EINDHOVEN UNIVERSITY OF TECHNOLOGY

ASAM : Automatic Architecture Synthesis and Application Mapping; dl. 2.2: Report on initial version of the hierarchical application model

Citation for published version (APA):
Micconi, L., Madsen, J., Pop, P., Kienhuis, B., Corvino, R., & Jozwiak, L. (2011). *ASAM : Automatic Architecture Synthesis and Application Mapping; dl. 2.2: Report on initial version of the hierarchical application model*. (ARTEMIS; Vol. 2009-1-ASAM-100265-D2.2). ASAM.

Document status and date:
Published: 01/01/2011

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 05. Oct. 2023

ASAM

Grant agreement no. 100265

Artemis Project

ASAM

Automatic Architecture Synthesis and Application Mapping

D2.2: Report on initial version of the hierarchical application model

Due Date of Deliverable — **1st May, 2011**

Completion Date of Deliverable — **1st May, 2011**

Start Date of Project — **1st May, 2010 – Duration 36 Months**

Lead partner for Deliverable — **DTU**

Author(s): — **L. Mecconi, J. Madsen, P. Pop (DTU), B. Keinhuis (COMPAAN), R. Corvino, L. Jozwiak (TUE)**

Revision: v1.0

Project co-funded by the Artemis Joint Undertaking Call 2009		
Dissemination Level		
PU	Public	x
PP	Restricted to other program participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Table of Contents

1 Introduction 3
2 Scope 4
3 Parallelization from C-code 5
3.1 Type of parallelism 5
3.2 The Compaan Compiler 6
4 Proposed Application Model 10
4.1 The Task Graph Model 10
4.2 The Proposed Application Model 11
4.3 Possible Extensions to the Application Model 13
5 Relations to KPN and micro-level DSE 14
5.1 Relation to KPN 14
5.2 Relation to micro-level DSE 15
6 Evaluation of the Proposed Application Model 19
7 References 21

1 Introduction

The aim of the ASAM (Automatic Architecture Synthesis and Application Mapping) project is to develop a general methodology that can be efficiently and effectively used in heterogeneous multiprocessor ASIP based embedded systems. This report is a deliverable of WP2 (Figure 1) and presents the proposal for a generic application model to be used in the ASAM project. The document is based on deliverable D1.1 and D1.2, which specifies the design methodology and flow, and D2.1, which specifies the generic platform model. In order to be able to map an application onto the generic platform model, the application needs to be captured by an abstract representation. The application model should capture the details needed by the micro-architecture level, i.e. a control-data flow model at the operator/instruction level, as well as the high-level abstraction needed at the macro-architecture level, i.e. a task graph model.

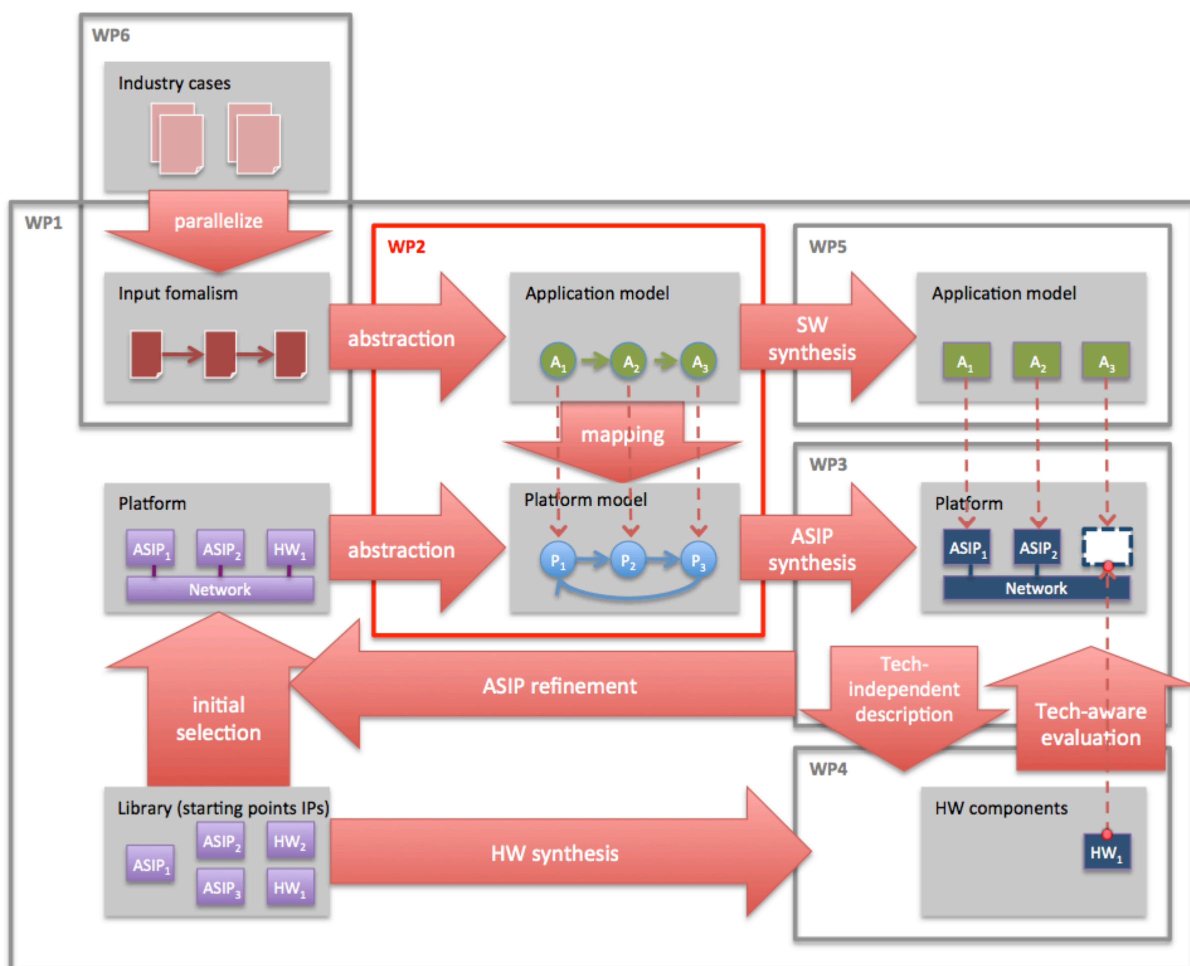


Figure 1 - ASAM work packages overview with WP2 highlighted

2 Scope

This document is intended to descripe the initial proposal for an application model for ASAM, with particular focus on the macro-architecture level. Being an initial model means that the proposed model is likely to be modified and extended during the next period of the ASAM project.

The document describes the requirements for the application model and presents the initial proposal for a hierarchical model, which will serve macro-architecture level synthesis. The macro-level model is based on a process network model (also referred to as a task graph), which is extended to reflect the elements of the generic platform model for easier mapping and which is suited for expressing high-level parallelism. The macro-level model is linked with the Kahn process network model, which is the output of the Compaan compiler tools. The micro-level model is based on a control-data flow (CDFG) model, which allows for identification of low-level parallelism, and to form clusters that may be transformed into new application specific instructions of the ASIP. The document describes the relation between the micro- and macro-level DSE.

The document evaluates the proposed application model by showing how it can be used for macro-level mapping onto a platform represented by the PCU model presented in deliverable D2.1.

3 Parallelization from C-code

This section presents the first step in the ASAM design flow, where the initial C-code is parallelized by the use of the Compaan compiler. This parallelization of the initial application is to be captured by the application model for the macro-level DSE.

3.1 Type of parallelism

The different types of parallelism to be expoited in the ASAM design flow, was presented in deliverable D1.1 [4]:

- **Operation Level Parallelism (OLP):** corresponds to application-domain-specific custom operations, that combine multiple RISC-equivalent operations (e.g. add, subtract, multiply, load, store, jump) into a single operation.
- **Data level Parallelism (DLP)**: present in algorithms that operate on large amounts of data, often repeating the same operation on different data elements and that can often be rewritten to operate on vectors. It allow to use of SIMD (Single Instruction Multiple Data) or vector operations that operate on operands organized as vectors of multiple data. This increases performance and also the efficiency of control resources (instruction fetch and decode is used more efficiently as only a single operation needs to be fetched and decoded). Moreover the code size is decreased and the number of register and memory accesses can be reduced by providing vector-wide data access.
- **Instruction-Level Parallelism (ILP)**: it consists in issuing multiple operations per instruction. To do this at low hardware cost a Very Long Instruction Word (VLIW) architecture can be used. The compiler identifies the amount of ILP that can be exploited.
- **Task Level Parallelism (TLP)**: Partitioning of an application over multiple processor cores working in parallel to boost performance. A heterogeneous multiprocessor solution (different types of processors each tuned to focus on a specific set of tasks), is often preferred to achieve the highest efficiency.

The task level parallelismis of particular importance at the macro-level, where we need to capture the high level characteristics of an application that can allow us to perform an efficient DSE of mapping and scheduling. The target platform model (PCU Model defined in deliverable D2.1) is a high level representation of the platform and offers a view of the processing units and interconnections that defines the HW architecture platform. In this context, i.e., the macro-level, we are only interested in the task parallelism that will allow the parallel execution of multiple tasks on different processors, taking into account the data dependencies among the tasks.

In order to axploite the other types of parallelism, additional information is needed. This information is only available at micro-level: the micro-level DSE will focus on the parallelism contained within each task (DLP, OLP and ILP) and how to use it to customize and design each ASIP processor.

According to these considerations and the need at macro-level to identify task that can be mapped on different processors and the communication (data dependencies) between them, we decided to

use a Task Graph model to describe an application.

3.2 The Compaan Compiler

The Compaan compiler is able to convert a C-code piece into a Kahn Process Network (KPN). To explain what this KPN looks like, it is best to follow an example in which we convert a simple C-code example into a KPN. The example code is:

```
#define MAX_I 100
#pragma compaan_procedure parallelization_example
void parallelization_example(int data_in[MAX_I], int data_out[MAX_I]) {

        int i;
        int j;

        const int P = 4; //parallelization factor
        const int MAX_IP = MAX_I / P;

        int a[MAX_IP][P];
        int b[MAX_IP][P];

        for (i = 0; i < MAX_IP; i = i + 1) {
        for (j = 0; j < P; j = j + 1) {
                a[i][j] = data_in[P * i + j];
                }
        }

        for (i = 0; i < MAX_IP; i = i + 1) {
               b[i][0] = Transform(a[i][0]);
               b[i][1] = Transform(a[i][1]);
               b[i][2] = Transform(a[i][2]);
               b[i][3] = Transform(a[i][3]);
        }

        for (i = 0; i < MAX_IP; i = i + 1) {
                                for (j = 0; j < P; j = j + 1) {
                                      data_out[P * i + j] = b[i][j];
                                }
                        }
}
```

The code starts by defining the symbol `MAX_I` which indicates how many tokens an array contains. Then the network function **`parallelization_example`** is created which provides the external data arrays `data_in[MAX_I]` and `data_out[MAX_I]`. After initializing some variables, the variable `P=4` is defined. This variable is used to define the 'parallelization' factor. The parallelization facto defines how the `MAX_I` tokens are processed over P streams. The number of tokens per stream are defined by variable `MAX_IP` and is used to define the arrays `a[]` and `b[]`.

Using the linearization function `P*i+j`, data from array `data_in[]` is put in four different arrays `a[][]`. Then the function 'transform' is executed in parallel on the P streams. Since `P=4`, the transform function is unrolled 4 times. Array `a[i][0]` is the first stream, `a[i][1]` the second stream etc. The result of applying the function transform is stored in array `b[][]` for each stream. Finally, the four different versions of b[][] are merged back into the single 1-D array `data_out[]` that is streamed out. Again a linearization technique is used to moved a 2-D array into a 1-D using

the function `P*i+j`.

The KPN graph we obtain for example parallelism is given in Figure 2.

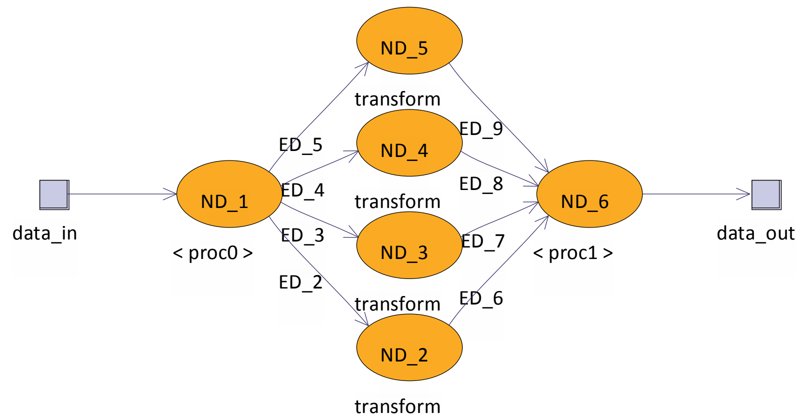


Figure 2 - KPN Graph for Data Parallelism

The next step is to have a look inside some of the nodes of the KPN. The most interesting nodes are ND_1 and ND_6 as they distribute and merge data stream. Since the structure of both is very similar, we only look at ND_1. We also look at node ND_5, which is an example of the simplest form. In Tabel 1, we show the processes in a Matlab like pseudo language.

Tabel 1 Code Example Process ND_1 and ND_5

Code ND_1

```
for i=0 : 1 : i : 2,
  for j=0 : 1 : j : 3,

    [ out_0 ] =
        compaan_outlinedproc0(
            data_in[4*i+j] );

    if j == 0,
      % link ED_2
      ed_2.write( out_0  );
    end

    if j-1 == 0,
      % link ED_3
      ed_3.write(  out_0 );
    end

    if j-2 == 0,
      % link ED_4
      ed_4.write(  out_0 );
    end
```

Code ND_5

```
for i=0 : 1 : i : 2,

  % link ED_5
  [ in_0 ]  = ed_5.read();

  [ out_0 ] = transform( in_0 );

  % link ED_9
  ed_9.write(  out_0  );

end // for i
```

```
    if j-3 == 0,
      % link ED_5
      ed_5.write(  out_0 );
    end

  end // for i
end // for j
```

In case of ND_1, two for-loops are defined with iterators i and j. Based on the value of iterators j, data that is created by function ‘compaan_outlinedproc0’ is distributed over 4 different edges. When j==3, data is sent to edge ED_5 that is connected to process ND_5. In the column for ND_5, is shown that for each value of for-loop i data is read from edge_5 using the function `get()`. A write to a edge is blocking. This means that when there is no space to write data, the process will completely stall until room becomes available. A read from an edge is also blocking. This means that when there is no data to read, the process will completely stall until data becomes available.

The function name `compaan_outlinedproc0` appears as the original statement is an assignment. The line “`a[i][j] = data_in[P * i + j]`”, cannot be interpreted directly by Compaan, as Compaan expects a function. Inside the Compaan compiler this statement is therefore converted automatically into the function `compaan_outlinedproc0`.

The ‘parallel’ example is quite simple. A more complex case is shown in Figure 3. It describe node ND_3 from a KPN describing QR decomposition. This example showns what a process in a KPN generated by Compaan look like. A process always starts with for-loops spanning the so-called node domain of a process. Then three phases can be identified; read, execute and write. In the read-phase (the green box), data is read from edges depending on whether the if-statement is taken. Each if-statement in the process is always expressed as an affine expression of the loop iterators. In the execute-phase (the read box), the input values in_0 and in_1 are passed on to the function ‘Vectorize’ that executes and produces the arguments out_0, out_1 and out_2. In the write-phase (the purple box), these arguments are distributed over the edges that connect to the port.

The right-hand side of Figure 3 shows again the ‘Vectorize’ function as it was defined in the original C-code. The function has two input arguments and 3 output results. Each input argument has in this case two ports to read data from. The output results either have one, one or multiple ports to write to. These ports connect to the edges in a KPN. An input or output argument can have multiple ports due to the dataflow analysis done by Compaan. In [1] is explained how this process takes place.

The kind of dataflow model Compaan captures is close to Cyclo Static Dataflow (CSDF). Intuitively, the for-loops in a program lead to the definition of cycles. Looking again at the right-hand side of Figure 3, the function Vectorize is fired repeatedly and on each firing a single token is read for each input argument and data is produced for each output value. However, from where this token is read or written depends on how long a cycle is. In a KPN generated by Compaan, these cycles are encoded in the static affine linear expression use in the if-statements in Figure 3.

An alternative view can be given for the process view in Figure 3. This view is called ‘Stream-based Functions’ and is described in [2]. This model is closer to the way dataflow models are expressed.

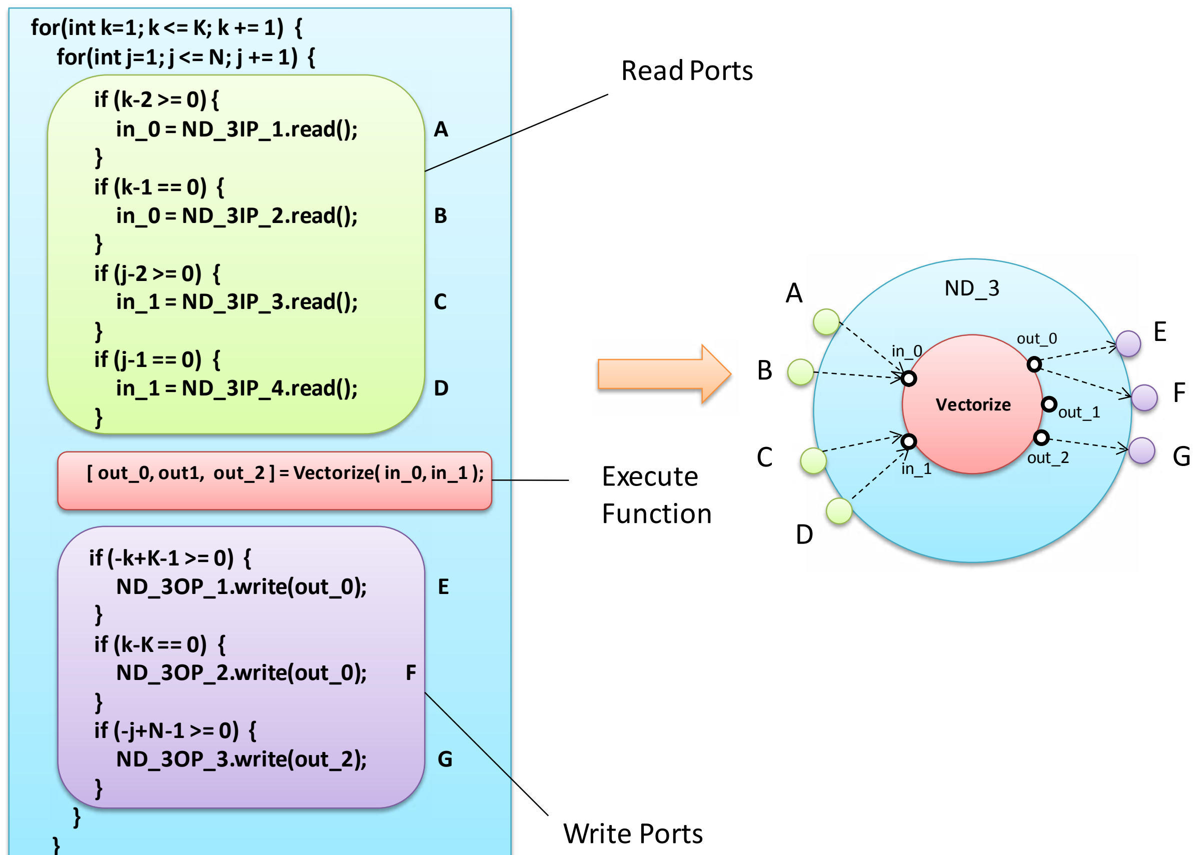


Figure 3 - Switching inside a KPN node

4 Proposed Application Model

The Kahn process network (KPN) [KPN] used by Compaan, is a special case of a more general class of process networks. KPNs are communicating through buffers (FIFO queues) using non-blocking write and blocking read operations. Only a single processing is allowed to read from a specific queue, which means that if data has to be sent to multiple processes, data has to be doublicated inside the process. In general, KPNs are difficult to statically schedule, as it is difficult to predict their precise behaviour over time.

We propose to use the more general process network model, also referred to as the task graph model, which is particularly suited for solving the task scheduling problem; in the following section we will present the task graph model and describe how it will fit into the macro-level mapping and scheduling DSE.

4.1 The Task Graph Model

Before provide a more formal description of the Task Graph for application modeling, we introduce a general definition of representing parallel computation as a Graph Model [3]:

> **Definition 1 - Graph Model as program model** – *A Graph Model can be used to represent the computation and the communication inside a program. The vertices model the computation, while the edges model the communication. Each vertex identifies a* node *to which a task is associated. The task represents the computation inside the node and contains sequential computation at different levels of granularity (atomic instruction/operation, a thread, a basic block or a sequence of these). A node can perform either communication or computation, but just one activity at a time.*

Starting from this general description, we can define the representation of a graph as a Task Graph as follows [3]:

> **Definition 2 - Task Graph** - *A Task Graph is a directed acyclic graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ that can be used as program model.* **V** *is a set of nodes that represent the tasks of the program, while* **E** *is the set of edges that model the communications between the tasks. A positive weight w, representing computation cost, is associated with each node $n \in \mathbf{V}$. An edge $e_{ij} \in \mathbf{E}$ from node n_i to n_j, $n_i, n_j \in \mathbf{V}$, represents the communication from node n_i to node n_j. A nonnegative weight c, representing the communication cost, is associated with each.*

Hence, a task graph is an acyclic graph in which every vertex represents a task that is part of the application. The edges between vertices show the dependences between the different tasks and impose a partial order of nodes. This partial order should be taken into account in order to achieve an efficient and effective mapping and scheduling. In particular, the edges characterize the data dependences: a task can start its execution only when the data provided by its input edges is available.

From a formal point of view, each task vertex represents a unit of work that can be performed [5], it means that different levels of granularity can be depicted (single instruction, set of instructions,

etc.). In our case, each vertex represents a piece of sequential code that composes our application. The different tasks can instead be executed in parallel (obviously according to the data dependencies).

As previously mentioned a Task Graph is able to capture the data dependencies among the tasks, but there is no information about the control dependencies, in fact it cannot cover the branching concept (e.g. if…else statements). Moreover it is not suitable for representing iterations and loops: cycles are not allowed and a loop can only be described by expressing every single iteration. Ultimately, this means that if the iteration number is generated at execution time, the task graph cannot be used. However, in this initial application model, we will assume that these limitations have no influence, as we only need to represent coarse grained, non-iterative computations. In Figure 4 an example of a task graph is shown.

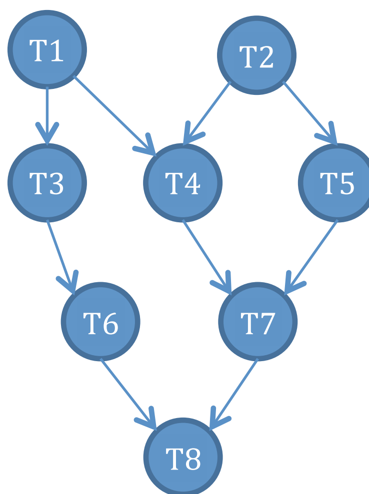


Figure 4 - Task Graph Example

Eventough the computation and communication costs are part of the definition of a task graph. The exact values are not known until the task graph has been mapped onto a platform. I.e., at this level of abstraction there is no notion of time, resources or other quantitative performance estimates. In order to obtain these, the application model must be mapped to the platform model and the task scheduling must be defined [4].

The information that is available for each task before mapping is the amount of data that a task can generate at its output edges. This type of information can influence the interconnections and the mapping decisions. For example, if two tasks has a very high communication cost, indicating that a large amount of data has to be exchanged between the two, it could indicate that a good solution would be to map both tasks on the same processing unit, and hence, reducing the latencies due to the transmission through the interconnection network.

4.2 The Proposed Application Model

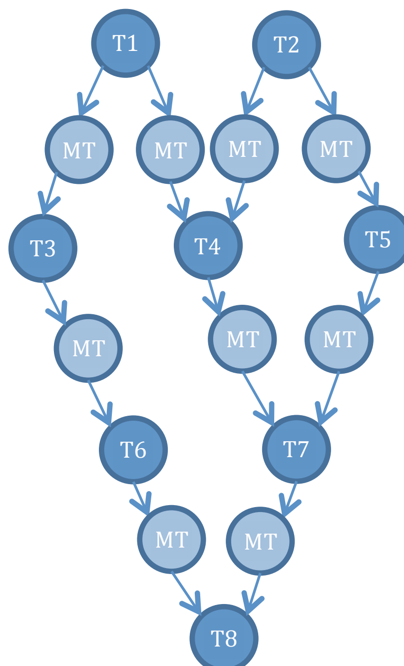
As already described, an important reason for choosing a task graph for the application modelling, is its compatibility with the KPN model generated by the Compaan compiler. The Compaan compiler

provides the translation from C code to a model that is able to capture different degrees of parallelism present in the application. The Compaan compiler can generate an large number of different KPNs with the same input/output behaviour, each prepresenting different degrees of parallelism.

The main difference between the two models is the way the communication between different tasks is interpreted. In the case of KPN the exchange of data (token) is done using FIFOs (First In First Out buffers) [4]. In the tasks graph model the communication is done through the exchange of service requests, making communication explicit and implementation independent. In section 5 we will elaborate on how a KPN model can be transformed into a task graph.

Besides being a suitable model for the behavior of an existing program, the task graph model can also be used to capture the behaviour of an application at an even higher level of abstraction, i.e., at the level where the application is being specified. A designer can use a functional block representation of his/her application and the task graph can be used to capture the application parallelism and the dependencies between functional blocks.

In order to build an application model that can support the scheduling and mapping DSE, the task graph is extended with another type of node: the message task (MT), which is similar to the model used in the DOL framework from ETHZ [7]. Figure 5 shows an example of an extended version of the task graph from Figure 4. The message task is used to model the message passing between two tasks, i.e., communication can only be done through message tasks. The motivation for introducing this type of node is to ease the mapping process of the application on the PCU model representing the HW platform. When two communicating tasks are mapped to different processing units of the platform, the data to be transferred, i.e. the message task, has to be mapped to the communication infrastructure of the platform. This may be a mapping to a single bus, a shared memory unit, or a



more

Figure 5 - Example of Task Graph with Message Task

elaborate path including multiple busses/links and switches. The separation between computation and communication allows us to focus on the cost of the communication between tasks mapped on different processing units. On the other hand, once a mapping is established, the message tasks between tasks mapped on the same processor could be removed, considering the communication time to be zero (at least as a first approximation).

4.3 Possible Extensions to the Application Model

One of the main challenges in the application modeling is not only to capture the task level parallelism and the data dependencies among them, but also to identify information that could help the mapping and scheduling DSE:

- The design should be able to specify application requirements that must be fulfilled by the platform. This could be access to specific resources required by a particular task. Such information could help the mapping process by identifying the best available ASIP processors for a given task.
- The application model should be enhanced such that estimation about the type of parallelism present in each single task can be inferred from the model. This would allow a more accurate mapping at the macro-level, in particular in the first stages, where the actual ASIP architecture is not yet known, but where the macro-level DSE has to explore the consequences of possible ASIP architectures. This could also act as a starting point for the micro-level DSE.
- Another possible extension of the application model is the introduction of nodes that are able to handle control dependencies. This extension can also be very useful at the micro-level DSE.

5 Relations to KPN and micro-level DSE

In the ASAM design flow, the application model has to be extracted from the KPN produced by the Compaan compiler. Besides being used for macro-level DSE, the application model has to work as an interface to the micro-level DSE. This section discusses the relation of the proposed application model to both the KPN and the CDFG (Control Data Flow Graph) of the micro-level DSE.

5.1 Relation to KPN

The Compaan compiler accepts the application C-code as its input and allows the designer to express the application behavior in the format of KPN representation. Compaan Technology translates the C-code into its corresponding KPN representation using mathematical techniques based on the polytope model. Compaan Design tools focus on the task-level parallelism that can further be sub-divided into Fork/Join type parallelism and pipeline parallelism - as shown in Figure 6. Compaan compiler can generate an indefinite number of different KPNs with the same input-output behavior. Each KPN will have different characteristics i.e. numbers of channels and processes. One can view this as expressing the C code (the functional description) in different degrees of parallelism – as shown in Figure 7. Since Compaan HotSpot Parallelizer realizes KPN implementation automatically from C code, it can be used to implement the partitioning decisions that are represented in an application model at the macro-level. The application model at the macro-level can use an interface with Compaan tool in order to transfer the partitioning decisions. Then Compaan tool can generate the KPN considering the decision provided by the macro-level. The partitioning, mapping and scheduling of the application specification computed with respect to the architectural constraints are required during the simulation and emulation.

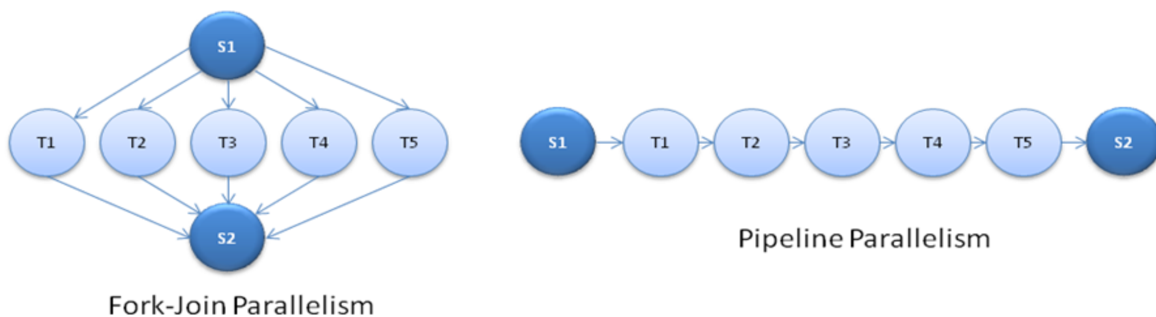


Figure 6. Fork/Join type parallelism and pipeline parallelism represented with Compaan tool

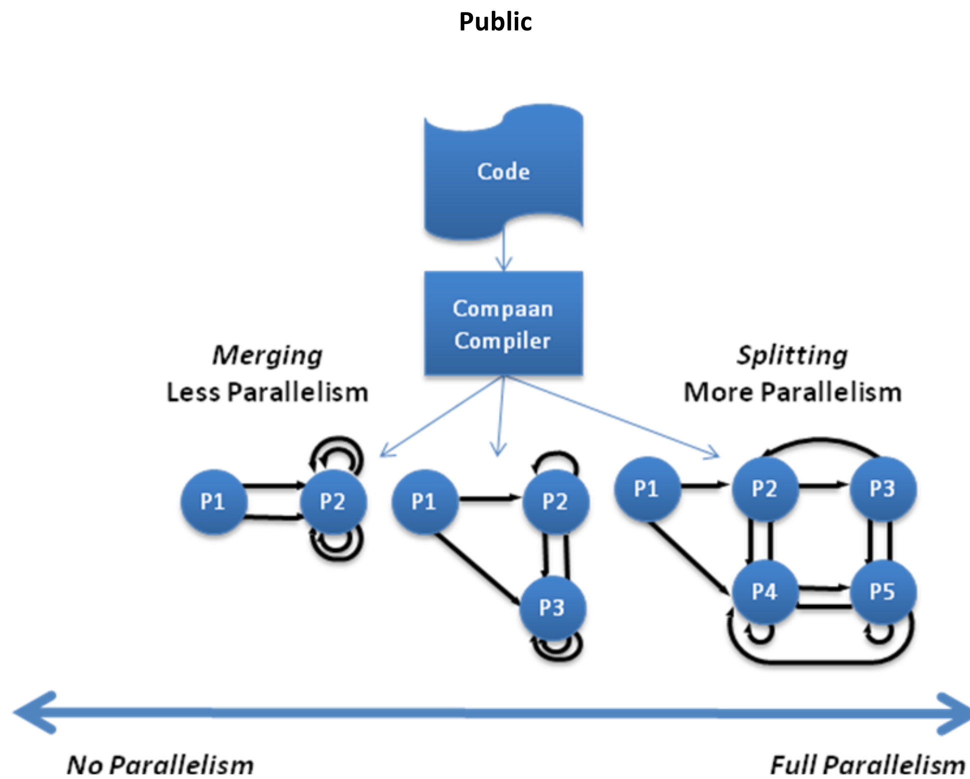


Figure 7. Different degrees of parallelism extracted with Compaan tool

As the Compaan compiler can provide different levels of parallelism from the same C code, an important decision is to identify which degree of parallelism is be suited for the macro-level DSE. One possibility that should be investigated is to assume the highest degree of parallelism (the KPN with the highest number of nodes) and then cluster nodes of the KPN into nodes of the task graph. This decision can be obtained from the mapping and being part of the design space exploration or can be a decision taken statically according to the platform and the type of applications. This approach could also be integrated with the micro-level DSE, i.e., the micro- and macro-level optimizations can collaborate in order to establish the best level of parallelism that can be exploited by each of them.

In a KPN the minimum unit of communication between different nodes is a token. The number of tokens in the Compaan KPN can be different (depending on the parallelism). Moreover, they can have different sizes. One possible way to represent the tokens inside the task graph is to associate one or more of them to the message task. An additional consideration for future investigation is if it is possible to use the token/message task as a reference for the parallelism level, i.e., once a minimum size of the token/message task is chosen, the parallelism of the application could be based on this.

5.2 Relation to micro-level DSE

It is important to consider how the application model can be extended from the macro-level in order to provide useful information to the micro-level DSE. At micro-level the model is required to able to capture conditional and loop statements. This could be handled by a nested graph model that represents the characteristic of every single task node at a lower level of detail. At the micro-level it

is necessary to work on a finer level of granularity, such as operation and instruction level. Instruction level parallelism is particularly well fited the VLIW architecture of Silicon Hive ASIP cores. At the inner level DSE, information such as loop parallelization, conditional statements, identification of recurrent combination of operations for custom instruction, need to be captured.

The micro-architecture design flow of the ASAM project has the following three exploration phases: the first phase, during which the application bottlenecks and application intrinsic parallelism are explored; the second phase, during which the parallelism is actually explored and exploited to construct a coarse parallel ASIP architecture in accordance with the micro-architecture in accordance with the micro-architecture issue decision model; and the phase of micro-architecture refinement, during which optimized application-specific instruction set of the issue slots are selected. After each phase, the micro-level DSE gives a feedback on the relevant parameters to the macro-level DSE. The macro-level DSE has the possibility to ask the micro-level DSE for specific services, i.e. to control which phase and how it has to be executed. For instance, to decide if after the first step the micro-level DSE has to continue or to stop and give feedback to the macro-level DSE.

Since the most external tools that are used, during the first phase, for the application analysis and parallelization accept as its input C code, the whole micro-architectural exploration takes as input a C code of the application part assigned to a single processor. The micro-architecture exploration will also receive as its input from the macro-architecture exploration the kind of the processor itself and the decision model needed by this application part and processor pair. Moreover, internally, the micro-level DSE will use a (hierarchical) control data-flow graph (CDFG) application representation annotated with some additional information on the data access patterns and data dependences. Indeed the different phases of the analysis need specific (additional or partial) information to quickly explore specific kind of parallelism and provide together a framework able to effectively and efficiently analyze and exploit all kinds of parallelism, except for the task-level parallelism.

Summing up, the micro-level DSE has to get from the macro-level synthesis a C code of the application part assigned to a single processor, the decision model (i.e. parametric constraints, objectives and tradeoff information) for the application part and the single processor, and in most cases the type of the single processor. In a specific case of a very initial exploration the processor type may not be specified. Moreover, some control signals precisely specifying the services required by the macro-architecture DSE from the micro-level DSE for the given input are required, indeed the early exploration iterations will certainly differ from the last fine-tuning iterations, and consequently, with the DSE progress different control signals will be needed for a more and more precise exploration. The micro-level DSE will not expect to get as its input any application or implementation model from the macro-level DSE, and will also not provide to the macro-level DSE any model.

The micro-level DSE will receive from the macro-level DSE the task to execute in the input form as briefly sketched above. During the task execution, it will provide to the macro-level DSE only the feedback in the form of the estimated values of some design parameters, together with their probabilities. With the progress of the task execution these estimated values will be of course more and more complete and precise. Only at the end of the task execution (i.e. after completely synthesizing an optimized ASIP processor for the given application part that satisfies all the

constraints and objectives) at will deliver to the macro-level DSE a complete processor description expressed in the Silicon Hive proprietary language TIM. The macro-level DSE will be then able to compose the whole system from all the synthesized optimized processors, memories and communication structures, and simulate or emulate it in the ASIP prototyping environment.

There are no strong relations between the abstract behavior models used at the macro-level DSE and micro-level DSE; especially, because the two DSEs will work on different abstraction/precision level. The only overlap is on the task-level parallelism. So, it is possible that the micro-level DSE will use a model equivalent to KPN as in the macro-level DSE for the task-level parallelism exploration, but it is less probable that the macro-level DSE will use a CDFG based models as in micro-level DSE. Also, the micro-level DSE certainly has to only get just the partial C code, processor type, the decision model, and control signals from the macro-level DSE. From this C code, the micro-level DSE will itself create its own internal CDFG based models.

Summing up, the micro-level exploration will use in parallel a (restructured) C-code of a given application part assigned to a given processor and a hierarchical CDFG annotated with information on data access patterns and data dependences corresponding to (a part of) the C-code. Figure 10 and Figure 8 give the CFG and the DFG of the code of Figure 9, respectively. Figure 11 gives the unrolled DFG of the code of Figure 9 and it shows how large and complex a (C)DFG may become in the case of unrolling. This makes the transfer of information between the exploration phases a complex task.

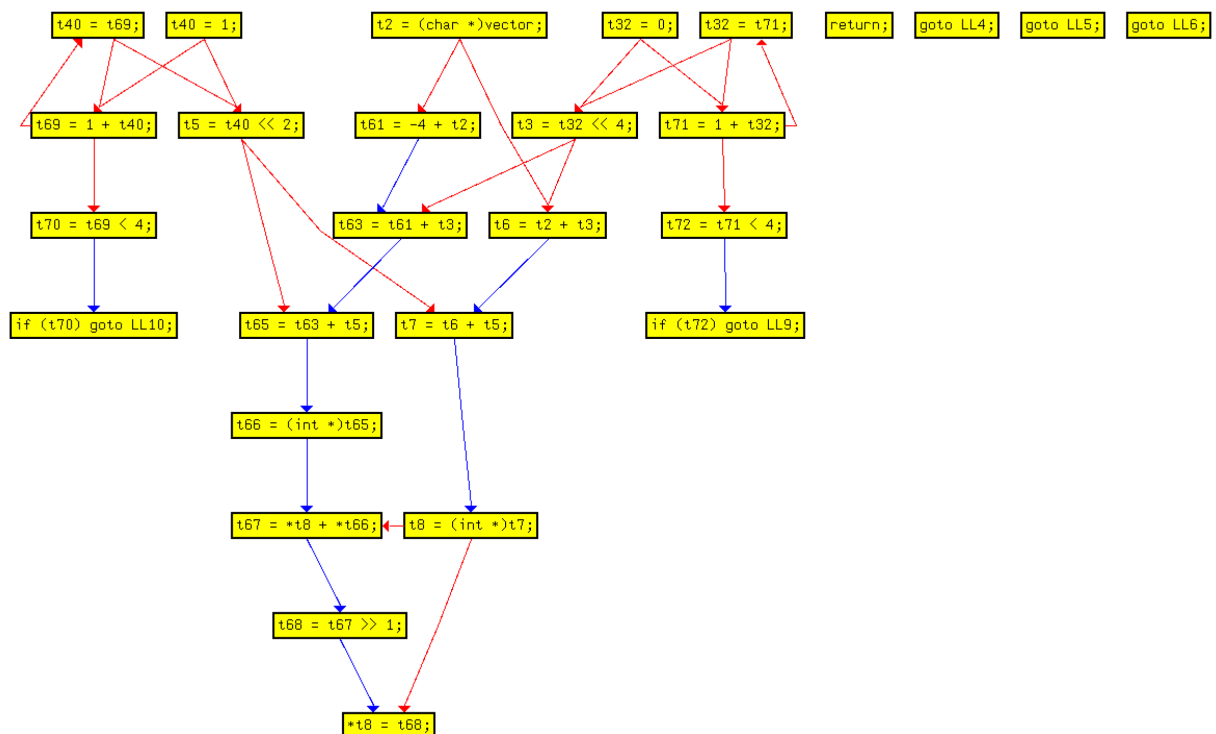


Figure 8. DFG of the code of Figure 9.

The first and second phases of the micro-level DSE will use a more abstract view of the DFG giving more explicitly the data access patterns and dependences in order to perform rapidly and precisely the required loop transformations. In this case, the analysis does not need information on the performed precise computations, as provides by a CDFG, but just information on data accesses and

dependences. However, the data dependences and access patterns are not sufficiently clearly and compactly expressed by the CDFG itself. A possible solution could be the usage of an abstract CDFG-based representation supplemented with an array oriented language [6] that gives partial and pertinent information to explore loop transformations.

```
int vector[4][4];

void fir(void){

    int i, j;

    for(i=0 ; i<4; i++){

        for(j=1; j<4;j++){

            vector[i][j]=(vector[i][j]
                          +vector[i][j-1])>>1;

        }
    }
}
```

Figure 9. C code of a FIR filter.

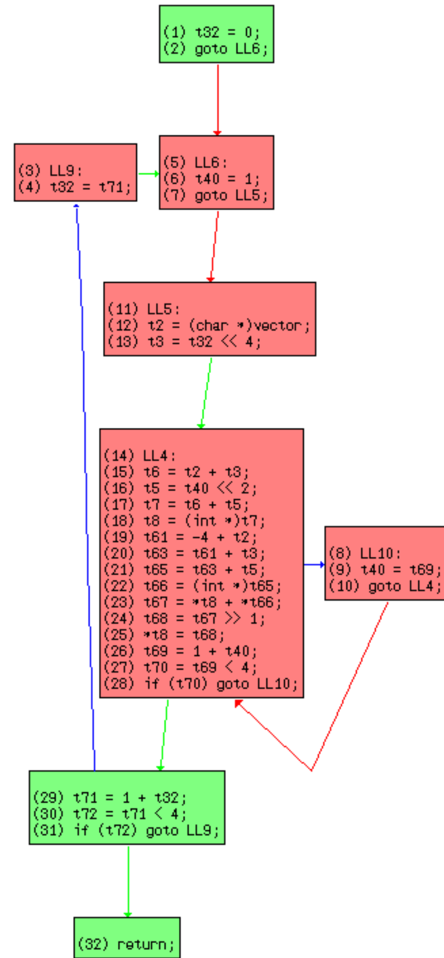


Figure 10. CFG of the code of Figure 9

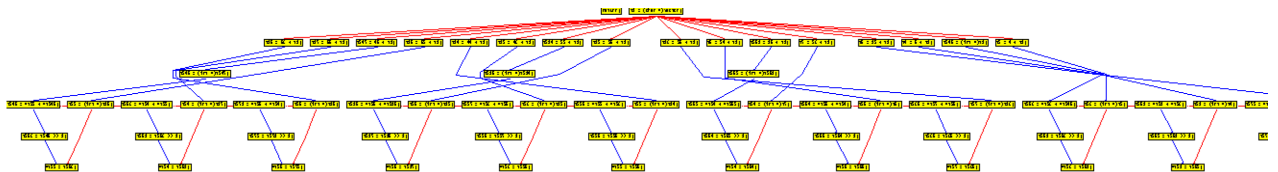


Figure 11. Unrolled DFG of the code of Figure 9. The aim of this figure is to show that the complexity of an unrolled DFG may be very high.

6 Evaluation of the Proposed Application Model

In this section we evaluate the proposed application model by showing how it can be used for macro-level mapping on to a PCU model for the platform. This is the level at which macro-level DSE takes place.

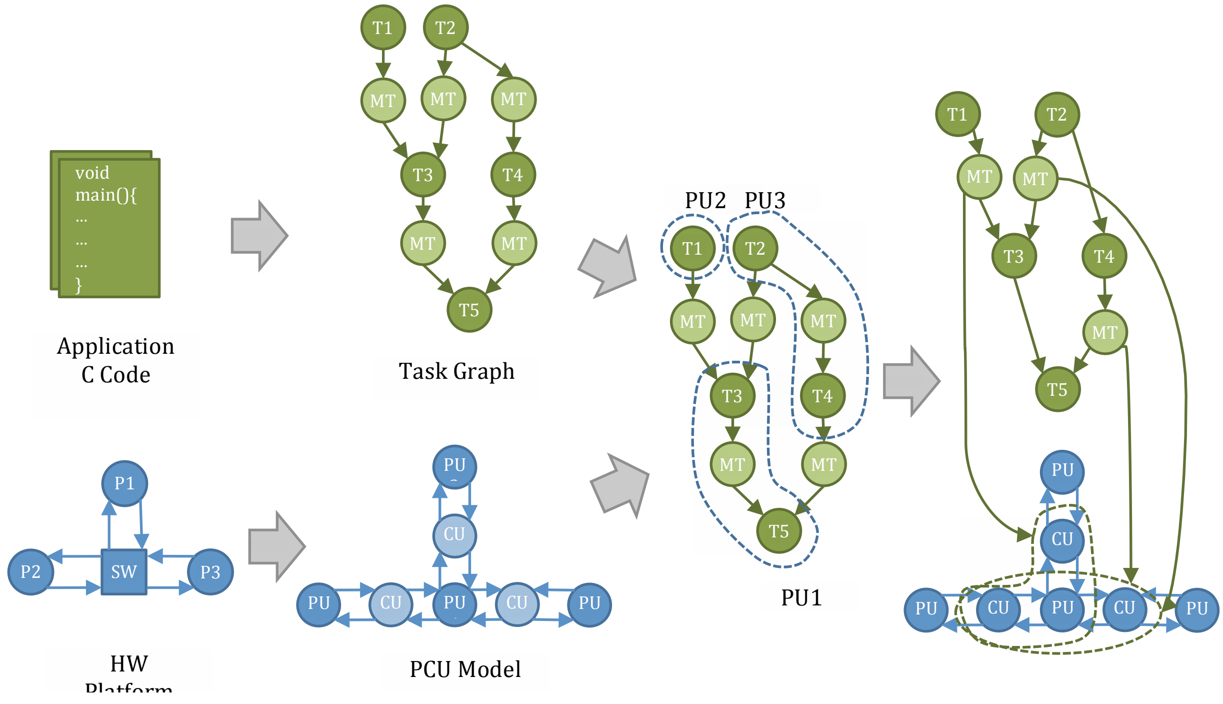


Figure 12 - Example of mapping with Task Graph and PCU model

Figure 12 shows an example of mapping an application onto a platform. The application (defined as C Code) is modelled by a task graph (with message tasks) and the HW platform is modelled as a PCU Model. The tasks are mapped on the processing units and the communication between tasks mapped on different processors, is highlighted through the message tasks (communication between PU1-PU3 and PU2-PU1). In this example, we consider that communication time between tasks mapped to the same processing unit as negligible (the corresponding message tasks are removed).

Figure 13 shows a second example of mapping. In this case we also consider the memories of the platform model. We assume that processor 1 (PU1) has a local memory, meanwhile processor 2 (PU2) uses a global shared memory for the storage of the data that need to be transferred from one task to another. All the data exchanges are performed using these memories: a task writes its output data to a memory making them available as input data for other tasks. In the figure, the flow of data is shown, in this case all message tasks are maintained since read/write to memory is now captured as part of the model. In particular, we can see that even though tasks T2 and T4 are mapped to the same processor (PU2), the transfer of data between them has to take place through the shared memory outside of PU2.

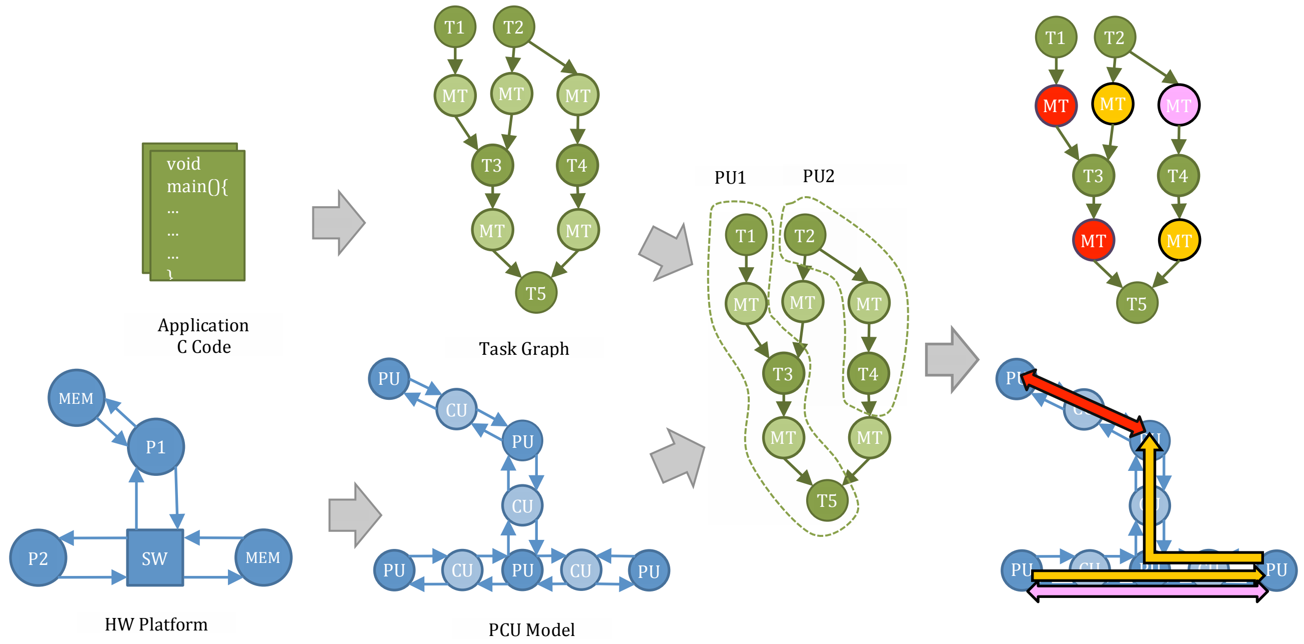


Figure 13 - Example of mapping with Task Graph and PCU Model with global and local memories

7 References

[1] Alexandru Turjan, Bart Kienhuis and Ed Deprettere, ``*Translating affine nested-loop programs to Process Networks*'', in proceedings of at the international conference on compilers, architecture, and synthesis for Embedded Systems (CASES'04), Sept 23 -- 25, 2004, Washington D.C

[2] Bart Kienhuis and Ed. F. Deprettere``*Modeling Stream-Based Applications using the SBF model of computation*'', Presented at: IEEE Workshop on Signal Processing Systems (SIPS 2001), Antwerp, Belgium, September 26-28, 2001

[3] O. Sinnen, "Task Scheduling for Parallel Systems", John Wiley, 2007

[4] M. Lindwer, L. Jozwiak, J. Madsen, B. Kienhuis, P. Meloni, L. Raffo, G. Notarangelo, "D1.1: Initial Design Methodology, Flow, and Tool Requirements", ASAM Project

[5] Theodore Johnson, "A Concurrent Dynamic Task Graph," icpp, vol. 2, pp.223-230, 1993 International Conference on Parallel Processing - ICPP'93 Vol2, 1993

[6] C. Glita, P. Dumont, and P. Boulet, "Arrys-OL with delays, a domain specific specification language for multidimensional intensive signal processing," Multidimentional System and Signal Processing, vol. 21, Mar. 2009, pp 105-131.

[7] L. Thiele et al., SHAPES @ TIK, http://www.tik.ee.ethz.ch/~shapes/dol.html