Implementation of Dynamic Streaming Applications on Heterogeneous Multi-Processor Architectures

Tomas Henriksson, Jeffrey Kang, and Pieter van der Wolf
Philips Research, High Tech Campus 31, 5656 AE Eindhoven, The Netherlands
tomas.henriksson@philips.com

ABSTRACT

System design based on static task graphs does not match well with modern consumer electronic devices with dynamic stream processing applications. We propose the TTL API for task graph reconfiguration services, which can be used to describe the dynamic behaviour of applications. We demonstrate the efficient implementation of the TTL API on a heterogeneous multi-processor architecture. It is possible to design dynamic streaming applications with reusable reconfiguration-aware tasks and we argue that the TTL API serves as a good starting point for standardization.

Categories and Subject Descriptors

D.2.11 [Software architectures]: Domain-specific architectures; C.3 [Special-purpose and application-based systems]: Real-time and embedded systems; D.2.13 [Reusable software]

General Terms

Design, Performance, Standardization

Keywords

System level design, dynamic applications, platform interface

1. INTRODUCTION

Modern consumer electronic (CE) devices execute audio, video, and communications applications. The devices are typically based on heterogeneous multi-processor architectures because they provide the best trade-off between flexibility, performance, and power consumption. The most performance demanding parts of the applications perform stream processing, which can be described as a graph of communicating tasks. If the tasks use a well-defined interface to interact with each other, the tasks can be easily integrated into a system. The interface decouples the tasks from each other and from the implementation of services, for example for the inter-task communication. If a single standard interface is agreed upon, the tasks can be reused on different multi-processor architectures. Reuse of tasks is one important way to improve design productivity of embedded systems.

Interface-based system level design with task graphs has concentrated on static applications described with e.g. Kahn Process Networks [1] or Synchronous Data Flow [2] models. Modern CE devices are multi-functional devices, which dynamically change application behaviour. Therefore static task graph models are no longer sufficient. New services have to be introduced to be able to describe the complete stream processing part of the applications. We call these new services *reconfiguration services*. There are two major requirements on the reconfiguration services:

1. They must allow for easy description of applications
2. It must be possible to implement them efficiently on a range of heterogeneous multi-processor architectures

The reconfiguration services that we consider, are used 1) at start-up of the system, 2) when the user triggers a change in the application, 3) when the input data triggers a change in the application, and 4) when the system is shut down. At start-up an initial task graph is constructed and started. For a TV set that is for example resuming the settings from when the TV was last switched off. The user interface can trigger changes of two principal types:

- Changes inside a task, which we call *parameter setting*
- Changes in the task graph, which we call *topology reconfiguration*

The same types of changes can be triggered by the input data. An example of parameter setting is an audio application, which plays a sequence of MP3 songs and needs to change the settings of the sample rate conversion task dependent on the sample rate in the new song. An example of topology reconfiguration is an MPEG 4 decoder, which has to launch a new VOP (video object plane) decoder task when a new VOP is detected in the encoded data stream. The focus of this paper is on topology reconfiguration.

We have previously presented the task transaction level (TTL) interface for inter-task communication and multi-tasking [10]. The aim of this paper is to present concepts and the TTL application programmer's interface (API) for task graph reconfiguration services and to show that the TTL API can be efficiently implemented on a multi-processor architecture. We prove that reusable tasks can be used for designing dynamic stream processing applications. The TTL API has been implemented on other multi-processor architectures as well. The portability of the TTL API makes it a good candidate for a standard.

The rest of this paper is structured as follows. In Section 2 related work is presented. In Section 3 we present our conceptual model and in Section 4 the TTL reconfiguration API is described. Section 5 presents a case study application and its implementation with the TTL API. In Section 6 the implementation of the reconfiguration services on a multi-processor architecture is explained and the results of the implementation are presented in Section 7. In Section 8 we discuss how the TTL concepts, API, and implementation fulfil the requirements and finally in Section 9 we present our conclusions.

2. RELATED WORK

In [3] and [4] APIs for task graph reconfiguration are presented and implementations are briefly discussed. The TTL API is inspired by that work. In contrast to those publications, we show how the TTL API can be efficiently implemented on a multi-processor architecture and how to construct applications by using the TTL API.

In [5] a formal model for specifying dynamically changing applications is described, but an API and a link to multi-processor imple-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'05, Sept. 19-21, 2005, Jersey City, New Jersey, USA.
Copyright 2005 ACM 1-59593-161-9/05/0009...$5.00.

mentation is missing. In [6] the need for analysing dynamically changing applications is acknowledged, but no API for implementing them is discussed. Also no implementation is presented. In [7] structurally adaptive systems for signal processing are discussed. This is similar to our work. A mechanism for restructuring is mentioned, but no API is presented. From the paper it is clear that only a single-processor implementation has been done. In [8] a coordination language, Manifold, for parallel applications is discussed, which has an API for dynamic reconfiguration. The implementation of that language on a multi-processor architecture is however not discussed. In [9] dynamic reconfiguration of a subsystem with hardware co-processors is described. The implementation is however only briefly touched upon. Our work distinguishes from previous work by the fact that we address a broad range of multi-processor architectures and that we thereby can standardize an API.

A related issue is how to make sure that the system resources are not overutilized and that the application fulfils all real-time requirements. This is however different from our work in the sense that it uses non-functional models to reason about the implementation, see for example [11]. The actual implementation, which we present in this paper, can be modelled and reasoned about with such methods.

3. CONCEPTUAL MODEL

We view the stream processing part of an application as a graph of communicating *tasks*. A task has *ports* to be able to communicate with other tasks. The ports of the tasks are connected via *channels* and thereby create a task graph. Apart from the tasks, there is also a *configuration manager* (CM), which manages the changes in the task graph topology, and one or more *schedulers*, which activate the tasks.

On certain occasions, the task graph topology has to change because of inputs from the user interface or specific patterns in the input data. It is possible to destroy the complete task graph and construct a new one from scratch for every change in the task graph topology. This is however not desirable because some parts of the task graph might have to continue executing for user perception reasons while reconfiguring other parts. Therefore we need reconfiguration services that can modify the task graph while some tasks are still executing. For that purpose the CM must be able to *connect* and *disconnect* ports from channels.

A task may only execute if all its active ports are connected to channels. So the CM must be able to *start* and *stop* tasks when they are properly configured. When a task is stopped, its internal state is lost. Sometimes it is desirable to change the task graph topology without losing the state of a task. Therefore the CM must also be able to *pause* and *continue* tasks. In order to let the tasks finish the processing of a semantically meaningful data entity, e.g. a video frame, the tasks themselves acknowledge a *stop* or *pause* command before they actually stop or pause. These tasks are called reconfiguration-aware.

Reconfiguration-aware tasks have additional complexity to handle reconfiguration and negatively impact the reconfiguration latency because tasks do not stop or pause immediately. It is desirable to be able to have simple tasks and to be able to stop tasks quickly. For that purpose the CM can *force* a task to stop. In that case the task is not involved and there is only interaction between the CM and the scheduler of the task.

In order to use memory resources efficiently, the CM must be able to dynamically *create* and *destroy* tasks and channels. The dynamics of a task can be described by its configuration state transition diagram, see Figure 1. Before a task even exists it has to be created. After creation the task is in the INITIAL configuration state. The CM then moves the task to the STOPPED configuration state by calling its initialization method, see Section 4. Only when a task is in the RUNNING configuration state the scheduler may activate it.

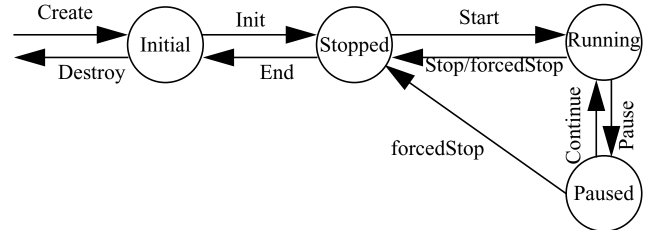

Figure 1. Configuration State Transition Diagram

When a task is created, it is decided on which *processor domain* it will execute. A processor domain may be a number of processors or one single processor. Every processor domain belongs to one scheduler, which takes care of activation of the tasks when they are in the RUNNING configuration state. The task is moved to the RUNNING configuration state when it is started.

Heterogeneous multi-processor architectures typically have several different memories. Data structures and channel buffers must reside in memories that can be accessed by the processor(s) on which related tasks execute. Therefore it must be possible to specify in which memory an allocation should be done. This is done by the use of *memory domains*.

4. API SPECIFICATION

The CM, which uses the reconfiguration services, typically executes on a microcontroller. For most microcontrollers, C compilers are used and therefore we chose to present the TTL reconfiguration API in C. APIs with the same semantics can however also be defined in other programming languages. The functions in the API are structured in 4 groups, see Table 1.

Group 1 consists of functions for creation and destruction. Both tasks and channels can be dynamically created and destroyed. The task designer has to provide four functions for each task, *mainMethod* for executing the task behaviour, *resetMethod* for resetting the task, *initMethod* for initializing the task, and *endMethod* for cleaning up the task. The *mainMethod* and the *resetMethod* are supplied to the scheduler in the *taskCreate* function. The other two functions are called directly from the CM. The *initMethod* gets the task, a memory domain, and possibly other parameters as inputs. It is responsible for allocating memory for a task specific data structure. This task specific data structure contains ports and task specific variables. With data structure in this paper we refer to a sequential memory block of known size, such as a C struct. The *endMethod* is called from the CM just before the task is destroyed. It is responsible for freeing any memory that the *initMethod* of the task has allocated, including the task specific data structure. The *resetMethod* is used by the scheduler to re-initialize the task specific data structure when a task has been stopped and will later be started again.

In TTL a task can be of three different task types, *processes, coroutines*, or *actors* [10]. The *mainMethods* of tasks of the three different task types have to be activated in different ways, therefore the scheduler must know of what type each task is. The channels can be located in different memories in a multi-processor architec-

ture. Therefore a memory domain for the location of the channel buffer is supplied in the *ttlChannelCreate* function.

Table 1. Groups of API functions

Group 1
ttlTask *ttlTaskCreate(ttlTaskType type, ttlProcessorDomain domain, void (*mainMethod)(ttlTask*), void (*resetMethod)(ttlTask*));
void ttlTaskDestroy(ttlTask *t);
ttlChannel *ttlChannelCreate(Uint token_size,Uint num_tokens, ttlMemoryDomain domain);
void ttlChannelDestroy(ttlChannel *c);
Group 2
void ttlCbOutPortConnect(ttlCbOutPort *p, ttlTask *t, ttlChannel *c);
void ttlCbInPortConnect(ttlCbInPort *p, ttlTask *t, ttlChannel *c);
void ttlCbOutPortDisconnect(ttlCbOutPort *p, ttlTask *t, ttlChannel *c);
void ttlCbInPortDisconnect(ttlCbInPort *p, ttlTask *t, ttlChannel *c);
Group 3
void ttlTaskStart(ttlTask *t);
void ttlTaskPause(ttlTask *t);
void ttlTaskContinue(ttlTask *t);
void ttlTaskStop(ttlTask *t);
void ttlTaskForcedStop(ttlTask *t);
Group 4
bool ttlTaskCheckPause(ttlTask *t);
bool ttlTaskCheckStop(ttlTask *t);
void ttlTaskAcknowledgePause(ttlTask *t);
void ttlTaskAcknowledgeStop(ttlTask *t);

Group 2 consists of functions to connect and disconnect ports to and from channels, examples for interface type CB [10] are shown. Group 3 consists of functions to start, pause, continue, and stop tasks. They all take the task as input. The functions in group 4 are used in the *mainMethod* of the tasks. The reconfiguration-aware tasks use the four functions to check for and acknowledge the pause and stop commands.

5. CASE STUDY

A case study has been performed with an audio application for MP3 decoding. The application consists of three use cases. At start-up the system always initializes use case 1.

5.1 Use Cases

In use case 1, an MP3 file is decoded and played back on one set of stereo speakers. One task reads the MP3 file from flash memory. A second task decodes the MP3 bitstream into raw audio. If the MP3 bitstream contains mono audio, it is duplicated so that the output from the MP3 decoding task is always 2 channel audio. A third task sends the audio samples to the D/A converters. The use case is shown in Figure 2.

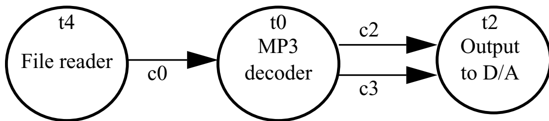

Figure 2. Use Case 1

In use case 2, an MP3 file is decoded and played back on two sets of stereo speakers. The file reader task and the MP3 decoding task are the same as in use case 1. After the MP3 decoding task, two fork tasks are added. They simply split a signal into two identical copies. There are two tasks that send stereo audio samples to the D/A converters. Use case 2 is shown in Figure 3.

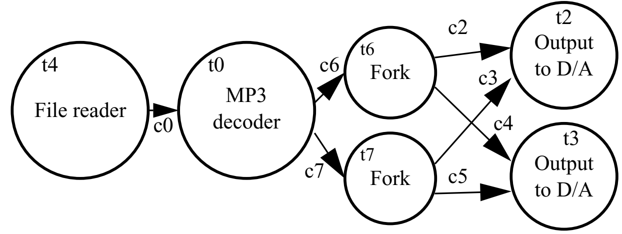

Figure 3. Use Case 2

In use case 3, two different MP3 files are decoded and played back on two sets of stereo speakers. The task graph is organized as two independent copies of use case 1. Use case 3 is shown in Figure 4.

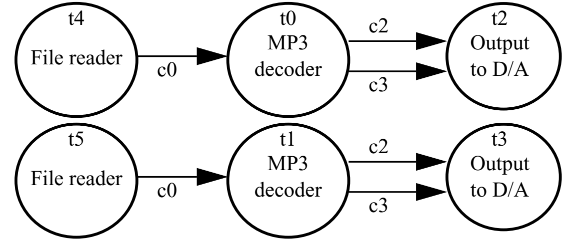

Figure 4. Use Case 3

5.2 Configuration Manager in the Case Study

When there is an input from the user interface to change to a different use case, the configuration manager stops and pauses tasks and disconnects ports that have different connections in the new use case. Then tasks and channels that are not part of the new use case, are destroyed and new tasks and channels are created. After that the ports are connected to channels according to the new use case and finally the tasks are started or continued.

The CM in the case study provides a single function, *setUseCase*, which is used to switch from one use case to another. The *setUseCase* function is structured as nested switch statements. Dependent on current and new use case action is taken. The code for changing from use case 1 to use case 2 is shown in Figure 5.

The transition starts with pausing the tasks *t0* and *t2* on lines 200-201. Then the channels *c2* and *c3* are disconnected from the MP3 decoder *t0* on lines 203-204. The ports are fields in the task specific data structure, which can be retrieved with the *ttlTaskGetSpecific* function as used on line 202. After that the construction phase is started. The output task *t3* is created and initialised on lines 205-207. In the *OutputTask_init* function a task specific data structure of type *OutputTask* is allocated, which contains the input ports *p0* and *p1*. The fork tasks *t6* and *t7* are created and initialised on lines 208-213. The channels *c4-c7* are created on lines 214-217, all in memory domain *md2* and with token size of 4 bytes. The construction phase ends with connecting the ports to the channels on lines 218-230. The input ports of output task *t2* are already connected to channels *c2* and *c3* so they do not have to be connected. The same holds for the ports connected to channel *c0* on the file reader task *t4* and the MP3 decoder task *t0*. Finally, the tasks *t0* and *t2* are continued and the tasks *t3*, *t6*, and *t7* are started on lines 231-235.

In the transition from use case 1 to use case 2 all existing tasks are retained. In other transitions tasks are destroyed and memory for the task specific data structures are freed. This allows for several use cases with different tasks with total system memory requirements dependent only on the most complex use case.

```
2     MP3DecTask *m0;
3     OutputTask *o1;
4     ForkTask *f0, *f1;
...   ...
199  case 2:
200     ttlTaskPause(t0);
201     ttlTaskPause(t2);
202     m0 = (MP3DecTask*)ttlTaskGetSpecific(t0);
203     ttlCbOutPortDisconnect(&m0->p1, t0, c2);
204     ttlCbOutPortDisconnect(&m0->p2, t0, c3);
205     t3 = ttlTaskCreate(TASK_TYPE_ACTOR, pdISR,
206        OutputTask_main, OutputTask_reset);
207     OutputTask_init(t3, md2);
208     t6 = ttlTaskCreate(TASK_TYPE_PROCESS, pd2,
209        ForkTask0_main, ForkTask_reset);
210     ForkTask_init(t6, md2);
211     t7 = ttlTaskCreate(TASK_TYPE_PROCESS, pd2,
212        ForkTask1_main, ForkTask_reset);
213     ForkTask_init(t7, md2);
214     c4 = ttlChannelCreate(4, 0x400, md2);
215     c5 = ttlChannelCreate(4, 0x400, md2);
216     c6 = ttlChannelCreate(4, 0x40, md2);
217     c7 = ttlChannelCreate(4, 0x40, md2);
218     f0 = (ForkTask*)ttlTaskGetSpecific(t6);
219     f1 = (ForkTask*)ttlTaskGetSpecific(t7);
220     o1 = (OutputTask*)ttlTaskGetSpecific(t3);
221     ttlCbOutPortConnect(&m0->p1, t0, c6);
222     ttlCbOutPortConnect(&m0->p2, t0, c7);
223     ttlCbInPortConnect(&f0->p0, t6, c6);
224     ttlCbOutPortConnect(&f0->p1, t6, c2);
225     ttlCbOutPortConnect(&f0->p2, t6, c4);
226     ttlCbInPortConnect(&f1->p0, t7, c7);
227     ttlCbOutPortConnect(&f1->p1, t7, c3);
228     ttlCbOutPortConnect(&f1->p2, t7, c5);
229     ttlRnInPortConnect(&o1->p0, t3, c4);
230     ttlRnInPortConnect(&o1->p1, t3, c5);
231     ttlTaskContinue(t0);
232     ttlTaskContinue(t2);
233     ttlTaskStart(t3);
234     ttlTaskStart(t6);
235     ttlTaskStart(t7);
236  break;
...   ...
```

Figure 5. Part of the setUseCase function

6. IMPLEMENTATION OF SERVICES

The reconfiguration services and the case study application have been implemented on a chip called CaRaCas. This section describes the implementation of the API presented in Section 4. The CaRaCas chip contains one ARM microcontroller and four DSPs. All processors have different memories and different address spaces, but the ARM can access all memories. An overview of the architecture is shown in Figure 6. The DSPs have three memories each, program memory (PM), data memory (XM), and coefficient memory (YM).

This distributed architecture facilitates power efficient implementations of advanced audio and communications applications. The architecture however poses challenges to a programmer because of the different memories, different address spaces, and special inter-processor communication links. Therefore it is a good architecture to show that the TTL API can be implemented efficiently on heterogeneous multi-processor architectures.

The CM executes in a thread of its own on the ARM and streaming tasks execute on all five processors. All five processors have autonomous non-preemptive schedulers and can execute several threads. No operating system has been used in the implementation. The DSPs must execute autonomously to make the architecture scalable. If a central resource would have to be involved for scheduling of all signal processing tasks, that central resource would be the bottleneck of the system. The reconfiguration services are used infrequently and the architecture is scalable to a large number of DSPs although the CM executes on a central resource.

One of the DSPs has two schedulers, one that executes in the interrupt service routine (ISR) and one that executes in normal mode. A processor domain is assigned per scheduler, so there are totally 6 processor domains, called *pdARM*, *pd0*, *pd1*, *pd2*, *pd3*, and *pdISR*. The tasks in the *pdISR* are of task type actor. All other processor domains execute tasks of task type process.

There are five memory domains specified, one for each of the data memories, *md0*, *md1*, *md2*, *md3*, and *mdARM*.

6.1 Creation and Destruction

When a task is created in the processor domains *pdARM*, *pd0*, *pd1*, *pd2*, and *pd3*, a thread is created, but the thread is not enabled for execution until the task is started. When a task is destroyed, the thread is likewise destroyed. The CM specifies the processor domain in the *ttlTaskCreate* call, so that the thread is created on the correct processor. For every task there is a generic task data structure (see Figure 7), which contains information about the processor domain for the task. There is also a processor domain specific scheduling data structure (see Figure 7), which is used by the schedulers. Finally each task has a task specific data structure, which is dependent on the functionality of the task. The task specific data structure contains ports and variables that the task needs for its execution. The task specific data structure has to be created in the *initMethod* of the task.

In the *pdISR* processor domain, threads are not used. When a task is created in *pdISR*, a generic task data structure and a scheduling data structure for the ISR scheduler are allocated and initialized.

When a task is destroyed, the scheduling data structure and the generic task data structure are freed. Before a task is destroyed, the CM has called the *endMethod* of the task to free also the task specific data structure and any other memory allocated by the task.

Channels are simpler than tasks. When a channel is created, a channel data structure (see Figure 7), containing information needed for connecting and disconnecting ports to and from channels, is allocated in *mdARM* as it is accessed by the ARM only. Then a

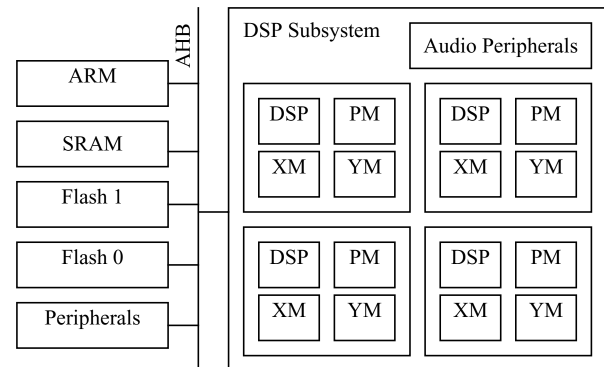

Figure 6. CaRaCas overview

channel buffer is allocated in the memory domain specified in the *ttlChannelCreate* call. When a channel is destroyed the memories are freed.

6.2 Connect and Disconnect

The channel administration in the CaRaCas chip is based on dual administration with two pointers [10]. The channel administration is stored in the port data structures, which are contained in the task specific data structures. When a connection is made, the port data structure is initialized with the base address of the channel buffer and the channel buffer size. The base address is different for different processors. The channel data structure (see Figure 7) contains information of connected ports and as soon as the second port is connected to a channel, a connection between the two port data structures is made by setting up remote pointers between them. A task may not be started before all active ports are connected.

When a port is disconnected from a channel, the channel data structure is updated and the remote pointers are invalidated in the port data structures.

6.3 Start, Pause, and Stop

Starting a task on a DSP is done by simply changing the status field in the scheduling data structure to RUNNING. The schedulers use round robin scheduling policies and inspect the status field before they activate a task. On the ARM, there is a queue of tasks which are in the RUNNING state, so when a task is started it is simply pushed on that queue.

Stopping and pausing of tasks is implemented with flags in the scheduling data structures (see Figure 7). The *ttlTaskStop* and *ttlTaskPause* functions in group 3 have blocking semantics, so they do not return until the tasks have acknowledged the pause or stop. The check functions that the tasks use to see if they should pause or stop simply return the corresponding flag from the scheduling data structure. The acknowledge functions are more complicated. On the DSPs the *ttlTaskAcknowledgeStop* function changes the status field to STOPPED and resets the point of execution.The *ttlTaskAcknowledgePause* function changes the status field to PAUSED and stores the point of execution. Then the scheduler immediately schedules the next task. So the *ttlTaskAcknowledgePause* function does not return until the CM has continued the task with a *ttlTaskContinue* call. The *ttlTaskAcknowledgeStop* function never returns. In the case of stopping, the *mainMethod* of the task is started from the beginning again when the task is restarted by the CM.

On the ARM, a task is removed from the queue of running tasks when it acknowledges a pause or stop command. At a stop command the point of execution is also reset.

The *ttlTaskForcedStop* function has a different implementation because it does not interact with the tasks. On the ARM the implementation is simple, the thread is removed from the ready queue. On the DSPs it is trickier because the tasks might be executing at the exact time of the *ttlTaskForcedStop* call. The *ttlTaskForcedStop* call cannot return until the task has stopped executing. Therefore the *ttlTaskForcedStop* function sets a flag in the scheduling data structure. This flag is checked by the DSP scheduler and acknowledged by changing the status field in the scheduling data structure to STOPPED. The *ttlTaskForcedStop* function, executing on the ARM, polls the status field and does not return until the status field has changed. During that polling phase, other threads are given the opportunity to execute on the ARM.

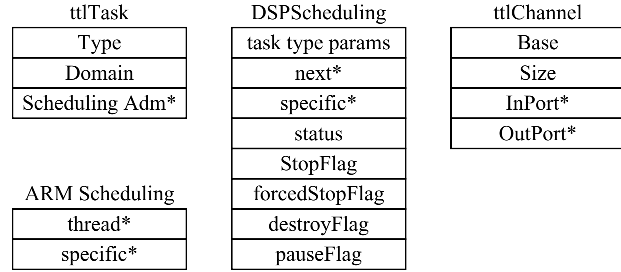

Figure 7. Data structures

6.4 Memory Management

Both tasks and channels can reside in any of the 5 data memories on the chip. The channel buffers must reside in the memory of the DSP where the consuming task executes because the DSPs can only read their local memories.

In the *ttlTaskCreate* and *ttlChannelCreate* functions, the memory domains are known and memory is allocated in the appropriate memory. A special memory allocation function *caracasMalloc* has been implemented, which next to the required number of bytes also gets a memory domain as input. Thereby it can allocate memory in the correct domain.

An additional requirement is that addresses are recomputed because potentially all five processors can access a single memory location using five different addresses. The ARM is responsible for translating its memory address map to the ones of the DSPs before filling in the scheduling data structures and configuring the ports. That also includes converting byte addresses used by the ARM to word addresses used by the DSPs.

7. RESULTS

The implementation of the reconfiguration services uses 224 bytes of program memory on the DSPs for the *ttlTaskCheckStop*, *ttlTaskAcknowledgeStop*, *ttlTaskCheckPause*, and *ttlTaskAcknowledgePause* functions. The program memory for the reconfiguration services implementations on the ARM can be seen in Table 2. The reason that the *ttlTaskContinue* function needs only one word is that it has the same implementation as the *ttlTaskStart* function and just calls that function. The reason that the *ttlCbOutPortConnect* function is more complex than the *ttlCbInPortConnect* function is that an output port can be connected to a channel which is located on another DSP and therefore an address remapping must take place. That address remapping is then calculated in the *ttlCbOutPortConnect* function.

Since the schedulers are non-preemptive, the response time for forcing a task to stop on the DSPs is dependent on the tasks themselves and on the channel sizes. The schedulers only execute when a task really blocks because of lack of data or room in a channel. Therefore a *ttlTaskForcedStop* call cannot be completed until a task blocks. On the other hand, a *ttlTaskPause* or *ttlTaskStop* call cannot be completed until a task comes to the part where it checks for and acknowledges a pause or stop command. This is true for all DSP schedulers except for the ISR scheduler. The ISR scheduler executes every time that there is an interrupt. Therefore the response time for reconfiguration calls to the tasks in the *pdISR* processor domain are dependent on the interrupt frequency. Any *ttlTaskForcedStop* call will be handled every interrupt period. Also the tasks in the *pdISR* processor domain typically have the property that if they

are reconfiguration-aware, they will acknowledge pause and stop commands on every activation. On the ARM a *ttlTaskForcedStop* call has no latency because it is known that the task does not execute at the same time as the CM. The response times of *ttlTaskStop* and *ttlTaskPause* are dependent on the tasks themselves just as on the DSPs.

Starting and continuing tasks has short response times because no acknowledgement is needed from either the tasks or the schedulers. A *ttlTaskStart* and *ttlTaskContinue* call uses 10-30 instructions dependent on which processor domain the task is located in. Tasks on the ARM use the most instructions because of the more complete multi-tasking implementation. Tasks on the DSPs use 10 instructions (*pd0*, *pd1*, *pd2*, and *pd3*) or 11 instructions (*pdISR*).

Table 2. Program memory usage on ARM

Function	Program memory (bytes)
ttlTaskCreate	392
ttlTaskDestroy	156
ttlChannelCreate	208
ttlChannelDestroy	44
ttlCb(Rn)InPortConnect	176
ttlCb(Rn)OutPortConnect	200
ttlCb(Rn)InPortDisconnect	48
ttlCb(Rn)OutPortDisconnect	48
ttlTaskStart	56
ttlTaskPause	140
ttlTaskContinue	4
ttlTaskStop	140
ttlTaskForcedStop	116
Total	1728

The reconfiguration-aware tasks call the *ttlTaskCheckPause* and *ttlTaskCheckStop* functions much more often than the corresponding acknowledge functions simply because reconfigurations do not happen often. The check functions on the DSPs use 5 instructions.

The connection of ports to channels uses 17-44 instructions on the ARM, dependent on whether a port was already connected to the other side of the channel or not and whether the producer and consumer execute on the same DSP or not. The disconnection of ports from channels uses 12 instructions on the ARM. The DSPs are not involved in connection and disconnection of ports to and from channels

One instruction on the DSPs executes in one clock cycle unless there is contention for the memory. On the ARM some instructions need more than one clock cycle to execute.

8. DISCUSSION

The concepts and the API presented in Section 3 and Section 4 can be used for easy description of applications as shown in the compact and simple CM in the case study. Partial task graph reconfigurations are possible, which helps to easily describe the dynamic behaviour of applications. Thereby the TTL API fulfils the first requirement, to allow for easy description of applications.

The TTL API can be efficiently implemented, which is shown by the less than 2 kB program code (ARM+DSP) in the CaRaCas implementation. The functions that are frequently used in the task execution, the check functions, use only 5 instructions. The other functions, used at reconfiguration points, also have short execution times. The connection of ports to channels is done without search operations thanks to the fact that the ports are part of the task specific data structure.

Processor domains and memory domains allow for implementation on different architectures. We mention here that our API has been implemented on other multi-processor architectures as well. The portability of the TTL API makes it a good candidate for standardization. A standard API allows for reusable reconfiguration-aware tasks and thereby improves design productivity.

9. CONCLUSIONS

We conclude that an API for dynamic stream processing applications can be efficiently implemented on a range of heterogeneous multi-processor architectures. The API can be standardized. By using a standard API, reusable reconfiguration-aware tasks can be designed. This allows for increased design productivity of dynamic streaming applications.

10. REFERENCES

[1] G. Kahn. The semantics of a simple language for parallel programming. In Information Processing (J.L. Rosenfeld, Ed. North-Holland Publishing Co.) 1974.

[2] E. A. Lee, D. G. Messerschmidt. Static scheduling of synchronous data flow graphs for digital signal processors. In Proceedings of the IEEE, vol. 75, pages 1235-1245, 1987.

[3] A. Nieuwland et al. C-HEAP: A Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems. Kluwer Journal of Design Automation for Embedded Systems, vol. 7, no. 3, pp. 233-270, ISSN 0929-5585, October 2002.

[4] K. Goossens. A Protocol and Memory Manager for On-chip Communication. In *ISCAS 2001 Conference Proceedings*, vol. II, pages 225-228. ISCAS, May 2001.

[5] M. Geilen, T. Basten. Reactive Process Networks, In *EMSOFT'04 Conference Proceedings*, pages 137-146. EMSOFT, September 2004.

[6] P. Yang et al. Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems. In *ISSS'02 Conference Proceedings*, pages 112-119. ISSS, October 2002.

[7] J. Sztipanovits, G. Karsai, T. Bapty. Self-Adaptive Software for Signal Processing. Communications of the ACM, Vol. 41, No. 5, pages 66-72, May 1998.

[8] G. A. Papadopoulos, F. Arbab. Dynamic Reconfiguration in Coordination Languages. In *HPCN'2000 Conference Proceedings*, pages 197-206, HPCN, May 2000.

[9] M. Rutten, E.-J. Pol, J. Eijndhoven, K. Walters, G. Essink. Dynamic recofiguration of streaming graphs on a heterogeneous multiprocessor architecture. In *SPIE Electronic Imaging: Embedded processors for Multimedia and Communications II*, vol. 5683, pp. 53-63, 2005.

[10] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, G. Essink. Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach. In *ISSS+CODES 2004 Conference Proceedings*, pages 206-217, ISSS+CODES, September 2004.

[11] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, B. Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *CASES'03 Conference Proceedings*, pages 63-72, CASES, October 2003.