Mapping of an MPEG-4 Shape-Texture Decoder onto an On-chip Multiprocessor

Peter Poplavko[1,3], Milan Pastrnak[1,2 *], Twan Basten[1], Jef van Meerbergen[1,3], Peter H.N. de With[1,2]
[1] Eindhoven University of Technology
P.O.Box 513, 5600 MB Eindhoven, The Netherlands
[2] LogicaCMG Eindhoven / [3] Philips Research Labs Eindhoven
{P.Poplavko, M.Pastrnak}@tue.nl

Abstract — **This paper presents a mapping approach targeting a distributed-memory on-chip multiprocessor platform. A differentiating factor of our approach is that we strive to ensure predictable performance. The approach is biased towards video-decoding application domain, and we a use the MPEG-4 shape-texture decoder job of [3] as a case study. We map that job to three processors and obtain the *job worst-case execution time* (JWCET) for predictable performance. However, highly dynamic data-dependent behavior of this benchmark can lead to huge overestimations of JWCET even when employing the approach of [1], which uses run-time knowledge of input data parameters. We improve this approach, by introducing *scenarios*, similar to [4]. For some reference input data stream, the use of scenarios has reduced JWCET overestimation from 177% to 55%.**

Keywords— **performance evaluation; network-on-chip; data flow graph; job scheduling**

I. INTRODUCTION

In order to run real-time multimedia applications on an on-chip multiprocessor system, techniques have to be developed for the *mapping* of those applications on the multiprocessor platform, e.g. [4]. Furthermore, multimedia applications should yield high visual/audio quality at low cost. The authors believe that, in order to meet the real-time constraints at low cost, predictable hardware and predictable behavior of applications are necessary. Therefore, predictability is a major requirement in our research.

To tackle the interactivity and dynamism of applications, a dynamic real-time scheduler is required for coordinate multiple *jobs* on the array of processors. Informally, a job is a task that has to produce known amounts of output data samples at each activation. It is important to mention that a job can run on multiple processors and that it has a deadline. The multi-job scheduler must ensure that the schedule meets all the deadlines if possible.

To reduce the complexity of mapping multiple jobs on the multiprocessor, the mapping problem is split into two levels [1]. At Level I, *intra-job mapping* is performed, producing a configuration and an execution-time estimation for each job separately and independently. At Level II, the dynamic real-time *multi-job scheduler* coordinates the job set, sticking to a limited set of choices produced at Level I.

In this paper, we study the intra-job mapping of an MPEG-based multimedia application. More particularly, we concentrate on the decoding of MPEG-4 non-rectangular shaped video objects. Arbitrary shapes can lead to dynamic aspects in the mapping of such applications on a chosen platform. The problem to be solved is to distribute the decoding job efficiently over a set of processors and predict its total execution time. A differentiating factor of our approach is that we ensure predictable performance of every job, despite the data-dependent dynamic execution times, and despite a pipelined implementation of the job's main loop, spread over multiple processors.

In previous work [1], a very similar intra-job mapping approach has been applied for a JPEG decoder job. In this paper, we use an MPEG-4 shape-texture decoder job of [3], which possesses a higher degree of dynamism due to data dependency and hence constitutes a greater challenge for the prediction of the execution time.

This paper is organized as follows. Section II gives background information about the target platform and the application model. In Section III, we present our intra-job mapping approach, leaving some issues still open, because this approach is under development. We show how this approach can be applied for our MPEG-4

*Supported by the European Union via the Marie Curie Fellowship program under the project number HPMI-CT-2001-00150.

case study. As will be shown, the method of [1] for dealing with data dependency can yield a huge overestimation of the total job execution time. In Section IV, we extend this method by *scenarios*, similar to [4], based on some application-specific knowledge. For that purpose, we briefly consider the basics of the shape-texture decoding algorithm. Section V concludes this paper and mentions future work.

II. BACKGROUND

A. Target platform

We target a distributed-memory multiprocessor system-on-chip with message-passing communication through an on-chip network. The main parts of the system are the *array* of processing tiles and the network. Different processing tiles, particularly suited for different application-specific subtasks, may have different *types*. For each type, a *sub-array* of the tiles of that type is defined. The sub-arrays may overlap.

The core of each tile is its processor, being a master of the local memory system of the tile. Further, the tile contains a communication assist, the second master device that loads/stores multiple concurrent streams of messages going to/coming from different point-to-point network connections, that are set up and maintained for communication between the given tile and remote tiles.

There are a few assumptions we make for the sake of predictable hardware behavior:

1. Processors do not use caches when real-time jobs are running. Accesses to remote memory controllers may be performed via explicit message passing through the communication assist. Previous work, e.g., [10], indicates that *a-priori* knowledge of access patterns of the application can be used to insert explicit copying of data from the remote memories to the local memory.
2. Local memory arbitration between the processor and the communication assist features tightly bounded access delays for both.
3. Network communication delays are tightly bounded.

These assumptions are essential, because they allow to offer predictable worst-case performance in a real-time system, instead of optimized average performance that general-purpose systems offer. More considerations on the target platform can be found in [1] and [3].

In the current case study, we assume RISC processor tiles (using ARM7) with a flat memory model with a single-cycle memory access delay. As the interconnection network, we further assume that the Æthereal network-on-chip is employed [5]. It is built from multiple network routers, and it features as a reconfigurable set of point-to-point connections C_k with guaranteed bandwidth B_k. The implementation uses a kind of TDMA scheme, and layout has been synthesized for a network router [6].

Note that, for simplicity, in the sequel, we often refer to the processing tiles as ‘processors’.

B. Jobs and resource estimation

In our application model, the application dynamically activates a sequence of application *jobs*, e.g., for decoding a sequence of frames for a video object on the display. Each job can use multiple processing tiles in parallel. Multiple jobs can run concurrently.

The resource estimation is a front-end of multiprocessors mapping [7]. It involves estimation of computation resources, namely, the processing time and the size of the local memories that hold the instructions and the local data. In this paper, we take into account only the estimation of *execution time*, leaving the local instruction and data memory management for future work. Of course, limited memory sizes can result in some extra processing load (to copy the data from/to remote memories), but we believe that we can extend our approach later to include that load into our models as well.

With the video-decoding application domain in mind, we see a job as a ‘for’ loop, taking *J iterations* to produce *J* samples of output data, where *J* is known at the arrival time of the job. In video decoding, *J* can be determined by the number of macroblocks to be decoded in a single video frame for a single video object. In the sequel, we index the loop iterations with variable *j*, where $j = 0 \,..\, J-1$.

The body of the ‘for’ loop can be described by a set of subtasks, called *actors*, and the dependencies between them. We denote the set of actors as $V=\{v_i\}$. An actor can be described as a function in a programming language like C/C++, representing an atomic unit of computation that can be assigned to a processing tile.

As in [3], within a certain degree of error acceptance, we assume that the execution time of an actor v_i can be expressed as a linear function t_i. The function t_i has a set of arguments, called *input data parameters*, with fixed coefficients that depend on the architecture of the tile to which the subtask is assigned and the implementation details of the algorithm:

$$t_i(p_{1,i}, p_{2,i}, \ldots)[j] = c_{0,i} + c_{1,i} \cdot p_{1,i}[j] + c_{2,i} \cdot p_{2,i}[j] + \ldots \quad (2.1).$$

The square brackets denote that both the individual

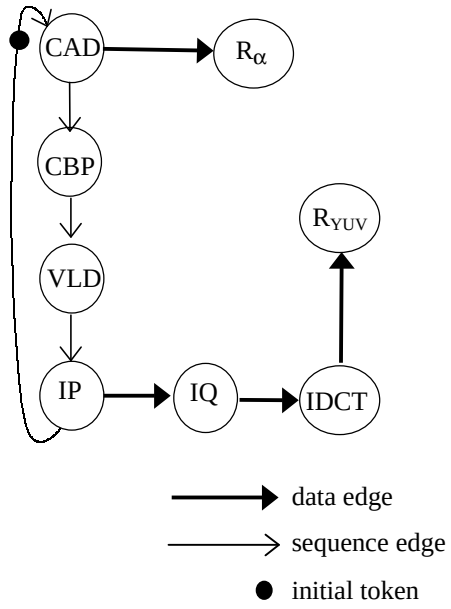

Figure 1: Computation graph for MPEG-4 intra-frame shape-texture decoding.

timing functions and the parameters $p_{k,i}[j]$ may change in each iteration *j*, because the input data changes at each iteration.

Within the job loop body, the dependencies between the actors (and hence the data parallelism) can be expressed using a data flow graph model of computation. We use a particular class of dataflow models, called *homogeneous synchronous data flow graphs* (HSDF) , introduced by Dennis [8].

We use HSDF graphs within our mapping trajectory in two places. First, they are used as a specification of job computations, in which case we refer to them as *computation graphs*. Second, we use them as so-called *inter-processor communication* (IPC) graphs to model a particular configuration of the job, which includes not only computation, but communication as well. We discuss IPC graphs in the next subsection.

Figure 1 shows the computation graph of an MPEG-4 shape-texture decoding job, introduced in [3] (It is an intra-frame decoder, which means that it does not decode any motion information, but only shape and texture [2]). The nodes of the computation graph are actors. The edges of the graph represent the dependencies between actor executions. There are *sequence* and *data edges*. In a computation graph, a sequence edge represents the fact that actors share common local state, and the destination actor may not start until the source actor has updated its local state and finished. A data edge represents the passing of data chunks, or *data tokens*. The size of these data chunks in bytes, or *token size*, may also be dynamic and specified by a linear function on input data parameters. Edges may also carry *initial tokens* that are available at time zero.

Each actor can start execution only when at least one token is available at each incoming edge. It consumes one input token per incoming edge, and, after a time delay equal to actor's execution time (as approximated by (2.1)), it produces one output token on each outgoing edge.

One iteration of the job loop involves one execution of all actors. Iterations of the HSDF graph may overlap in time. We assume that after *J* iterations, the actors are blocked and the graph eventually comes into its initial state.

For almost all computation actors in Figure 1, actor-timing models have been obtained in [3], and we use these models here. In experiments [3] the average error of these models was below 10%. We have refined the computation graph by introducing data reading actors, R_α and R_{YUV}, that copy the output data to some memory storage. The actor acronyms (like CAD, CBP, etc.) are explained in Appendix I. The actor timing models and token sizes of the data edges are all collected in Appendix II.

III. MAPPING APPROACH

The purpose of this section is to introduce our mapping trajectory, leaving the multi-job scheduler mostly outside the scope of this paper. We demonstrate mapping of the MPEG-4 shape-texture decoding job of Figure 1. In one of the mapping steps the job-level timing analysis introduced in [1] is applied. We show that, when applied directly, this timing analysis yields non-satisfactory results. We deal with this problem in the next section.

A. Problem description

To reduce the complexity of the dynamic multi-job scheduling problem, the mapping of the computation graphs of each job can be done without reconsidering the mapping of other jobs, thus focusing only on the internals of each separate job. We call this kind of mapping the *intra-job mapping*, and the resulting set of mapping decisions is called job *configuration*.

Many existing techniques for mapping HSDF graphs to multiprocessors assume static execution times of the actors. These techniques can be re-applied also when the execution times change dynamically if the dynamic execution times are approximated with worst-case static times and the approximation remains valid and tight

enough for all *J* iterations of the job. Unfortunately, this doesn't hold for applications like shape-texture decoding. In the next section we generalize this method of static approximation by splitting the set of job iterations into multiple scenarios. In this section, we still assume that static worst-case actor execution time values are used for all *J* iterations, and we obtain them by using a worst-case parameter set in equation (2.1).

We assume that for every actor only one processor type is chosen. The actor can be mapped to any processor in the sub-array of processors corresponding to that type. Instead of mapping to a particular processor in the sub-array, the mapping is done to a *virtual processor*, whose position in the sub-array is not known. The data communication is mapped to virtual connections, defined only in terms of bandwidth and propagation delay. Virtual connections and processors together constitute a virtual platform. The mapping to the real physical platform is done at run-time by the multi-job scheduler.

A good explanation of the required mapping decisions that fits our purposes can be found in [7], where the mapping is organized as a design flow consisting of several steps.

We will refer to our case study as an example to illustrate the mapping steps. In this paper, due to current limitations of our experimental set-up, we assume that the token sizes are fixed to the worst-case values, both in the implementation and the timing models. The execution time of the data-copying actors, namely, R_α and R_{YUV}, are fixed accordingly (see Appendix II).

B. Mapping steps

Step 1. Processor Assignment

In [7], this step involves a *partitioning* of the HSDF between processors and placement of the partitions on the physical processors. In the case of mapping to a virtual platform, only partitioning needs to be done, and the placement is done by the multi-job scheduler later. The partitioning has two objectives. First, *load balancing* should be done, so that the sum of actor execution times per virtual processor, or processing loads, are balanced. A balanced load is more likely to result in an optimal performance. The second objective is *minimization of communication*. Every data edge connecting different processors contributes to communication. So, it can be favorable to "hide" the data edges inside the partitions.

An extra constraint to the processor assignment is that the computation actors joined by sequence edges should be assigned to the same processor, because they share common local state.

It is due to that constraint that we assign in our example actors CAD, CBP, VLD and IP to a single processor, named Proc1. Because CAD actor is very computationally expensive in the worst case, this assignment already puts most of the load on Proc1.

Further, let us assume that only a certain processor type can access the off-chip memory. We assign R_α and R_{YUV} to a separate virtual processor Proc2 of that type and do not put anymore load on it to save it for other jobs wanting to access the off-chip memory. Finally, to offload Proc1, we assign the other actors, i.e., IQ and IDCT to Proc3, hiding the data edge between them.

Step 2. Routing

In [7], this step implicitly includes *bandwidth allocation* for communication and physical network routing. In a virtual platform, the topology of physical network and positions of the routing sources and sinks are not known yet, so only the *bandwidth allocation* is done for the data edges crossing the partition boundaries (see the previous step), and the routing is done later by the multi-job scheduler.

In our example, we have three data edges that cross the partition boundaries, namely: (CAD, R_α), (IP, IQ), and (IDCT, R_{YUV}). So, three independent virtual connections are required; we call them C_α, C_{DCT}, and C_{YUV}. The bandwidth allocated to these connections should be high enough to keep the transfer delays of the connections low enough so that communication will not become a performance bottleneck [1].

An upper bound on the transfer delay on a given connection depends on the data token size of the edge. For the TDMA scheme of the Æthereal network, the transfer delay t_T in clock cycles is given by:

$$t_T(token\_size) = \left\lceil \frac{token\_size}{B \cdot link\_rate} \right\rceil \cdot S \qquad (3.1)$$

where *B* is an integer bandwidth allocated for the connection, with $B < S$; *S* and *link_rate* are fixed integer parameters of the router (we assume $S = 256$, which is the size of the TDMA frame in time slots, and *link_rate* = 6 bytes per clock cycle). The worst-case token sizes are given in Appendix II. In our case study, we assume the smallest bandwidth assignment $B = 1$ to all three connections. Applying equation (3.1), we obtain the worst-case delays for three connections defined above as follows: $t_{T\,\alpha}$ = 2.8K cycles; $t_{T\ DCT} = t_{T\ YUV}$ = 33K cycles. These numbers are used in the IPC graph later on.

Step 3. Scheduling (Actor Ordering)

This step decides for each virtual processor the order in which actors assigned to it execute within the software loop running on that processor. To avoid deadlock, the ordering should be consistent with the dependencies between actors.

For virtual processor Proc1, the order has been already defined by the sequence edges of the computation graph (see Figure 1). The sequence edge holding the initial token precedes the first actor in that order (CAD). Similarly for Proc3, the data precedence requires that IQ must be executed first and then IDCT can start. For Proc2, we intuitively see that actor R_α will always receive input data earlier than R_{YUV}, so we put R_α as the first.

Step 4. Buffer Sizing

Every connection uses two communication FIFO buffers, namely, the input buffer at the processing tile that sends the data and the output buffer at the tile that receives the data. The buffer size is the number of tokens that fit in the buffer. Note that buffer storage is allocated for the worst-case sizes of the data tokens.

Within our approach, some ideas on finding optimal buffer sizes have been presented in [1]. In our example, we assume by default that the input and output buffer size is 1 token.

Step 5. Construction of IPC Graph

At this point, the configuration of the job on the virtual platform is complete. For the subsequent timing analysis step, it has been proposed in [1] to use IPC Graphs as timing models of the job configurations. A basic assumption taken in [1], and also in this paper, is that all virtual processors assigned to the job are fully available to that job during the execution time (no processor sharing). This is a valid assumption if so-called *gang scheduling* is used at the of multi-job scheduling level [9]. However, we will reconsider this assumption in our future work to incorporate processor sharing into the timing models.

Figure 2 shows the IPC graph for the configuration generated in Steps 1-4. All actors of Figure 1 can be found back here. Each set of actors belonging to the same processor, is joined by a sequence edge cycle, putting the actors in the order chosen in Step 3. Figure 2 explicitly indicates a three-actor cycle for Proc1, a single-actor cycle for Proc2 and a two-actor cycle for Proc3.

We decided to merge some actors together, because this way they can reuse the same space and data in the

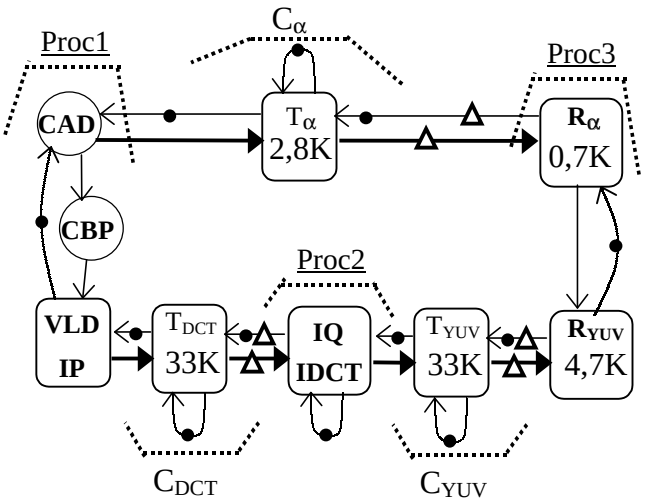

Figure 2: IPC graph of MPEG-4 shape-texture decoder.

communication memory (see [1] for a detailed explanation of modeling communication buffers). For the actors which have static execution times both in the model and in the implementation (due to fixed token sizes), Figure 2 shows execution time annotations (in clock cycles).

The figure indicates explicitly the parts of the graph corresponding to virtual connections. Each virtual connection is modeled by a transfer actor, that transfers the data token input buffer into the network and has delay given by equation (3.1). Each transfer actor is involved in three cycles. Let us take connection C_α as an example. One-actor cycle $(T_\alpha)^*$ models the fact that transfers of different tokens cannot overlap in time. Cycles $(T_\alpha,CAD)^*$ and $(T_\alpha, \Delta,R_\alpha, \Delta)^*$ model the input and the output FIFO buffer correspondingly. They both contain two edges, one of them modeling data, and the other one modeling the space in the FIFO and carrying the number of initial tokens equal to the size of the buffer. Both edges of the output buffer are marked with the sign Δ to denote that they pass the tokens with some small delay, modeling the propagation delay of the network connection. See [1] for information on the modeling of virtual connections in general.

Step 6. Timing analysis

To make the scheduling decisions, the multi-job scheduler may use an estimation of the workload of the job in terms of its worst-case execution time (JWCET). This estimation is performed by the timing analysis step based on the IPC graph.

For static actor execution times, one may expect a linear growth of JWCET with the number of iterations *J*. For any IPC graphs with some general properties, in [1]

it has been shown that if J is large enough, the linear upper bound of the execution time indeed exists, and can be defined by:

$$\text{JWCET}(J) = p \cdot (J-1) + \sigma, \qquad (3.2)$$

where p and σ are parameters that are functions of the static actor execution times. For p this function can be computed in polynomial time on the number of actors, for σ a heuristic can be used [1].

In our case study, we have derived parameters p and σ from an IPC graphs with worst-case actor execution times obtained from (2.1) by using the worst-case values of every parameter in one of the benchmark input bit-streams for MPEG-4 decoder. The resulting JWCET value, which we call *parametrical JWCET* [1], is roughly 36M cycles.

We have also obtained estimation of the execution time of the job in the real implementation, by simulating the execution of the IPC graph with dynamic actor execution times using the actual values of parameters in every iteration The real execution time estimation yielded roughly 13M cycles. We can observe 177% overestimation of the real execution time by the parametrical JWCET in this case. In the next section, we refine this method, in order to obtain a better estimation.

IV. Scenarios for the Shape-Texture Decoder

Due to the variation of the input data parameters and, as a consequence, a wide range of actor execution times in different iterations of our MPEG-4 decoding job benchmark, we have observed a huge overestimation of worst-case execution time. So, some extra application-specific knowledge should be applied to improve that result. We start this section with a brief excursion to the algorithm of the part of MPEG-4 standard that has actually been implemented in our case study. Then, based on this knowledge, we improve the execution time prediction by introducing a scenario approach, similar to the one introduced by [4] for another mapping approach.

A. Relevant application-specific information

MPEG-4 uses the concept of video objects (VOs) to enable interactive multimedia applications. Every VO is represented by a sequence of different video frames at different points in time. These frames are called Video Object Planes (VOPs). This video object plane is rectangular, and it is composed from smaller units called macroblocks (MB). An MB is a 16x16 pixel block that is the smallest data unit of a video object.

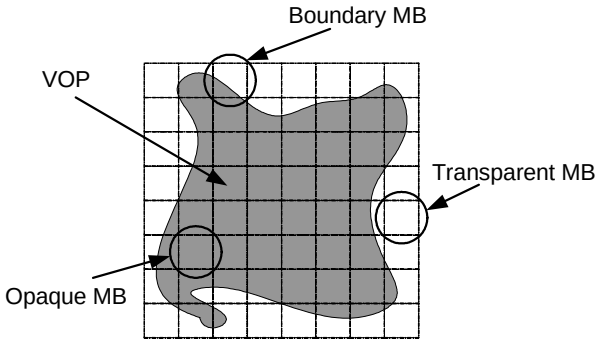

Figure 3: Arbitrarily shaped VOP representation.

Figure 3 depicts a typical composition of MBs within a VOP, representing an arbitrarily shaped video object. As is visible from Figure 3, even an arbitrarily shaped (sometimes referred to as non-rectangular) VO is covered with a grid of macroblocks. The standard distinguishes for intra-coded VO three types of macroblocks: boundary, opaque, and transparent. The boundary macroblock is defined as a macroblock that contains both transparent pixels and opaque pixels (see Figure 3).

Our application job case study of Figures 1 and 2 actually represents a decoder of an intra-frame VOP. Such a VOP contains only video texture and shape information. In one iteration of the job, one macroblock is decoded, so, J is actually the total number of macroblocks in the VOP.

Transparent MBs contain just a few bits of information used to detect them. They require very little processing. Opaque MBs require only the texture information to be decoded. Boundary MBs contain both texture and shape, and require the most computation [3]. For that reason, the required amount of computations can change drastically from one iteration of the job to another.

Note that the texture is represented in the YUV color plane. The shape is represented as a so-called "binary α-plane". This, by the way, explains some underscore notations we have used for the actors and connections of the job.

B. Scenario approach

Here we define a *scenario* as a set of (or a logical predicate on) input data parameter values, within which the IPC graph actor execution times remain in a certain range. Here we focus in particular on two parameters: μ and φ_Σ (see Appendix II). Their values determine the macroblock type as shown in Table 1.

Table 1: Timing model parameters and semantics.

μ	φ_Σ	MB type
0	0	Transparent
0	1	Opaque
1	1	Boundary

We introduce three correspondent scenarios: the 'transparent', the 'opaque' and the 'boundary'. For every iteration, it is true that one of these scenarios holds.

Scenario A is said to *cover* scenario B if scenario A assumes a worse values of the parameter set than scenario B. For example, 'boundary' scenario covers the other two scenarios.

We say that a continuous iteration interval j_1, j_1+1… j_2, where $0 \le j_1, j_2 \le J-1$ *can be covered* by scenario A, if it holds for all iterations in the interval or covers a scenario that holds there.

Obviously, the 'boundary' scenario can cover the complete iteration interval 0...J–1. Because our reference bitstream contains boundary blocks, it can be said that the parametrical worst-case approach, as defined in Section II-D, uses the 'boundary' scenario for all iterations, because it takes the worst-case values of every parameter in the interval 0...J–1.. By distinguishing the other two scenarios, we want to reduce the overestimation of execution time obtained there.

Figure 4 illustrates the method that we have applied. Figure 4(a) shows, for some job example, a horizontal axis of iterations and the scenario that holds for each iteration (assuming that the scenarios are numbered). Figure 4(b) shows with solid line the execution times of some IPC graph actor and with a dashed line the parametrical worst-case bound for that actor in that scenario. Figure 4(c) shows a set of intervals and scenarios that cover them. Note also that in this example, to reduce the number of intervals, it has been chosen in this case to merge three intervals of Figure 4(a) into one interval in Figure 4(c).

Given the sequence of intervals, the job execution can be illustrated with a corresponding sequence of so-called time shapes, as shown in Figure 4(d). Time shapes show which processors are busy at which moment of time. It can be seen that, because the execution of multiple iterations of the job can overlap in time, the time shapes also overlap.

We assume that the multi-job scheduler of the video decoder, when calculating the JWCET, can quickly

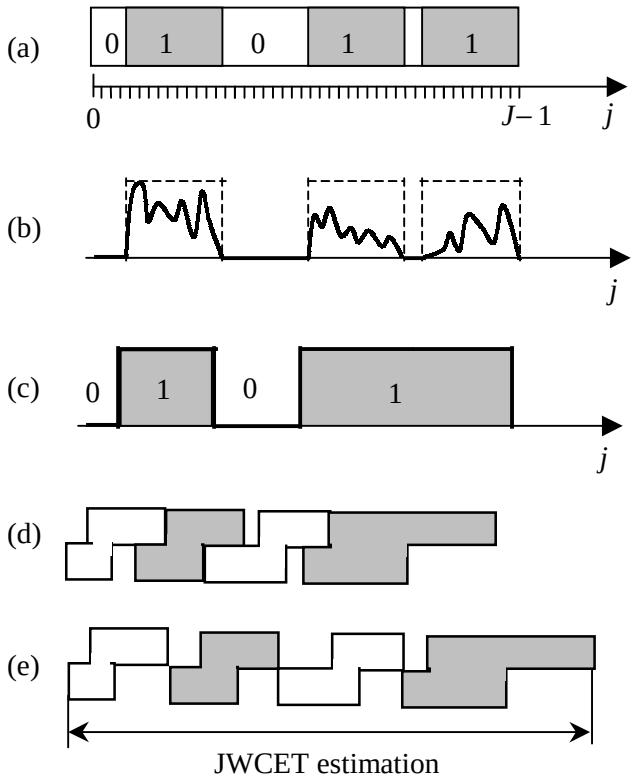

Figure 4: Obtaining JWCET estimation.

(a) scenarios; (b) actor execution time;
(c) intervals; (d) scenario time shapes;
(e) adding the time intervals of the shapes.

apply equation (3.2) to compute execution time of every time shape, but that it is computationally expensive to compute the overlapping of all shapes (see Figure 4(c)). The scheduler will rather add up their execution times, and obtain a pessimistic estimation of job WCET (see Figure 4(e)). By adding up JWCET expressions of (3.2) for all time shapes and grouping them by the scenario, we obtain the following expression:

$$\text{JWCET} = \sum_s \left(p_s \cdot J_s + L_s \cdot (\sigma_s - p_s) \right), \qquad (4.1)$$

where J_s is the total number of iterations in all intervals of scenario s; L_s is the total number of intervals of scenario s; p_s and σ_s are the corresponding characteristics of the IPC graph with static actor execution times obtained from the worst case set of parameters in all intervals of scenario s.

Based on the parameter values, the video encoder can detect a set of intervals and the scenarios that cover those intervals (see Figure 4(c)). To support execution time estimation, the encoder can provide the scenario information in an extension of the image header. It is enough to provide per each scenario the values J_{sc} and L_{sc}

and the set of worst-case values of all parameters in the scenario, which can be used to obtain p_s and σ_s. This way, the amount of information to be added in the header can be kept small.

We have applied this method to our reference bit-stream, using three scenarios defined above. We have detected and merged the scenario intervals manually. Our method estimated job WCET as roughly 20 million cycles, which is only 55% overestimation over 13 million cycles of the real execution time, which is a clear improvement compared to 177% of the estimation that does not distinguish between scenarios.

V. Conclusions and Future Work

To manage the resources in an interactive video decoding application, dynamic multi-job scheduling is needed, with every job possibly running on multiple processors. In this paper, we have studied a multiprocessor mapping that can be applied in such scheduling. The main contribution of our research is in detailed description of the mapping steps, applicable to highly dynamic systems. To demonstrate our concepts in practice, we have presented a case study of mapping of an MPEG-4 shape-texture decoding job of [3] to three processors of the platform. We have tuned the timing analysis part of our mapping approach to this application. This resulted in introducing 'scenario' method, similar to [4]. This method enabled to reduce the overestimation of the run-time worst-case execution time analysis from 177% to 55% when decoding an intra-frame of one of the test bitstreams.

In our future work, we will reuse this case study in the multi-job scheduler for profiles of MPEG-4 coding that allow manipulation of multiple arbitrarily shaped objects by managing the multiple virtual platforms of the corresponding jobs on a single physical platform. To achieve this, we extend our application with motion compensation.

Our approach supports only predictable multiprocessors, and, for that reason, we do not allow caching of instruction/data memory during the execution of the jobs. We will try to manage and model mapping of large local-state data structures to remote memories using the same kind of communication as for the data tokens.

Acknowledgement

The authors are grateful to Marco Bekooij, Philips Research Labs Eindhoven, for valuable feedback on some details concerning the mapping approach.

References

[1] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman, Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-Chip, to appear in *ACM Proceedings CASES'03*, Oct.30-Nov.2, 2003.

[2] N. Brady, MPEG-4 Standardized Methods for the Compression of Arbitrarily Shaped Video Objects, *IEEE Transactions Circuits and Systems for Video Technology*, vol. 9, no. 8, pp. 1170-1189, December 1999.

[3] M. Pastrnak, P. Poplavko, P.H.N. de With, J. van Meerbergen, On Resource Estimation of MPEG-4 Video Decoding for A Multiprocessor Architecture, *Proceedings of PROGRESS 2003*, http://www.stw.nl

[4] P. Yang, *et al*, Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems, *ACM Proceedings ISSS'02*, pp. 112-119, 2002.

[5] K. Goossens, *et al*., Guaranteeing the Quality of Services in Networks on Chip, in *Networks on Chip*, ed. A. Jantsch and H. Tenhunen, Kluwer Academic Publishers, pp. 61-82, 2003.

[6] E. Rijpkema, K.G.W. Goossens, and A. Radulescu, Trade Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip. *ACM Proceedings of DATE'03*, pp. 350-355, 2003.

[7] R. Lauwereins, M. Engels, M. Ade, J.A. Peperstraete, Grape-II: A System-Level Prototyping Environment for DSP Applications, *IEEE Computer*, vol. 28, no. 2, 35-43, February 1995.

[8] J. B. Dennis. First Version of a Data-Flow Procedure Language, *Proceedings of the Colloque sur la Programmation*, vol. 19 of *Lecture Notes in Computer Science*, pp. 362-376. Springer-Verlag, 1975.

[9] D. G. Feitelson, L. Rudolph, Gang Scheduling Performance Benefits for Fine-Grain Synchronization, *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 306-318, 1992.

[10] F. Catthoor, K. Danckaert, S. Wuytack, N.D. Dutt, Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors, *IEEE Design & Test of Computers*, vol. 18, no. 3, pp. 70 –82, May-June 2001.

Appendix I Actor Acronyms

CAD – context-based arithmetic decoder; this actor decodes the binary shape of a boundary macroblock;

CBP – coded block pattern; this actor detects which of the four luminance blocks of the macroblock are non-transparent and have coded AC coefficients;

VLD – variable length decoder; this actor decodes DCT coefficients of all non-transparent luminance and chrominance blocks of one macroblock;

IP – inverse prediction; updates the DCT coefficients based on the prediction information from already decoded neighbor macroblocks;

IQ – inverse quantization; multiplication of DCT coefficients by coefficients from a quantization matrix;

IDCT – inverse cosine transform; if macroblock is not transparent, translates it from DCT domain into the pixel color-space domain.

APPENDIX II TIMING MODELS AND TOKEN SIZES*

Timing Models

$t_{CAD} = 8.21K + 148.74K \cdot \mu + 240 \cdot N_{bits1} + 270 \cdot N_{bits2} + 210 \cdot N_{bits3}$.

$t_{CBP} = 15 + 966 \cdot \xi + 132 \cdot \omega + 13 \cdot N_{empty\text{-}pix}$

$t_{VLD} = 627 \cdot \varphi + 74 \cdot N_{VLD\text{-}bytes} + 107 \cdot N_{DC} + 87 \cdot N_{DC+} + 297 \cdot N_{AC\text{-}NE} + 621 \cdot N_{AC\text{-}E1} + 619 \cdot N_{AC\text{-}E2} + 921 \cdot N_{AC\text{-}E3}$

$t_{IP} = 285 \cdot \varphi + 746 \cdot \gamma$

$t_{IQ} = 1.39K \cdot \varphi + 44 \cdot N_{AC}$

$t_{IDCT} = 78 + 41.31K \cdot \varphi_{\Sigma}$.

$t_R = 328 + 5.75 \cdot token\_size$ (applies to R_α and R_{YUV})

Token Sizes

$token\_size_\alpha = 64bytes \cdot \mu$; the worst case is 64 bytes.

$token\_size_{DCT} = 4bytes + 128bytes \cdot \varphi$; the worst case is 772 bytes

$token\_size_{YUV} = 1byte + 768bytes \cdot \varphi_{\Sigma}$; the worst case is 769 bytes

Table 2: Timing model parameters and semantics.

μ	1, if the macroblock is of type 'boundary' 0, otherwise.
$N_{bits1\text{-}3}$	Number of bits of type 1,2,3 (as defined in the standard) used to code shape
ξ	1, if the macroblock is not transparent; 0, otherwise.
ω	1, if ξ=1 and at least one 8x8 block in the BAB is completely transparent; 0, otherwise.
$N_{empty\text{-}pix}$	Total number of first empty pixels in all 8x8 BAB sub-blocks (if pixels are checked in the scanning order), hence $N_{empty\text{-}pix} < 16 \times 16$.
φ	Number of non-transparent sub-blocks in the macroblock, hence $\varphi \le 6$.
$N_{VLD\text{-}bytes}$	Number of the bit stream bytes shifted into the 64-bit local buffer, while reading VLD coded bits; here we cannot give a bound.
N_{DC}	Number of the non-zero DC DCT coefficients in the macroblock, thus $N_{DC} \le 6$.
N_{DC+}	Number of the non-zero DC DCT coefficients coded by more than 1 bit, $N_{DC+} \le N_{DC}$.
$N_{AC\text{-}NE}$	Number of non-zero AC coefficients coded by a 'normal' VLC code.
$N_{AC\text{-}E1\text{-}3}$	Number of AC coefficients coded by an ESC code, using types 1, 2, or 3.
γ	1, if the macroblock uses AC prediction 0, otherwise.
N_{AC}	Number of non-zero AC coefficients in the macroblock, $N_{AC} = N_{AC\text{-}NE} + N_{AC\text{-}E1} + N_{AC\text{-}E2} + N_{AC\text{-}E3}$, $N_{AC} \le 6 \times 63$.
φ_{Σ}	1, if $\varphi > 0$, 0, otherwise.

* Most of information here is taken from [3]