Program Analysis and Source-Level Communication Optimizations for High-Level Synthesis

Christophe Alias, Alain Darte, Alexandru Plesco

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Program Analysis and Source-Level Communication Optimizations for High-Level Synthesis

Christophe Alias — Alain Darte — Alexandru Plesco

N° 7648 — version 1

initial version June 2011 — revised version Juin 2011

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--7648--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Program Analysis and Source-Level Communication Optimizations for High-Level Synthesis

Christophe Alias* , Alain Darte* , Alexandru Plesco*

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet COMPSYS

Rapport de recherche n° 7648 — version 1 — initial version June 2011 —
revised version Juin 2011 — 16 pages

Abstract: The use of hardware accelerators, e.g., with GPGPUs or customized circuits using FPGAs, are particularly interesting for accelerating data- and compute-intensive applications. However, to get high performance, it is mandatory to restructure the application code, to generate adequate communication mechanisms, and to compile the different communicating processes so that the resulting application is highly-optimized, with full usage of the memory bandwidth. In the context of the high-level synthesis (HLS) of hardware accelerators, we show how to automatically generate such an optimized organization for an accelerator communicating to an external DDR memory. Our technique relies on loop tiling, the generation of pipelined processes (overlapping communications & computations), and the automatic design (synchronizations and sizes) of local buffers.

Our first contribution is a program analysis that specifies the data to be read from and written to the external memory so as to reduce communications and reuse data as much as possible in the accelerator. This specification, which can be used in different contexts, handles the cases where data can be redefined in the accelerator and/or approximations are needed because of non-analyzable data accesses. Our second contribution is an optimized code generation scheme, entirely at source-level, that allows us to compile all the necessary glue (the communication processes) with the same HLS tool as for the computation kernel. Both contributions use advanced polyhedral techniques for program analysis and transformation. Experiments with Altera HLS tools show the correctness and efficiency of our technique.

Key-words: Polyhedral optimizations, communication optimizations, pipelined processes, DDR memory, hardware accelerators, HLS.

* Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Analyse de programme et optimisations des communications au niveau source pour la synthèse de haut niveau

Résumé :

Les accélérateurs matériels, comme par exemple via l'utilisation de GPGPUs ou de circuits dédiés sur FPGAs, sont particulièrement intéressants pour accélérer les applications gourmandes en calculs et en accès aux données. En revanche, pour obtenir de bonnes performances, il est indispensable de restructurer le code de l'application, de générer des mécanismes de communication adéquats, et de compiler les différents processus communicants de sorte que l'application résultante soit hautement optimisée, avec un bon usage de la bande passante vers la mémoire. Dans le contexte de la synthèse de haut niveau (HLS) d'accélérateurs matériels, nous montrons comment générer automatiquement une telle organisation optimisée pour un accélérateur communiquant avec une mémoire externe DDR. Notre technique repose sur le « tiling » (calcul par bloc), la génération de processus pipelinés (en recouvrant calculs et communications), et la conception automatique (synchronisations et tailles) de « buffers » locaux.

Notre première contribution est une analyse de programme qui spécifie les données à lire depuis la mémoire externe et à écrire dans cette mémoire de façon à réduire les communications et à réutiliser les données, autant que faire se peut, dans l'accélérateur. Cette spécification, qui peut être utilisée dans d'autres contextes, prend en compte les cas où les données peuvent être redéfinies dans l'accélérateur et/ou des approximations sont nécessaires du fait d'accès non analysables. Notre seconde contribution est un schéma de génération de code optimisé, entièrement au niveau source, qui nous permet de compiler tous les mécanismes d'optimisation (les processus communicants) avec le même outil de HLS que le noyau de calcul lui-même. Ces deux contributions utilisent des techniques polyédriques avancées d'analyse et de transformation de programme. Les expérimentations menées avec l'outil de synthèse d'Altera C2H montre la correction et l'efficacité de notre technique.

Mots-clés : Optimisations polyédriques, optimisations des communications, processus pipelinés, mémoire DDR, accélérateurs matériels, synthèse de haut niveau.

Program Analysis and Source-Level Communication Optimizations for High-Level Synthesis

Abstract – **The use of hardware accelerators, e.g., with GPGPUs or customized circuits using FPGAs, are particularly interesting for accelerating data- and compute-intensive applications. However, to get high performance, it is mandatory to restructure the application code, to generate adequate communication mechanisms, and to compile the different communicating processes so that the resulting application is highly-optimized, with full usage of the memory bandwidth. In the context of the high-level synthesis (HLS) of hardware accelerators, we show how to automatically generate such an optimized organization for an accelerator communicating to an external DDR memory. Our technique relies on loop tiling, the generation of pipelined processes (overlapping communications & computations), and the automatic design (synchronizations and sizes) of local buffers.**

Our first contribution is a program analysis that specifies the data to be read from and written to the external memory so as to reduce communications and reuse data as much as possible in the accelerator. This specification, which can be used in different contexts, handles the cases where data can be redefined in the accelerator and/or approximations are needed because of non-analyzable data accesses. Our second contribution is an optimized code generation scheme, entirely at source-level, that allows us to compile all the necessary glue (the communication processes) with the same HLS tool as for the computation kernel. Both contributions use advanced polyhedral techniques for program analysis and transformation. Experiments with Altera HLS tools show the correctness and efficiency of our technique.

I. Introduction

High-level synthesis tools [1], e.g., Catapult-C, C2H, Gaut, Impulse-C, Pico-Express, Spark, Ugh, provide a convenient level of abstraction (in C-like languages) to implement complex designs. Most of these tools integrate state-of-the-art back-end compilation techniques and are thus able to derive an optimized internal structure, thanks to efficient techniques for scheduling, resource sharing, and finite-state machines generation. However, integrating the automatically-generated hardware accelerators within the complete design, with optimized communications, synchronizations, and local buffers, remains a very hard task, reserved to expert designers. In addition to the VHDL glue that must sometimes be added, the input program must often be rewritten, in a proper way that is not obvious to guess. For HLS tools to be viable, these issues need to be addressed: a) the interface should be part of the specification and/or generated by the HLS tool; b) HLS-specific optimizing program restructuring should be available, either in the tool or accessible to the designer, so that high performances (mainly throughput, with limited memory size for housekeeping) can be achieved. Such high-level transformations and optimizations are standard in high-performance compilers, not yet in high-level synthesis, even if their interest has been demonstrated through hand-made designs or restructuring methodologies [2], [3], [4].

The goal of this paper is to show how such a restructuring can be *fully automated*, thanks to advanced polyhedral code analysis and code generation techniques, entirely at source level (i.e., in C). We focus on the optimization of hardware accelerators working on a large data set that cannot be completely stored in local memory, but need to be transferred from a DDR memory at the highest possible rate, and possibly stored temporarily locally. For such a memory, the throughput of memory transfers is asymmetric: successive accesses to the same DDR row are pipelined an order of magnitude faster than when the states of the finite-state machine controlling the DDR must be changed to access different rows. In other words, accessing data by blocks is a direct way of improving the performances: if not, the hardware accelerator, even highly-optimized, keeps stalling and runs at the frequency of the DDR accesses. The same situation occurs when accessing a bus for which burst communications are more efficient, when optimizing communications for GPGPUs or, more generally, when communications, between an external large memory and an accelerator with a limited memory, should be reduced (thanks to data reuse in the accelerator), pipelined, and preferably performed by blocks. This is why we believe that our techniques, although developed for HLS and specialized to Altera C2H [5], can be interesting in other contexts.

Our technique relies on loop tiling to increase the granularity of computations and communications. Each strip of tiles is optimized as follows. Transfers from and to the DDR are pipelined, in a blocking and double-buffering fashion, thanks to the introduction of software-pipelined communicating processes. Data reuse within a strip is exploited by accessing data from the accelerator and not from the DDR when already present. Local memories are automatically generated so as to store communicated data and exploit data reuse. To make this scheme possible, our main contributions are the following:

Program analysis We propose a complete specification of the sets of data to be transferred for a strip of tiles. It is general enough to handle the case where data used in a strip can also be redefined in the strip or and the case where data read and written are over-approximated.

Code generation Our code generation is parameterized by a "scheduling function" that expresses the tiling of loops and the pipelining of tiles. Based on this function, the size of local buffers, the scanning of data sets to access the DDR row-wise, and the generation of communicating processes are automatically performed.

Integration A unique feature of our technique is that the original computation kernel and all generated communicating processes are expressed in C and compiled into hardware with the same HLS tool, used as a back-end compiler.

Section II details the specification for optimized data transfers in a tile strip. Section III presents the different steps of the code generation. Section IV provides experimental results comparing the performances of the hardware accelerators generated from the program optimized by hand and from the program optimized automatically, thanks to our method. Section V concludes and gives future directions.

II. Communication optimization

Our method can be applied to accelerate a kernel onto which *loop tiling* [6] and polyhedral transformations [7] can be applied, i.e., a set of `for` nested loops, manipulating arrays and scalar variables, whose iterations can be represented by an *iteration domain* using polyhedra. This is the case when loop bounds and `if` conditions are affine expressions of surrounding loops counters and structure parameters.

A. Loop tiling and transformation function

Loop tiling is a standard loop transformation, which has proven to be effective for automatic parallelization and data locality improvement. With loop tiling, the iteration domain is partitioned into rectangular blocks (tiles) of iterations to be executed atomically. Loop tiling can be viewed as a composition of strip-mining and loop interchange. Strip-mining introduces two kinds of loops: the *tile loops*, which iterate over the tiles, and the *intra-tile loops*, which iterate in a tile. This step is always legal. Then, loop interchange pushes the intra-tile loops in the inner dimensions of the loop nest. In some cases, preliminary loop transformations, such as loop skewing, are necessary to make the loops tilable. We call *tile strip* the set of tiles described by the last tile loop, for a given iteration of the outer tile loops. This notion will be widely used in our approach, as most optimizations will be done within a tile strip, parameterized by the counters of the outer tile loops.

A loop tiling for a statement S, surrounded by n nested loops with iteration domain $\mathcal{D}_S$, can be expressed with a n-dimensional affine function $\vec{i} \mapsto \theta(S,\vec{i})$ and one (to make things simpler) size parameter b, where $\vec{i}$ is the *iteration vector* specifying the iterations of $\mathcal{D}_S$. A tile, defined by n loop counters I_1, ..., I_n, contains $\vec{i}$ if $I_k b \leq \theta(S,\vec{i}) < (I_k+1)b$, for $k \in [1..n]$. Adding these constraints, for a fixed value b, to those expressing $\mathcal{D}_S$ gives an iteration domain $\mathcal{D}'_S$ of dimension $2n$. If the transformation θ corresponds to n permutable loops, then a valid sequential schedule of the tiled code is:

$$\theta_{\text{tiled}}(S, I_1 \dots I_n, \vec{i}) = (I_1, \dots, I_n, \theta(S,\vec{i}))$$

Giving S, $\mathcal{D}'_S$, and θ_{tiled} to a polyhedral code generator will generate the tiled code [8], [9]. We point out however that we do not apply such a rewriting as a preliminary transformation as this would complicate our subsequent optimizations. Instead, all analysis and code generation steps are parameterized by the function θ. As will be explained in Section III-B, this function is also used to express the relative schedules of read, write, and computation processes and help us design the adequate local buffers in a double-buffering fashion. Actually, "double-buffering" is a language simplification: we do not really use two buffers, but one larger buffer. However, two successive blocks of computation in a tile strip are indeed pipelined with two blocks of communications, so as to overlap communications and computations. In our current implementation, we leave the choice of tiling to the user, which must be specified by means of a function θ.

B. Communication coalescing

This section presents a method to select the array regions to be loaded from and stored to the DDR for each tile. This step will impact two important criteria: a) the amount of communications with the DDR and b) the size of the local memory. At first glance, it may seem that these criteria are antagonistic. Actually, we prove that, with our scheme, this is not the case. Both can be minimized at the same time.

To perform data transfers, several solutions are possible. The most naive one is to access the DDR each time a data access is performed in the code. This solution does not require any local memory but is very inefficient: the latency to the DDR has to be paid for each access, which takes roughly 400 ns on our platform. Accesses must therefore be pipelined (a feature available in Altera HLS tool C2H) so that the accelerator throughput now depends not on the DDR latency, but on its throughput. If successively accessed data are not in the same DDR row, the accelerator is then able to receive 32 bits every 80 ns. However, if data accesses are reorganized by blocks on the same row, thanks to loop tiling, the accelerator may achieve full rate, i.e., receive 32 bits every 10 ns. But to sustain this rate and not pay any DDR latency, communications must be fully pipelined. This can be done thanks to *communication coalescing*, which amounts to host transfers out of a tile and regroup the same accesses to eliminate redundancy [10], [11].

To exploit communication coalescing, several approaches are again possible, depending on when transfers are performed. The first one is to load, just before executing a tile, all the data read in the tile, then to store to the DDR all data written in the tile. This solution, although correct, would not exploit data reuse and, unless no dependence exists between successive tiles, would forbid to overlap computations and communications. The other extreme solution is to first load all data needed in a tile strip, then to execute all tiles in the strip, before finally storing to the DDR all data produced by the strip. This would exploit data reuse but would require a large local memory to store all needed data. Furthermore, the computations would have to wait for all data to arrive before starting. The strategy we formalize in this section consists in sending load and store requests to the DDR only when needed. For each tile, we load from the DDR the data read for the *first time* in the current tile strip and we store to the DDR the data written for the *last time* in the current tile strip. Meanwhile, the data is kept and used (read and written) in the local memory, exploiting data reuse. As a bonus, the method no longer requires two consecutive tiles of a tile strip to be dependence-free. Indeed, as the data concerned by the inter-tile data dependences are kept in local memory, the sequential execution of tiles guarantees the correctness of the program.

C. General specification

For every tile t, we want to specify $\text{Load}(t)$, the data to be loaded from the DDR just before executing the tile, and $\text{Store}(t)$, the data to be stored to the DDR just after executing the tile. Let $\text{In}(t)$ be the data read in the tile t (before being possibly rewritten) and $\text{Out}(t)$ be the data written in t. Here, we assume the sets $\text{In}(t)$ and $\text{Out}(t)$ to be exact. The case of approximation is studied later (Section II-E).

Definition 1 (Valid Load)**.** *The function $t \mapsto \text{Load}(t)$ is valid if and only if (iff) the following conditions hold for any tile T.*

(i) $\cup_{t \leq T} \{\text{In}(t) \setminus \text{Out}(t' < t)\} \subseteq \text{Load}(t \leq T)$.
(ii) $\text{Out}(t < T) \cap \text{Load}(T) = \emptyset$.

The notation $\text{Load}(t < T)$ stands for $\cup_{t<T}\text{Load}(t)$ (same for the other expressions). Figure 1 illustrates Definition 1. Condition (i) means that all the data needed by the tile T, i.e., those input to the tile T but not produced by a previous tile, are loaded just before this tile or earlier. Condition (ii) means that there is no overwriting of a data already alive and modified in the buffer. This arises when a data is written in a previous tile before being read in the current tile. Without this condition, some data would possibly be loaded from the DDR and overwrite the existing value, which would be incorrect.

Definition 2 (Valid Store)**.** *The function $t \mapsto \text{Store}(t)$ is valid iff the following conditions hold ($T_{\max}$ last tile of the strip):*

(i) $\text{Out}(t \leq T_{\max}) = \text{Store}(t \leq T_{\max})$.
(ii) $\text{Store}(T) \cap \text{Out}(t > T) = \emptyset$ *for any tile* T.

Definition 2 is illustrated in Figure 2. Conditions (i) and (ii) mean that we expect to store exactly the data locally modified. Condition (ii) means that a data is stored after its last write. This is actually stronger than what is really needed for a validity condition, as a value could be stored several times. But this assumption will simplify the proofs, without hurting the correctness of the whole construction. We could also define more complex schemes, allowing for example to load from the DDR a value modified in the tile strip. But we would need to be able to guarantee that this value was already stored in the DDR and not modified again before the load. This would also imply a combined definition of the Load and Store functions.

Definition 3 (Exact Load)**.** $\text{Load}(t)$ *is exact iff the following conditions hold for any tile T:*

(i) $\cup_{t \leq T} \{\text{In}(t) \setminus \text{Out}(t' < t)\} = \text{Load}(t \leq T)$.
(ii) $\text{Load}(T) \cap \text{Load}(T') = \emptyset$ *for any tile* $T' \neq T$.

In Definition 1, Condition (i) was a simple inclusion in the validity definition, which allows to define valid solutions that load more data than needed. Now, the equality means that we load exactly the data needed and only them. The difference with $\text{Out}(t' < t)$ avoids to load the data already modified before executing the tile T. Condition (ii) means that all $\text{Load}(T)$ must be disjoint, thus forbids redundant loads, i.e., data loaded several times, as illustrated in Figure 1. Note however that this may increase the size of the local memory. But, again, this assumption simplifies our general scheme.

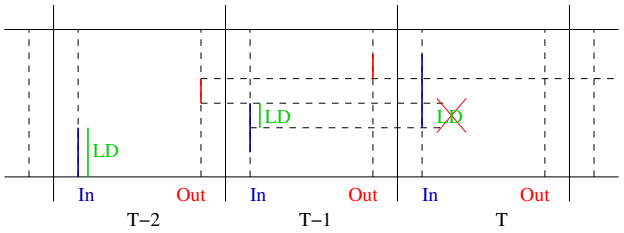

Figure 1. Valid loads refined with exact loads for a 2D example.

Definition 4 (Exact Store)**.** $\text{Store}(t)$ *is exact iff the following condition hold.*

(i) The function $t \mapsto \text{Store}(t)$ is valid.
(ii) $\text{Store}(T) \cap \text{Store}(T') = \emptyset$ *for any tiles* $T \neq T'$.

Similarly, the equality in Condition (i) means that we expect to store exactly the data modified: here, it cannot be an over-approximation otherwise the execution of the tile strip would store an undefined value to the DDR, possibly leading to an incorrect code if one of the extra stores overwrites a meaningful value. Condition (ii) means that all $\text{Store}(T)$ are disjoint, thus forbids redundant stores, i.e., a value defined by the tile strip is stored only once. This is illustrated in Figure 2.

D. Exact formulation

The previous definitions do not explicit the Load and Store operators for a given tile T. The following theorem expresses a solution, which corresponds to the case where loads are performed as late as possible and stores as soon as possible. Note that, unlike for the Store operator, an exact Load operator is completely determined by $\text{Load}(T_{\min})$, the data loaded for the very first tile $T_{\min}$ of the tile strip, as $\text{Load}(T) = \text{Load}(t \leq T) \setminus \text{Load}(t < T)$, and these two terms are fully defined by the functions In and Out.

Theorem 1. *The functions $T \mapsto \text{Load}(T)$ and $T \mapsto \text{Store}(T)$*

- $\text{Load}(T) = \text{In}(T) \setminus \{\text{In}(t < T) \cup \text{Out}(t < T)\}$
- $\text{Store}(T) = \text{Out}(T) \setminus \text{Out}(t > T)$

define load and store operators that are valid and exact.

The proof of this result and of the following theorems are provided in the appendix for the review process. Intuitively, $\text{Load}(T)$ gets all the data read in the tile T and removes data already read ($\text{In}(t < T)$) and data already alive ($\text{Out}(t < T)$). The latter contains the data read earlier, and data written earlier without a previous read, that we actually want to remove. As for $\text{Store}(T)$, it is exactly the data written for the last time in T. We select the data written in the current tile ($\text{Out}(T)$), which are not written later ($\text{Out}(t > T)$).

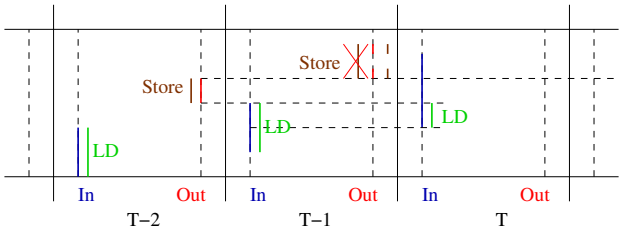

Figure 2. Valid stores refined with exact stores for a 2D example.

Provided the sets $\mathrm{In}(t)$ and $\mathrm{Out}(t)$ can be precisely defined, the previous theorem specifies how to compute the exact sets of data to load from and to store to the DDR for every tile. Otherwise, we usually have two options.

(i) Identify a subset of programs making possible an exact computation. This is the option we chose in our current implementation. The detail of the algorithm and the implementation considerations will be given later.

(ii) Deal with approximation. In this case, we need to express the validity conditions relating the operators Load and Store to the approximated In and Out, and then to exhibit such operators. We now discuss this second option.

E. A conservative approximation formulation

We now give a sufficient condition for the validity of Load and Store dealing with the approximation. We also exhibit a definition of the operators Load and Store, which is valid if some additional conditions are met. The key simplifying idea is to keep a scheme in which any data dependence within the tile strip corresponds, after load and store insertions, to a dependence in the local memory, i.e., no data is stored to the DDR and then read from it in the same tile strip.

In the following, we assume the set $\mathrm{In}(t)$ to be over-approximated by the set $\overline{\mathrm{In}}(t)$, i.e., $\mathrm{In}(t) \subseteq \overline{\mathrm{In}}(t)$ for each tile t, and the set $\mathrm{Out}(t)$ to be under- and over-approximated by the sets $\underline{\mathrm{Out}}(t)$ and $\overline{\mathrm{Out}}(t)$: $\underline{\mathrm{Out}}(t) \subseteq \mathrm{Out}(t) \subseteq \overline{\mathrm{Out}}(t)$, for each tile t. We adapt the validity conditions given by Definitions 1 and 2. The next theorems give a *sufficient* condition on the sets $\overline{\mathrm{In}}$, $\underline{\mathrm{Out}}$, $\overline{\mathrm{Out}}$ for Load and Store to be valid.

Theorem 2 (Valid approx. Load)**.** *The function* $t \mapsto \mathrm{Load}(t)$ *is valid if the following holds, for any tile* T*:*

(i) $\cup_{t \leq T} \left\{ \overline{\mathrm{In}}(t) \setminus \underline{\mathrm{Out}}(t' < t) \right\} \subseteq \mathrm{Load}(t \leq T)$.
(ii) $\overline{\mathrm{Out}}(t < T) \cap \mathrm{Load}(T) = \emptyset$.

Theorem 2 is illustrated in Figure 3. As for Definition 1, Condition (i) means that Load contains all the data read by the current tile, without those already written in previous tiles. The term $\overline{\mathrm{In}}(t)$ can cause useless data to be read, but the term $\underline{\mathrm{Out}}(t' < t)$ cannot cause already-written data to be loaded, because of Condition (ii). This condition removes from $\mathrm{Load}(t)$ the data previously written, and possibly more, depending on the approximation, but the loads are guaranteed to contain the data read, thanks to the term $\overline{\mathrm{In}}(t)$ in Condition (i). *In fine*, the approximation can cause loads of useless data, as well as redundant loads along the tile strip, but any data modified in the tile strip and read after is read from the local memory.

The conditions for Store are more tricky as, at first glance, Store does not accept any approximation. Indeed, if Store is over-approximated, extra data may be stored to the DDR, causing useful data in the DDR to be crushed. Conversely, if Store is under-approximated, we may forget to store useful outputs defined by the tile strip. Both approximations may cause an incorrect execution. Actually, an over-approximation of the Store operator can be correct if the extra data to be stored are exactly those already present in the DDR, as illustrated in Figure 4. This would not crush any data and would keep the program semantics. This condition holds when $\mathrm{Store}(T) \subseteq \mathrm{Load}(t \leq T) \cup \mathrm{Out}(t \leq T)$ for each tile T, i.e., any data stored from the local memory to the DDR and not defined by the previous tiles (T included) was previously loaded from the DDR. Then, a sufficient condition for the Store operator to be an over-approximation can be given as follows.

Theorem 3 (Valid approx. Store)**.** *The function* $t \mapsto \mathrm{Store}(t)$ *is valid if the following holds (with* $T_{\max}$ *last tile of the strip):*

(i) $\overline{\mathrm{Out}}(t \leq T_{\max}) \subseteq \mathrm{Store}(t \leq T_{\max})$.
(ii) $\mathrm{Store}(T) \cap \overline{\mathrm{Out}}(t > T) = \emptyset$ *for any tile* T.
(iii) $\mathrm{Store}(T) \subseteq \overline{\mathrm{In}}(t \leq T) \cup \underline{\mathrm{Out}}(t \leq T)$ *for any tile* T*, when used with a valid approximated* Load*.*

We now exhibit Load and Store operators verifying these conditions. The worst-case solution is to load, before the first tile, all data potentially read in the tile strip and to store, after the last tile, all data potentially written in the tile strip, i.e., $\mathrm{Load}(T_{\min}) = \overline{\mathrm{In}}(t \leq T_{\max})$ and $\mathrm{Load}(t) = \emptyset$ if $t \neq T_{\min}$, $\mathrm{Store}(T_{\max}) = \overline{\mathrm{Out}}(t \leq T_{\max})$ and $\mathrm{Store}(t) = \emptyset$ if $t \neq T_{\max}$. According to Theorems 2 and 3, this scheme is valid if $\overline{\mathrm{Out}}(t \leq T_{\max}) \subseteq \overline{\mathrm{In}}(t \leq T_{\max}) \cup \underline{\mathrm{Out}}(t \leq T_{\max})$, which means that we should also pre-load all data that cannot be proved to be defined in the tile strip, i.e., $\mathrm{Load}(T_{\min}) = \overline{\mathrm{In}}(t \leq T_{\max}) \cup \left\{ \overline{\mathrm{Out}}(t \leq T_{\max}) \setminus \underline{\mathrm{Out}}(t \leq T_{\max}) \right\}$. More generally, consider the following operators (compare with Theorem 1):

$$\mathrm{Load}(T) = \overline{\mathrm{In}}(T) \setminus \left\{ \overline{\mathrm{In}}(t < T) \cup \overline{\mathrm{Out}}(t < T) \right\} \quad (1)$$

$$\mathrm{Store}(T) = \overline{\mathrm{Out}}(T) \setminus \overline{\mathrm{Out}}(t > T) \quad (2)$$

First note that, if the sets $\mathrm{Out}(T)$ are not approximated, i.e., $\underline{\mathrm{Out}}(T) = \mathrm{Out}(T) = \overline{\mathrm{Out}}(T)$, then there is no problem: the operators Load and Store are both valid (and even exact w.r.t. the sets $\overline{\mathrm{In}}(T)$) with the same proof as in Thm. 1. The difficulty arises when the sets $\mathrm{Out}(T)$ are not exact, as we now show.

It is easy to see that $\mathrm{Store}(t \leq T_{max}) = \overline{\mathrm{Out}}(t \leq T_{max})$ and that $\mathrm{Store}(T) \cap \overline{\mathrm{Out}}(t > T) = \emptyset$ for any tile T, hence Conditions (i) and (ii) of Theorem 3 are satisfied. The function

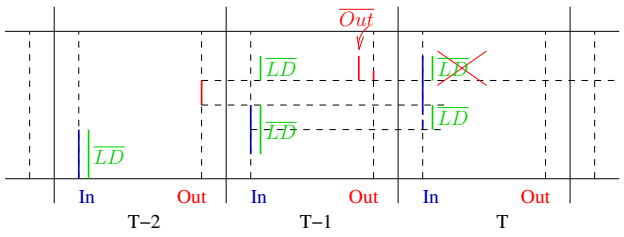

Figure 3. Valid approximated loads for a 2D example.

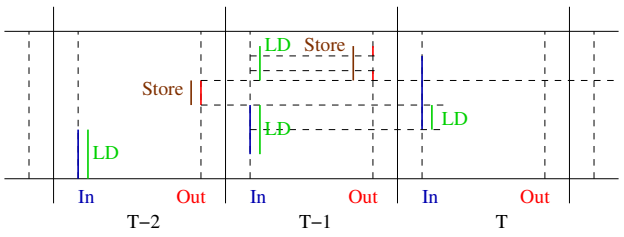

Figure 4. Valid approximated stores for a 2D example.

$t \mapsto \text{Store}(t)$ is valid if Condition (iii) is fulfilled, i.e., if $\text{Store}(T) \subseteq \overline{\text{In}}(t \leq T) \cup \underline{\text{Out}}(t \leq T)$ holds for any tile T, in other words $\overline{\text{Out}}(T) \subseteq \overline{\text{In}}(t \leq T) \cup \underline{\text{Out}}(t \leq T) \cup \overline{\text{Out}}(t > T)$. This condition can always be fulfilled by increasing the over-approximation $\overline{\text{In}}$ of In, i.e., with extra pre-loads. Unfortunately, this is not as simple with the Load operator. Indeed, the over-approximation on $\text{Out}(t < T)$ can prevent effective input data to be loaded. As is, this definition does not directly comply with the validity conditions of Theorem 2. We provide additional constraints on $\overline{\text{In}}$, $\overline{\text{Out}}$, and $\underline{\text{Out}}$, which are sufficient to ensure the validity of the operators Load and Store. What is needed is again to pre-load values that are needed but that cannot be proved to be defined in the tile strip.

Theorem 4. *The operators in Equations 1 and 2 define valid load and store operators if the following holds for any tile T:*
(i) $\overline{\text{Out}}(T) \subseteq \overline{\text{In}}(t \leq T) \cup \overline{\text{Out}}(t > T) \cup \underline{\text{Out}}(t \leq T)$.
(ii) $\overline{\text{In}}(T) \cap \left\{\overline{\text{Out}}(t < T) \setminus \underline{\text{Out}}(t < T)\right\} \subseteq \overline{\text{In}}(t < T)$.

This construction could be used to design a conservative analysis that would cope with every kind of programs, providing that Constraints (i) and (ii) of Theorem 4 are met. How to compute the sets $\overline{\text{In}}$, $\underline{\text{Out}}$, and $\overline{\text{Out}}$ verifying these constraints is beyond the scope of this thesis, and is left for future work. But, still, one can already noticed that a direct approach consisting in over-approximating the sets $\overline{\text{In}}(T)$ into sets $\overline{\overline{\text{In}}}(T)$ will work. Indeed, the constraints on sets $\overline{\text{In}}(T)$ always give a "lower bound", but no "upper bound". For example, a simple solution to solve Constraint (i) is to add $\overline{\text{Out}}(T) \setminus \underline{\text{Out}}(T)$ to $\overline{\text{In}}(T)$. Then, Constraint (ii), starting from these new sets $\overline{\text{In}}(T)$, define new lower bounds, by induction, for decreasing values of T. The induction can be avoided and approximated by defining directly $\overline{\overline{\text{In}}}(T) = \cup_{t>T} \left\{\overline{\text{In}}(t) \cap (\overline{\text{Out}}(t' \leq T) \setminus \underline{\text{Out}}(t' < t))\right\} \cup \overline{\text{In}}(T)$.

Note also that any over-approximation of $\text{Load}(T)$ is valid as long as Condition (ii) of Theorem 2 is satisfied. All these theoretical results give opportunities for handling cases where program analysis cannot be performed exactly or when approximating Load and Store sets allows a better packing of data to be transferred. Again, this optimization is left for future work. We believe also that abstract interpretation techniques could be used to compute the sets $\overline{\text{In}}$, $\underline{\text{Out}}$, and $\overline{\text{Out}}$, as in [12]. Then, a post-processing is needed to meet Constraints (i) and (ii), as explained above with the sets $\overline{\overline{\text{In}}}(T)$.

III. Algorithms for analysis and code generation

The previous section provides a specification of optimized data transfers, which is more general than state-of-the-art communication vectorization and coalescing. Indeed, standard communication optimizations can eliminate redundant successive reads and redundant successive writes, but a data that is read and written alternatively several times in the tile strip will generate as many remote accesses. With our technique, the data is loaded and stored back only once: in the meantime, it is locally stored and possibly modified, thus reducing communications and increasing local reuse. To make this possible, we need to show how we compute the set of data stored locally (Section III-A), how we map them to local memory (Section III-B), and how we generate the corresponding communication processes (Section III-C).

A. An exact solution for computing the Load and Store sets.

In our current implementation, we focused on kernels in the polytope model [13], where exact analysis is possible. For this framework, we developed an algorithm, detailed in this section, that implements the exact Load and Store operators specified in the previous section. It computes the expressions of $\text{Load}(T)$ and $\text{Store}(T)$ for a *parametric* (symbolic) tile T, which are used to generate the C communicating functions.

Our algorithm specifies the function Load, not directly with the sets In and Out as in Theorem 1, but thanks to the computation of FirstRead, the set of operations responsible for a *first read* within the *parameterized tile strip* considered, i.e., a read from a memory (array) location that has never been read before in the tile strip. Theorem 5 gives the link with Theorem 1. If values read in the tile strip are not written earlier in the tile, we can define $\text{Load}(T)$ to be the first reads that belong to the tile T, i.e., $\text{FirstRead} \cap T$. Actually, $\text{FirstRead} \cap T$ is a set of operations, not a set of data as $\text{Load}(T)$, so we write $\text{Load}(T) = \text{Input}(\text{FirstRead} \cap T)$ where Input gives the data read by a set of operations (defined at the granularity of a memory access). $\text{FirstRead} \cap T$ contains more information than $\text{Load}(T)$, which will be helpful for code generation. Similarly, $\text{Store}(T)$ is obtained from LastWrite, the operations responsible for a *last write* within the tile strip, i.e., $\text{Store}(T) = \text{Output}(\text{LastWrite} \cap T)$, where Output gives the data written by a set of operations.

Example 1. *Figure 5 shows how we apply double buffering and communication coalescing on the matrix multiply example. The loops are tiled along the canonical directions, which induces communications for matrices a and b (by blocks) while the matrix c is modified locally. We write (I, J, K) the loop counters of the tile loops, while (i, j, k) iterates into the tile specified by (I, J, K), which is defined, for a size 32, by:*

$$0 \leq i, j, k \leq N \wedge 32I \leq i \leq 32I + 31 \wedge$$
$$32J \leq j \leq 32J + 31 \wedge 32K \leq k \leq 32K + 31$$

The tile size can be specified as an input by the user, but (so far) cannot be parameterized, as this would lead to non-affine constraints. The tile strip defined by I and J is obtained by projection of the previous inequalities. Then, all expressions

```
for(i=0; i<=N; i++) {
  for(j=0; j<=N; j++) {
    for(k=0; k<=N; k++) {
      c[i][j] = c[i][j]+
        a[i][k]*b[k][j];
    }
  }
}
```

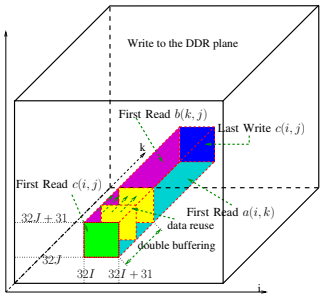

Figure 5. Applying the double-buffering scheme on matrix multiply.

are given in terms of I and J, i.e., parameterized by the tile strip. *This feature is essential. For the tile strip (I, J), we get:*

Array	FirstRead()	LastWrite()
$a(i,j)$	$(S, i, 32J, j)$	$\emptyset$
$b(i,j)$	$(S, 32I, j, i)$	$\emptyset$
$c(i,j)$	$(S, i, j, 0)$	(S, i, j, N)

In a more general situation, defining the operator Load just with the first reads is incorrect. Indeed, as explained earlier, for an array c, the first reads of $c(\vec{i})$ that are preceded by a write of $c(\vec{i})$ should not be loaded, otherwise, this will cause a load to overwrite the value of $c(\vec{i})$. To avoid this situation, one can define the *first read* of $c(\vec{i})$ to be the first executed operation among all the reads of $c(\vec{i})$ and the writes of $c(\vec{i})$. Then, to define Load(T), it suffices to restrict to the reads, the writes corresponding to the forbidden case. This way, the operators Load and Store are exact in the meaning of Definitions 3 and 4.

Theorem 5. *The following operators are valid and exact:*

$$\begin{aligned} \text{Load}(T) &= \text{Input}(\text{FirstRead} \cap T) \\ \text{Store}(T) &= \text{Output}(\text{LastWrite} \cap T) \end{aligned}$$

Proof: By construction, Load(T) contains all the first reads (with the definition that takes into account previous writes) in T, thus all data that are read in T, not read earlier, and not defined earlier. Thus, Load(T) is exactly defined as in Theorem 1 by $\{\text{In}(T) \setminus \text{In}(t < T)\} \setminus \text{Out}(t < T)$. Similarly, Store$(T)$ contains all the last writes in T, which means all the data written in T and not written later again. In other words, $\text{Load}(T) = \text{Out}(T) \setminus \text{Out}(t > T)$. Theorem 1 then shows that the sets Load and Store are valid and exact. ■

We now describe how to compute the sets FirstRead (the sets LastWrite are computed in a similar way). For the sake of clarity, we first explain how to compute FirstRead among reads only, then how to modify the algorithm to get FirstRead among reads and writes, and how to filter the actual reads.

For each array c, FirstRead(c) is obtained by first extracting the set of operations reading a given $c(\vec{i_0})$, where $\vec{i_0}$ is a *parameter*. Then, we compute the read which is scheduled first in the tile strip, which boils down to compute the lexicographic minimum for a union of polytopes, as for instance-wise exact data-flow analysis [14]. Then FirstRead(c) is given as a discussion on the parameters value, including the array cell $\vec{i_0}$.

In the polytope model, all reads to c can be written as:

$$S_\ell : \vec{i} \in D_\ell : \ldots = \ldots c[u_\ell(\vec{i})] \ldots$$

where D_ℓ is the iteration domain of the statement S_ℓ, $\vec{i}$ is an iteration vector, and u_ℓ is an *affine function*. The reads of $c(\vec{i_0})$ in S_ℓ is the set of operations $(S_\ell, \vec{i})$ that read $c(\vec{i_0})$, i.e., $u_\ell(\vec{i}) = \vec{i_0}$, and that are actually executed, i.e., $(\vec{i} \in D_\ell)$:

$$\text{Read}(c, S_\ell) = \{\vec{i} \in D_\ell \mid u_\ell(\vec{i}) = \vec{i_0}\}$$

If c occurs once in S_ℓ, Read(c, S_ℓ) is a polytope (actually the integer points in a polytope). Otherwise, in general, the result of Read is (the integer points in) a *union of polytopes*.

Every statement S is given an affine schedule θ_S, assigning an execution date to every iteration vector. This schedule is obtained from the tiling specified by the user, as discussed in Section II-A. We extend the definition of Read by providing the execution date of $\vec{i}$, $\vec{t} = \theta_{S_\ell}(\vec{i})$ together with $\vec{i}$:

$$\text{Read}(c, S_\ell) = \{(\vec{t}, \vec{i}) \mid \vec{t} = \theta_{S_\ell}(\vec{i}) \wedge u_\ell(\vec{i}) = \vec{i_0} \wedge \vec{i} \in D_\ell\}$$

To find the first instance of S_ℓ reading the cell $c(\vec{i_0})$, we need to get the pair $(\vec{t}, \vec{i})$ in Read(c, S_ℓ) that minimizes the execution date $\vec{t}$, i.e., the lexicographic minimum $(\vec{t}_{\text{FirstRead}}, \vec{i}_{\text{FirstRead}}) = \min_{\ll} \text{Read}(c, S_\ell)$. The first instance of S_ℓ reading $c(\vec{i_0})$ is $(S_\ell, \vec{i}_{\text{FirstRead}})$, with execution date $\vec{t}_{\text{FirstRead}}$. To compute FirstRead$(c)$, we to proceed the same way for every assignment reading c, getting as many local first instances, and to compute the global minimum, which is still a lexicographic minimum. In other words, if $S_1, \ldots, S_n$ denote the assignments reading c, the global FirstRead(c) is:

$$\text{FirstRead}(c) = \min_{\ll}(\text{Read}(c, S_1) \cup \ldots \cup \text{Read}(c, S_n))$$

which is computed with the equivalent form:

$$\text{FirstRead}(c) = \min_{\ll}(\min_{\ll} \text{Read}(c, S_1), \ldots, \min_{\ll} \text{Read}(c, S_n))$$

The inner lexicographic minima apply on polytopes that depend on parameters such as $\vec{i_0}$ (the array cell whose first read is searched). As $\vec{i_0}$ is unknown, the result is a discussion on $\vec{i_0}$, giving for such or such domain the corresponding pair $(\vec{t}_{\min}, \vec{i}_{\min})$. We actually get a set of clauses:

$$\left[\vec{n} \in D'_1 : (\vec{t}_{\min_1}, \vec{i}_{\min_1})\right] \vee \ldots \vee \left[\vec{n} \in D'_p : (\vec{t}_{\min_p}, \vec{i}_{\min_p})\right]$$

where $\vec{n}$ is the vector of parameters (including $\vec{i_0}$), and $\vec{t}_{\min_k}$ and $\vec{i}_{\min_k}$ are *affine expressions of the parameters*. Parametric integer linear programming [15], widely used in program analysis in the polytope model, gives the result as a selection tree on $\vec{n}$, the QUAST (quasi-affine selection tree).

The outer minimum is the lexicographic minimum of a set of clauses, obtained by standard combination techniques [14]. We also tag each set of clauses with the corresponding assignment (S_ℓ), so it will be remembered during the combination. Finally, the result gives FirstRead(c) as a set of tagged clauses:

$$\left[\vec{n} \in D'_1 : (S_{\ell_1}, \vec{i}_{\min_1})\right] \vee \ldots \vee \left[\vec{n} \in D'_p : (S_{\ell_p}, \vec{i}_{\min_p})\right]$$

The vector of parameters $\vec{n}$ includes $\vec{i_0}$ and also the counters defining the tile strip being considered (I and J in the matrix-multiply example). Also, our method works with any affine (multi-dimensional) schedule, which makes it very general.

This explained how to compute FirstRead(t) among reads only. A simple modification allows to compute FirstRead(t) among reads and writes. Each time S_ℓ writes the array cell $c[v_\ell(\vec{i})]$, it suffices to complete the set Read(c, S_ℓ) with $(\vec{t}, \vec{i}) \mid \vec{t} = \theta_{S_\ell}(\vec{i}) \wedge v_\ell(\vec{i}) = \vec{i_0} \wedge \vec{i} \in D_\ell\}$. We now get two sets of clauses $R_{\min}(k)$ and $W_{\min}(k)$. We tag $W_{\min}(k)$, then we compute the global minimum with the method described above. Then, it suffices to remove from the final system of clauses, the tagged clauses coming from a $W_{\min}(k)$, i..e, to not consider the leaves of the QUAST tagged with writes.

B. Local memory management and code generation

With our method, the variables are loaded in the local memory just before needed and then stored back to the DDR at the expiration of their lifetime in the tiled strip. Meanwhile, the computations are done in the local memory, using the temporary images of the variables. We now explain *how* they are stored in the local memory. Each variable must be mapped to the local memory in such a way that (i) two data live at the same time cannot be mapped to the same local address, and (ii) the local memory size must be as small as possible.

Unlike the methods developed in [16], [17], which try to pack data optimally (in size), possibly with complex and expensive mapping functions and reorganization, we prefer to rely on array contraction based on modular mappings [18]. In this framework, an array cell $a(\vec{i})$ is mapped to a local array cell $a\_tmp(\sigma(\vec{i}))$ where $\sigma(\vec{i}) = A\vec{i} \bmod \vec{b}$, A is an integer matrix, and $\vec{b}$ is an integral vector defining a modulo operation component-wise. In many cases, the array index functions are *uniform* (i.e., they are translations with respect to the loop indices as in `a[i][j-1]`) and the program reads and writes consecutive array cells. The set of live array cells is a window sliding during a tiled program execution, allowing efficient memory optimizations [19]. The framework presented in [18] generalizes this situation, given an analysis of live array cells.

Back to Example 1. *As shown earlier, for every (complete) tile (I, J, K), the following region is loaded in a_tmp:*

$$\text{Load}_{\text{a}}(I, J, K) = \text{a}[bI : bI + b - 1][bK : bK + b - 1]$$

(Here b, the tile size, is 32). Meanwhile, the double buffering process loads in a_tmp the data for the next tile $(I, J, K+1)$:

$$\text{Load}_{\text{a}}(I, J, K+1) = \text{a}[bI : bI + b - 1][bK + b : bK + 2b - 1]$$

This means that at the same time*, a_tmp needs at most $b \times 2b$ array cells: b cells in the first dimension and $2b$ cells in the second dimension. Now, the issue is to map the array a to the local array a_tmp. This can be done thanks to the mapping:*

$$\sigma : \text{a}[i][k] \mapsto \text{a\_tmp}[i \bmod b][k \bmod 2b]$$

Here, the mapping corresponds exactly to two blocks, used in a double-buffering manner. In general, the situation can be more complex, in particular due to local reuse in the tile strip.

The principles of the array contraction technique developed in [18] for the theoretical part and [20] for the program analysis part are the following. First, a conflict relation $\bowtie$ is defined, relating array cells whose lifetime conflict: $a(\vec{i}) \bowtie a(\vec{j})$ if the lifetime intervals of $a(\vec{i})$ and $a(\vec{j})$ are not disjoint. From the relation $\bowtie$, the *conflict polyhedron* $DS = \{\vec{i} - \vec{j} \mid a(\vec{i}) \bowtie a(\vec{j})\}$ is derived, which represents the sliding window mentioned above. Then, an *admissible lattice* for DS is built, i.e., an integer lattice L such that $DS \cap L = \{0\}$. A mapping σ is finally derived from L so that $\ker \sigma = L$. For our implementation, we use the tool CLAK made available by the authors of [20]. It takes as input a polyhedron (the set DS) and produces an admissible lattice for DS and a corresponding mapping σ.

Back to Example 1. *For array a, for a given tile, the region $[bI : bI + b - 1][bK : bK + b - 1]$ is used while the region $[bI : bI + b - 1][bK + b : bK + 2b - 1]$ is loaded. Thus, the conflict polyhedron is $DS = [-b+1; b-1] \times [-2b+1; 2b-1]$.*

The integer lattice L generated by the vectors $(b, 0)$ and $(0, 2b)$ is the smallest admissible lattice for DS: it has determinant $2b^2$ and, during program execution, $2b^2$ array cells are simultaneously live. In general however, the best modular mapping may lead to a size larger than the maximal number of simultaneously-live array cells, though acceptable.

From the lattice $L = (b, 0)\mathbb{Z} + (0, 2b)\mathbb{Z}$, the mapping $\sigma(i, j) = (i \bmod b, j \bmod 2b)$ is derived. It has $\ker \sigma = L$.

Given a schedule θ defining an execution order $\ll$, two array cells $a(\vec{i})$ and $a(\vec{j})$ conflict (see Figure 6) if there exist a read R_i of $a(\vec{i})$, a read R_j of $a(\vec{j})$, a write W_i of $a(\vec{i})$ executed before R_i, a write W_j of $a(\vec{j})$ before R_j, such that:

$$\theta(W_i) \ll \theta(R_j) \wedge \theta(W_j) \ll \theta(R_i)$$

This amounts to consider that an array cell is live from its very first write to its very last read, even when it is written several times and live only on several smaller "intervals". For our specific usage, this means that a local array cell is considered live from the time it is loaded to its last use in the tile strip. This is exactly the semantics we need, if we can express θ.

To express θ, we need to derive from the original kernel a double-bufferized version with the adequate schedules, then to apply the method mentioned above to contract the local arrays. It is actually sufficient to specify the double-bufferized version as a system of clauses expressing the loads, the reads/writes in the tile strip, and the stores, together with a schedule. This is done by defining load, compute, and store clauses as follows.

For each array a, we compute FirstRead(a) as described previously. We get a system of clauses:

$$\text{FirstRead}(a) = \left[\vec{i} \in D_1 : S_1(\vec{i})\right] \vee \ldots \vee \left[\vec{i} \in D_n : S_n(\vec{i})\right]$$

To get the restriction on a tile T, we intersect each domain D_i with T, to get the expression of FirstRead(a, T) as:

$$\left[\vec{i} \in (D_1 \cap T) : S_1(\vec{i})\right] \vee \ldots \vee \left[\vec{i} \in (D_n \cap T) : S_n(\vec{i})\right]$$

For further clause manipulations, we rewrite properly each $S_k(\vec{i})$ as a load into a_tmp. If the access to a corresponding to the first read in $S_k(\vec{i})$ is $a(u_k(\vec{i}))$, we redefine $S_k(\vec{i})$ as $a\_tmp(u_k(\vec{i})) = a(u_k(\vec{i}))$ (this is before array contraction). Then, we add these clauses to the set $\mathcal{L}$ of *load clauses*. Similarly, we write the original kernel array accesses as a system of clauses. As the computations operate on local arrays, we substitute each array a by its local image a_tmp. We add these clauses to the set $\mathcal{C}$ of *compute clauses*. For the stores, we replace each last write referenced as $a(v_k(\vec{i}))$ in $S_k(\vec{i})$ by $a(v_k(\vec{i})) = a\_tmp(v_k(\vec{i}))$ (again before array contraction) and we add these clauses to the set $\mathcal{S}$ of store clauses.

It remains to specify for each clause of $\mathcal{L}$, $\mathcal{C}$, and $\mathcal{S}$ the schedule specifying the double-buffering execution order. As the double-buffering scheme operates on blocks of two tiles,

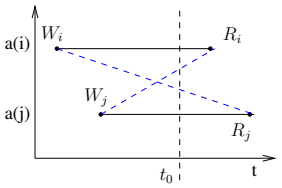

Figure 6. Condition $a(i) \bowtie a(j)$.

Fct	Schedule	Fct	Schedule
$\mathcal{L}_0$	$(K,0)$		
$\mathcal{C}_0$	$(K,1)$	$\mathcal{L}_1$	$(K-1,1)$
$\mathcal{S}_0$	$(K,2)$	$\mathcal{C}_1$	$(K-1,2)$
		$\mathcal{S}_1$	$(K-1,3)$

Figure 7. Pipelined schedule.

the schedule cannot be specified directly as an affine function. To overcome this issue, we partition each of the sets of clauses $\mathcal{L}$, $\mathcal{C}$, and $\mathcal{S}$ in two parts, emulating a loop unrolling by 2. We define $\mathcal{L}_0$, $\mathcal{C}_0$, and $\mathcal{S}_0$ (resp. $\mathcal{L}_1$, $\mathcal{C}_1$, and $\mathcal{S}_1$) the restriction to *even* (resp. *odd*) tiles. If K is the innermost tile counter (iterating on the tile strip), it suffices to add the constraint $K = 2p$ (resp. $K = 2p+1$) to each domain of $\mathcal{L}$, $\mathcal{C}$, and $\mathcal{S}$, where p is a fresh integer variable. Now, it is easy to specify the double buffering with an affine schedule θ_{db}. For example, the schedule corresponding to the scheme described in Figure 8 is given in the table of Figure 7.

This schedule is affine and specifies the same double-buffering execution order. It is actually very coarse-grain, as each function is assumed to execute its operations in parallel. Of course, this is not the case, but this is actually sufficient to specify the conflicts among the local variables, the only important thing for array contraction. Finally, we get the mapping functions σ for each local array by applying the array contraction technique we described before, to the program defined by the set of clauses $\mathcal{L}_0$, $\mathcal{C}_0$, $\mathcal{S}_0$, $\mathcal{L}_1$, $\mathcal{C}_1$, and $\mathcal{S}_1$, and the schedule θ_{db}, focusing on accesses to local arrays.

C. Generation of load, compute, and store kernels

It remains to generate the final, `C2H`-compliant, C program implementing the double-bufferized input kernel version. We generate 5 functions (we call them *drivers*) that are translated by `C2H` into separate hardware accelerators. For each tile strip, the function Compute() processes all the tiles sequentially, whereas the functions Load0(), Load1(), Store0(), and Store1() process the tiles two-by-two, starting from the first (resp. second) tile for Load0() and Store0() (resp. Load1() and Store1()). Each driver contains a loop nest iterating over the tiles. For each tile $(T_1, \ldots, T_n)$, a piece of code (we call it *kernel*) is executed performing the required loads, computations, or stores. The drivers are meant to be run in parallel and respect the double-buffering schedule thanks to synchronization signals, as depicted in Figure 8. Dotted arrows represent kernel-level synchronizations used to sequentialize the accesses to the DDR. These synchronizations, implemented as blocking reads and writes in FIFOs of size 1, are sent as soon as the last request to the DDR within a tile is done, to avoid the penalty due to the finite-state machine (FSM) structure of loops in `C2H`, and are thus embedded in the corresponding kernel.

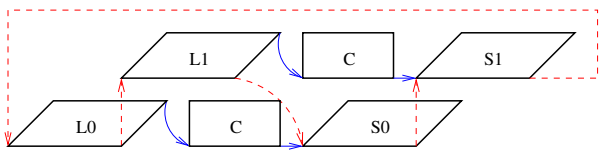

Figure 8. Pipelined double-buffered synchronization signals and schedule.

The remaining synchronizations, represented with blue arrows are not embedded in the kernels but outside, to make sure that DDR latencies are respected. This way, computations and communications are pipelined and latencies are hidden. The subtleties of this implementation and the interaction with the specificities of `C2H` are out of the scope of this paper. [1] All input programs are compiled according to this template. The tricky part is to feed the template with the kernels for loads, computations, and stores. We now show how this is done.

Code generation in the polytope model has been addressed with success leading to powerful methods and tools such as CLooG [8]. A direct approach is to feed ClooG with the system of clauses $\mathcal{L}$, $\mathcal{C}$, and $\mathcal{S}$ defined previously, together with a sequential schedule. In our context however, this gives a correct but inefficient code. A better solution is to generate each kernel as a single "linearized" loop executing one instruction per iteration. This has several benefits:

- In `C2H`, data fetches in loops are pipelined to hide latency: a special state is added (after a precomputed constant number of cycles) which stalls the FSM until the data is received. After a loop, the long pipeline of the loop must be drained, resulting in an important penalty. Linearizing nested loops avoids these nested penalties.
- Reading or writing successively in different rows of the DDR degrades the throughput. With a single loop, iterating sequentially on different data sets (thanks to `if` statements), we can specify the order in which loads and stores are scanned. Furthermore, such a loop is nicely pipelined with `C2H`, with one DDR access per iteration.

Such a linearization can be done with the Boulet-Feautrier algorithm [9]. The program to be generated is specified as a system of clauses: $\vec{i} \in D_1 : S_1(\vec{i}) \vee \ldots \vee \vec{i} \in D_n : S_n(\vec{i})$ and a schedule θ. The algorithm generates two functions, First() and Next(). **First()** gives the first operation to be executed, according to the specified schedule θ. This amounts to compute a lexicographic minimum, as for the FirstRead and LastWrite functions. The result is a selection tree giving the first operation depending on parameters. **Next()** maps an operation $S_\ell(\vec{j})$ to its immediate successor in the execution order specified by θ. The computation is similar to First(), with the additional constraint that the result must be executed after $S_\ell(\vec{j})$. Again, a selection tree is generated that gives the next operation depending on parameters value, including $\vec{j}$. When the next operation does not exist, the tree leads to a leaf with a special operation $\perp$. The final code looks like:

```
ω := First();
while(ω ≠ ⊥) {
  Execute(ω);
  ω := Next(ω);
}
```

In this code, Execute(ω) is a macro in charge of executing the operation ω, typically a single load request.

To achieve spatial locality in the DDR accesses, we scan the loads (same for stores) as follows. If the read of a_ℓ in $S_k(\vec{i})$ is done with the reference $a_\ell(u_k(\vec{i}))$, we attach,

[1] Due to double blind review, we cannot refer to any publication where all this is explained. We will add a reference if this work is accepted.

to each clause $\{\vec{i} \in (D_k \cap t) : S_k(\vec{i})\}$ of the set of clauses $\mathcal{L}$, the schedule $\theta_{\text{gen}}(S_k, \vec{i}) = (\ell, u_k(\vec{i}))$. This defines a lexicographic order: the first dimension ℓ implies that the arrays are read one after the other, with no interleaving. The second dimension $u_k(\vec{i})$ implies that the array cells are loaded in the increasing lexicographic order of the indices, thus as much as possible row by row. Giving $\mathcal{L}$ together with θ_{gen} to the Boulet-Feautrier method gives the desired load kernel.

We point out that the generation of selection trees, in particular for **Next()**, must be done carefully to avoid a code blow-up. For each clause, a selection tree is computed, which represents the closest instance to $S_\ell(\vec{j})$ w.r.t. θ. These selection trees are then combined to obtain the final tree, using several tricks and simplifications techniques, similar to those described in [14]. Combining the selection trees by increasing distance to $S_\ell(\vec{j})$ leads to drastic simplifications. Then, several pattern-matching recipes are applied to remove unreachable branches and to simplify the tree. As simplifications are applied on the fly, this also reduces the global time complexity of the process.

To conclude, the fact that all these methods – the computation of first reads and last writes (as data-flow analysis), the computation of a modular mapping for designing local buffers, and the scanning of polytopes for kernel generation – can be expressed for a given schedule θ makes the whole technique transparent, without even generating an initial loop tiling.

IV. Experimental results

We implemented our methods using the polyhedral libraries PIP [15] and Polylib [21]. Our prototype takes as input the C source code of a small kernel to be optimized and generates a C source code, which fully implements a `C2H`-compliant double-bufferized optimization. The input parameters, such as the loop tiling, are specified with pragmas in the source code. The HLS tool `C2H` is then used to automatically generate the complete hardware implementation, with its interface. The designs were synthesized on the Altera Stratix II `EP2S180F1508C3` FPGA, running at 100 MHz, and connected to an outside DDR memory, of specification JEDEC DDR-400 128 Mb x8, CAS 3.0, running at 200 MHz.

Even for elementary kernels, generating adequate C codes that can be automatically synthesized with no additional handmade VHDL glue, while exploiting the maximal DDR bandwidth, is tricky. With C2H, we showed that this is feasible, if codes and synchronizations are written in a specific, though generic, way. We first optimized by hand several elementary kernels for which communication re-organization is mandatory. We used this preliminary study to design our automatic code generator and evaluate its quality. Achieving the same performance automatically was already a challenge. The first kernel is a DMA transfer from and to the DDR:

```
for (i=0; i<n; i++) b[i] = a[i];
```

The second kernel sums 2 vectors, using 3 non-aliasing arrays:

```
for (i=0; i<n; i++) c[i] = a[i] + b[i];
```

The third kernel, the matrix-multiply example of Figure 5, is already more involved: the original code has only a few lines

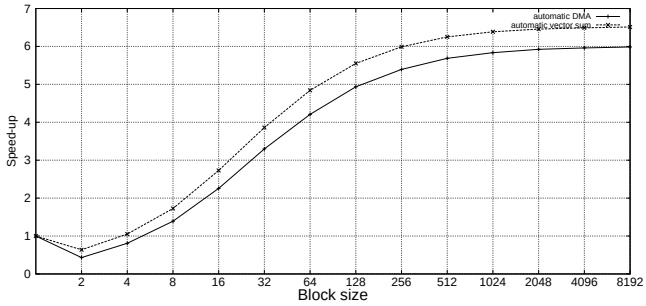

Figure 9. Automatically-optimized vs. original code: DMA and vector sum.

but the version optimized by hand (more or less a double-buffered matrix multiply by block) has more than 500 lines!

We used ModelSim to evaluate our solution. As depicted in Figures 9 and 10, the optimized versions can run 6x or more faster than the direct implementations (remember that the maximal speed-up is at most 8, if we start from a code where successive DDR accesses are in different rows). Note again that these speed-ups are obtained not because the computations are parallelized (tiles are run in sequential), not only because the communications are pipelined (this is also the case in the original versions), but because the DDR requests are reorganized so that successive accesses are on the same row as much as possible, because some communications overlap computations, and because some data reuse is exploited.

However, to achieve this, there is a price to pay from the point of view of hardware resources, in addition to the local memories involved to store data locally. This is illustrated by Figure 11, which gives different parameters measuring the hardware usage: the number of look-up tables (column "ALUT"), of registers ("Ded. reg."), of dedicated registers and additional registers used by the synthesis tool ("Total reg."), and of hard 9-bit multiplication IP cores ("DSP block"). Also, compared to the manually-transformed versions, the automatic versions use slightly more ALUT and registers, mostly because they use two separate FIFOs for synchronization between the drivers Load0() and Load1(), and the driver Compute() (we changed the design to make it more generic). They also use more multipliers to perform tile address calculations, which could be removed by standard strength reduction techniques.

The optimized versions also have a slightly smaller maximum running frequency than the original designs (see column "Max. freq." in MHz). This is mostly due to the Avalon interconnect routing. However, if the designs already saturate

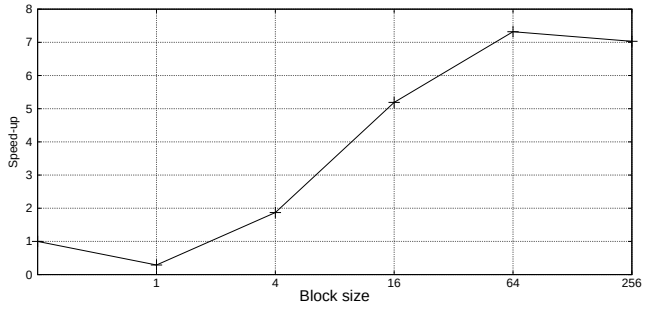

Figure 10. Automatically-generated vs. original code: matrix multiply.

Kernel	ALUT	Ded. reg.	Total reg.
System alone	4406	3474	3606
DMA original	4598	3612	3744
DMA manual opt.	9853	10517	10649
DMA automatic	11052	12133	12265
Vector sum original	5333	4607	4739
Vector sum manual opt.	10881	11361	11493
Vector sum automatic	11632	13127	13259
Mat. mul. original	6452	4557	4709
Mat. mul. manual opt.	15255	15630	15762
Mat. mul. automatic	24669	32232	32364

Kernel	DSP block	Max freq.	Speed-up
System alone	8	205.85	
DMA original	8	200.52	1
DMA manual opt.	8	162.55	6.01
DMA automatic	48	167.87	5.99
Vector sum original	8	189.04	1
Vector sum manual opt.	8	164	6.54
Vector sum automatic	48	159.8	6.51
Mat. mul. original	40	191.09	1
Mat. mul. manual opt.	188	162.02	7.37
Mat. mul. automatic	336	146.25	7.32

Figure 11. Synthesis results of DMA, vector sum, and matrix multiply kernels, for original code, manual optimization, and automatic generation.

the memory bandwidth at 100 MHz, running the systems at a higher frequency will not speed up them anyway. Compared to the manually-optimized versions, the reduction of the maximal frequency could come from more complex codes and the use of double-port memories available in the FPGA. The use of such memories induces additional synthesis constraints.

V. Conclusion

In the context of high-level synthesis (HLS) of hardware accelerators mapped on FPGA, we proposed an automatic translation method to optimize, at source level, a kernel linked to an external DDR memory. The result is a set of C functions, translated by the C-to-VHDL compiler of Altera `C2H` in separate communicating hardware accelerators, which exploit as much possible the bandwidth to the DDR. Our method relies on a code restructuring that combines loop tiling (specified by the user), advanced communication coalescing and data reuse, pipelining of communicating processes in a double-buffer fashion, buffer size optimization, and optimized loop linearization. It has been fully implemented as a prototype, and the first experimental results show that the method is effective and gives promising results compared to handmade design. To our knowledge, this is the first time, in the context of HLS, that such accelerators are automatically generated.

So far, we restricted our implementation to static control programs for which the sets Load(T) and Store(T) (communications for a tile T) can be built by computing the first reads and last writes of each array location. However, the specification we gave for performing communication coalescing and data reuse in a tile strip is much more general. It also covers the case where the sets of In(T) and Out(T) (data read and written in a tile T) are approximated. Building the sets Load(T) and Store(T) through the computation of the sets In(T) and Out(T), or approximations, may lead to faster algorithms and to sets less complex to scan. Analysis techniques such as those developed in PIPS [22] may be useful. This has still to be explored. We also believe that our general formulation to optimize transfers of remote data, while exploiting data reuse, is of interest for other types of architectures where a kernel is deported on an accelerator with a smaller local memory, as for GPGPUs [16], [17].

References

[1] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[2] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark: A high-level synthesis framework for applying parallelizing compiler transformations," *International Conference on VLSI Design*, pp. 461–466, 2003.

[3] H. Devos, K. Beyls, M. Christiaens, J. Van Campenhout, E. D'Hollander, and D. Stroobandt, "Finding and applying loop transformations for generating optimized FPGA implementations," in *Transactions on High-Performance Embedded Architectures and Compilers I*, ser. Lecture Notes in Computer Science, Springer, 2007, vol. 4050, pp. 159–178.

[4] J. M. P. Cardoso and P. C. Diniz, *Compilation Techniques for Reconfigurable Architectures*. Springer, 2009.

[5] "Altera C2H: Nios II C-to-hardware acceleration compiler," http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html.

[6] J. Xue, *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *ACM International Conference on Programming Languages Design and Implementation (PLDI'08)*, Tucson, Arizona, Jun. 2008, pp. 101–113.

[8] "CLooG code generator in the polyhedral model," http://www.cloog.org/.

[9] P. Boulet and P. Feautrier, "Scanning polyhedra without Do-loops," in *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, 1998, pp. 4–9.

[10] D. Chavarría-Miranda and J. Mellor-Crummey, "Effective communication coalescing for data-parallel applications," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. Chicago, IL, USA: ACM, 2005, pp. 14–25.

[11] W.-Y. Chen, C. Iancu, and K. Yelick, "Communication optimizations for fine-grained UPC applications," in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. IEEE Computer, 2005, pp. 267–278.

[12] B. Creusillet, "Analyses de régions de tableaux et applications," Ph.D. dissertation, École des mines de Paris, Dec. 1996.

[13] P. Feautrier, *The Data Parallel Programming Model*, ser. LNCS Tutorial. Springer Verlag, 1996, vol. 1132, ch. Automatic Parallelization in the Polytope Model, pp. 79–103.

[14] ——, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, Feb. 1991.

[15] ——, "Parametric integer programming," *RAIRO Recherche Opérationnelle*, vol. 22, no. 3, pp. 243–268, 1988.

[16] A. Größlinger, "Precise management of scratchpad memories for localizing array accesses in scientific codes," in *International Conference on Compiler Construction (CC'09)*, ser. Lecture Notes in Computer Science, O. de Moor and M. Schwartzbach, Eds., vol. 5501. Springer-Verlag, 2009, pp. 236–250.

[17] S. Baghdadi, A. Größlinger, and A. Cohen, "Putting automatic polyhedral compilation for GPGPU to work," Jul. 2010, presentation at CPC'10, International Workshop on Compilers for Parallel Computers.

[18] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1242–1257, Oct. 2005.

[19] Y. Bouchebaba and F. Coelho, "Tiling and memory reuse for sequences of nested loops," in *8th International Euro-Par Conference (Euro-Par'02)*, ser. Lecture Notes in Computer Science, B. Monien and R. Feldmann, Eds., vol. 2400. Springer-Verlag, 2002, pp. 255–264.

[20] C. Alias, F. Baray, and A. Darte, "Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose," in *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, San Diego, USA, Jun. 2007.

[21] "Polylib – A library of polyhedral functions," http://www.irisa.fr/polylib.

[22] "PIPS: Automatic parallelizer and code transformation framework," http://pips4u.org/.

APPENDIX

As the review process is double blind, we cannot refer to the PhD manuscript where all details of this work are provided. For completeness of the review process, we give here, in an appendix, the proofs of the theorems of Section II. If the paper is accepted, we will refer to the PhD thesis in the final version.

Theorem 1. *The functions* $T \mapsto \text{Load}(T)$ *and* $T \mapsto \text{Store}(T)$
- $\text{Load}(T) = \text{In}(T) \setminus \{\text{In}(t < T) \cup \text{Out}(t < T)\}$
- $\text{Store}(T) = \text{Out}(T) \setminus \text{Out}(t > T)$

define load and store operators that are valid and exact.

Proof: First, consider the Load operator. By definition, $\text{Load}(t \leq T) = \cup_{t \leq T} \{(\text{In}(t) \setminus \text{In}(t' < t)) \setminus \text{Out}(t' < t)\}$ is a subset of $\cup_{t \leq T} \{\text{In}(t) \setminus \text{Out}(t' < t)\}$. Conversely, let x be in this latter union and let t_0 be the smallest tile index such that $x \in \text{In}(t_0) \setminus \text{Out}(t < t_0)$. By definition, for all $t < t_0$, $x \notin \text{Out}(t)$, thus $x \notin \text{In}(t)$, by minimality of t_0. Thus x belongs to $\text{Load}(t_0)$ and, finally, to $\text{Load}(t \leq T)$. This proves that $\text{Load}(t \leq T) = \cup_{t \leq T} \{\text{In}(t) \setminus \text{Out}(t' < t)\}$, which is Condition (i) of Definition 3 (exact Load). As for Condition (ii) of Definition 3, it holds from the fact that $\text{Load}(T) \cap \text{Load}(T')$, for $T' < T$, is a subset of $(\text{In}(T) \setminus \text{In}(t < T)) \cap \text{Load}(T')$, thus a subset of $(\text{In}(T) \setminus \text{In}(T')) \cap \text{Load}(T')$ and, finally, of $(\text{In}(T) \setminus \text{Load}(T')) \cap \text{Load}(T')$, which is empty.

Note also that $\text{Out}(t < T) \cap \text{Load}(T) = \text{Out}(t < T) \cap ((\text{In}(T) \setminus \text{In}(t < T)) \setminus \text{Out}(t < T)) = \emptyset$, thus Condition (ii) of Definition 1 (valid Load) is satisfied. As this condition is always satisfied for the first tile $T_{\min}$ and as the Load operator is uniquely defined given $\text{Load}(T_{\min})$, this proves that an exact load operator is always valid. There is no need to add Condition (ii) of Definition 1 in Definition 3.

Now, consider the Store operator: $\text{Store}(T) \cap \text{Store}(T') = \emptyset$ for $T' < T$ and $\text{Store}(t \leq T_{\max}) = \text{Out}(t \leq T_{\max})$. This proves Condition (ii) of Definition 4 (exact Store) and Condition (i) of Definition 2 (valid Store). Condition (ii) of Definition 2 is also satisfied as, by definition of the store operator, $\text{Out}(t > T)$ is removed from $\text{Store}(T)$. ■

Theorem 2 (Valid approx. Load)**.** $t \mapsto \text{Load}(t)$ *is valid if the following holds, for any tile* T*:*

(i) $\cup_{t \leq T} \left\{\overline{\text{In}}(t) \setminus \underline{\text{Out}}(t' < t)\right\} \subseteq \text{Load}(t \leq T)$.
(ii) $\overline{\text{Out}}(t < T) \cap \text{Load}(T) = \emptyset$.

Proof: It is sufficient to prove that these conditions imply the correctness conditions given by Definition 1. Let us first check that Condition (i) of Definition 1 holds. This is clear because $\text{In}(T) \setminus \text{Out}(t' < t)$ is included in $\overline{\text{In}}(t) \setminus \underline{\text{Out}}(t' < t)$, thus in $\text{Load}(t \leq T)$. Similarly, since, for each tile t, $\text{Out}(t) \subseteq \overline{\text{Out}}(t)$, Condition (ii) of Definition 1 is verified. ■

Theorem 3 (Valid approx. Store)**.** *The function* $t \mapsto \text{Store}(t)$ *is valid if the following holds (with* $T_{\max}$ *last tile of the strip):*

(i) $\overline{\text{Out}}(t \leq T_{\max}) \subseteq \text{Store}(t \leq T_{\max})$.
(ii) $\text{Store}(T) \cap \overline{\text{Out}}(t > T) = \emptyset$ *for any tile* T.
(iii) $\text{Store}(T) \subseteq \overline{\text{In}}(t \leq T) \cup \underline{\text{Out}}(t \leq T)$ *for any tile* T*, when used with a valid approximated* Load*.*

Proof: The first two conditions show that $\text{Store}(T)$ is an over-approximation that verifies Conditions (i) and (ii) of Definition 2 (valid Store), because of the approximation $\text{Out}(t) \subseteq \overline{\text{Out}}(t)$ for each tile t. Condition (iii) ensures the correctness of Store, as the previous discussion explained, if it implies $\text{Store}(T) \subseteq \text{Load}(t \leq T) \cup \text{Out}(t \leq T)$ for each tile T. If the function $t \mapsto \text{Load}(t)$ is a valid approximated Load, then $\cup_{t \leq T} \left\{\overline{\text{In}}(t) \setminus \underline{\text{Out}}(t' < t)\right\} \subseteq \text{Load}(t \leq T)$ (Condition (i) of Theorem 2). Furthermore, $\cup_{t \leq T} \left\{\overline{\text{In}}(t) \setminus \underline{\text{Out}}(t' < t)\right\} \cup \text{Out}(t \leq T) = \cup_{t \leq T} \overline{\text{In}}(t) \cup \text{Out}(t \leq T) = \overline{\text{In}}(t \leq T) \cup \text{Out}(t \leq T)$. Thus, if $\text{Store}(T) \subseteq \overline{\text{In}}(t \leq T) \cup \underline{\text{Out}}(t \leq T)$, then $\text{Store}(T) \subseteq \overline{\text{In}}(t \leq T) \cup \text{Out}(t \leq T)$. Thus $\text{Store}(T) \subseteq \text{Load}(t \leq T) \cup \text{Out}(t \leq T)$ and the Store operator is valid. ■

Theorem 4. $\text{Load}(T) = \overline{\text{In}}(T) \setminus \left\{\overline{\text{In}}(t < T) \cup \overline{\text{Out}}(t < T)\right\}$ *and* $\text{Store}(T) = \overline{\text{Out}}(T) \setminus \overline{\text{Out}}(t > T)$ *define valid load and store operators if the following holds for any tile* T*:*

(i) $\overline{\text{Out}}(T) \subseteq \overline{\text{In}}(t \leq T) \cup \overline{\text{Out}}(t > T) \cup \underline{\text{Out}}(t \leq T)$.
(ii) $\overline{\text{In}}(T) \cap \left\{\overline{\text{Out}}(t < T) \setminus \underline{\text{Out}}(t < T)\right\} \subseteq \overline{\text{In}}(t < T)$.

Proof: We already proved the validity of Store provided Constraint (i), which can be easily interpreted: it means that if a data appears to be defined in a tile T, then either it can be proved to be defined in T or earlier (set $\underline{\text{Out}}(t \leq T)$), or it will appear to be defined again later (and will be stored later, set $\overline{\text{Out}}(t > T)$), or it has been accessed in T or earlier (thus loaded or defined earlier, if Load is valid). For the validity of Load, Condition (ii) of Thm. 2 is satisfied as $\overline{\text{Out}}(t < T)$ is removed. It remains to consider Condition (i) of Thm. 2.

The proof is similar to the proof of Theorem 1. First, it is immediate to show that the set $\text{Load}(t \leq T)$ is a subset of $\cup_{t \leq T} \left\{\overline{\text{In}}(t) \setminus \underline{\text{Out}}(t' < t)\right\}$ since $\underline{\text{Out}}(t' < t) \subseteq \overline{\text{Out}}(t' < t)$. Now, let x be in this latter union and let t_0 be the smallest tile index such that $x \in \overline{\text{In}}(t_0) \setminus \underline{\text{Out}}(t < t_0)$. By construction, for all $t < t_0$, $x \notin \underline{\text{Out}}(t)$. This implies $x \notin \overline{\text{In}}(t)$ for all $t < t_0$, otherwise this would contradict the minimality of t_0. Thus x belongs to $\left\{\overline{\text{In}}(t_0) \setminus \underline{\text{Out}}(t < t_0)\right\} \setminus \overline{\text{In}}(t < t_0)$. Now, if $x \notin \text{Load}(t_0)$ then, by definition of $\text{Load}(t_0)$, x must belong to $\overline{\text{Out}}(t < t_0) \setminus \underline{\text{Out}}(t < t_0)$. We now use the last condition of the theorem: $x \in \overline{\text{In}}(t < t_0)$, a contradiction. Thus $x \in \text{Load}(t_0)$ and Condition (i) of Thm. 2 is satisfied with an equality. ■

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399