Distributed Memory Allocation Technique for Synchronous Dataflow Graphs

Karol Desnos, Maxime Pelcat, Jean-François Nezan, Slaheddine Aridhi

Distributed Memory Allocation Technique for Synchronous Dataflow Graphs

Karol Desnos, Maxime Pelcat, Jean-François Nezan
IETR, INSA Rennes, CNRS UMR 6164, UBL
email: kdesnos, mpelcat, jnezan@insa-rennes.fr

Slaheddine Aridhi
Texas Instruments France
email: saridhi@ti.com

***Abstract*—This paper introduces a new distributed memory allocation technique for applications modeled with Synchronous Dataflow (SDF) graphs. This technique builds on a State-of-the-Art shared memory allocation technique based on a weighted graph, called Memory Exclusion Graph (MEG). A MEG captures the memory reuse opportunities between memory objects that must be allocated before the execution of an SDF graph. The algorithms detailed in this paper enable a single MEG to be split into separate MEGs, each of which is associated with a memory bank accessible only by one core of the architecture. The proposed technique is implemented within a rapid prototyping framework and is evaluated by deploying real computer vision applications on a Multiprocessor System-on-Chip (MPSoC). Results show a systematic performance improvement due to better memory usage, with application speedups ranging from 2% up to 380%.**

I. Introduction

In 1995, Wulf and McKee [17] introduced the "memory wall" problem which identifies the low rate of improvement in memory bandwidth as a major concern for developers of embedded systems. Since then, industrial efforts to keep up with Moore's Law has led to the evolution of architectures that are increasingly more parallel with soon thousands of processing elements embedded into a single chip [15]. As a result of this multi- and manycore trend, distributed memory architecture has been widely adopted by processor manufacturers to avoid hitting the memory wall.

An application specified with a dataflow graph [8] consists of a set of processing entities, named actors, connected by a set of First-In First-Out queues (FIFOs) transmitting data quanta, named data-tokens, between actors. An actor starts its preemption-free execution (that is, it fires) when its input FIFOs contain enough data-tokens. When deploying an application specified with a dataflow graph on a multicore architecture, the primary task of memory allocation is to assign an address range of memory to store each data token that transit through the FIFOs of the dataflow graph.

The contribution of this paper is a novel compile-time memory allocation technique for dataflow graphs which supports distributed memory architectures, shared memory architectures, and architecture combining both shared and distributed memory resources. This memory allocation technique builds on memory minimization techniques for shared memory architectures published in [5], [4]. From these shared memory techniques, the new technique inherits memory reuse capabilities at graph level [4], compatibility with buffer merging for memory reuse between different FIFOs at actor level [5], and code generation supporting cache incoherent architectures [4].

The context of this work is presented in Section II. Section III contains a set of memory reuse techniques used as a basis for the new distributed memory allocation technique, which is then detailed in Section IV. Finally, an experimental comparison of shared and distributed allocation techniques for the deployment of two image processing applications on an Multiprocessor System-on-Chip (MPSoC) is presented in Section V.

II. Related Work

A. Synchronous Dataflow (SDF)

1) The Semantics of the SDF Model of Computation (MoC): The SDF Model of Computation (MoC) chosen for this study is among the most commonly used dataflow MoCs. The SDF MoC is a specialization of the dataflow MoC where token production and consumption rates of actors set by actor firing rules are fixed scalars and are known at compile-time [7]. This property makes it possible to analyze SDF graphs during their compilation. Static analysis can be used to check consistency and schedulability properties that imply deadlock-free execution and bounded FIFO memory needs. If an SDF graph is consistent and schedulable, a fixed sequence of actor firings, called iteration, can be repeated indefinitely to execute the graph [8].

Formally, an SDF graph $G = \langle A, F \rangle$ is a directed graph containing a set of actors A that are interconnected by a set of FIFOs F. $prod(f)$ and $cons(f)$ denote the actors in A producing and consuming tokens respectively on a FIFO f. For each FIFO $f \in F$ connected to an actor $a \in A$, a data rate is specified by the function $rate : A \times F \rightarrow \mathbb{N}^*$.

On the SDF graph presented in Figure 1, expressions displayed adjacent to the data ports are token production and consumption rates on the connected FIFO. Expressions depend on static parameters w and h, the image width and height respectively, and on n, the number of slices into which the image is split to provide parallel processing.

2) Memory Footprint Minimization in the Literature: Minimization of the memory footprint allocated for the deployment of SDF graphs has been the subject of many publications, both for mono-core [11], [16] and multicore architectures [4], [9]. Minimizing the memory footprint of dataflow applications is usually achieved by using FIFO dimensioning techniques [1], [11], [12], [16]. FIFO dimensioning consist of finding an execution order of the graph actor, called a schedule, that minimizes the memory space allocated to each FIFO of the SDF graph. A disadvantage of these techniques is that memory reuse between FIFOs is not considered [1], [12], [16], or only single-core architectures are considered [11]. The proposed distributed memory minimization technique is based on a memory allocation model that fosters memory reuse between several FIFOs.

Some publications tackle the issue of memory allocation of SDF graphs in a distributed memory context [2], [6], [9]. The technique presented in [6] analyzes the scheduling of inter-processor communications in order to allocate buffers of communications which occur simultaneously in different memory banks, thus constituting a shared memory space. Although this technique targets shared memory architectures, its consideration of several memory banks makes it similar to an approach for distributed memory. In [2], a FIFO sizing technique is applied in a distributed memory context with limited memory resources. This allows FIFO sizing to be used as a buffering technique for inter-processor communications. The results published in the study focus on application latency and throughput under

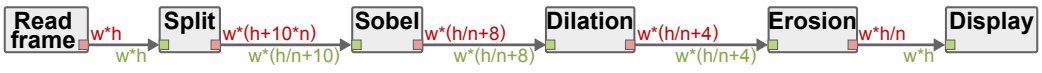


Fig. 1. Synchronous Dataflow (SDF) graph of the Sobel application

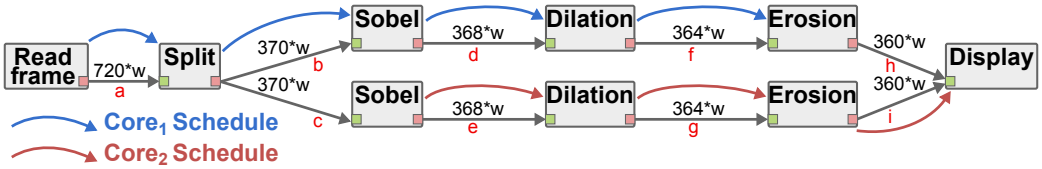


Fig. 2. Single-rate SDF graph of the Sobel application from Figure 1 for a 720p resolution ($h = 720$, $w = 1280$, and $n = 2$). Blue and red arrows depict a schedule of the SDF graph on an architecture with two cores.

memory constraints. In [9], authors introduce a mapping technique for SDF actors on a distributed memory architecture with constrained memory resources. The memory-constrained mapping problem is then formulated as an integer linear program of polynomial complexity. The commonality of all these techniques is that none consider memory reuse between different FIFOs. Memory reuse is the rationale behind the proposed study.

III. Existing Memory Reuse Framework

The distributed memory allocation technique used in this paper consists of the following steps, of which Step 5 (Section IV) constitutes the novelty of this paper:

1) Convert the original SDF graph to single-rate SDF graph.
2) Build a Memory Exclusion Graph (MEG) (Section III-A).
3) Update the MEG with scheduling information [4].
4) Apply buffer merging to the MEG. (Section III-B)
5) Create a specific MEG for each memory component.
6) Allocate each specific MEGs to its memory component.
7) Generate code with explicit cache management [3], [4].

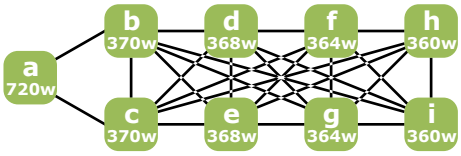
A. Memory Exclusion Graph (MEG)

Before allocating an SDF graph in distributed memory, transformations are applied to reveal its parallelism and memory characteristics.

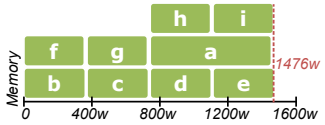
The first step of the proposed technique is the transformation of the original SDF graph into an equivalent single-rate SDF graph where each original FIFO is replaced with one or several single-rate FIFOs whose consumption and production rates are all identical [14]. Assuming that executions of successive graph iterations never overlap, each single-rate FIFO is thus a buffer of fixed size accessed by two actors. Figure 2 presents the single-rate SDF graph derived from the original SDF graph from Figure 1. Replacing original multi-rate FIFOs with single-rate buffers yields many advantages [3]: FIFO management overhead is removed; actor code does not require push or pop primitives as pointers to single-rate buffers are sufficient; and lifetimes of single-rate buffers span from the start of their producer actor execution, to the completion of their consumer actor.

The second step of the proposed technique consists of building an undirected graph, called Memory Exclusion Graph (MEG), whose weighted vertices are the memory objects that must be allocated in memory to support the execution of the application [4]. Memory object is a term used variously to designate a single-rate buffer, an initial token, or a working memory used by an actor during its execution. In a MEG, two memory objects are connected with an edge called exclusion if they can not be allocated in overlapping memory ranges. An exclusion is added between two memory objects if they may store valid data simultaneously. An algorithm which constructs a MEG from a single-rate SDF graph can be found in [4].

Formally, a Memory Exclusion Graph (MEG) is an undirected weighted graph denoted by $MEG = \langle M, E, w \rangle$ where: M is the set of memory objects. $m(f) \in M$ and $m(a) \in M$ are the memory objects associated with the single-rate FIFO $f \in F$, and the working memory of actor $a \in A$, respectively. E is the set of memory exclusions. $w : M \rightarrow \mathbb{N}^*$ is a function where $w(m)$ is the size (in bytes) of a memory object m.



(a) Memory Exclusion Graph (MEG)



(b) Memory Allocation

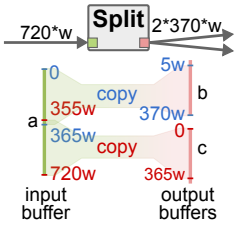
Fig. 3. MEG derived from the SDF graph of Figure 2

Figure 3a shows the MEG corresponding to the single-rate buffers contained in the SDF graph of Figure 2. Working memory of actors has been omitted for conciseness. Figure 3b presents a valid memory allocation derived from the MEG of Figure 3a.The horizontal axis represents addresses of shared memory, and vertical axis is used to denote reuse of memory ranges to store several memory objects.

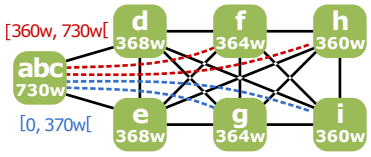
B. Buffer Merging

The memory reuse opportunities modeled in MEGs result from a graph-level analysis of the data dependencies between single-rate buffers. Hence, MEGs are unable to capture memory reuse opportunities between input and output buffers of an actor, resulting from internal data dependencies.

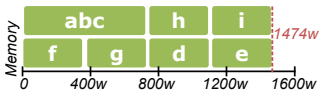
Figure 4a illustrates the internal data dependencies of the *Split* actor. The purpose of this actor is to copy two overlapping slices from its input buffer to its output buffers. Hence, memory can be saved by merging the output buffers of the actor directly within their corresponding ranges of the input buffer. Figure 4b presents the MEG obtained by merging buffers a, b, and c into a single memory object of weight $730w$. To specify that the memory allocated for the new abc buffer can be partially reused to allocate buffers f, h, g, and i,



(a) Merging opportunity



(b) Merging a, b, and c buffers. Dotpoint edges represent partial exclusions.



(c) Memory Allocation

Fig. 4. Buffer merging opportunity for *Split* actor

partial exclusions, depicted with dotpoint edges and excluded ranges from abc, were added to the MEG. Memory allocation resulting from the MEG of Figure 4b is presented in Figure 4c.

The following notation is used for MEG $= \langle M, E, w \rangle$: $M_{merged} \subset M$ is the set of buffers that result from the merging of several memory objects of the original graph. $m_{merged} \in M_{merged}$ is a set of the pair $\{\langle m_k, off_k \rangle\}$, where m_k is a merged memory object from the original MEG on which the buffer merging technique was applied, and $off_k \in \mathbb{N}$ is the start offset in bytes of m_k within m_{merged}. $f(m) \in F$ is used to denote the single-rate FIFO associated with a memory object $m \in M$, assuming that this memory object is not associated with the working memory of an actor. This is always true for buffers in M_{merged}.

IV. Proposed Distributed Memory Allocation Technique

A. Memory Architectures

The three memory architectures depicted in Figure 5 have been chosen to assess the distributed memory allocation technique. With shared memory, all cores of the architecture access a unique memory space that is used to share data among them. With distributed memory, each core of the architecture is associated with a memory component accessible only by this core. With heterogeneous memory, Each core can access both a private memory component accessible only to this core, and a shared memory space.

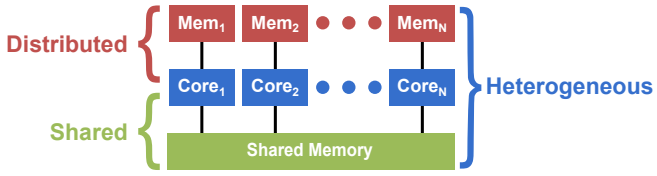


Fig. 5. Memory architectures

Using MEGs (Section III) to perform memory allocation for shared memory architectures, all memory objects allocated in shared memory will be accessible to all cores. Hence, shared memory is used for both storing buffers accessed by actors mapped on a single core and for storing buffers used for inter-processor communication.

When deploying an SDF graph on a distributed architecture, special care must be taken to ensure that a memory object allocated in a distributed memory component will be accessible to all cores executing actors accessing this memory object.

ALGORITHM 1: Split MEG into memory-specific MEGs

Input: $G = \langle A, F \rangle$ a single-rate SDF graph
$MEG_0 = \langle M, E, w \rangle$ the original MEG derived from G
$C = \{c_i, i = 1..n\}$ the set of cores of an architecture
$map : A \rightarrow C$ the mapping actors A on cores C
Output: $\{MEG_i, i = 1..n\}$ the memory-specific MEGs associated to $c_i \in C$
MEG_S the MEG associated to shared memory

```
1  if HETEROGENEOUS_MEMORY then
2  |  MEG_S ← MEG_0
3  |  for each a ∈ A do MEG_S.M ← MEG_S.M \ m(a) endfor
4  else
5  |  MEG_S ← ∅
6  endif
7  for each c_i ∈ C do
8  |  MEG_i ← MEG_0
9  |  for each a ∈ A do
10 |  |  if map(a) ≠ c_i then
11 |  |  |  MEG_i.M ← MEG_i.M \ m(a)
12 |  |  endif
13 |  endfor
14 |  Call Algorithm 2 for c_i, MEG_i, and MEG_S
15 |  for each f ∈ F with m(f) ∉ M_merged do
16 |  |  if DISTRIBUTED_MEMORY then
17 |  |  |  if map(prod(f)) ≠ c_i and map(cons(f)) ≠ c_i then
18 |  |  |  |  MEG_i.M ← MEG_i.M \ {m(f)}
19 |  |  |  endif
20 |  |  else if HETEROGENEOUS_MEMORY then
21 |  |  |  if map(prod(f)) = map(cons(f)) = c_i then
22 |  |  |  |  MEG_S.M ← MEG_S.M \ {m(f)}
23 |  |  |  else
24 |  |  |  |  MEG_i.M ← MEG_i.M \ {m(f)}
25 |  |  |  endif
26 |  |  endif
27 |  endfor
28 endfor
```

B. Memory-Specific MEG: No Buffer Merge

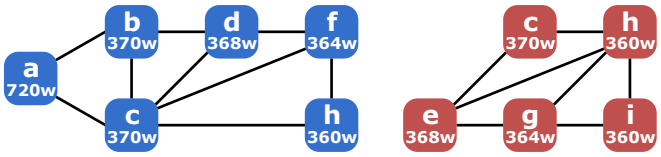
To benefit from both the memory reuse opportunities captured by the MEG of an application, and the performance due to distributed memory, it is necessary to identify which memory can be used to allocate each memory object of the MEG. The simplest solution is to use actor mapping information of an application and split its original MEG into several MEGs, each associated with a specific memory component. The algorithm used to split the MEG into memory-specific MEGs is presented in Algorithm 1. This algorithm is compatible both with distributed and heterogeneous memory organizations.

Algorithm 1 iterates over the cores of the targeted architecture, and verifies whether the memory object $m(f)$ of each single-rate graph FIFO f is made accessible to the actors using it when $m(f)$ is allocated in the memory component associated with the current core. Hence, the algorithm outputs as many memory-specific MEGs MEG_i as there are memory components in the targeted architecture.

Line 8 of the algorithm initializes each output memory-specific MEG MEG_i with a copy of the original MEG: MEG_0. The purpose of the final portion of the algorithm is to remove the memory objects that are not allocated in a memory component from the corresponding MEG_i. Lines 9 to 13 of the algorithm deal with the working memory of each actor: the corresponding memory object is removed from all MEG_i, except if this actor is mapped. The processing of memory objects corresponding to single-rate FIFO depends on the targeted architecture.

The following sections describe MEG splitting methods for distributed memory and heterogeneous memory.

1) Splitting an original MEG for Distributed Memory: The heart of the algorithm for distributed memories (lines 16 to 19) checks if neither $prod(f)$ nor $cons(f)$, the producer and consumer actors of a FIFO f, are mapped to the current core c_i. Meeting this condition means that $m(f)$, the memory object associated with single-rate FIFO f, is never accessed by c_i and can be removed from MEG_i.

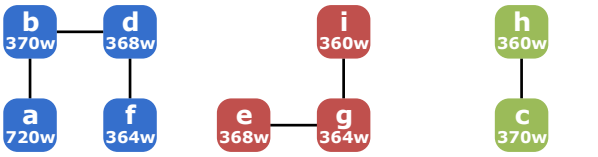


(a) Core 1 MEG (b) Core 2 MEG

Fig. 6. Memory-specific MEGs: Distributed scenario

Figures 6a and 6b show the memory-specific MEGs obtained when Algorithm 1 is applied to the MEG from Figure 3a with the mapping from Figure 2, and assuming a distributed memory architecture. A memory object is duplicated in two memory-specific MEGs if its producer and consumer actors are mapped on two different cores. Inter-core communications ensures that coherent data is available in the memory allocated to memory components h and c.

2) Splitting an original MEG for Heterogeneous Memory: The heterogeneous memory portion of the algorithm (lines 20 to 25) checks if both $prod(f)$ and $cons(f)$, the producer and consumer actors of FIFO f, are mapped to core c_i. Meeting this condition means that $m(f)$, the memory object associated with single-rate FIFO f, is only accessed by core c_i and can be removed from MEG_S, i.e. the memory-specific MEG associated with shared memory. Otherwise, $m(f)$ is removed from MEG_i as it is accessed by other core(s) and should be allocated either in shared memory or in the memory associated with a core distinct from the current core c_i.



(a) Core 1 MEG (b) Core 2 MEG (c) Shared MEG

Fig. 7. Memory-specific MEGs: Heterogeneous scenario

Figure 7 illustrates the memory-specific MEGs obtained when applying Algorithm 1 to the original MEG from Figure 3a with the mapping from Figure 2, and assuming a heterogeneous memory architecture. Memory objects accessed by several cores are not duplicated as they are allocated in the shared memory supporting inter-core communication.

C. Memory-Specific MEG: With Buffer Merging

Since a memory object resulting from the merge of several buffers does not have a unique producer and consumer actor, it is necessary to include the call for Algorithm 2 in Line 14. Algorithm 2 processes each buffer resulting from a buffer merge. The behavior of Algorithm 2 differs depending on the targeted memory organization, as described in following sections.

ALGORITHM 2: Preprocess merged buffers

Input: c_i a core of the architecture
MEG_i the MEG associated with c_i
MEG_S the MEG associated with shared memory
Output: Updated MEG_i and MEG_S

1 **for each** $m_{merged} \in MEG_i.M_{merged}$ **do**
2 **if** *DISTRIBUTED_MEMORY* **then**
3 $R \leftarrow \emptyset$ // byte ranges of m_{merged} accessed by c_i
4 $M_{accessed} \leftarrow \emptyset$
5 **for each** $m_k \in m_{merged}$ **do**
6 **if** $map(prod(f(m_k))) = c_i$ ***or*** $map(cons(f(m_k))) = c_i$ **then**
7 $M_{accessed} \leftarrow M_{accessed} \cup \{m_k\}$
8 $R \leftarrow R \cup [off_k, off_k + w(m_k)[$
9 **endif**
10 **endfor**
11 $MEG_i.M \leftarrow MEG_i.M \setminus m_{merged}$
12 **for each** *range* $r \in R$ **do**
13 $m_{new} \leftarrow$ new memory object of size $r.length$
14 **for each** $m_k \in M_{accessed}$ *with* $[off_k, off_k + w(m_k)[\cap r \neq \emptyset$ **do**
15 Merge m_k within m_{new} with offset $(off_k - r.start)$
16 **endfor**
17 $MEG_i.M \leftarrow MEG_i.M \cup \{m_{new}\}$
18 **endfor**
19 **else if** *HETEROGENEOUS_MEMORY* **then**
20 **if** $\forall m_k \in m_{merged}$, $map(prod(f(m_k))) = map(cons(f(m_k))) = c_i$ **then**
21 $MEG_S.M \leftarrow MEG_S.M \setminus \{m_{merged}\}$
22 **else**
23 $MEG_i.M \leftarrow MEG_i.M \setminus \{m_{merged}\}$
24 **endif**
25 **endif**
26 **endfor**

1) Merged Buffers and Distributed Memory: When targeting distributed memory, Lines 3 to 10 of Algorithm 2 are used to identify which ranges of bytes R of the current buffer $m_{merged} \in M_{merged}$ are accessed by actors mapped on core c_i. At the end of this loop, R may contain zero, one or several non-contiguous ranges of bytes of m_{merged} accessed by c_i, and $M_{accessed}$ contains the set of buffers merged within m_{merged} that are accessed by c_i.

Lines 12 to 18 of Algorithm 2 create a new memory object m_{new} for each non-contiguous range of bytes $r \in R$ accessed by core c_i. All buffers in $M_{accessed}$ whose offset off_k in m_{merged} overlaps with the current range of bytes r are merged within the new memory object at Line 22. Only new memory objects m_{new} are kept in MEG_i, the processed buffer m_{merged} is removed from MEG_i at Line 11.

(a) Core 1 MEG (b) Core 2 MEG

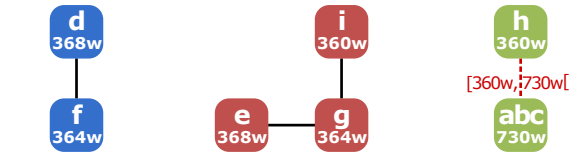
Fig. 8. Merged buffers: Distributed scenario

Figure 8 illustrates the memory-specific MEGs obtained when applying Algorithms 1 and 2 to the original MEG from Figure 4b with the mapping from Figure 2, and assuming a distributed memory architecture. All three merged buffers, a, b, and c, from the original MEG

are accessed by core 1. Consequently, buffer abc is completely copied in the memory-specific MEG associated with core 1 (Figure 8a), thus decreasing the allocated memory footprint to $1102w$ bytes.

2) Merged Buffers and Heterogeneous Memory: When targeting heterogeneous memory, Lines 19 to 25 of Algorithm 2 are used to check whether a buffer m_{merged} resulting from a merging operation should be allocated in a distributed memory component or in shared memory. A buffer m_{merged} resulting from a merging operation will be allocated in the private memory component of a core c_i if all actors accessing merged buffers $m_k \in m_{merged}$ are mapped on c_i.

This strategy was adopted to favor memory reuse opportunities offered by buffer merging over allocation of memory objects in distributed memory components. Indeed, buffer merging removes many memory copy operations (`memcpy()` in C language) between merged buffers, which improves performance more than allocating mergeable memory objects in distributed memory components [5].



(a) Core 1 MEG (b) Core 2 MEG (c) Shared MEG

Fig. 9. Merged buffers: Heterogeneous scenario

Figure 9 shows memory-specific MEGs and allocations obtained when applying Algorithms 1 and 2 to the MEG from Figure 4b with the mapping from Figure 2 assuming heterogeneous memory. In this example, one of the merged buffers, buffer c, is accessed by more than one core, resulting in the allocation in shared memory of the whole buffer produced by the merge of buffers a, b, and c (Figures 9c).

V. Experiments

The distributed memory allocation technique proposed in this paper is implemented within the open-source rapid prototyping framework PREESM [13]. This implementation is used to assess the behavior and impact of the proposed technique on real applications, both theoretically and practically, by generating code for an MPSoC.

A. Experimental context

1) Applications: Two applications are used to assess the proposed memory allocation technique: *Sobel*, an image processing application and *Stereo*, a computer vision application. The properties of these applications are presented in Table I. For each application, the number of actors $|A|$ and FIFOs $|F|$ of the original and single-rate SDF graphs are presented as well as the number of memory objects $|M|$ and the number of exclusions $|E|$ of the original MEG.

	SDF graph		Single-rate	MEG	
Application	$\|A\|$	$\|F\|$	$\|A\|$	$\|F\| = \|M\|$	$\|E\|$
Sobel	6	7	29	36	524
Stereo	28	42	320	1067	310230

TABLE I
PROPERTIES OF THE TEST GRAPHS

The *Sobel* application is the original SDF graph presented in Figure 1 with parameters $h = 720$, $w = 1280$, and $n = 8$. The purpose of this application is to apply several commonly used 2D image filters in order to detect the edges of the processed image. The *Stereo*-matching application is a State-of-the-Art computer vision algorithm whose purpose is to extract 3D information from a pair of 2D images [10]. The large number of actors, FIFOs and exclusions of this application make it an interesting test case for the proposed allocation technique. These two applications are chosen from an open-source repository of PREESM applications (https://github.com/preesm/preesm-apps), to allow these experiments to be reproduced.

2) Target Architecture: : The *TMS320C6678* is an MPSoC manufactured by Texas Instruments. This MPSoC contains eight C66x Digital Signal Processors (DSPs). In these experiments, a heterogeneous memory organization is used on the *C6678* with 512 Mbytes of external shared memory, and 512 kBytes of L2 private memory for each core. The TMS320C6678 has no hardware cache coherence mechanism between the private L1 32 kBytes caches of the 8 cores.

B. Memory Footprint Study

Using the technique introduced in Section IV, memory allocation of applications presented in Table I is performed for each memory organization presented in Figure 5. Memory footprints resulting from the allocation of the two applications are plotted in Figures 10 and 11.

Fig. 10. Sobel application memory footprints

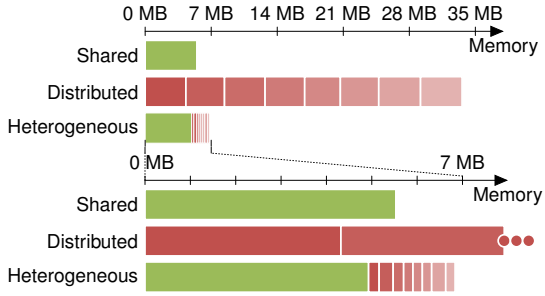


Fig. 11. Stereo application memory footprints

In Figures 10 and 11, for all memory organizations, a green rectangle represents the footprint allocated in shared memory, and red rectangles represent the total footprint allocated in the private memory component of all the cores. In Figure 11, an enlarged view of the 0-7MB portion of the plot is presented below the overall graph, clarifying details between the *Shared* and *Heterogeneous* scenarios.

As shown in these results, memory footprints allocated for distributed and heterogeneous memory organizations are larger than for shared memory organization. This increase of footprint has two causes. First, splitting the original MEG of an application into memory-specific MEGs removes memory reuse opportunities between memory objects allocated in separate memory components. Second, when generating memory-specific MEGs for distributed memory organization, memory objects accessed by two cores are duplicated in their respective MEGs.

In these experiments, the heuristic algorithm responsible for mapping and scheduling SDF actors on cores of the targeted architecture does not consider memory. Instead, latency-oriented optimizations may result in application mappings where most memory objects are

accessed by several cores, and which may undermine the efficiency of the proposed memory distribution technique [13].

In the heterogeneous scenario, approximately 2/3 of the memory objects of *Sobel*, representing 44% of the memory footprint, and 1/3 of the memory objects of *Stereo*, representing 28% of the memory footprint, can be allocated in private memory components. Memory footprints allocated in the heterogeneous scenario are only slightly larger than memory footprints allocated in the shared scenario; footprints are only 15% larger for *Sobel*, and 23% larger for *Stereo*.

In the distributed scenario, memory footprints are 51% larger for *Sobel* and 606% larger for *Stereo* when compared with those of the shared scenario. The large memory footprint of the *Stereo* application is caused by the duplication of large memory objects corresponding to the pair of input images processed by the application, and accessed by actors mapped on all cores. A finer modeling of the *Stereo* application, where the input image is sliced instead of being duplicated, would decrease the memory footprint in the distributed scenario.

The results using heterogeneous memory show the efficiency of the proposed allocation technique. To further increase the efficiency of the proposed method, a memory-aware mapping technique such as that proposed in [9] will be considered in future work.

C. Performance Study

Using PREESM, code was generated for the *C6678* architecture for the two applications. Because the current code generation only supports shared memory-based communications, no code was generated for the distributed memory organization. Table II shows the resulting performance of the *Sobel* and *Stereo* applications. In this table, performance is expressed as the number of frames per second (fps) processed by the applications. Columns *No cache* and *L1 cache* give the performance of applications when the private L1 caches are deactivated and activated, respectively.

		Performance	
Application	Archi.	Caches off	Caches on
Sobel	Shared	1.08 fps	12.39 fps
	Hetero	5.18 fps	12.66 fps
Stereo	Shared	0.40 fps	2.50 fps
	Hetero	0.54 fps	2.97 fps

TABLE II
APPLICATION PERFORMANCE

The performance of the *Sobel* application with deactivated caches is increased by 380% for the heterogeneous scenario compared to the shared scenario. The better result is due to the performance gap between memory accesses in shared L3 memory, and in private L2 memory components. This example shows that application performance can be dramatically increased with only 44% of the memory allocated in the private memory of the cores. The performance of the *Sobel* application when L1 caches are activated is only 2% higher for the heterogeneous scenario than for the shared scenario. The near equivalence of performance between the two scenarios is due to the efficiency of the caching mechanism thus masking the majority of differences in access times between shared and private memory components.

The performance of the *Stereo* application for the heterogeneous scenario is 35% better than for the shared scenario, when the caches are deactivated. When L1 caches are activated, the performance in the heterogeneous scenario remains 19% higher than for the shared scenario. In the *Stereo* application, large memory objects are accessed by the application, thus decreasing the efficiency of the caching mechanism while preserving the benefits of the distributed memory allocation technique. These results show that the memory distribution technique proposed in this paper has a positive impact on the performance of applications executed on a real off-the-shelf MPSoC.

VI. CONCLUSION

This paper introduced a new memory allocation technique for the deployment of applications modeled with dataflow graphs on multicore architectures with distributed memory components. The proposed technique builds on a State-of-the-Art memory allocation technique for shared memory architectures. While providing allocation of dataflow applications in distributed memory components, the proposed technique preserves memory reuse opportunities between the application buffers, including memory reuse specified using a State-of-the-Art buffer merging technique. Experiments show that using the proposed technique when deploying a real image processing application on an off-the-shelf MPSoC leads to better application performance. Future work on this topic includes predicting time performances related to memory allocation and combining the proposed memory allocation technique with a memory-aware mapping and scheduling algorithm.

REFERENCES

[1] M. Benazouz, O. Marchetti, A. Munier-Kordon, and P. Urard, "A new approach for minimizing buffer capacities with throughput constraint for embedded system design," in *AICCSA*. IEEE, 2010, pp. 1–8.

[2] P. M. Carpenter, A. Ramirez, and E. Ayguadé, *HiPEAC*. Springer, 2010, ch. Buffer Sizing for Self-timed Stream Programs on Heterogeneous Distributed Memory Multiprocessors, pp. 96–110.

[3] L. Cudennec, P. Dubrulle, F. Galea, T. Goubier, and R. Sirdey, "Generating code and memory buffers to reorganize data on many-core architectures," *Procedia Computer Science*, vol. 29, pp. 1123–1133, 2014.

[4] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi, "Memory analysis and optimized allocation of dataflow applications on shared-memory mpsocs," *JSPS, Springer*, vol. 80, no. 1, pp. 19–37, 2014.

[5] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi, "On memory reuse between inputs and outputs of dataflow actors," *ACM TECS*, vol. 15, no. 30, p. 25, January 2016.

[6] D. Lee, S. S. Bhattacharyya, and W. Wolf, "High-performance buffer mapping to exploit DRAM concurrency in multiprocessor DSP systems," in *RSP*. IEEE, 2009, pp. 137–144.

[7] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, sept. 1987.

[8] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[9] Y. Lesparre, A. Munier-Kordon, and J.-M. Delosme, "Compile-time mapping of dataflow applications with buffer minimization," in *IDEA*, 2015, pp. 9–12.

[10] A. Mercat, J.-F. Nezan, D. Menard, and J. Zhang, "Implementation of a Stereo Matching Algorithm Onto a Manycore Embedded System," in *ISCAS*. IEEE, 2014, pp. 1296–1299.

[11] P. Murthy and S. Bhattacharyya, "Shared memory implementations of synchronous dataflow specifications," in *DATE*. ACM, 2000.

[12] T. M. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, University of California, 1995.

[13] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming," in *EDERC*. IEEE, Sep. 2014.

[14] J. Pino, S. Bhattacharyya, and E. Lee, "A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs," University of California, Tech. Rep., 1995.

[15] Semiconductor Industry Association, "International Technology Roadmap for Semiconductors (ITRS): System Drivers," 2011.

[16] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *DAC*. ACM, 2006, pp. 899–904.

[17] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH*, vol. 23, no. 1, pp. 20–24, 1995.