Hybrid Compile and Run-Time Memory Management for a 3D-Stacked Reconfigurable Accelerator

Lovic Gauthier
Department of Informatics
Kyushu University
744, motooka, nishi-ku,
Fukuoka, Japan 819-0395
lovic@ariake-nct.ac.jp

Shinya Ueno
Department of Informatics
Kyushu University
744, motooka, nishi-ku,
Fukuoka, Japan 819-0395
ueno@soc.ait.kyushu-u.ac.jp

Koji Inoue
Department of Informatics
Kyushu University
744, motooka, nishi-ku,
Fukuoka, Japan 819-0395
inoue@ait.kyushu-u.ac.jp

ABSTRACT

This paper presents a hybrid compile and run-time memory management technique for a 3D-stacked reconfigurable accelerator including a memory layer composed of multiple memory units whose parallel access allows a very high bandwidth. The technique inserts allocation, free and data transfers into the code for using the memory layer and avoids memory overflows by adding a limited number of additional copies to and from the host memory. When compile-time information is lacking, the technique relies on run-time decisions for controlling these memory operations. Experiments show that, compared to a pessimistic approach, the overhead for avoiding overflows can be cut on average by 27%, 45% and 63% when the size of each memory unit is respectively 1kB, 128kB and 1MB.

1 Introduction

Recently, many-core architectures are popular solutions for coping with the increasing demand of processing efficiency in embedded systems. Although such architectures do not overcome the memory wall limitation by themselves, the recent progresses in 3D stacking are promising for addressing this issue [8]. While not as popular, coarse-grain reconfigurable computing is still making progresses, and can be particularly effective for dataflow computations [5, 3].

In this context, we study the design of a 3D-stacked hybrid many-core and reconfigurable datapath accelerator chip made of 3-layers. The first layer is a many-core which communicates through a network-on-chip (NoC) using direct memory access units (DMA), the second is a memory layer and the third is made of small coarse-grain reconfigurable datapaths interconnected through interleaved oct-trees. This layered architecture is also decomposed into tiles: each core is on top of one memory tile (comprising four memory units), this latter being on top of one reconfigurable datapath. Figure 1 represents each layer for the case of an architecture comprising 49 tiles and whose datapath layer is an oct-tree of depth three.

We are currently working on a compiler for such a chip. It compiles C programs and converts their loops to sets of dataflow graphs and data transfers which are to be executed respectively by the datapaths and the DMAs. Future work will add the generation of parallel threads for the many-core layer.

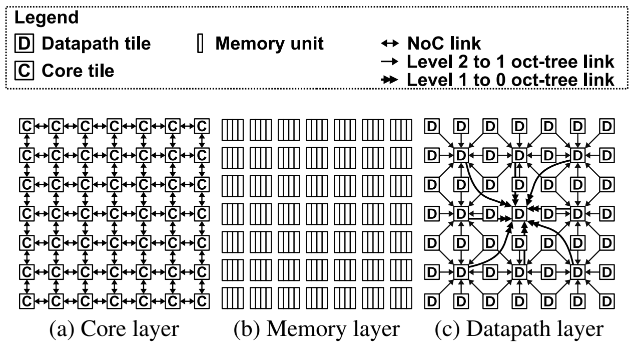


Figure 1: Two layers of the architecture

In this paper we present the technique used in the compiler for generating the memory management for the target accelerator. The proposed technique is hybrid since it inserts at compile time the allocation, the free and the data transfer operations, but leaves to the run-time their control when information is lacking. The main contributions are the following: first, the overhead of managing the multiple memory units is cut by treating them as if they were a single memory, i.e., allocating or freeing a block takes actually effect on all of them. Second, the technique generates several alternatives for managing the memory which are selected at run time depending on the current memory availability and requirements. Third, it performs a fine analysis of the accessed elements and makes use of the stride copies of the DMAs to avoid unnecessary data transfers.

The rest of the paper is organized as follows: first, section 2 gives some related works. Then, section 3 describes the architecture of the accelerator and section 4 presents the compiler. Then, section 5 gives an overview of the library used for controlling the chip, section 6 explains how the memory of the accelerator is managed and section 7 explains how this management is generated. Finally, section 8 presents some experiments and section 9 concludes this paper.

2 Related works

Various many-core architectures have been proposed [23, 12, 17]. 3D architectures are also starting to be explored for achieving faster and more energy-efficient memory accesses by stacking memory devices on top of the processors [20, 13], by benefiting from a better integration and by using various technologies in the same chip. While less considered, coarse grain reconfigurable architectures are still making progresses [5, 3]. The architecture proposed in [25] has some similarities with ours but is only 2D and do not separate dataflow (they say stream) computations from multi-processor ones.

978-1-4799-1400-5/13/$31.00 ©2013 IEEE

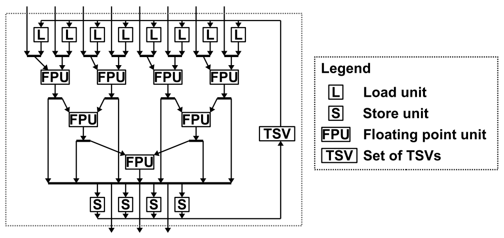


Figure 2: A datapth tile

There is a large amount of work for mapping C programs onto reconfigurable architectures. Several of them target instruction set extensions like [15, 4]. Recently, the main objective becomes the mapping of the sole loops for reducing the reconfiguration or initialization costs [1, 19, 21, 10, 9]. Our mapping approach too is dedicated to the loops. Yet the emphasis of this paper is the memory management and the data transfers. In this topic, a few research works about reconfigurable computing do introduce techniques for managing the memory and reducing the number of data transfers [22, 3, 29]. Still, the most active researches are about scratch-pad memory [18, 16, 27] or GPUs [2, 6]. These approaches usually support single or shared memory organizations, and have various contributions like compile-time or operating-system-based allocation and copy policies [18, 27, 29], new memory allocators [16], or schedule-based optimizations for reducing the cost of data transfers [22, 2, 6]. They are complementary to our approach since it supports run-time decisions and targets a distributed memory organization where the datapath tiles can only access their local memories.

3 The target architecture

Overview As mentioned in the introduction, the target architecture is made of three layers: an NoC-based many core layer, a memory layer and a datapath layer. This layered architecture is also decomposed into several tiles so that each core is on top of four memory units, these latter being on top of one reconfigurable datapath.

Inter-layer communication is provided by through silicon vias (TSV) so that each core and each datapath tile has a fast and low energy access to their corresponding memory tile. On the other hand, accessing memories of the other tiles requires to go through the NoC using direct memory access devices (DMA). There is one such DMA per core, each of them supports single stride access and is able to transmit one data (up to 64 bits) per cycle. In addition, all the DMAs can run in parallel.

Figure 2 gives an overview of one datapath tile: each of these tiles includes seven floating point units (FPU), eight load and four store units. In addition, each load and store unit has access to only one of the memory unit of the lower layer. Consequently, each memory unit is accessed by two load and one store units. Furthermore, each load and store unit supports single stride memory access.

The datapath tiles are then connected together according to interleaved oct-trees, so that large dataflows can be mapped. In the case of figure 1, there are 49 tiles which form a 3-level oct-tree whose root is the center. This structure allows a pipeline execution of the whole tiles as a single datapath, but also the independent pipeline execution of several groups of tiles.
N.B.: the memory management approach proposed in this paper is actually independent of the structure of the datapath layer. Hence we plan in future work to evaluate it on other architectures.

Controls The control of the chip and the execution of the sequential code are ensured by the core which is located into the center of the core layer. This core, called the *host*, is the only one to have access to the external memory. The other cores, the datapath tiles and the DMAs are controlled using registers accessible through the NoC. It is also possible for the host to control simultaneously several datapaths, cores and DMAs by sending broadcast messages among the NoC which update the relevant control registers. In return, each tile can signal through the NoC when its task completes. The configuration of each datapath tile is fetched from the corresponding memory tile, so that the reconfiguration time is actually very short.

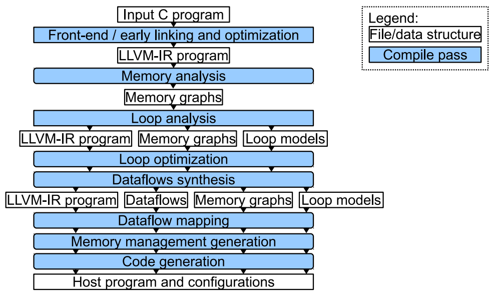


Figure 3: The compiling flow

4 The compiler

This section gives an overview of the full compiler in order to precise the context of the memory management generation technique presented in this paper.

The compiler is based on LLVM [11] a framework built upon an intermediate representation (IR) which is a single-static assignment (SSA)-based executable language. We also use Polly [26], an LLVM-based polyhedral optimizer, for analyzing and optimizing the loops.

The compiling flow is given in Figure 3. The input is a C program which can be composed of several files. The output is a single standalone IR file containing the compiled host program and all the additional information for the mapped loops and the data transfers. This IR file can then be converted to any assembly language using an LLVM-compatible code generator like the *llc* tool [14].

The compiling steps are as follows: first, standard steps of parallelizing compilers are applied. They include the front-end (with early link and sequential optimizations), the memory analysis (inter-procedural), and the Polly-based loop analysis and optimization. These two Polly-based steps generate polyhedral models for each loop and try to remove the control and inter-iteration dependencies which restrain the mapping onto the datapath layer. Since the memory and loop analysis are used in the work presented in this paper, they are detailed in section 7.2.
The following steps are dedicated to the mapping of loops onto the datapath layer and are subject to another paper. The *dataflow synthesis* step converts the body of the loops to dataflows and optimizes them. The conversion is straight forward since the IR already includes the data dependence links. The optimization consists mainly into balancing the datafows for shorter pipeline depth.
Then the *dataflow mapping* step maps the dataflows onto the datapath. First it maps a single instance of each of them individually using a branch and bound-based algorithm whose objective is to minimize the depth and width of the mapped dataflow. The optimal is quickly reached thanks to pre-computation tables: a first set of tables is used for constant-time mapping of sub-dataflows within each tile (small enough for complete enumeration of the possible cases), and another set is used for constant time routing among the tiles (the tree structure limits the number of possibilities). While for a single loop, minimizing the depth of the mapping is enough for optimal performance, when there are several loops, data transfer is also to be taken into account. For that purpose, a second pass tunes the mapping of the dataflows so that their inputs or outputs accessing common arrays are mapped to load or store units linked to identical memory unit. The tuning is performed at two level: by swapping load or store units within tiles and by swapping tiles. Finally, the throughput is increased further by duplicating each dataflow into several instances

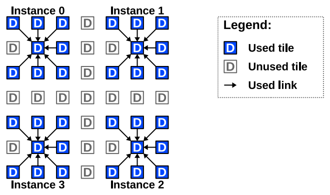
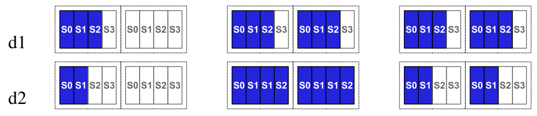


Figure 4: A dataflow parallelized with four instances



(a) Before (b) Bad parallel. (c) Good parallel.

Figure 5: Parallelizing dataflows

among the datapath layer. More precisely, assuming that a dataflow has d instances, their parallel execution allows to divide the number of iteration by d. For instance, in figure 4, four instances of a dataflow are used for the loop so that the number of iterations can be divided by four. The throughput gain of such a parallelization is however counter-balanced by the inter-loop communication. For instance, Figure 5 shows two cases of parallelization for two dataflows, d_1 and d_2, assumed to read the same array a. It is also assumed that dataflow d_1 is executed before dataflow d_2. In the figure, two datapath tiles are represented for each case, each tile being decomposed into four parts, called slices and noted respectively `S0`, `S1`, `S2` and `S3`. Each slice accesses a different memory unit and is colored when occupied. Figure 5a gives the mapping of the two dataflows before parallelization. In the case of Figure 5b, dataflow d_1 has two instances and dataflow d_2 has four. Since the accessed memories do not match, a has to be copied before executing d_2. By contrast, in the case of Figure 5c, both dataflows have two instances but they access the same memories so that no additional copy of a is required. Hence, the parallelization algorithm we designed for this step tries to limit the number of required copies by adjusting number of instances of each dataflow. Since the space to explore is huge, we consider a predefined set of distributions for the dataflows, each being identified by their number of instances. For example, the first distribution supports 196 instances whereas the last distribution supports only one instance. Currently, twelve different distributions are used.

5 The accelerator's library

A program for the accelerator is compiled to be executed by the sole host core. For that purpose, the compiler replaces the parts of the program which are to be accelerated (i.e., executed by the DMAs or the datapath) by calls to library functions dedicated to configuring, controlling and executing the accelerator. For this paper, the focus is on the memory-related functions described in table 1 (for sake of concision the `tdac_` prefix will be omitted in the rest the paper).

Table 1: Functions of the accelerator's library

`int tdac_mem()`	Returns the largest size which can be allocated into the memory layer.
`int tdac_mem_save(int size)`	Saves size bytes from the allocated space of the memory layer to the host memory. This operation is used to temporarily free space in the memory layer.
`int tdac_mem_restore()`	Restores to the memory layer the data previously saved by `tdac_mem_save`.
`int tdac_alloc(Buffer* buf,int size)`	Allocates or resizes a buffer in the memory layer.
`int tdac_free(Buffer* buf)`	frees a buffer from the memory layer.
`int tdac_copy(Buffer* s, int spd, int sst, Buffer* d, int dpd, int dst, int sz, int n)`	Copies `n` elements of size `sz` from buffer `s` to buffer `d` with respective pads `spd`, `dpd` and respective strides `sst` and `sdst`.

Table 2: The fields of the buffer structure

`dist`	the distribution of the buffer over the memory layer.
`mem`	the memory for the first instance of the buffer.
`size`	the size of the buffer (0 if not allocated).
`ad`	the address inside the memories.
`pad`	the starting offset from `address`.

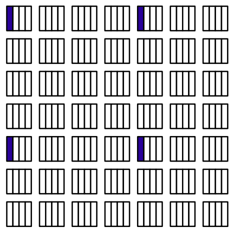


Figure 6: The memory units of a buffer whose distribution is 4 and initial memory is 0.

They operate on *buffer* objects which describe sets of memory blocks sharing the same address range but located in different memory units. These buffers are also used for setting up the inputs and the outputs of the dataflows (and, in the future, the threads) mapped onto the accelerators. The fields of the buffer objects are given in table 2. Among them, `dist` and `mem` give respectively the number of memories and the identifier of the first memory to access. As explained in section 4, there are actually predefined ways of duplicating dataflows over the datapath identified by the number of instances. This number is set to the `dist` field of the buffers. For instance, assuming a distribution of 4 (i.e., four blocks are used) and a memory of 0 (i.e., the first block is to allocated to memory 0), the resulting set of blocks is given in figure 6. Since, the distribution and the memory identifier are always used together, we call their pair a *geometry*. The other fields are used for the control at run time of the buffers.

The originality of these six functions is that they manage all the memories units simultaneously as if they were a single memory. For instance, `alloc(buf_a,32)` allocates 32B for each memory. The goal is to quicken the management of the multiple memories of the layer. This is based on the observation that an optimal usage of the architecture is achieved when the number of used tiles is maximized, which is what the compiler tries to achieve. Hence, a large part of the memory layer tends to be used for each mapped loop so that about the same amount of memory is used in each unit.

Regarding the implementation, the allocation and free functions utilize the host core only and keep the state of the memory locally (i.e., in the memory tile of the host core) whereas the data transfer functions make use of all the DMAs of the accelerator.

6 The memory management policies

In this section we present how the above-mentioned library functions are used for managing the memory.

6.1 Motivating example

Let us consider the C code given in figure 7a: this is a two-nested loop which put into two-dimensional (2D) array `c` the product of each element of 2D array `a` by the corresponding element of 2D array `b`. It is assumed that all the arrays contains 4-byte elements, and that the buffers corresponding arrays `a`, `b` and `c` are assigned to different memory units so that they can use the same address range.

```
for(j=0; j<M; ++j)
 for(i=0; i<N-1; ++i)
  c[j][i] = a[j][i]*b[j][i];
```

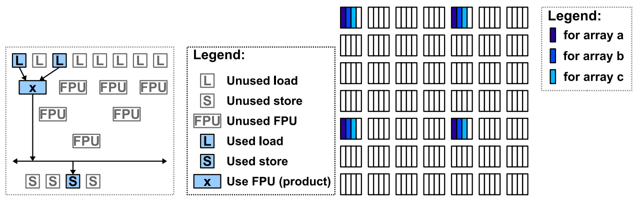
(a) Input C code

```
for(j=0; j<M; ++j)
  dp_run(dp0,N);

```

(b) Code after mapping

Figure 7: A nested loop



(a) Mapped dataflow (b) Used memory units

Figure 8: Memory units used by a mapped dataflow

```
1   alloc(buf_a,N*M*4);
2   buf_b->ad = buf_a->ad;
3   buf_c->ad = buf_a->ad;
4   dp_in(dp0,buf_a,buf_b);
5   dp_out(dp0,buf_c);
6   copy(a,buf_a,0,4,N*M);
7   copy(b,buf_b,0,4,N*M);
8   for(j=0; j<M; ++j)
9       dp_run(dp0,N);
10  copy(buf_c,c,0,4,N*M);
11  free(buf_a);
```

```
1   alloc(buf_a,N*M*4);
2   buf_b->ad = buf_a->ad;
3   buf_c->ad = buf_a->ad;
4   copy(a,buf_a,0,4,N*M);
5   copy(b,buf_b,0,4,N*M);
6   dp_in(dp0,buf_a,buf_b);
7   dp_out(dp0,buf_c);
8   for(j=0; j<M; ++j)
9       dp_run(dp0,N);
10  copy(buf_c,c,0,4,N*M);
11
```

(a) Single dataflow (b) a will be read afterward

Figure 9: Policy when there is no memory shortage

If the inner-most nest of the loop is mapped onto the datapath layer the resulting code will be like the one shown in figure 7b: only the outer nest remains (line 1), and the inner nest, including the products, is replaced by a call to a library function (line 2) which runs the datapath layer with a configuration named dp0 (which have been generated by the mapping step of the compiler) for N execution cycles. Actually, this code should be written in the IR language, but for sake of concision, the C idiom is used instead.

The code of figure 7b still lacks the memory management though. Assuming that the dataflow corresponding to dp0 has been mapped according to figure 8a and parallelized to four instances, the arrays will have to be distributed as shown in figure 8b: each array is split in four, each resulting subarray being assigned to a different memory.

With this configuration, the program including the most simple memory management is shown in figure 9a. First, before the loop can be executed, the buffers corresponding to the arrays have to be allocated and the input data copied from the host memory to the memory layer. The allocation is performed by lines 1-3 in the figure. Since the inputs and outputs of the dataflow are assigned to different memory units, only one buffer is actually to allocate (cf. section 5), this is the role of line 1. The other buffers are then set to share the same address space as buf_a by lines 2-3. Then, buf_a and buf_b are bound to the inputs of the datapath and buf_c to the output by lines 4-5. Finally the copies are performed by line 6-7.
N.B.: For readability, a simplified version of the copy function is used in the figures, it copies contiguous blocks of memory and requires five arguments only: the source and the destination buffers, the pad (for both buffers) the size and the number of elements.
From there the loop is executed (lines 8-9). This execution produces data into buf_c which are copied back to the host memory by line 10. Finally, since the buffers are of no use they can be freed, which is done at line 11 (only buf_a is freed since buf_b and buf_c share the same address space).

Now, if a further dataflow reads array a again with the same geometry, it is better to keep buf_a allocated so that an array transfer from the host memory can be skipped. Even when the further dataflow accesses a from a different geometry, it is still preferable to keep buf_a allocated, since a parallel copy among several memory units is faster than a copy involving the host. This later case is shown in figure 9b (the second loop is not represented but assumed to be present), where the free of the last line has been removed.
N.B.: since the assignment of the addresses of the buffers and their binding to the datapath are all similar, these operations will be omitted in the code examples given in the rest of the paper.

```
1   alloc(buf_a,N*4);
2   ...
3   for(j=0; j<M; ++j) {
4     copy(a,buf_a,j*N*4,4,N);
5     copy(b,buf_b,j*N*4,4,N);
6     dp_run(dp0,N);
7     copy(buf_c,c,j*N*4,4,N);
8   }
9   free(buf_a);
```

```
1   alloc(buf_a,N);
2   ...
3   for(j=0; j<M; ++j) {
4       for(k=0; k<4; ++i) {
5           copy(a,buf_a,j*N*4+k*N,4,N/4);
6           copy(b,buf_b,j*N*4+k*N,4,N/4);
7           dp_run(dp0,N/4);
8           copy(buf_c,c,j*N*4+k*N,4,N/4);
9       }
10  }
11  free(buf_a);
```

(a) Copies inside a nest (b) The mapped loop is split

Figure 10: Policies when the memory is too small

6.2 Dealing with memory shortages

When there is not enough memory for allocating new buffers, several techniques can be considered.

The simplest technique is to set the size of the buffers for the access range of one inner nest instead of the totality of the array. This technique is shown in figure 10a where the allocation of line 1 has a size of N*4 (corresponding to the access range of the inner-most nest mapped to the datapath) instead of N*M*4. Since the buffers are too small, the corresponding data transfers must be performed inside the upper nest instead of outside. That is, in the figure, lines 4-5 copy parts of arrays a and b from the host and line 7 copies a part of array c to the host[1]. This technique can be applied for any nest of a loop, an inner nest requiring smaller buffer than an upper one. This gain of memory comes with a cost though: first, the copy function have an important initial delay (due to the propagation through the NoC), and second, it prevents the reuse of the concerned buffers outside the nest as it is the case for buffers buf_a, buf_b and buf_c in the figure.

When the previous technique is not enough because the iteration ranges are too large, another technique is to split the nest mapped onto the datapath to match the available memory. This is shown in figure 10b where the datapath is executed four times so that only N bytes are required. For that purpose a loop nest is added (lines 4-9 in the figure) and the number of execution cycles of the datapath is divided (second argument of line 7 in the figure). The overhead of this technique is even higher than the previous one since a new nest is created which increases the number of times the datapath is executed.

Finally, if memory is lacking because of buffers previously allocated but not freed yet, the mem_save and mem_restore functions of the library presented in section 5 can be used.

6.3 The proposed management

For choosing which of the above techniques can be used it is necessary to known the size of the arrays. Since this is not always possible at compile time, our approach is to generate different versions for each loop and select which one should be executed at run-time when the required amount of memory is known, i.e., just before the loop is to be executed. Up to three versions are generated for each loop: for the first implementation (already given in figure 9a), the full arrays are allocated, for the second, the allocation is tried at each nest, and for the third, the inner-most nest is split.

The second implementation, shown in figure 11a, is a generalization of the example given in figure 10a: at each nest, the available memory and the state of the buffers (i.e., size of allocation) are checked and the allocations are performed accordingly. In the figure, the memory check is performed at the very beginning (line 1-2) and its result saved in variable sel which is then used for deciding the amount of memory to allocate (lines 3-6) and whether to copy outside the loop (lines 8-10 and 21-22) or inside it (lines 13-16 and 18-19).

The third implementation splits the inner-most (mapped) nest into div iterations[2] so that the buffer can fit into the memory. This

[1]The execution of the datapath also updates the pad of the buffers.
[2]Splitting means creating a new inner-most nest executed div times

```
1  int sel = (tdac_mem() < N*M*4) &
2            (buf_a.size != N*M*4);
3  if (sel)
4       alloc(buf_a,N*4);
5  else
6     alloc(buf_a,N*M*4);
7  ...
8  if (!sel) {
9       copy(a,buf_a,0,4,N*M);
10      copy(b,buf_b,0,4,N*M);
11 }
12 for(j=0; j<M; ++j) {
13   if (sel) {
14      copy(a,buf_a,j*N*4,4,N);
15      copy(b,buf_b,j*N*4,4,N);
16   }
17   dp_run(dp0,N);
18   if (sel)
19     copy(buf_c,c,j*N*4,4,N);
20 }
21 if (!sel)
22     copy(buf_c,c,0,4,N*M);
23 free(buf_a);
```

(a) Intermediate implementation

```
1  int div,rem,k,num;
2  int mem = tdac_mem();
3  int sz = mem;
4  if (sz<TH) {
5    mem_save(sz-TH);
6    sz = TH;
7  }
8  num = sz/4;
9  div = N/num;
10 rem = N%num;
11 alloc(buf_a,sz);
12 ...
13 for(j=0; j<M; ++j) {
14   for (k=0; k<div; ++k) {
15     copy(a,buf_a,j*N*4+k*N,4,num);
16     copy(b,buf_b,j*N*4+k*N,4,num);
17     dp_run(dp0,num);
18     copy(buf_c,c,j*N*4+k*N,4,num);
19   }
20   if (rem) {
21     copy(a,buf_a,j*N*4+k*N,4,rem);
22     copy(b,buf_b,j*N*4+k*N,4,rem);
23     dp_run(dp0,rem);
24     copy(buf_c,c,j*N*4+k*N,4,rem);
25   }
26 }
27 free(buf_b);
28 free(buf_c);
29 if (sz<TH)
30     mem_restore();
```

(b) Conservative implementation

Figure 11: Proposed implementations when memory is lacking

last implementation, given in figure 11b, is a generalization of the example given in figure 10b: it sets up at run time the buffer sizes and the number of times the datapath is to be executed. For that purpose it computes the number of execution cycles (`num`) and the number of times the datapath must be executed (`div`) from the available memory size (`sz`). In the figure, these computations correspond to lines 8-9, and the new iteration nest is in lines 14-19. Since the initial number of execution cycles (`N`) is not necessarily an exact multiple of `num`, the remainder `rem`, computed line 10, is used lines 20-25 for executing the datapath for the last cycles. Finally, if the remaining memory for allocation is very small, the number of times the datapath is executed can be too large for the overhead to be acceptable. In order to avoid that, a threshold is used (`TH` in the figure): when the available memory is smaller than this, a part of the memory layer is saved with function `mem_save` (lines 4-7), and the restoration is applied after the loop execution with `mem_restore` (lines 29-30).

These three implementations (or policies) are called respectively *greedy*, *intermediate* and *conservative*.

7 The generation of the memory management

This section describes how the compiler transforms the input code for supporting the memory management policies described in the previous section.

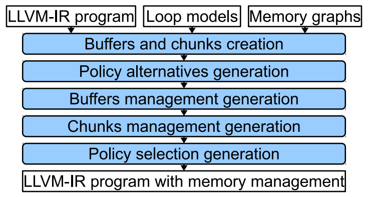

LLVM-IR program
Loop models
Memory graphs
Buffers and chunks creation
Policy alternatives generation
Buffers management generation
Chunks management generation
Policy selection generation
LLVM-IR program with memory management

Figure 12: The memory management generation flow

7.1 Overview

An array can be present into several memories, spread or duplicated among them, depending on the loops which access it. The copies of the arrays are represented both at compile and at run time by the buffer objects defined in section 5. By convention, one buffer is assigned per geometry of each array (i.e., per distribution and base memory used for accessing the array). When a loop is executed, the elements it accesses must be up-to-date inside the relevant buffers. For describing such valid or invalid states, the content of a buffer is described in the compiler as sets of elements called *chunks*. These chunks are built from the access patterns of the loops and are used for determining the elements which are to be copied among the memories but also the size of the buffers. For instance, considering the case of figure 11a, a chunk of `N` floats is copied from host buffer `a` starting from element `j*N` to buffer `buf_a`.

The generation flow consists then into determining the allocation size of each buffer and which chunks are to be transferred among them before executing each loop. This flow is given in figure 12. The input comprises the IR, the result of the memory and the loop analyses, and the datapath mappings (in future works, the results of threads' mapping on to the cores will also be used). The output is the IR updated with the memory management and data transfer functions.

The steps of the generation flow are detailed among the following sections but before, the memory and loop analysis of the compiler are presented in the next section since their results are important parameters for determining the memory management. In addition, section 7.3 gives details about how the chunks are computed.

7.2 Preliminary analysis

In addition to the input program (in IR form) and the dataflow mappings, the memory management generation requires information about the loops (iteration ranges) and the accesses (range, pattern and dependencies). This information has actually been produced by *memory analysis* and *loop analysis*, two early steps of the compiler which are also used for the loops optimizations.

Memory analysis:
In the LLVM-IR, the memory accesses are all represented by load or store instructions, but the accessed memory objects and their dependencies are not explicit. This step generates this information as dependence graphs among the sole load and store instructions. These graphs are used when optimizing the matching of the memories of the mapped loops and when generating the memory management.

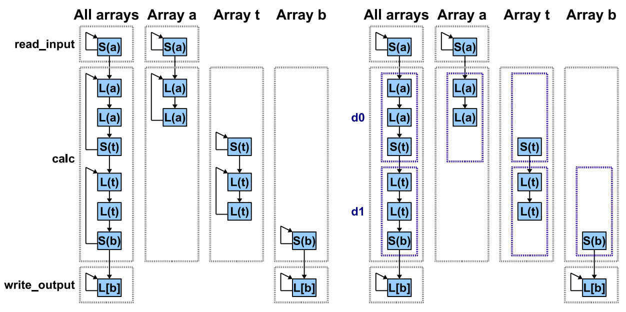
First, an alias analysis (i.e., analysis of the pointers of the program) is applied. Several algorithms exist for such an analysis, among them, we chose the Steensgaard's algorithm [24] which is one of the fastest (almost $O(S)$ where S is the number of statements) while still being safe since no alias is overlooked. Although often over-pessimistic, this algorithm is usually accurate with SSA-based representations like the IR we use [7]. The result of this analysis is a table, called *alias table* giving for each load or store instruction the list of the memory objects which may be accessed. From here, all the memory objects accessed through an index or a pointer are equivalent to arrays.

The next part of this step is the actual construction of the load and store dependence graphs. For our compiler, these graphs are the *memory graph* which gives the dependencies for all the memory

```
void calc(float a[M][N], float b[M][N]) {
    float t[M][N];
    int i,j;
    for(j=0; j<M; ++j) {
        for(i=0; i<N-1; ++i) {
            t[j][i] = a[j][i]*a[j][i+1];
        }
    }
    for(i=0; i<N; ++i) {
        for(j=0; j<M-1; ++j) {
            b[j][i] = t[j][i]*t[j+1][i];
        }
    }
}

void main(int argc, char* argv[]) {
    int i,j;
    float a[M][N], b[M][N];
    read_input(a);
    calc(a,b);
    write_output(b);
}
```

Figure 13: A C program whose loop can be mapped onto the datapath layer



(a) Before loop analysis (b) After loop optimization

Figure 14: Memory and array graphs

accesses and the *array graphs* which give such dependencies for each array taken alone. For all these graphs, the vertices represent the accesses to arrays and the edges represent dependencies (control and sequential) between them. For instance with the program of figure 13 the resulting graphs will be given in Figure 14a. After the loops being optimized (cf. section 4), the back edges of loops which do not have inter-iteration dependence are removed, as it is the case for loops $d0$ and $d1$ of Figure 14b.

Theses graphs are built from the LLVM-IR. First, the basic blocks are converted to sequences of memory accesses by removing all the instructions but the loads and the stores. Then, the empty blocks (which did not contained any load nor store instruction) are merged with their successor (if any) or predecessor. The resulting graph is the memory graph and from it, the array access graphs are extracted using the alias tables.

Loop analysis:
This pass identifies and analyses the loops of the program. It uses Polly [26], a framework which converts the loops it identifies to polyhedral models (or polytopes) from which dependencies can be computed and optimizations can be easily applied. In this framework, the loops are considered as polytopes whose dimensions are the numbers of nesting levels. For instance, a nest of two loops as in figure 7a will be converted to a 2D polytope. With this model the nesting structure of the loops is not relevant any more but instead, iteration ranges are modeled by integer bounds while memory accesses patterns and schedules are modeled by linear functions. After normalization, the loop indexes have all a stride of 1 and a lower bound equal to 0. For the memory management generation, only the access patterns are used, they are represented as follows:

$$\left(\sum_k i_k * S\right) + P \qquad 0 \leq i_k < N_k \tag{1}$$

In the equation, i_k represents the index variables, $[0, N_k)$ the corresponding iteration ranges, S_k the corresponding access strides and P the pad (i.e., the offset of the first accessed element). By convention, the k indices are enumerated in the order of the corresponding nests, i.e., the largest k stands for the inner-most nest.

7.3 Chunk arithmetic

Chunk arithmetic is used for computing the size of the buffers and the set of elements to copy among them. As explained earlier, a chunk is described like a memory access pattern, i.e., it is represented as a linear function whose variable can take any natural value among a given interval (a loop iteration range). In this section, a chunk c is represented as follows:

$$\left(\sum_k i_k(c) * S_k(c)\right) + P(c) \qquad 0 \leq i_k(c) < N_k(c) \tag{2}$$

This equation is identical to equation 1. The chunks have also an element size, noted E. For simplifying the explanations it is assumed that all the strides are positive (negative strides are supported and only require to invert the ranges when computing).

Since each buffer includes a set of chunks, its size can be computed as the distance between the chunk element with the lowest offset and the chunk element of the highest offset. This size can therefore be computed as follows (c represents a chunk of buffer B):

$$max_{c\in B}\left(\sum_k N_k(c) * S_k(c) * E\right) - min_{c\in B}(P(c)) \tag{3}$$

When the iteration ranges are not known at compile time, this expression is symbolic and inserted into the code after simplification. The size computed by equation 3 is used for the greedy policy, where the totality of the chunk is accessed. For the intermediate policy, the allocation is to perform for the parts accessed by inner nests only. Therefore, the indexes of the higher nests are omitted. For instance, for a buffer allocated from the second nest, the size will be as follows:

$$max_{c\in B}\left(\sum_{k>2} N_k(c) * S_k(c) * E\right) - min_{c\in B}\left(P(c)\right) \tag{4}$$

Multi-dimension chunks as described by equation 2 are difficult to transfer since the DMAs support only single-stride accesses. Our approach is therefore to simplify the chunks to linear functions with single indexes before using them. The simplifying procedure merges iteratively the totality of the indexes for the greedy policy and the indexes of the inner-most to the last upper nest for the intermediate policy. For merging two indexes i and j, several cases are considered. Each of them is described as follows and is correspondingly illustrated in figure 15 (it is assumed that the stride of i, i.e., S_i, is smaller than the stride of j):

1. If S_j equals $N_i * S_i$, the resulting access pattern is actually linear with one index so that the corresponding chunk does not need approximation: its stride and upper bound are respectively S_i and $N_j * N_i * S_i$.
2. If S_j is a multiple of S_i and $S_j <= N_i * S_i$, the resulting access pattern actually goes over same elements several times but its range is still linear with one index only so that the corresponding chunk does not need approximation: its stride and upper bound are respectively S_i and $N_i * S_i + (N_j - 1) * S_j$
3. If S_j is a multiple of S_i and $S_j > S_i * N_i$ the elements are all accessible trough a single linear index, but there are holes between each iteration of j. Our approach is to approximate the chunk by extending the iterations of i over the holes so that the result's respective stride and upper bound are S_i and $N_i*S_i+(N_j-1)*S_j$.
4. If S_j is not a multiple of S_i, the chunk is approximated to a contiguous set of elements: its index is E (the size of one element) and its upper bound is $N_i * S_j + (N_j - 1) * S_j$.

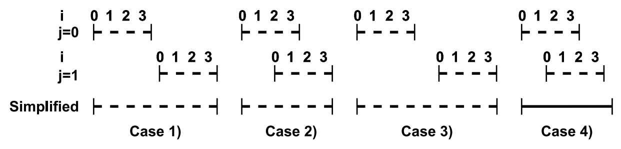


Figure 15: Chunk simplification cases: index i and j are merged (two iterations of j are shown, the dashes represent the accessed array elements)

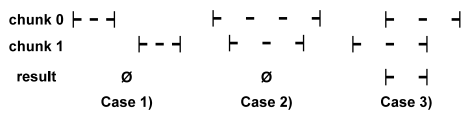


Figure 16: Chunks intersection cases

In practice, case 1 is the most frequent (it was the only encountered for our experiments), cases 2 and 3 happen is case of window-based computations, and case 4 is very rare.

It will be seen below that the approximated chunks cannot be used for intersections, hence when simplifying a chunk, all the merge operations which do not approximate are tried first. This allows to obtain non approximated chunks for case like the tiled loop.

The determination of the elements to copy among the buffers requires to intersect and subtract chunks as it will be seen in section 7.6. For simpler computations, two chunks c_0 and c_1 are considered for intersection only if they have the same element size (which is usually the case in practice) and if they have not been approximated, otherwise the intersection is set to empty. Indeed, when chunks are approximated, their intersection result might be wrong and could lead to use of elements which are not up-to-date. When not skipped, the result of the intersection is still empty for the two following cases ($S(c)$, $P(c)$ and $U(c)$ are respectively the stride the pad and the upper bound of a chunk noted c):

1. when the segments covered by the chunks do not overlap, i.e., when $P(c_0) > U(c_1)$ or when $P(c_1) > U(c_0)$.
2. when both chunks' iterations do not access any common element within the bounds of the chunks, i.e., when the following equation has no integer solution within the iteration ranges:

$$P(c_0) + S(c_0) * i_0 = P(c_1) + S(c_1) * i_1 \qquad (5)$$

This equation is actually a linear Diophantine equation (it can be reformulated as follows: $ai_0 + bi_1 + c = 0$), which is solved using the extended euclidean algorithm. Both cases are illustrated in figure 16 as case 1 and case 2.

Otherwise, the result is a chunk whose stride is the least common multiple (LCM) of $S(c_0)$ and $S(c_1)$, whose pad is $max(P(c_0), P(c_1))$, and whose upper bound is $min(U(c_0), U(c_1))$ (case 3 in figure 16).

Subtracting chunk c_1 from chunk c_0 is then straight forward and results into zero to four chunks depending on the LCM and the relative positions of c_0 and c_1 as it can be seen in figure 17.

The following sections detail each step of the memory management generation flow.

7.4 Buffers and chunks creation

This step first creates one buffer per geometry of each array. For simpler processing by the compiler, buffers are also created for the arrays stored into the host memory.

Second, buffers assigned to different memory units accessed by same dataflows are grouped. When grouped, buffers are set to use the same memory address range (but are still assigned to different memory units). It is for instance the case for arrays a, b and c in figure 9a (considering the mapping given in figure 8a). This saves memory because the allocation system is global as mentioned in section 5. For a given group, the memory size is obtained from the largest buffer and only one allocation is required.

Third, the step creates one chunk per loop memory access and one chunk per statically initialized array. The structure of each chunk is identical to the corresponding access pattern. Yet, the iteration ranges defining the chunks are not necessarily fixed at compile time: in such cases, symbolic expressions are used.

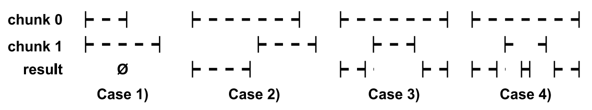


Figure 17: Chunks subtraction cases

```
 1 /* Before splitting */
 2 ...
 3 for(j=0; j<M; ++j)
 4     dp_run(dp0,N/4);
 5
 6 /* After splitting */
 7 ...
 8 for(j=0; j<M; ++j) {
 9   for(k=0;k<4;++k)
10     dp_run(dp0,N/4);
11   if (N%4>0)
12     dp_run(dp0,N%4);
13 }
```

(a) Compile-time split

```
 1 /* After splitting. */
 2 int mem = tdac_mem();
 3 int nm = mem/4>=2*N ? mem/4 : 2*N;
 4 ...
 5 for(j=0; j<M; ++j) {
 6   for(k=0;k<N/nm;++k)
 7     dp_run(dp0,nm);
 8   if (N%nm>0)
 9     dp_run(dp0,N%nm);
10 }
```

(b) Run-time split

Figure 18: Loop splitting for the conservative policy

7.5 Policy alternatives generation

This step generates up to three new code snippets for each loop, corresponding to the greedy, the intermediate and the conservative policies. Depending on the knowledge of the available memory, one or several snippets is inserted. For instance if it is known at compile time using the buffers size, that there will not be enough memory for a given greedy snippets, it is discarded.

The code of the greedy and intermediate policies is kept unchanged, but for the conservative case, the inner-most nest (which is mapped onto the datapath) is split according to the available memory. For that purpose, a new nest iterating div times (the number of split) is created, the dataflow execution instruction is moved in it, and the corresponding configuration set to divide the execution time by div (i.e., set to n/div if n is the initial number of execution cycles). If the array access ranges are all known at compile time, the split size is computed from the accessed chunks' size as shown in figure 18a. In the figure, the index of the new nest is `k` and the required size is `2*N` which corresponds to two chunks (it is assumed that `buf_a` and `buf_b` are grouped, but not `buf_c`)[3]. Otherwise the split size is determined at run-time as shown in figure 18b (from `nm` computed by line 3). For both cases, the number of split parts may not divide the number of iterations, hence an additional datapath run is inserted for the remainder (lines 11-12 in figure 18a.)

7.6 Management instructions insertion

Now that the policy code snippets of the loops are generated, the operations performing the management must be inserted. They include the allocation and free of the buffers and the copies of the chunks.

Buffer management, insertion of allocation and free instructions This step inserts allocation and free instructions for one buffer per group. For the remaining buffers, it is enough to update their address after the allocation as it can be seem in figure 9a where only `buf_a` is allocated. Depending on the policy, the size of the allocation and the location of the insertions are computed differently.

For the greedy policy, the allocation size for a group is fixed and is the size of the largest buffer computed as explained in section 7.3. When the sizes of chunks are not known at compile time, the symbolic expressions given by equation 3 are used after simplification (e.g., in figure 9a the allocation expression is `N*M*4`). For each group of buffer the allocation instructions are placed just before the

[3]N.B.: the allocations, copies and frees are not generated yet.

```
int mem = tdac_mem();
int sz = M*N*4*2;
if (mem>=sz) {
   alloc(buf_a,M*N*4);
   alloc(buf_c,M*N*4);
} else {
   alloc(buf_a,N*4);
   alloc(buf_c,N*4);
}
...
for(k=0;k<L; ++k) {
   if (mem>=sz) {
      copy(a,buf_a,4,N*M);
      copy(b,buf_b,4,N*M);
   }
   for(j=0; j<M; ++j) {
      if (mem<sz) {
         copy(a,buf_a,4,N);
         copy(b,buf_b,4,N);
      }
      dp_run(dp0,N);
      if (mem>=sz) {
         copy(buf_c,c,4,N);
      }
   }
   if (mem>=sz) {
      copy(buf_c,c,4,N*M);
   }
}
```

Figure 19: Checking the available memory before allocating

corresponding loop. In case of a reused buffer, only the first loop accessing the corresponding array has an allocation. Such loops are found using the array graphs described in section 7.2. If it is not known at compile time which loop is executed first, allocation instructions are inserted for all the candidates since these instructions can be applied several times on a same buffer (they have both the allocation and resize functionality as mention in section 5). Similarly each free instruction is inserted after the last loop (executed on the datapath) accessing the buffer to be freed.

For the intermediate case, the procedure is similar but applied on each nest of each loop, the sizes being computed using the indexes of the inner loops as explained in section 7.3 with equation 4. For instance, in figure 10a, the allocation is only `N` elements instead of `N*M`. The allocation and free instructions are inserted within the treated nest, but further LLVM optimizations may put them outside the nest as it is the case of the figure. When there are more than two levels of nest (including the one mapped onto the datapath layer), several allocations are inserted for each of them. They are executed only if they do not require too much memory and if an allocation was not performed in an upper nest. Hence memory checks are inserted: they verify if the available size is smaller than the one required for the upper nest but large enough for the current nest. Figure 19 gives such an example where it is assumed that `buf_a` and `buf_b` are in the same group but `buf_c` is in a different group[4].
N.B.: when several loops are within a same nest, buffer reuse can be applied for them similarly to the greedy policy.

For the conservative case, the allocation instructions are inserted for the inner-most nests only. The sizes are obtained directly by dividing the available memory by the number of arrays to allocate since the inner-most nests are split. In addition, memory save instructions are inserted just before the computation of the allocation parameters if the available memory is lower than a predefined threshold. The corresponding memory restore instructions are inserted just after the loops (cf. figure 11b).

Finally, regarding the buffer reuse, the conflicts between the greedy policy and the others are avoided by adding guards checking the state of the buffers susceptible to be reused before allocating them.

Chunks management, insertion of copy instructions The chunk copies are placed before each loop for the greedy policy, inside each nest for the intermediate policy, and within the split loop for the conservative policy. For the intermediate policy, similarly to the buffer allocation, the chunk copies for each nest are to be performed only if the size fits.

Algorithm 1 Generation of chunk copies

```
function MAKE_COPIES($v$,$b$,$cs$)
    if ($v$.visited) then return
    end if
    $v.visited$ = true
    if then$v! = b$
        $inter \leftarrow cs \cap v.chunk$
        if !APPROXIMATED($inter$) then
            if $v.write$ then
                GENERATE_FROM_HOST($v$,$b$)(return )
            end if
        end if
        generate_copy($v$,$b$,$inter$)
    end if
    for all $pv \in v.previouses$ do
        MAKE_COPIES($pv$,$b$,$cs - inter$)
    end for
end function
```

The main task of this step is to calculate the chunks which are to be copied. These chunks correspond to the parts of the arrays that are not up-to-date when read by a loop. They are obtained for each array by subtracting the access ranges (represented as chunks too) of the successive loops provided they are assigned to the same memory. Otherwise, a write access forces the full range to be copied, and a read access can be ignored. Concretely, for each read access of each loop, the corresponding array graph is traversed backward, i.e., the successive predecessors are visited. For that purpose, function MAKE_COPIES given in algorithm 1 is called on each access to treat (noted b) as follows: MAKE_COPIES$(b, b, b.chunk)$. The arguments are the current access v, the access to treat b, and the chunks to treat cs. For each previous vertex in the array graph, a copy is generated from the intersection of the current chunks cs and the new one $v.chunk$ provided the result, *inter* in the algorithm, is not approximated as defined in section 7.3. Otherwise, function GENERATE_FROM_HOST generates copies from the host and ends the recursion. Finally, the function recurses on the previous vertices of v using the chunks obtained by subtracting chunk *inter* (which does not need to be copied any longer) from each chunk of cs.

The host copies inserted when no intersection is found are the followings: when the previous access was a write, a copy of the corresponding datapath result is inserted into its nest and a similar copy from the host is inserted just before the datapath execution of the current vertex. This latter copy is the only one to insert when the previous was a read.

In the case where several paths lead to different loops, flags are used to know which one has been executed for performing the corresponding copies.

When a buffer contains only parts of chunks, i.e., for the intermediate and conservative policies, two cases are considered: if both accesses are within the same nest, the algorithm is applied as is. For the second case, host copies are inserted.

7.7 Policy selection generation

This last step generates the policy selection procedure. This procedure compares for each loop the available memory with the amount required by each policy and selects the fastest policy which does not overflow. The memory requirement of a policy is computed from the sizes of its buffers as follows:

$$\sum_{g \in G} max_{b \in g}(size(b)) \qquad (6)$$

In the equation, G is the set of the groups for a loop, b a buffer of group g and $size(b)$ its size. When the sizes of some buffers are not known at compile time, the comparison is generated using the corresponding symbolic expressions. Figure 20 illustrates such a case

[4]The copy shown in the figure are discussed in the next section.

```
int mem = tdac_mem();
if(mem>=max(O*M*N*4,O*L*N*4) {
  /* Fast implementation. */
  alloc(buf_a,O*M*N*4);
  alloc(buf_d,O*L*N*4);
  ...
  copy(a,buf_a,4,O*M*N);
  copy(d,buf_d,4,O*L*N);
  for(k=0;k<O;++k) {
    for(j=0;j<M;++j) {
      run(dp2,N);
    }
    for(j=0;j<L;++j) {
      run(dp3,N);
    }
  }
} else
if (mem>=N*4) {
  /* Intermediate implementation. */
  ...
} else {
  /* Safe implementation. */
  ...
}
```
Figure 20: Policy selection

for a loop including two dataflow computations (i.e., two inner-most nest mapped onto the datapath layer) where the sizes of `buf_a` and `buf_d` are not known. It is also assumed that these two buffers are not grouped together. For concision, several part of the code, including the policies details are omitted.

8 Experiments

8.1 Experimental environment

We simulated an accelerator including 49 tiles through a cycle accurate simulator we designed. It is able to simulate the control and the execution of both the datapath layer and the DMA-based data transfers among the memories, and the behavior of the NoC.

For the applications, we used kernels from the polybench-3.2 benchmark [28] (*2mm*, *adi*, *atax*, *bicg*, *covariance*, *doitgen* and *syr2k*), selected for being different enough from each other, and one kernel designed internally for an automotive image processing application (*gaussian*). The execution of the kernels have been simulated using three memory management policies: *optimistic* is the policy where the arrays are all fully copied to the required memory once for all at the beginning of the program and is the common approach for managing accelerators' memory (i.e., it is up to the application programmer to avoid memory shortage), *pessimistic* is the default safe policy which allocates the minimal size for each array, splits the loops to match this size and always copies the data from the host memory before executing a dataflow then copies them to the host after this execution. Finally ours is the hybrid policy we propose in this paper. When all the arrays can be allocated to memory layer, ours is close to optimistic, and when all the loops must be split to fit into the memory layer, ours is close to pessimistic. Nonetheless, the greedy, intermediate and safe policies used in our approach are applied at the loop level and not the totality of the program as it is the case for optimistic and pessimistic.
Regarding the host code size, ours was on average 133% and 51% larger than respectively optimistic and pessimistic. This is significant, however only the host code is increased, the remaining code, i.e., the configuration data, remains unchanged.

8.2 Experimental results

Figure 21 gives for each kernel mapped onto the datapath layer the execution time of the pessimistic policy (*pess.*) and ours (*ours*) normalized with the optimistic policy for 1kB, 128kB and 1MB per memory unit. This corresponds respectively to 196kB, 24.5MB and 196MB for the total memory layer. In reality, this optimistic policy fails in most of the cases due to memory shortages: it is only with 1MB per memory unit that adi, atax, bicg, doitgen and gaussian do not have such shortages. These optimistic results with 1MB per memory unit are the one used for the normalization in the figure. The execution times are decomposed into computation time (including the host execution time, e.g., the memory management overhead) and data transfer time (for the data transfers among the memories).

The most notable result is that the pessimistic approach suffers from very important data transfer overhead. This is because no buffer can be shared among different dataflows and the host memory is always used as intermediate. By contrast the proposed approach allows to save some expensive memory transfers with only 1kB per memory unit for all the kernels but syr2k and adi. On average, compared to the pessimistic approach, our approach reduces the execution time by 27%, 45% and 63% for respectively 1KB, 128KB and 1MB per memory unit. For doitgen ours is particularly efficient when there is 128KB or more per memory unit. This is because this application includes loops with three levels of nest (instead of two for the other applications) so that the intermediate policy performs very well. Finally, it can be observed, that for most applications, a very large memory size is required for the optimistic policy. The computation time can also vary significantly, but not as much as the data transfer time. This increase in time is mainly due to the initializations of the datapath executions: the smaller the memory, the more loop splits are necessary, and the higher number of datapath initializations are required. The data transfer initializations too contribute a little to this slowdown, and they also affect our policy.

9 Conclusion

In this paper we presented a C compiler targeting the datapath layer of a 3D reconfigurable accelerator with purpose to introduce a new hybrid technique for managing the allocations, frees and data transfers for the memory layer. This technique uses a global allocation and free system for coping with the multiple memory units of the layer, relies on run-time decisions for controlling the allocations, frees and data transfers, and limits the number of data transfers with the host memory required for avoiding memory overflows. Experimental results show that without any management, memory overflow quickly happens, and that the overhead induced by a default safe management if very high. Using the proposed management, the execution is still safe, but the execution is on average faster by 27%, 45% and 63% with respective memory sizes of 1kB, 128kB and 1MB than for the default safe one. As yet, this technique has only been tried on one datapath layer is used but it is also applicable any other dataflow-based datapath architecture and for multi-thread execution on the many-core layer. Hence, we plan as future work to evaluate its performance on other architectures. In addition, a better memory usage should be achievable if memories of the memory layer are managed individually instead of globally as it is the case currently, provided techniques are found for coping with the additional overhead.

Acknowledgments

This research was supported in part by New Energy and Industrial Technology Development Organization (NEDO) and Japan Science and Technology Agency (JST), CREST.

References

[1] F. Barat, M. Jayapala, P. Op de Beeck, and G. Deconinck. Software pipelining for coarse-grained reconfigurable instruction set processors. ASP-DAC '02, pages 338–. IEEE Computer Society, 2002.

[2] M. Bauer, H. Cook, and B. Khailany. Cudadma: optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 12:1–12:11, New York, NY, USA, 2011. ACM.

[3] J. a. M. P. Cardoso, P. C. Diniz, and M. Weinhardt. Compiling

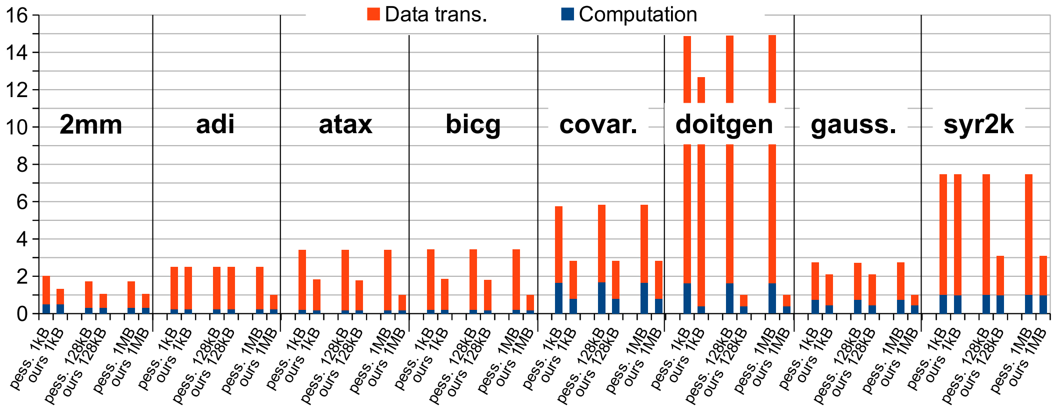


Figure 21: Execution time normalized with the optimistic case (1 means no overhead)

for reconfigurable computing: A survey. *ACM Comput. Surv.*, 42(4):13:1–13:65, June 2010.

[4] Z. Chen, R. N. Pittman, and A. Forin. Combining multicore and reconfigurable instruction set extensions. FPGA '10, pages 33–36. ACM, 2010.

[5] C. Galuzzi and K. Bertels. The instruction-set extension problem: A survey. *ACM Trans. Reconfigurable Technol. Syst.*, 4(2):18:1–18:28, May 2011.

[6] C. Gou and G. N. Gaydadjiev. Elastic pipeline: addressing gpu on-chip shared memory bank conflicts. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 3:1–3:11, New York, NY, USA, 2011. ACM.

[7] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. *SIGPLAN Not.*, 33(5):97–105, May 1998.

[8] P. Jacob, A. Zia, O. Erdogan, P. Belemjian, J.-W. Kim, M. Chu, R. Kraft, J. McDonald, and K. Bernstein. Mitigating memory wall effects in high-clock-rate and multicore cmos 3-d processor memory stacks. *Proceedings of the IEEE*, 97(1):108 –122, jan. 2009.

[9] C. Jang, J. Kim, J. Lee, H.-S. Kim, D.-H. Yoo, S. Kim, H.-S. Kim, and S. Ryu. An instruction-scheduling-aware data partitioning technique for coarse-grained reconfigurable architectures. LCTES '11, pages 151–160. ACM, 2011.

[10] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek. Memory access optimization in compilation for coarse-grained reconfigurable architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 16(4):42:1–42:27, Oct. 2011.

[11] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. CGO '04, pages 75–. IEEE Computer Society, 2004.

[12] F. Lemonnier and P. Millet. Flextiles: self adaptive heterogeneous manycore based on flexible tiles (fp7 project). INA-OCMC '12, pages 37–38. ACM, 2012.

[13] C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari. Bridging the processor-memory performance gap with 3d ic technology. *Design Test of Computers, IEEE*, 22(6):556 – 564, nov.-dec. 2005.

[14] LLVM. llc. `llvm.org/docs/CommandGuide/llc.html`.

[15] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot. Constraint programming approach to reconfigurable processor extension generation and application compilation. *ACM Trans. Reconfigurable Technol. Syst.*, 5(2):10:1–10:38, June 2012.

[16] R. McIlroy, P. Dickman, and J. Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 31–40, New York, NY, USA, 2008. ACM.

[17] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications. DAC '12, pages 1137–1142. ACM, 2012.

[18] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *ACM Trans. Embed. Comput. Syst.*, 8(3):21:1–21:32, Apr. 2009.

[19] T. Oh, B. Egger, H. Park, and S. Mahlke. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. *SIGPLAN Not.*, 44(7):21–30, June 2009.

[20] Y. Pan and T. Zhang. Improving vliw processor performance using three-dimensional (3d) dram stacking. ASAP '09, pages 38–45. IEEE Computer Society, 2009.

[21] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. PACT '08, pages 166–176. ACM, 2008.

[22] M. Sanchez-Elez, M. Fernandez, M. Anido, H. Du, N. Bagherzadeh, and R. Hermida. Low energy data management for different on-chip memory levels in multi-context reconfigurable architectures. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10036–, Washington, DC, USA, 2003. IEEE Computer Society.

[23] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, Aug. 2008.

[24] B. Steensgaard. Points-to analysis in almost linear time. POPL '96, pages 32–41. ACM, 1996.

[25] M. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, and J. Kim. Evaluation of the raw microprocessor: an exposed-wire-delay architecture for ilp and streams. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 2 – 13, june 2004.

[26] G. Tobias, Z. Hongbin, A. Raghesh, S. Andreas, G. Armin, and P. Louis-Noël. Polly - polyhedral optimization in llvm. Apr. 2011.

[27] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, May 2006.

[28] O. S. University. the polyhedral benchmark suite. `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`.

[29] M. Vuletić, L. Pozzi, and P. Ienne. Virtual memory window for application-specific reconfigurable coprocessors. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 948–953, New York, NY, USA, 2004. ACM.