This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

Design-Trotter: System-Level Dynamic Estimation Task
a first step towards platform architecture selection

Yannick Le Moullec[1], Jean-Philippe Diguet[2], Thierry Gourdeaux[2], Jean-Luc Philippe[2]
[1] KOM Department, Aalborg University, DK-9220 Aalborg, Denmark
[2] LESTER Laboratory, University of South Brittany, F-56321 Lorient Cedex, France
ylm@kom.aau.dk, jean-philippe.diguet@univ-ubs.fr

Abstract

The objective of this work is to explore the design-space of digital embedded systems, before the high-level synthesis or compilation steps, in order to converge towards promising architecture-application matchings. This paper presents the first step of the "Design-Trotter" framework. This step, dedicated to the system-level design-space exploration, is performed before the SoC architecture definition and consists of functional and algorithmic explorations of the application. The first sub-goal of this work is to exhibit all the available parallelism of the application by means of an efficient graph representation. The second sub-goal is to guide the designer by means of dynamic estimates. These estimates are dynamic since they are represented by parallelism vs. delay trade-off curves, where a point represents a possible architecture model in terms of parallelism options for both processing and data-transfer operations and also for local memory requirements. This paper presents the original techniques that we have developed and some experiments that illustrate how the designer can benefit from our work to build or select an architecture.

Keywords: system-level, design-space exploration, functional/algorithmic exploration, metrics, parallelism, SoC.

1 Introduction

This work has been elaborated on the basis of the following observations. Firstly, the trade-off between energy savings, area usage and real-time constraints required for future embedded systems, for instance in telecommunications, imposes to optimize the usage of silicon. The necessary improvements of the efficiency ratios ($Mips/Watt$ and $Watt/mm^2$) can only be reached through i) the implementation of massive spatial parallelism, ii) the use of limited clock frequencies and iii) the design of more or less dedicated hardware modules [22]. VLSI technology now enables massive integration of processing units and fast communication channels on a single chip [5]. This search for parallelism and dedicated hardware opportunities constitutes an important task of the design flow. This task has to be performed before the complete definition of the target architecture and can be defined as system-level architectural exploration.

Secondly, though we now observe that CAD tools for co-synthesis and co-simulation have reached a reasonable degree of maturity, system-level exploration tools still remain at a research stage [18]. However, fast estimations based on largely abstract and incomplete specifications (of the application and the architecture) are vital at system level [10]. Most of existing tools provide an exploration based on a fixed architecture (e.g., an embedded processor connected to a FPGA) and a library of pre-characterized functions (on the elements of the fixed architecture). Such static libraries usually quantify one software implementation and one or two hardware implementations. Thus, the architectural exploration is bounded by the fixed function granularity and limited to hardware/software partitioning. Clearly, another step has to be performed on top of the one described above to really guide or parameterize architectural choices. The work described in this paper deals with this step.

The final point considers design decisions. It is crucial to get early estimations: firstly to shorten design delays, secondly to measure the impact of algorithmic choices and transformations rapidly and thirdly to adapt the subsequent architectural choices to the application parallelism. At system-level, we focus on this type of information by means of logical cycles and algorithmic operations, which are technology independent. As explained later, the architectural model can then be detailed in order to produce more accurate estimations when the target architecture has been chosen.

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

1.1 Contribution

Our work aims at bridging the gap between the specification of a system and the definition of a target (or a set of target) architecture(s) for that system. The exploration of the solution space for embedded systems can have different meanings. Our work focuses on the following aspects:

1. *Exhibition and Exploitation of the Parallelism*

   Parallelism is a key parameter in the design of digital systems. It has a direct impact on several performance factors such as execution time, energy consumption, and area. Therefore we seek to explore the potential parallelism of an application in terms of a) type (data-transfer and data-processing), b) granularity level, and c) type (spatial, temporal). In our work these aspects are addressed by means of our graph-based internal representation (detailed in 3.2.1) and via the time constrained core of our estimator (used with several time constraints). This estimator rapidly provides dynamic exploration of an application by means of parallelism vs. delay trade-off curves on which a point corresponds to a potential architecture. The designer has then access to a set of solutions from which he can choose the most promising ones according to his design constraints. These solutions can then be refined and mapped to technology dependent architectural models, using the next steps of our flow [21], [16] or existing co-design tools. The scheduler uses several new techniques regarding the resources computation, load balancing, loop and control estimations as well as HCDFG estimations. Those techniques are described in section 4.

2. *Target Architecture Definition*

   At the system level we assume that the target architecture is not yet defined. Therefore we can only consider algorithmic operators (operators performing the operations found in the application specification) since the goal of our work is to guide the architecture definition. At that level, no directive related to a specific technology is introduced. For that purpose we use an abstract model to represent the architecture components (for example time is expressed in cycles). The model used to describe this abstract architecture is defined in 3.2.2.

3. *Impact of Specification Choices*

   Since the core of our estimation framework is based on a fast scheduler, the designer can evaluate very rapidly the impact of algorithmic transformations. This feature is very important since it enables the exploration of several specifications for a given application. This is exemplified in section 5.

The rest of the paper is organized as follows. Section 2 presents related research work. In section 3 the framework steps and the models used for the application and the abstract architecture specifications are described. The different points of our approach regarding the system level estimation step are presented in section 4. Section 5 presents experimental results and discussions related to the different aspects of the method. Finally we conclude and present some perspectives in section 6.

2 State of the art

In this section we focus on the system-level exploration before any architectural definition. We do not address exploration based on hardware/software partitioning, nevertheless some recent references have been given in a previous paper [1] regarding real-time systems and in [19] for reconfigurable architectures.

System-level design-space exploration and platform-based design have been addressed by previous research work. In [3], a set of definitions and a methodology for platform-based design are presented. The main goal of the methodology is to favor design re-use and regularity in order to cope with increasing development costs. The design method is based on abstraction layers named "platforms". Platform components (including the architecture which is called "architecture platform") are partially or fully pre-designed. The method is based on a "meeting-in-the-middle" process where both specifications and abstractions of potential implementations are iteratively refined and finally meet each others.

ARTEMIS [2] is a modeling and simulation environment which permits design-space exploration at several abstraction levels. The methodology consists in mapping applications described as Kahn Process Networks onto an architecture model which is roughly and manually defined by the designer in a first approach. Both the application and architecture models are simulated (based on trace-driven approach). The performance of the system (represented by the architecture units workload), which results from the simulation, is then manually analyzed by the designer in order to perform HW/SW partitioning and iteratively refine the architecture.

Once the architecture has been defined by previous methods, it can be further refined. Platune [24] is a tool for exploring the configuration space of parameterized SoCs. The method enables all Pareto-optimal configurations in terms of execution time and power to

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

be found. This is done by taking into account the dependencies between the SoC parameters such as cache size and associativity, bus width, buffer sizes etc.

In DTSE [8], which is dedicated to data-transfer dominated applications, a complete design-space exploration is applied in order to gradually define an area and power efficient memory organization. The first steps (from pruning analysis to memory hierarchy) could be usefully performed before our estimation framework in order to provide promising memory hierarchies for data-transfer dominated functions. Regarding the following step, we propose a framework which is more general and not only focused on data management. Moreover the data-transfer scheduling in Atomium (a module of the DTSE methodology) is a force directed scheduling which is too complex to be used for fast and dynamic estimations. Finally, the last steps of Atomium can be profitable after the platform architectural choice, during the architecture design in order to optimize memory assignment and in-place data mapping.

All the previous methods and tools, except [8], rely on the designer's experience to define an initial architecture which will be further refined by means of design-space exploration. The first step of our method consists in exploring and exploiting the potential parallelism of an application. This dynamic exploration is expressed by means of trade-off curves on which a point corresponds to a potential architecture. This is a guidance tool for the designer, which helps him to define the initial architecture. Therefore our work is complementary to what already exists and helps the designer to choose or build an architecture. This architecture can then be refined by existing co-design methods or by the next steps of our flow as discussed in [16] and [21].

3 Design Flow and Models

3.1 Main steps of the design flow

The ideas presented in 1.1 have been implemented in our framework called Design-Trotter. Its flow is depicted in Fig.1.

3.1.1 System specification

The system to be estimated can firstly be specified at the task level. For example we have used methods and tools such as Radha Ratan in [1] and Esterel Studio in [14]. This work is not presented in this paper.

The actual entry point of our work is a task definition specified as a set of C functions. These functions are parsed into our internal representation, HCDFG, detailed in 3.2.1. The choice of the C language has been motivated by the availability of many standard specifications written in C and by the fact that the C language is still the defacto programming language for embedded systems. However, our framework is open and other languages could be considered in the future.

3.1.2 Function characterization

The characterization (point (1) in Fig.1) has two objectives. Firstly, it is used to sort the application functions according to their *criticity*. The criticity of a function is expressed as:

$$\gamma = \frac{NB\,op}{CP}$$

where $NBop$ is the sum of data-transfer and processing operations and CP is the critical path. This metric provides a first indication about the potential parallelism of the function. For instance, if $\gamma = 1$, there is hardly any chance of obtaining efficient hardware implementations. Secondly, it indicates the function orientation, i.e., the nature of the dominant types of operations in that function. By counting tests, data-transfers and processing operations within the HCDFG representation we can compute ratios which indicate the function orientation. These metrics are detailed in [15].

Once the functions have been characterized they are estimated. The context of this work is the design of a heterogeneous SoC architecture with (near) optimal performance/area and power/area ratios. Because of the size of the design-space, the estimation process is divided into two levels: the system level and the architectural level. The first one is performed by the function estimation step (point (2) in Fig.1) which is based on an abstract architecture model in order to rapidly explore a large set of parallelism options. The architecture model is abstract since it is not yet defined and that system level exploration corresponds to an algorithmic level exploration of the parallelism. The second one (corresponding to HW and SW projections, (points (3) and (4) in Fig.1) uses technological libraries to get accurate estimation results. In this paper we present the algorithms developed for the system level exploration; information regarding the architectural level can be found in [21] (hardware projection) and [16] (software projection).

3.1.3 Function level estimation

High-level synthesis tool use static (i.e., predefined) libraries that describe atomic components such as multipliers and adders. This is not sufficient for complex applications. These applications are made of several functional blocks (filters, transforms, quantification, etc.) that can be very different in terms of size, power consumption, and cost characteristics. These functional

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

blocks are complex enough to justify a dynamic architectural exploration.

The estimation step (point (2) in Fig.1) is based on a fast, time-constrained, list scheduler. The estimates produced by this step are dynamic since they are represented by parallelism vs. delay trade-off curves. Each point on a trade-off curve corresponds to a specific schedule for a given cycle budget. It indicates the amount of required processing and memory resources to meet the time constraint and therefore represents a possible architecture model in terms of parallelism options for both processing and data-transfer operations.

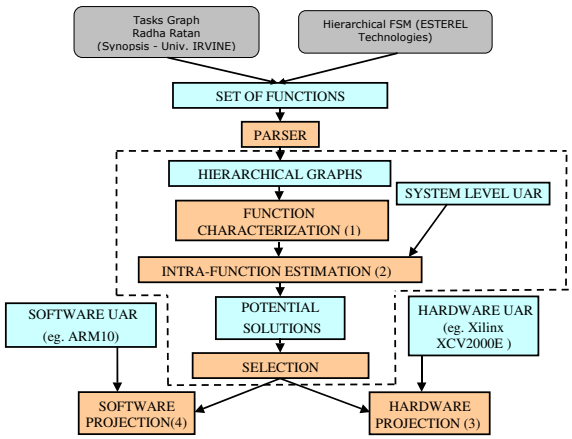


Figure 1: Design-Trotter design flow. The parts in the dotted box are presented in this article.

3.2 Models and Definitions

3.2.1 Application Specification

Definitions Each C function of the specification is a node at the top level of the Hierarchical Control and Data Flow Graph (HCDFG). A function is a HCDFG.

There are three types of elementary (i.e., non hierarchical) nodes, the granularity of which depends on the granularity of the architectural model defined in 3.2.2:

A **processing node** represents an arithmetic or logic operation, the granularity of the node depends on the architectural model: (ALU, MAC, +,-, etc.).

A **memory node** represents a data-transfer. The main node parameters are the transfer direction (read / write), the data format and the hierarchy level which can be fixed by the designer.

A **conditional node** represents a test operation (if, case, loops, etc.)

Three types of oriented edges are used to indicate scheduling constraints:

A **control dependency** indicates an order between operations without data-transfer, for instance a test operation must be scheduled before the mutual exclusive branches. The control dependency edges can also be used to impose a given order between independent operations or graphs with the intention of favoring resource optimization (data-reuse for instance).

A **scalar data dependency** between two nodes A and B indicates that node A uses a scalar data produced by node B.

A **multidimensional data dependency** is a data-dependency where the data produced is no more a scalar but an array. For instance such an edge is created between a *loop* CDFG reading an array transformed by another *loop* CDFG.

A **DFG** is a graph which contains only elementary memory and processing nodes. Namely it represents a sequence of non conditional instructions of the 'C' code.

A **CDFG** is a graph which represents a test or a loop pattern with associated DFGs.

A **HCDFG** is a graph which contains only elementary conditional nodes, HCDFGs and CDFGs.

Graph creation rules The decomposition principle is quite simple. The graph is traveled with a depth-first search algorithm. A H/CDFG is created when a conditional node is found in the next hierarchy level. When no more conditional nodes are found, a DFG is built. In order to facilitate the estimation process, CDFG patterns have been defined to identify rapidly *loop, if, etc.* specific nodes. Another important point is that the model covers the processing complexity of the complete application. Thus, the computation of array indexes (address computation), conditional tests and loop index evolution are represented with DFGs. A HCDFG example is given in Fig.14

3.2.2 Architecture Specification

The exploration process is based on a generic load-store architecture model, defined as a parameterizable processing unit supplied with data by a parameterizable memory hierarchy. The exploration process provides the estimated minimum processing parallelism, bandwidth parallelism and memory size required for a given cycle budget. The type of resources available are defined in the abstract architecture model. At the system level, the designer defines a set of rules, named "UAR" (User Abstract Rules). The processing part is characterized by the type of available resources: ALU, MAC, etc. and the operations they can perform; a number of cycles is associated to every type of operator. Regarding the memory part, the user defines the number of levels (L_i) of the hierarchy and the number of cycles (l_i) associated for each type of access. In Fig.2, four

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

levels are considered. The register level L_0, two cache levels L_1 and L_2 and a main memory L_3.

Fig.3 shows an example of two User Abstract Rules files. The first one (left) is the initial file where all resources have a latency equal to one cycle. When the designer starts to refine the architectural model (using results given by the system level estimation), he can add new types of resources or specify resource latencies like in the second file (right). Thus the designer may improve his analysis by means of system level estimation.

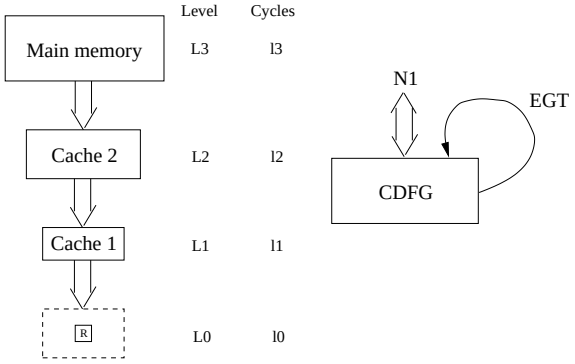


Figure 2: Memory hierarchy model

```
<LIBRARY> system
  <OPERATOR> Alu
      <OPERATIONS>
          "+","-","*","/",
          "<=","=","!=",">="
      <ENDOPERATIONS>
      <ATTRIBUTES>
          latency:cycle:=1
          datawidth:INT:=8
      <ENDATTRIBUTES>
  <ENDOPERATOR>
  <OPERATOR> Mac
      <OPERATIONS> "*+"
      <ENDOPERATIONS>
      <ATTRIBUTES>
          latency:cycle:=1
          datawidth:INT:=8
      <ENDATTRIBUTES>
  <ENDOPERATOR>

/* MEMORY */
  <MEMORY> RAM_DP
      <ATTRIBUTES>
          access_mode:=rw
          latency_read:cycle:=1
          latency_write:cycle:=1
      <ENDATTRIBUTES>
  <ENDMEMORY>
<ENDLIBRARY>
```

```
<LIBRARY> system
  <OPERATOR> Alu
      <OPERATIONS>
          "+","-","*","/",
          "<=","=","!=",">="
      <ENDOPERATIONS>
      <ATTRIBUTES>
          latency:cycle:=2
          datawidth:INT:=32
      <ENDATTRIBUTES>
  <ENDOPERATOR>
  <OPERATOR> Mac
      <OPERATIONS> "*+"
      <ENDOPERATIONS>
      <ATTRIBUTES>
          latency:cycle:=3
          datawidth:INT:=32
      <ENDATTRIBUTES>
  <ENDOPERATOR>

/* MEMORY */
  <MEMORY> RAM_DP_LEVEL_1
      <ATTRIBUTES>
          access_mode:=rw
          latency_read:cycle:=1
          latency_write:cycle:=2
      <ENDATTRIBUTES>
  <ENDMEMORY>
  <MEMORY> RAM_DP_LEVEL_2
      <ATTRIBUTES>
          access_mode:=rw
          latency_read:cycle:=2
          latency_write:cycle:=3
      <ENDATTRIBUTES>
  <ENDMEMORY>
<ENDLIBRARY>
```

Figure 3: UAR file examples (left: first approach; right: refinement)

In this paper we focus on performing the design space exploration before the architecture selection. For this, we use a generic architecture model based on an algorithmic-level UAR. Regarding the subsequent architectural-levels of Design-Trotter, HW and SW UAR models are used (cf. Fig.1). They provide accurate execution time and area information; these libraries are obtained either by synthesizing operators on specific FPGAs or, when a software projection is performed [16], by using the processor specification tool Armor [7] which enables execution delays and dependencies between instructions to be indicated accurately.

4 System Level Estimation

In this section we present the different points of the system level estimation. Firstly, we introduce some memory considerations and define the Data Reuse Metric (DRM). Then we present the sub-steps of the estimation process : DFGs estimation by means of scheduling (guided by the DRM metric), CDFGs estimation for loop and control structures and HCDFGs estimation by means of combination techniques.

4.1 Memory features

4.1.1 Introduction: local/global accesses

A key point in the following sections is the notion of local and global accesses from a graph point of view. We have distinguished several types of memory nodes:

- N1: data identified as inputs/outputs of a graph ;
- N2: temporary data produced by internal processings ;
- N3: input data reused in a graph (N1 subset);
- N4: accumulation data annotated with pragmas during the specification.

Without any architectural assumption we consider a "graph-based" hierarchy as a systematic analysis support. A node is called local if it is internal to a graph and is called global if it is external to the graph (i.e., it represents an I/O of the graph). N1 data are always global, N4 data always local, N2 and N3 data are initially local but can generate Extra Global Transfers (EGT). The DRM metric gives the global/local accesses ratio. A local access which produces a memory conflict (when the local memory is full) involves a global read and write operations, thus the number of extra global accesses is $\beta \times EGT$ where β is the average number of cycles required to access to the global memory. In the case of a single main memory which requires only one cycle per access $\beta = 2$ (read plus write).

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

Our framework currently focuses on data-transfer and processing parallelism exploration. The estimation of the memory size is not the objective of our work. However, we may need some information about the memory organization in order to quantify data accesses. Firstly, we have introduced a technique for data reuse quantification before design space exploration, it is detailed in (4.1.2). Secondly, as summarized in 4.1.3, we have added a fast and simple local and background memory estimation technique.

4.1.2 Data Reuse Metric (DRM)

In order to guide the analysis of the functions and the scheduling of the DFGs, we have defined a metric called *DRM: Data Reuse Metric*. This metric takes into account the local memory size, which has to be fixed or estimated. Given the abstraction level of the method, accurate temporal estimations are not essential. In fact, the distribution of memory accesses over the time (expressed in logical cycles) is sufficient to compare several solutions. Moreover, as we want to obtain the estimates rapidly, methods such as clique coloring have been discarded.

We use average data lifetime to estimate the quantity of data alive in each cycle, from which the minimum memory size can be derived. The minimum and maximum data-life of data d are defined as follows:

$$MinDL(d) = ASAP(d^n) - ALAP(d^1) + 1$$

$$MaxDL(d) = ALAP(d^n) - ASAP(d^1) + 1$$

where $ASAP$ and $ALAP$ are the earliest and latest scheduling dates , respectively; d^1 and d^n the earliest and the latest read access to data d for a given time constraint, respectively. The average data-life of data d is then given by:

$$AvDL(d) = \frac{1}{2}\left(MinDL(d) + MaxDL(d)\right)$$

Finally, the number of data alive per cycle is given by:

$$AAD = \frac{1}{T}\sum_{d} AvDL(d)$$

where T is the number of cycles allocated to the estimated function. The number of local transfers turning into global transfers because of a too small local memory is given by:

$$EGT = \begin{cases} (AAD - UM)T \; if \; AAD > UM \\ 0 \; otherwise \end{cases}$$

where UM is the local memory sized. UM can be defined by the user, its default value is AAD.

If we consider the memory hierarchy given in Fig.2 with the following characteristics, L1: $l1 = 1$, L2: $l2 = 2$, L3: $l3 = 3$, and if MR_i is the miss ratio of the cache level i, then:

$$\beta = 1.(1 - MR_1) + 2.MR_1.(1 - MR_2) + 3.MR_1.MR_2$$

If we generalize to K levels of hierarchy, we obtain:

$$\beta = \sum_{k=1}^{K} \left(l_k.(1 - MR_k).\prod_{j=1}^{K-1} MR_j \right)$$

Finally the DRM metric is obtained as follows:

$$DRM = \frac{N1 + \beta.EGT}{N1 + N2 + N3 + N4}$$

Remark: Note that for a given DFG the data-transfer scheduling is independent from the cache technique (write back, write through). However in the context of power estimation, transfers between cache and main memory must be taken into account [23].

4.1.3 Memory size estimation for loop-CDFGs

This not the main focus of this paper, however it is necessary to explain how we consider memory size estimation for the bandwidth and processing parallelism estimations. The memory space is divided into parts, the first one is the local memory (L_0) that represents the memory space required to supply the processing unit with data for a given cycle budget. Without any space constraints, the local memory size is increased or decreased during the scheduling process according to the application requirements. Data-reuse is considered since a data is assumed to be available in local memory during its span life, namely between from its first read until its last write. As scheduling is performed with polynomial algorithms, we obtain a very fast local memory size estimation. In another mode of use, the user can specify a maximum value for the memory local memory size; in such a case, when the limit is reached, some extra cycles are added during scheduling depending on the number of cycles associated with the memory level L_1.

The second memory aspect concerns the background memory (L_i levels with $i > 0$). The designer can choose a memory hierarchy for each loop-oriented function of the application and then associate a cycle budget for each type of access. This memory hierarchy decision could stem from a high level memory management as explained in [23].

We have implemented a simple and fast method that consists in considering a single level background memory with a size equal to the sum of declared arrays. Moreover, each loop-CDFG is characterized by the size of input and output arrays in order to provide memory hierarchy candidates. The access delays (cycles) are decided by the designer (the default value is one cycle).

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

As a result, an histogram provides the designer with the memory size for each data-type (byte, long, etc.). Some more accurate values can also be obtained at the system level: we have also implemented the background memory estimation method based on polyhedra computation detailed in [6] and [17]. This work has been performed with the Polylib library from IRISA [9], that has been adapted to our HCDFG graph. Regarding this point, it appeared that the HCDFG structure was particulary efficient to retrieve the assignment order and the index computation since each array access is enclosed in one or more loop-CDFG where information like loop bounds and index evolutions are specified. However, this technique has limitations that can be awkward in the context of design space exploration. Firstly, it implies to choose a sequential execution in the case of parallel loops and, as explained later, our method tries to use the available parallelism when necessary. Secondly it cannot produce optimized results when the evolution of array indexes is data-dependent. Such an example can be observed in section 5 from the *object motion detection* application where accesses to the array *TabEqui[]* depend on the data from *CharLabel[]*. Background memory estimation is better used as a first design step to tune array sizes and fixe loop ordering by introducing artificial loop dependencies if necessary. As a result, a first memory organization is obtained and can then be refined with the memory hierarchy decision step based on data reuse optimization. In a global and complete design flow, our estimation framework, based on loop dependencies and array size declarations can then take benefit from these two first steps.

4.2 Adaptive DFG Estimation

4.2.1 Scheduling principle

With the information contained in the graph and the designer rules (UAR), the goal of the method is to minimize the quantity of resources and the bandwidth requirement for several time constraints.

The scheduling principle is a time-constrained *list-scheduling* heuristic where the number of resources of type i allocated at the beginning is given by the lower bound:

$$Nb\ resource_{\ type\ i} = \lceil \frac{Nb\ operations_{\ type\ i}}{T} \rceil$$

The method includes three sub-algorithms: processing first, data-transfer first and mixed.

Scheduling first the most critical type of operations enables a reduction of the quantity of resources used for that type; this implies further constraints for the scheduling of the other type. The type of algorithm to be applied can be selected by the designer or automatically chosen by our tool thanks to the DRM metric. Functions with high DRM values are data-transfer oriented, whereas functions with low DRM values are processing oriented. Defining precisely the boundaries between the different orientations is not a trivial task and we are currently working on this aspect.

Processing first and data-transfer first algorithms are used for processing and data-transfer oriented functions respectively. For each cycle the average number of resources (resp. buses) is computed and the processing (resp. memory) nodes are scheduled. Then ASAP and ALAP dates of memory (resp. processing) nodes are updated according to the scheduling dates of the processing (resp. memory) nodes. Finally memory (resp. processing) nodes are scheduled, using the same average computation technique.

However, traditional architectural synthesis approaches based on the dissociation memory/processing are not always adapted to all cases. Indeed, for some cases, it is necessary to handle processing and memory scheduling simultaneously (usually when memory transfers and processing are balanced). The third algorithm, named "mixed", is used when it is not possible to define the function orientation precisely. In this case, both memory and processing nodes are handled with the same priority. As with the two other algorithms, the average number of resources is computed for every cycle and nodes are scheduled according to their priority (which is a function of the nodes ASAP and ALAP dates as well as their mobility).

In order to minimize the quantity of necessary resources and to favor reuse, new techniques have been employed. These are described in the following sections.

4.2.2 Online Average Resource Computation (OARC)

Heuristics such as list-scheduling allow rapid computation of the resource/delay trade-off curves. However, heuristics may lead to the rejection of locally optimal solutions. Therefore, because of data dependencies, the quantity of resources which has been allocated at the beginning of the scheduling process is not sufficient, new ones are allocated during scheduling, but will not be used optimally. To minimize over-allocation of resources, we propose computing the average number of resources for every cycle while scheduling (complexity $O(1)$) in order to "smooth" the trade-off curves. The average number of resources of type i at cycle C_j is computed as follows:

$$Nb_{avg}\ R_i(C_j) = \lceil \frac{NRN_i(C_j)}{NRC(C_j)} \rceil$$

where Nb_{avg} R_i is the average number of resources of type i, NRN_i the number of remaining nodes using resources of type i and NRC is the number of remaining cycles. The example in Fig.4 (schedule of a DCT) shows how the OARC method saves two memory access resources. Memory nodes are white and processing nodes are grey. In case (A) the average number of memory access resources is computed only once and equals 1 ($\lceil \frac{23}{24} \rceil$). Memory nodes are therefore scheduled using this resource. As long as the nodes mobility is different of 0 no extra resource is allocated. At cycle 24 there are 4 memory nodes remaining, the mobility of which equals 0. As 3 extra resources are allocated and used during one cycle only, they are not efficiently used. In case (B) scheduling is performed using the OARC technique. The first part of the resulting schedule is identical to the previous case. The difference appears at cycle 18 where the average number of memory access resources equals 2 ($\lceil \frac{8}{7} \rceil$). So at cycle 18 one extra memory access resource is allocated and is used more efficiently than the 3 others in the previous case. In case (A) the total number of resources equals 6 (2 processing resources + 4 memory access resources) whereas in case (B) this number equals 4 (2 processing resources + 2 memory access resources).

4.2.3 The "draw pile"

This technique is applied for processing first and data-transfer first algorithms. Once processing nodes or memory nodes have been scheduled, some non-used cycles can remain, (i.e., the execution time is shorter than the time constraint). These cycles located at the end of the schedule are collected in a "draw pile". While scheduling memory access or processing nodes respectively, it is possible to "pick" free cycles in the pile, in order to shift the nodes for which the number of resources is not sufficient. Once the processing or memory access schedule has been obtained, it cannot be modified. The only possible variation is to shift a set of scheduled nodes to the "right" by taking advantage of the freedom offered by the "draw pile" while maintaining their relative ordering. The example in Fig.5 depicts how this technique works: the data-transfer first algorithm is used with a time constraint equal to 22 cycles. Firstly, (A), the memory nodes are scheduled using two resources ($\lceil \frac{23}{22} \rceil$). Then, (B), the processing nodes are scheduled using 4 resources and 10 cycles (13 to 22) are not used. When the "draw pile" technique is applied, (C), it is possible to shift the memory nodes schedule to the right thus enabling the insertion of some processing nodes. Then only one processing resource is needed instead of four previously.

4.3 CDFG Estimation

4.3.1 Loop scheduling

The general scheme used to estimate a loop is as follows: 1) the three parts of the loop (evaluation, core and evolution) are estimated using DFG scheduling and CDFG combinations (cf. 4.4). 2) the loop "pattern" is estimated by performing a sequential combination of the three parts. 3) the whole loop is estimated by repeating N times the loop pattern (with N the number of iterations). However, in order to explore fully the available parallelism we have developed techniques to unfold loops.

4.3.2 Loop exploration

Unfolding loops permits exposure of the potential parallelism and thus also allows a reduction of the function critical path. This can be helpful to successfully schedule a function in a given time constraint. However, unfolding a loop is limited by data dependencies. We distinguish two types of dependencies: 1) memory dependencies, and 2) data-flow (a.k.a true) dependencies.

Memory dependencies This type of dependency can be eliminated by using structural transformations as depicted in Alg.1. In the general case, a loop of size N, unfolded L times, is transformed into a L branches tree and its critical path is given by:

$$CP_L = \lceil \frac{N}{L} \rceil + \lceil Log_2(L) \rceil$$

Algorithm 1 Memory dependencies elimination: example

```
y(k) = 0
for i = 0 to N - 1 do
  y(k) = y(k) + a_i x(k - i)
end for
```

can be re-written as:

```
y1(k) = 0
y2(k) = 0
for i = 0 to N/2 - 1 do
  y1(k) = y1(k) + a_i x(k - i)
  y2(k) = y2(k) + a_{i+N/2} x(k - (i + N/2))
end for
y(k) = y1(k)+y2(k)
```

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

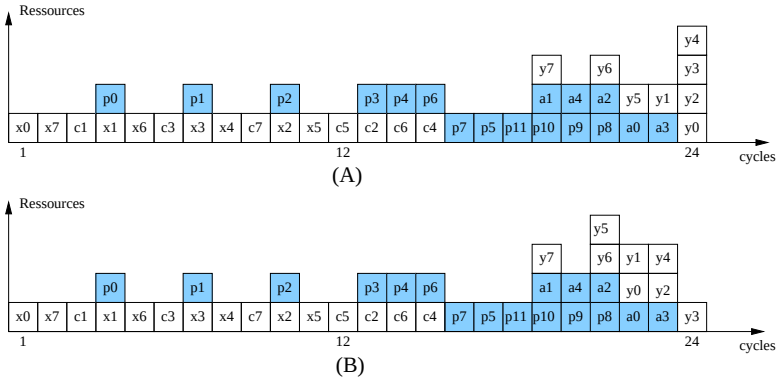


Figure 4: The OARC technique: (A) scheduling without the OARC technique, (B) scheduling with the OARC technique, which saves 2 memory access resources. (Memory accesses (x's and c's) are represented by white boxes, processing nodes (p's and a's) are represented by grey boxes)

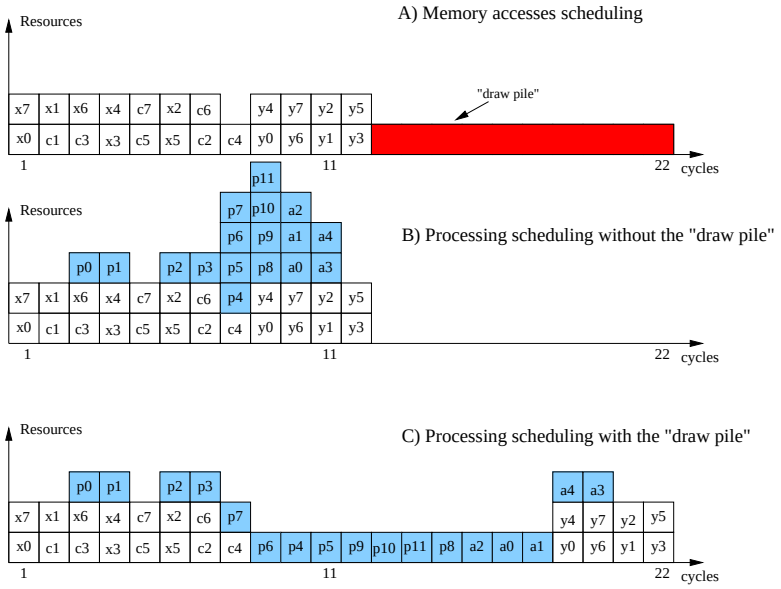


Figure 5: The draw pile technique: (A) memory accesses are scheduled, (B) processing nodes are scheduled without the "draw pile" technique and (C) processing nodes are scheduled with the draw pile technique, which saves 3 processing resources (Memory accesses (x's and c's) are represented by white boxes, processing nodes (p's and a's) are represented by grey boxes)

Data-flow dependencies For data-flow dependencies, researches usually focus on the unfolding factor α necessary to obtain the optimal rate (i.e., the unfolding factor which maximizes the throughput). In [4], it is shown that α can be computed as:

$$\alpha_{mgpr} = \lceil \frac{T_{max} PGCD(\Delta_{cr}, T_{cr})}{T_{cr}} \rceil \frac{\Delta_{cr}}{PGCD(\Delta_{cr}, T_{cr})}$$

where T_{max} is the latency of the slowest operator, Δ_{cr} the number of delays and T_{cr} the accumulated latency of the critical cycle.

Our problem is different, we have to find the minimum unfolding factor to schedule a loop respecting a time constraint T. Therefore, we have re-formulated the problem as: if a loop requires T" to be scheduled without unfolding, we must find the unfolding factor to gain $G = T" - T$ cycles. We can show that the number of cycles obtained after unfolding a loop with a factor α is computed as:

$$T" = \frac{T + (\alpha - 1) d_{min}}{\alpha}$$

where

$$d_{min} = \lceil max_{cycles}(\frac{T_{cr}}{\Delta_{cr}}) \rceil$$

Finally the unfolding factor α to gain G cycles is given by:

$$\alpha = \frac{1}{1 - \frac{G}{T - d_{min}}}$$

The main points of our loop management policy are: *i)* loop unfolding is included in the trade-off curves computation as a solution to reach resource lower bounds, *ii)* data-flow and memory dependencies are both considered whereas they are usually implemented in frameworks dedicated to different application domains and *iii)* the unfolding factor is computed with the aim to meet time constraints instead of finding the optimal rate.

4.3.3 Control Scheduling

This section explains how control constructs (i.e., CDFGs) are estimated. DFG cores contained in CDFGs are estimated as explained in the previous section. We have to deal with two types of control: *i)* non-deterministic control (tests which are not solvable during compilation or synthesis [11]) and *ii)* deterministic control (bounded loops and tests which can be removed). Here we present our method for non-deterministic control. We have to deal with two cases: equiprobable and non-equiprobable branches. Control management consists in allocating the necessary resources to execute the branches and to schedule these branches with the aim of favoring resource reuse. We use application profiling to obtain branches visit probabilities.

Equiprobable branches: 1) Compute average resource quantity for all the branches (as well as memory size if it has not been defined by the user).
2) Allocate resources according to the MAX values (between all the branches).
3) Schedule the branches, starting with the most critical one. If the number of resources is not sufficient new resource allocation is allowed.

Non-equiprobable branches: 1) Consider two branches a and b with probabilities Pa and Pb which are non-equiprobable (e.g., $Pa \ll Pb$). First a is scheduled, i.e., the most probable (and not the most critical) one. This results in a schedule of length T (equal to the time constraint T_c).
2) Then the other branch is scheduled reusing the resources allocated for the first one (for non-equiprobable branches we do not increase resources if their type is already present in the first branch). This resource constrained scheduling results in a schedule of length T'. If $T' \leq T$ then the schedule obtained is considered as valid. If $T' > T$ then we need to allocate more time to the second branch (as we do not want to allocate more resources). As the probability of going through branch a is much higher than for branch b, we try to reduce the scheduling length T to "save" cycles that can be allocated later on to branch b. In this case, the respect of the time constraint is considered globally, i.e., the cycles which are saved on the most probable branch will be used to enable the respect of the time constraint for the second branch. The time constraint is considered more important than the cost of the extra buffer required to absorb the heterogeneousness of path delays. The number of passages through a is equal to $Pa \times N$, where N is the number of iterations. In $\Delta T \times Pa \times N$ the b branch can be executed $Pi \times N$ times ($\Delta T = T' - T$). Thus we must schedule branch a within T" such as: $T" = T - \lceil \frac{\Delta T}{Pa/Pb} \rceil$.
3) However, reducing T may lead to increase resources for branch a. In that case, we select the solution which minimizes the number of resources required to perform the function with respect to the time constraint T_c (i.e., reduce or not T for branch a).

4.4 HCDFG Combination

The estimation process for a complete function is hierarchical. Firstly, the lowest levels (DFGs) are estimated by means of adaptive scheduling, OARC and draw pile (4.2.1). Then CDFGs are estimated, using the loop and control patterns as described previously ((4.3)). Finally combination rules are applied in order to estimate sequential and parallel executions (which permits to estimate hierarchy patterns).

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

During this bottom-up approach, the combination order is guided by the criticity metric (cf. 3.1.2): when three or more elements have to be combined, the two most critical ones are combined first and so on. Moreover, we want to produce the estimates very rapidly in order to explore large design-spaces. So, considering the important number of points in each trade-off curve we cannot apply an exhaustive search of the Pareto points like in [20]. Instead, we use a technique based on CDFGs trade-off curve peculiar points (PP). A peculiar point corresponds to a solution for which the cost in terms of resources decreases for a given time constraint. A peculiar point is noted *PP* and the set of peculiar points for a graph x is noted *PPGx[]*. Note that the term resource is then is used for processing units, data transfers and local memories.

4.4.1 Sequential CDFGs

The method used to estimate sequential CDFGs is illustrated in Fig.6 where the trade-off curve of the global CDFG1-2 is optimally built while combining the trade-off curves of the sequential CDFG1 and CDFG2.

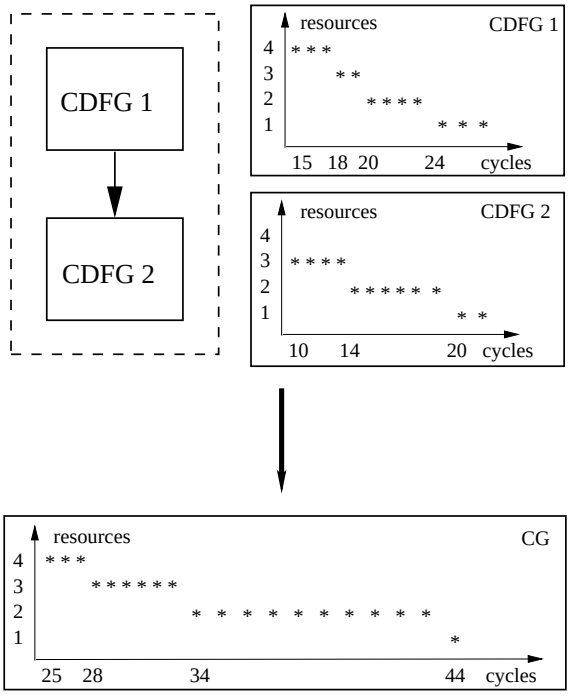


Figure 6: Sequential CDFGs combination.

The minimum execution time of the sequence $[CDFG1 \ then \ CDFG2]$ is given by $t_{min} = t_{min}(CDFG1) + t_{min}(CDFG2) = 25$. The algorithm firstly selects the CDFG with the highest cost in terms of resources (resources which can therefore be reused by the second CDFG). Then the next peculiar point of the selected CDFG is processed and the second point of the global CDFG trade-off curve is created for a constraint which is demonstrated in this example: $t_2 = t_{2^{nd}PP}(CDFG1) + t_{min}(CDFG2) = 18 + 10 = 28$. Then, the process is iterated until the minimum (1 resource or a lower bound fixed by the designer) is reached. The algorithm is presented in Alg.2, it's complexity is O(n) with n the number of peculiar points.

Algorithm 2 Sequential CDFGs combination algorithm

```
PPG1[1..maxIndex1]:  Peculiar Points of graph 1
PPG2[1..maxIndex2]:  Peculiar Points of graph 2
PPCG[1..maxIndexG]: Peculiar Points of the combined graph
PPCG[1]← PPG1[1]+PPG2[1]
indexCG← 1 index1← 1 index2← 1
t1← PPG1[1]
t2← PPG2[1]
tmax← PPG1[maxIndex1]+PPG1[maxIndex2]
t← PPCG[1]
while t≤Tmax do
  indexCG++
  if G1[PPG1[index1]]>G2[PPG2[index2]] then
    t1← PPG1[index1+1]
    index1++
  else if G1[PPG1[index1]]<G2[PPG2[index2]] then
    t2← PPG2[index2+1]
    index2++
  else
    if G1[PPG1[index1+1]]>G2[PPG2[index2+1]] then
      t1← PPG1[index1+1]
      index1++
    else
      t2← PPG2[index2+1]
      index2++
    end if
  end if
  CG[t1+t2]← MAX_{j=1;2}[G_j[PPGi][ij]]
  PPCG[indexCG← t1+t2]
end while
```

4.4.2 Parallel CDFGs

The method used to estimate parallel CDFGs is illustrated in Fig.7. In the case of concurrent CDFGs, an important issue is that CDFGs can share resources in a given cycle. The minimum cycle budget is $t_{min} = [MAX[PPG1[1], PPG2[1]]$. The algorithm principle is to select the CDFG with the maximal cost and to extend its cycle budget to its next PP. Then a fusion of scheduling profiles (a trade-off point corresponds to a specific scheduling result) of both CDFGs is performed and the maximal value gives the Y point of the global CDFG trade-off curve. Thus, the algorithm pointer jumps between the PPs of both CDFGs while always selecting the CDFG with the highest cost. The algorithm is detailed in Alg.3. There are possible optimizations: i) with or without profile fusion (in the later case resource sharing is not taken into account for each cycle), ii) fusion with or without taking into account graph mobility (relatively to each other). Here we consider profile fusion without mobility. Note that profile fusion increases the estimation complexity; this

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

option can be de-activated if speed is the main concern. The algorithm complexity is O(N.T) if the fusion option is used and O(N) otherwise, with N the number of PPs and T the average time constraint.

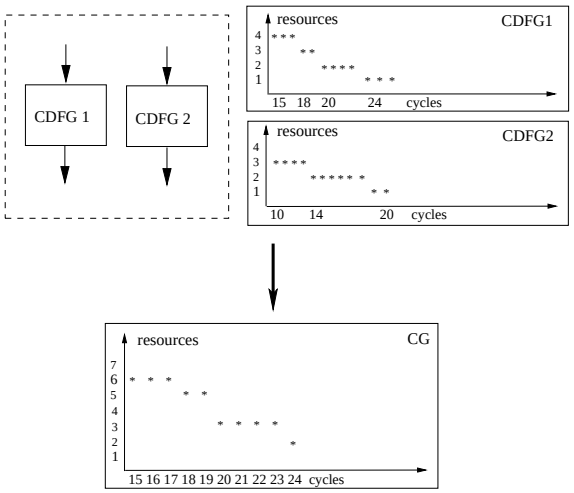


Figure 7: Parallel CDFGs combination, without profile fusion.

5 Experimental Results

Our framework has been implemented in a Java tool named Design-Trotter. Some views of the tool are shown in Fig.8 where (A) is the result window for the hierarchy level selected in the HCDFG structure available in (B). In (A) the designer gets dynamic resource estimations including processing units, data accesses, the local memory size, loop unrolling factors and the DRM for each Pareto cycle budget between the critical path and the completely sequential solution. In (B) the designer can select any graph or subgraph of the application. In window (1), the designer selects a solution, namely a cycle budget ($T = 8136$) for a given sub-graph IF#2. This action induces the opening of the estimation windows for the sub-graphs (2),(3) and (4) with an indication about the cycle budget that have been selected in sub-graphs (2),(3) and (4) to compute the solution selected in (1). Finally, when DFG (5) is reached then the associated schedule is provided.

In order to illustrate our methodology and the techniques discussed in the previous sections, we have experimented with several examples which are now discussed.

5.1 Adaptive DFG Scheduling

The different scheduling algorithms have been applied to the Lee DCT example (Fig.9), for several time con-

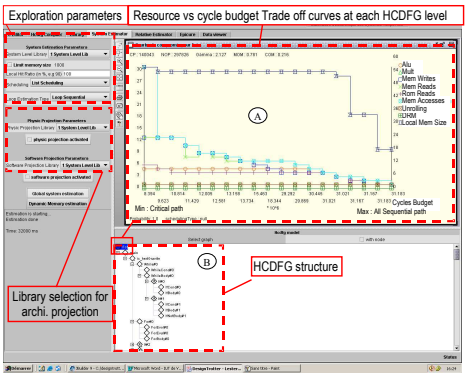
Algorithm 3 Parallel CDFGs combination algorithm

```
PPG1[1..maxIndex1]:  Peculiar Points of graph 1
PPG2[1..maxIndex2]:  Peculiar Points of graph 2
PPCG[1..maxIndexG]: Peculiar Points of the combined graph
Gi[k,1..k] profile of graph i (from 1 to k cycles) for
trade-off curve of time constraint k
tmin← MAX[PPG1[1];PPG2[1]]
tmax← MAX[PPG1[maxIndex1];PPG1[maxIndex2]]
t←tmin
indexCG← 1 index1← 1 index2← 1
while t≤Tmax do
  max← 0
  mobility,j ←index-extension
  for c=1 TO t do
    CG[indexCG,c]← G1[t,c]+G2[t,c]
    if CG[indexCG,c]>max then
      max← CG[indexCG,c]
    end if
  end for
  CG[t]← max
  PPCG[indexCG]← t
  if PPG1[index1+1]<PPG2[index2+1] then
    t← PPG1[index1+1]
    index1++
  else
    t← PPG2[index2+1]
    index2++
  end if
  indexCG++
end while
cost← CG[indexCG-1]
if cost>1 then
  t← number of operation
end if
while cost>1 do
  max← 0
  for c=1 TO t do
    CG[indexCG,c]← G1[t,c]+G2[t,c]
    if CG[indexCG,c]>max then
      maw← CG[indexCG,c]
    end if
  end for
  cost← max
  t++
end while
CG[indexCG]← 1
PPGC[indexCG]← t



/************************************/
/**********Index Extension**********/
if ∃ PPG_k[indexk]<tmin with k∈1,2 then
  if ∃ j>indexk such as PP_k[j]<tmin then
    index_k ←j
  end if
end if
/************************************/
```

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005



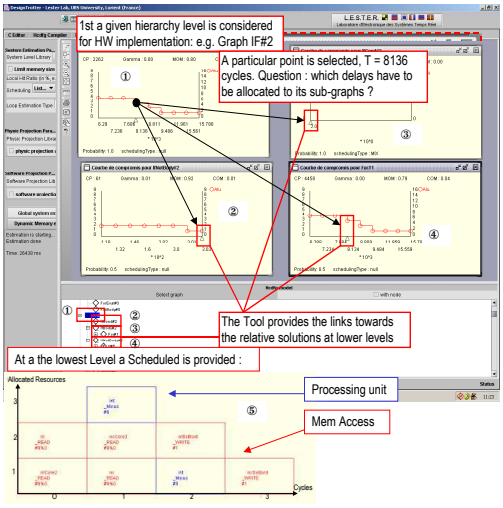


Figure 8: Design-Trotter User Interface

straints. The results (delay/resources trade-off curves) presented in Fig.10 to Fig.12, enables the comparison of the amount of processing and data-transfer resources for 4 scheduling algorithms. Indeed, apart from the three scheduling algorithms ("processing first", "memory first" and "mixed"), we have implemented the Force-Directed Scheduling (FDS) algorithm (with a "mixed" approach) in order to compare and tune our algorithms to existing work. The memory size can be fixed by the designer in order to compare several algorithms with several time constraints while achieving a schedule within the critical path time constraint. In this example, the local memory size chosen equals 8 data. In fact, the designer has also the possibility to enable the infinite local memory size option of our tool. In that case the tool saves the peak value during the scheduling of the requested local memory size and hands it out to the designer.

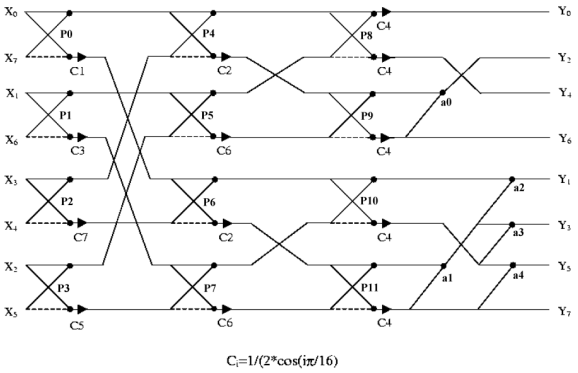


Figure 9: Lee algorithm for the DCT computation

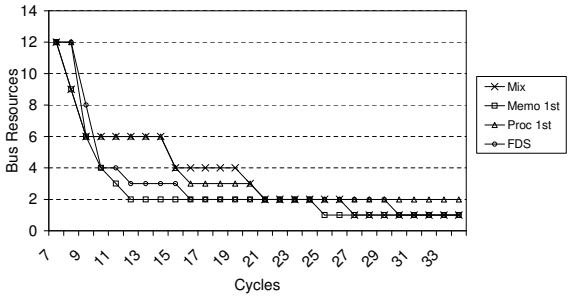


Figure 10: Comparison of the scheduling algorithms: data-transfer resource trade-off curve obtained for a local memory size equal to 8

Regarding our case study (Figs. 10, 11, 12) the first observation that can be made is that although the FDS scheduling algorithm gives good results, its computational time is prohibitive in the context of system-level design-space exploration. Indeed, 50 minutes were nec-

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

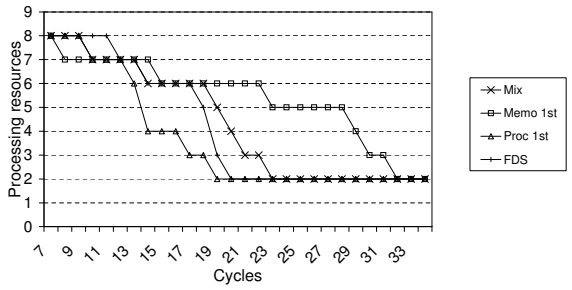


Figure 11: Comparison of the scheduling algorithms: processing resource (ALU + MAC) trade-off curve obtained for a local memory size equal to 8

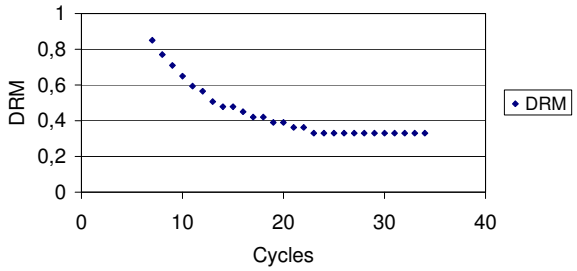


Figure 12: DRM metric evolution for a local memory size equal to 8

essary to compute the 28 points of the trade-off curve with the FDS algorithm whereas only 0.6 seconds (both on a 1Ghz Pentium III) have been necessary with the three algorithms that we have presented previously.

With regard to the three other scheduling algorithms, the following remarks can be made. When DRM values are high the function is data-transfer oriented, in this case with a local memory size equal to 8 and for tight time constraints (from 7 cycles to 19 cycles in Fig.11) the memory first algorithm enables a reduction of data-transfer resources thanks to its scheduling order. For lower DRM values the function is processing oriented. In this case, the processing first algorithm minimizes processing resources (from 19 cycles to 26 cycles in Fig.10). Moreover, processing operation regularity can lead to a measure of data-transfer regularity (as in this example), in such cases the processing first algorithm can perform better than the mixed algorithm. Finally, when neither memory size nor time constraints are hard, the mixed algorithm takes advantage of its flexibility, i.e., it is not worth scheduling one type of node before an other one (from 27 cycles to 34 cycles in Fig.11 and Fig.10).

5.2 CDFG Estimation: loops

Algorithm 4 Adaptive filter example

$y = 0$
`for i=0 to 1023 do`
  $y = h(i) \times e(k-i) + y$
`end for`
$s(k) = y$
$adapt = 2\mu \times (yt - y)$
$h(0) = h(0) + adapt \times xt$
`for i=0 to 1023 do`
  $h(i) = h(i) + adapt \times e(i)$
`end for`
$@e(k) = @e(k) + 1 modulo 1024$

The second example is an adaptive filter, which is described in Alg.4. The filter example has been used to test loop unfolding within CDFGs , the results are presented in Fig.13. Without unfolding (unfolding factor = 0, cf. Fig.13), 10250 cycles are necessary to perform the algorithm. If the function must be performed in less than 3500 cycles it is necessary to unfold the loops with $\alpha = 3$ (cf. Fig.13); the new loop critical path is then $CL_3 = \lceil 5120/3 \rceil + \lceil Log_2(3) \rceil = 1709$ and the whole function needs 3428 cycles to be performed. If the time constraint is even smaller, e.g., if the function must be performed in less than 2600 cycles, it is necessary to unfold the loop with $\alpha = 4$ (cf. Fig.13), the new critical path is $CL_4 = \lceil 5120/4 \rceil + \lceil Log_2(4) \rceil = 1282$ and the function requires 2574 cycles.

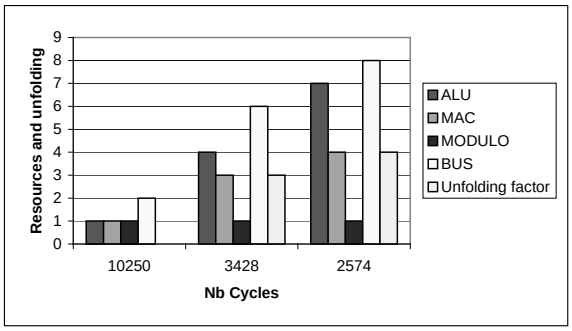


Figure 13: Loop unfolding and necessary resources vs. time constraints for the adaptive filter example

5.3 CDFG Estimation: control scheduling

The third example is an *object motion detection* described in [13]. This application is typically embedded in video cameras and used for parking lot monitoring (detection of car and person movements), person counting in places such as subways and so on. We have used a large set of representative input data (from a parking lot monitoring appliance) to produce a profiling of the application functions. Algo.5 shows a simplified ver-

sion of the *labeling* function of the application. After a threshold stage, the current initial image is processed in order to assign an object label to each pixel.

Algorithm 5 A control structure within the *labeling* function from the object motion estimation application

```
for y=0 TO number of lines (nL) do
  for x=0 TO number of columns (nC) do
    ...
    if ((CharLabel[xyOffset-1]==0)
    and(CharLabel[xyOffset-nC]==0)) then
      CharLabel[xyOffset]=newLabel;
      newLabel=newLabel+5;
    else
      CharLabel[xyOffset]=TabEqui[CharLabel[xyOffset-1]];
      TabEqui[CharLabel[xyOffset-Image.nC]]
      =CharLabel[xyOffset];
    end if
    ...
  end for
end for
```

5.3.1 Distribution of cycles

Probabilities for the two branches of the example presented in Alg.5 have been obtained from the profiling step. The branch taken when the test condition is true ('then' statement) is labeled a, the other one ('else' statement) is labeled b. The probabilities to pass through branches a and b are $Pa = 0.984$ and $Pb = 0.016$ respectively. As $Pa \ll Pb$, the method schedules the branch a with a time constraint $Tc = 7$ cycles (its minimum execution time). Then it schedules the b branch, but that one requires 14 cycles to be scheduled. In order to "give" more time to the b branch the a branch is re-scheduled with a new time constraint given by: $T" = 7 - \lceil \frac{14-7}{\frac{0.984}{0.016}} \rceil = 6$. In this case it is possible to save $N \times 0.984 \times 6$ cycles which can be used to compensate the $N \times 0.016 \times (14-7)$ cycles (with N the number of iterations). As it is possible to schedule branch a within 6 cycles, this solution has been chosen.

5.3.2 Memory size consideration

In the previous example (Alg.5), the CDFG input declarations produce storage resources for the initial image: `Char[256x256]`, the labeled image: `CharLabel[256x256]` and for the pixel/label translation array: `TabEqui[256]`. Then the memory estimation indicates a potential second level of hierarchy including two arrays of 256 octets due to `Char[XYoffset-1] and Char[XYoffset - Nc]` and `CharLabel[XYoffset-1] and Charlabel[XYoffset - Nc]` respectively. This memory hierarchy information is locally stored as CDFG attributes and can be accessed by the designer.

5.4 CDFG Combination

In order to illustrate how the combination of CDFGs works, we use the example of the *gravity test* function, Fig. 14, from the object motion estimation application. The resulting trade-off curves are presented on Figs.15-20. The estimation process follows the bottom-up approach described in section 4.4. Firstly, leaf DFGs (DFG2, DFG3 and DFG core4) are estimated. Then CDFGs IF3, FOR4, FOR3, ELSE1, FOR3, DFG1, FOR2, FOR1 and IF1 are estimated through the combination rules described in 4.4. Finally the whole HCDFG is estimated. In this example the estimator has found that CDFG ELSE1 can reuse all the resources of CDFG IF1, therefore the HCDFG estimation is equal to the estimation of graph FOR1.

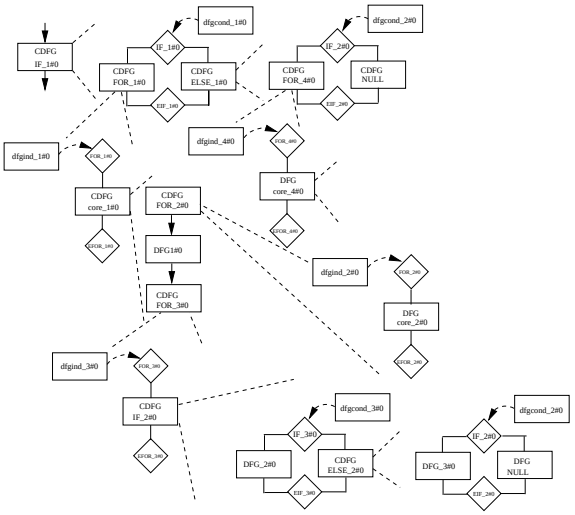


Figure 14: HCDFG of the *gravity test* function from the object motion estimation application

Analysis of the trade-off curves In this paragraph we show how the information produced by the DesignTrotter tool can be used by the designer in order to build or select an architecture. If we refer to the trade-off curves (Fig.15 to Fig.20) produced for the *gravity test* function we observe that several architectural solutions are conceivable. We can distinguish three mains options: A) if the function has to be executed as fast as possible the solution (noted (1) on the trade-off curves) must be used. In this case the function is executed in 2589 cycles and requires 8 simultaneous global memory accesses, 2 multipliers and 2 ALUs. When such memory bandwidth and processing parallelism requirements must be met a dedicated architecture such as a FPGA or ASIC can be used. As data are 8 bits wide, a possible implementation would require two 32 bits data buses and two address buses. An alternative would be to use

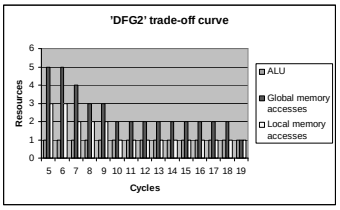


Figure 15: DFG_2 trade-off curve

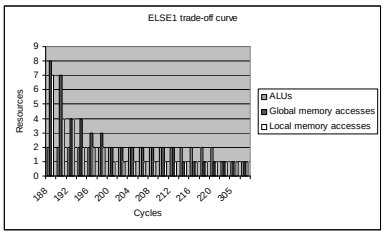


Figure 17: $ELSE_1$ trade-off curve

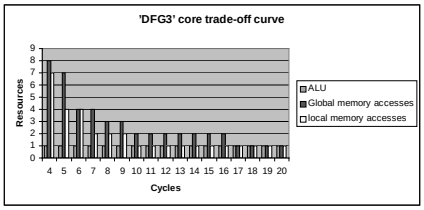


Figure 16: DFG_3 trade-off curve

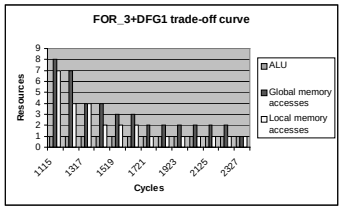


Figure 18: FOR_3+DFG_1 trade-off curve

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

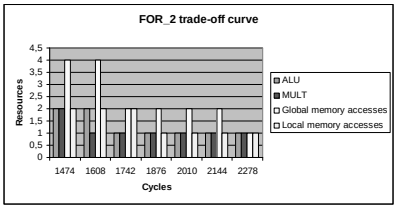


Figure 19: FOR_2 trade-off curve

a single 64 bits data bus and a single address bus. B) if the target architecture has to be relatively simple (e.g., because of cost) the designer must select a point on the trade-off curve where the number of required resources is as low as possible.

For example point (3) implies the use of unitary quantities of resources (1 global memory access, 1 multiplier and one ALU). If the time required to execute the function (4605 cycles or about 50% slower than with the dedicated architecture solution) is sufficient to respect the time constraint, then the designer can select this point. In that case, a simple RISC processor like the Arm9 could do the job.

C) finally if the time constraint is located between these two extremes, then a DSP based solution could be used. For example point (2) requires 2 simultaneous global memory accesses but only 1 multiplier and 1 ALU to execute the function in 2958 cycles. By looking at the result provided by Design-Trotter, a system designer can rapidly see that a powerful DSP like the TMS320C6201 that uses a complex architecture with a deep pipeline (up to 11 stages), VLIW instructions and features parallelism possibilities (up to 8 parallel instructions) would be under-exploited. Instead, the designer should choose a simpler model like the ADSP 21161N which has the right architectural features for

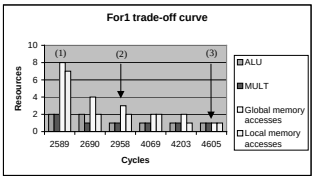


Figure 20: FOR_1 trade-off curve (= CDFG $IF_1 \sharp 0$ = **whole HCDFG)**

This is the authors' version of the paper that has been published in the
Journal of Embedded Computing, Volume 1, Number 4/2005, Copyright IOS Press 2005

implementing that specific solution.

6 Conclusions and perspectives

In this paper we have presented the *Design-Trotter* framework and more specifically the step dedicated to system-level design-space exploration for the implementation of embedded applications. This step is located before the target architecture selection and the fine architectural estimation [21]. The major contributions of this work are the HCDFG-based representation, the abstract architecture specification, the characterization of the application and the fast parallelism exploration with trade-off curves. The results produced by the Design-Trotter framework provide very useful information to the designer. Firstly, the characterization step indicates the available average parallelism and both processing and data-transfer orientation of the application functions. This orientation information has two purposes: 1) it can be used by the designer to select a type of architecture (GPP, DSP, ASIC...) and 2) it is used to guide the function estimation step. Secondly, the function estimation step exhibits and explores the potential parallelism of the functions for several time constraints. Our method permits the selection of the most appropriate scheduling algorithm (processing first, data-transfer first or mixed). This selection takes into account the most critical aspects of a function. The estimation process starts with the parallelism exploration of DFGs (equivalent to C basic blocs). Then higher granularity levels of parallelism are explored hierarchically, with a bottom-up approach. The parallelism vs. cycle budget tradeoffs are computed while favoring the reuse of resources by taking into account parameters such as loop unrolling and mutual exclusions due to conditional statements. The resulting delay/resources trade-off curves also indicate the parallelism lower and upper bounds of the application. Thus, the designer can choose the size or can scale its architecture according to these parallelism bounds. The estimates produced by the Design-Trotter tool provide relevant information very rapidly, which help the designer to take decisions very early in the design process in order to build or select the system architecture. The applicability of the method has been demonstrated in several examples, which illustrate how the designer can explore the design-space by referring to the trade-off curves.

The Design-Trotter framework includes other features which are not detailed in this paper. Firstly, the hardware and software projections, [21], [16] enable the refinement the estimation on more detailed architectures. Secondly, an extra type of architecture is taken into account by means of exploration of reconfigurable SoCs as explained in [12]. Moreover, to estimate a whole application (like the object motion detection) the designer can use two strategies: 1) the application must be specified as a single global function. In this case the application is estimated through the combination rules presented in 4.4 and results are available for each sub-function; or 2) all the functions of the application must first be specified individually and estimated within the function step, then HW/SW partitioning steps of Design-Trotter [1] can estimate the whole application. Strategy 1) is to be used when a single target is considered and strategy 2) is to be used when the functions are clearly designed for a multi-device target. The contribution of the inter-function step is then the modeling of communications between the devices.

Our approach is complementary to existing HLS tools. It is a two steps process: 1) parallelism exploration and 2) HW/SW projections which provide a fast design-space exploration presenting a good trade-off between speed and accuracy for the exploration process. It can be used to guide HLS tools which will generate the final HDL and/or software codes. Actually the accuracy levels obtained for the hardware projection step are about 10% and 18% for temporal and area aspects respectively [21].

The main perspective regarding Design-Trotter is an engineering effort located at the interface between the design space exploration and high level synthesis and/or compiler tools for (re)configurable processors. The objective is to automatically produce the SystemC / C code with associates compilation/synthesis scripts compliant with the designer choices, after the design space exploration step with DT. This point addresses the urgent need of the embedded systems industry for a substantial improvement of design productivity by providing efficient and automated design space exploration methods and tools that are linked to the existing implementation techniques based on manual and tedious iterative procedures of code development and results analysis. Such a complete flow is able to achieve unified, integrated and optimized embedded system design flows that start from a high level specification, and which are enable to estimate, analyze and optimize the performances of an implementation and provide the capabilities to explore various architectural choices in the entire design space.

References

[1] A.Azzedine, J-Ph.Diguet, and J-L.Philippe. Large exploration for hw/sw partitioning of multirate and aperiodic real-time systems. In *International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, USA, May 2002.

[2] A.D.Pimentel. The artemis workbench for system-level performance evaluation of embedded systems. *Int. Journal of Embedded Systems, Vol. 1 (No. 7)*, 2005.

[3] A.L.Sangiovanni-Vincentelli, L.P.Carloni, F.De Bernardinis, and M.Sgroi. Benefits and challenges of platform-based design. In ACM/IEEE *Design Automation Conference(DAC)*, San Diego, USA, June 2004.

[4] D-J.Wang and Y.H.HU. Rate optimal scheduling of recursive DSP algorithms by unfolding. *TCS*, 41:672–675, october 1994.

[5] H.-J. Stolberg et. al. An soc with two multimedia dsps and a risc core for video compression applications. *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, 2004.

[6] F.Balasa, F.Catthoor, and H.De Man. Background memory area estimation for multidimensional signal processing systems. IEEE *Transaction on* VLSI *Systems*, 3(2), June 1995.

[7] F.Charot and V.Messé. A flexible code generation framework for the design of application specific programmable processors. In *International Symposium on Hardware/Software Codesign (CODES)*, Roma, Italy, 1999.

[8] IMEC. Data transfer and storage exploration webpage. http://www.imec.be/design/dtse/, 2005.

[9] IRISA. Polylib. http://www.irisa.fr/polylib.

[10] J.Plantin and E.Stoy. Aspects on system-level design. In *International Symposium on Hardware/Software Codesign (CODES)*, Rome,Italy, April 1999.

[11] K.Kuchcinski and C.Wolinski. Global approach to scheduling complex behaviors based on hierarchical conditional dependency graphs and constraint programming. *Journal of Systems Architecture, Elsevier Science, Volume 49, Issues 12-15*, 2003.

[12] L.Bossuet, W.Burleson, G.Gogniat, V.Anand, A.Laffely, and J.L.Philippe. Targeting tiled architectures in design exploration. In *10th Reconfigurable Architectures Workshop (RAW)*, Nice, France, April 2003.

[13] L.Letellier and E.Duchesne. Motion detection algorithms. Technical report, L.C.E.I, C.E.A, Saclay, France, 2001.

[14] M.Auguin, K.Ben Chehida, J-Ph.Diguet, X.Fornari, A-M.Fouilliart, C.Gamrat, G.Gogniat, P.Kajfasz, and Y.Le Moullec. Partitioning and CoDesign tools & methodology for Reconfigurable Computing: the EPICURE philosophy. In 3^{rd} *Int. Work. on Systems, Architectures, Modeling Simulation (SAMOS03)*, Samos, Greece, July 2003.

[15] Y.Le Moullec, J-Ph.Diguet, N.Ben Amor, T.Gourdeaux, and J-L.Philippe. Algorithmic-level specification and characterization of embedded multimedia applications with design trotter. *To appear in the Journal of VLSI Signal Processing, Springer, 2005*, 2005.

[16] Y.Le Moullec, J-Ph.Diguet, and P.Koch. A power aware system-level design space exploration framework. In *DDECS'02*, Brno, Czech Republic, April 2002.

[17] P.Grun, F.Balasa, and N.D.Dutt. Memory size estimation for multimedia applications. In *International Symposium on Hardware/Software Codesign (CODES)*, Seattle, USA, March 1998.

[18] R.Goering. Systemc, tlm tools missing as esl interest grows. EE Times, June 2005.

[19] R.L.Lysecky and F.Vahid. A study of the speedups and competitiveness of fpga soft processor cores using dynamic hardware/software partitioning. In *Design Automation and Test in Europe Conference (DATE)*, Munich, Germany, March 2005.

[20] S.A.Blythe and R.A.Walker. Efficient optimal design space characterization methodologies. ACM *Transaction on Design Automation of Electronic Systems*, 5(3), July 2000.

[21] S.Bilavarn, G.Gogniat, J.L Philippe, and L.Bossuet. Low complexity design space exploration from early specifications. *to appear in IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 2005.

[22] S.Borkar. A vlsi system perspective for microprocessors beyond 90 nm. In *ISQED'03 Keynote Speeches*, Monterey, USA, March 2003.

[23] S.Wuytack, J-Ph.Diguet, F.Catthoor, and H.De man. Formalized methodology for data reuse exploration for low-power hierarchical memory

mappings. IEEE *Transaction on* VLSI *Systems*, 6(4):529–537, December 1998.

[24] T.Givargis, F.Vahid, and J. Henkel. System-level exploration for pareto-optimal configurations in parameterized system-on-a-chip. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI), vol. 10, no. 4*, 2002.