id 2

#ifdef kernel_task_1
#define kernel_task_1_p 1
#define P4A__a__0_cons_p 0
#define P4A__a__0_prod_p 1
#define P4A__a__1_cons_p 0
#define P4A__a__1_prod_p 0
#define P4A__a__2_cons_p 0
#define P4A__a__2_prod_p 0
#else
#define kernel_task_1_p 0
#endif

#ifdef kernel_task_2
#define kernel_task_2_p 1
#define P4A__a__0_cons_p 0
#define P4A__a__0_prod_p 0
#define P4A__a__1_cons_p 0
#define P4A__a__1_prod_p 1
#define P4A__a__2_cons_p 0
#define P4A__a__2_prod_p 0
#else
#define kernel_task_2_p 0
#endif

#ifdef kernel_task_3
#define kernel_task_3_p 1
#define P4A__a__0_cons_p 0
#define P4A__a__0_prod_p 0
#define P4A__a__1_cons_p 0
#define P4A__a__1_prod_p 0
#define P4A__a__2_cons_p 1
#define P4A__a__2_prod_p 0
#else
#define kernel_task_3_p 0
#endif

#ifdef P4A_sesam_server_a
#define P4A_sesam_server_a_p 1
#define P4A__a__0_cons_p 1
#define P4A__a__0_prod_p 0
#define P4A__a__1_cons_p 1
#define P4A__a__1_prod_p 0
#define P4A__a__2_cons_p 0
#define P4A__a__2_prod_p 1
#else
#define P4A_sesam_server_a_p 0
#endif
```

Figure 4: Buffers description header file

```
int main()
{
  P4A_scmp_reset();
  int i, t, a[20], b[20];
  for(t = 0; t <= 99; t += 1) {
    {
      //PIPS generated variable
      int (*P4A__a__0)[10] = (int (*)[10]) 0;
      P4A_malloc((void **) &P4A__a__0, sizeof(int)*10,
                 P4A__a__0_id,
                 P4A__a__0_prod_p || P4A__a__0_cons_p, P4A__a__0_prod_p);

      if (kernel_task_1_p)
        for(i = 0; i <= 9; i += 1)
          (*P4A__a__0)[i-0] = i+t;
      P4A_copy_from_accel_1d(sizeof(int), 20, 10, 0,
                 P4A_sesam_server_a_p ? &a[0] : NULL, *P4A__a__0,
                 P4A__a__0_id, P4A__a__0_prod_p || P4A__a__0_cons_p);
      P4A_dealloc(P4A__a__0, P4A__a__0_id,
                  P4A__a__0_prod_p || P4A__a__0_cons_p, P4A__a__0_prod_p);
    }
    {
      //PIPS generated variable
      int (*P4A__a__1)[10] = (int (*)[10]) 0;
      P4A_malloc((void **) &P4A__a__1, sizeof(int)*10,
                 P4A__a__1_id,
                 P4A__a__1_prod_p || P4A__a__1_cons_p, P4A__a__1_prod_p);

      if (kernel_task_2_p)
        for(i = 10; i <= 19; i += 1)
          (*P4A__a__1)[i-10] = 2*i+t;
      P4A_copy_from_accel_1d(sizeof(int), 20, 10, 10,
                 P4A_sesam_server_a_p ? &a[0] : NULL, *P4A__a__1,
                 P4A__a__1_id, P4A__a__1_prod_p || P4A__a__1_cons_p);
      P4A_dealloc(P4A__a__1, P4A__a__1_id,
                  P4A__a__1_prod_p || P4A__a__1_cons_p, P4A__a__1_prod_p);
    }
    {
      //PIPS generated variable
      int (*P4A__a__2)[10] = (int (*)[10]) 0;
      P4A_malloc((void **) &P4A__a__2, sizeof(int)*10,
                 P4A__a__2_id,
                 P4A__a__2_prod_p || P4A__a__2_cons_p, P4A__a__2_prod_p);
      P4A_copy_to_accel_1d(sizeof(int), 20, 10, 10,
                 P4A_sesam_server_a_p ? &a[0] : NULL, *P4A__a__2,
                 P4A__a__2_id, P4A__a__2_prod_p || P4A__a__2_cons_p);

      if (kernel_task_3_p)
        for(i = 10; i <= 19; i += 1)
          sesam_printf("a[%d] = %d\n", i, (*P4A__a__2)[i-10]);
      P4A_dealloc(P4A__a__2, P4A__a__2_id,
                  P4A__a__2_prod_p || P4A__a__2_cons_p, P4A__a__2_prod_p);
    }
  }
  return(0);
}
```

Figure 5: Code after transformation to use the Par4All scmp run-time

```
#define kernel_task_1 1
#include <sesam_com.h>
#include "p4a_scmp.h"
#include "scmp_buffers.h"
#include "data_flow03_event_val.h"

int main()
{
  P4A_scmp_reset();
  int i, t, a[20], b[20];
  for(t = 0; t <= 99; t += 1) {
    {
      //PIPS generated variable
      int (*P4A__a__0)[10] = (int (*)[10]) 0;
      P4A_malloc((void **) &P4A__a__0, sizeof(int)*10,
                 P4A__a__0_id,
                 P4A__a__0_prod_p || P4A__a__0_cons_p, P4A__a__0_prod_p);

      if (kernel_task_1_p)
        for(i = 0; i <= 9; i += 1)
          (*P4A__a__0)[i-0] = i+t;
      P4A_copy_from_accel_1d(sizeof(int), 20, 10, 0,
                 P4A_sesam_server_a_p ? &a[0] : NULL, *P4A__a__0,
                 P4A__a__0_id, P4A__a__0_prod_p || P4A__a__0_cons_p);
      P4A_dealloc(P4A__a__0, P4A__a__0_id,
                  P4A__a__0_prod_p || P4A__a__0_cons_p, P4A__a__0_prod_p);
    }
    {
      //PIPS generated variable
      int (*P4A__a__1)[10] = (int (*)[10]) 0;
      P4A_malloc((void **) &P4A__a__1, sizeof(int)*10,
                 P4A__a__1_id,
                 P4A__a__1_prod_p || P4A__a__1_cons_p, P4A__a__1_prod_p);

      if (kernel_task_2_p)
        for(i = 10; i <= 19; i += 1)
          (*P4A__a__1)[i-10] = 2*i+t;
      P4A_copy_from_accel_1d(sizeof(int), 20, 10, 10,
                 P4A_sesam_server_a_p ? &a[0] : NULL, *P4A__a__1,
                 P4A__a__1_id, P4A__a__1_prod_p || P4A__a__1_cons_p);
      P4A_dealloc(P4A__a__1, P4A__a__1_id,
                  P4A__a__1_prod_p || P4A__a__1_cons_p, P4A__a__1_prod_p);
    }
    {
      //PIPS generated variable
      int (*P4A__a__2)[10] = (int (*)[10]) 0;
      P4A_malloc((void **) &P4A__a__2, sizeof(int)*10,
                 P4A__a__2_id,
                 P4A__a__2_prod_p || P4A__a__2_cons_p, P4A__a__2_prod_p);
      P4A_copy_to_accel_1d(sizeof(int), 20, 10, 10,
                 P4A_sesam_server_a_p ? &a[0] : NULL, *P4A__a__2,
                 P4A__a__2_id, P4A__a__2_prod_p || P4A__a__2_cons_p);

      if (kernel_task_3_p)
        for(i = 10; i <= 19; i += 1)
          sesam_printf("a[%d] = %d\n", i, (*P4A__a__2)[i-10]);
      P4A_dealloc(P4A__a__2, P4A__a__2_id,
                  P4A__a__2_prod_p || P4A__a__2_cons_p, P4A__a__2_prod_p);
    }
  }
  return(ev_T001);
}
```

Figure 6: Final code for kernel Task 1

```
#define P4A_sesam_server_a 1
#include <sesam_com.h>
#include "p4a_scmp.h"
#include "scmp_buffers.h"
#include "data_flow03_event_val.h"

int main()
{
  P4A_scmp_reset();
  int i, t, a[20], b[20];
  for(t = 0; t <= 99; t += 1) {
    {
      //PIPS generated variable
      int (*P4A__a__0)[10] = (int (*)[10]) 0;
      P4A_malloc((void **) &P4A__a__0, sizeof(int)*10,
                 P4A__a__0_id,
                 P4A__a__0_prod_p || P4A__a__0_cons_p, P4A__a__0_prod_p);

      if (kernel_task_1_p)
        for(i = 0; i <= 9; i += 1)
          (*P4A__a__0)[i-0] = i+t;
      P4A_copy_from_accel_1d(sizeof(int), 20, 10, 0,
                 P4A_sesam_server_a_p ? &a[0] : NULL, *P4A__a__0,
                 P4A__a__0_id, P4A__a__0_prod_p || P4A__a__0_cons_p);
      P4A_dealloc(P4A__a__0, P4A__a__0_id,
                  P4A__a__0_prod_p || P4A__a__0_cons_p, P4A__a__0_prod_p);
    }
    {
      //PIPS generated variable
      int (*P4A__a__1)[10] = (int (*)[10]) 0;
      P4A_malloc((void **) &P4A__a__1, sizeof(int)*10,
                 P4A__a__1_id,
                 P4A__a__1_prod_p || P4A__a__1_cons_p, P4A__a__1_prod_p);

      if (kernel_task_2_p)
        for(i = 10; i <= 19; i += 1)
          (*P4A__a__1)[i-10] = 2*i+t;
      P4A_copy_from_accel_1d(sizeof(int), 20, 10, 10,
                 P4A_sesam_server_a_p ? &a[0] : NULL, *P4A__a__1,
                 P4A__a__1_id, P4A__a__1_prod_p || P4A__a__1_cons_p);
      P4A_dealloc(P4A__a__1, P4A__a__1_id,
                  P4A__a__1_prod_p || P4A__a__1_cons_p, P4A__a__1_prod_p);
    }
    {
      //PIPS generated variable
      int (*P4A__a__2)[10] = (int (*)[10]) 0;
      P4A_malloc((void **) &P4A__a__2, sizeof(int)*10,
                 P4A__a__2_id,
                 P4A__a__2_prod_p || P4A__a__2_cons_p, P4A__a__2_prod_p);
      P4A_copy_to_accel_1d(sizeof(int), 20, 10, 10,
                 P4A_sesam_server_a_p ? &a[0] : NULL, *P4A__a__2,
                 P4A__a__2_id, P4A__a__2_prod_p || P4A__a__2_cons_p);

      if (kernel_task_3_p)
        for(i = 10; i <= 19; i += 1)
          sesam_printf("a[%d] = %d\n", i, (*P4A__a__2)[i-10]);
      P4A_dealloc(P4A__a__2, P4A__a__2_id,
                  P4A__a__2_prod_p || P4A__a__2_cons_p, P4A__a__2_prod_p);
    }
  }
  return(ev_T004);
}
```

Figure 7: Final code for server task of array a

```
#include <sesam_com.h>
#include "p4a_scmp.h"
#include "data_flow03_event_val.h"

int main(){
        return(ev_T000);
}
```

Figure 8: Start task

```
#include <sesam_com.h>
#include "p4a_scmp.h"
#include "data_flow03_event_val.h"

int main(){
        return(END_APPLI);
}
```

Figure 9: End task

A start task and an end task are then created (see Figure 8 and 9) for synchronization purposes, as well as a control task (Figure 10), written in a proprietary language. It mainly describes the explicit task synchronizations, and the necessary resources, such as the stack sizes of all tasks. Our compiler generates arbitrary stack sizes which have to be further hand modified to meet the stack sizes actually required for each task.

All generated files are placed in a new directory placed in directory `applis_processing/`, which is created if it does not exists). The control task file is placed in directory `applis/`. `Makefile.arp` file is also created in directory `applis_processing/` for building the whole application.

2.2 SCMP run-time

The compilation process presented in the previous section relies on a specific version of the Par4All run-time for the allocation/freeing of shared buffers, and the synchronizations through these buffers of kernel and server tasks. This section describes all the run-time functions for the scmp architecture. It's important to keep in mind that each Par4All run-time function call in a kernel task, corresponds to a similar call in the server task, which is executed concurrently. As a consequence, to each wait corresponds one and only one send.

2.2.1 Allocation

The allocation function is called each time a buffer is required, even inside surrounding loops. A crude implementation, systematically invoking `sesam_reserve_data` and `sesam_data_assignation` at each call, would have led to a potentially huge overhead in execution time compare to a manual version which would place these calls outside of the nested loops if the allocated size is independent of loop varying variables.

Instead, an optimization is performed at run-time: a global variable[2] stores the buffer current size, and each time the allocation function is called, the needed size is checked against the current size; a new allocation is performed only if the needed size is different from the current size. The code for the allocation function is provided in Figure 11. The global

[2]There is one global variable per task process, it is not a variable allocated in the shared memory.

```
NBTASK   6
NBEVENT  5
DEADLINE          1000000
NAME_TASKS   T000  T001  T002  T003  T004  T005
SOURCE     T000  T001  T002  T003  T004  T005
SIZE_STACK   1024  1024  1024  1024  1024  1024
INIT  T000;
NEXT  T000 = T001,T002,T003,T004,ev_T000;
NEXT  T001,T002,T003,T004 = T005,ev_T001,ev_T002,ev_T003,ev_T004;
NEXT  T005 = END;
LENGTH  T000,  1 = 1000;
LENGTH  T001,  1 = 10000;
LENGTH  T002,  1 = 10000;
LENGTH  T003,  1 = 10000;
LENGTH  T004,  1 = 10000;
LENGTH  T005,  1 = 1000;
ENDAPPLI;
```

Figure 10: Control task

array is initialized by a call to function `P4A_scmp_reset()` (Figure 12) at the beginning of each task.

Notice that the allocation is performed only by the producer of the data. This is controlled through an argument of the function (`producer_p`). Notice also that a blocking wait is systematically executed if the buffer has never been allocated. It ensures that it is ready to be written (if it doesn't have to be re-allocated) or to be freed (if it's allocated size has to be changed). In the case where the allocated size has to be changed, a blocking wait is issued just after the allocation, to ensure that the data is ready before being written. In all cases, the buffer is ready to be written on exit of the function.

Function `sesam_map_data` is called whenever the task produces or consumes the buffer (`do_it` is true): it return value is a pointer towards the shared buffer, which can be handled as a usual variable. The current maximum of eight simultaneously mapped data per task is checked neither at compile time nor at run time. So the task specifier has to check before hand that each kernel task does not use more than eight input/output array data.

Lastly, the code is identical for server and kernel tasks. Different versions may be required to handle tasks which produce *and* consume the same original data or for which the input data set cannot be accurately represented (see Section 4).

2.2.2 Deallocation

The *deallocation* function (Figure 13) is called after the execution of the core of the task. On the consumer side, it warns the producer, that the buffer is now available for write usage. On both sides, the `sesam_unmap_data` function is called to release a mapped buffer slot.

Notice that this function does not free the shared buffer because we do not know at this time if it will be re-used in next iterations. As a consequence, a buffer for a task which does not belong to a loop nest, or a buffer used in the last iteration of a loop nest, is never freed. This may be annoying in the case of successive loop nests re-using the same original data for instance.

2.2.3 Copying to a kernel task from a server task

Once the buffer has been allocated and is ready for writing, using it for a server to kernel communication is really simple. Figures 14 and 15 respectively give the functions used on

```
size_t current_buffer_size[NB_BUFFERS];

void P4A_scmp_malloc(void **dest, size_t n,
                     unsigned int dest_id, int do_it, int producer_p)
{
  if(do_it)
  {
    if (producer_p)
    {
      size_t current_size = current_buffer_size[dest_id];
      if (current_size != 0)
      {
        P4A_sesam_wait_all_pages(dest_id, current_size, O_WRITE);
      }
      if (current_size !=0 && current_size != n)
      {
        sesam_free_data(dest_id);
        current_size = 0;
      }
      if (current_size == 0)
      {
        size_t i;
        sesam_reserve_data(n/sesam_get_page_size()+1);
        sesam_data_assignation(dest_id, n/sesam_get_page_size()+1, 1);
        P4A_sesam_wait_all_pages(dest_id, n, O_WRITE);
     }
    }
    current_buffer_size[dest_id] = n;
    *dest = sesam_map_data(dest_id);
  }
}
```

Figure 11: Par4All scmp run-time: allocation function

```
void P4A_scmp_reset()
{
  int i;
  for (i = 0; i <NB_BUFFERS; i ++)
    {
      current_buffer_size[i] = 0;
    }
}
```

Figure 12: Par4All scmp run-time: reset function

```
void P4A_scmp_dealloc(void *dest,
                      unsigned int dest_id, int do_it, int producer_p)
{
  if (do_it)
  {
    if (!producer_p)
    {
      size_t current_size = current_buffer_size[dest_id];
      P4A_sesam_send_all_pages(dest_id, current_size, O_WRITE);
    }
    sesam_unmap_data(dest);
  }
}
```

Figure 13: Par4All scmp run-time: deallocation function

```
void P4A_copy_to_accel_1d(size_t element_size,
                          size_t d1_size,
                          size_t d1_block_size,
                          size_t d1_offset,
                          const void *host_address,
                          void *accel_address,
                          unsigned int accel_address_id,
                          int do_it)
{
  if (do_it)
  {
    P4A_sesam_wait_all_pages(accel_address_id, element_size* d1_block_size,
                             O_READ);
  }
}
```

Figure 14: Par4All scmp run-time: Copying to a kernel task from a server task (kernel task version)

the kernel task and server task sides for one-dimensional arrays[3]. The server tasks copies its data in the buffer, and then releases the latter for read access; concurrently, the kernel task waits for the buffer to be ready for read access.

The kernel task version does not use its `accel_address` argument, which corresponds to the address of the original variable, leaving way for further elimination of the variable declaration if the function were inlined.

2.2.4 Copying from a kernel task to a server task

Figures 16 and 17 respectively give the functions used on the kernel task and server task sides for a communication from the kernel task to the server task. The kernel task releases the buffer for read access, while the server task waits for this release before copying back the buffer content into its own data.

2.2.5 Miscellaneous functions

Up to now, waits and sends have been performed through calls to `P4A_sesam_wait_all_pages` and `P4A_sesam_send_all_pages`. However, the Sesam hal divides shared buffers in *pages*

[3]Functions for two-dimensional arrays have also been implemented, and the run-time could easily be extended to cope with arrays of more dimensions.

```
void P4A_copy_to_accel_1d_server(size_t element_size,
                                 size_t d1_size,
                                 size_t d1_block_size,
                                 size_t d1_offset,
                                 const void *host_address,
                                 void *accel_address,
                                 unsigned int accel_address_id,
                                 int do_it)
{
  size_t i;
  if (do_it)
    {
      char * cdest = accel_address;
      const char * csrc = d1_offset*element_size + (char *)host_address;
      for(i = 0; i < d1_block_size*element_size; i++)
        cdest[i] = csrc[i];
      P4A_sesam_send_all_pages(accel_address_id, d1_block_size*element_size,
                               O_READ);
    }
}
```

Figure 15: Par4All scmp run-time: Copying to a kernel task from a server task (server task version)

```
void P4A_copy_from_accel_1d(size_t element_size,
                            size_t d1_size,
                            size_t d1_block_size,
                            size_t d1_offset,
                            void *host_address,
                            const void *accel_address,
                            unsigned int accel_address_id,
                            int do_it)
{
  if (do_it)
    {
      P4A_sesam_send_all_pages(accel_address_id, element_size*d1_block_size,
                               O_READ);
    }
}
```

Figure 16: Par4All scmp run-time: Copying from a kernel task to a server task (kernel task version)

```
void P4A_copy_from_accel_1d_server(size_t element_size,
                              size_t d1_size,
                              size_t d1_block_size,
                              size_t d1_offset,
                              void *host_address,
                              const void *accel_address,
                              unsigned int accel_address_id,
                              int do_it)
{
  int i;
  if (do_it)
  {
    P4A_sesam_wait_all_pages(accel_address_id, d1_block_size*element_size,
                              O_READ);

    char * cdest = d1_offset*element_size + (char *)host_address;
    const char * csrc = accel_address;
    for(i = 0; i < d1_block_size*element_size; i++)
      cdest[i] = csrc[i];
  }
}
```

Figure 17: PAR4ALL SCMP run-time: Copying from a kernel task to a server task (server task version)

```
static void P4A_sesam_wait_all_pages(unsigned int id, int size, int RW)
{
  int i;
  for (i = 0; i < size / sesam_get_page_size() + 1; i++)
    sesam_wait_page(id, i, RW, 0);
}

static void P4A_sesam_send_all_pages(unsigned int id, int size, int RW)
{
  int i;
  for (i = 0; i < size/sesam_get_page_size()+1; i++)
    sesam_send_page(id, i, RW);
}
```

Figure 18: PAR4ALL SCMP run-time: waiting and sending functions

of fixed size, and provides primitives to handle one page at a time. However, in our programming model, we always communicate the whole content of a buffer. This is why our run-time relies on the two previously mentioned functions (Figure 18), which, given the size of a buffer, respectively send or wait for all the pages of a buffer.

3 A real case study: Thales Sensing Application

The application developed by THALES for the SCALOPES project is "a sensing application that aims at detecting GSM signals within a wide band acquisition"[5]. Each iteration of the main loop goes through the following steps:

- Input: extraction of the buffered signal from the input file;
- Computation Step 1: transposition and wide band filtering;
- Computation Step 2: transposition and narrow band filtering;

- Computation Step 3: PSD computation (FFT)
- Output: write the results in an output file.

Computation Step 2 is itself composed of several transposition and filtering tasks embedded in two partially nested loops. Throughout the application, data flows from one (sub-)step to the next one. Each (sub-)step uses different data as input and output, making the application an ideal candidate for the SCMP architecture, provided that tasks are sufficiently well-balanced.

Some modifications were required to fit the SESAM simulator and PAR4ALL requirements. They are described in Section 3.1. Next, we describe the compilation output and some post-processing necessary to limit the memory footprint of the application. Then, we present some performance results.

3.1 Modification of the original application

First of all, we specialized the application to keep only the low sensitivity quarter band part, to meet the requirements of the SCALOPES project. We also removed arguments passed to the main program, and we removed all system calls, to timing functions for instance, because they are not currently supported by the SESAM HAL.

The SESAM API does not provide any dynamic allocation functions, apart from the allocation of shared buffers. On the other hand, PAR4ALL code generation for hybrid machines does not cope well with pointers [8]. The first modification was then to replace dynamically allocated arrays into stack allocated arrays, which was quite easy in this case. As a consequence, all allocating error testing code was removed.

One difficult part was to deal with the fact that the SESAM API does not currently support IOs, apart from the printing data to the standard output. A partial input data set and coefficient data sets were provided by the CEA in the form of header files declaring global arrays. The extraction of the input data and the reading of the coefficient files had to be modified to import their data from the global arrays. At the same time, we outlined these tasks into specific functions to ease their declarations as tasks.

Currently, our SCMP compiler only support tasks declared in the `main()` function. At first, we inlined the sole function called in `main()`, `processingLowSensitivity()`, at its call site and added labels to identify the different tasks. The resulting application is called Version 1. We did the same with `processingStep2LowSensitivity()` after testing Version 1, to check whether a finer task grain would be beneficial. The resulting application is called Version 2.

3.2 Compilation and Postprocessing

Our PAR4ALL SCMP compiler automatically generates all the tasks codes, including the control task description), places them in the right directorie, and also generates the `Makefile.arp` file to build the resulting application for the SESAM simulator. Some post-processing is performed semi-automatically by a PYTHON script to replace array declarations by one-element arrays in tasks where they are not needed. This could be done by a dead code elimination phase such as the one provided by PIPS, but we did not have time to test it.

For Version 1, 19 tasks are generated, including the start and end tasks, and 9 server tasks.

For Version 2, 31 tasks are generated, including the start and end tasks, and 12 server tasks.

3.3 Performance results

The performance results presented here are provided by Nicolas Ventroux from CEA.

A sequential version has been built by the CEA to serve as a reference, and ran on one processor on the simulator. All the concurrent versions have been run on 4 processors.

The CEA manually generated concurrent version runs 2.35 faster than the seuquential version.

Our Version 1 runs 1.20 faster. It is disavantaged by the fact that only one task is in charge of Computation Step 2, which constitutes a performance bottle neck.

Version 2, in which Computation Step 2 is splitted into several smaller tasks, runs 1.86 times faster, getting closer to the manually generated application despite the server tasks overhead. These encouraging results may be partly explained by the fact that a communication between two kernel tasks through a server task involves two shared buffers and the stack allocated copy of the original array on the server task. This almost amounts to using three buffers, except that we have to copy the buffered data to/from the stack allocated array. We have actually observed that the input data reading task is three iterations ahead of Computation Step 1 task. On the other hand, the manually generated application uses a double-buffering scheme for inter-task communications.

However, the run-time and memory footprint overheads both leave ways to further optimizations.

4 Optimizations and Enhancements

Along this report, we suggested several possible optimizations. They are discussed below, as well as new ones:

1. To reduce the memory footprint of each task, dead code elimination, including unused data declarations removal, would be necessary after inlining PAR4ALL functions.

2. Server tasks could be avoided, in particular in case of a pure streaming application in which entire arrays smoothly flow from a unique task to another. This case is easy to detect: each buffered array has a unique producer, and a unique consumer. The server task could then be declared as being the producer kernel task, and a special set of functions used instead of those provided in Section 2.2.

   However, it would still be necessary to copy the content of the temporary variables generated by `isolate_statement` into the original variable (for instance `a0` into `a`). This copy could be removed if the temporary variables have the same layout compared to the original variable in the producer and the consumer. But double buffers may be necessary to keep sufficient parallelism between tasks.

   These optimizations require that a task graph is available, whose edges are annotated with array regions corresponding to inter-task communications.

3. Allocations and communications should also be performed as soon as possible, as done in PAR4ALL for the optimization of communications for GPU [7]. For instance, if the allocated size is loop independent, the call to the allocation function could be moved outside. And if all iterations of a loop import values initialized prior to the loop, the communications can be moved outside the loop. This is the case in Thales application for the three coefficient arrays, which are all initialized before the main loop, and for which we generate communications inside the loop whenever a task requires them.

4. Current run-time does not support tasks which are at the same time producer and consumer of a given data. This case may arise in the original code, but also because

`isolate_statement` safely generates extra copies from the server to the kernel when the input data set cannot be accurately represented.

5. Tuning the stack sizes must currently be done manually, by running the application, and correcting them either by giving higher values in case of failures, or by lowering the values from the results of the simulation. Statically over-estimating stack usage would be a great help here. It's an untractable issue in the general case, but current limitations on programs handled by Sesam may render the problem solvable [1, 4, 3, 2].

5 Conclusion

In this technical report, we have presented our compilation process for the SCMP architecture, from C applications in which tasks are already identified. The performance results for a real sensing application from THALES have shown the relevance of the chosen approach. Further optimizations are proposed to lessen the performance gap between automatically and manually generated codes.

References

[1] AbsInt stack analyzer. `http://www.absint.com/stackanalyzer/`.

[2] Bound-T time and stack analyser. `http://www.bound-t.com/`.

[3] GNATstack. `http://www.adacore.com/home/products/gnatpro/add-on_technologies/stack_analysis`.

[4] TinyOS stack analyzer. `http://docs.tinyos.net/index.php/Stack_Analysis#Reducing_Stack_Memory_Usage`.

[5] Artemis Call 2008, SCALOPES: Sensing Application. Technical report, 2008.

[6] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Pierre Villalon. Pips is not (just) polyhedral software. Impact, 2011.

[7] Mehdi Amini, Fabien Coelho, François Irigoin, and Ronan Keryell. Compilation et optimisation statique des communications hôte-accélérateur. RenPar, 2011.

[8] François Irigoin Corinne Ancourt, Béatrice Creusillet and Ronan Keryell. Par4All & PIPS Programming rules. Technical report, HPC Project & Mines ParisTech, 2011.

[9] Serge Guelton, François Irigoin, and Ronan Keryell. Transformations source-à-source pour le calcul hétérogène. RenPar, 2011.