Chapter 9
Design Space Exploration for Run-Time Management of a Reconfigurable System for Video Streaming

Giovanni Mariani, Chantal Ykman-Couvreur, Prabhat Avasare, Geert Vanmeerbeeck, Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria

Abstract This Chapter reports a case study of Design Space Exploration for supporting *Run-time Resource Management* (RRM). In particular the management of system resources for an MPSoC dedicated to multiple MPEG4 encoding is addressed in the context of an *Automotive Cognitive Safety System* (ACSS). The run-time management problem is defined as the minimization of the platform power consumption under resource and *Quality of Service* (QoS) constraints.

The Chapter provides an insight of both, design-time and run-time aspects of the problem. During the prelimiary design-time Design Space Exploration (DSE) phase, the best configurations of run-time tunable parameters are statically identified for providing the best trade-offs in terms of run-time costs and application QoS. To speed up the optimization process without reducing the quality of final results, a multi-simulator framework is used for modeling platform performance.

At run-time, the RRM exploits the design-time DSE results for deciding an operating configuration to be loaded for each MPEG4 encoder. This operation is carried out dynamically, by following the QoS requirements of the specific use-case.

9.1 Introduction

The amount of resources available on MPSoC platforms is growing following the Moore's law. To fully exploit the potential capabilities of future MPSoC, programmers need to split applications in multiple threads in such a way that these can be allocated to different processors to be executed concurrently. There is not a unique way to parallelize an application but different parallel versions of the same application can be analyzed and used to trade off the application performance and the number of threads in the application. If every thread needs a dedicated processor to be executed, the availability of different parallelized versions can provide also

G. Mariani (✉)
ALaRI, University of Lugano, Switzerland
e-mail: giovanni.mariani@usi.ch

C. Silvano (eds.), *Multi-objective Design Space Exploration of Multiprocessor SoC Architectures*,
DOI 10.1007/978-1-4419-8837-9_9,

a trade off between resources needed to execute the application and the application performance.

In this context, a *Run-time Resource Management layer* (RRM), part of the Operating System (OS), is expected to dynamically decide which application version has to be executed for each active application. In particular the design of future MPSoC should consider a complex multi-programmed scenario where many applications, each one composed of multiple parallel threads, are competing for accessing the shared computing resources.

The RRM problem becomes more complex as the number of run-time tunable parameters to be controlled by the OS increases; e.g., when operating frequencies of processors executing different threads have to be set together with the application parallelization. The RRM should set at run-time, for each active application, an operating configuration to dynamically match the user requirements while minimizing non-functional costs as energy or power consumption of the MPSoC platform.

Run-time decision making involves the solution of a combinatorial problem to fit, with the lowest cost and highest QoS, a set of predetermined resources. In fact, deciding which operating configuration has to be set for each application can be modeled as a Multi-dimension Multiple-choice Knapsack Problem (MMKP) [9] and belongs to the *NP-hard* class of problems [3].

Performing an exhaustive exploration of the possible candidate solutions at run-time would be too slow and even a trial-and-error procedure, where the RRM physically tries on the platform different operating configurations and then takes decisions based on observed system behaviors, would be unacceptable. To cope with this problem, in Chap. 5 we introduced a lightweight RRM exploiting design-time knowledge for efficiently selecting at run-time a reasonable assignment of the run-time tunable parameters while, in Chap. 6, we presented system-wide OS level methodologies that can be applied when detailed design-time information are not available.

The present Chapter demonstrates the effectiveness of the approach proposed in Chap. 5 for an industrial scenario. In this context, we target both the number of allocated cores and their operating frequencies as run-time configurable parameters of an application. Our case study concerns the automotive field and the RRM is responsible for allocating system resources to multiple instances of MPEG4 encoder where each instance has a different QoS requirement in terms of frame rate. The partitioning of system resources is here obtained by loading different parallel versions of the MPEG4 encoder to the different application instances. Each MPEG4 parallel version is composed of a different amount of threads and thus it needs for a different number of cores to be executed.

This Chapter shows both design-time and run-time sides of the methodology. During the design-time DSE phase, a multi-level simulation framework is used to search for the Pareto optimal operating configurations within the space identified by the run-time tunable parameters. Within the multi-level simulation framework a detailed but slow cycle accurate simulator and an approximate but fast High Level Simulator (HLSim) are available. This allows the quick exploration on HLSim of a large amount of the design space while the accuracy of the final solution is obtained

via validation on the cycle accurate simulator. This allows to speed up the exploration without reducing the quality of the final results.

The Chapter is organized as follows. In Sect. 9.2 we present the use case scenario introducing the MPEG4 application and its different parallel versions. Then, Sect. 9.3 gives an insight of the target MPSoC platform and the simulators used during the design-time DSE. Section 9.4 reports the results obtained from the proposed methodology during the design-time DSE phase first (Sect. 9.4.1) and, second, from the RRM during the run-time execution (Sect. 9.4.2). The Chapter finally concludes in Sect. 9.5 summarizing the most relevant results.

9.2 Case Study

The case study presented in this Chapter concerns the management of system resources for a multiple-stream MPEG4 encoding chip dedicated to *automotive cognitive safety* tasks. In particular we are targeting an MPEG4 encoding for a 4CIF video resolution. The application code is specifically optimized for compilation on the target MPSoC system in which the main computational element is the coarse-grain reconfigurable ADRES processor [5].

9.2.1 Application Description

The MPEG4 encoder is an industry-standard, block-based, hybrid video encoder. The structure of the MPEG4 encoder is shown in Fig. 9.1 and details can be found in [7]. Mainly the MPEG4 encoder is composed of the following functional blocks:

- **Motion Estimation (ME)** compares the current frame with a reference frame previously processed in order to estimate the motion within the frames.
- **Motion Compensation (MC)** compensates the estimated motion in the goal of increasing the efficiency of the compression.

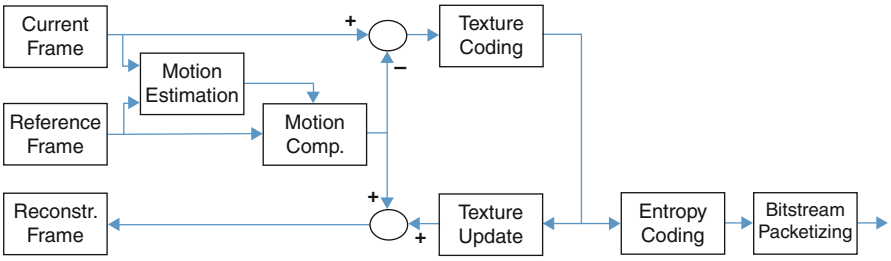

Fig. 9.1 Overview of the MPEG4 encoder application

- **Texture Coding (TC)** performs discrete cosine transform and quantization over motion compensated residuals ('texture').
- **Texture Update (TU)** uses the output of TC in order to locally reconstruct the original frame as it would appear after the decoding phase. This reconstructed frame can be useful later on as reference frame.
- **Entropy Coding (EC)** encodes the motion vectors to produce a compressed bitstream.
- **Bitstream Packetizing (BP)** prepares the packets containing the output data.

These functional blocks are implemented in application kernels (computational intensive nested loops) which have been optimized for compilation on VLIW architectures, in particular for the execution on an ADRES processor [5] used in our MPSoC platform.

The RRM can generate a trade off between application performance and resource usage by selecting a specific parallelization to be executed on the platform. To do so, the RRM needs different parallel versions of the same application, i.e., different binaries which perform the same functionalities while using a different amount of computing elements.

A set of parallel versions has been generated starting from a sequential implementation based on the MPEG4 Simple Profile reference code. First of all, the initial sequential version has been pruned and cleaned to set-up the parallelization procedure. Then, the sequential application is parallelized using *MPSoC Parallelization Assist* (MPA) tool [6].

MPA is a tool which supports MPSoC programmers on investigating different parallelization alternatives for a given application. Once the MPSoC programmer specifies a parallelization for the application, MPA is able to automatically insert into the sequential code all program lines needed to spawn parallel threads and to implement inter-thread communication. For generating different versions of the same application, the programmer should profile the sequential application, understand how kernels can be assigned to different threads and specify different parallelization opportunities to MPA, without any further handmade modification to the application code.

MPA is able to handle parallelizations either at *functional* or at *data* level (a combination of both functional and data parallelism is also handled). In practice, different functional kernels can be organized over different threads (functional parallelization) or the same kernel(s) can be divided over different threads by dividing them w.r.t. loop indices (data parallelization). In the second case, each thread performs the same functionalities over a different part of the dataset.

Once different parallel versions are generated with MPA, the obtained codes can be compiled and generated binaries can be executed on the target platform. In particular, Fig. 9.2 shows the parallel versions of the MPEG4 encoder studied during this Chapter. Within Fig. 9.2, the functional blocks are reported in solid boxes while the thread partitioning is represented by dotted lines.

In this case study, every thread needs a computing element to be executed and a computing element cannot execute more than one thread. Thus, the number of threads

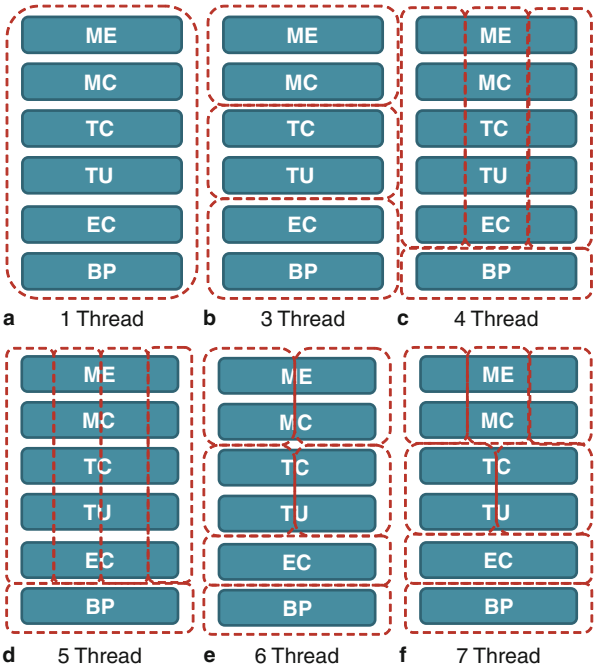

Fig. 9.2 MPEG4 encoder versions parallelized over a given number of threads (up to 7)

in an application version is equivalent to the resource cost ρ of the specific version. The six available MPEG4 encoder versions are parallelized for the execution on 1, 3, 4, 5, 6 and 7 processors. Since every version is parallelized over a different number of threads, in the following the resource cost ρ is also used to uniquely identify the MPEG4 encoder version.

9.2.2 *Automotive Cognitive Safety System*

The use case studied in this Chapter is an *Automotive Cognitive Safety System* (ACSS). In this context, a vehicle is equipped with a wide range of digital sensors (such as cameras and radars) and it is able to identify emergency conditions. The final goal of the ACSS is to keep passengers safer assisting the driver during emergency situations. In particular, the ACSS is able to identify and actuating emergency measures for the following scenarios:

- Forward collision warning.
- Automatic pre-crash emergency braking.
- Lane departure warning and guidance.
- Lane change and blind spot assist.

The vehicle is also provided with three digital cameras associated with left, right and central mirrors respectively, as shown in Fig. 9.3.

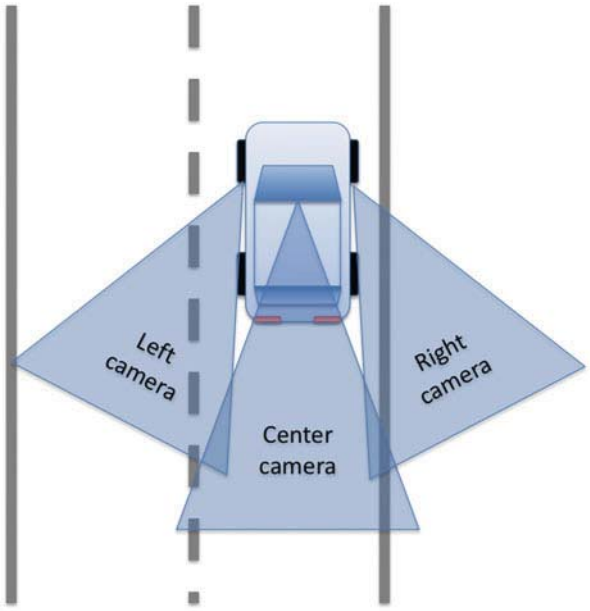

Fig. 9.3 Mirror views of the target vehicle

These cameras are connected to the target ADRES-based MPSoC platform which performs the encoding of the three video streams by means of the MPEG4 encoder presented in Sect. 9.2.1. For this purpose, an instance of the MPEG4 encoder is executed for each of the three views. The encoded streams are then sent to an off-chip Central Safety Unit (CSU), reducing the needs of on-board bus bandwidth.

Moreover, the dashboard has a set of displays that can be used to reproduce the actual content of the video streams sent to the CSU. Thus, the driver can watch live-views from the mirror cameras. The driver can independently switch on and off each of the camera views via the steering wheel interface.

We assume that the minimum requirement needed to let the CSU operating correctly is 15 frame per second (*fps*) per each video stream. Since this requirement is too low for providing the driver with a good video quality, the frame rate might be increased when the mirror views are displayed on the dashboard. In fact, when live-views are enabled, the CSU communicates new frame rate requirements to the MPSoC platform performing the video encoding.

In particular, when some live-views are activated, the requirements are decided on the basis of the vehicle speed and on the proximity of other vehicles. The following criteria are used:

- Lateral cameras: 15 *fps* under 10 km/h, 25 *fps* for over 110 km/h, interpolated linearly for intermediate speed. 30 *fps* when a vehicle is in proximity.
- Center camera: 15 *fps* under 20 km/h, 20 *fps* over 120 km/h, interpolated linearly for intermediate speed.

The frame rate requirements are not continuously updated but new requirements are communicated only when the vehicle speed passes certain thresholds or when another vehicle enters/exits the proximity areas. There is a threshold every 10 km/h for the lateral cameras and every 20 km/h for the central camera.

We here recall from Chap. 5 that the RRM needs also application priorities to decide which requirement should be relaxed when would be otherwise impossible to solve the RRM problem matching all requirements given the limited amount of system resources. The CSU communicates to the MPSoC platform also the priorities of the multiple video streams together with the frame rate requirements. By default the highest priorities are given to the central and left cameras. When a vehicle is in proximity, then the lateral camera closer to the approaching vehicle and the central camera get the highest priorities.

9.3 Platform Description

The industrial architecture studied is the 8-cores MPSoC shown in Fig. 9.4. The platform is composed of seven ADRES (coarse-grain Architecture for Dynamically Reconfigurable Embedded System) cores [5] and one StrongARM. The ADRES core is a power-efficient flexible computing element which combines a coarse-grain array with a VLIW DSP. The MPEG4 application has been optimized for compilation for the ADRES core in such a way that the data-flow loops can be efficiently accelerated by exploiting the coarse-grain array for loop level parallelization. The potentialities of the ADRES cores are fully exploited by executing the MPEG4 encoder instances while the RRM is executed on the StrongARM processor. The on-chip interconnect consists of a 32-bit wide Network-on-Chip with two switches.

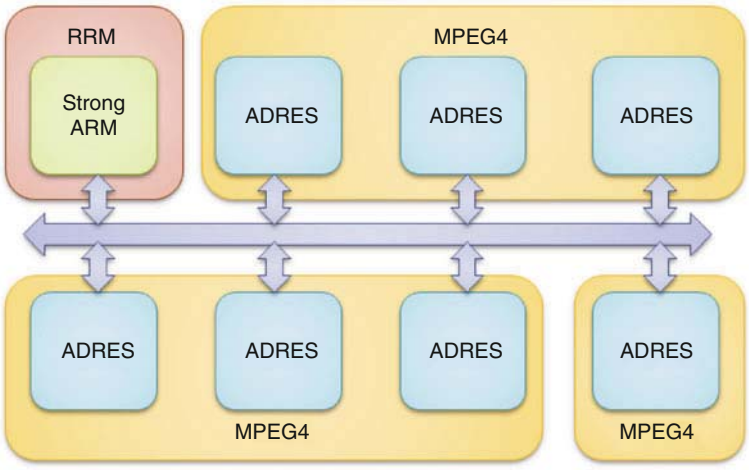

Fig. 9.4 Target MPSoC platform executing the RRM and the three MPEG4 encoder instances

The RRM can trade-off frame rate and power consumption of the multiple MPEG4 by acting on two parameters: the parallelizations of the MPEG4 instances and the frequencies of the ADRES cores. In this sense the operating frequency of each ADRES core can be dynamically modified independently by adopting *Dynamic Voltage and Frequency Scaling* (DVFS) techniques. In particular the frequency range for the ADRES cores is $\Phi = \{20, 60, 100, 140, 180, 220\}$ MHz, while the StrongARM is operating at 206 MHz.

9.3.1 Simulation Tool-Chain

For evaluating performance values of the MPSoC platform, a multi-simulator framework has been used [1]. More specifically, a *Transaction-Level Model simulator* (TLMsim) built with *CoWare platform* design tools enables the cycle-accurate analysis of the multi ADRES system and the underlying communication infrastructure.

This cycle-accurate simulator is too slow for enabling the analysis of many operating configurations for the MPEG4 encoder application. To cope with this problem a *High Level Simulator* (HLSim) has been used.

HLSim exploits back-annotated information of execution time derived from the cycle-accurate simulator. Execution time figures for all the kernels are recorded during cycle-accurate simulations and are fed back into HLSim as input while running the application.

In practice, during a first recording phase, the sequential version of the application is simulated on the cycle-accurate simulator; application kernels are profiled and performance indices (e.g. processor execution cycles) are saved into a database. Then, HLSim uses this profiled data library during its application simulation to derive accurate timing for application execution. HLSim does timed simulation of the application meaning that HLSim keeps track of local time in each thread and this time is appropriately adjusted during thread synchronizations. Performance indices are also scaled to the frequencies of the processors where kernels are executed. Given an application version and the frequency of each thread, the timing feedback mechanism let HLSim to provide a very fast but approximate evaluation of the platform performance indices.

Comparing HLSim with the cycle accurate TLMsim on simulating the MPEG4 encoding of 10 frames in 4CIF resolution, we obtain that HLSim has an execution time of 45 s with respect to TLMsim which requires 4 h. This simulation time saving is obtained at the cost of the simulation accuracy, in fact HLSim has a simulation error that is always lower than 20%.

The availability of HLSim enables the design-time DSE phase to quickly investigate many frequency combinations for each application version. The results obtained by HLSim for some simulations are then validated with the cycle accurate simulator. Cycle-accurate simulator can also be used for DSE phase but within a small focused interesting region obtained from HLSim-based DSE phase. This strategy of using two simulators enables application designers to evaluate DSE phase efficiently [1].

Finally, to complete the performance analysis of the simulation tool-chain, the performance figures of the StrongARM executing RRM are obtained using the *Simlt-ARM* simulator [8].

9.4 Experimental Results

This section presents the results obtained on applying the methodology proposed in Chap. 5 to the ACSS use case. In particular, we start presenting the results of the design-time DSE phase whose goal is the identification of the Pareto optimal operating points of the target application (Sect. 9.4.1). Then, behaviors of the proposed RRM on the specific case study are presented in Sect. 9.4.2.

9.4.1 Design Space Explorations Using Multi-Simulator Framework

In our use case, there are different versions of the MPEG4 encoder, each one identified by a specific resource cost ρ.

An operating configuration of the MPEG4 encoder is:

$$\boldsymbol{c} = \langle \rho, \phi, \pi, \tau \rangle \tag{9.1}$$

where ρ is the resource cost, ϕ is the frequency vector containing an operating frequency $\phi \in \Phi$ for each of the ρ threads, while π and τ are power and performance values obtained via simulation.

The design-time DSE phase has the goal of identifying the set of operating configurations $\boldsymbol{C}$ which are Pareto optimal considering resource cost, power consumption and average frame execution time (i.e., $\boldsymbol{c}[\rho]$, $\boldsymbol{c}[\pi]$ and $\boldsymbol{c}[\tau]$). During this design-time DSE phase, the frequency vector ϕ representing operating frequencies of platform cores is independently optimized for each MPEG4 version. In fact an independent optimization problem targeting minimization of power consumption $\boldsymbol{c}[\pi]$ and average frame execution time $\boldsymbol{c}[\tau]$ is solved for each version presented in Fig. 9.2. Results obtained from the different optimizations are then merged together and processed as explained in Chap. 5 to obtain the Pareto optimal operating points $\boldsymbol{C}$ and a user value $v(\boldsymbol{c})$ for each configurations $\boldsymbol{c} \in \boldsymbol{C}$.

The design space of each optimization problem grows exponentially with the number of threads ρ in the given application version. While the design space for the sequential version ($\rho = 1$) is composed of $|\Phi| = 6$ points only, the design space of the version parallelized over 7 threads is composed of $|\Phi|^7 = 279{,}936$ points. Given this difference in the design space dimensions, we use different algorithms to perform different optimizations (see Chap. 3). In particular all optimizations are performed with the Multicube Explorer tool [10]. Within this optimization environment we use a

Table 9.1 Design space size and simulations required for the optimization of each version of the MPEG4 encoder. In the last column, a summary of the overall design space and number of simulations needed for the optimization of all versions

ρ	1	3	4	5	6	7	Grand total
Design space	6	216	1,296	7,776	46,656	279,936	335,886
Simulations	6	216	309	567	694	1,077	2,869

full search optimization approach for the application versions where the design space is small enough (i.e., for $\rho \leq 3$); the Non-dominated Sorting Genetic Algorithm (NSGA-II) [2] is used when the design space grows too much (i.e., for $\rho > 3$). In particular the genetic algorithm is run with a population size $|\Phi| \times \rho$ for 50 generations.

Table 9.1 reports for each parallelized version of the application the design space size and the number of simulations needed to perform the optimization. Due to the huge number of simulations needed to perform the overall exploration (i.e., 2,869), only the *HLSim* is used within the optimization loop. Note that one simulation on HLSim takes around 45 s whereas the same simulation takes around 4 h on cycle-accurate simulator. Figure 9.5 shows the derived final Pareto configurations for the MPEG4 encoder encoding 10 frames at 4CIF resolution.

The *CoWare-based simulator* (TLMsim) is built to model a Transaction Level Model (TLM) of the platform at a *cycle-accurate level*. This simulator is used to validate the results obtained by HLSim and also this simulator is used to do a localized, focused explorations of the parameters not supported by HLSim. As explained

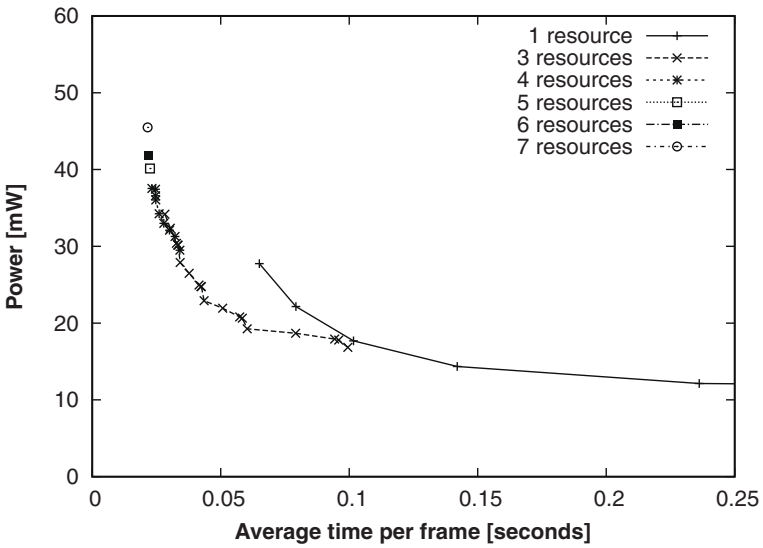

Fig. 9.5 Pareto optimal operating configurations for the MPEG4 encoder (encoding 10 frames at 4CIF resolution) by varying the number of resources

before, timing spent in different kernels during cycle-accurate simulations is profiled and fed back to HLSim. By this timing back-annotation, we remain accurate enough ($< 20\%$ error) between HLSim and cycle-accurate simulations w.r.t. execution time and power consumption. Further, Pareto operating points obtained by HLSim are also validated with cycle-accurate simulator. By using such a two-simulator approach, DSE phase can be done efficiently and accurately to reach optimum operating points in a short amount of time. Note that using such multi-level simulators to reduce number of simulations is orthogonal to sophisticated modeling techniques (e.g., Response Surface Modeling techniques presented in Chap. 4) supported by DSE tools [1].

To give an idea of the time saved by the proposed methodology we obtain that:

- The *exhaustive exploration* (i.e., the simulation of all 335886 configurations) with *TLMsim* would take 1,343,544 h; almost **153 years**.
- Performing the *NSGA-II optimizations* using *TLMsim* would require **16 months**.
- The *exhaustive exploration* with *HLSim* would take about **6 months**.
- Performing the *NSGA-II optimizations* using *HLSim* requires about **36 h**.

9.4.2 Run-Time Resource Management

9.4.2.1 Simulation of an Urban Environment

To generate dynamic QoS requirements for the RRM, the overall ACSS has been simulated for several reasonable driving patterns and conditions [4]. Although we simulated different patterns, for conciseness and clarity we will report results obtained for a specific pattern reproducing an urban scenario. The car speed and the frame rate requirements are shown in Fig. 9.6.

The vehicle, starting from a stationary position with speed equal to 0 km/h starts to accelerate a few before the 35th second. The vehicle reaches a maximum speed of 60 km/h around the 50th second and it keeps a constant speed for a while. Then, at about the 60th second, the vehicle starts decelerating to reach again a stationary condition at about the 70th second (Fig. 9.6a). All three live-views are active on the dashboard at the beginning of the simulation. The live-view of the central camera is deactivated from the driver at around 50 s and the frame rate of the corresponding video stream is reduced to 15 *fps* as required from the CSU (Fig. 9.6c). During the simulation two vehicles pass within the proximity area on the left side. The frame rate requirement of the corresponding camera increases to 30 *fps* (Fig. 9.6b).

9.4.2.2 Run-Time Resource Management Behaviors

Whenever the CSU sets new frame rate requirements, the RRM is invoked to globally select one operating configuration for each MPEG4 encoder with the goal of fitting

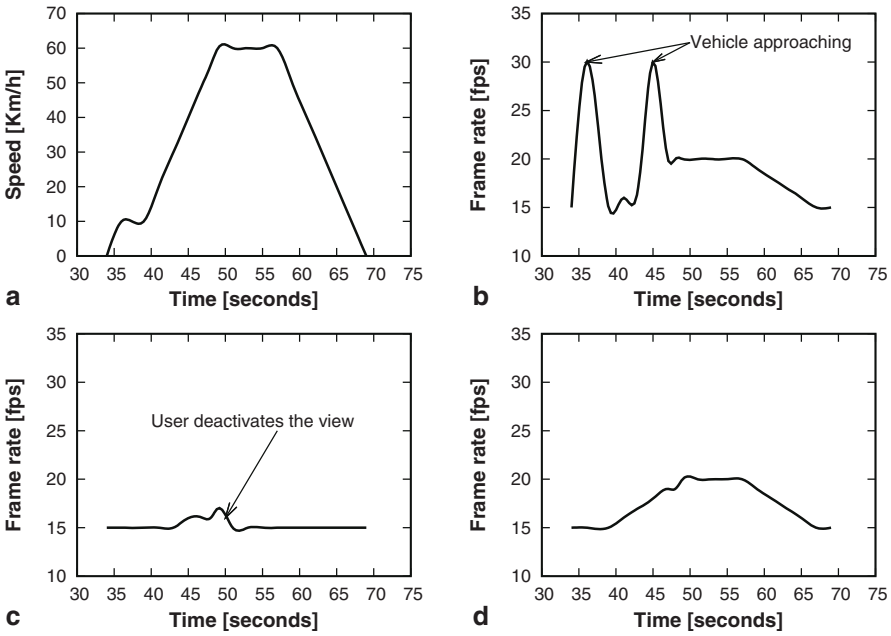

Fig. 9.6 Car speed and QoS for the urban case-study. **a** Car Speed, **b** Left camera, **c** Center camera, **d** Right camera

the QoS requirements while minimizing the power consumption without exceeding the amount of computational resources physically available on the MPSoC platform.

Figure 9.7 shows the selected MPEG4 encoder configurations in terms of number of allocated ADRESs and their average operating frequency set by the RRM for the three video streams given the driving pattern presented in Fig. 9.6.

Figure 9.8 reports also the power consumption of the overall MPSoC platform together with the penalties on the video stream requirements (i.e., the difference between the required frame rates and the achieved ones).

When the frame rate requirement is 15 *fps*, the RRM can set either a three-resources low frequency configuration or a one-resource high frequency configuration. In a normal stationary situation, the RRM provides one resource to the left camera and three resources to the central and right camera.

When a vehicle is in proximity of the left camera, the QoS requirement of this camera is of 30 *fps*. To fit in this requirement, the RRM has to set the left camera MPEG4 encoder in an operating configuration with many cores operating at high frequency. For these cases, required resources are taken from other video-streams which will move to a configuration with low resources and high frequency. In the overall system there will be many cores operating at high frequency and these situations are clearly characterized by peaks in the power consumption as visible in Fig. 9.8a.

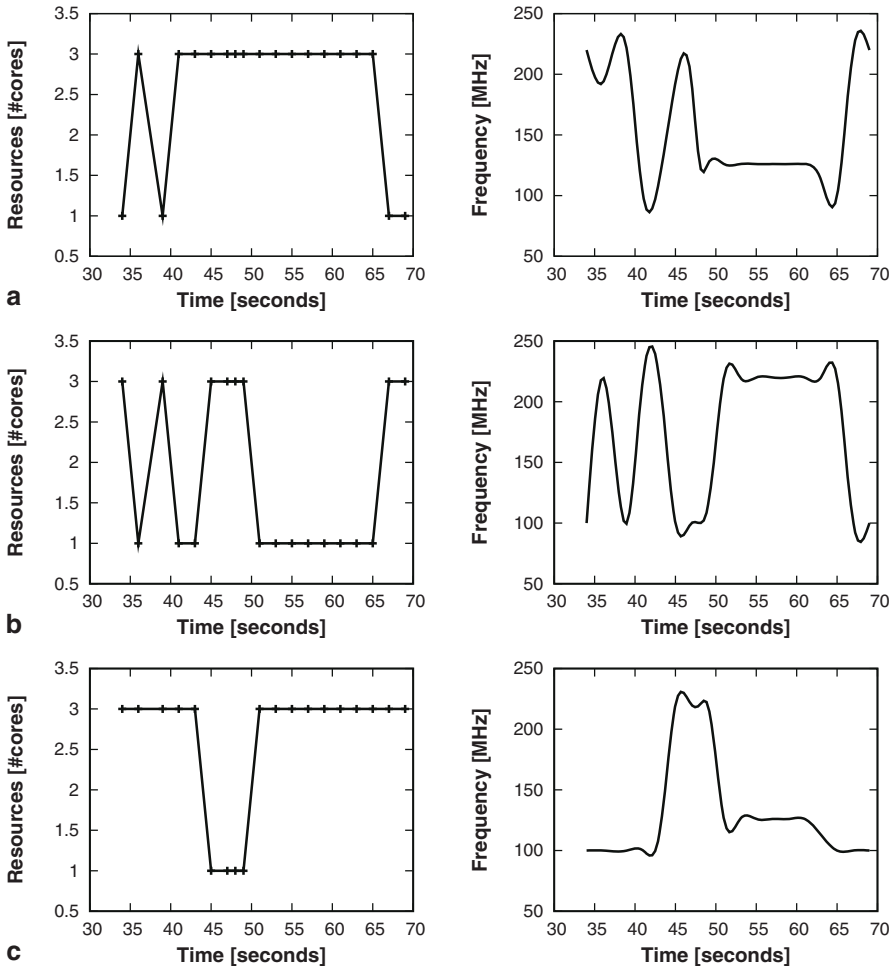

Fig. 9.7 Resource allocation and average operating frequency for the three MPEG4 instances. **a** Left camera, **b** Center camera, **c** Right camera

As the car speed increases, the frame rate requirements for the lateral cameras increase faster than the requirement for the central camera (Fig. 9.6). This brings around the 40th second to a situation where all requirements can still be met reducing the resources for the central camera while providing three cores to each lateral camera. This situation stands until the requirements cannot be met anymore. In this last situation, the requirement of the lowest priority application, i.e., the MPEG4 encoder of the right camera, is relaxed and only one resource is assigned to it. This situation is also visible in Fig. 9.8b, where the right camera frame rate is lower than the QoS requirement by around 4 *fps* (QoS penalty of -4 *fps*).

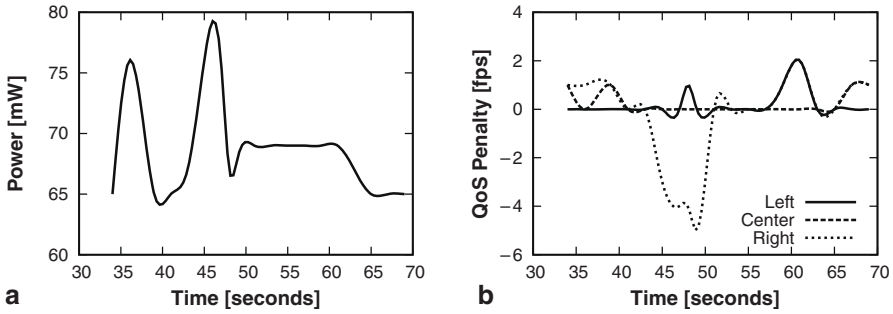

Fig. 9.8 System power profile and QoS penalty for the three MPEG4 instances. **a** Power, **b** QoS Penalty

During the simulations, the RRM routine (executed on StrongARM running at 206 MHz) for the operating point selection always found a solution within a computation time of 1 ms. This run-time overhead is enough small to be considered negligible in the given context.

To further clarify the advantage given by the availability of different parallelized versions of the same application, Fig. 9.9 shows a *bubble plot* representing the operating configurations effectively used by the RRM within the proposed urban scenario. A bubble plot is a way of representing the relationship between three or four variables on a scatter-plot. Observations on two variables are plotted in the usual way on the x and y axis using circles as symbols. The radii of the circles are made proportional to the associated values for the third variable while different gray levels are used to represent the fourth variable. In our specific case, the average operating frequency and the power consumption of the operating configurations are respectively reported on the x and y axis; frame rate supported from the configurations is proportional to the circles dimensions while the gray level represents the amount of ADRES cores needed to execute the configuration.

It is worth to note that among the configurations loaded by the RRM, only one uses the sequential version of the MPEG4 encoder (i.e., the version running on one

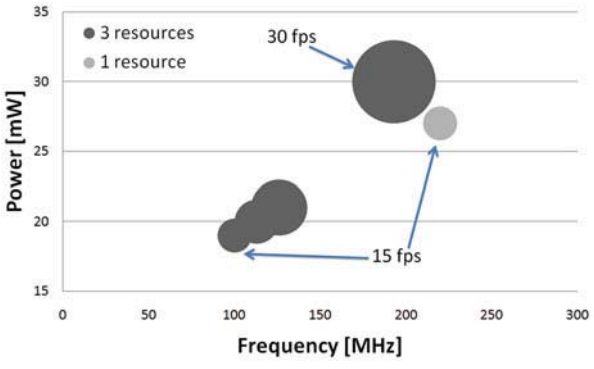

Fig. 9.9 Bubble plot of operating points loaded by the Run-time Resource Management

single ADRES core). This particular configuration executes the MPEG4 encoder at 220 MHz and provides a frame rate of 15 *fps*.

The MPSoC platform with seven ADRES cores is over-dimensioned for the minimal requirement of 15 *fps* for each video stream needed by the CSU to operate correctly. In fact for executing 3 MPEG4 encoders at 15 *fpss*, 3 ADRES cores would be enough. The availability of different parallel versions provide to the RRM the possibility to allocate the remaining resources to minimize the platform power consumption. In particular, the RRM under the minimal requirements steadily selects a configuration with 3 resources for the right and central camera. The RRM takes this decision since the MPEG4 encoder running on 3 cores can be executed at about 100 MHz, obtaining the same QoS and saving more than 5 mW of power consumption (per 10 frames).

We can also observe that whenever a solution which matches all QoS requirements exists, this is correctly identified by the priority-based RRM. In fact, from Figs. 9.8b and 9.6 it is possible to notice that there is a QoS penalty only when all cameras require more than 15 *fps*. In fact, providing more than 15 *fps* requires as least three ADRES cores. Thus, due to resource constraints, when all video streams require more than 15 *fps*, the frame rate of the lowest priority stream has to be relaxed.

9.5 Conclusions

In this Chapter we presented an application of the RRM to the management of resources dedicated to a multiple-stream MPEG4 encoder chip within a *Automotive Cognitive Safety System* scenario.

First of all the MPEG4 encoder application has been presented together with the specific use case requirements. The design space identified by the run-time tunable parameters has been considered to be too huge to be explored exhaustively during the design time DSE phase. This is the reason for performing a heuristic optimization within an integrated DSE framework which provides heuristic optimization algorithms and RSM techniques as the one described in Chaps. 3 and 4.

For enabling this design-time optimization phase, a multi-simulator framework was adopted. We showed that a *High Level Simulator* (HLSim) can be used to quickly evaluate performance indices of many candidate operating configurations. These obtained performance results are validated with cycle accurate simulations.

Once the Pareto optimal operating configurations are identified, the RRM previously proposed in Chap. 5 has been proven to be effective at identifying an operating configuration for each of the multiple video stream. In the experiments, the RRM always identified a configuration which is able to satisfy all constraints when this solution exists. When the QoS requirements are too high to be matched with the available resources, the priority based algorithm can still identify a solution to the problem by relaxing the constraint on the application having the lowest priority.

Note that the techniques for run-time resource management discussed in this Chapter are valid when ample design time information of all the running applications

is available. When such a design-time information of different applications is not available, system wide approaches for run-time management at the Operating System (OS) level can be applied. Such OS level approaches are discussed in Chap. 6.

References

1. Avasare, P., Vanmeerbeeck, G., Kavka, C., Mariani, G.: Practical approach for design space explorations using simulators at multiple abstraction levels. In: Design Automation Conference (DAC) Users' Track (2010)
2. Deb, K., Agrawal, S., Pratab, A., Meyarivan, T.: A Fast and Elitist Multi-Objective Genetic Algorithm: NSGA-II. Proceedings of the Parallel Problem Solving from Nature VI Conference pp. 849–858 (2000). URL citeseer.ist.psu.edu/article/deb00fast.html
3. Garey, M.R., Johnson, D.S.: Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences). W. H. Freeman (1979)
4. Mariani, G., Avasare, P., Vanmeerbeeck, G., Ykman-Couvreur, C., Palermo, G., Silvano, C., Zaccaria, V.: An industrial design space exploration framework for supporting run-time resource management on multi-core systems. In: DATE 2010 - International Conference on Design, Automation and Test in Europe., pp. 196 –201. Dresden, Germany (2010)
5. Mei, B., Sutter, B., Aa, T., Wouters, M., Kanstein, A., Dupont, S.: Implementation of a coarse-grained reconfigurable media processor for avc decoder. Journal of Signal Processing Systems **51**(3), 225–243 (2008). DOI http://dx.doi.org/10.1007/s11265-007-0152-8
6. Mignolet, J.Y., Baert, R., Ashby, T.J., Avasare, P., Jang, H.O., Son, J.C.: Mpa: Parallelizing an application onto a multicore platform made easy. IEEE Micro **29**, 31–39 (2009). DOI http://doi.ieeecomputersociety.org/10.1109/MM.2009.46
7. Richardson, I.: H. 264 and Mpeg-4 Video Compression: Video Coding for Next-generation Multimedia. John Wiley & Sons (2003)
8. Simlt-arm (2007). Http://simit-arm.sourceforge.net/
9. Ykman-Couvreur, C., Nollet, V., Catthoor, F., Corporaal, H.: Fast multi-dimension multi-choice knapsack heuristic for MP-SoC run-time management. In: Proceedings of International Symposium on System-on-Chip, pp. 1–4 (2006). DOI 10.1109/ISSOC.2006.321966
10. Zaccaria, V., Palermo, G., Mariani, G.: Multicube explorer (2008). Http://www.multicube.eu