Procedia Computer Science
Volume 29, 2014, Pages 1123–1133
ICCS 2014. 14th International Conference on Computational Science

Generating Code and Memory Buffers to Reorganize Data on Many-core Architectures

Loïc Cudennec[1], Paul Dubrulle[1], François Galea[1], Thierry Goubier[1], and Renaud Sirdey[1]

CEA, LIST, Saclay, France
`firstname.name@cea.fr`

Abstract
The dataflow programming model has shown to be a relevant approach to efficiently run massively parallel applications over many-core architectures. In this model, some particular builtin agents are in charge of data reorganizations between user agents. Such agents can *Split*, *Join* and *Duplicate* data onto their communication ports. They are widely used in signal processing for example. These system agents, and their associated implementations, are of major importance when it comes to performance, because they can stand on the critical path (think about Amdhal's law). Furthermore, a particular data reorganization can be expressed by the developer in several ways that may lead to inefficient solutions (mostly unneeded data copies and transfers). In this paper, we propose several strategies to manage data reorganization at compile time, with a focus on indexed accesses to shared buffers to avoid data copies. These strategies are complementary: they ensure correctness for each system agent configuration, as well as performance when possible. They have been implemented within the Sigma-C industry-grade compilation toolchain and evaluated over the Kalray MPPA 256-core processor.

Keywords: Many-core, Dataflow, Compilation, Data reorganization

1 Introduction

Many-core processors are composed by hundreds, if not thousands processing cores on a single chip. These cores are connected by a network on chip (NoC) that mirrors the global chip organization, from a flat 2-D mesh, to complex hierarchical clustered architectures. Many-core are expected to enter green high performance computing, embedded systems and mobile devices, as they offer processing power while keeping a reasonable power consumption. However, getting high performance is conditioned by the efficient use of the chip, mainly relying on massive parallelism that is still hard to express in regular programming languages.

In this context, the dataflow paradigm appears to be a relevant approach. Processings are executed by agents (or tasks, depending on the language) onto data that are exchanged using a set of user-defined communication links. The inherent parallelism comes from the

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.101

concurrent execution of agents on different data. With dataflow languages, developers can focus on the processing code while all communications and synchronizations between agents are transparently optimized by the compilation toolchain and then handled by the runtime.

However, transparency has a price when it comes to performance. While in regular parallel programming languages it is possible to finely tune every aspect of task creation, communications, data localization and migration, with dataflow programming these tuning are now the responsability of the sole compiler. In order to hide these parallelism considerations, most of the toolchains achieve compilation steps such as graph instanciation, task scheduling, task placement and communication routing, with a few, if not any help from the developer. Compilation results are part of the runtime, a code-generated library that gathers all information to run a particular application.

One of the main application performance issue is to efficiently manage communications between tasks. In the dataflow programming model, communications share some similarities with the message passing programming model. However, the underlying implementation does not necessarily rely on message passing. Depending on the targeted architectures, communications can be achieved using message passing, unified and shared memory, or other complex systems like a DMA (direct memory access). Therefore, choosing the right method to implement communications has two goals: correctness and efficiency. And this has to be done to every single communication channel.

In dataflow languages, some particular agents - that we call *system agents* in the following - are defined to reorganize data between processing agents. For example, some widely-used system agents include the *Split* agent that spreads data among communication ports, the *Join* agent that merges data, and the *Duplicate* agent. Despite the fact that these agents do not necessarily have a *real* existence at run time [1], their implementations might imply non negligible processing costs, which is particularly true when composed together (e.g. a split followed by a join agent). From the compilation point of view, system agent composition brings severe complexity for both correctness and efficiency goals. One of the main objective of the compiler is to limit the number of online buffers and data copies to achieve communications.

In this paper, we present and evaluate several contributions that have been proposed to manage system agents within a state-of-the-art industrial-grade dataflow compiler [3]. We describe three compilation strategies to reorganize data that can be deployed on different many-core architectures. The first strategy consists in calculating access indexes to shared buffers. This appears to be the most efficient approach because processors directy read and write memory without indirection. However this can only be applied in particular data reorganizations, using a physical shared memory. The second approach consists in generating micro-code to drive a DMA. This approach is less efficient than the latter one, but it can be applied without physical shared memory. Finally, the compiler can fall back into a user-code compilation mode that consists in writing genuine dataflow source code for each system agent. This approach is clearly inefficient because of the numerous buffers and data copies that are involved. However, this can be safely used in all situations and it ensures to find a solution even with complex system agent compositions.

The remaining of this paper is organized as follow. The second section presents the dataflow paradigm and data reorganizations. Section three presents different compilation strategies that are evaluated in section four. Section five concludes this paper.

[1] the concept of agents proposed by the programming language does not necessarily match with the concepts used by the underlying system (e.g. system thread).

2 Data reorganization and the Dataflow Paradigm

Dataflow programming is one paradigm in which processings are organized within communicating tasks. A dataflow program consists in defining tasks (also called *agents* in the remainder of the paper), and in describing the application structure (basically connecting ports in order to build a graph). Data are written onto output ports and read onto input ports, in a sequential and FIFO way. For example, a picture can be sent between two agents pixel by pixel, line by line or even in a single block of data.

Most of the applications require more or less complex *data reorganizations* between tasks. As an example, a simple edge detection application can be written using two successive parallel filter steps, as illustrated in figure 1. In this example, processings are first applied on the lines, then on the columns of each image. As filters read an write pixels sequentially, a transposition of the image is applied in between. This transposition is implemented thanks to the combination of a join and a split agent with appropriate production and consumption quantities (the k parameter).

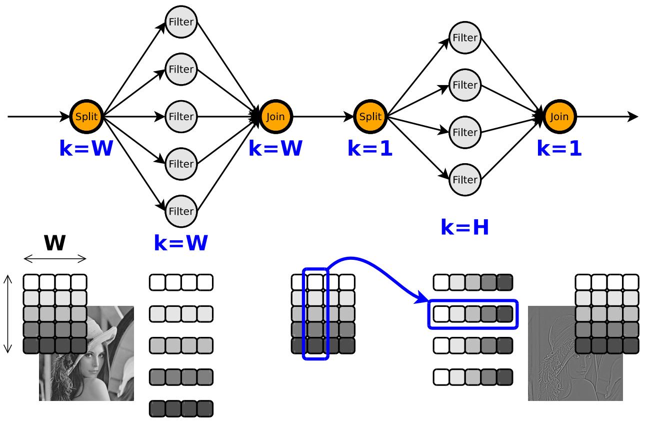


Figure 1: *Communication graph for an edge detection application. System agents are used to spread lines, then columns among filtering agents. Consumptions and productions are given by k.*

System agents. However, some agents may build data from non-contiguous parts of the original data. For example, reading a macro-block [2] (say BW wide and BH height) in a picture (W wide and H height) requires to select some sparse parts of a given number of lines. This pattern, as applied on a streamed picture, consists in looping BH times a read of BW pixels followed by a shift of the image width W minus the block width BW. There are basically two ways of doing this. A first one is to build the macro-block within the producer agent and directly write it onto the output port to the consumer. This makes the producer wait for all the pixels (even the unneeded ones) before sending the block, leading to an excessive use of memory (it may not be possible to store all the data needed to build the block, especially in the context of embedded architectures). It also relies on user-code that is in charge of building and filling an appropriate memory structure, which is quite an annoying, error-prone and repetitive task for the developer. A second way, as illustrated in figure 2, is to use *builtin agents.* In this example, the image is split onto the seven output ports of the Split agent. Each port receives a sequence of pixels as described in the right part of the figure. From outputs 1, 3 and 5, it is possible to rebuild the block thanks to a round-robin Join agent. This solution has two major interests. First, there is no user-code involved (except maybe for the production and consumption quantities that are set for each system agent port): processing agents written by the developer receive ready-to-use data. Second, the data reorganization is under the control of the compilation toolchain and as

[2]we consider a macro block as a block of pixels, not following the MPEG definition

a matter of fact, it benefits from a dedicated implementation, as well as several optimizations for the targeted platform.

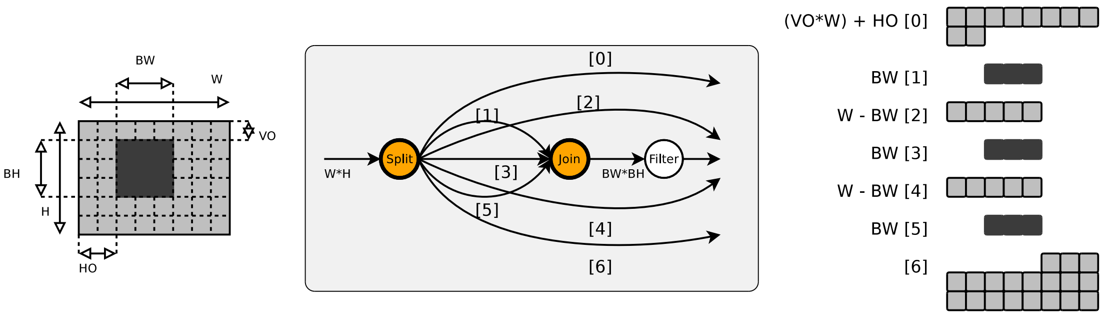


Figure 2: *Using a Split and a Join to read a block within an image. Image size:* $W * H$*, block size:* $BW * BH$*, offset:* $(VO * W) + HO$*.*

Data reorganization compilation. In order to let the compiler deal with data reorganization, some stream and dataflow-based languages propose a set of system agents. These builtin agents can be used thanks to reserved keywords and an appropriate syntax. This is particularly true for Streamit [14] (using *Pipeline*, *Splitter*, *Joiner* and *FeedbackLoop* composing filters), Sigma-C [12] (using *Split*, *Join* and *Dup* system agents), Tau-C [11] (using *Split*, *Join* and *Dup* system tasks), SPL [13] (using *Sort*, *Split*, *Join*, *Aggregate* stream operators) and GreenStreams [4] (using *SplitJoin* and *FeedbackLoop* stream filters). In a slighly different approach, CAL [8] proposes to bind data sequences between ports thanks to the definition of index sequences (refered as *multiport input patterns*). While most of the publications available for these languages deal with expressiveness, application case studies and general compilation strategies, a very few information are given on the particular problem of the implementation of data reorganization. In the Streamit project [2, 10], the authors present several contributions in which the compiler is able to modify the application graph, based on the recognition of system agent patterns. *Fission* and *reordering* transformations simplify the application graph to get better performance. In [7, 6], transformations are also applied at compile time onto system agents in order to adapt the degree of parallelism to the execution platform. However, these optimizations remain at a high level of abstraction in the application description and do not give elements on generating an efficient runtime for data reorganization.

The Sigma-C dataflow language. In this paper, we show how data reorganizations are processed at compile time to be handled in a runtime for a dataflow programming language. We focus on the Sigma-C [12] language that has been proposed to enhance manycore programmability, and has been successfully implemented and validated with the Kalray 256-core MPPA processor [1]. Sigma-C is based on a deterministic process network with process behaviour specifications, which is very close to cycle-static dataflow (CSDF [5]). Sigma-C has been designed as an extension of the regular ANSI C language, with some additional keywords to declare agents and establish connections between ports. In the model, the agent specification defines all consumptions and productions that occur on communication ports each time the agent is fired up (what is called an *occurrence*). It is possible to specify different consumptions and productions depending on the occurrence number. For example, a classic behaviour for the round-robin split agent is to read k_j data on its input port and to write k_j data on output port i (i being incremented the next occurrence and looping to the first output port when reaching the last one). This accurate specification is used by the compiler in order to verify properties such as communication buffer sizing, deadlocks, task placement, routing and tuning of the degree of parallellism. The specifications play a central role for building the runtime [3], as they are used to determine the scheduling of the task occurrences, the number of communication buffers, their sizes and as presented in the next section, the data access patterns.

Equivalence of pointers. The data reorganization compilation has several constraints. A first constraint, tightly coupled with the *programmability* of the Sigma-C language, makes possible to transparently access communication ports by defining aliases. These aliases can be manipulated either as primitive types in case of a single data, or as a memory pointer in case of complex structures. This is convenient from the developer point of view because ports can be directly read and written without the use of a dedicated API. Port aliasing is motivated by the possibility to transparently manage all communications and to ease the porting of regular C applications. However, transparency brings complexity on the compiler side that has to ensure that all aliases point to a contiguous memory space. This property, called *equivalence of pointers*, allows to access buffers without any indirection. Pointer arithmetic is offered without degrading the performance (for example reading an array by adding its aliased based memory address and an online-evaluated offset). For each occurrence of the task and for each of its port, the compiler is in charge of building a memory space (shared with one or more ports) that is big enough to host the expected data, and to properly update this buffer with incoming data. Furthermore, if an input port is connected to a combination of system agents, the data update has to mirror the corresponding reorganization.

Performance and correctness. Another constraint is to offer reasonable *performance* while executing data reorganizations. In parallel and distributed computing, a large amount of time can be spend in copying data between buffers. This can be seen for example in high performance networking with zero-copy protocols that let the user code directly access data received at the bottom of the network stack. In our context, we minimize the number of buffers by merging them into shared buffers when possible. Finally, a constraint consists in finding a solution to each system agent combination (several system agents connected together with at least one communication port, also called a *system surface* in the following). If most of the time there is a trivial way for compiling a single agent, agent combinations introduce complex memory access patterns for agents connected to the system surface. In that particular cases, if these patterns are too complex to calculate, the compiler falls back to a simple mode with multiple temporary buffers.

3 Data Reorganization Strategies at Compile Time

The Sigma-C toolchain is made of four main compilation steps. The first step applies lexical parsing and semantic analysis. The second step instantiates the application graph and adapts the parallelism degree. The third step proceeds to the buffer sizing, task scheduling and the place and route of tasks onto the hardware. Finally, the fourth step generates the runtime and

[3]In this paper we consider the runtime as a set of instanciated parameters and some generated pieces of code, to be used with system functions that are part of a system software.

processes a hierchical linking with the user code and the system software. We describe how these steps are involved in the compilation of system agents, using three different strategies.

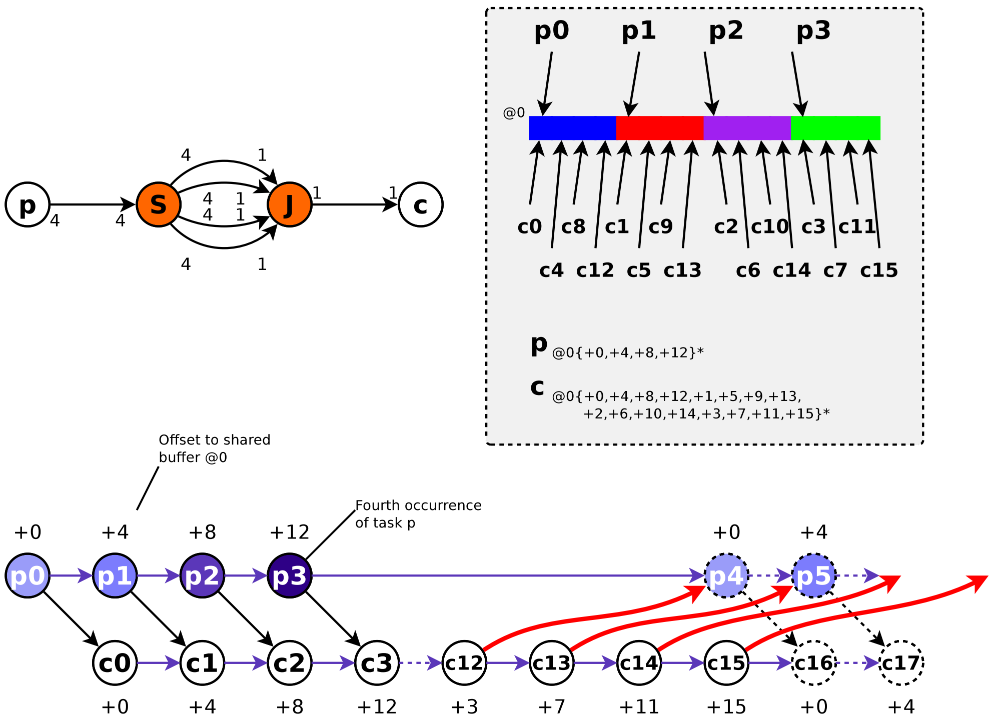
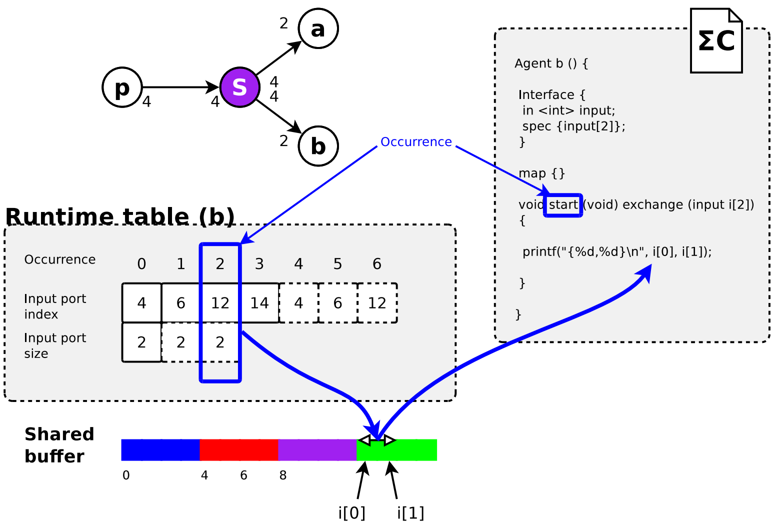

p
S
J
c
4
1
p0
p1
p2
p3
@0
c0 c8 c1 c9 c2 c10 c3 c11
c4 c12 c5 c13 c6 c14 c7 c15
p @0{+0,+4,+8,+12}*
c @0{+0,+4,+8,+12,+1,+5,+9,+13, +2,+6,+10,+14,+3,+7,+11,+15}*
Offset to shared buffer @0
Fourth occurrence of task p
+0 +4 +8 +12 +0 +4
p4 p5
c12 c13 c14 c15 c16 c17
+0 +4 +8 +12 +3 +7 +11 +15 +0 +4

Figure 3: *Access patterns and the occurrence scheduling graph.*

Shared Buffers and Indexes Generation. Shared buffers are one efficient strategy to reorganize data. For each system agent surface a shared buffer is created and for each communication port belonging to the edge of the surface, a data access pattern is calculated. This pattern defines, for each occurrence of the task, the offset and the access size to be applied on the shared buffer. Figure 3 shows a surface made of two system agents (a split and a join), connected to one producer p and one consumer c. Productions and consumptions are indicated along the communication links. On the right side, we represent the corresponding shared buffer @0 and the different accesses made by the user agents. The buffer has a length of 32 elements, and is partially updated by each occurrence of p, starting at address 0 for p_0, 4 for p_1, 8 for p_2 and 12 for p_3. The pattern is thereafter repeated for the next occurrences. Consumer c is given as well. At the bottom of the figure, we show the scheduling graph for tasks p and c. Nodes represent the task occurrences and the edges the dependencies. Dependencies can be implicit (between two consecutive occurrences of the same task), explicit (two different tasks) or directed by the producer-consumer pattern (for example, occurrence p_4 requires c_{12} to write in the first part of the buffer to avoid overwriting unread data). A direct mapping is done between this scheduling graph and the access patterns.

This direct mapping is illustrated in Figure 4: a single split agent is connected to one producer p and two consumers a and b. On the right side is the complete Sigma-C source code of consumer b. The *interface* section declares one input port and the *spec* section indicates that the input port receives two data each time the *start* function is fired up (which happens at each occurrence). Within this function, a read access is made to the input port, aliased to i. A representation of the runtime access table is given for the input port of b. It indicates that when executing the start function at occurrence number 2, the i alias (and therefore $i[0]$) points to the twelfth element of the shared buffer. This table is represented in a slightly different way in the runtime, using increments instead of offsets. This example preserves the equivalence of pointer as alias i always points to a contiguous memory space of size 2. This is a direct consequence of the production and consumption quantities of the split agent and its consumers that divide each other.



```
Agent b () {

 Interface {
  in <int> input;
  spec {input[2]};
 }

 map {}

 void start (void) exchange (input i[2])
 {

  printf("{%d,%d}\n", i[0], i[1]);

 }

}
```

Occurrence	0	1	2	3	4	5	6
Input port index	4	6	12	14	4	6	12
Input port size	2	2	2				

Figure 4: *From the Sigma-C source code to the runtime access table.*

Figure 5 illustrates what happens when such conditions are not met and the equivalence of pointers (EoP) is broken. On the left part is the application graph with a surface made of six system agents. On the right part we represent the different accesses onto the shared buffer. This buffer has a size of 32 elements and is equally divided into 4 parts, following the productions of agent p. Below the buffer, we represent the memory access positions for different agents. Each access has a number giving the order (this corresponds to the occurrence number, starting from 1). For example, the first access of agent S_0 is made at offset 0, then at offsets 4, 16 and 20. According to this, we calculate the expected offsets for the agents connected after S_0. At the end of the surface, consumer c follows the access pattern given at the bottom of the figure. This incremental method is also used by the algorithm in charge of calculating the access patterns within the compiler. Figure 5 shows two compilation cases regarding EoP: branch S_0-J_0 accesses 8 elements at offsets 0 and 16, in which c is always able to read 2 contiguous elements, whereas branch S_1-J_1 leads to sparse memory sections (red sections on the figure). Indeed, the memory space attached to the third occurrence of agent S_1 is composed by two sections: two elements starting at offset 14 and one element located at offset 24. Non contiguous sections also refers to the problem of *data alignment*, in which memory operations are efficient if data is located at a memory offset that is equal to some multiple of the word size. In this example, EoP is broken by the fact that S_1 consumptions do not divide S productions (same for J_1 and S_1). On the figure, system agents that are connected to agents whose consumptions divide productions are in violet, and in orange otherwise. As for the compiler's algorithm, we consider each system agent and compare its output productions with the consumptions of the ports it is connected to. Different situations are then handled.

The equivalence of pointers is always preserved when the consumptions divide the produc-

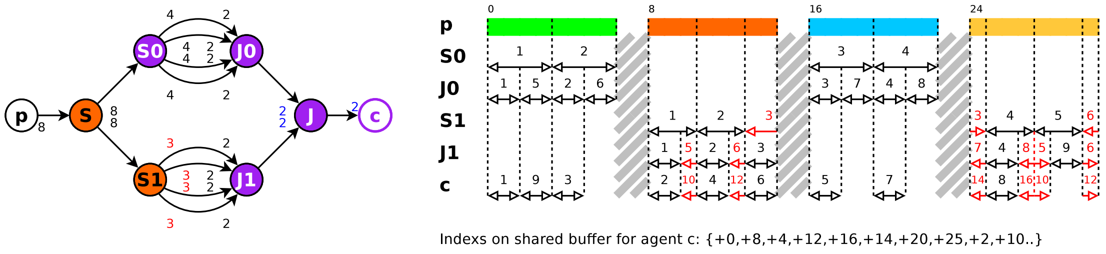

p
S
S0
J0
S1
J1
J
c
Indexs on shared buffer for agent c: {+0,+8,+4,+12,+16,+14,+20,+25,+2,+10..}

Figure 5: *Equivalence of pointer with a consumer connected to a system surface.*

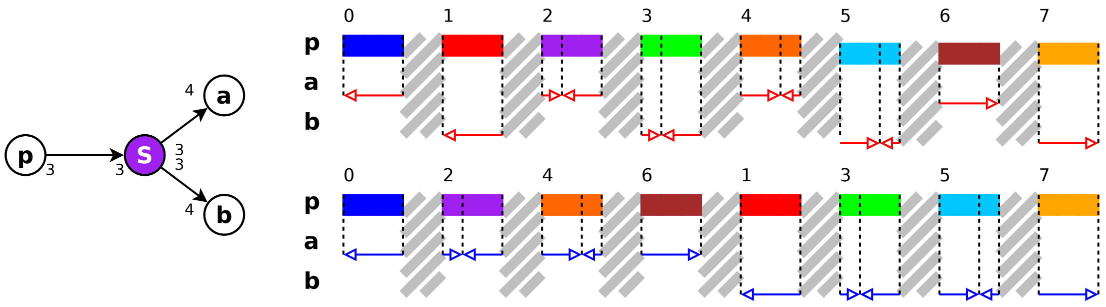


Figure 6: *Reorganizing the producer writing pattern to keep the equivalence of pointer.*

tions (whatever the system agents are). Other configurations generally do not work as this. However, some of them can be slightly modified in order to get the EoP property back. These modifications are made by the compiler and work as presented in figure 6 in which the split agent productions are not written sequentially (as in the top diagram), but instead a contiguous space is reserved to each output port (bottom diagram).

The shared buffer size is calculated while compiling the system surface. The optimal size is the minimum size that allows a maximum degree of parallelism (giving enough space for producers to write while consumers read). However, due to memory restrictions at run time (at least in embedded systems), the buffer size can also be seen as a tradeoff between parallelism and memory footprint. Usually, a good guess is to use the least common multiple of the sum of the productions and the sum of the consumptions.

DMA Micro-Code Generation. Shared buffer compilation is tightly coupled with the targeted hardware architecture: to be efficient, agents connected to the same system surface have to be mapped [9] onto cores that share a physical memory (let's call it a cluster). With both complexity of applications and many-core memory hierarchies, some surfaces may not find a cluster and have to be split among them. Several many-core systems offer DMA engines to move data from one cluster to another. In the Sigma-C compiler, a set of basic surfaces have been identified (such as split-join), and it is possible to generate the corresponding DMA micro-code. This micro-code can either be made of nested loops (up to 3 if $3D$ DMA engine), or a sequence of indexes, which is better regarding performance.

User Code Generation. Whenever it is not possible to calculate access patterns onto a shared buffer, neither a DMA micro-code, the compiler falls back into a safe transformation of system agents into user agents. A genuine Sigma-C source code is generated from system parameters attached to the remaining system agents. This source code is merged to the project thanks to iterative compilation. This simple strategy ensures to find a solution in complex situations, however this leads to the use of one buffer and data copies per communication port, which is far from being efficient.

4 Experiments

In this section, we evaluate the shared buffer and the user agent compilation strategies for three synthetic applications and an industrial motion detection application. Experiments are conducted over the instruction set simulator (ISS) of the Kalray MPPA 256-core processor. This processor is made of 16 clusters interconnected by a dual torus network-on-chip, each cluster being composed of 16 cores and a shared 2 MB scratchpad memory. The ISS is an accurate simulator that mirrors the architecture behaviour, down to the core registers. Synthetic applications include a simple system surface made of 2 successive identity agents, a surface of cascading split agents (7 split agents organized in 3 levels, each split having 2 outputs) and a simple matrix transposition surface (a split reading line by line connected to a join reading element by element).

The motion detection application takes two images and draws rectangles that highlight parts that have changed. Input images are both split thanks to 2 duplicate agents within 8 processing pipelines made of 6 user agents and a duplicate agent. Finally, 3 join agents rebuild the resulting image. When compiling this application, 11 system agents are transformed into shared buffers, while the 2 remaining, part of those in charge of rebuilding the image, are transformed into user agents.

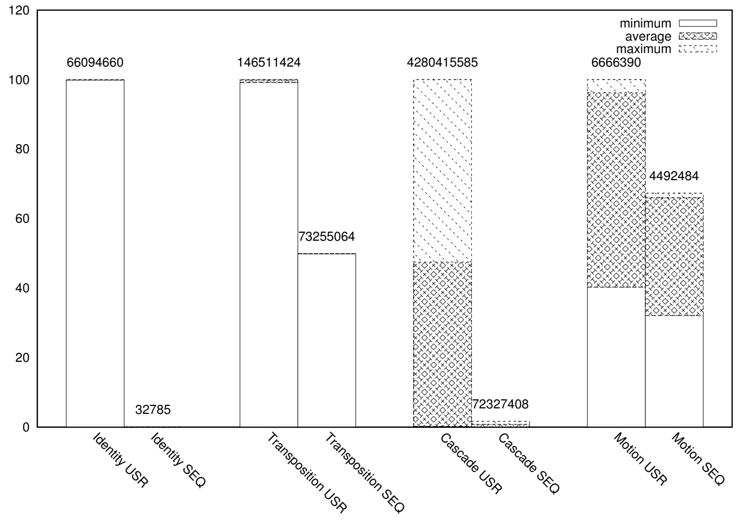


Figure 7: *Instruction Set Simulator cycles are given to complete one application cycle. We compare applications compiled with the user agents strategy (USR) with the shared buffer strategy (SEQ). For each application, the representation is normalized to the maximum value. Smaller is better.*

In these experiments, we evaluate the time neededed to process a piece of data, corresponding to the complete cycle of the application. This includes all user processings and data reorganizations. Figure 7 shows the number of simulation cycles (one ISS cycle is 2.5ns) needed to perform one application cycle, for the applications compiled in both user (USR) and shared buffer (SEQ) modes. The general observation is that shared buffers significantly improve performance for all the applications. This is particularly true for complex system surfaces such as the split cascade (the identity application is quite obvious and should be read as a test case). The industrial motion detection application also benefits from shared buffers by running 1.5x times faster. This also demonstrates that even if all the algorithm complexity stands within the user code, data reorganization plays a key role in the overall performance.

5 Conclusion

Data reorganization can be a major issue while running applications onto may-core processors, due to the complex memory movements it involves. Dataflow programming languages transparently manages communications, but at the price of relying on a compiler that efficiently works on it. In this paper we focus on the use of shared buffers and the calculation of memory access patterns. This strategy has shown to be efficient while targeting the 256-core MPPA processor, as it is possible to use shared physical memories. The compiler is also able to switch back to a micro-code generation for a DMA engine, or to a user source code generation. Experiments show expected improvements over regular data reorganizations, especially in presence of large system surfaces and complex reorganizations. It also improves performance in the context of the motion detection application (an industrial test case for the MPPA processor) that runs up to 1.5x times faster. Some future works can focus on generalizing the shared buffer strategy to all system agent combinations, regardless of their productions and consumptions.

References

[1] The kalray mppa 256 manycore processor. Kalray S.A. http://www.kalray.eu/.

[2] Saman Amarasinghe, Michael I. Gordon, Michal Karczmarek, Jasper Lin, David Maze, Rodric M. Rabbah, and William Thies. Language and compiler design for streaming applications. *Int. J. Parallel Program.*, 33(2):261–278, June 2005.

[3] Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Loïc Cudennec, Vincent David, Philippe Doré, Paul Dubrulle, Benoît Dupont De Dinechin, François Galea, Thierry Goubier, Michel Harrand, Samuel Jones, Jean-Denis Lesage, Stéphane Louise, Nicolas Morey Chaisemartin, Thanh Hai Nguyen, Xavier Raynaud, and Renaud Sirdey. Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor. In *Alchemy 2013 - Architecture, Languages, Compilation and Hardware support for Emerging ManYcore systems*, volume 18, pages 1624–1633, Barcelona, Espagne, June 2013.

[4] Thomas W. Bartenstein and Yu David Liu. Green streams for data-intensive software. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 532–541, Piscataway, NJ, USA, 2013. IEEE Press.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, 1996.

[6] Sergiu Carpov, Loïc Cudennec, and Renaud Sirdey. Throughput constrained parallelism reduction in cyclo-static dataflow applications. In *International Conference on Computational Science (ICCS 2013)*, volume 18, pages 30–39, Barcelona, Espagne, June 2013.

[7] Loïc Cudennec and Renaud Sirdey. Parallelism reduction based on pattern substitution in dataflow oriented programming languages. In *Proceedings of the 12th International Conference on Computational Science*, ICCS'12, Omaha, Nebraska, USA, June 2012.

[8] J. Eker and J. W. Janneck. Cal language report specification of the cal actor language. Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley, 2003.

[9] François Galea and Renaud Sirdey. A parallel simulated annealing approach for the mapping of large process networks. In *IPDPS Workshops*, pages 1787–1792. IEEE Computer Society, 2012.

[10] Michael Gordon, William Thies, Michal Karczmarek, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002.

[11] Thierry Goubier, Damien Couroussé, and Selma Azaiez. Programming the sthorm manycore: Tauc. In *Design, Automation and Test in Europe (DATE 2013), STHORM Workshop*, March 2013.

[12] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. Sigma-C: A programming model and language for embedded manycores. In Yang Xiang, Alfredo Cuzzocrea, Michael Hobbs, and Wanlei Zhou, editors, *Algorithms and Architectures for Parallel Processing*, volume 7016 of *Lecture Notes in Computer Science*, pages 385–394. Springer Berlin / Heidelberg, 2011.

[13] Sherif Sakr. An introduction to infosphere streams: A platform for analyzing big data in motion. Technical report, May 2013.

[14] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *In International Conference on Compiler Construction*, pages 179–196, 2001.