

# SP@CE - An SP-based Programming Model for Consumer Electronics Streaming Applications <sup>\*</sup>

Ana Lucia Varbanescu<sup>1</sup>, Maik Nijhuis<sup>2</sup>, Arturo González- Escribano<sup>3</sup>,  
Henk Sips<sup>1</sup>, Herbert Bos<sup>2</sup>, and Henri Bal<sup>2</sup>

<sup>1</sup> Department of Computer Science, Delft University of Technology, The Netherlands

<sup>2</sup> Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

<sup>3</sup> Departamento Informatica, Universidad de Valladolid, Spain

**Abstract.** Consumer Electronics (CE) devices are becoming the favorite target platforms for multimedia streaming applications, but finding the right solutions for efficient programming, both in terms of development time and application performance is not trivial. In this context, we present SP@CE, a new programming model that tackles the challenge of designing and developing CE multimedia streaming applications. SP@CE is an extension of the Series-Parallel Contention (SPC) programming model. It is designed to balance the specific requirements we have identified for the CE streaming applications with the simplicity and efficiency of SPC. Thus, SP@CE allows programmers to design and implement event-aware dynamically reconfigurable streaming applications, exploiting both data and task parallelism in a straightforward manner. To enable the use of SP@CE, we have designed and prototyped a three-layer framework, that offers support in designing the SP application, optimizing the implementation based on a performance prediction loop, and executing the application on the target platform. For evaluating the entire system, we have used SP@CE to implement a set of real-life streaming applications and we present the results obtained by running them on a multi-processor system-on-chip (MPSoC) CE platform, namely the Wasabi/SpaceCAKE architecture from Philips. The experiments show that SP@CE does enable rapid application development, as it is easy to use, induces low overhead, offers high code reuse potential, and takes advantage of the inherent application parallelism.

## 1 Introduction

Ten years ago, the field of consumer electronics (CE) was limited to television, home hi-fi, and home appliances. Nowadays, it expands to include many other modern electronics fields, ranging from mobile phones to car navigation systems, from house security devices to interactive information displays. These systems spend most of their resources on processing complex multimedia, including video and sound playing, real-time animations, real-time information retrieval and presentation. The applications have to process continuous and virtually infinite flows of data, namely data streams, and they have to be able to run concurrently, to answer user-generated events and to reconfigure themselves on-demand.

To meet these challenges, we present SP@CE, a new programming model dedicated to streaming applications for multiprocessor consumer electronic devices. SP@CE is an extension of the Series-Parallel Contention (SPC) programming model. Besides the features inherited from SPC, like ease of programming, explicit parallelism, and predictability, SP@CE offers novel solutions for dealing with streaming and user-interaction, both essential to CE

---

<sup>\*</sup> This work is supported by the Dutch government's STW/PROGRESS project DES.6397.

applications. The *SP@CE framework* is a natural extension of the programming model, providing the user a productive tool for application design, performance evaluation, optimization and execution. In some of our previous work, we have presented that both Hinch, the SP@CE run-time system, optimized for multimedia streaming applications [1], and PAM-SoC, the performance predictor [2] that supports SP@CE's feedback loop. In this paper, we focus more on the requirements, the design and the novel features of the SP@CE programming model, pointing out the support the framework provides for them.

The paper is structured as follows: Section 2 presents the specifics of streaming applications and Section 3 discusses requirements identified as essential for consumer electronics applications. Section 4 details the SP@CE programming model, while Section 5 presents our experiments and their results. Section 6 discusses some related work, while Section 7 presents our conclusions and the future work directions.

## 2 Streaming Applications

A *streaming application* is a data-intensive application that has to read, process, and output streams of data. Building on the work in [3,4], we identify the following representative features of a streaming application:

- The application processes *data streams*, which are continuous, virtually infinite, with a given *data rate*. The elements of a data-stream are of the same type, and they exhibit low reusability: an element is useful/used for a limited (usually short) period of time and then discarded. The *active window* of a data stream contains the elements required for the current processing.
- Data processing is performed by *filters* or *components*. A component reads its input streams, processes the elements in their active windows, and outputs results in its output streams. Ideally, components are independent entities, implemented such that they allow concurrent execution, allowing to exploit the task-parallelism of the application.
- An application is a *component graph*, i.e., a collection of components connected by data streams. The application is executed as an implicit infinite loop. Each *application iteration* takes the time needed for the active window of the application input stream to be processed and written to the output stream. Depending on the filters organization and/or parallelization, the application may process data at higher or lower rates.
- The *control flow* of the application allows (1) taking different execution paths based on conditionals, and (2) reshaping the component graph. The interaction between the data flow and the control flow of the application must be specified and formalized.

While an up-to-date “Streaming Programming” paradigm is not yet entirely agreed upon - see the various definitions in [5, 6, 3], the consumer electronics industry demands dedicated, more productive and more efficient tools for such applications. The SP@CE framework is a possible answer to these demands.

## 3 CE platforms

An essential requirement for consumer electronics software is to be *reactive*, i.e., to be able to respond and manage user interaction. Thus, besides streaming, CE applications must feature event awareness and handling, dynamic reconfigurability and performance predictability. This section provides insights on these three specific aspects. To exemplify the concepts, we use a TV-like picture-in-picture (PiP) environment, where the user can dynamically control the number of pictures on display (show/remove a picture), as well as their positioning (move the picture on the screen).

### 3.1 Dynamic reconfigurability

*Dynamic application reconfigurability* is the ability of the system to modify the execution parameters of a running application without stopping and restarting it. For example, if the user decides to add a new picture in the PiP environment, the new addition should not affect the displaying of the previously visible pictures. At the implementation level, dynamic reconfigurability translates into the ability of the application to reconfigure itself transparently to the user. Typically, such a reconfiguration implies application graph restructuring by nodes and/or edges addition or removal. Furthermore, reconfigurability requires a dynamic resource scheduler, able to map the new application graph on the existing resources on-the-fly.

### 3.2 Event awareness and event handling

*Event awareness and handling* refer to the ability of an application to detect user requests and respond accordingly. In the PiP environment, if the user presses a button to add an extra picture, an external event for the application is generated. This event can interrupt the current processing at a suitable moment (event awareness) and the application reconfigures according to the generated event (event handling). At the implementation level, event awareness requires the application control flow to be able to receive and adapt to external commands, while event handling imposes the application to determine and execute the appropriate action in a timely manner (i.e., not necessarily instantly, but within the limits of user non-observability).

### 3.3 Performance predictability

In general, CE systems are soft real-time systems: application deadlines do exist, but they are not critical. Deadlines can be used to determine the optimal bounds for classical performance metrics, like execution time or speed-up. In essence, available resources and parallelism must be used as much as necessary in order to meet the deadlines; further optimizations may not be essential, unless they focus on the minimization of other parameters - power consumption, memory footprint, etc. *Performance predictability* is the ability of the application to evaluate itself, providing an useful feedback loop in exploring the design space options for parallelization and resource mapping. In the PiP example, assume 3 active pictures and 4 processors available. Two possible solutions for resource mapping: (1) decode each frame on one processor, and let the fourth processor make the assembly and display, or (2) make each processor compute one quarter of each active picture and display its part. The performance prediction mechanism has to decide, for each of these implementations, if they meet the deadline imposed by the required frame rate.

## 4 SP@CE

This section describes the design of the novel programming model SP@CE, and it briefly presents the early prototype implementation of its subsequent framework.

### 4.1 The SPC programming model

SPC is a programming model that imposes specific restrictions to the usage of synchronization mechanisms in order to achieve performance analyzable. The central philosophy in SPC is to express parallel computations in terms of processes and resources. The “SP” prefix stands for *series-parallel* and it suggests that the application must be expressed in terms of an SP structured computation, which constraints the condition synchronization

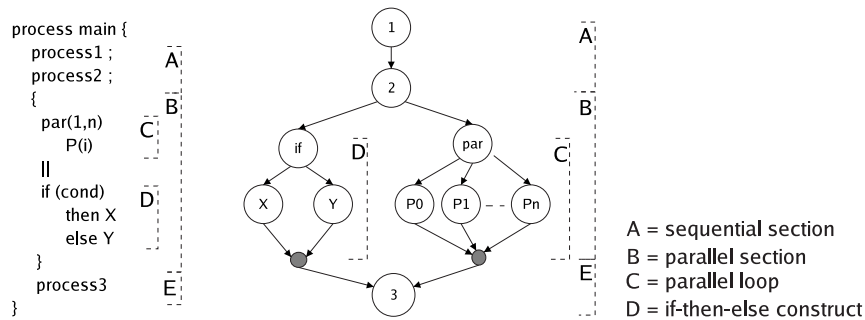


Fig. 1. An SP code snippet and its corresponding graph

patterns to only SP graphs<sup>4</sup> [7]. Several languages have considered an SP/nested parallelism model to allow for ease of programming in high-performance computing: the BSP model and its applications [8], Cilk [9], Satin [10], NESL [11]. The suffix “C” in SPC stands for *contention* and it refers to the use of resource contention to express mutual exclusion and, as a consequence, to describe scheduling constraints [12].

Despite these apparent expressivity limitations, it has been proved that SPC can capture the essential parallelism of an application. The loss in performance when remodeling a non-SP application to its best SP equivalent is typically bounded to few percents [13, 14].

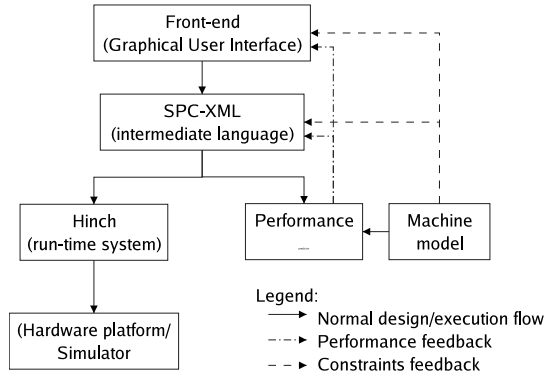
From the user point of view, SPC can be seen as a coordination paradigm that specifies the synchronization model of the application. Programming in SPC loosely refers to (1) expressing data computation as processes, (2) applying compositions between these processes, and (3) expressing mutual exclusion between processes in terms of resource contention (when required). Processes are usually expressed in a familiar sequential language, like C. Composition operators between processes allow sequential composition and loops, parallel composition and loops, with `fork/join` semantics, and conditionals. An illustrative example for the usage of these operators is presented in Figure 1.

While the processes are implemented to exploit the unbounded parallelism of the problem, resources are introduced as limitations of the actual parallelism of the system. The mutual exclusion synchronization mechanism provided by SPC is based on resource contention: two processes are in contention if they require the same resource to execute. Thus, they shall be binded on the same resource: `process(i) -> channelA` specifies that all `process(i)`'s must use the resource `channelA` mutually exclusive. To solve the contention, processes are dynamically scheduled with a user-specified scheduling policy (like FCFS, for example). Resources are universal in SPC: they can be either logical (critical sections in the program) or physical (processing units), to allow for a generic approach. And although resources in SPC are only meant as synchronization providers, they can be further used to facilitate the application mapping on real hardware resources.

We have chosen SPC as the basis of our SP@CE model because of its appealing features:

- **Ease of programming:** SP is a natural way of reasoning about data-parallel applications [15], while resource contention for synchronization avoids the typical hard-to-detect synchronization errors.
- **Explicit unbounded parallelism:** an application is designed and implemented in its SP form to exploit all the available parallelism of the problem. The resource mappings constrain the problem to the available parallelism of the system. However, this decoupling

<sup>4</sup> Another common name for *SP programming* is *nested-parallel programming*.



**Fig. 2.** The SP@CE framework components and their interaction

allows the application to be portable and easily re-targetable for a different execution platform.

- **Mixed parallelism:** SPC allows for data and task parallelism combinations. Typically, task parallelism will be nesting data parallelism. However, the boundary between data and task parallelism can be easily shifted, allowing finer or coarser grain parallelism.
- **Dynamic process-to-resource scheduling based on user defined policies:** SPC facilitates an easy transition from its abstract resources to the real hardware resources.
- **Performance analyzability and predictability:** based on estimating the application critical path augmented with the contention serialization penalties, a dedicated performance predictor, like PAMELA[16] can estimate the execution time of an SPC application.

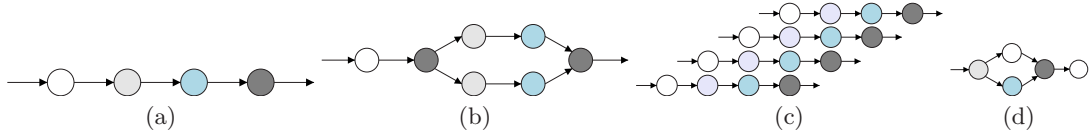
However, SPC has no native support for the other two critical requirements of CE applications, namely event awareness and reconfigurability. The novelty of the SP@CE model is to coherently extend the SPC model with these notions.

#### 4.2 SP@CE design and implementation

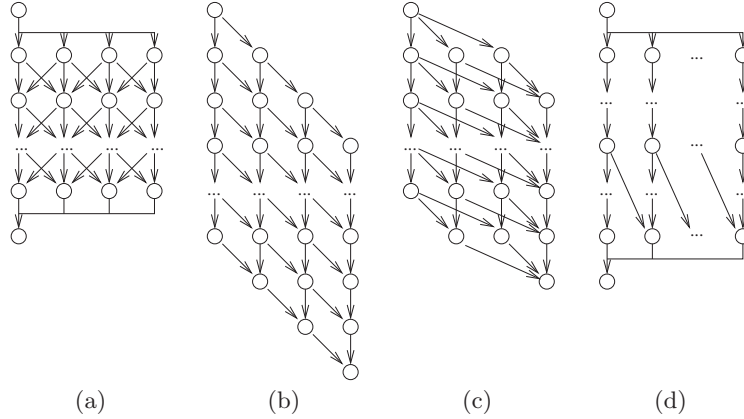
SP@CE aims to preserve the properties of SPC and tune them for streaming applications with CE requirements. Thus, we have identified the following SP@CE design requirements:

- Limit the application graph to an SP form - offers ease of programming, explicit data- and task-parallelism and performance prediction
- Focus on data streaming - provide data streams as a predefined data type
- Facilitate component-based design - simplifies code reuse and rises the level of abstraction
- Combine data and control flow in a coherent model
- Use a synchronous execution model - provides synchronization points that can be used for reconfiguration
- Allow dynamic reconfigurability - standard component interfaces allow runtime plug-and-play capabilities
- Provide event awareness and handling

To satisfy the design requirements and provide the user with all the means for implementing applications in SP@CE, we have designed the SP@CE framework, presented in Figure 2. Top-down, the layers are: the front-end layer, i.e., the user interface, the intermediate representation layer, and the dual-path execution layer, instrumented by PAM-SoC [2] for performance prediction and Hinch[1] for application execution on the target platform.



**Fig. 3.** Predefined compositions in SP@CE (similarly shaded bubbles execute the same code): (a) sequential, (b) parallel, (c) pipeline, (d) branch



**Fig. 4.** Supported non-SP compositions in SP@CE: (a) neighbor synchronization, (b) macropipeline, (c) fork-join/broadcast-reduction and (d) paired synchronization

**Front end** The front end is the main user interface with SP@CE, allowing users to specify components functionality - using a “classical” programming language, like **C**, and interconnections - by drawing the application graph. In the application graph, components are graph nodes, while edges are connecting components data ports with streams and components event ports with a centralized event manager. The front end separates components into: (1) *kernels* - simple data processing operations, (2) *loops* - loops of kernels, and (3) *hierarchies* - components build by connecting other “smaller” components.

*Building the data flow.* To simplify the task of constructing the application graph, the graphical user interface provides a few predefined SP-compliant constructs (see Figure 3):

- *sequential* and *sequential loop*: a chain of components that are executed in sequence, once or several (finite) times.
- *parallel* and *parallel loop*: components run in parallel; for loops, barrier semantics is applied for each iteration
- *pipeline*: the stages of the pipeline must be kernels or loops, a limitation that simplifies the SPC-compliance verification, without affecting the generality of streaming applications.
- *branch*: the user has to specify the conditional, as well as the **then** and **else** blocks.

Furthermore, based on the analysis performed by [17], we plan to support several well-known non-SP data-parallel computation structures, like neighbor synchronization, macropipeline, fork-join/broadcast-reduction and paired synchronization, presented in Figure 4. These structures can be automatically transformed into their SP equivalents with little overhead [12]. The transformation can be performed during the conversion from the graphical

interface to the intermediate code, but the effort of supporting these complex structures requires further analysis from the perspective of their usage in streaming applications.

*Building the control flow.* We divide the application control flow into three sub-categories: (1) *internal* control flow, i.e., inside the components, (2) *border* control flow, i.e., conditionals inside a process that affect its streaming behavior, and (3) *external* control flow, i.e., application events, either self- or user-generated.

Internal control flow is naturally managed by the components implementation language (in our case,  $\mathcal{C}$ ), as it is part of the processing. Border control flow influences the streaming behavior of the application. If, for example, due to a conditional inside a component, an output stream does not advance, the entire flow of application may stall, as SP@CE requires a fixed data rate for the streams. To prevent this, the programmer should define a “null-action” for each stream (e.g., duplication of the previous stream element), allowing the stream to advance without altering data.

Finally, because external control has to be processed at the level of the entire application, SP@CE uses a global *event manager* to gather events and propagate them to the application components. The event manager is implemented as a finite state machine (FSM) which has the possible events as inputs, and the generated events as outputs. The user’s task is to decide the logic of the FSM, and to correctly connect its outputs (i.e., the generated commands) to the event ports of the components. The FSM is evaluated at the end of each application iteration, allowing event handling in a timely manner, and with very little interruptions in the data flow.

*Reconfiguration.* Reconfiguration in SP@CE is supported by declaring reconfigurable subgraphs. These subgraphs contain optional parts that can be enabled or disabled as needed. Reconfiguration can only occur at special synchronization points, when the whole subgraph is idle, e.g., at the start or at the end of the subgraph iteration. By restricting reconfiguration to subgraphs, other parts of the application can continue execution without being interrupted.

To specify reconfiguration, the user has several options: (1) to add a component, (2) to remove components, and (3) to rearrange the graph. To add a component, the user must specify its type, the instantiation (i.e., initialization) values, and the connection options (i.e., to which other components it connects, and how). To remove a component, the user specifies which components is to be removed and, eventually, streams removal and/or reconnection. To rearrange the graph, the user must provide the list of modified connections. All these operations will typically be executed by the event handlers.

**SPC-XML** A first precompilation step converts this graphical representation into an SPC representation, for which we have chosen an existing language, namely SPC-XML [18]. The generated SPC-XML specification represents the high level structure of the application, i.e., an XML form of the drawn application graph, fully SPC compliant. The components code and interface details are simply propagated in SPC-XML. Thus, the final SPC-XML representation of an application specifies both functionality and coordination. It contains enough information to generate, by direct transformations, both the application code and the application model needed for the performance prediction module.

**Hinch** The application execution is supported by Hinch [1], a runtime system that takes care of load balancing the application over the available computation nodes, provides streaming communication primitives to the components, and supports dynamic reconfiguration.

Given that we have implemented SP@CE as a component-based model, this choice imposes Hinch to use components as its main entities, with a predefined set of properties:

- All components adhere to a single interface, which provides an abstraction of the *component* to Hinch. In this way, connecting and executing are addressed similarly for all component instances.
- Components have *input* and *output* ports, of two main types: streams (i.e., data) and events. Events are used to implement high-priority branches - for example, a monitoring component can signal a scalable image processing component to reduce its quality if the computed load on the system becomes too high.
- Components can be recursively grouped, allowing hierarchical compositions
- Component reuse is enabled by allowing multiple instances of a component to be active in different parts of the application.
- Components can be parametrized to accommodate different stream sizes, or, if given functions as parameters, to act as skeletons for sets of functions.

In Hinch, the application is built by grouping components recursively. The application model is a dataflow process network [19], in which the components are the actors. The application is run by executing *iterations* of the dataflow graph. In each iteration, each actor is fired one or several times, depending on the application data rates. One firing corresponds to running one iteration of the component. For example, in a video processing application, one iteration of a component may consist of processing one image frame from the video stream.

A graph iteration begins by scheduling the initial component(s). The other components are scheduled as soon as their predecessors in the dataflow graph have finished. Given that SP@CE supports iteration pipelining, multiple iterations can be active concurrently, which requires components to be aware of this and provide the necessary locking of their internal data structures to avoid race conditions. Although Hinch has no restriction on the shape of the dataflow graph, the graph will generally be SP-compliant, as it is generated from the high-level SP representation of the application.

**PAM-SoC** PAM-SoC is based on the PAMELA methodology[20], a static performance prediction methodology for general purpose parallel platforms (GPPPs). The PAMELA toolchain facilitates symbolic cost modeling. It features a modeling language, a compiler, and a performance analysis technique that enables PAMELA models to be compiled into symbolic performance models. The prediction trades accuracy for the lowest possible solution complexity. In this symbolic cost modeling, SP-compliant parallel programs are mapped into explicit, algebraic performance expressions in terms of program parameters (e.g., problem size), and machine parameters (e.g., number of processors). Instead of being simulated, the model is automatically *compiled* into a *symbolic cost model*, that can be further compiled into a *time-domain cost model* and, finally, evaluated into a *time estimate*.

In order to address the specifics of embedded multi-core hardware platforms, we have developed PAM-SoC, a toolchain that includes, beside PAMELA, new techniques for machine modeling and dedicated tools for gathering memory behavior statistics [2]. To predict the performance of an application, PAM-SoC couples the application model with the target machine model, computing the lower bound of the execution time of the application on the target architecture. Both models are written in the PAMELA modeling language, a process-oriented language designed to capture concurrency and timing behavior of parallel systems [21]. Note that the *machine model* is expressed in terms of available resources and an abstract instruction set (AIS) for using these resources. The machine model is currently developed by

hand. The *application model* of the parallel program is implemented using an (automated) translator from the source instruction set to the machine AIS.

The role of PAM-SoC in the SP@CE framework is to predict the performance of the application in a form that can be used as feedback for the application design. PAM-SoC is able to (1) estimate the average execution time of a given application and (2) identify the potential resources that generate bottlenecks. Given these information, the user should be able to tune the application design and/or implementation to alleviate the bottlenecks and bring the execution time within the required limits. Minor tuning can be done directly at the SPC-XML layer, while major bottlenecks must be dealt with at the design level (i.e., the front end). Although the scenario we propose is not an automated one, but an user-assisted iterative solution, we believe that the iteration speed and the reasonably accurate performance predictions (error within 15%) allow for efficient implementations.

## 5 Experiments

In this section, we present our initial results with the SP@CE prototype. We focus mainly on the expressiveness issues, discussing the way streaming consumer electronics applications can be programmed. We first describe the experimental setup, followed by the applications used in the experiments and the results. In this paper, we focus on evaluating (1) the overhead of the SP@CE framework by comparing functionally equivalent applications, developed with and without the SP@CE framework, and (2) the performance of the SP@CE applications when running in parallel. More specific details about the runtime system implementation and behaviour in terms of performance, reconfigurability latency and reconfiguration overhead are presented in [1].

### 5.1 Experimental setup

All experiments are performed using the SpaceCake architecture[22], provided by Philips. This architecture has multiple identical processing tiles that communicate using distributed memory. Each tile is a shared memory multiprocessor on chip, called Wasabi. Wasabi contains a general purpose processor core, multiple TriMedia DSP cores, and specialized hardware to speedup specific operations. Per tile, each core has its own L1 cache, while the L2 cache is shared between all cores.

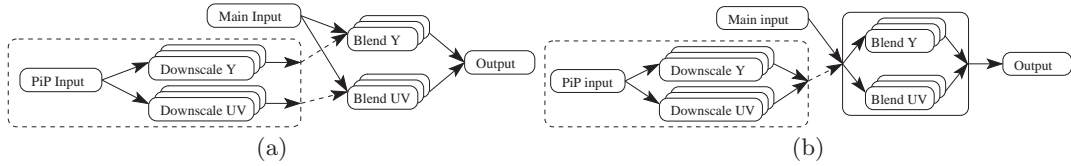
Since SpaceCake hardware is not available, all experiments are run using Wasabi’s cycle accurate simulator, provided by Philips, which simulates a single tile with multiple TriMedia cores.

In all the experiments, we measure and compare the relative performance of the main computational part of the applications. To avoid distorting the results with the overhead introduced by the simulation I/O mechanisms, the input file(s) are fully read at initialization, and the final output results are discarded. The SP@CE component architecture simplifies the transition from these testing prototypes to real applications, as these “dummy” input and output components may be easily replaced with functional ones.

### 5.2 Applications

**Motion JPEG.** The first application we evaluated is a Motion-JPEG decoder (MJPEG). It takes a file with 50 concatenated 1280x720 jpeg coded images as input and decodes these into planar Y, U and V buffers. As shown in Figure 6, this application consists of three main components:

1. MJPEG input. This is a simple component that splits the mjpeg file into separate jpeg files. It supplies the next component with a jpeg file in each application iteration.



**Fig. 5.** Picture-in-Picture application graph: (a)Non-SP, (b)SP-compliant

2. JPEG bit stream decoder. This component decodes the jpeg file into Discrete Cosine Transformed (DCT) blocks for each color component in the image. This includes: jpeg header decoding, Huffman decompression, and inverse-zigzag coding. The component can either run in a pipeline fashion, decoding multiple jpeg images concurrently, or it can run in a sliced mode, decoding one jpeg image split up into slices (i.e., adjacent sets of lines). In the sliced mode, the data processing in the bit stream decoder is fully sequential (slice after slice), but the model allows the next component to start running as soon as the first image slice is available. In the non-sliced mode, the next component can be run when all DCT blocks are decoded, but the following image is already in the pipeline. The estimates given by PAM-SoC, confirmed by real measurements, have indicated that the non-sliced mode performs better. Thus, guided by the SP@CE integrated tools, we have taken the appropriate design decisions and used the non-sliced version.
3. JPEG DCT decoder. This component generates pixel data from the input DCT blocks by performing an inverse discrete cosine transform (IDCT) followed by shift and bound operations. There is one DCT decoder for each color component in the image. Since there is no data dependency between the DCT blocks, data parallelism can be exploited by decoding multiple image slices simultaneously.

**Picture-in-Picture.** The second application we have evaluated is Picture-in-Picture (PiP). The application combines 96 images from multiple uncompressed 720x576 image streams into a single image stream by scaling down image streams and blending these into the main (background) image stream. We have four versions of the PiP application (PiP-0 to PiP-3), with 0, 1, 2, and 3 pictures-in-picture, respectively.

The components and data streams in the PiP application are shown in Figure 5(a). The downscale and blend components are run using data-parallelism. The full arrows in the figure correspond to luminance (Y) and packed chrominance (UV) streams. As the original graph is non-SP, we have converted it to its SP form by introducing a new synchronization point before the blender components. The resulting application graph is shown in Figure 5(b)<sup>5</sup>.

This procedure shows how a non-SP graph is redesigned as SP. Although the SP version presents more dependences and the two blend components may have to wait for both luminance and chrominance streams downscaling, the inherent load-balance of the downscaling process alleviates performance penalties. Other forms of SP-graphs could be selected for applications with similar structure but different load-balance conditions. The SP@CE prediction tool shows which gives the best performance.

**Motion JPEG + Picture-in-Picture.** We have also created an application (JPiP) by adjusting PiP to use components from MJPEG as input components, instead of the standard input components. The application combines 16 images from multiple jpeg-compressed

<sup>5</sup> For clarity, the graphs only show the dependencies between the components, and not all individual streams. Each dependency corresponds to a luminance stream (Y), chrominance stream (UV), or both (YUV).

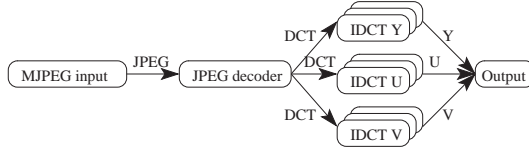


Fig. 6. Motion JPEG, SP@CE implementation

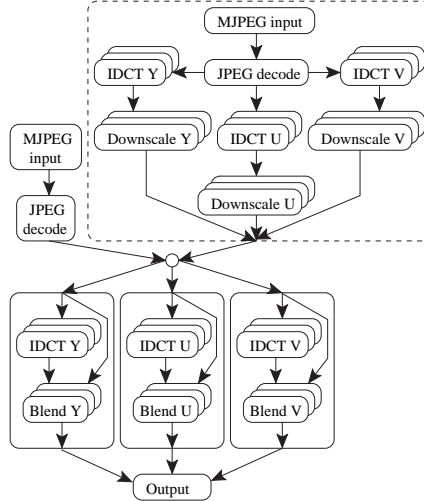


Fig. 7. Combined Motion JPEG/Picture-In-Picture application

1280x720 image streams into a single 1280x720 image stream. Similarly to PiP, we have four versions JPiP (JPiP-0 to JPiP-3), with 0, 1, 2, and 3 pictures-in-picture, respectively.

The structure of JPiP, with one picture-in-picture, is shown in Figure 7. Being a combination of PiP and MJPEG, JPiP has three downscaling components for each picture-in-picture and three blenders, instead of two. To reduce synchronization overhead, the data-parallel components from both applications are grouped together.

The JPiP application is also a good example of the usability of SP@CE. Without SP@CE, it would have taken quite some effort to build a JPiP equivalent, as all communication and scheduling have to be programmed manually. Further more, code re-use would have been hindered: even though equivalents of MJPEG and PiP are available, various parts of these applications have to be adjusted to fit the new communication and scheduling patterns. With SP@CE, the only thing that had to be done was initializing and connecting existing components. Code reuse is practically optimal, as the components themselves needed no modifications at all.

### 5.3 Sequential overhead

To estimate the SP@CE model overhead, we have compared the execution time of the SP@CE versions of PiP and MJPEG against their reference implementations. These reference implementations are sequential. Figure 8 shows the execution times of the non-SP@CE implementations, compared to a sequential and a parallel SP@CE version. SP@CE adds a small overhead (within 10%), due to its component-based structure. However, exactly due to its parameterized component-based structure, it allows for the same application to be executed in parallel. Given the much better execution time of the parallel version, we consider the sequential overhead not significant.

The overhead in the (sequential) SP@CE PiP applications is due to the fact that the blender is a separate component, while it is integrated in the downscaler in the non-SP@CE version. Profiling information shows that down scaling the image takes an almost equal amount of cycles for both versions. The difference lies in the amount of data copies, which is larger with a separate blender. However, we expect redundant buffering introduced by the SP structured form of component composition to be easily detected and eliminated by an optimization stage.

The non-SP@CE version of MJPEG decodes the DCT blocks as soon as they are decoded from the bit stream. It is 14% faster than the sequential SP@CE version. Profiling information shows that half the difference is due to communication overhead of the DCT buffers. Buffering DCT data causes data cache misses, both at the writing side (bit stream decoder) and the reading side (DCT decoder). The other half of the difference is added by the SP@CE model, due to some inefficiencies in data management and some code redundancies, mostly derived from generalization and support for parallelism. Better optimization in the SP@CE-generated code may alleviate them. The runtime system (Hinch) does not add significant overhead.

#### 5.4 Parallel Performance

Figure 9 shows the speedup of the SP@CE applications when run on multiple TriMedia nodes. Because reference parallel implementations of the used benchmarks are not (publicly) available, we compare the parallel performance against the sequential SP@CE versions. PiP-0 does not exhibit much speedup because it is a trivial application that merely copies its input to its output. It is limited by memory bandwidth, not by processing power. The efficiency of PiP-1 decreases beyond seven nodes because there is no more parallelism to exploit. PiP-2 and PiP-3 do not suffer from this problem and show efficiencies of above 98% at nine nodes. The speedup for MJPEG does not increase much when it is run at more than four nodes. Beyond this point, the added compute power is hardly used because there is only little additional parallelism to exploit.

The performance of JPiP-0 resembles that of MJPEG since these applications are highly identical: the main differences are the blender components, which are only present in JPiP. Like in PiP-0, the blend components in JPiP-0 do nothing but copying their single input to their output. Profiling information shows that less than two percent of all computation in JPiP-0 is spent in the blend components. In JPiP-1, JPiP-2, and JPiP-3 there is an abundance of parallelism to exploit. These applications therefore achieve good speedup figures, e.g., JPiP-3 has an efficiency of 96 % at 9 nodes.

To summarize, the results of the experiments presented in this section provide evidence that the SP@CE model is a suitable option for implementing predictable parallel streaming applications. Furthermore, while the model and its framework do not induce high overheads, they provide good performance in terms of applications speed-ups.

## 6 Related Work

We relate our work with different types of solutions - languages, design frameworks, and models - for programming streaming applications. For a reference survey on the origins and developments of streaming programming languages, we relate the reader to [3]. The survey presents reference languages like Lucid, LUSTRE, ESTEREL, and many others, until the mid-90's. Our data-flow approach on streaming, together with the representation of streams by their temporal instances largely follows the Lucid approach [23]. The model of synchronizing the application by iteration is similar to the approach of synchronous languages

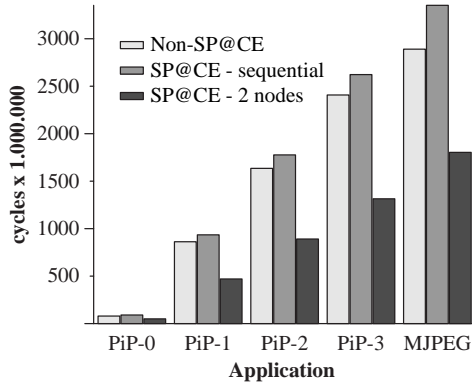


Fig. 8. SP@CE overhead

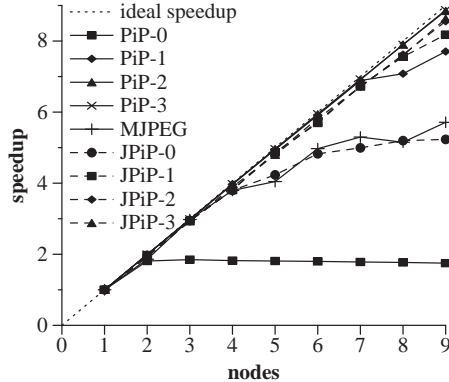


Fig. 9. SP@CE speedup

presented by [24] for LUSTRE and [25] for ESTEREL. However, none of these languages take into consideration issues like parallelization or reconfiguration, while events are only marginally discussed.

The most influential “modern” streaming language is StreamIt [26], which also expresses an application as a hierarchical graph of filters connected by streams. To insure correct composition of the filters, only a small number of composition operators are permitted. Components functionality is developed in C and/or Java, allowing code reusability and making the language reasonably user-friendly. However, StreamIt solutions for dealing with reconfiguration and events are cumbersome and limited compared to our approach. Finally, while StreamIt is elegantly exploiting task parallelism, data parallelism is only partially supported. Compared to the lower-level model of languages like Brook [5] or Stream-C/Kernel-C [27], our component-based model raises the level of abstraction, being easier to use for both design and implementation.

Nizza is a framework proposed in [28] as a methodology for developing multimedia streaming applications. Similar to our framework, Nizza uses a data-flow model, and it proposes a complete design-to-implementation flow, but it lacks a generic concept of events and reconfiguration is not entirely dynamic (it requires a restart of the framework). Also, as Nizza targets desktop applications, no performance feedback loop is included.

TStreams [6] is an abstract, dedicated model for parallel streaming applications, based on the same explicit parallelism approach as SP@CE. It remains to be seen if the model implementation is able to preserve these features. The Space-Time Memory abstraction (STM) [29] is a model with a different look on streams: an application is a collection of streams that need processing, so threads can attach them, process, and detach from them as required. The system is dynamic, natively reconfigurable and time-synchronous, being able to exploit both task and data parallelism. Again, the major drawback is in the model implementation that preserves these properties and remains programmer-friendly. Although SP@CE’s model is simpler, it allows for a user-friendly implementation that offers a good compromise between the abstraction level and usability.

Kahn Process Networks (KPN) [30] are a popular streaming model for the embedded systems industry, because they are determinate and compositional. However, KPNs have no global state, and they are not reactive to external events. Models like Context Aware Process Networks model (CAPN) [31] and Reactive Process Networks (RPN) [32] alleviate this problems by extending KPN with global state and event awareness, but they sacrifice its determinate property. As a result, they are not predictable. These models do not tackle

dynamic reconfiguration and do not include data parallelism facilities, which are both strong points of SP@CE.

Data-flow models are extensively used for expressing streaming applications [19, 33]. SP@CE follows a similar graph-of-tasks approach as these models, and it is similar, in its synchronous approach, with the Synchronous Data-Flow [34, 35] model. Still, most data-flow models implementations do not tackle dynamic reconfiguration (with an exception in the Parameterized Data Flow model [36]) and do not include data parallelism features. Furthermore, note that an important advantage of SP@CE over generic data-flow models is predictability and analyzability.

## 7 Conclusions and Future Work

We have presented SP@CE, a new programming model for streaming applications for MP-SoC Consumer Electronics platforms. One of the main contributions of this work is the analysis of the specific requirements for streaming applications running on consumer electronics platforms. We believe that we have identified and listed *all* the properties that *must* be provided by a dedicated programming model aiming to increase programming correctness, efficiency and productivity. A further step was the SP@CE programming model itself, as an extension of the SPC model that embeds all the aforementioned properties

To prove the usability of SP@CE, we have designed a three-layer framework that assists the programmer in the design-to-execution flow of a streaming application. The SP@CE framework includes an user-friendly front-end, an XML-based intermediate representation, a runtime system and a performance feedback loop. A prototype of this framework has been used to experiment with several real streaming applications on a given multiprocessor CE platform. We have presented the results of these experiments, which prove that SP@CE's component-based approach provides good performance numbers, low overhead and nearly optimal code reuse.

For future work, on short term, our main target is to further validate the results by implementing more applications and more CE platforms. Further, we aim to make several enhancements of the framework prototype, including a complete graphical implementation of the front-end, more aggressive optimization engines for both SPC-XML and Hinch, and fully static performance prediction with PAM-SoC.

## References

1. Nijhuis, M., Bos, H., Bal, H.: Supporting reconfigurable parallel multimedia applications. In: Euro-Par'06. (2006) 765–776
2. Varbanescu, A.L., van Gemund, A., Sips, H.: PAM-SoC: A toolchain for predicting MPSoC performance. In: Euro-Par'06. (2006) 111–123
3. Stephens, R.: A survey of stream processing. *Acta Informatica* **34**(7) (1997) 491–541
4. Thies, W., Gordon, M.I., Karczmarek, M., Lin, J., Maze, D., Rabbah, R.M., Amarasinghe, S.: Language and compiler design for streaming applications. In: IPDPS'04 - Workshop 10. Volume 11. (2004)
5. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream computing on graphics hardware. In: SIGGRAPH 2004, (ACM Press)
6. Knobe, K., Offner, C.D.: Compiling to TStreams, a new model of parallel computation. Technical report (2005)
7. Valdes, J., Tarjan, R.E., Lawler, E.L.: The recognition of series parallel digraphs. In: STOC '79, New York, NY, USA, ACM Press (1979) 1–12

8. Valiant, L.G.: A bridging model for parallel computation. *Commun. ACM* **33**(8) (1990) 103–111
9. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: *Symp. on Foundations of Computer Science*. (1994) 356–368
10. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications. In: *PPoPP'01*, ACM Press (2001) 34–43
11. Blelloch, G.E.: NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University (1993)
12. González-Escribano, A.: Synchronization Architecture in Parallel Programming Models. PhD thesis, Dpto. Informatica, University of Valladolid (2003)
13. van Gemund, A.: The importance of synchronization structure in parallel program optimization. In: *ICS '97: Proc. 11th international conference on Supercomputing*, New York, NY, USA, ACM Press (1997) 164–171
14. Salamon, A.Z.: Task graph performance bounds through comparison methods. Technical report, Dept. of Computer Science, University of the Witwatersrand, Johannesburg, South Africa (2001)
15. Skillicorn, D.B., Talia, D.: Models and languages for parallel computation. *ACM Comput. Surv.* **30**(2) (1998) 123–169
16. van Gemund, A.: Performance prediction of parallel processing systems: The PAMELA methodology. In: *ICS'93*. (1993) 318–327
17. Malony, A.D., Mertsiotakis, V., Quick, A.: Automatic scalability analysis of parallel programs based on modeling techniques. In: *TOOLS'94*. (1994) 139–158
18. González-Escribano, A., van Gemund, A., Cardeñoso-Payo, V.: SPC-XML: A structured representation for nested-parallel programming languages. Volume 3648., Springer-Verlag (2005) 782–792
19. Lee, E.A., Parks, T.M.: Dataflow process networks. In: *Proc. of the IEEE*. (1995) 773–799
20. van Gemund, A.: Performance Modeling of Parallel Systems. PhD thesis, Delft University of Technology (1996)
21. van Gemund, A.: Symbolic performance modeling of parallel systems. *IEEE TPDS* (2003)
22. Stravers, P., Hoogerbrugge, J.: Single chip multiprocessing for consumer electronics. In Bhattacharyya, ed.: *Domain-Specific Processors*. Marcel Dekker (2003)
23. Ashcroft, E., Wadge, W.: *Lucid, the Dataflow Programming Language*. Academic Press (1985)
24. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. *Proc. IEEE* **79**(9) (1991) 1305–1320
25. Berry, G., Gonthier, G.: The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* **19**(2) (1992) 87–152
26. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: *Computational Complexity*. (2002) 179–196
27. Kapasi, U., Dally, W.J., Rixner, S., Owens, J.D., Khailany, B.: The Imagine stream processor. In: *ICCD'02, IEEE* (2002) 282–288
28. Tanguay, D., Gelb, D., Baker, H.H.: Nizza: A framework for developing real-time streaming multimedia applications. Technical report (2004)
29. Rehg, J., Ramachandran, U., Jr., R.H., Joerg, C., Kontothanassis, L., Nikhil, R., Kang, S.: Space-Time Memory: a parallel programming abstraction for dynamic vision applications. Technical report (1997)
30. Kahn, G.: The semantics of a simple language for parallel programming. In: *IFIP Congress '74*, New York, NY, North-Holland (1974) 471–475
31. van Dijk, H.W., Sips, H., Deprettere, E.F.: Context-aware process networks, Los Alamitos, CA, USA, IEEE Computer Society (2003) 6–16
32. Geilen, M., Basten, T.: Reactive process networks. In: *EMSOFT'04*. (2004) 137–146
33. Ko, D.I., Bhattacharyya, S.S.: Modeling of block-based DSP systems. Volume 40., Kluwer Academic Publishers (2005) 289–299
34. Lee, E., Messerschmitt, D.: Synchronous Data Flow. *IEEE Trans. Comp.* **36**(1) (1987) 24–35
35. Stuijk, S., Basten, T.: Analyzing concurrency in streaming applications. Technical report (2005)
36. Bhattacharya, B., Bhattacharyya, S.S.: Parameterized dataflow modeling for DSP systems. *IEEE Trans. on Signal Processing* **49**(10) (2001) 2408–2421