Analysis, Design and Management of Multimedia Multiprocessor Systems.

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 28 april 2009 om 10.00 uur

door

Akash Kumar

geboren te Bijnor, India

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. H. Corporaal

Copromotoren:
dr.ir. B. Mesman
en
dr. Y. Ha

Analysis, Design and Management of Multimedia Multiprocessor Systems.
/ by Akash Kumar. - Eindhoven : Eindhoven University of Technology, 2009.
A catalogue record is available from the Eindhoven University of Technology Library
ISBN 978-90-386-1642-1
NUR 959
Trefw.: multiprogrammeren / elektronica ; ontwerpen / multiprocessoren /
ingebedde systemen.
Subject headings: data flow graphs / electronic design automation /
multiprocessing systems / embedded systems.

Analysis, Design and Management of Multimedia Multiprocessor Systems.

The work in this thesis is supported by STW (Stichting Technische Wetenschappen) within the PreMADoNA project.

Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 175.

Printing: Printservice Eindhoven University of Technology

Abstract

Analysis, Design and Management of Multimedia Multiprocessor Systems.

The design of multimedia platforms is becoming increasingly more complex. Modern multimedia systems need to support a large number of applications or functions in a single device. To achieve high performance in such systems, more and more processors are being integrated into a single chip to build Multi-Processor Systems-on-Chip (MPSoCs). The heterogeneity of such systems is also increasing with the use of specialized digital hardware, application domain processors and other IP (intellectual property) blocks on a single chip, since various standards and algorithms are to be supported. These embedded systems also need to meet timing and other *non-functional* constraints like low power and design area. Further, processors designed for multimedia applications (also known as *streaming processors*) often do not support preemption to keep costs low, making traditional analysis techniques unusable.

To achieve high performance in such systems, the limited computational resources must be shared. The concurrent execution of dynamic applications on shared resources causes interference. The fact that these applications do not always run concurrently only adds a new dimension to the design problem. We define each such combination of applications executing concurrently as a *use-case*. Currently, companies often spend 60-70% of the product development cost in verifying all feasible use-cases. Having an analysis technique can significantly reduce this development cost. Since applications are often added to the system at run-time (for example, a mobile-phone user may download a Java application at run-time), a complete analysis at design-time is also not feasible. Existing techniques are unable to handle this dynamism, and the only solution left to the designer is to over-dimension the hardware by a large factor leading to increased area, cost and power.

In this thesis, a run-time performance prediction methodology is presented that can accurately and quickly predict the performance of multiple applications before they execute in the system. Synchronous data flow (SDF) graphs are used to model applications, since they fit well with characteristics of multimedia applications, and at the same time allow analysis of application performance. Further, their atomic execution requirement matches well with the non-preemptive nature of many streaming processors. While a lot of techniques are available to analyze performance of single applications, for multiple applications this task is a lot harder and little work has been done in this direction. This thesis presents one of the first attempts to analyze performance of multiple applications executing on heterogeneous non-preemptive multiprocessor platforms.

Our technique uses performance expressions computed off-line from the application specifications. A run-time iterative probabilistic analysis is used to estimate the time spent by tasks during the contention phase, and thereby predict the performance of applications. An admission controller is presented using this analysis technique. The controller admits incoming applications only if their performance is expected to meet their desired requirements.

Further, we present a design-flow for designing systems with multiple applications. A hybrid approach is presented where the time-consuming application-specific computations are done at design-time, and in isolation with other applications, and the use-case-specific computations are performed at run-time. This allows easy addition of applications at run-time. A run-time mechanism is presented to manage resources in a system. This ensures that once an application is admitted in the system, it can meet its performance constraints. This mechanism enforces budgets and suspends applications if they achieve a higher performance than desired. A resource manager (RM) is presented to manage computation and communication resources, and to achieve the above goals of performance prediction, admission control and budget enforcement.

With high consumer demand the time-to-market has become significantly lower. To cope with the complexity in designing such systems, a largely automated design-flow is needed that can generate systems from a high-level architectural description such that they are not error-prone and consume less time. This thesis presents a highly automated flow – *MAMPS* (*Multi-Application Multi-Processor Synthesis*), that synthesizes multi-processor platforms for multiple applications specified in the form of SDF graph models.

Another key design automation challenge is fast exploration of software and hardware implementation alternatives with accurate performance evaluation, also known as design space exploration (DSE). This thesis presents a design methodology to generate multiprocessor systems in a systematic and fully automated way for *multiple use-cases*. Techniques are presented to merge multiple use-cases into one hardware design to minimize cost and design time, making it well-suited for fast DSE of MPSoC systems. Heuristics to partition use-cases are also presented such that each partition can fit in an FPGA, and all use-cases can be catered for. The above tools are made available on-line for use by the research community. The tools allow anyone to upload their application descriptions and generate the FPGA multi-processor platform in seconds.

Acknowledgments

I have always regarded the journey as being more important than the destination itself. While for PhD the destination is surely desired, the importance of the journey can not be underestimated. At the end of this long road, I would like to express my sincere gratitude to all those who supported me all through the last four years and made this journey enjoyable. Without their help and support, this thesis would not have reached its current form.

First of all I would like to thank Henk Corporaal, my promoter and supervisor all through the last four years. All through my research he has been very motivating. He constantly made me think of how I can improve my ideas and apply them in a more practical way. His eye for details helped me maintain a high quality of my research. Despite being a very busy person, he always ensured that we had enough time for regular discussions. Whenever I needed something done urgently, whether it was feedback on a draft or filling some form, he always gave it utmost priority. He often worked in holidays and weekends to give me feedback on my work in time.

I would especially like to thank Bart Mesman, in whom I have found both a mentor and a friend over the last four years. I think the most valuable ideas during the course of my Phd were generated during detailed discussions with him. In the beginning phase of my Phd, when I was still trying to understand the domain of my research, we would often meet daily and go on talking for 2-3 hours at a go pondering on the topic. He has been very supportive of my ideas and always pushed me to do better.

Further, I would like to thank Yajun Ha for supervising me not only during my stay in the National University of Singapore, but also during my stay at TUe. He gave me useful insight into research methodology, and critical comments on my publications throughout my PhD project. He also helped me a lot to arrange the administrative things at the NUS side, especially during the last phase of my PhD. I was very fortunate to have three supervisors who were all very hard working and motivating.

My thanks also extend to Jef van Meerbergen who offered me this PhD position as part of the PreMaDoNA project. I would like to thank all members of the PreMaDoNA project for the nice discussions and constructive feedback that I got from them.

The last few years I had the pleasure to work in the Electronic Systems group at TUe. I would like to thank all my group members, especially our group leader Ralph Otten, for making my stay memorable. I really enjoyed the friendly atmosphere and discussions that we had over the coffee breaks and lunches. In particular, I would like to thank Sander for providing all kinds of help from filling Dutch tax forms to installing printers in Ubuntu. I would also like to thank our secretaries Rian and Marja, who were always optimistic and maintained a friendly smile on their face.

I would like to thank my family and friends for their interest in my project and the much needed relaxation. I would especially like to thank my parents and sister without whom I would not have been able to achieve this result. My special thanks goes to Arijit who was a great friend and cooking companion during the first two years of my PhD. Last but not least, I would like to thank Maartje who I met during my PhD, and who is now my companion for this journey of life.

Akash Kumar

Contents

CHAPTER 1

Trends and Challenges in Multimedia Systems

Odyssey, released by Magnavox in 1972, was the world's first video game console [Ody72]. This supported a variety of games from tennis to baseball. Removable circuit cards consisting of a series of jumpers were used to interconnect different logic and signal generators to produce the desired game logic and screen output components respectively. It did not support sound, but it did come with translucent plastic overlays that one could put on the TV screen to generate colour images. This was what is called as the first generation video game console. Figure 1.1(a) shows a picture of this console, that sold about 330,000 units. Let us now forward to the present day, where the video game consoles have moved into the seventh generation. An example of one such console is the PlayStation3 from Sony [PS309] shown in Figure 1.1(b), that sold over 21 million units in the first two years of its launch. It not only supports sounds and colours, but is a complete media centre which can play photographs, video games, movies in high definitions in the most advanced formats, and has a large hard-disk to store games and movies. Further, it can connect to one's home network, and the entire world, both wireless and wired. Surely, we have come a long way in the development of multimedia systems.

A lot of progress has been made from both applications and system-design perspective. The designers have a lot more resources at their disposal – more transistors to play with, better and almost completely automated tools to place and route these transistors, and much more memory in the system. However, a number of key challenges remains. With increasing number of transistors has come increased power to worry about. While the tools for the back-end (synthesizing a

(a) Odyssey, released in 1972 – an example from first generation video game console [Ody72].

(b) Sony PlayStation3 released in 2006 – an example from the seventh generation video game console [PS309]

Figure 1.1: Comparison of world's first video console with one of the most modern consoles.

chip from the detailed system description) are almost completely automated, the front-end (developing a detailed specification of the system) of the design-process is still largely manual, leading to increased design time and error. While the cost of memory in the system has decreased a lot, its speed has little. Further, the demands from the application have increased even further. While the cost of transistors has declined, increased competition is forcing companies to cut cost, in turn forcing designers to use as few resources as necessary. Systems have evolving standards often requiring a complete re-design often late in the design-process. At the same time, the time-to-market is decreasing, making it even harder for the designer to meet the strict deadlines.

In this thesis, we present analysis, design and management techniques for multimedia multi-processor platforms. To cope with the complexity in designing such systems, a largely automated design-flow is needed that can generate systems from a high-level system description such that they are not error-prone and consume less time. This thesis presents a highly automated flow – *MAMPS* (Multi-Application Multi-Processor Synthesis), that synthesizes multi-processor platforms for not just multiple applications, but *multiple use-cases*. (A *use-case* is defined as a combination of applications that may be active concurrently.) One of the key design automation challenges that remain is fast exploration of software and hardware implementation alternatives with accurate performance evaluation. Techniques are presented to merge multiple use-cases into one hardware design to minimize cost and design time, making it well-suited for fast design space exploration in MPSoC systems.

In order to contain the design-cost it is important to have a system that is neither hugely over-dimensioned, nor too limited to support the modern applications. While there are techniques to estimate application performance, they often end-up providing a high-upper bound such that the hardware is grossly over-dimensioned. We present a performance prediction methodology that can accurately and quickly predict the performance of multiple applications before they execute in the system. The technique is fast enough to be used at run-time as well. This allows run-time addition of applications in the system. An admission controller is presented using the analysis technique that admits incoming applications only if their performance is expected to meet their desired requirements. Further, a mechanism is presented to manage resources in a system. This ensures that once an application is admitted in the system, it can meet its performance constraints. The entire set-up is integrated in the *MAMPS* flow and available on-line for the benefit of research community.

This chapter is organized as follows. In Section 1.1 we take a closer look at the trends in multimedia systems from the applications perspective. In Section 1.2 we look at the trends in multimedia system design. Section 1.3 summarizes the key challenges that remain to be solved as seen from the two trends. Section 1.4 explains the overall design flow that is used in this thesis. Section 1.5 lists the key contributions that have led to this thesis, and their organization in this thesis.

1.1 Trends in Multimedia Systems Applications

Multimedia systems are systems that use a combination of content forms like text, audio, video, pictures and animation to provide information or entertainment to the user. The video game console is just one example of the many multimedia systems that abound around us. Televisions, mobile phones, home theatre systems, mp3 players, laptops, personal digital assistants, are all examples of multimedia systems. Modern multimedia systems have changed the way in which users receive information and expect to be entertained. Users now expect information to be available instantly whether they are traveling in the airplane, or sitting in the comfort of their houses. In line with users' demand, a large number of multimedia products are available. To satisfy this huge demand, the semiconductor companies are busy releasing newer embedded, and multimedia systems in particular, every few months.

The number of features in a multimedia system is constantly increasing. For example, a mobile phone that was traditionally meant to support voice calls, now provides video-conferencing features and streaming of television programs using 3G networks [HM03]. An mp3 player, traditionally meant for simply playing music, now stores contacts and appointments, plays photos and video clips, and also doubles up as a video game. Some people refer to it as the convergence of information, communication and entertainment [BMS96]. Devices that were traditionally meant for only one of the three things, now support all of them. The devices have also shrunk, and they are often seen as fashion accessories. A mobile phone that was not very *mobile* until about 15 years ago, is now barely thick enough to support its own structure, and small enough to hide in the smallest of ladies-purses.

Further, many of these applications execute concurrently on the platform in different combinations. We define each such combination of simultaneously active applications as a **use-case**. (It is also known as *scenario* in literature [PTB06].) For example, a mobile phone in one instant may be used to talk on the phone while surfing the web and downloading some Java application in the background. In another instant it may be used to listen to MP3 music while browsing JPEG pictures stored in the phone, and at the same time allow a remote device to access the files in the phone over a bluetooth connection. Modern devices are built to support different use-cases, making it possible for users to choose and use the desired functions concurrently.

Another trend we see is increasing and evolving standards. A number of standards for radio communication, audio and video encoding/decoding and interfaces are available. The multimedia systems often support a number of these. While a high-end TV supports a variety of video interfaces like HDMI, DVI, VGA, coaxial cable; a mobile phone supports multiple bands like GSM 850, GSM 900, GSM 180 and GSM 1900, besides other wireless protocols like Infrared and Bluetooth [MMZ$^+$02, KB97, Blu04]. As standards evolve, allowing faster and more efficient communication, newer devices are released in the market to match those specifications. The time to market is also reducing since a number of companies are in the market [JW04], and the consumers expect quick releases. A late launch in the market directly hurts the revenue of the company.

Power consumption has become a major design issue since many multimedia systems are hand-held. According to a survey by TNS research, two-thirds of mobile phone and PDA users rate *two-days of battery life during active use* as the most important feature of the ideal converged device of the future [TNS06]. While the battery life of portable devices has generally been increasing, the active use is still limited to a few hours, and in some extreme cases to a day. Even for other plugged multimedia systems, power has become a global concern with rising oil prices, and a growing awareness in people to reduce energy consumption.

To summarize, we see the following trends and requirements in the application of multimedia devices.

- An increasing number of multimedia devices are being brought to market.
- The number of applications in multimedia systems is increasing.
- The diversity of applications is increasing with convergence and multiple

standards.

- The applications execute concurrently in varied combinations known as use-cases, and the number of these use-cases is increasing.
- The time-to-market is reducing due to increased competition, and evolving standards and interfaces.
- Power consumption is becoming an increasingly important concern for future multimedia devices.

1.2 Trends in Multimedia Systems Design

A number of factors are involved in bringing the progress outlined above in multimedia systems. Most of them can be directly or indirectly attributed to the famous Moore's law [Moo65], that predicted the exponential increase in transistor density as early as 1965. Since then, almost every measure of the capabilities of digital electronic devices – processing speed, transistor count per chip, memory capacity, even the number and size of pixels in digital cameras – are improving at roughly exponential rates. This has had two-fold impact. While on one hand, the hardware designers have been able to provide bigger, better and faster means of processing, on the other hand, the application developers have been working hard to utilize this processing power to its maximum. This has led them to deliver better and increasingly complex applications in all dimensions of life – be it medical care systems, airplanes, or multimedia systems.

When the first Intel processor was released in 1971, it had 2,300 transistors and operated at a speed of 400 kHz. In contrast, a modern chip has more than a billion transistors operating at more than 3 GHz [Int09]. Figure 1.2 shows the trend in processor speed and the cost of memory [Ade08]. The cost of memory has come down from close to 400 U.S. dollars in 1971, to less than a cent for 1 MB of dynamic memory (RAM). The processor speed has risen to over 3.5 GHz. Another interesting observation from this figure is the introduction of dual and quad core chips since 2005 onwards. This indicates the beginning of multi-processor era. As the transistor size shrinks, they can be clocked faster. However, this also leads to an increase in power consumption, in turn making chips hotter. Heat dissipation has become a serious problem forcing chip manufacturers to limit the maximum frequency of the processor. Chip manufacturers are therefore, shifting towards designing multiprocessor chips operating at a lower frequency. Intel reports that *under-clocking* a single core by 20 percent saves half the power while sacrificing just 13 percent of the performance [Ros08]. This implies that if the work is divided between two processors running at 80 percent clock rate, we get 74 percent better performance for the same power. Further, the heat is dissipated at two points rather than one.

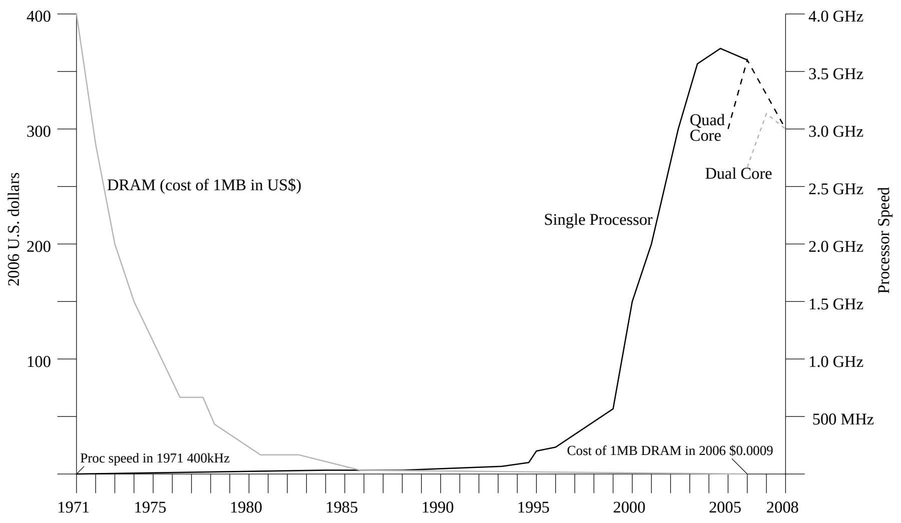


Figure 1.2: Increasing processor speed and reducing memory cost [Ade08].

Further, sources like Berkeley and Intel are already predicting hundreds and thousands of cores on the same chip [ABC$^+$06, Bor07] in the near future. All computing vendors have announced chips with multiple processor cores. Moreover, vendor road-maps promise to repeatedly double the number of cores per chip. These future chips are variously called *chip multiprocessors*, *multi-core chips*, and *many-core chips*, and the complete system as *multi-processor systems-on-chip* (MPSoC).

Following are the key benefits of using multi-processor systems.

- They consume less power and energy, provided sufficient task-level parallelism is present in the application(s). If there is insufficient parallelism, then some processors can be switched off.

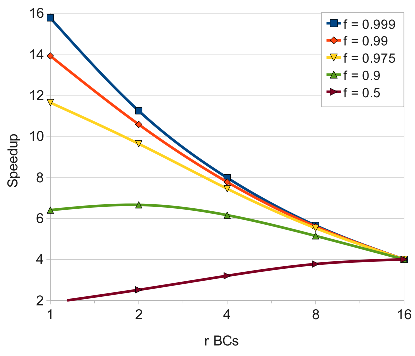
- Multiple applications can be easily shared among processors.

- Streaming applications (typical multimedia applications) can be more easily pipelined.

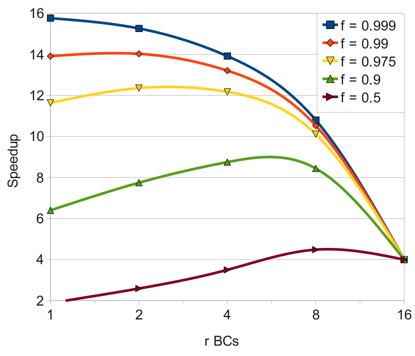
- More robust against failure – a Cell processor is designed with 8 cores (also known as SPE), but not all are always working.

- Heterogeneity can be supported, allowing better performance.

- It is more scalable, since higher performance can be obtained by adding more processors.



(a) Homogeneous systems



(b) Heterogeneous systems

Figure 1.3: Comparison of speedup obtained by combining r smaller cores into a bigger core in homogeneous and heterogeneous systems [HM08].

In order to evaluate the true benefits of multi-core processing, Amdahl's law [Amd67] has been augmented to deal with multi-core chips [HM08]. Amdahl's law is used to find the maximum expected improvement to an overall system when only a part of the system is improved. It states that if you enhance a fraction f of a computation by a speedup S, the overall speedup is:

$$Speedup_{enhanced}(f, S) = \frac{1}{(1-f) + \frac{f}{S}}$$

However, if the sequential part can be made to execute in less time by using a processor that has better sequential performance, the speedup can be increased. Suppose we can use the resources of r base-cores (BCs) to build one bigger core, which gives a performance of *perf(r)*. If $perf(r) > r$ i.e. super linear speedup, it is always advisable to use the bigger core, since doing so speeds up both sequential and parallel execution. However, usually $perf(r) < r$. When $perf(r) < r$, trade-off starts. Increasing core performance helps in sequential execution, but hurts parallel execution. If resources for n BCs are available on a chip, and all BCs are replaced with n/r bigger cores, the overall speedup is:

$$Speedup_{homogeneous}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f.r}{perf(r).n}}$$

When heterogeneous multiprocessors are considered, there are more possibilities to redistribute the resources on a chip. If only r BCs are replaced with 1 bigger core, the overall speedup is:

$$Speedup_{heterogeneous}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}}$$

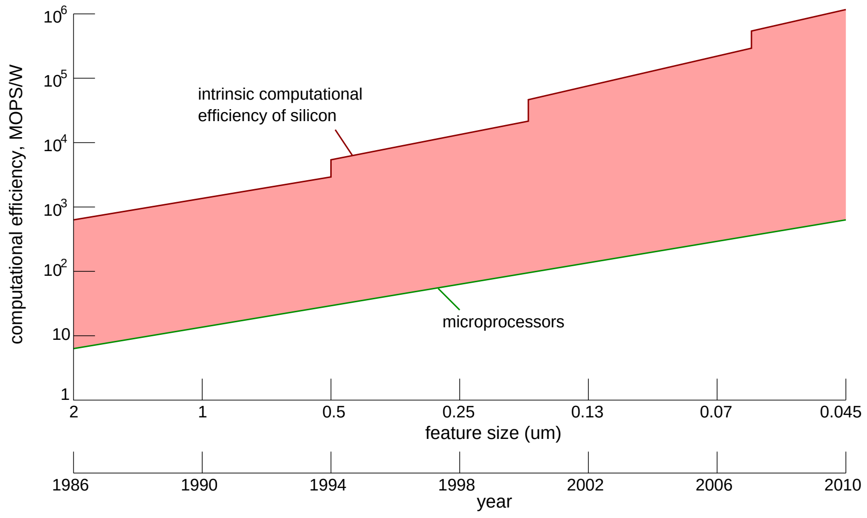


Figure 1.4: The intrinsic computational efficiency of silicon as compared to the efficiency of microprocessors.

Figure 1.3 shows the speedup obtained for both homogeneous and heterogeneous systems, for different fractions of parallelizable software. The x-axis shows the number of base processors that are combined into one larger core. In total there are resources for 16 BCs. The origin shows the point when we have a homogeneous system with only base-cores. As we move along the x-axis, the number of base-core resources used to make a bigger core are increased. In a homogeneous system, all the cores are replaced by a bigger core, while for heterogeneous, only one bigger core is built. The end-point for the x-axis is when all available resources are replaced with one big core. For this figure, it is assumed that $perf(r) = \sqrt{r}$. As can be seen, the corresponding speedup when using a heterogeneous system is much greater than homogeneous system. While these graphs are shown for only 16 base-cores, similar performance speedups are obtained for other bigger chips as well. This shows that using a heterogeneous system with several large cores on a chip can offer better speedup than a homogeneous system.

In terms of power as well, heterogeneous systems are better. Figure 1.4 shows the intrinsic computational efficiency of silicon as compared to that of microprocessors [Roz01]. The graph shows that the flexibility of general purpose microprocessors comes at the cost of increased power. The upper staircase-like line of the figure shows Intrinsic Computational Efficiency (ICE) of silicon according to an analytical model from [Roz01] ($MOPS/W \approx \alpha/\lambda V_{DD}^2$, α is constant, λ is feature size, and V_{DD} is the supply voltage). The intrinsic efficiency is in theory bounded on the number of 32-bit mega (adder) operations that can be achieved per second per Watt. The performance discontinuities in the upper staircase-like line are caused by changes in the supply voltage from 5V to 3.3V, 3.3V to 1.5V, 1.5V to 1.2V and 1.2 to 1.0V. We observe that there is a gap of 2-to-3 orders of magnitude between the intrinsic efficiency of silicon and general purpose microprocessors. The accelerators – custom hardware modules designed for a specific task – come close to the maximum efficiency. Clearly, it may not always be desirable to actually design a hypothetically maximum efficiency processor. A full match between the application and architecture can bring the efficiency close to the hypothetical maximum. A heterogeneous platform may combine the flexibility of using a general purpose microprocessor and custom accelerators for compute intensive tasks, thereby minimizing the power consumed in the system.

Most modern multiprocessor systems are heterogeneous, and contain one or more application-specific processing elements (PEs). The CELL processor [KDH$^+$05], jointly developed by Sony, Toshiba and IBM, contains up to nine-PEs – one general purpose PowerPC [WS94] and eight *Synergistic Processor Elements (SPEs)*. The PowerPC runs the operating system and the control tasks, while the SPEs perform the compute-intensive tasks. This Cell processor is used in PlayStation3 described above. STMicroelectronics Nomadik contains an ARM processor and several *Very Long Instruction Word (VLIW)* DSP cores [AAC$^+$03]. Texas Instruments OMAP processor [Cum03] and Philips Nexperia [OA03] are other examples. Recently, many companies have begun providing configurable cores that are targeted towards an application domain. These are known as *Application Specific Instruction-set Processors (ASIPs)*. These provide a good compromise between general-purpose cores and ASICs. Tensilica [Ten09, Gon00] and Silicon Hive [Hiv09, Hal05] are two such examples, which provide the complete toolset to generate multiprocessor systems where each processor can be customized towards a particular task or domain, and the corresponding software programming toolset is automatically generated for them. This also allows the re-use of IP (Intellectual Property) modules designed for a particular domain or task.

Another trend that we see in multimedia systems design is the use of *Platform-Based Design* paradigm [SVCBS04, KMN$^+$00]. This is becoming increasingly popular due to three main factors: (1) the dramatic increase in non-recurring engineering cost due to mask making at the circuit implementation level, (2) the reducing time to market, and (3) streamlining of industry – chip fabrication and system design, for example, are done in different companies and places. This paradigm is based on segregation between the system design process, and the system implementation process. The basic tenets of platform-based design are identification of design as *meeting-in-the-middle process*, where successive refinements of specifications meet with abstractions of potential implementations, and the identification of precisely defined abstraction layers where the refinement to the subsequent layer and abstraction processes take place [SVCBS04]. Each layer supports a design stage providing an opaque abstraction of lower layers that allows accurate performance estimations. This information is incorporated in appropriate parameters that annotate design choices at the present layer of abstraction. These layers of abstraction are called platforms. For MPSoC system design, this

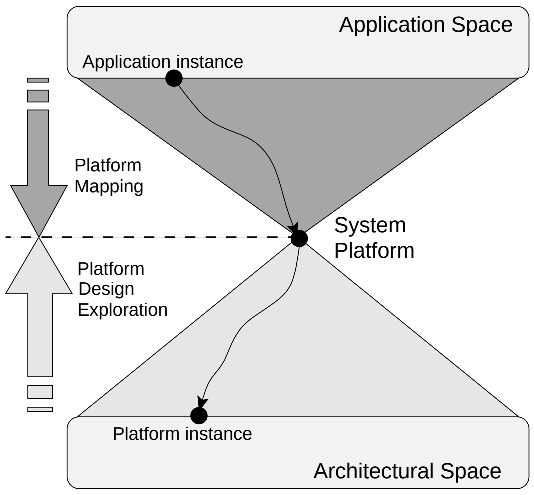


Figure 1.5: Platform-based design approach – system platform stack.

translates into abstraction between the application space and architectural space that is provided by the system-platform. Figure 1.5 captures this system-platform that provides an abstraction between the application and architecture space. This decouples the application development process from the architecture implementation process.

We further observe that for high-performance multimedia systems (like cell-processing engine and graphics processor), non-preemptive systems are preferred over preemptive ones for a number of reasons [JSM91]. In many practical systems, properties of device hardware and software either make the preemption impossible or prohibitively expensive due to extra hardware and (potential) execution time needed. Further, non-preemptive scheduling algorithms are easier to implement than preemptive algorithms and have dramatically lower overhead at run-time [JSM91]. Further, even in multi-processor systems with preemptive processors, some processors (or co-processors/ accelerators) are usually non-preemptive; for such processors non-preemptive analysis is still needed. It is therefore important to investigate non-preemptive multi-processor systems.

To summarize, the following trends can be seen in the design of multimedia systems.

- *Increase in system resources:* The resources available for disposal in terms of processing and memory are increasing exponentially.
- *Use of multiprocessor systems:* Multi-processor systems are being developed for reasons of power, efficiency, robustness, and scalability.
- *Increasing heterogeneity:* With the re-use of IP modules and design of cus-

tom (co-) processors (ASIPs), heterogeneity in MPSoCs is increasing.

- *Platform-based design:* Platform-based design methodology is being employed to improve the re-use of components and shorten the development cycle.

- *Non-preemptive processors:* Non-preemptive processors are preferred over preemptive to reduce cost.

1.3 Key Challenges in Multimedia Systems Design

The trends outlined in the previous two sections indicate the increasing complexity of modern multimedia systems. They have to support a number of concurrently executing applications with diverse resource and performance requirements. The designers face the challenge of designing such systems at low cost and in short time. In order to keep the costs low, a number of design options have to be explored to find the optimal or near-optimal solution. The performance of applications executing on the system have to be carefully evaluated to satisfy user-experience. Run-time mechanisms are needed to deal with run-time addition of applications. In short, following are the major challenges that remain in the design of modern multimedia systems, and are addressed in this thesis.

- *Multiple use-cases:* Analyzing performance of multiple applications executing concurrently on heterogeneous multi-processor platforms. Further, this number of use-cases and their combinations is exponential in the number of applications present in the system. (*Analysis and Design*)

- *Design and Program:* Systematic way to design *and* program multi-processor platforms. (*Design*)

- *Design space exploration:* Fast design space exploration technique. (*Analysis and Design*)

- *Run-time addition of applications:* Deal with run-time addition of applications – keep the analysis fast and composable, adapt the design (-process), manage the resources at run-time (e.g. admission controller). (*Analysis, Design and Management*)

- *Meeting performance constraints:* A good mechanism for keeping performance of all applications executing above the desired level. (*Design and Management*)

1.3.1 Analysis

We present a novel probabilistic performance prediction (P^3) algorithm for predicting performance of multiple applications executing on multi-processor platforms. The algorithm predicts the time that tasks have to spend during contention phase for a resource. The computation of accurate waiting time is the key to performance analysis. When applications are modeled as synchronous dataflow (SDF) graphs, their performance on a (multi-processor) system can be easily computed when they are executing *in isolation* (provided we have a *good* model). When they execute concurrently, depending on whether the used scheduler is static or dynamic, the arbitration on a resource is either fixed at design-time or chosen at run-time respectively (explained in more detail in Chapter 2). In the former case, the execution order can be modeled in the graph, and the performance of the entire application can be determined. The contention is therefore modeled as dependency edges in the SDF graph. However, this is more suited for static applications. For dynamic applications such as multimedia, dynamic scheduler is more suitable. For dynamic scheduling approaches, the contention has to be modeled as waiting time for a task, which is added to the execution time to give the total response time. The performance can be determined by computing the performance (throughput) of this resulting SDF graph. With lack of good techniques for accurately predicting the time spent in contention, designers have to resort to worst-case waiting time estimates, that lead to over-designing the system and loss of performance. Further, those approaches are not scalable and the over-estimate increases with the number of applications.

In this thesis, we present a solution to performance prediction, with easy analysis. We highlight the issue of *composability* i.e. mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation, as far as possible. This limits computational complexity and allows high dynamism in the system. While in this thesis, we only show examples with processor contention, memory and network contention can also be easily modeled in SDF graph as shown in [Stu07]. The technique presented here can therefore be easily extended to other system components as well. The analysis technique can be used both at design-time and run-time.

We would ideally want to analyze each application in isolation, thereby reducing the analysis time to a linear function, and still reason about the overall behaviour of the system. One of the ways to achieve this, would be complete *virtualization*. This essentially implies dividing the available resources by the total number of applications in the system. The application would then have exclusive access to its share of resources. For example, if we have 100 MHz processors and a total of 10 applications in the system, each application would get 10 MHz of processing resource. The same can be done for communication bandwidth and memory requirements. However this gives two main problems. When fewer than 10 tasks are active, the tasks will not be able to exploit the extra available processing power, leading to wastage. Secondly, the system would be grossly over-dimensioned when the peak requirements of each application are taken into account, even though these peak requirements of applications may rarely occur and never be at the same time.

Figure 1.6 shows this disparity in more detail. The graph shows the period of

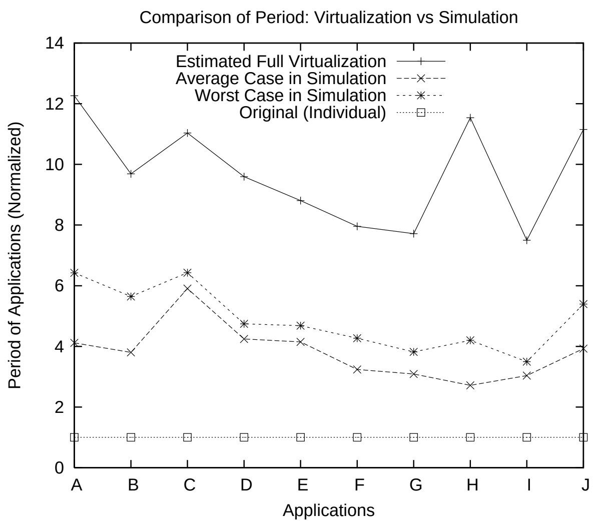


Figure 1.6: Application performance as obtained with full virtualization in comparison to simulation.

ten streaming multimedia applications (inverse of throughput) when they are run concurrently. The period is the time taken for one iteration of the application. The period has been normalized to the original period that is achieved when each application is running in isolation. If full virtualization is used, the period of applications increases to about ten times on average. However, without virtualization, it increases only about five times. A system which is built with full-virtualization in mind, would therefore, utilize only 50% of the resources. Thus, throughput decreases with complete virtualization.

Therefore, a good analysis methodology for a modern multimedia system

- provides accurate performance results, such that the system is not over-dimensioned,
- is fast in order to make it usable for run-time analysis, and to explore a large number of design-points quickly, and
- easily handles a large number of applications, and is composable to allow run-time addition of new applications.

It should be mentioned that often in applications, we are concerned with the long-term throughput and not the individual deadlines. For example, in the case of JPEG application, we are not concerned with decoding of each macro-block, but the whole image. When browsing the web, individual JPEG images are not as important as the entire page being ready. Thus, for the scope of this thesis, we consider long-term throughput i.e. cumulative deadline for a large number of iterations, and not just one. However, having said that it is possible to adapt the analysis to individual deadlines as well. It should be noted that in such cases, the estimates for individual iteration may be very pessimistic as compared to long-term throughput estimates.

1.3.2 Design

As is motivated earlier, modern systems need to support many different combinations of applications – each combination is defined as a *use-case* – on the same hardware. With reducing time-to-market, designers are faced with the challenge of designing and testing systems for multiple use-cases quickly. Rapid prototyping has become very important to easily evaluate design alternatives, and to explore hardware and software alternatives quickly. Unfortunately, lack of automated techniques and tools implies that most work is done by hand, making the design-process error-prone and time-consuming. This also limits the number of design-points that can be explored. While some efforts have been made to automate the flow and raise the abstraction level, these are still limited to single-application designs.

Modern multimedia systems support not just multiple applications, but also multiple use-cases. The number of such potential use-cases is exponential in the number of applications that are present in the system. The high demand of functionalities in such devices is leading to an increasing shift towards developing systems in software and programmable hardware in order to increase design flexibility. However, a single configuration of this programmable hardware may not be able to support this large number of use-cases with low cost and power. We envision that future complex embedded systems will be partitioned into several configurations and the appropriate configuration will be loaded into the *reconfigurable platform* (defined as a piece of hardware that can be configured at runtime to achieve the desired functionality) on the fly as and when the use-cases are requested. This requires two major developments at the research front: (1) a systematic design methodology for allowing multiple use-cases to be merged on a single hardware configuration, and (2) a mechanism to keep the number of hardware configurations as small as possible. More hardware configurations imply a higher cost since the configurations have to be stored in the memory, and also lead to increased switching in the system.

In this thesis, we present *MAMPS* (Multi-Application Multi-Processor Synthesis) – a design-flow that generates the entire MPSoC for multiple use-cases from application(s) specifications, together with corresponding software projects for automated synthesis. This allows the designers to quickly traverse the design-space and evaluate the performance on real hardware. Multiple use-cases of applications are supported by merging such that minimal hardware is generated. This further reduces the time spent in system-synthesis. When not all use-cases can be supported with one configuration, due to the hardware constraints, multiple configurations of hardware are automatically generated, while keeping the number of partitions low. Further, an area estimation technique is provided that can accurately predict the area of a design and decide whether a given system-design is feasible within the hardware constraints or not. This helps in quick evaluation of designs, thereby making the DSE faster.

Thus, the design-flow presented in this thesis is unique in a number of ways: (1) it supports multiple use-cases on one hardware platform, (2) estimates the area of design before the actual synthesis, allowing the designer to choose the right device, (3) merges and partitions the use-cases to minimize the number of hardware configurations, and (4) it allows fast DSE by automating the design generation and exploration process.

The work in this thesis is targeted towards heterogeneous multi-processor systems. In such systems, the mapping is largely determined by the capabilities of processors and the requirements of different tasks. Thus, the freedom in terms of mapping is rather limited. For homogeneous systems, task mapping and scheduling are coupled by performance requirements of applications. If for a particular scheduling policy, the performance of a given application is not met, mapping may need to be altered to ensure that the performance improves. As for the scheduling policy, it is not always possible to steer them at run-time. For example, if a system uses first-come-first-serve scheduling policy, it is infeasible to change it to a fixed priority schedule for a short time, since it requires extra hardware and software. Further, identifying the ideal mapping given a particular scheduling policy already takes exponential time in the total number of tasks. When the scheduling policy is also allowed to vary independently on processors, the time taken increases even more.

1.3.3 Management

Resource management, i.e. managing all the resources present in the multiprocessor system, is similar to the task of an operating system on a general purpose computer. This includes starting up of applications, and allocating resources to them appropriately. In the case of a multimedia system (or embedded systems, in general), a key difference from a general purpose computer is that the applications (or application domain) is generally known, and the system can be optimized for them. Further, most decisions can be already taken at design-time to save the cost at run-time. Still, a complete design-time analysis is becoming increasingly harder due to three major reasons: 1) little may be known at design-time about the applications that need to be used in future, e.g. a navigation application like Tom-Tom may be installed on the phone after-wards, 2) the precise platform may also not be known at design time, e.g. some cores may fail at run-time, and 3) the number of design-points that need to be evaluated is prohibitively large. A run-time approach can benefit from the fact that the exact application mix is known, but the analysis has to be fast enough to make it feasible.

In this thesis, we present a **hybrid approach** for designing systems with multiple applications. This splits the management tasks into off-line and on-line. The time-consuming application specific computations are done at design-time and for each application independent from other applications, and the use-case specific computations are performed at run-time. The off-line computation includes tasks like application-partitioning, application-modeling, determining the task execution times, determining their maximum throughput, etc. Further, parametric equations are derived that allow throughput computation of tasks with varying execution times. All this analysis is time-consuming and best carried out at design-time. Further, in this part no information is needed from the other applications and it can be performed in isolation. This information is sufficient enough to let a run-time manager determine the performance of an application when executing concurrently on the platform with other applications. This allows easy **addition of applications at run-time**. As long as all the properties needed by the run-time resource manager are derived for the new application, the application can be treated as all the other applications that are present in the system.

At run-time, when the resource manager needs to decide, for example, which resources to allocate to an incoming application, it can evaluate the performance of applications with different allocations and determine the best option. In some cases, multiple quality levels of an application may be specified, and at run-time the resource manager can choose from one of those levels. This functionality of the resource manager is referred to as **admission control**. The manager also needs to ensure that applications that are admitted do not take more resources than allocated, and starve the other applications executing in the system. This functionality is referred to as **budget enforcement**. The manager periodically checks the performance of all applications, and when an application does better than the required level, it is suspended to ensure that it does not take more resources than needed. For the scope of this thesis, the effect of task migration is not considered since it is orthogonal to our approach.

1.4 Design Flow

Figure 1.7 shows the design-flow that is used in this thesis. Specifications of applications are provided to the designer in the form of Synchronous Dataflow (SDF) graphs [SB00, LM87]. These are often used for modeling multimedia applications. This is further explained in Chapter 2. As motivated earlier in the chapter, modern multimedia systems support a number of applications in varied combinations defined as **use-case**. Figure 1.7 shows three example applications – A, B and C, and three use-cases with their combinations. For example, in *Use-case 2* applications A and B execute concurrently. For each of these use-cases, the performance of all active applications is analyzed. When a suitable mapping to hardware is to be explored, this step is often repeated with different mappings, until the desired performance is obtained. A probabilistic mechanism is used to

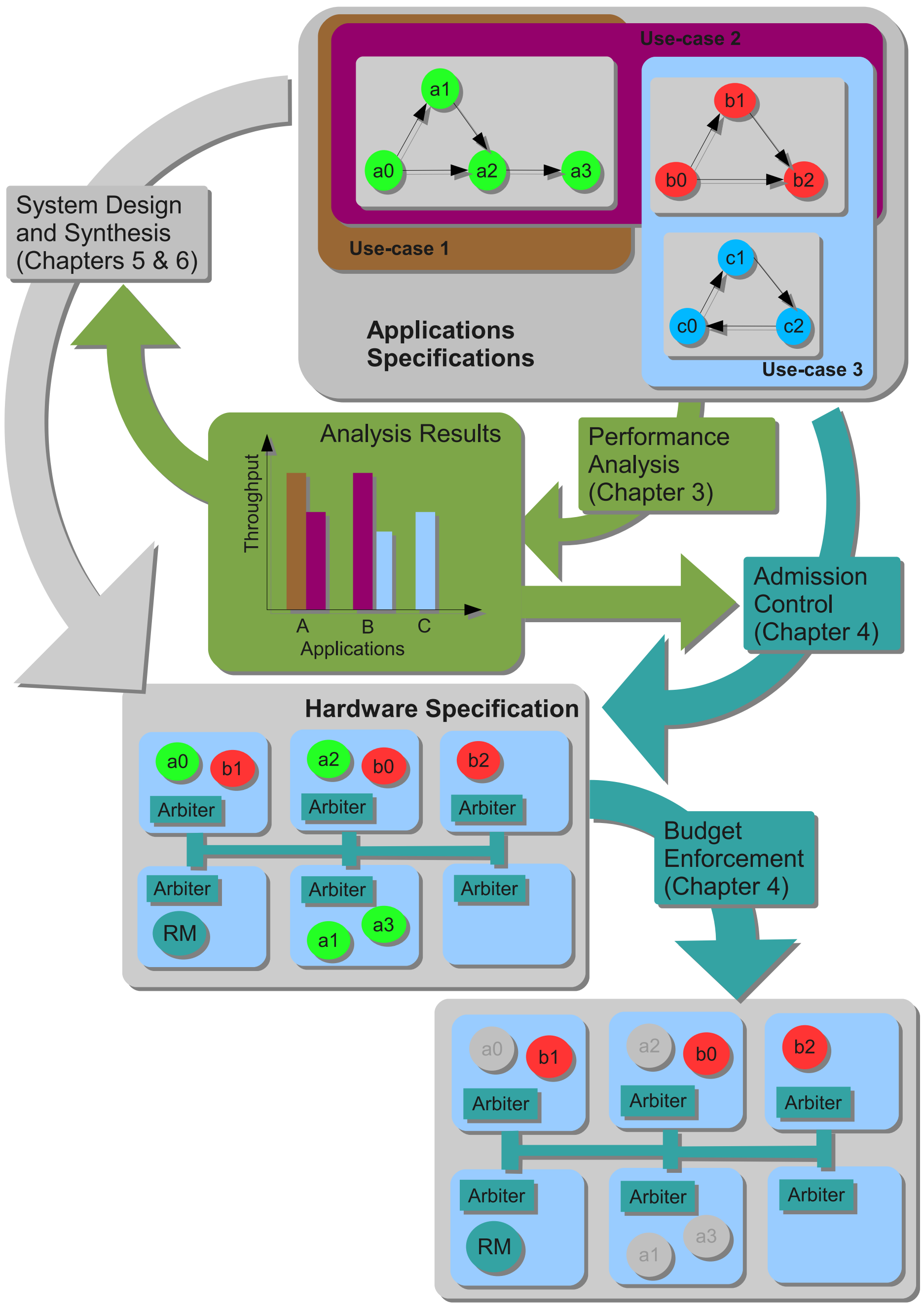

Use-case 2
a1
a0
a2
a3
b1
b0
b2
Use-case 1
c1
c0
c2
Applications Specifications
Use-case 3
System Design and Synthesis (Chapters 5 & 6)
Analysis Results
Throughput
A
B
C
Applications
Performance Analysis (Chapter 3)
Admission Control (Chapter 4)
Hardware Specification
Arbiter
RM
Budget Enforcement (Chapter 4)

Figure 1.7: Complete design flow starting from applications specifications and ending with a working hardware prototype on an FPGA.

estimate the average performance of applications. This **performance analysis** technique is presented in Chapter 3.

When a satisfactory mapping is obtained, the system can be designed and synthesized automatically using the **system-design** approach presented in Chapter 5. **Multiple use-cases** need to be merged on to one hardware design such that a new hardware configuration is not needed for every use-case. This is explained in Chapter 6. When it is not possible to merge all use-cases due to resource constraints (slices in an FPGA, for example), use-cases need to be partitioned such that the number of hardware partitions are kept to a minimum. Further, a fast area estimation method is needed that can quickly identify whether a set of use-cases can be merged due to hardware constraints. Trying synthesis for every use-case combination is too time-consuming. A novel area-estimation technique is needed that can save precious time during design space exploration. This is explained in Chapter 6.

Once the system is designed, a run-time mechanism is needed to ensure that all applications can meet their performance requirements. This is accomplished by using a resource manager (RM). Whenever a new application is to be started, the manager checks whether sufficient resources are available. This is defined as **admission-control**. The probabilistic analysis is used to predict the performance of applications when the new application is admitted in the system. If the expected performance of all applications is above the minimum desired performance then the application is started, else a lower quality of incoming application is tried. The resource manager also takes care of **budget-enforcement** i.e. ensuring applications use only as much resources as assigned. If an application uses more resources than needed and starves other applications, it is suspended. Figure 1.7 shows an example where application A is suspended. Chapter 4 provides details of two main tasks of the RM – admission control and budget-enforcement.

The above flow also allows for run-time addition of applications. Since the performance analysis presented is fast, it is done at run-time. Therefore, any application whose properties have been derived off-line can be used, if there are enough resources present in the system. This is explained in more detail in Chapter 4.

1.5 Key Contributions and Thesis Overview

Following are some of the major contributions that have been achieved during the course of this research and have led to this thesis.

- A detailed analysis of why estimating performance of multiple applications executing on a heterogeneous platform is so difficult. This work was published in [KMC+06], and an extended version is published in a special issue of the Journal of Systems Architecture containing the best papers of the Digital System Design conference [KMT+08].

- A probabilistic performance prediction (P^3) mechanism for multiple applications. The prediction is within 2% of real performance for experiments done. The basic version of the P^3 mechanism was first published in [KMC$^+$07], and later improved and published in [KMCH08].
- An admission controller based on P^3 mechanism to admit applications only if they are expected to meet their performance requirements. This work is published in [KMCH08].
- A budget enforcement mechanism to ensure that applications can all meet their desired performance if they are admitted. This work is published in [KMT$^+$06].
- A *Resource Manager* (RM) to manage computation and communication resources, and achieve the above goals. This work is published in [KMCH08].
- A design flow for multiple applications, such that composability is maintained and applications can be added at run-time with ease.
- A platform synthesis design technique that generates multiprocessors platforms with ease automatically and also programs them with relevant program codes, for multiple applications. This work is published in [KFH$^+$07].
- A design flow explaining how systems that support multiple use-cases should be designed. This work is published in [KFH$^+$08].

A tool-flow based on the above for Xilinx FPGAs that is also made available for use on-line for the benefit of research community. This tool is available on-line at `www.es.ele.tue.nl/mamps/` [MAM09].

This thesis is organized as follows. Chapter 2 explains the concepts involved in modeling and scheduling of applications. It explores the problems encountered when analyzing multiple applications executing on a multi-processor platform. The challenge of *Composability*, i.e. being able to analyze applications in isolation with other applications, is presented in this chapter. Chapter 3 presents a performance prediction methodology that can accurately predict the performance of applications at run-time before they execute in the system. A run-time iterative probabilistic analysis is used to estimate the time spent by tasks during contention phase, and thereby predict the performance of applications. Chapter 4 explains the concepts of resource management and enforcing budgets to meet the performance requirements. The performance prediction is used for admission control – one of the main functions of the resource manager. Chapter 5 proposes an automated design methodology to generate program MPSoC hardware designs in a systematic and automated way for multiple applications named *MAMPS*. Chapter 6 explains how systems should be designed when multiple use-cases have to be supported. Algorithms for merging and partitioning use-cases are presented in this chapter as well. Finally, Chapter 7 concludes this thesis and gives directions for future work.

CHAPTER 2

Application Modeling and Scheduling

Multimedia applications are becoming increasingly more complex and computation hungry to match consumer demands. If we take video, for example, televisions from leading companies are already available with high-definition (HD) video resolution of 1080x1920 i.e. more than 2 million pixels [Son09, Sam09, Phi09] for consumers and even higher resolutions are showcased in electronic shows [CES09]. Producing images for such a high resolution is already taxing for even high-end MPSoC platforms. The problem is compounded by the extra dimension of multiple applications sharing the same resources. *Good modeling* is essential for two main reasons: 1) to predict the behaviour of applications on a given hardware without actually synthesizing the system, and 2) to synthesize the system after a feasible solution has been identified from the analysis. In this chapter we will see in detail the model requirements we have for designing and analyzing multimedia systems. We see the various models of computation, and choose one that meets our design-requirements.

Another factor that plays an important role in multi-application analysis is determining when and where a part of application is to be executed, also known as *scheduling*. Heuristics and algorithms for scheduling are called *schedulers*. Studying schedulers is essential for good system design and analysis. In this chapter, we discuss the various types of schedulers for dataflow models. When considering multiple applications executing on multi-processor platforms, three main things need to be taken care of: 1) *assignment* – deciding which task of application has to be executed on which processor, 2) *ordering* – determining the order of task-execution, and 3) *timing* – determining the precise time of task-execution[1]. Each of these three tasks can be done at either compile-time or run-time. In this chapter, we classify the schedulers on this criteria and highlight two of them most suited for use in multiprocessor multimedia platforms. We highlight the issue of *composability* i.e. mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation, as far as possible. This limits computational complexity and allows high dynamism in the system.

This chapter is organized as follows. The next section motivates the need of modeling applications and the requirements for such a model. Section 2.2 gives an introduction to the synchronous dataflow (SDF) graphs that we use in our analysis. Some properties that are relevant for this thesis are also explained in the same section. Section 2.3 discusses the models of computation (MoCs) that are available, and motivates the choice of SDF graphs as the MoC for our applications. Section 2.4 gives state-of-the-art techniques used for estimating performance of applications modeled as SDF graphs. Section 2.5 provides background on the scheduling techniques used for dataflow graphs in general. Section 2.6 extends the performance analysis techniques to include hardware constraints as well. Section 2.8 provides a comparison between static and dynamic ordering schedulers, and Section 2.9 concludes the chapter.

2.1 Application Model and Specification

Multimedia applications are often also referred to as **streaming applications** owing to their repetitive nature of execution. Most applications execute for a very long time in a fixed execution pattern. When watching television for example, the video decoding process potentially goes on decoding for hours – an hour is equivalent to 180,000 video frames at a modest rate of 50 frames per second (fps). High-end televisions often provide a refresh rate of even 100 fps, and the trend indicates further increase in this rate. The same goes for an audio stream that usually accompanies the video. The platform has to work continuously to get this output to the user.

In order to ensure that this high performance can be met by the platform, the designer has to be able to model the application requirements. In the absence of a good model, it is very difficult to know in advance whether the application performance can be met at all times, and extensive simulation and testing is needed. Even now, companies report a large effort being spent on verifying the timing requirements of the applications. With multiple applications executing on multiple processors, the potential number of use-cases increases rapidly, and so does the cost of verification.

We start by defining a use-case.

[1]Some people also define only ordering and timing as scheduling, and assignment as *binding* or *mapping*.

Definition 1 (USE-CASE:) *Given a set of n applications $A_0, A_1, \dots A_{n-1}$, a use-case U is defined as a vector of n elements $(x_0, x_1, \dots x_{n-1})$ where $x_i \in \{0,1\} \; \forall \;\, i = 0,1,\dots n-1$, such that $x_i = 1$ implies application A_i is active.*

In other words, a use-case represents a collection of multiple applications that are active simultaneously. It is impossible to test a system with all potential input cases in advance. Modern multimedia platforms (high-end mobile phones, for example) allow users to download applications at run-time. Testing for those applications at design-time is simply not possible. A good model of an application can allow for such analysis at run-time.

One of the major challenges that arise when mapping an application to an MPSoC platform is dividing the application load over multiple processors. Two ways are available to parallelize the application and divide the load over more than one processor, namely task-level parallelism (also known as pipe-lining) and data-level parallelism. In the former, each processor gets a different part of an application to process, while in the latter, processors operate on the same functionality of application, but different data. For example, in case of JPEG image decoding, inverse discrete cosine transform (IDCT) and colour conversion (CC), among other tasks, need to be performed for all parts (macro-blocks) of an image. Splitting the task of IDCT and CC on different processors is an example of task-level parallelism. Splitting the data, in this case macro-blocks, to different processors is an example of data-level parallelism. To an extent, these approaches are orthogonal and can be applied in isolation or in combination. In this thesis, we shall focus primarily on task-level parallelism.

Parallelizing an application to make it suitable for execution on a multiprocessor platform can be a very difficult task. Whether an application is written from start in a manner that is suitable for SDF model, or whether an SDF model is extracted from the existing (sequential) application, in either case we need to know how long the execution of each program segment will take; how much data and program memory will be needed for it; and when communication program segments are mapped on different processors, how much communication buffer capacity do we need. Further, we also want to know what is the maximum performance that the application can achieve on a given platform, especially when sharing the platform with other applications. For this, we have to also be able to model and analyze scheduling decisions.

To summarize, following are our requirements from an application model that allow mapping and analysis on a multiprocessor platform:

- *Analyze computational requirements:* When designing an application for MPSoC platform, it is important to know how much computational resource an application needs. This allows the designers to dimension the hardware appropriately. Further, this is also needed to compute the performance estimates of the application as a whole. While sometimes, average case analysis of requirements may suffice, often we also need the worst case estimates, for example in case of real-time embedded systems.

- *Analyze memory requirements:* This constraint becomes increasingly more important as the memory cost on a chip goes high. A model that allows accurate analysis of memory needed for the program execution can allow a designer to distribute the memory across processors appropriately and also determine proper mapping on the hardware.

- *Analyze communication requirements:* The buffer capacity between the communicating tasks (potentially) affects the overall application performance. A model that allows computing these buffer-throughput trade-offs can let the designer allocate appropriate memory for the channel and predict throughput.

- *Model and analyze scheduling:* When we have multiple applications sharing processors, scheduling becomes one of the major challenges. A model that allows us to analyze the effect of scheduling on applications performance is needed.

- *Design the system:* Once the performance of system is considered satisfactory, the system has to be synthesized such that the properties analyzed are still valid.

Dataflow models of computation fit rather well with the above requirements. They provide a model for describing signal processing systems where infinite streams of data are incrementally transformed by processes executing in sequence or parallel. In a dataflow model, processes communicate via unbounded FIFO channels. Processes read and write atomic data elements or tokens from and to channels. Writing to a channel is non-blocking, i.e. it always succeeds and does not stall the process, while reading from a channel is blocking, i.e. a process that reads from an empty channel will stall and can only continue when the channel contains sufficient tokens. In this thesis, we use synchronous dataflow (SDF) graph to model applications and the next section explains them in more detail.

2.2 Introduction to SDF Graphs

Synchronous Data Flow Graphs (SDFGs, see [LM87]) are often used for modeling modern DSP applications [SB00] and for designing concurrent multimedia applications implemented on multi-processor systems-on-chip. Both pipelined streaming and cyclic dependencies between tasks can be easily modeled in SDFGs. Tasks are modeled by the vertices of an SDFG, which are called *actors*. The communication between actors is represented by *edges* through which it is connected to other actors. Edges represent *channels* for communication in a real system.

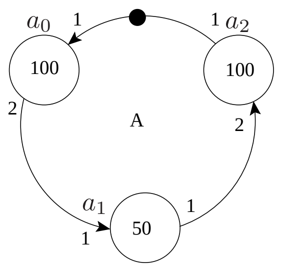


Figure 2.1: Example of an SDF Graph

The time that the actor takes to execute on a processor is indicated by the number inside the actor. It should be noted that the time an actor takes to execute may vary with the processor. For sake of simplicity, we shall omit the detail as to which processor it is mapped, and just define the time (or clock cycles) needed on a RISC processor [PD80], unless otherwise mentioned. This is also sometimes referred to as *Timed SDF* in literature [Stu07]. Further, when we refer to the time needed to execute a particular actor, we refer to the worst-case execution-time (WCET). The average execution time may be lower.

Figure 2.1 shows an example of an SDF graph. There are three actors in this graph. As in a typical data flow graph, a directed edge represents the dependency between actors. Actors need some input data (or control information) before they can start, and usually also produce some output data; such information is referred to as *tokens*. The number of tokens produced or consumed in one execution of actor is called *rate*. In the example, a_0 has an input rate of 1 and output rate of 2. Further, its execution time is 100 clock cycles. Actor execution is also called *firing*. An actor is called *ready* when it has sufficient input tokens on all its input edges and sufficient buffer space on all its output channels; an actor can only fire when it is ready.

The edges may also contain *initial tokens*, indicated by bullets on the edges, as seen on the edge from actor a_2 to a_0 in Figure 2.1. In the above example, only a_0 can start execution from the initial state, since the required number of tokens are present on its only incoming edge. Once a_0 has finished execution, it will produce 2 tokens on the edge to a_1. a_1 can then proceed, as it has enough tokens, and upon completion produce 1 token on the edge to a_2. However, a_2 has to wait before two executions of a_1 are completed, since it needs two input tokens.

A number of properties of an application can be analyzed from its SDF model. We can calculate the maximum performance possible of an application. We can identify whether the application or a particular schedule will result in a deadlock. We can also analyze other performance properties, e.g. latency of an application, buffer requirements. Below we give some properties of SDF graphs that allow modeling of hardware constraints that are relevant to this thesis.

2.2.1 Modeling Auto-concurrency

The example in Figure 2.1 brings a very interesting fact to notice. According to the model, since a_1 requires only one token on the edge from a_0 to fire, as soon as a_0 has finished executing and produced two tokens, two executions of a_1 can start simultaneously. However, this is only possible if a_1 is mapped and allowed to execute on multiple processors simultaneously. In a typical system, a_1 will be mapped on a processor. Once the processor starts executing, it will not be available to start the second execution of a_1 until it has at least finished the first execution of a_1. If there are other actors mapped on it, the second execution of a_1 may even be delayed further.

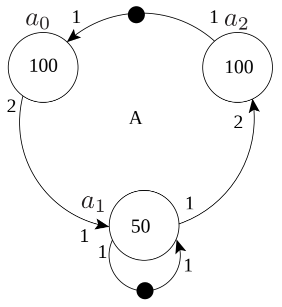


Figure 2.2: SDF Graph after modeling auto-concurrency of 1 for the actor a_1

Fortunately, there is a way to model this particular resource conflict in SDF. Figure 2.2 shows the same example, now updated with the constraint that only one execution of a_1 can be active at any one point in time. In this figure, a *self-edge* has been added to the actor a_1 with one initial token. (In a self-edge, the source and destination actor is the same.) This initial token is consumed in the first firing of a_1 and produced after a_1 has finished the first execution. Interestingly enough, by varying the number of initial tokens on this self-edge, we can regulate the number of simultaneous executions of a particular actor. This property is called **auto-concurrency**.

Definition 2 (AUTO-CONCURRENCY) *The* **auto-concurrency** *of an actor is defined as the maximum number of simultaneous executions of that actor.*

In Figure 2.2, the auto-concurrency of a_1 is 1, while for a_0 and a_2 it is infinite. In other words, the resource conflict for actors a_0 and a_2 is not modeled. In fact, the single initial token on the edge from a_2 to a_0 limits the auto-concurrency of these two actors to one; a self-edge in this case would be superfluous.

2.2.2 Modeling Buffer Sizes

One of the very useful properties of SDF graphs is its ability to model available buffers easily. Buffer-sizes may be modeled as a back-edge with initial tokens. In such cases, the number of tokens on that edge indicates the buffer-size available. When an actor writes data on a channel, the available size reduces; when the receiving actor consumes this data, the available buffer increases, modeled by an increase in the number of tokens.

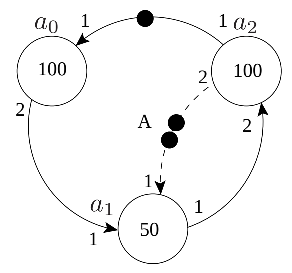


Figure 2.3: SDF Graph after modeling buffer-size of 2 on the edge from actor a_2 to a_1

Figure 2.3 shows such an example, where the buffer size of the channel from a_1 to a_2 is shown as two. Before a_1 can be executed, it has to check if enough buffer space is available. This is modeled by requiring tokens from the back-edge to be consumed. Since it produces one token per firing, one token from the back-edge is consumed, indicating reservation of one buffer space on the output edge. On the consumption side, when a_2 is executed, it frees two buffer spaces, indicated by a release of two tokens on the back-edge. In the model, the output buffer space is claimed at the start of execution, and the input token space is released only at the end of firing. This ensures atomic execution of the actor.

2.3 Comparison of Dataflow Models

While SDF graphs allow analysis of many properties and are well-suited for multimedia applications, they do have some restrictions. For example, conditional and data-dependent behaviour cannot be expressed in these models. In this section, we provide an overview of the other models of computation (MoC). In [Stu07], Stuijk has summarized and compared many models on the basis of their expressiveness and succinctness, efficiency of implementation, and analyzability. Expressiveness determines to what extent real-applications can be represented in a particular model. Models that are static in nature (e.g. SDF) cannot capture behaviour of highly dynamic applications (e.g. object segmentation from an input sequence) accurately. Succinctness (or compactness) determines how compact that representation is. Efficiency of implementation determines how easily the application model can be implemented in terms of its schedule length. Analyzability determines to what extent the model can be analyzed and performance properties of applications determined. As mentioned in the earlier section, this is one of the most important considerations for us. In general, a model that can be easily analyzed at design-time is also more efficient for implementation, since most scheduling and resource assignment decisions can be made at design-time. Figure 2.4 shows how different models are placed on these three axes.

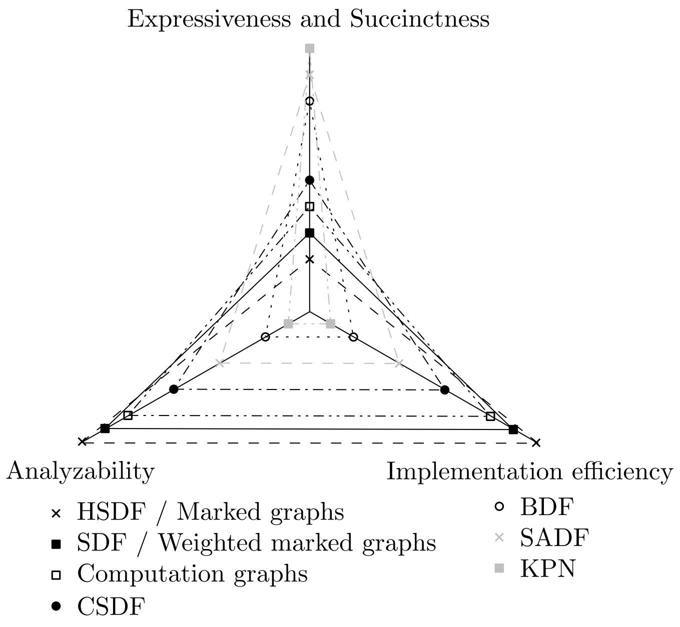


Figure 2.4: Comparison of different models of computation [Stu07].

Kahn Process Network

Kahn process network (KPN) was proposed by Kahn in 1974 [Kah74]. The amount of data read from an edge may be data-dependent. This allows modeling of any continuous function from the inputs of the KPN to the outputs of the KPN with an arbitrarily small number of processes. KPN is sufficiently expressive to capture precisely all data dependent dataflow transformations. However, this also implies that in order to analyze properties like the throughput or buffer requirements of a KPN all possible inputs have to be considered.

Scenario Aware Dataflow

Scenario aware dataflow (SADF) was first introduced by Theelen in 2006 [TGB+06]. This is also a model for design-time analysis of dynamic streaming and signal processing applications.This model also allows for data-dependent behaviour in processes. Each different execution pattern is defined as a *scenario*. Such scenarios denote different modes of operations in which resource requirements can differ considerably. The scenario concept enables to coherently capture the variations in behaviour of different processes in a streaming application. A key novelty of SADF is the use of a stochastic approach to capture the scenario occurrences as well as the occurrence of different execution times within a scenario in an abstract way. While some properties of these graphs like deadlock and throughput are possible to analyze at design time, in practice this analysis can be quite slow. This model is less compact than KPN, since all scenarios have to be explicitly specified in the model, and known at design time. This also makes it less expressive since not all kinds of systems can be expressed accurately in SADF.

Boolean Dataflow

The last model of computation that we discuss having data-dependent behaviour is boolean dataflow (BDF) model [Lee91, BL93]. In this model, each process has a number of inputs and outputs to choose from. Depending on the value of *control tokens* data is read from one of the input channels, and written to one of the output channels. This model is less expressive than the earlier two models discussed, since the control freedom in modeling processes is limited to either true or false. Similar to the earlier two models discussed, the analyzability is limited.

Cyclo Static Dataflow

Now we move on to the class of more deterministic data flow models of computation. In a cyclo-static dataflow (CSDF) model [LWAP94, BELP96], the rates of data consumed and produced may change between subsequent firings. However, the pattern of this change is pre-determined and that makes it more analyzable at design time. These graphs may be converted to SDF graphs, and are therefore as expressive as SDF graphs. However, the freedom to change the rates of data makes them more compact than SDF in representing some applications. They are also as analyzable, but slower if we consider the same number of actors, since the resulting schedule is generally a little more complex.

Recently, special channels have been introduced for CSDF graphs [DBC+07]. Often applications share buffers between multiple consumers. This cannot be directly described in CSDF. The authors show how such implementation specific aspects can be modeled in CSDF without the need of extensions. Thus, the analyzability of the graph is maintained, and appropriate buffer-sizes can be computed from the application model.

Computation Graphs

Computation graphs were first introduced by Karp and Miller in 1966 [KM66]. In these graphs there is a threshold set for each edge specifying the minimum number of tokens that should be present on that edge before an actor can fire. However, the number of tokens produced and consumed for each edge is still fixed. These models are less expressive than CSDF, but more analyzable. A synchronous data flow graph is a subset of these computation graphs.

Synchronous Dataflow

Synchronous dataflow (SDF) graphs were first proposed by Lee and Messerschmitt in 1987 [LM87]. However, as has been earlier claimed [Stu07], these correspond to subclass weighted marked graph [TCWCS92] of Petri nets, which is a general purpose model of computation with a number of applications [Pet62, Mur89]. SDF graphs have a constant input and output rate that does not change with input or across different firings. They also don't support execution in different scenarios as may be specified by data. Therefore, their expressivity is rather limited. However, this also makes them a lot easier to analyze. Many performance parameters can be analyzed as explained in Section 2.4.

Homogeneous Synchronous Data Flow

Homogeneous Synchronous Data Flow (HSDF) graphs are a subset of SDF graphs. In HSDF graph model, the rates of all input and output edges is one. This implies that only one token is read and written in any firing of an actor. This limits the expressiveness even more, but makes the analysis somewhat easier. HSDF graphs can be converted into SDF and vice-versa. However, in practice the size of an HSDF for an equivalent SDFG may be very large as shown by examples in [Stu07]. Lately, analysis techniques have been developed that work almost as fast directly on an SDF graph as on an HSDF graph (for the same number of nodes) [Gha08]. Therefore, the added advantage of using an HSDF graph is lost.

After considering all the alternatives, we decided in favour of SDF graphs since their ability to analyze applications in terms of other performance requirements, such as throughput and buffer, was one of our key requirements. Further, a number of analysis tools for SDF graph were available (and more in development) when this research was started [SDF09]. However, since SDF graphs are not able to express some real applications accurately, we do have to pay a little overhead in estimating performance. For example, the execution time is assumed to be the worst-case execution-time. Thus, in some cases, the performance estimates may be pessimistic.

2.4 Performance Modeling

In this section, we define the major terminology that is relevant for this thesis.

Definition 3 (Actor Execution Time) *Actor execution time, $\tau(a)$ is defined as the time needed to complete execution of actor a on a specified node. In cases where the required time is not constant but varying, this indicates the maximum time for actor execution.*

$\tau(a_0) = 100$, for example, in Figure 2.3.

Definition 4 (Iteration) *An* **iteration** *of a graph is defined as the minimum non-zero execution (i.e. at least one actor has executed) such that the initial state of the graph is obtained.*

In Figure 2.3, one iteration of graph A is completed when a_0, a_1 and a_2 have completed one, two and one execution(s) each respectively.

Definition 5 (Repetition Vector) *Repetition Vector q of an SDF graph A is defined as the vector specifying the number of times an actor in A is executed for one iteration of A.*

For example, in Figure 2.3, q[a0 a1 a2] = [1 2 1]. It should be mentioned that any integer multiple of repetition vector defined above is also a repetition vector. The above definition gives a *minimal* repetition vector in which all entries are integers.

Definition 6 (Application Period) *Application Period $Per(A)$ is defined as the time SDFG A takes to complete one iteration* on average.

$Per(A) = 300$ in Figure 2.3, assuming it has sufficient resources and no contention, and all the actors fire as soon as they are ready. (Note that actor a_1 has to execute twice.)

Definition 7 (Application Throughput) *Application Throughput, Thr_A is defined as the number of iterations of an SDF graph A in one second.*

This is simply the inverse of period, $Per(A)$, when period is defined in seconds. For example, an application with a throughput of 50 Hz takes 20 ms to complete one iteration. When the graph in Figure 2.3 is executing on a single processor of 300 MHz, the throughput of A is 1 MHz since the period is 1 micro-second.

Throughput is one of the most interesting properties of SDF graphs relevant to the design of any multimedia system. Designers and consumers both want to know the sustained throughput the system can deliver. This parameter often directly relates to the consumer. For example, throughput of an H.264 decoder may define how many frames can be decoded per second. A higher throughput in this case directly improves the consumer experience.

2.4.1 Steady-state vs Transient

Often, in an application, it takes a few iterations of the application before it starts its periodic behaviour. For example, consider the application graph as shown earlier in Figure 2.1, but now with three initial tokens on the edge from a_2 to a_0. Consider further, that each of the three actors is mapped on a multi-processor system with three processors, P_0, P_1 and P_2, such that actor a_i is mapped on P_i for i = 0, 1, 2. Let us assume that the processors are connected to each other with a point-to-point connection with infinite bandwidth with directions similar to channels in the application graph. Figure 2.5 shows the updated graph and the three processor system with the appropriate mapping.

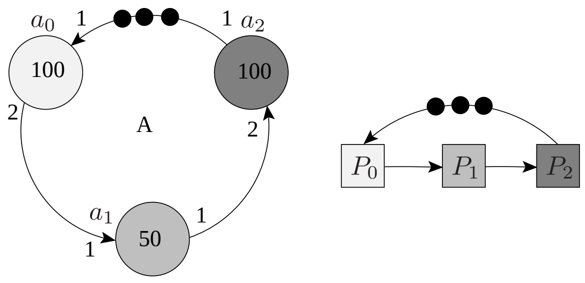


Figure 2.5: SDF Graph and the multi-processor architecture on which it is mapped

In this example, if we look at the time taken for one single iteration, we get a period of 300 cycles. However, since each actor has its own dedicated processor, soon we get the token distribution as shown in Figure 2.6. From this point onwards, all the actors can continue firing indefinitely since all actors have sufficient tokens and dedicated resources. Thus, every 100 cycles, an iteration of application A is completed. (Note that the first iteration still takes 300 cycles to be completed.) This final state is called *steady-state*. The initial execution of the graph leading to this state is called *transient phase*.

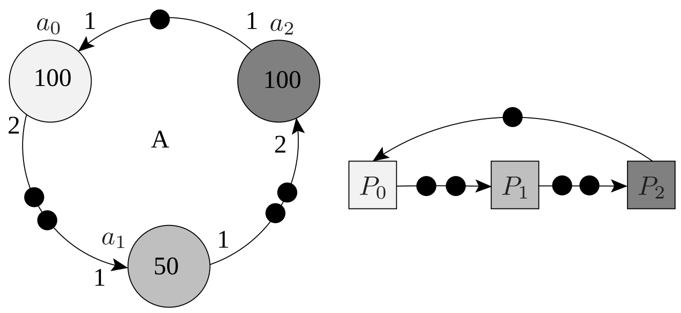


Figure 2.6: Steady-state is achieved after two executions of a_0 and one of a_1

For the graph as shown in Figure 2.5, the maximal throughput for 300 MHz processors is 3 MHz or three million iterations per second. In this thesis when we refer to the throughput of a graph, we generally refer to the maximal achievable throughput of a graph, unless otherwise mentioned. This only refers to the steady-state throughput. When we use the term *achieved throughput* of a graph, we shall refer to the long-term average throughput achieved for a given application. This also includes the transient phase of an application. Please note that for *infinitely long* execution, the long-term average throughput is the same as the steady-state throughput.

Another way to define throughput is the rate of execution of an *output* actor divided by its repetition vector entry. If we consider actor a_2 as the output actor of application A, we see that the throughput of the application is the same as the execution rate of a_2, since its repetition vector entry is 1.

2.4.2 Throughput Analysis of (H)SDF Graphs

A number of analysis techniques are available to compute the throughput of SDF graphs [Gha08, Stu07, SB00, Das04, BKKB02]. Most of these techniques first convert an SDF graph into a homogeneous SDF (HSDF) graph. HSDF is a special class of SDF in which the number of tokens consumed and produced is always equal to one. Techniques are available to convert an SDF into HSDF and the other way around [SB00]. After conversion to HSDF, throughput is computed as the inverse of the *maximal cycle mean (MCM)* of the HSDF graph [Das04, KM66]. MCM in turn is the maximum of all *cycle-means*. A cycle-mean is computed as the weighted average of total delay in a cycle divided by the number of tokens in it.

The conversion to HSDF from an SDF graph may result in an explosion in the number of nodes [PL95]. The number of nodes in the corresponding HSDF graph for an SDF graph is determined by its repetition vector. There are examples of real-applications (H.263 in this case), where an SDF model requires only 4 nodes and an HSDF model of the same application has 4754 nodes [Stu07]. This makes the above approach very infeasible for many multimedia applications. Lately, techniques have been presented that operate on SDF graphs directly [GGS$^+$06, Gha08]. These techniques essentially simulate the SDF execution and identify when a steady-state is reached by comparing previous states with current state. Even though the theoretical bound is high, the experimental results show that these techniques can compute the throughput of many multimedia applications within milliseconds.

A tool called SDF^3 has been written and is available on-line for use by the research community [SGB06a, SDF09]. Beside being able to generate random SDF graphs with specified properties, it can also compute throughput of SDF graphs easily. It allows to visualize these graphs as well, and compute other performance properties. The same tool was used for throughput computation and graph generation in many experiments conducted in this thesis.

However, the above techniques only work on a particular execution time of actors. If there is any change in the actor execution time, the entire analysis has to be repeated. Recently, a technique has been proposed in [GGBS08] that allows variable execution time. This technique computes equations that limit the application period, for a given range of actor execution times. When the exact actor execution time is known, these equations can be evaluated to compute the actual period of the application. This idea is used in Chapter 3 to compute throughput of applications.

It should be mentioned that the techniques mentioned here do not take into account resource contention and essentially assume that infinite resources are available, except SDF^3. SDF^3 also takes resource contention into account but is limited to preemptive systems. Before we see how throughput can be computed when considering limited computation resources, we review the basic techniques used for scheduling dataflow graphs.

2.5 Scheduling Techniques for Dataflow Graphs

One of the key aspects in designing multiprocessor systems from any MoC is scheduling. Compile-time scheduling promises near-optimal performance at low cost for final system, but is only suitable for static applications. Run-time scheduling can address a wider variety of applications, at greater system cost. Scheduling techniques can be classified in a number of categories based on which decisions are made at compile time (also known as design-time) and which decisions are made at run-time. [LH89, SB00]. There are three main decisions when scheduling tasks (or actors) on a processor:[2] 1) which tasks to **assign** to a given processor, 2) what is the **order** of these tasks on it and 3) what is the **timing** of these tasks. We consider four different types of schedulers.

[2]When considering mapping of channels on the network, there are many more categories.

1. The first one is **fully static** where everything is decided at compile time and the processor has to simply execute the tasks. This approach has traditionally been used extensively for DSP applications due to their repetitive and constant resource requirement. This is also good for systems where guarantees are more important and (potential) speed-up from earlier execution is not desired. Further, the run-time scheduler becomes very simple since it does not need to check for availability of data and simply executes the scheduled actors at respective times. However, this mechanism is completely static, and cannot handle any dynamism in the system like run-time addition of applications, or any unexpectedly higher execution time for a particular iteration.

2. The second type is **self-timed**, where the assignment and ordering is already done at compile time. However, the exact time for firing of actors is determined at run-time, depending on the availability of data. Self-timed scheduling is more suitable for cases when the execution time of tasks may be data-dependent, but the variation is not very large. This can often result in speed-up of applications as compared to analysis at design-time, provided the worst-case execution time estimates are used for analyzing the application performance. Since earlier arrival of data cannot result in later production of data, the performance bounds computed at compile-time are preserved. However, this also implies that the schedule may become non-work-conserving, i.e. that a task may be waiting on a processor, while the processor is sitting idle waiting for the task in order.

3. The third type is **static assignment**, where the mapping is already fixed at compile time, but the ordering and timing is done at run-time by the scheduler. This allows the schedule to become work-conserving and perhaps achieve a higher overall throughput in the system. However, it might also result in a lower overall throughput since the bounds cannot be computed at compile-time (or are not preserved in this scheduler). This scheduling is most applicable for systems where applications have a large variation in execution time. While for a single application, the order is still imposed by the data-dependency among tasks and makes self-timed more suitable, for multiple applications the high variation in execution time, makes it infeasible to enforce the static-order.

4. The last one is called **fully dynamic** where mapping, ordering and timing are all done at run-time. This gives full freedom to the scheduler, and the tasks are assigned to an idle processor as soon as they are ready. This scheduler also allows for task migration. This may result in a yet higher throughput, since this tries to maximize the resource utilization and minimize the idle time, albeit at the cost of performance guarantee. It should be noted that run-time assignment also involves a (potentially) higher overhead in data movement. When the assignment is fixed at compile time, the

Table 2.1: The time which the scheduling activities "assignment", "ordering", and "timing" are performed is shown for four classes of schedulers. The scheduling activities are listed on top and the strategies on the left [LH89].

Scheduler	Assignment	Ordering	Timing	Work-conserving
Fully static	compile	compile	compile	no
Self timed	compile	compile	run	no
Static assignment	compile	run	run	processor-level
Fully dynamic	run	run	run	yes

task knows the processor to which the receiving actor is mapped *apriori*.

These four scheduling mechanisms are summarized in Table 2.1. As we move from fully-static to fully-dynamic scheduler, the run-time scheduling activity (and correspondingly overhead) increases. However, this also makes it more robust for handling dynamism in the system. The last column shows the work-conserving nature of the schedulers. A fully-static scheduler is non-conserving since the exact time and order of firing (task execution) is fixed. The self-timed schedule is work-conserving only when at most one task is mapped on one processor, while the static-assignment is also work-conserving for multiple applications. However, in static assignment if we consider the whole system (i.e. multiple processors), it may not be work-conserving, since tasks may be waiting to execute on a particular processor, while other processors may be idle.

In a homogeneous system, there is naturally more freedom to choose which task to assign to a particular processor instance, since all processors are identical. On the contrary, in a heterogeneous system this freedom is limited by which processors can be used for executing a particular task. When only one processor is available for a particular task, the mapping is inherently dictated by this limitation. For a complete heterogeneous platform, a scheduler is generally not fully dynamic, unless a task is allowed to be mapped on different types of processors. However, even in those cases, assignment is usually fixed (or chosen) at compile time. Further, the execution time for multimedia applications is generally highly variant making a fully-static scheduler often infeasible. Designers therefore have to make a choice between a self-timed or a static-assignment schedule. The only choice left is essentially regarding the ordering of tasks on a processor.

In the next section, we shall see how to analyze performance of multiple applications executing on a multiprocessor platform for both self-timed and static-assignment scheduler. Since the only difference in these two schedulers is the time at which ordering of actors is done, we shall refer to self-timed and static-assignment scheduler as **static-ordering** and **dynamic-ordering** scheduler respectively for easy differentiation.

2.6 Analyzing Application Performance on Hardware

In Section 2.4, we assumed we have infinite computing and communication resources. Clearly, this is not a valid assumption. Often processors are shared, not just among tasks of one application, but also with other applications. We first see how we can model this resource contention for a single application, and later for multiple applications.

We start with considering an HSDF graph with constant execution times to illustrate that even for HSDF graphs it is already complicated. In [BKKB02], the authors propose to analyze performance of a *single application* modeled as an HSDF graph mapped on a multi-processor system by modeling dependencies of resources by adding extra edges to the graph. Adding these extra edges enforces a strict order among the actors mapped on the same processor. Since the processor dependency is now modeled in the graph itself, we can simply compute the maximum throughput possible of the graph, and that corresponds to the maximum performance the application can achieve on the multiprocessor platform.

Unfortunately, this approach does not scale when we move on to the SDF model of an application. Converting an SDF model to an HSDF model can potentially result in a large number of actors in the corresponding HSDF graph. Further, adding such resource dependency edges essentially enforces a static-order among actors mapped on a particular processor. While in some cases, only one order is possible (due to natural data dependency among those actors), in some cases the number of different orders is also very high. Further, different orders may result in different overall throughput of the application. This becomes even worse when we consider multiple applications. This is shown by means of an example in the following sub-section.

2.6.1 Static Order Analysis

In this sub-section, we look at how application performance can be computed using static-order scheduler, where both processor assignment and ordering is done at compile-time. We show that when multiple applications are mapped on multiple processors sharing them, it is 1) difficult to make a static schedule, 2) time-consuming to analyze application performance given a schedule, and 3) infeasible to explore the entire scheduling space to find one that gives the best performance for all applications.

Three application graphs – A, B and C, are shown in Figure 2.7. Each is an HSDF with three actors. Let us assume actors T_i are mapped on processing node P_i where T_i refers to a_i, b_i and c_i for $i = 1, 2, 3$. This contention for resources is shown by the dotted arrows in Figure 2.8. Clearly, by putting these dotted arrows, we have fixed the actor-order for each processor node. Figure 2.8 shows just one such possibility when the dotted arrows are used to combine the three task graphs. Extra tokens have been inserted in these dotted edges to indicate the initial state of arbiters on each processor. The tokens indicating processor contention are

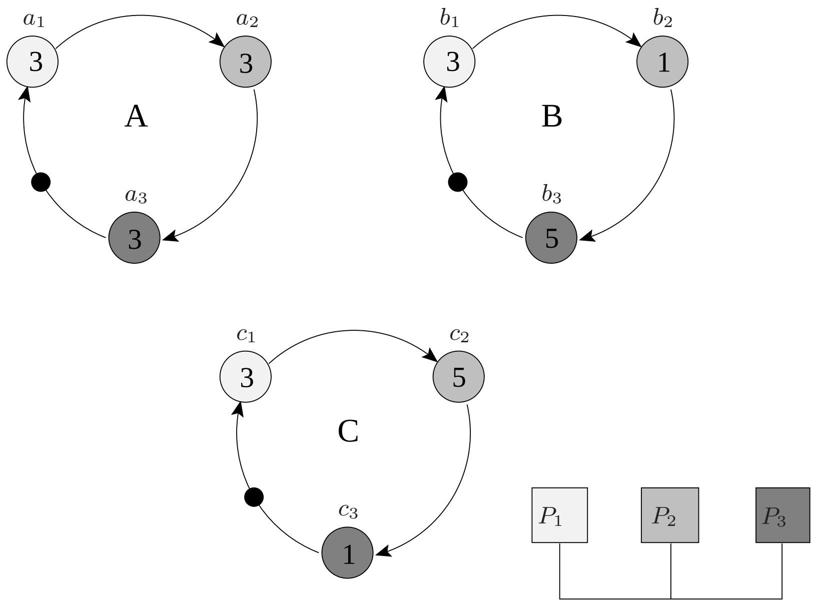


Figure 2.7: Example of a system with 3 different applications mapped on a 3-processor platform.

shown in gray, while the regular data tokens are shown in black. Clearly, this is only possible if each task is required to be run an equal number of times. If the rates of each task are not the same, we need to introduce multiple copies of actors to achieve the required ratio, thereby increasing analysis complexity.

When throughput analysis is done for this complete graph, we obtain a mean cycle count of 11. The bold arrows represent the edges that limit the throughput. The corresponding schedule is also shown. One actor of each application is ready to fire at instant t_0. However, only a_1 can execute since it is the only one with a token on all its incoming edges. We find that the graph soon settles into the periodic schedule of 11 clock cycles. This period is denoted in the schedule diagram of Figure 2.8 between the time instant t_1 and t_2.

Figure 2.9 shows just another of the many possibilities for ordering the actors of the complete HSDF. Interestingly, the mean cycle count for this graph is 10, as indicated by the bold arrows. In this case, the schedule starts repeating after time t_1, and the steady state length is 20 clock cycles, as indicated by difference in time instants t_1 and t_2. However, since two iterations for each application are completed, the average period is only 10 clock cycles.

From arbitration point of view, if application graphs are analyzed in isolation, there seems to be no reason to prefer actor b_1 or c_1 after a_1 has finished executing on P_1. There is at least a delay of 6 clock cycles before a_1 needs P_1

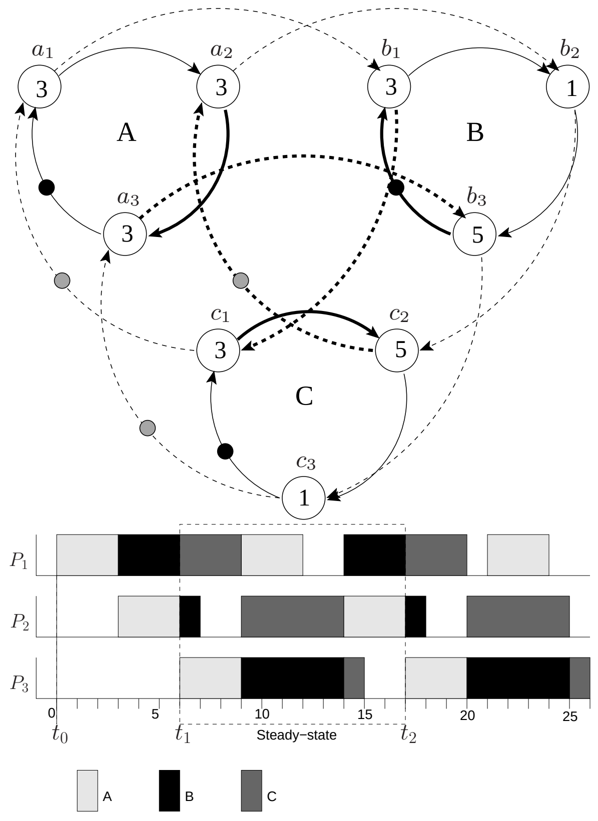


Figure 2.8: Graph with clockwise schedule (static) gives MCM of 11 cycles. The critical cycle is shown in bold.

again. Also, since b_1 and c_1 take only 3 clock cycles each, 6 clock cycles are enough to finish their execution. Further both are ready to be fired, and will not cause any delay. Thus, the local information about an application and the actors that need a processor resource does not easily dictate preference of one task over another. However, as we see in this example, executing c_1 is indeed better for the overall performance. Computing a static order relies on the global information and produces the optimal performance. This becomes a serious problem when considering MPSoC platforms, since constructing the overall HSDF graph and then computing its throughput is very compute intensive. Further, this is not suitable for dynamic applications. A small change in execution time may change the optimal schedule.

The number of possibilities for constructing the HSDF from individual graphs is very large. In fact, if one tries to combine g graphs of say a actors, scheduled

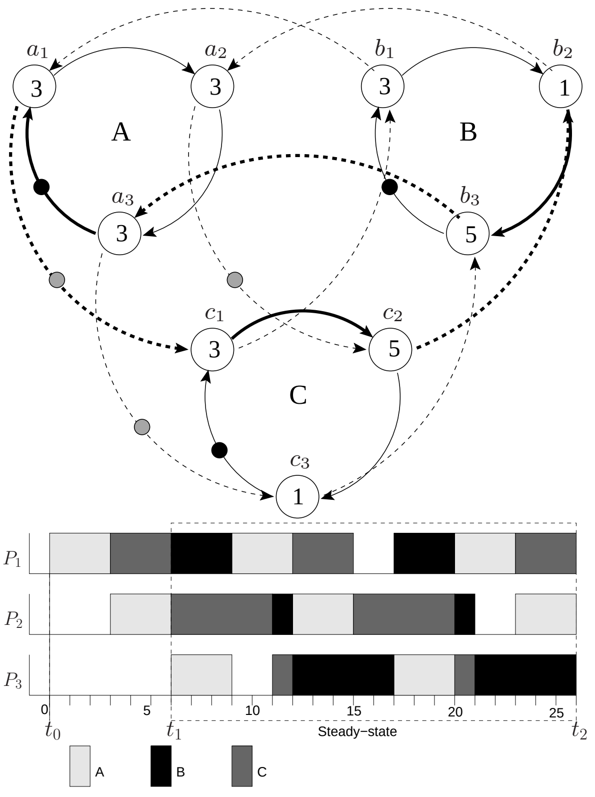


Figure 2.9: Graph with anti-clockwise schedule (static) gives MCM of 10 cycles. The critical cycle is shown in bold. Here two iterations are carried out in one steady-state iteration.

in total on a processors, there are $((g-1)!)^a$ unique combinations, each with a different actor ordering, for only *single* occurrence of each application actor. (Each processor has g actors to schedule, and therefore $(g-1)!$ unique orderings on a single processor. This leads to $((g-1)!)^a$ unique combinations, since ordering on each processor is independent of ordering on another.) To get an idea of vastness of this number, if there are 5 graphs with 10 actors each we get 24^{10} or close to $6.34 \cdot 10^{13}$ possible combinations. If each computation would take only 1ms to compute, 2009 years are needed to evaluate all possibilities. This is only considering the cases with equal rates for each application, and only for HSDF graphs. A typical SDF graph with different execution rates would only make the problem even more infeasible, since the transformation to HSDF may yield many actor copies. An exhaustive search through all the graphs to compute optimal static order is simply not feasible.

Deadlock Analysis

Deadlock avoidance and detection is an important concern when applications may be activated dynamically. Applications modeled as (H)SDF graphs can be analyzed for deadlock occurrence within an application. However, deadlock detection and avoidance between multiple applications is not so easy. When static order is being used, every new use-case requires a new schedule to be loaded into the kernel. A naive reconfiguration strategy can easily send the system into deadlock. This is demonstrated with an example in Figure 2.10.

Say actors a_2 and b_3 are running in the system on P_2 and P_3 respectively. Further assume that static order for each processor currently is $A \rightarrow B$ when only these two applications are active, and with a third application C, $A \rightarrow B \rightarrow C$ for each node. When application C is activated, it gets P_1 since it is idle. Let us see what happens to P_2: a_2 is executing on it and it is then assigned to b_2. P_3 is assigned to c_3 after b_3 is done. Thus, after each actor is finished executing on its currently assigned processor, we get a_3 waiting for P_3 that is assigned to task c_3, b_1 waiting for P_1 which is assigned to a_1, and c_2 waiting for P_2, which is assigned to b_2.

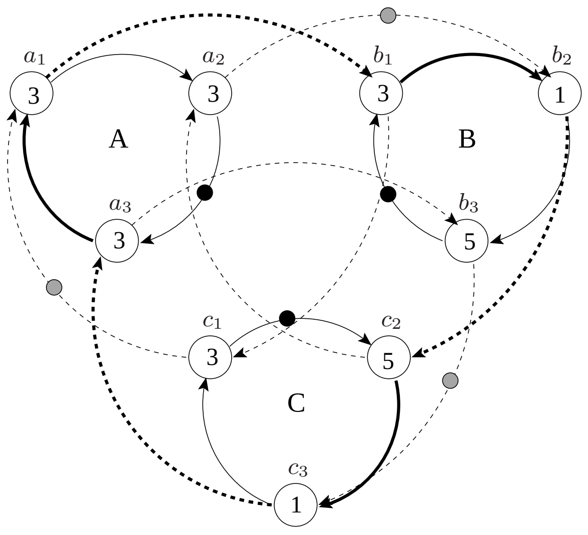


Figure 2.10: Deadlock situation when a new job, C arrives in the system. A cycle $a_1, b_1, b_2, c_2, c_3, a_3, a_1$ is created without any token in it.

Looking at Figure 2.10, it is easy to understand why the system goes into a deadlock. The figure shows the state when each actor is waiting for a resource and not able to execute. The tokens in the individual sub-graph show which actor is ready to fire, and the token on the dotted edge represents which resource is available to the application. In order for an actor to fire, the token should be present on all its incoming edges – in this case both on the incoming dotted edge and the solid edge. It can be further noted that a cycle is formed without any token in it. This is clearly a situation of deadlock [KM66] since the actors on this cycle will never be enabled. This cycle is shown in Figure 2.10 in bold edges. It is possible to take special measure to check and prevent the system from going into such deadlock. This, however, implies extra overhead at both compile-time and run-time. The application may also have to be delayed before it can be admitted into the system.

We can therefore conclude that computing a static order for multiple applications is very compute intensive and infeasible. Further, the performance we obtain may not be optimal. However, the advantage of this approach is that we are guaranteed to achieve the performance that is analyzed for *any* static order at design-time *provided the worst-case execution time estimates are correct.*

2.6.2 Dynamic Order Analysis

In this sub-section, we look at static-assignment scheduler, where only processor assignment is done at compile-time and the ordering is done at run-time. First-come-first-serve (FCFS) falls under this category. Another arbiter that we propose here in this category is round-robin-with-skipping (RRWS). In RRWS, a recommended order is specified, but the actors can be skipped over if they are not ready when the processor becomes idle. This is similar to the fairness arbiter proposed by Gao in 1983 [Gao83]. However, in that scheduler, all actors have equal weight. In RRWS, multiple instances of an actor can be scheduled in one cycle to provide an easy rate control mechanism.

The price a system-designer has to pay when using dynamic scheduling is the difficulty in determining application performance. Analyzing application performance when multiple applications are sharing multiprocessor platform is not easy. An approach that models resource contention by computing **worst-case-response-time** for TDMA scheduling (requires preemption) has been analyzed in [BHM$^+$05]. This analysis also requires limited information from the other SDFGs, but gives a very conservative bound that may be too pessimistic. As the number of applications increases, the minimum performance bound decreases much more than the average case performance. Further, this approach assumes a preemptive system. A similar worst-case analysis approach for round-robin is presented in [Hoe04], which also works on non-preemptive systems, but suffers from the same problem of lack of scalability.

Let us revisit the example in Figure 2.7. Since 3 actors are mapped on each processor, an actor may need to wait when it is ready to be executed at a processor. The maximum waiting time for a particular actor can be computed by considering the *critical instant* as defined by Liu and Layland [LL73]. The **critical instant** for an actor is defined as an instant at which a request for that actor has the largest response time. The **response time** is defined as the sum of an actor's waiting time and its execution time. If we take worst case execution time, this can be translated as the instant at which we have the largest waiting time. For dynamic scheduling mechanisms, it occurs when an actor becomes ready just after all the other actors, and therefore has to wait for all the other actors. Thus, the total waiting time is equal to the sum of processing times of all the other actors on that particular node and given by the following equation.

$$t_{wait}(T_{ij}) = \sum_{k=1, k \neq i}^{m} t_{exec}(T_{kj}) \tag{2.1}$$

Here $t_{exec}(T_{ij})$ denotes the execution time of actor T_{ij}, i.e. actor of task T_i mapped on processor j. This leads to a waiting time of 6 time units as shown in Figure 2.11. An extra node has been added for each 'real' node to depict the waiting time (WT_a_i). This suggests that each application will take 27 time units in the worst case to finish execution. This is the maximum period that can be obtained for applications in the system, and is therefore guaranteed. However, as we have seen in the earlier analysis, the applications will probably settle for a period of 10 or 11 cycles depending on the arbitration decisions made by the scheduler. Thus, the bound provided by this analysis is about two to three times higher than real performance.

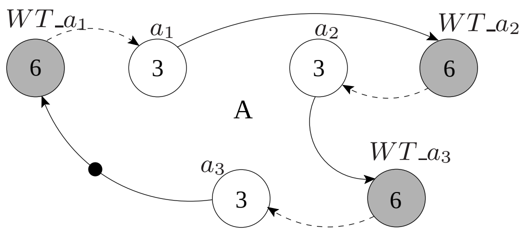


Figure 2.11: Modeling worst case waiting time for application A in Figure 2.7.

The deadlock situation shown in Figure 2.10 can be avoided quite easily by using dynamic-order scheduling. Clearly, for FCFS, it is not an issue since resources are never blocked for non-ready actors. For RRWS, when the system enters into a deadlock, the arbiter would simply skip to the actor that is ready to execute. Thus, processors 1, 2 and 3 are reassigned to B, C and A as shown in Table 2.2. Further, an application can be activated at any point in time without worrying about deadlock. In dynamic scheduling, there can never be a deadlock due to dependency on processing resources for atomic non-preemptive systems.

Table 2.2: Table showing the deadlock condition in Figure 2.10.

Node	Assigned to	Task waiting	Reassigned in RRWS
P1	A	B	B
P2	B	C	C
P3	C	A	A

2.7 Composability

As highlighted in Chapter 1, one of the key challenges when designing multimedia systems is dealing with multiple applications. For example, a mobile phone supports various applications that can be active at the same time, such as listening to mp3 music, typing an sms and downloading some java application in the background. Evaluating resource requirements for each of these cases can be quite a challenge even at design time, let alone at run time. When designing a system, it is quite useful to be able to estimate resource requirements early in the design phase. Design managers often have to negotiate with the product divisions for the overall resources needed for the system. These estimates are mostly on a higher level, and the managers usually like to adopt a *spread-sheet* approach for computing it. As we see in this section, it is often not possible to use this view.

We define *composability* as mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation, as far as possible. Note that this is different from what has been defined in literature by Kopetz [KO02, KS03]. Composability as defined by Kopetz is *integration of a whole system from well-specified and pre-tested sub-systems without unintended side-effects.* The key difference in this definition and our definition is that composability as defined by Kopetz is a property of a system such that the performance of applications in isolation and running concurrently with other applications is the same. For example, say we have a system with 10 applications, each with only one task and all mapped on the same processor. Let us further assume that all tasks take 100 time units to execute in isolation. According to the definition of Kopetz, it will also take 100 time units when running with the other tasks. This can only be achieved in two ways.

1. We can assume complete *virtualization* of resources, and that each application gets one-tenth of processor resources. This implies that we only use one-tenth of the resources when only one application is active. Further, to achieve complete virtualization, the processor has to be preempted and its context has to be switched every single cycle[3].

2. We consider a worst-case schedule in which all applications are scheduled, and the total execution time of all applications is 100 time units. Thus, if a particular application is not active, the processor simply waits for that

[3]This could be relaxed a bit, depending on the observability of the system.

many time units as it is scheduled for. This again leads to under-utilization of the processor resources. Besides, if any application takes more time, then the system may collapse.

Clearly, this implies that we cannot harness the full processing capability. In a typical system, we would want to use this compute power to deliver a better quality-of-service for an application when possible. We want to let the system execute as many applications as possible with the current resource availability, and let applications achieve their best behaviour possible in the given use-case. Thus, in the example with 10 applications, if each application can run in 10 time units in isolation, it might take 100 time units when running concurrently with all the other applications. We would like to predict the application properties given the application mix of the system, with as little information from other applications as possible.

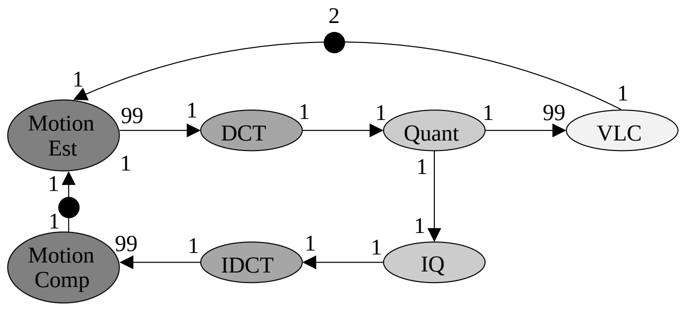
Some of the things we would like to analyze are for example, deadlock occurrence, and application performance. Clearly, since there is more than one application mapped on a multi-processor system, there will be contention for the resources. Due to this contention, the throughput analyzed for an application in isolation is not always achievable when the application runs together with other applications. We see how different levels of information from other applications affect analysis results in the next sub-section.

2.7.1 Performance Estimation

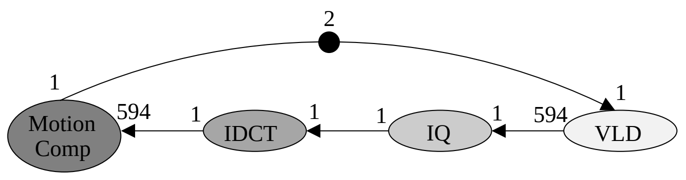
Let us consider a scenario of video-conferencing in a hand-held device. Figure 2.12 shows SDF graphs for both H263 encoding and decoding applications. The encoder model is based on the SDF graph presented in [OH04], and the decoder model is based on [Stu07][4]. The video-stream assumed for the example is of QCIF resolution that has 99 macro-blocks to process, as indicated by the rates on the edges. Both encoding and decoding have an actor that works on variable length (VLC and VLD respectively), quantization (Quant and IQ respectively), and discrete cosine transform (DCT and IDCT respectively). Since we are considering a heterogeneous system, the processor responsible for an actor in encoding process is usually responsible for the corresponding decoding actor. When the encoding and decoding are done concurrently, the DCT and IDCT are likely to be executed on the same processor, since that processor is probably more suited for cosine transforms. This resource dependency in the encoder and decoder models is shown by shading in Figure 2.12. Thus, the resource dependency in encoding and decoding is exactly reversed. A similar situation happens during decoding and encoding of an audio stream as well.

A simple example is shown in Figure 2.13 to illustrate the same behaviour as presented above. The figure shows an example of two application graphs A and B with three actors each, mapped on a 3-processor system. Actors a_1 and b_1 are

[4]The self-edges are removed for simplicity.



(a) SDF model of H263 encoder



(b) SDF model of H263 decoder

Figure 2.12: SDF graphs of H263 encoder and decoder.

mapped on p_1, a_2 and b_2 are mapped on p_2, and a_3 and b_3 are mapped on p_3. Each actor takes 100 clock cycles to execute. While both applications A and B might look similar, the dependency in A is anti-clockwise and in B clockwise to highlight the situation in the above example of simultaneous H263 encoding and decoding.

Let us try to *add* the resource requirement of actors and applications, and try to reason about their behaviour when they are executing concurrently. Each processor has two actors mapped; each actor requires 100 time units. If we limit the information to only actor-level, we can conclude that one iteration of each a_1 and b_1 can be done in a total of 200 time units on processor P_1, and the same holds for processors P_2 and P_3. Thus, if we consider a total of 3 million time units, each application should finish 15,000 iterations, leading to 30,000 iterations in total. If we now consider the graph-level local information only, then we quickly realize that since there is only one initial token, the minimum period of the applications is 300. Thus, each application can finish 10,000 iterations in 3 million time units. As it turns out, none of these two estimates are achievable.

Let us now increase the information we use to analyze the application performance. We consider the worst-case response time as defined in Equation 2.1 for each actor. This gives us an upper bound of 200 time units for each actor. If we now use this to compute our application period, we obtain 600 time units for each application. This translates to 5,000 iterations per application in 3 million time units. This is the guaranteed lower bound of performance. If we go one

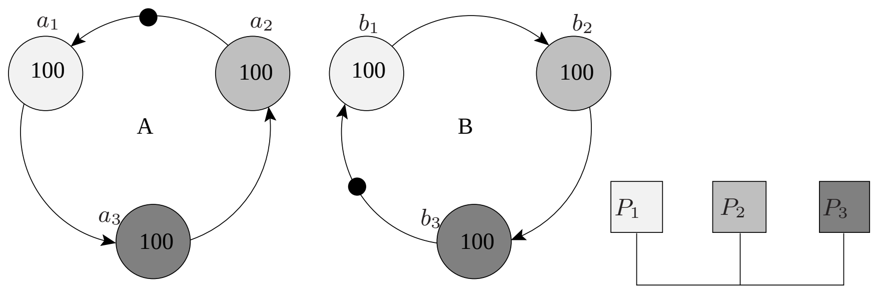


Figure 2.13: Two applications running on same platform and sharing resources.

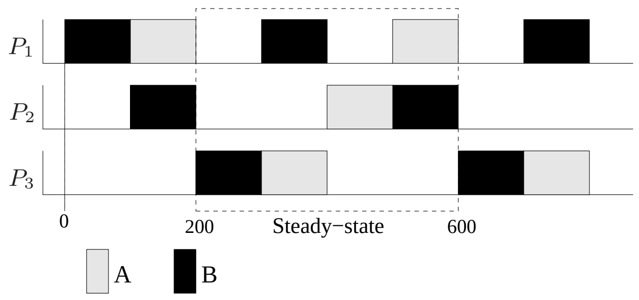


Figure 2.14: Static-order schedule of applications in Figure 2.13 executing concurrently.

stage further, and try to analyze the full schedule of this two-application system by making a static schedule, we obtain a schedule with a steady-state of 400 time units in which each application completes one iteration. The corresponding schedule is shown in Figure 2.14. Unlike the earlier predictions, this performance is indeed what the applications achieve. They will both complete one iteration every 400 time units. If we consider dynamic ordering and let the applications run on their own, we might obtain the same order as in Figure 2.14, or we might get the order as shown in Figure 2.15. When the exact execution order is not specified, depending on the scheduling policy the performance may vary. If we consider a first-come-first-serve approach, it is hard to predict the exact performance since the actors have equal execution time and they arrive at the exact time. If we assume for some reason, application A is checked first, then application A will execute twice as often as B, and vice-versa. The schedule in Figure 2.15 assumes application B has preference when both are ready at the exact same time. The same behaviour is obtained if we consider round-robin approach with skipping. Interestingly, the number of combined application iterations are still 15,000 – the same as when static order is used.

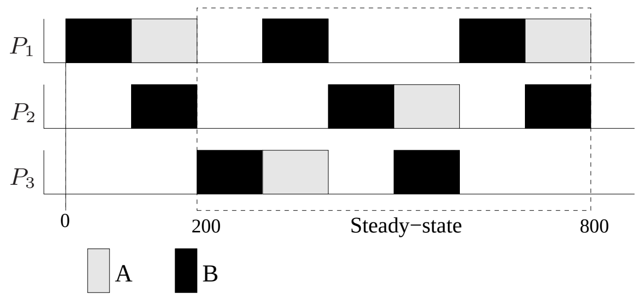


Figure 2.15: Schedule of applications in Figure 2.13 executing concurrently when B has priority.

Table 2.3: Estimating performance: iteration-count for each application in 3,000,000 time units

Appl.	Only actors	Only graph	WC Analysis (both graphs)	Static	RRWS/FCFS A pref	RRWS/FCFS B pref
A	15,000	10,000	5,000	7,500	10,000	5,000
B	15,000	10,000	5,000	7,500	5,000	10,000
Total	30,000	20,000	10,000	15,000	15,000	15,000
Proc Util	1.00	0.67	0.33	0.50	0.50	0.50

Table 2.3 shows how different estimating strategies can lead to different results. Some of the methods give a false indication of processing power, and are not achievable. For example, in the second column only the actor execution time is considered. This is a very naive approach and would be the easiest to estimate. It assumes that all the processing power that is available for each node is shared between the two actors equally. As we vary the information that is used to make the prediction, the performance prediction also varies. This example shows why composability needs to be examined. Individually each application takes 300 time units to complete an iteration and requires only a third of processor resources. However, when another application enters in the system, it is not possible to schedule both of them with their lowest period of 300 time units, even though the total request for a node is only two-third. Even when preemption is considered, only one application can achieve the period of 300 time units while the other of 600. The performance of the two applications in this case corresponds to the last two columns in Table 2.3. Thus, predicting application performance when executing concurrently with other applications is not very easy.

Table 2.4: Properties of Scheduling Strategies

Property		**Static Order**	**Dynamic Order**
Design time overhead	Calculating Schedules	- -	++
Run-time overhead	Memory requirement	-	++
	Scheduling overhead	++	+
Predictability	Throughput	++	- -
	Resource Utilization	+	-
New job admission	Admission criteria	++	- -
	Deadlock-free guarantee	-	++
	Reconfiguration overhead	-	+
Dynamism	Variable Execution time	-	+
	Handling new use-case	- -	++

2.8 Static vs Dynamic Ordering

Table 2.4 shows a summary of various performance parameters that we have considered, and how static-order and dynamic-order scheduling strategy performs considering these performance parameters. The static-order scheduling clearly has a higher design-time overhead of computing the static order for each use-case. The run-time scheduler needed for both static-order and dynamic-order schedulers is quite simple, since only a simple check is needed to see when the actor is active and ready to fire. The memory requirement for static scheduling is however, higher than that for a dynamic mechanism. As the number of applications increases, the total number of potential use-cases rises exponentially. For a system with 10 applications in which up to 4 can be active at the same time, there are approximately 400 possible combinations – and it grows exponentially as we increase the number of concurrently active applications. If static ordering is used, besides computing the schedule for all the use-cases at compile-time, one also has to be aware that they need to be stored at run-time. The scalability of using static scheduling for multiple applications is therefore limited.

Dynamic ordering is more scalable in this context. Clearly in FCFS, there is no such overhead as no schedule is computed beforehand. In RRWS, the easiest approach would be to store all actors for a processor in a schedule; when an application is not active, its actors are simply skipped, without causing any trouble for the scheduler. It should also be mentioned here that if an actor is required to be executed multiple number of times, one can simply add more copies of that actor in this list. In this way, RRWS can provide easy rate-control mechanism.

The static order approach certainly scores better than a dynamic one when it comes to predictability of throughput and resource utilization. Static-order approach is also better when it comes to admitting a new application in the system since the resource requirements prior and after admitting the application are known at design time. Therefore, a decision whether to accept it or not is easier to make. However, extra measures are needed to reconfigure the system properly so that the system does not go into deadlock as mentioned earlier.

A dynamic approach is able to handle dynamism better than static order since orders are computed based on the worst-case execution time. When the execution-time varies significantly, a static order is not able to benefit from early termination of a process. The biggest disadvantage of static order, however, lies in the fact that any change in the design, e.g. adding a use-case to the system or a new application, cannot be accommodated at run-time. The dynamic ordering is, therefore, more suitable for designing multimedia systems. In the following chapter, we show techniques to predict performance of multiple applications executing concurrently.

2.9 Conclusions

In this chapter, we began with motivating the need of having an application model. We discussed several models of computation that are available and generally used. Given our application requirements and strengths of the models, we chose the synchronous dataflow (SDF) graphs to model application. We provided a short introduction to SDF graphs and explained some important concepts relevant for this thesis, namely modeling auto-concurrency and modeling buffer-sizes on channels. We explained how performance characteristics of an SDF graph can be studied without considering hardware constraints.

The scheduling techniques used for dataflow analysis were discussed and classified depending on which of the three things – assignment, ordering, and timing – is done at compile-time and which at run-time. We highlighted two arbiters – static and dynamic ordering, which are more commonly used, and discussed how application performance can be analyzed considering hardware constraints for each of these arbiters.

We then highlighted the issue of *composability* – mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation, as far as possible. We demonstrated with a small, but realistic example, how predicting performance can be difficult when even small applications are considered. We also saw how arbitration plays a significant role in determining the application performance. We summarized the properties that are important for an arbiter in a multimedia system, and decided that considering the high dynamism in multimedia applications, the dynamic-ordering is more suitable.

CHAPTER 7

Conclusions and Future Work

In this chapter, the major conclusions from this thesis are presented, together with several issues that remain to be solved.

7.1 Conclusions

The design of multimedia platforms is becoming increasingly more complex. Modern multimedia systems need to support a large number of applications or functions in a single device. To achieve high performance in such systems, more and more processors are being integrated into a single chip to build Multi-Processor Systems-on-Chip (MPSoCs). The heterogeneity of such systems is also increasing with the use of specialized digital hardware, application domain processors and other IP (intellectual property) blocks on a single chip, since various standards and algorithms are to be supported. These embedded systems also need to meet timing and other *non-functional* constraints like low power and design area. Further, processors designed for multimedia applications (also known as *streaming processors*) often do not support preemption to keep costs low, making traditional analysis techniques unusable.

To achieve high performance in such systems, the limited computational resources must be shared. The concurrent execution of dynamic applications on shared resources causes interference. The fact that these applications do not always run concurrently only adds a new dimension to the design problem. We defined each such combination of applications executing concurrently as a *use-case*. Currently, companies often spend 60-70% of the product development cost in verifying all feasible use-cases. Having an efficient, but accurate analysis technique can significantly reduce this development cost. Since applications are often added to the system at run-time (for example, a mobile-phone user may download a Java application at run-time), a complete analysis at design-time is also not feasible. Existing techniques are unable to handle this dynamism, and the only solution left to the designer is to over-dimension the hardware by a large factor leading to increased area, cost and power.

In Chapter 3 of this thesis, a run-time performance prediction methodology is presented that can accurately and quickly predict the performance of multiple applications before they execute in the system. Synchronous data flow (SDF) graphs are used to model applications, since they fit well with characteristics of multimedia applications, and at the same time allow analysis of application performance. Further, their atomic execution requirement matches well with the non-preemptive nature of many streaming processors. While a lot of techniques are available to analyze performance of single applications, for multiple applications this task is a lot harder and little work has been done in this direction. This thesis presents one of the first attempts to analyze performance of multiple applications executing on heterogeneous non-preemptive multiprocessor platforms.

Our technique uses performance expressions computed off-line from the application specifications. A run-time iterative probabilistic analysis is used to estimate the time spent by tasks during the contention phase, and thereby predict the performance of applications. The average error in prediction using iterative probability is only 2% and the maximum error is 3%. Further, it takes about four to six iterations for the prediction to converge. The complexity and execution time of the algorithm is very low – it takes only 3ms to evaluate the performance of ten applications on a 50MHz embedded processor. This also proves the suitability of the technique for design space exploration on a regular desktop running at about 3GHz where the same analysis takes just 50 microseconds.

Further, we presented a design-flow for designing systems with multiple applications in Chapter 4. A hybrid approach is presented where the time-consuming application-specific computations are done at design-time, and in isolation from other applications, and the use-case-specific computations are performed at run-time. This allows easy addition of applications at run-time. Further, a run-time mechanism is presented to manage resources in a system. This ensures that no application starves due to another application. This mechanism enforces budgets and suspends applications if they achieve a higher performance than desired. This allows other applications to also achieve their desired performance. A resource manager (RM) is presented to manage computation and communication resources, and to achieve the above goals of performance prediction, admission control and budget enforcement. A case-study done with two application models – H263 and JPEG, shows the effectiveness of budget enforcement in achieving the desired performance of both applications.

With high consumer demands the time-to-market has become significantly lower. To cope with the complexity in designing such systems, a largely automated design-flow is needed that can generate systems from a high-level architectural description such that they are not error-prone and their design consumes less time. A highly automated flow – *MAMPS* (Multi-Application Multi-Processor Synthesis) is presented in Chapter 5, that synthesizes multi-processor platforms for multiple applications specified in the form of SDF graph models. The flow has been used to implement a tool that directly generates multi-processor designs for Xilinx FPGAs, complete with hardware and software descriptions. A case study done with the tool shows the effectiveness of the tool in which 24 design points were explored to compute the optimal buffer requirements of multiple applications in about 45 minutes including FPGA synthesis time.

One of the key design automation challenges that remain is fast exploration of software and hardware implementation alternatives with accurate performance evaluation, also known as design space exploration (DSE). A design methodology is presented in Chapter 6 to generate multiprocessor systems in a systematic and fully automated way for *multiple use-cases*. Techniques are presented to merge multiple use-cases into one hardware design to minimize cost and design time, making it well-suited for fast DSE of MPSoC systems. Heuristics to partition use-cases are also presented such that each partition can fit in an FPGA, and all use-cases can be catered for. A case study with mobile-phone applications shows an 11-fold reduction in DSE time.

7.2 Future Work

While this thesis presents solutions to various problems in analysis, design and management of multimedia multiprocessor systems, a number of issues remain to be solved. Some of these are listed below.

1. **Hard-real time support:** While the analysis techniques presented in this thesis are aimed towards multimedia systems that do not require a hard-bound on performance, they can easily be extended to support hard-real time applications as well. However, as has been mentioned earlier, that generally translates to a poor resource utilization. Techniques that can achieve high utilization and provide hard-bounds on performance still need to be developed. One option is to consider joining multiple application graphs with very few edges – only the minimum number needed to achieve rate-control – and then derive a static-order for that graph. This would achieve high-utilization and provide hard-bounds on performance. However, a potential drawback of this scheme is that for every possible use-case, a static order has to be stored, and care has to be taken that the system does not go into deadlock while switching between use-cases.

2. **Soft-real time guarantee:** The analysis technique presented in Chapter 3 is very accurate and fast. However, it does not provide any guarantee on the accuracy of the results. Even for soft-real time applications like video

and audio processing, some sort of measure of accuracy of results is desirable. Extending the probabilistic analysis to support this would increase the potential of this analysis technique. The designer wants to know how the applications would perform with the given resources, and accordingly increase or decrease the resources needed for system design.

3. **Model network and memory:** In this thesis, we have often ignored the contention for memory and network resources. Even though, in theory, this contention can be naturally modeled in SDF graph as well, it remains to be seen how well the technique applies, and how does it affect the performance of multiple applications. With a complete design-flow where the memory and network contention are also modeled, a designer will be able to make choices about the distribution of memory and network resources in the system, and about the allocation to different applications.

4. **SDF derivation:** Throughout this thesis, we have assumed that an SDF model of application has already been derived. In practice, this task can be very time consuming, and mostly manual. Automated extraction of parallel models from a sequential specification of an application is still an open problem. While a lot of tools are available to help in this derivation, most of them require extensive human interaction. This makes the design space exploration very time-consuming. The extraction of worst-case execution-times needed for each task is also very difficult. While static code analysis can provide very high estimates for task execution-time, profiling is only as accurate as the input sequence. This makes the compromise between an accurate and reasonable model rather difficult.

5. **Other models:** In this thesis, we have used synchronous dataflow for modeling applications. While these models are very good in expressing streaming behaviour, they are not always the best for expressing the dynamism in the applications. A number of other models are available that allow for dynamic behaviour in the model, e.g. CSDF, SADF and KPN. While CSDF is still static, it allows for different channel rates during different iterations of the actors. Developing analysis techniques for those models would help provide predictability to dynamic applications as well, and satisfy both the designers and the consumers.

6. **Achieving predictability in suspension:** In Section 4.3, a technique has been suggested to achieve predictability by using suspension. This technique is very powerful as it allows the designer to specify the desired application performance. By varying the time the system spends in each state, the performance of applications can be changed. While the basic idea has been outlined, the size of the time-wheel affects the performance significantly. The technique can also be adapted to support hard-real time tasks by using a conservative analysis, such as worst-case waiting-time analysis.

7. **Design space exploration heuristics:** In this thesis, we have concentrated on enabling design space exploration. Various techniques are provided for performance analysis of multiple applications to give feedback to the designer. Further, hardware design flow for rapid prototyping and performance evaluation is provided. However, we have not focused on heuristics to explore mapping options for optimizing performance and generating designs that satisfy the constraints of area and power.

8. **Optimizing the use-case partitions:** The use-case partitioning algorithm can be adapted to consider the relative frequency of the use of each use-case. The use-cases can first be sorted in the decreasing order of their use, and then the first-fit algorithm can be applied. The algorithm can therefore first pack all the most frequently used use-cases together in one hardware partition, thereby reducing the reconfiguration from one frequently used use-case into another. However, for an optimal solution of the partition problem, many other parameters need to be taken into account, for example reconfiguration time and average duration for each use-case. More research needs to be done in this to verify the suitability and effectiveness of this approach.

The above are some of the issues that need to be solved to take the analysis, design and management of multimedia multiprocessor systems into the next era.

Bibliography

[AAC$^+$03] A. Artieri, VD Alto, R. Chesson, M. Hopkins, and M.C. Rossi. Nomadik Open Multimedia Platform for Next-generation Mobile Devices. Technical Article TA305, ST Microelectronics, 2003.

[AB99] L. Abeni and G. Buttazzo. QoS guarantee using probabilistic deadlines. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems, 1999.*, pages 242–249, York, UK, 1999. IEEE.

[ABC$^+$06] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report 2006-183, Electrical Engineering and Computer Sciences, University of California Berkeley, December 2006.

[Ade08] S. Adee. the data: 37 years of moore's law. *Spectrum, IEEE*, 45(5):56–56, May 2008.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[BABP06] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 15–20, 3001

Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[Bar06] S. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 32(1):9–20, 2006.

[BCPV96] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[BELP96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, Feb 1996.

[BHM$^+$05] Marco Bekooij, Rob Hoes, Orlando Moreira, Peter Poplavko, Milan Pastrnak, Bart Mesman, Jan David Mol, Sander Stuijk, Valentin Gheorghita, and Jef van Meerbergen. Dataflow analysis for real-time embedded multiprocessor system design. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, pages 81–108. Springer, 2005.

[BKKB02] Neal Bambha, Vida Kianzad, Mukul Khandelia, and Shuvra S. Bhattacharyya. Intermediate representations for design automation of multiprocessor DSP systems. *Design Automation for Embedded Systems*, 7(4):307–323, 2002.

[BL93] J.T. Buck and E.A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, 1:429–432 vol.1, Apr 1993.

[Blu04] SIG Bluetooth. Bluetooth Specification Version 2.0+ EDR. *Bluetooth SIG Standard*, 2004.

[BML99] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee. Synthesis of Embedded Software from Synchronous Dataflow Specifications. *The Journal of VLSI Signal Processing*, 21(2):151–166, 1999.

[BMS96] T.F. Baldwin, D.S. McVoy, and C. Steinfield. *Convergence: Integrating Media, Information & Communication.* Sage Publications, 1996.

[Bor07] Shekhar Borkar. Thousand core chips: a technology perspective. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 746–749, New York, NY, USA, 2007. ACM.

[CDVS07] Johan Cockx, Kristof Denolf, Bart Vanhoof, and Richard Stahl. Sprint: a tool to generate concurrent transaction-level models from sequential code. *EURASIP Journal on Applied Signal Processing*, 2007(1):213–213, 2007.

[CES09] CES. Consumer electronics show. Available from: http://www.cesweb.org/, 2009.

[CK96] Y. Cai and M. C. Kong. Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems. *Algorithmica*, 15(6):572–599, 1996.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press, Cambridge, MA, USA, 2001.

[Cum03] Peter Cumming. *Winning the SoC Revolution*, chapter THE TI OMAP PLATFORM APPROACH TO SOC. Kluwer Academic Publishers, 2003.

[Das04] Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):385–418, 2004.

[DBC$^+$07] Kristof Denolf, Marco Bekooij, Johan Cockx, Diederik Verkest, , and Henk Corporaal. Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. *EURASIP Journal on Advances in Signal Processing*, 2007, 2007.

[DD86] S. Davari and S. K. Dhall. An on line algorithm for real-time tasks allocation. *IEEE Real-time Systems Symposium*, pages 194–200, 1986.

[dK02] E.A. de Kock. Multiprocessor mapping of process networks: a JPEG decoding case study. In *Proceedings of 15th ISSS*, pages 68–73, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[EGCMT70] Jr. E. G. Coffman, R. R. Muntz, and H. Trotter. Waiting time distributions for processor-sharing systems. *J. ACM*, 17(1):123–130, 1970.

[Gao83] G.R. Gao. *A pipelined code mapping scheme for static data flow computers.* PhD thesis, Massachusetts Institute of Technology, 1983.

[GDR05] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):414–421, 2005.

[GGBS08] A.H. Ghamarian, M.C.W. Geilen, T. Basten, and S. Stuijk. Parametric throughput analysis of synchronous data flow graphs. In *Design Automation and Test in Europe*, pages 116–121, Los Alamitos, CA, USA, 10-14 March 2008. IEEE Computer Society.

[GGS$^+$06] A.H. Ghamarian, M.C.W. Geilen, S. Stuijk, T. Basten, B.D. Theelen, M.R. Mousavi, A.J.M. Moonen, and M.J.G. Bekooij. Throughput Analysis of Synchronous Data Flow Graphs. In *Sixth International Conference on Application of Concurrency to System Design, (ACSD)*, pages 25–36, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[Gha08] A.H. Ghamarian. *Timing Analysis of Synchronous Data Flow Graphs*. PhD thesis, Eindhoven University of Technology, 2008.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman & Co., New York, NY, USA, 1979.

[Gon00] R.E. Gonzalez. Xtensa: a configurable and extensible processor. *Micro, IEEE*, 20(2):60–70, Mar/Apr 2000.

[Hal05] TR Halfhill. Busy Bees at Silicon Hive. *Microprocessor Report*, June 2005.

[Hiv09] Silicon Hive. Silicon hive. Available from: http://www.siliconhive.com, 2009.

[HM03] T. Halonen and J. Melero. *GSM, GPRS and EDGE Performance: Evolution Towards 3G/UMTS*. Wiley, 2003.

[HM08] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[Hoe04] R. Hoes. Predictable Dynamic Behavior in NoC-based MPSoC. Available from: www.es.ele.tue.nl/epicurus/, 2004.

[HOL09] HOL. Head-of-line blocking [Online]. Available from: `http://en.wikipedia.org/wiki/Head-of-line_blocking`, 2009.

[HQB07] Shaoxiong Hua, Gang Qu, and Shuvra S. Bhattacharyya. Probabilistic design of multimedia embedded systems. *Trans. on Embedded Computing Sys.*, 6(3):15, 2007.

[Int09] Intel. Intel press kit. Available from: http://www.intel.com/pressroom/kits/45nm/index.htm, 2009.

[JB06] Ahmed Jerraya and Iuliana Bacivarov. Performance Evaluation Methods for Multiprocessor System-on-Chip Design. In *EDA for IC System Design, Verification and Testing*, pages 6.1–6.14. Taylor and Francis, 2006.

[JSM91] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of 12th IEEE Real-Time Systems Symposium*, pages 129–139, 1991.

[JSRK05] Yujia Jin, Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. An automated exploration framework for FPGA-based soft multiprocessor systems. In *3rd CODES+ISSS*, pages 273–278, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[JW04] Ahmed Jerraya and Wayne Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, San Francisco, CA, 2004.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.

[KB97] JM Kahn and JR Barry. Wireless infrared communications. *Proceedings of the IEEE*, 85(2):265–298, 1997.

[KDH$^+$05] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor, 2005.

[KFH$^+$07] Akash Kumar, Shakith Fernando, Yajun Ha, Bart Mesman, and Henk Corporaal. Multi-processor System-level Synthesis for Multiple Applications on Platform FPGA. In *Proceedings of 17th International Conference on Field Programmable Logic and Applications*, pages 92–97, New York, NY, USA, 2007. IEEE Circuits and Systems Society.

[KFH$^+$08] Akash Kumar, Shakith Fernando, Yajun Ha, Bart Mesman, and Henk Corporaal. Multiprocessor systems synthesis for multiple use-cases of multiple applications on fpga. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):1–27, 2008.

[KHHC07] Akash Kumar, Andreas Hansson, Jos Huisken, and Henk Corporaal. An FPGA Design Flow For Reconfigurable Network-Based Multi-Processor Systems On Chip. In *Design, Automation and Test in Europe*, pages 117–122, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[KM66] Richard M Karp and Raymond E Miller. Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, nov 1966.

[KMC$^+$06] Akash Kumar, Bart Mesman, Henk Corporaal, Jef van Meerbergen, and Ha Yajun. Global Analysis of Resource Arbitration for MPSoC. In *DSD '06: Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pages 71–78, Washington, DC, USA, 2006. IEEE Computer Society.

[KMC$^+$07] Akash Kumar, Bart Mesman, Henk Corporaal, Bart Theelen, and Yajun Ha. A probabilistic approach to model resource contention for performance estimation of multi-featured media devices. In *Design Automation Conference*, pages 726–731, New York, NY, USA, 2007. ACM.

[KMCH08] Akash Kumar, Bart Mesma, Henk Corporaal, and Yajun Ha. Accurate Run-Time Performance Prediction for Multi-Application Multi-Processor Systems. Technical report, Tech. Univ. Eindhoven, `http://www.es.ele.tue.nl/esreports/`, 2008.

[KMN$^+$00] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[KMT$^+$06] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and H. Yajun. Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip. In *ESTMED '06: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 33–38, Washington, DC, USA, 2006. IEEE Computer Society.

[KMT$^+$08] Akash Kumar, Bart Mesman, Bart Theelen, Henk Corporaal, and Yajun Ha. Analyzing composability of applications on mpsoc platforms. *J. Syst. Archit.*, 54(3-4):369–383, 2008.

[KO02] H. Kopetz and R. Obermaisser. Temporal composability [real-time embedded systems]. *Computing & Control Engineering Journal*, 13(4):156–162, Aug 2002.

[KPBT06] S. Kunzli, F. Poletti, L. Benini, and L. Thiele. Combining Simulation and Formal Methods for System-level Performance Analysis. In *Design, Automation and Test in Europe*, volume 1, pages 1–6. IEEE, 2006.

[KS03] Hermann Kopetz and Neeraj Suri. Compositional design of rt systems: A conceptual basis for specification of linking interfaces. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium* *on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, page 51, Washington, DC, USA, 2003. IEEE Computer Society.

[Lee91] E.A. Lee. Consistency in dataflow graphs. *Parallel and Distributed Systems, IEEE Transactions on*, 2(2):223–235, Apr 1991.

[LH89] E.A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time dsp. In *Global Telecommunications Conference, 1989, and Exhibition. Communications Technology for the 1990s and Beyond. GLOBECOM '89., IEEE*, volume 2, pages 1279–1283, 1989.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[LM87] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, Feb 1987.

[LWAP94] R. Lauwereins, P. Wauters, M. Ade, and J.A. Peperstraete. Geometric parallelism and cyclo-static data flow in grape-ii. *Rapid System Prototyping, 1994. Shortening the Path from Specification to Prototype. Proceedings., Fifth International Workshop on*, pages 90–107, Jun 1994.

[LWM$^+$02] Rudy Lauwereins, Chun Wong, Paul Marchal, Johan Vounckx, Patrick David, Stefaan Himpe, Francky Catthoor, and Peng Yang. Managing dynamic concurrent tasks in embedded real-time multimedia systems. In *Proceedings of the 15th international symposium on System Synthesis*, pages 112–119, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[LYBJ01] D. Lyonnard, S. Yoo, A. Baghdadi, and A.A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Design Automation Conference*, pages 518–523, New York, NY, USA, 2001. ACM Press.

[MAM09] MAMPS. Multi-Application Multi-Processor Synthesis [Online]. Username: `todaes`, Password: `guest`. Available at: `http://www.es.ele.tue.nl/mamps/`, 2009.

[MCR$^+$06] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli. A methodology for mapping multiple use-cases onto networks on chips. In *Design, Automation and Test in Europe*, pages 118–123, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[MEP04] Sorin Manolache, Petru Eles, and Zebo Peng. Schedulability analysis of applications with stochastic task execution times. *Trans. on Embedded Computing Sys.*, 3(4):706–735, 2004.

[MMB07] Orlando Moreira, Jacob Jan-David Mol, and Marco Bekooij. Online resource management in a multiprocessor with a network-on-chip. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1557–1564, New York, NY, USA, 2007. ACM.

[MMZ$^+$02] R. Magoon, A. Molnar, J. Zachan, G. Hatcher, W. Rhee, S.S. Inc, and N. Beach. A single-chip quad-band (850/900/1800/1900 MHz) direct conversion GSM/GPRS RF transceiver with integrated VCOs and fractional-n synthesizer. *Solid-State Circuits, IEEE Journal of*, 37(12):1710–1720, 2002.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, 1965.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[NAE$^+$08] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest, and H. Corporaal. Run-time management of a mpsoc containing fpga fabric tiles. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):24–33, jan 2008.

[NAMV05] V. Nollet, P. Avasare, J-Y. Mignolet, and D. Verkest. Low cost task migration initiation in a heterogeneous mp-soc. In *Design, Automation and Test in Europe*, pages 252–253, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[Nol08] Vincent Nollet. *Run-time management for Future MPSoC Platforms.* PhD thesis, Eindhoven University of Technology, 2008.

[NSD06] H. Nikolov, T. Stefanov, and E. Deprettere. Multi-processor system design with ESPAM. In *Proceedings of the 4th CODES+ISSS*, pages 211–216, New York, NY, USA, 2006. ACM Press.

[NSD08] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):542–555, March 2008.

[OA03] J.A. De Oliveira and H. Van Antwerpen. *Winning the SoC Revolution*, chapter The Philips Nexperia digial video platforms. Kluwer Academic Publishers, 2003.

[Ody72] Magnavox Odyssey. World's first video game console. Available from: http://en.wikipedia.org/wiki/Magnavox_Odyssey, 1972.

[OH04] H. Oh and S. Ha. Fractional rate dataflow model for efficient code synthesis. *Journal of VLSI Signal Processing*, 37(1):41–51, May 2004.

[PD80] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8(6):25–33, 1980.

[Pet62] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Rheinisch-Westfälisches Institut f. instrumentelle Mathematik an d. Univ., 1962.

[Phi09] Philips. Royal philips. Available from: www.philips.com, 2009.

[PL95] J.L. Pino and E.A. Lee. Hierarchical static scheduling of dataflow graphs onto multipleprocessors. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 4, pages 2643–2646, Detroit, MI, USA, 1995. IEEE.

[PS309] PS3. Sony playstation 3. Available from: http://www.playstation.com/, 2009.

[PTB06] J. M. Paul, D. E. Thomas, and A. Bobrek. Scenario-oriented design for single-chip heterogeneous multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):868–880, August 2006.

[RJE03] K. Richter, M. Jersak, and R. Ernst. A formal approach to MPSoC performance verification. *Computer*, 36(4):60–67, 2003.

[Rob00] T.G. Robertazzi. *Computer Networks and Systems: Queueing Theory and Performance Evaluation.* Springer, 2000.

[Ros08] P.E. Ross. Why cpu frequency stalled. *Spectrum, IEEE*, 45(4):72–72, April 2008.

[Roz01] E. Roza. Systems-on-chip: what are the limits? *ELECTRONICS AND COMMUNICATION ENGINEERING JOURNAL*, 13(6):249–255, 2001.

[RVB07] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. Function level parallelism driven by data dependencies. *ACM SIGARCH Computer Architecture News*, 35(1):55–62, 2007.

[Sam09] Samsung. Samsung. Available from: http://www.samsung.com, 2009.

[SB00] S. Sriram and S.S. Bhattacharyya. *Embedded Multiprocessors; Scheduling and Synchronization.* Marcel Dekker, New York, NY, USA, 2000.

[SDF09] SDF3. SDF3: SDF For Free [Online]. Available at: `http://www.es.ele.tue.nl/sdf3/`, 2009.

[SGB06a] S. Stuijk, M. Geilen, and T. Basten. SDF3: SDF For Free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD).*, pages 276–278, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[SGB06b] S. Stuijk, M.C.W. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Design Automation Conference*, pages 899–904, New York, NY, USA, 2006. ACM Press.

[SKMC08] Ahsan Shabbir, Akash Kumar, Bart Mesman, and Henk Corporaal. Enabling mpsoc design space exploration on fpgas. In *Proceedings of International Multi Topic Conference (IMTIC)*, New York, NY, USA, 2008. Springer.

[SKMC09] Ahsan Shabbir, Akash Kumar, Bart Mesman, and Henk Corporaal. *Enabling MPSoC Design Space Exploration on FPGAs*, volume 20 of *Communications in Computer and Information Science*, chapter 44, pages 412–421. Springer Berlin Heidelberg, 2009.

[Son09] Sony. World of sony. Available from: http://www.sony.com, 2009.

[Stu07] S. Stuijk. *Predictable mapping of streaming applications on multiprocessors.* PhD thesis, Eindhoven University of Technology, 2007.

[SVCBS04] Alberto Sangiovanni-Vincentelli, Luca Carloni, Fernando De Bernardinis, and Marco Sgroi. Benefits and challenges for platform-based design. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 409–414, New York, NY, USA, 2004. ACM.

[SZT$^+$04] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette. System design using Kahn process networks: the Compaan/Laura approach. In *Design, Automation and Test in Europe*, pages 340–345, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[Tak62] L. Takacs. *Introduction to the Theory of Queues.* Greenwood Press, 1962.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of ISCAS 2000 Geneva.*, volume 4, pages 101–104, Geneva, Switzerland, 2000. IEEE.

[TCWCS92] E. Teruel, P. Chrzastowski-Wachtel, J. Colom, and M. Silva. On weighted t-systems. *Application and Theory of Petri Nets 1992*, pages 348–367, 1992.

[Ten09] Tensilica. Tensilica - the dataplane processor company. Available from: http://www.tensilica.com, 2009.

[TFG+07] B.D. Theelen, O. Florescu, M.C.W. Geilen, J. Huang, P.H.A. van der Putten, and J.P.M. Voeten. Software/Hardware Engineering with the Parallel Object-Oriented Specification Langauge. In *Proceedings of the Fifth ACM-IEEE International Conference on Formal Methods and Models for Codesign*, pages 139–148, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[TGB+06] B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita, and S. Stuijk. A Scenario-Aware Data Flow Model for Combined Long-Run Average and Worst-Case Performance Analysis. In *Proceedings of the International Conference on Formal Methods and Models for Co-Design*. IEEE Computer Society Press, 2006.

[TNS06] TNS. Tns research [Online]. Available from: `http://www.tns.lv/?lang=en&fullarticle=true&category=showuid&id=2288`, 2006.

[TW08] David Terr and Eric W. Weisstein. Symmetric polynomial. Available from: mathworld.wolfram.com/SymmetricPolynomial.html, 2008.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[Wik08] Wikipedia. Linear programming [Online]. Available from: `http://en.wikipedia.org/wiki/Linear_programming`, 2008.

[Wol04] W. Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41st DAC '04*, pages 681–685, 2004.

[WS94] Shlomo Weiss and James E. Smith. *IBM Power and PowerPC*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[Xil09] Xilinx. Xilinx Resource page [Online]. Available from: `http://www.xilinx.com`, 2009.

[ZF93] H. Zhang and D. Ferrari. Rate-controlled static-priority queueing. In *INFOCOM '93. Proceedings.Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future. IEEE*, pages 227–236, San Francisco, CA, USA, 1993.

Glossary

Acronyms and abbreviations

ASIC	Application specific integrated circuit
ASIP	Application specific instruction-set processor
BDF	Boolean dataflow
CF	Compact flash
CSDF	Cyclo static dataflow
DCT	Discrete cosine transform
DSE	Design space exploration
DSP	Digital signal processing
FCFS	First-come-first-serve
FIFO	First-in-first-out
FPGA	Field-programmable gate array
FSL	Fast simplex link
HSDFG	Homogeneous synchronous dataflow graph
IDCT	Inverse discrete cosine transform
IP	Intellectual property
JPEG	Joint Photographers Expert Group
KPN	Kahn process network
LUT	Lookup table
MAMPS	Multi-Application Multi-Processor Synthesis.
MB	Microblaze
MoC	Models of Computation
MCM	Maximum cycle mean
MPSoC	Multi-processor system-on-chip

POOSL	Parallel object oriented specification language
QoS	Quality-of-service
RAM	Random access memory
RCSP	Rate controlled static priority
RISC	Reduced instruction set computing
RM	Resource manager
RR	Round-robin
RRWS	Round-robin with skipping
RTL	Register transfer level
SADF	Scenario aware dataflow
SDF	Synchronous dataflow
SDFG	Synchronous dataflow graph
SMS	short messaging service
TDMA	Time-division multiple access
VLC	Variable length coding
VLD	Variable length decoding
VLIW	Very long instruction word
WCET	Worst case execution time
WCRT	Worst case response time
XML	Extensible markup language

Terminology and definitions

Actor	A program segment of an application modeled as a vertex of a graph that should be executed atomically.
Composability	Mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation, as far as possible.
Control token	Some information that controls the behaviour of actor. It can determine the rate of different ports in some MoC (say SADF and BDF), and the execution time in some other MoC (say SADF and KPN).
Critical Instant	The critical instant for an actor is defined as an instant at which a request for that actor has the largest response time.
Multimedia systems	Systems that use a combination of content forms like text, audio, video, pictures and animation to provide information or entertainment to the user.
Output actor	The last task in the execution of an application after whose execution one iteration of the application can be said to have been completed.
Rate	The number of tokens that need to be consumed (for input rate) or produced (for output rate) during an execution of an actor.

Reconfigurable platform	A piece of hardware that can be programmed or reconfigured at run-time to achieve the desired functionality.
Response time	The time an actor takes to respond once it is ready i.e. the sum of its waiting and its execution time.
Scenario	A mode of operation of a particular application. For example, an MPEG video stream may be decoding an I-frame or a B-frame or a P-frame. The resource requirement in each scenario may be very different.
Scheduling	Process of determining when and where a part of application is to be executed.
Task	A program segment of an application that is executed atomically.
Token	A data element that is consumed or produced during an actor-execution.
Use-case	This refers to a combination of applications that may be active concurrently. Each such combination is a new use-case.
Work-conserving schedule	This implies if there is work to be done (or task to be executed) on a processor, it will execute it and not wait for some other work (or task). A schedule is work-conserving when the processor is not idle as long as there is any task waiting to execute on the processor.

Curriculum Vitae

Akash Kumar was born in Bijnor, India on November 12, 1980. After finishing the middle high-school at the Dayawati Modi Academy in Rampur, India in 1996, he proceeded to Raffles Junior College, Singapore for his pre-university education. In 2002, he completed Bachelors in Computer Engineering (First Class Honours) from the National University of Singapore (NUS), and in 2004 he completed joint Masters in Technological Design (Embedded Systems) from Eindhoven University of Technology (TUe) and NUS.

In 2005, he began working towards his joint Ph.D. degree from TUe and NUS in the Electronic Systems group and Electical and Computer Engineering department respectively. His research was funded by STW within the PreMaDoNA project. It has led, among others, to several publications and this thesis.

List of Publications

Journals and Book Chapters

- Ahsan Shabbir, Akash Kumar, Bart Mesman and Henk Corporaal Enabling MPSoC Design Space Exploration on FPGAs. In *Wireless Networks, Information Processing and Systems, Communications in Computer and Information Science*, Vol. 20, pp. 412-421, ISSN: 1865-0929. Springer, 2009. doi:10.1007/978-3-540-89853-5_44.

- Akash Kumar, Shakith Fernando, Yajun Ha, Bart Mesman and Henk Corporaal. Multi-processor Systems Synthesis for Multiple Use-Cases of Multiple Applications on FPGA. In: *ACM Transactions on Design Automation of Electronic Systems.* Vol 13, Issue 3, July 2008, pp. 1-27, ISSN: 1084-4309. ACM, 2008. doi:10.1145/1367045.1367049.

- Akash Kumar, Bart Mesman, Bart Theelen, Henk Corporaal and Yajun Ha. Analyzing Composability of Applications on MPSoC Platforms. In *Journal of Systems Architecture.* Vol 54, Issues 3-4, March-April 2008, pp. 369-383. ISSN: 1383-7621. Elsevier B.V., 2007. doi:10.1016/j.sysarc.2007.10.002.

- Akash Kumar and Sergei Sawitzki. High-Throughput and Low-Power Reed Solomon Decoded for Ultra Wide Band. In *Intelligent Algorithms, Philips Research Book Series*, Vol 7, pp. 299-316, ISBN: 1-4020-4953-6. Springer, 2006. doi:10.1007/1-4020-4995-1_17.

- G. Mohan, K. Akash and M. Ashish. Efficient techniques for improved QoS performance in WDM optical burst switched networks. In *Computer*

Communications, Vol. 28, Issue 7, 2 May 2005, pp. 754-764. ISSN: 0140-3664. Science Direct, 2005. doi:10.1016/j.comcom.2004.10.007.

- G. Ciobanu, R. Desai, A. Kumar. Membrane systems and distributed computing. In *Membrane Computing, Lecture Notes in Computer Science*, Vol. 2597, pp. 187-202, ISSN: 0302-9743. Springer, 2003. doi:10.1007/3-540-36490-0.

Conference Papers

- Ahsan Shabbir, Akash Kumar, Bart Mesman and Henk Corporaal. Enabling MPSoC Design Space Exploration on FPGAs. In *Proceedings of International Multi Topic Conference (IMTIC)*, Apr 2008. Pakistan 2008. Springer.
- Akash Kumar and Kees van Berkel. Vectorization of Reed Solomon Decoding and Mapping on the EVP. In *Proceedings of Design Automation and Test in Europe (DATE)*, Mar 2008, pp.450-455. ISBN:978-3-9810801-3-1. Munich, Germany, 2008. IEEE Computer Society.
- Akash Kumar, Shakith Fernando, Yajun Ha, Bart Mesman, and Henk Corporaal. Multi-processor System-level Synthesis for Multiple Applications on Platform FPGA. In *Proceedings of Field Programmable Logic (FPL) Conference*, Aug 2007, pp. 92-97. ISBN: 1-4244-1060-6. Amsterdam, The Netherlands, 2007. IEEE Circuit and Systems Society.
- Akash Kumar, Bart Mesman, Bart Theelen, Henk Corporaal and Yajun Ha. A Probabilistic Approach to Model Resource Contention for Performance Estimation of Multi-featured Media Devices. In *Proceedings of Design Automation Conference (DAC)*, Jun 2007, pp. 726-731. ISBN: 978-1-59593-627-1. San Diego, USA, 2007. IEEE Computer Society.
- Akash Kumar, Andreas Hansson, Jos Huisken and Henk Corporaal An FPGA Design Flow for Reconfigurable Network-Based Multi-Processor Systems-on-Chip. In *Proceedings of Design Automation and Test in Europe (DATE)*, Apr 2007, pp. 117-122. ISBN: 978-3-9810801-2-4. Nice, France, 2007. IEEE Computer Society.
- Akash Kumar, Bart Mesman, Bart Theelen, Henk Corporaal and Yajun Ha. Resource Manager for Non-preemptive Heterogeneous Multiprocessor System-on-chip. In *Proceedings of the 4th Workshop on Embedded Systems for Real-Time Multimedia (Estimedia)*, Oct 2006, pp. 33-38. ISBN: 0-7803-9783-5. Seoul, Korea, 2006. IEEE Computer Society.
- Akash Kumar, Bart Mesman, Henk Corporaal, Jef van Meerbergen and Yajun Ha. Global Analysis of Resource Arbitration for MPSoC. In *Proceedings*

of the 9th Euromicro Conference on Digital Systems Design (DSD), Aug 2006. pp. 71-78. ISBN: 0-7695-2609-8. Dubrovnik, Croatia, 2006. IEEE Computer Society.

- Akash Kumar, Bart Theelen, Bart Mesman and Henk Corporaal. On Composability of MPSoC Applications. In *Advanced Computer Architecture and Compilation for Embedded Systems (ACACES)*, Jul 2006, pp. 149-152, ISBN: 90-382-0981-9. L'Aquila, Italy, 2006.

- Akash Kumar, Ido Ovadia, Jos Huisken, Henk Corporaal, Jef van Meerbergen and Yajun Ha. Reconfigurable Multi-Processor Network-on-Chip on FPGA. In *Proceedings of 12th Conference of the Advanced School for Computing and Imaging (ASCI).* Jun 2006, pp. 313-317, ISBN: 90-810-8491-7. Lommel, Belgium, 2006.

- Akash Kumar and Sergei Sawitzki. High-Throughput and Low-Power Architectures for Reed Solomon Decoder. In *Proceedings of the 39th Asilomar Conference on Signals, Systems, and Computers*, Oct 2005. pp. 990-994. ISBN: 1-4244-0132-1. Pacific Grove, U.S.A., 2005. IEEE Circuit and Systems Society.

- Akash Kumar and Sergei Sawitzki. High-Throughput and Low-Power Reed Solomon Decoded for Ultra Wide Band. In *Proceedings of Philips Symposium on Intelligent Algorithms*, Dec 2004. Philips High Tech Campus, Eindhoven, 2004.

- G. Mohan, M. Ashish, and K. Akash. Burst Scheduling Based on Time-slotting and Fragmentation in WDM Optical Burst Switched Networks. In *Proceedings of IASTED International Conference on Wireless and Optical Communications WOC*, July 2002, pp. 351-355. Banff, Canada.

Technical Reports

- Akash Kumar, Bart Mesman, Henk Corporaal and Yajun Ha. Accurate Run-time Performance Prediction for Multi-Application Multi-Processor Systems. ES Report ESR-2008-07. Jun 16, 2008. Eindhoven University of Technology.

- Akash Kumar, Bart Mesman, Henk Corporaal, Bart Theelen and Yajun Ha. A Probabilistic Approach to Model Resource Contention for Performance Estimation of Multi-featured Media Devices. ES Report ESR-2007-02. Mar 25, 2007. Eindhoven University of Technology.

- Akash Kumar. High-Throughput Reed Solomon Decoded for Ultra Wide Band. Masters Thesis, Dec 2004. National University of Singapore and Eindhoven University of Technology.

- Akash Kumar. Wavelength Channel Scheduling Using Fragmentation Approach in Optical Burst Switching Networks. Bachelors Thesis, June 2002. National University of Singapore.

Reader's Notes