Special Session

Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach

Pieter van der Wolf, Erwin de Kock, Tomas Henriksson, Wido Kruijtzer, Gerben Essink
Philips Research, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
Pieter.van.der.Wolf@philips.com

ABSTRACT

We present design technology for the structured design and programming of embedded multi-processor systems. It comprises a task-level interface that can be used both for developing parallel application models and as a platform interface for implementing applications on multi-processor architectures. Associated mapping technology supports refinement of application models towards implementation. By linking application development and implementation aspects, the technology integrates the specification and design phases in the MPSoC design process. Two design cases demonstrate the efficient implementation of the platform interface on different architectures. Industry-wide standardization of a task-level interface can facilitate reuse of function-specific hardware / software modules across companies.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures. C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems, signal processing systems.* D.1.3 [**Programming Techniques**]: Concurrent Programming.

General Terms

Design, performance, standardization.

Keywords

System design method, media processing, task-level interface, platform interface, multiprocessor mapping, code transformation.

1. INTRODUCTION

Modern consumer devices need to offer a broad range of functions at low cost and with low energy consumption. The core of such devices often is a multiprocessor System-on-Chip (MPSoC) that implements the functions as an integrated hardware / software solution. The integration technology used for building such MPSoCs from a set of hardware and software modules is typically based on low-level interfaces for the integration of the modules. For example, the usual way of working is to use bus interfaces for the integration of hardware devices, with ad-hoc mechanisms based on memory mapped registers and interrupts to synchronize hardware and software modules. Further, support for reuse is typically poor and a method for exploring trade-offs is often missing. As a consequence MPSoC integration is a labor-intensive and error-prone task and opportunities for reuse of hardware and software modules are limited.

Integration technology for MPSoCs should be based on an *abstract interface* for the integration of hardware and software modules. Such abstract interface should help to close the gap between the application models used for specification and the optimized implementation of the application on a multi-processor architecture. The interface must enable *mapping technology* that supports systematic refinement of application models into optimized implementations. Such interface and mapping technology will help to structure MPSoC integration, thereby enhancing both the productivity and the quality of MPSoC design.

We present design technology for MPSoC integration with an emphasis on three contributions:

1. We present TTL, a task-level interface that can be used both for developing parallel application models and as a platform interface for integrating hardware and software tasks on a platform infrastructure. The TTL interface makes services for inter-task communication and multi-tasking available to tasks.
2. We show how mapping technology can be based on TTL to support the structured design and programming of embedded multi-processor systems.
3. We show that the TTL interface can be implemented efficiently on different architectures. We present both a software and a hardware implementation of the interface.

After discussing related work in Section 2, we present the requirements for the TTL interface in Section 3. The TTL interface is presented in Section 4. Section 5 discusses the mapping technology, exemplified by several code examples. We illustrate the design technology in Sections 6 and 7 with two industrial design cases: a multi-DSP solution and a smart-imaging multi-processor. We present conclusions in Section 8.

2. RELATED WORK

Interface-based design has been proposed as a way to separate communication from behavior so that communication refinement can be applied [1]. Starting from abstract token passing semantics, communication mechanisms are incrementally refined down to the level of physical interconnects. In [2] and [3] a library-based approach is proposed for generating hardware and software wrappers for the integration of heterogeneous sets of components. The wrappers provide the glue to integrate components having different (low-level) interfaces. No concrete interface is proposed.

In [4] transaction level models (TLM) on the device or component level are discussed.

In contrast, we present an abstract task-level interface, named TTL, which can be implemented as platform interface. This interface is the target for the mapping of tasks. Previously, several task-level interfaces and their implementations have been developed at Philips [5][6][7]. TTL brings these interfaces together in a single framework, to unify them as a set of interoperable interface types.

The data transfer and storage exploration (DTSE) method [8] of IMEC focuses on source code transformation to optimize memory accesses and memory footprint. To our knowledge the method does not address the mapping of concurrent applications onto multiprocessor platforms. The Task Concurrency Management [9] method focuses on run-time scheduling of tasks on multiprocessor platforms to optimize energy consumption under real-time constraints. The interaction between these tasks is based on low-level primitives such as mutexes and semaphores. As a result, the tasks are less re-usable than TTL tasks and the design and transformation of tasks is more difficult and time consuming.

The Open SystemC Initiative [10] provides a modeling environment to enable system-level design and IP exchange. Currently, the environment does not standardize the description of tasks at the high level of abstraction that we aim at. However, TTL can be made available as a class library for SystemC in the future.

3. TTL INTERFACE REQUIREMENTS

We present a design method for implementing media processing applications as MPSoCs. A key ingredient of our design method is the Task Transaction Level (TTL) interface. On the one hand, application developers can use TTL to build executable specifications. On the other hand, TTL provides a platform interface for implementing applications as communicating hardware and software tasks on a platform infrastructure. The TTL interface enables mapping technology that automates the refinement of application models into optimized implementations. Using the TTL interface to go from specification to implementation allows the mapping process to be an iterative process, where during each step selected parts of the application model are refined. Figure 1 illustrates the basic idea, with the TTL interface drawn as dashed lines.

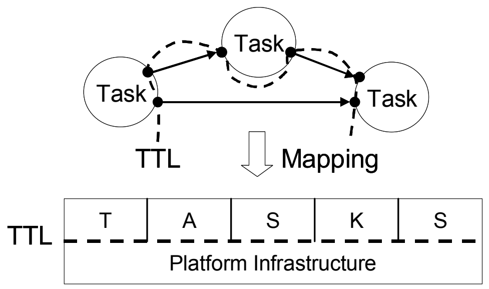

Figure 1: TTL interface for building parallel application models and implementing them on a platform infrastructure.

For the TTL interface to provide a proper foundation for our design method, it must satisfy a number of requirements. First it must offer well-defined semantics for modeling media processing applications. It must allow parallelism and communication to be made explicit to enable mapping to multi-processor architectures.

Further, the TTL interface must be an *abstract* interface. This makes the interface easy to use for application development because the developer does not have to consider low-level details. An abstract interface also helps to make tasks reusable, as it hides underlying implementation details. For example, if a task uses an abstract interface for synchronization with other tasks, it can be unaware and independent of the implementation of the synchronization with, for example, semaphores or some interrupt-based scheme.

The platform infrastructure makes *services* available to tasks via the TTL platform interface. Specifically, these are services for *inter-task communication*, *multi-tasking,* and *(re)configuration*. Rather than offering a low-level interface and implementing e.g. synchronization as part of all the tasks, we factor out such generic services from the tasks to implement them as part of the platform infrastructure. This implementation is done once for a platform, optimized for the targeted application domain and the underlying multiprocessor architecture.

An abstract platform interface provides freedom for implementing the platform infrastructure. It must allow a broad range of platform implementations, including different multiprocessor architectures. For example, both shared memory and message-passing architectures should be supported. Further, the abstraction allows critical parts of a platform implementation to be optimized transparently and enables evolution of a platform implementation as technology evolves. For example, smooth transition from bus-based interconnects towards the use of network-on-chip technology should be supported.

The TTL interface must allow *efficient* implementations of the platform infrastructure and the tasks integrated on top of it. To enable integration of hardware and software tasks, the interface must be available both as an API and as a hardware interface. An example of how the TTL interface could manifest itself in a simple multiprocessor architecture is shown in Figure 2.

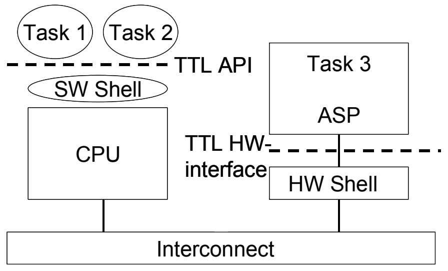

Figure 2: TTL interface as software API and as hardware interface in example architecture.

In the left part of Figure 2 the TTL interface is implemented as an API of a software shell executing on a CPU. Software tasks executing on the CPU can access the platform services via the API. In the right part of Figure 2 a task is implemented as an application-specific processor (ASP). The TTL interface for integrating the ASP is available as a hardware interface. A hardware shell implements the platform services on top of a lower interconnect. Such interconnect could, for example, have an interface like AXI [11], OCP [12], or DTL [13].

4. TTL INTERFACE

In this section we present the Task Transaction Level (TTL) interface. Specifically, we discuss the TTL interface for inter-task communication and multi-tasking services. We do not discuss reconfiguration. In this paper all task graphs are static.

4.1 Inter-Task Communication

We define the following terminology and associated *logical model* for the communication between tasks. The logical model provides the basis for the definition of the TTL inter-task communication interface. It identifies the relevant entities and their relationships. See Figure 3.

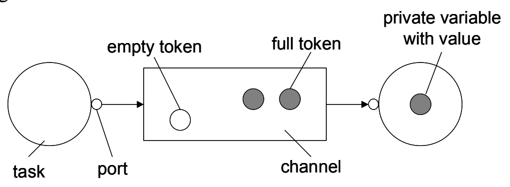

Figure 3: Logical model for inter-task communication.

A *task* is an entity that performs computations and that may communicate with other tasks. Multiple tasks can execute concurrently to achieve parallelism. The medium through which the data communication takes place is called a *channel*. A task is connected to a channel via a *port*. A channel is used to transfer *values* from one task to another. A *variable* is a logical storage location that can hold a value. A *private variable* is a variable that is accessible by one task only. A *token* is a variable that is used to hold a value that is communicated from one task to another. A token can be either *full* or *empty*. Full tokens are tokens that contain a value. Empty tokens do not contain a valid value, but merely provide space for a task to put a value in. We also refer to full and empty tokens as *data* and *room*, respectively.

Tasks communicate with other tasks by calling TTL interface functions on their ports. Hence, a task has to identify a port when calling an interface function. We focus on streaming communication: communicating tasks exchange sequences of values via channels. A set of communicating tasks is organized as a task graph.

4.1.1 Interface Types

Considering the varying needs for modeling media processing applications and the great variety in potential platform implementations, it is not likely that a single narrow interface can satisfy all requirements. For example, applications may process tokens of different granularities, where streams of tokens may or may not be processed strictly in order. Platform implementations may have different costs associated with synchronization between tasks, data transfers, and the use of memory. Certain architectures efficiently implement message-passing communication whereas shared memory architectures offer a single address space for memory-based communication between tasks.

In our view designers are best served if they are offered a palette of communication styles from which they can use the most appropriate one for the problem at hand. The TTL interface offers support for different communication styles by providing a set of different *interface types*. Each interface type is easy to use and implement. All interface types are based on the same logical model, which enables interoperability across interface types. A task designer must select an interface type for each port. Different interface types can be used in a single model, even in a single task. This allows models to be refined iteratively, where during each step selected parts of a model are refined.

In defining the interface types, we have to choose which properties to support and which properties to combine in a particular interface type. Some properties hold for all interface types. Specifically, all channels are uni-directional and support reliable and ordered communication. TTL supports arbitrary communication data types, but each individual channel can communicate tokens of a single type only. Multi-cast is supported, i.e. a channel has one producing task but can have multiple consuming tasks. The TTL interface types are listed in Table 1.

Table 1: TTL Interface Types.

Acronym	Full name
CB	Combined Blocking
RB	Relative Blocking
RN	Relative Non-blocking
DBI	Direct Blocking In-order
DNI	Direct Non-blocking In-order
DBO	Direct Blocking Out-of-order
DNO	Direct Non-blocking Out-of-order

4.1.2 Interface Type CB

The interface type CB provides two functions for communication between tasks:

```
write (port, vector, size)
read (port, vector, size)
```

The `write` function is used by a producer to write a vector of `size` values into the channel connected to `port`. The `read` function is used by a consumer to read a vector of `size` values from the channel connected to `port`. The `write` and `read` functions are also available as scalar functions that operate on a single value at a time. The `write` and `read` functions are blocking functions, i.e. they do not return until the complete vector has been written or read, respectively. This interface type is based on our earlier work on YAPI [5].

This interface type is the most abstract TTL interface type. Since it hides low-level details from the tasks, it is easy to use and supports reuse of tasks. The `write` and `read` functions perform both the synchronization and the data transfer associated with communication. That is, they check for availability of room/data, copy data to/from the channel, and signal the availability of data/room. The length of the communicated vectors may exceed the number of tokens in the channel. The platform implementation may transfer such vectors in smaller chunks, transparent to the communicating tasks [14]. This interface type is named CB as it combines (C) synchronization and data transfer in a single function with blocking (B) semantics.

This interface type can be implemented efficiently on message-passing architectures or on shared memory architectures where the processors have local buffers that can hold the values that are read or written. However, on shared memory architectures where the processors do not have such local buffers, this interface type may yield overhead in copying data between private variables, situated in shared memory, and the channel buffer in shared memory.

4.1.3 Interface Types RB and RN

To provide more flexibility for making trade-offs upon task implementation, the other TTL interface types offer separate functions for synchronization and data transfer. The availability of room or data can be checked explicitly by means of an *acquire* function and can be signaled by means of a *release* function. The acquire function can be blocking or non-blocking. A non-blocking acquire function does not wait for data or room to be available, but returns immediately to report success of failure. The functions for the producer are:

```
reAcquireRoom (port, count)
tryReAcquireRoom (port, count)
store (port, offset, vector, size)
releaseData (port, count)
```

`reAcquireRoom` is the blocking acquire function and `tryReAcquireRoom` is the non-blocking acquire function. The acquire and release functions synchronize for vectors of `count` tokens at a time. The acquire functions are named "reacquire" since they also acquire tokens that have previously been acquired and not yet released. That is, they do not change the state of the channel. This helps to reduce the state saving effort for tasks as the acquire function can simply be issued again upon a next task invocation. This behavior is similar to GetSpace in [6]. Data accesses can be performed on acquired room with the `store` function, which copies a vector of `size` values to the acquired empty tokens. The `store` function can perform out-of-order accesses on the acquired empty tokens using a *relative* reference `offset`. An offset of 0 refers to the oldest acquired and not yet released token. The `store` function is also available as a scalar function. The `releaseData` function releases the `count` oldest acquired tokens as full tokens on `port`.

The functions for the consumer are:

```
reAcquireData (port, count)
tryReAcquireData (port, count)
load (port, offset, vector, size)
releaseRoom (port, count)
```

These interface types are named RB and RN with the R of relative, B of blocking, and N of non-blocking.

Offering separate functions for synchronization and data transfer allows data transfers to be performed on a different granularity and rate than the related synchronizations. This may, for example, be used to reduce the cost of synchronization by performing synchronization at a coarse grain outside a loop, while performing computations and data transfers at a finer grain inside the loop. This interface type can be used to avoid the overhead of memory copies on shared memory architectures at a lower cost than with CB, as coarse-grain synchronization can be combined with small local buffers, e.g. registers, for fine-grain data transfers. Additionally, for some applications the support for out-of-order accesses helps to reduce the cost of private variables that are needed in a task. Further, with this interface type, tasks can selectively load only part of the data from the channel, thereby allowing the cost of data transfers to be reduced. The drawback, compared to CB, is that these interface types are less abstract.

4.1.4 Interface Type DBI and DNI

The RB and RN interface types hide the memory addresses of the tokens from the tasks. This supports reuse of tasks. However, it may also incur inefficiencies upon data transfers, like function call overhead, accesses to the channel administration, and address calculations. To avoid such inefficiencies, TTL offers interface types that support *direct* data accesses. In these interface types the acquire functions return a reference to the acquired token in the channel. This reference can subsequently be used by the task to directly access the data / room in the channel without using a TTL interface function. The functions for the producer are:

```
acquireRoom (port, &token)
tryAcquireRoom (port, &token)
token->field = value
releaseData (port)
```

The functions for the consumer are:

```
acquireData (port, &token)
tryAcquireData (port, &token)
value = token->field
releaseRoom (port)
```

The acquire and release functions acquire / release a single token at a time. Supporting vector operations for these interface types would result in a complex interface. For example, it would expose the wrap-around in the channel buffer or would require a vector of references to be returned. Since tasks must still be able to acquire more than one token, these acquire functions acquire the first un-acquired token and change the state of the channel, unlike the reacquire functions of RB and RN. The release functions release the oldest acquired token on `port`. The interface types are named DBI and DNI with the D of direct, B of blocking, N of non-blocking, and I of in-order as tokens are released in the same order as they are acquired. These interface types can be implemented efficiently on shared memory architectures [7] and are suited for software tasks that process coarse-grain tokens.

4.1.5 Interface Type DBO and DNO

In some cases tasks do not finish the processing of data in the same order as the data was acquired. In particular when large tokens are used, it should be possible to release a token as soon as a task is finished with it. For this purpose TTL offers the DBO and DNO interface types (O for out-of-order). The only difference with the DBI and DNI interface types is in the release functions:

```
releaseData (port, &token)
releaseRoom (port, &token)
```

The token reference allows the task to specify which token should be released. The out-of-order release supports efficient use of memory at the cost of a more complex implementation of the channel.

4.2 TTL Multi-Tasking Interface

To support different forms of multi-tasking, TTL offers different ways for tasks to interact with the scheduler. Thereto TTL supports three *task types*.

The task type *process* is for tasks that have their own (virtual) thread of execution and that do not explicitly interact with the scheduler. This task type is suited for tasks that have their private processing resource or that rely on the platform infrastructure to perform task switching and state saving implicitly. For example, this task type is well suited for software tasks executing on an OS.

The task type *co-routine* is for cooperative tasks that interact explicitly with the scheduler at points in their execution where task switching is acceptable. For this purpose TTL offers a `suspend` function. This task type may be used to reduce the task-switching overhead by allowing the task to suspend itself at points where only little state needs to be saved.

The task type *actor* is for fire-exit tasks that perform a finite amount of computations and then return to the scheduler, similar to a function call. Unless explicitly saved, state is lost upon return. This task type may be used for a set of tasks that have to be scheduled statically.

4.3 TTL APIs

The TTL interface is available both as C++ and as C API. The use of C++ gives cleaner descriptions of task interfaces, due to C++ support for templates and function overloading. We use C to link to software compilers for embedded processors and hardware synthesizers since most of them do not support C++ as input language. For both the C++ API and the C API we offer a generic run-time environment, which can be used for functional modeling and verification of TTL application models.

5. MULTIPROCESSOR MAPPING

In this section we present a systematic approach to map applications efficiently onto multiprocessors. The key advantage of TTL is that it provides a smooth transition from application development to application implementation. In our approach, we rewrite the source code of applications to improve efficiency. We focus on source code transformations for multiprocessor architectures taking into account costs of memory usage, synchronization cycles, data transfer cycles, and address generation cycles. We do not consider algorithmic transformations because these transformations are application-specific. Typically, application developers perform these transformations. We also do not consider code transformations for single target processors because these transformations are processor-specific. We assume that processor-specific compilers and synthesizers support these transformations, although in today's practice programmers also write processor-specific C.

In the remainder of this section we present methods and tools to transform source code. First we present source code transformations to illustrate the advantages of using TTL. Next we present tools that we developed to automate these transformations.

5.1 Source Code Transformation

We use a simple example to illustrate the use of TTL. The example consists of an inverse quantization (IQ) task that produces data for an inverse zigzag (IZZ) task; see Figure 4. We focus on the interaction between these two tasks.

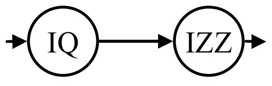

Figure 4: IQ and IZZ example.

The TTL interface supports different inter-task communication interface types that provide a trade-off between abstraction and efficiency. We illustrate this by means of code fragments. To save space we indicate scopes by means of indentation rather than curly braces.

5.1.1 Optimization for Single Interface Types

The most abstract and easy-to-use interface type is CB, which combines synchronization and data transfer in write and read functions. Figure 5 shows a fragment of the IQ task that reads input (Line 08), performs the inverse quantization (Line 09) and writes output using a scalar write operation (Line 10). The write function terminates when the value of variable Cout has been transferred to the channel. This is repeated for all 64 values of a block (Line 05) and for all blocks in a minimum coding unit (Line 03 and 04). Figure 6 shows a fragment of the IZZ task using a vector read function (Line 05). The read function terminates when 64 values from the channel have been transferred to the variable Cin. Subsequently, these values are reordered (Line 06 and 07) and written to the output (Line 08). The channel from the IQ task to the IZZ task implements the write and read functions that handle both the synchronization and the data transfer. Note that the length of the communicated vectors is not bounded by the number of tokens in the channel, which makes tasks independent of their environment.

```
01 void IQ::main()
02   while (true)
03     for(int j=0; j<vi; j++)
04       for(int k=0; k<hi; k++)
05         for(int l=0; l<64; l++)
06           VYApixel Cin;
07           VYApixel Cout;
08           read(CinP, Cin);
09           Cout = QTable[ti][l]*Cin;
10           write(CoutP, Cout);
```

Figure 5: IQ using interface type CB.

```
01 void IZZ::main()
02   while (true)
03     VYApixel Cin[64];
04     VYApixel Cout[64];
05     read(CinP, Cin, 64);
06     for (int i=0; i<64; i++)
07       Cout[zigzag[i]] = Cin[i];
08     write(CoutP, Cout, 64);
```

Figure 6: IZZ using interface type CB.

A potential performance problem of the IQ task in Figure 5 is that for each pixel, the output synchronizes with the input of the IZZ task. In [15] we demonstrated that this is costly in terms of cycles per pixel if the write function is implemented in software. We can solve this problem by calling the write function outside the inner loop as shown in Figure 7 in Line 10. To this end, we need to store a block of pixels locally in the IQ task (Line 05).

```
01 void IQ::main()
02   while (true)
03     for(int j=0; j<vi; j++)
04       for(int k=0; k<hi; k++)
05         VYApixel Cout[64];
06         for(int l=0; l<64; l++)
07           VYApixel Cin;
08           read(CinP, Cin);
09           Cout[l] = QTable[ti][l]*Cin;
10         write(CoutP, Cout, 64);
```

Figure 7: IQ using interface type CB and vector write.

Similar source code transformations to reduce the synchronization rate are possible for the other TTL interface types.

5.1.2 Optimization across Interface Types

The disadvantage of the IQ task of Figure 7 is the additional local memory requirement. Interface type RB splits synchronization and data transfer in separate functions such that the synchronization rate can be decreased without additional local memory requirements.

```
01 void IQ::main()
02   while (true)
03     for(int j=0; j<vi; j++)
04       for(int k=0; k<hi; k++)
05         reAcquireRoom(CoutP, 64);
06         for(int l=0; l<64; l++)
07           VYApixel Cin;
08           read(CinP, Cin);
09           store(CoutP, l, QTable[ti][l]*Cin);
10         releaseData(CoutP, 64);
```

Figure 8: IQ using interface type RB.

Figure 8 shows how to decrease the synchronization rate from pixel rate to block rate at the output of the IQ task. Note that here we assume that the channel can store at least 64 pixels, otherwise the call of the function reAcquireRoom at Line 05 will never terminate. This assumption on the environment is not needed with interface type CB. Hence, the IQ task of Figure 8 puts more constraints on its use.

```
01 void IZZ::main()
02   while (true)
03     VYApixel Cout[64];
04     reAcquireData(CinP, 64);
05     for(int i=0; i<64; i++)
06       VYApixel Cin;
07       load(CinP, i, Cin);
08       Cout[zigzag[i]] = Cin;
09     write(CoutP, Cout, 64);
10     releaseRoom(CinP, 64);
```

Figure 9: IZZ using interface type RB.

Figure 9 shows the IZZ task with separate synchronization and data transfer. The IQ task and the IZZ task do not need to store blocks locally to interact with each other. They share the tokens in the channel. If the IQ task and the IZZ task need to execute concurrently, then the channel must be able to contain two blocks, i.e., 128 pixels.

The load function (Figure 9, Line 07) and the store function (Figure 8, Line 09) use relative addressing. The advantage of this is that the address generation for the FIFO can be implemented in the load and store functions. Hence, address generation is hidden for the tasks.

Interface type DBI uses direct addressing rather than relative addressing. Direct addressing has advantages if the tokens of a channel and the variables of a task are stored in the same memory. In that case the tokens and the variables should be mapped onto the same memory locations to avoid in-place copying in the memory during the transfer of data from and to the tokens. Such copying occurs for instance in Figure 9 at Line 07 where a value from the channel is copied into variable Cin. Furthermore, the cost of calling the load and store functions can be avoided. The disadvantage of direct addressing is that the addresses of the tokens are exposed to tasks. To avoid that tasks must take care of wrap-around in the FIFO only scalar functions are available. Hence, typically it is more efficient to choose larger tokens if the synchronization rate has to be low. Figure 10 shows the IQ task using direct addressing on its output. We declare a pointer Cout in Line 04 that is given a value in Line 05. After the room has been acquired, Cout points to a block of 64 pixels. The channel data type is also block of 64 pixels. The pointer Cout is used to set the value of the pixels in Line 09 avoiding a call of a store function. Similarly, Figure 11 shows the IZZ task using direct addressing on its input avoiding both a call to a load function and a copy operation from the channel to a variable. Note that the granularity of synchronization between the IQ output and the IZZ input must be identical, because only scalar functions are available. For this reason, the IQ task and the IZZ task have become less re-usable.

```
01 void IQ::main()
02   for(int j=0; j<vi; j++)
03     for(int k=0; k<hi; k++)
04       VYApixel *Cout;
05       acquireRoom(CoutP, Cout);
06       for(int l=0; l<64; l++)
07         VYApixel Cin;
08         read(CinP, Cin);
09         Cout[l] = QTable[ti][l]*Cin;
10       releaseData(CoutP);
```

Figure 10: IQ using interface type DBI.

```
01 void IZZ::main()
02   while (true)
03     VYApixel *Cin;
04     VYApixel Cout[64];
05     acquireData(CinP, Cin);
06     for(int i=0; i<64; i++)
07       Cout[zigzag[i]] = Cin[i];
08     write(CoutP, Cout, 64);
09     releaseRoom(CinP);
```

Figure 11: IZZ using interface type DBI.

5.1.3 Non-Blocking Interface Types

So far, we only discussed interface types that provide blocking synchronization functions. These interfaces are easy to use because programmers do not have to program what should happen when access to tokens is denied. Sometimes blocking synchronization is not efficient, for instance, if the state of a task is large such that it is costly to save it. In that case it may be more efficient to let the programmer decide what should happen. For this reason, non-blocking synchronization functions are needed. Figure 12 shows how the IZZ task can be modeled as an actor. When the actor is fired, it first checks for available data on its input (Line 02) and then for available room on its output (Line 03). If the data is available but the room is not available, then the actor can return without saving its state. In the next firing, it can redo the checks since the tryReAcquire functions do not modify the state of the channels. If both the data and the room are available, it is guaranteed that the actor can complete its execution.

```
01 void IZZ::main()
02   if (!tryReAcquireData(CinP, 64)) return;
03   if (!tryReAcquireRoom(CoutP, 64)) return;
04   for (unsigned int i=0; i<64; i++)
05     VYApixel Cin;
06     load(CinP, i, Cin);
07     store(CoutP, zigzag[i], Cin);
08   releaseRoom(CoutP, 64);
09   releaseData(CinP, 64);
```

Figure 12: IZZ using interface type RN.

5.1.4 Channel and Task Merging and Splitting

Channel and task merging and splitting are important for load balancing. In [15] we applied task merging to reduce the data transfer load, since the cost of data transfer from the IQ task to the IZZ task is large compared to the amount of computation that the IZZ task performs. Figure 13 shows how the IQ task and the IZZ task can be merged.

```
01 void IQ_IZZ::main()
02   while (true)
03     for(int j=0; j<vi; j++)
04       for(int k=0; k<hi; k++)
05         VYApixel Cin[64];
06         VYApixel Cout[64];
07         read(CinP, Cin, 64);
08         for(int l=0; l<64; l++)
09           Cout[zigzag[l]]=QTable[ti][l]*Cin[l];
10         write(CoutP, Cout, 64);
```

Figure 13: Merged IQ and IZZ task.

The merging of the two tasks is based on the observation that the loop structure of the IZZ task fits in the loop structure of the IQ task. If one wants to merge two arbitrary tasks, this is not always the case. A more generic approach to statically schedule the firings of tasks is exemplified in Figure 14. The new task IQ_IZZ executes an infinite loop from which it calls the IQ and IZZ task by means of function calls. The communication between the IQ function and the IZZ function does not have to be synchronized explicitly because the calling order of the functions guarantees the availability of data and room. For this reason, we replace the channel by a variable mcu (minimum coding unit) that is declared in Line 03. The blocks in the mcu are passed by reference to the IQ function and the IZZ function.

```
01 void IQ_IZZ::main()
02   while (true)
03     VYApixel mcu[vi][hi][64];
04     IQ(mcu);
05     for(int j=0; j<vi; j++)
06       for(int k=0; k<hi; k++)
07         IZZ(mcu[j][k]);
08
09 void IQ_IZZ::IQ(mcu)
10   for(int j=0; j<vi; j++)
11     for(int k=0; k<hi; k++)
12       for(int l=0; l<64; l++)
13         VYApixel Cin;
14         read(CinP, Cin);
15         mcu[j][k][l] = QTable[ti][l]*Cin;
16
17 void IQ_IZZ::IZZ(block)
18   VYApixel Cout[64];
19   for(int i=0; i<64; i++)
20     Cout[zigzag[i]] = block[i];
21   write(CoutP, Cout, 64);
```

Figure 14: Statically scheduled IQ and IZZ tasks.

5.2 Automated Transformation

We aim to automate the above-mentioned source code transformations to support the proposed method by tools. It is not the goal to automate the design decision making process, because experiences in high-level synthesis and compilation tools show that it is hard to automate this while maintaining transparency for users. Our goal is to automate the rewriting of the source code according to the design decisions of users. This approach has two advantages. First, design decisions are explicitly formulated rather than implicitly coded in the source code. Second, the source code can be rewritten automatically such that modifications and bug fixes in the original specification can be inserted automatically in architecture-specific versions of the code. In this way a large set of design descriptions can be kept consistent.

5.2.1 Parser Generation

The first step in automatic source code transformation is to be able to parse programs and to build data types that support source code transformation. For this purpose, we use an in-house tool called C3PO in combination with the publicly available parser generator tool ANTLR [17]. C3PO takes a grammar as input and synthesizes data types for the non-terminals in the rules of the grammar as well as input for ANTLR. We use C3PO and ANTLR to generate a C++ parser and a heterogeneous abstract syntax tree (AST). We use the same tools to generate visitors for the AST that transform the code. After transformation, we generate new C++ or C code from the AST. The transformations that we target are typically inter-file transformations. For this reason, we process all source files simultaneously as opposed to the usual single file compilation for single processors.

5.2.2 Iterative Transformation

Source code transformation typically is an iterative process in which many versions of the same program are generated. Automatic source code transformation has the advantage that the generated source code is consistently formatted and that the transformations can be repeated if necessary. This makes it possible to keep all versions of a program consistent automatically. For version management we have adopted CVS. Each iteration uses three versions of the source code. The first version is the result of the previous iteration or the original code if it is the first iteration. The second version is manually augmented source code that is the input for the automatic transformation. The augmentation can contain for instance design constraints and design decisions. The third version is the code that is automatically generated. If the original code changes, for instance, due to bug fixes or specification changes, then the changes can be automatically inserted in the second version of the code by the version management tool. The modified second version of the code is then given as input to the transformation tools in order to produce the third version of the code that is the starting point for the next iteration.

5.2.3 Automatic Interface Type Refinement

We illustrate automatic interface refinement using the example of IQ and IZZ. The original source code of the tasks is given in Figure 5 and Figure 6. The resulting code is given in Figure 8 and Figure 9. The complete code is distributed over six files: a source file and a header file for the definition of each of the two tasks, and a source file and a header file for the definition of the task graph that instantiates and connects the two tasks. All these files require changes if the communication between the two tasks changes. This has been automated in the following way. We augment the source code of the tasks with synchronization constraints. In Figure 5 between Line 04 and 05 we add the line ENTRY(P) and at the end of the text we add the line LEAVE(P), both in the scope of the loop in Line 04. This annotation means that we want to synchronize the output of the IQ task on blocks of 64 pixels. Similarly we add synchronization constraints ENTRY(C) and LEAVE(C) to the IZZ task in Figure 6 between Line 04 and 05 and at the end of the text, respectively, both in the scope of the loop of Line 02. Assuming that the channel between the two tasks is called iqizzbuf, we provide the transformation tool with the design information shown in Figure 15.

```
01 iqizzbuf[“64”]
02   Channel<VYApixel> USING RbIn, RbOut
03     “64”*IZZ::C <= “64”*IQ::P
04     “64”*IQ::P <= “64”*IZZ::C+64
05 STORAGE IQ
06   Cout -> ../iqizzbuf TRANSFORMATION T1
07 STORAGE IZZ
08   Cin -> ../iqizzbuf TRANSFORMATION T2
09 SYNCHRONIZATION
10   IQ, IZZ -> iqizzbuf
```

Figure 15: Design constraints and decisions.

This information means that we want the iqizzbuf channel to have 64 tokens (Line 01). Furthermore, the channel should be implemented in data type Channel, it should handle tokens of type VYApixel, and it should connect to interface type RB both for output and for input (Line 02). Line 03 and 04 denote the synchronization constraints: the amount of consumption should not exceed the amount of production but the difference between the amount of production and consumption may not exceed the buffer capacity of the channel. Line 06 and 08 denote that the variables Cout and Cin of the IQ task and the IZZ task, respectively, should be mapped on the iqizzbuf channel using Transformation T1 and T2 that are available in a library. This introduces the calls to load and store functions. The result of the call to the load function in the IZZ task is stored in a new variable, also called Cin. Line 10 denotes that the IQ task and the IZZ task should be synchronized using the iqizzbuf channel. This introduces the calls to acquire and release functions at the positions indicated by the ENTRY and LEAVE annotations in the augmented source code. The resulting source code is given in Figure 8 and Figure 9.

5.2.4 Processor and Channel Binding

The last phase of source code transformation is the link to existing compilers and synthesizers in order to map the individual tasks to hardware and software. To this end, programmers specify a binding of tasks to processor types and/or processor instances. From that information the necessary input, i.e. C files and makefiles, for compilation or synthesis to the target processor is generated. Furthermore, the programmer specifies specific implementations of channels. For instance, the same interface type can be implemented differently for intra-processor communication and for inter-processor communication because of efficiency reasons. Each implementation has its own set of names for its interface functions since function overloading is not available in C. The generated C code contains the data types and function calls that correspond to the implementations of the channels that the programmer has chosen.

5.2.5 Other Transformations

There are other transformations that are beyond the scope of this paper. We briefly mention them here. We support structure transformation to change the hierarchy of task graphs. We support instance transformations such that multiple instances of the same task or task graph can be transformed individually. Finally, we plan to support channel and task merging and splitting [15] by connecting to the Compaan tool suite [16].

6. TTL ON AN EMBEDDED MULTI-DSP

In this section we present the implementation of TTL on a multi-DSP. The objectives are to show 1) how TTL can be implemented and that a TTL implementation is cheap, 2) trade-offs between the implementation cost and the abstraction level of the TTL interfaces, and 3) how TTL supports the exploration of trade-offs between e.g. memory use and execution cycles. The TTL implementation is done without special hardware support. We first present the multi-DSP architecture. Then we describe how the implementation of five TTL interface types has been done and we present quantitative results. Finally, the results for an implementation of an MP3 decoder application are presented.

6.1 The Multi-DSP Architecture

The embedded multi-DSP is a template that allows an arbitrary number of DSPs [18]. Each DSP has it own memory, which in limited ways can be accessed by (some of) the other DSPs. A DSP with memory and peripherals is called a DSP subsystem (DSS), see Figure 16. The DSPs do not have a shared address space. Communication between the DSSs is done through memory mapped uni-directional point-to-point links. Thus, two DSPs may refer to a single memory location with different addresses. Data may be routed from one point-to-point link to another and so on until it reaches its destination.

In our instance, the DSP Epics7B from Philips Semiconductors was used. The DSP, which is mainly used for audio applications, has a dual Harvard architecture with 24 bits wide data path and 12 bit coefficients.

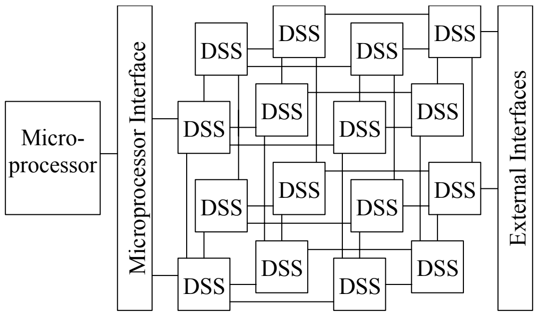

Figure 16: Multi-DSP architecture. Here an instance with 16 DSP subsystems is shown.

6.2 TTL Implementation

There are two criteria to decide which TTL interface type to use for a certain application on a certain architecture. First the interface type must match the application characteristics. Second, the implementation of the interface type on the target architecture must be efficient.

For audio applications, DBO and DNO are not needed because audio applications do not have large amounts of data that are produced or consumed out-of-order. Therefore, the other five interface types have been implemented on the multi-DSP architecture in order to determine the cost of the implementations. Most of the TTL functions have been implemented in optimized assembly code. It is justifiable to spend the effort because the TTL functions are implemented only once and used by many applications.

A TTL channel implementation consists of two parts, the channel buffer and the channel administration. In the multi-DSP architecture, no special-purpose memories exist, so the channel buffer is a circular buffer in a RAM. This is where the tokens are stored. The channel administration is a structure that holds information about the state of the channel. In the multi-DSP architecture, the channel buffer has to be located in the memory of the DSS where the consumer is executed. This is due to the uni-directional point-to-point links in the architecture.

6.2.1 Channel Administration

The channel administration keeps track of how many tokens there are in the channel and how many of those are full and empty respectively. It also provides a way to get the next full and the next empty token from the channel. When the channel buffer is implemented as a circular buffer in a RAM, the channel administration can be implemented in two different ways with two variables to keep track of the state of the channel. The first alternative is to use two pointers, one to the start of the empty tokens and one to the start of the full tokens. The second alternative is to have one pointer and one counter, for example a pointer to the start of the full tokens and a counter telling how many full tokens there are in the channel. This requires atomic increment and decrement operations, which are not supported on the multi-DSP architecture. Therefore the channel administration is implemented with two pointers. The producer updates the pointer to the empty tokens (WRITE_POINTER) and the consumer updates the pointer to the full tokens (READ_POINTER) and thereby no atomic operations are needed [7]. Another method to avoid the need for atomic updates is to use two counters and two pointers. That method is explained in Section 7.

When the two pointers point to the same memory location, it is not clear if the channel is full or empty unless wrap-around counters are used. Wrap-around counters imply expensive acquire functions. To avoid that problem we have implemented a channel administration that does not allow the pointers to point to the same memory location unless the channel is empty. We thereby have a memory overhead of the size of one token in the channel buffer. In the indirect interfaces the token size is always one word.

Both the producer and the consumer need to access the channel administration. In the multi-DSP there are no shared memories, therefore the channel administration has to be duplicated and present in the two DSSs involved in the communication. The two copies are called the local and remote channel administration. See Figure 17. Since the producer and the consumer refer to the channel buffer with different addresses, this must be taken into consideration when updating the remote channel administration. We keep a pointer to the base address in the local address space (BASE_POINTER) and a pointer to the base address in the remote address space (BASE_RA). These two pointers are used to calculate the pointer value to be stored in the remote channel administration. The channel administrations as well as the channel buffer must be located in memory areas that are accessible via the point-to-point links.

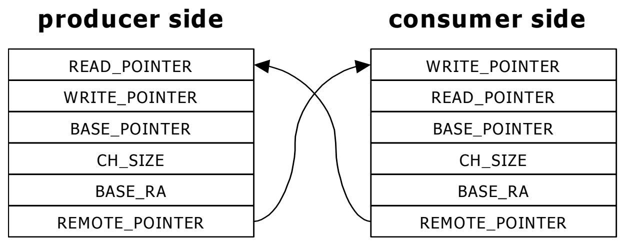

Figure 17: Double channel administration for the indirect interface types.

As an example of the implementation of the TTL functions, pseudo code for the tryReAcquireData function in RN is shown in Figure 18.

```
Boolean tryReAcquireData(port p, uint n) {
  uint available_data;
  available_data = (p->write_pointer –
            p->read_pointer) modulo p->ch_size;
  if (available_data >= n)
    return true;
  else
    return false;   }
```

Figure 18: Pseudo code for tryReAcquireData (RN).

6.3 Implementation Results

The acquire functions for the RN interface type use 9 instructions. The release functions use 15 instructions. The vector load and store functions use a loop unrolling of 2 and achieve 2.5 instructions per data word with an overhead of 24 instructions to set up the data transfer. The scalar load and store functions are in-lined in the task code and use each 10 instructions.

The acquire functions for the direct interface type DNI use between 19 and 33 instructions, dependent on the state of the channel. The release functions use between 29 and 38 instructions. No data transfer functions are used. The cost of the data transfers is comparable to the cost of accessing private data structures in the task.

For the blocking interface types, it is not as easy to determine the cost in terms of instructions for the individual acquire functions, because they may include task switches. However, an acquire function in RB, that does not trigger a task switch uses 18 instructions. The release functions and the data transfer functions for RB have the same cost as those for RN. The same applies to the release functions of DBI with respect to DNI.

In CB, synchronization and data transfer is combined into one single function. The cost of the implementation is approximately the sum of the costs of the three corresponding functions in RB.

6.3.1 Evaluation Application

An MP3 decoder application has been used for the evaluation of the TTL implementations on the multi-DSP. The MP3 decoder application was available as a sequential C program. The application was converted into a TTL task and additional TTL tasks were added for mimicking the rest of a complete application and for handling the interaction with the simulation environment.

The application has been implemented with all five interface types. The RN and DNI implementations use TTL actors and the other types use TTL processes. The application has also been implemented with four different granularities of the communication for the RN interface type. In the implementations with the direct interface types, DNI and DBI, the channel between the input task and the MP3 task uses RN and RB interface types respectively. This is due to the fact that the amount of communicated data on that channel is data dependent.

6.3.2 Simulation Results

Table 2 shows the results of the various interface types with frame-based communication. All channel buffers have been sized so that they can hold one frame. The memory is the total data memory for the whole application. The number of cycles is the number used by the whole application to decode a test file.

Table 2: Simulation results for the whole application.

TTL IF Type	#Cycles	Part in TTL	#Memory words
CB	4557960	2.9%	12493
RB	4555124	2.8%	12494
RN	4550595	2.2%	12365
DBI	4515245	1.1%	9162
DNI	4510808	0.5%	9041

The blocking implementations use somewhat more memory and have some cycle overhead compared to the non-blocking implementations, when comparing RB to RN and DBI to DNI. This is due to the fact that the multi-tasking costs both memory for storing register contents and cycles to save and restore the register contents. The DNI and DBI interface type use considerably less memory and less cycles than the other interface types. This is because the data in the channels is accessed directly, without copying it to and from private variables. The CB version has similar performance as the RB version.

For the DNI implementation, about 0.5% of the cycles are spent in the TTL functions and 99.5% of the cycles are spent in the tasks. This is of course dependent on the application as well as on the implementation of the TTL functions.

Figure 19 shows the trade-offs that can be made by changing the granularity of the communication. Here a change of a factor of 36 has been pursued on the channel between the MP3 task and the output task. In the MP3 decoder, this is made possible by using a sub-frame decoding method, which allows the MP3 decoder to output blocks smaller than a frame.

The memory is reduced both in the channel buffer, in the MP3 task and in the output task. The channel buffer sizes have been adjusted to match the granularity of the communication. The cycle overhead for the small granularity communication has two reasons. Smaller granularity implies more frequent synchronization calls and smaller buffers imply more frequent task switching.

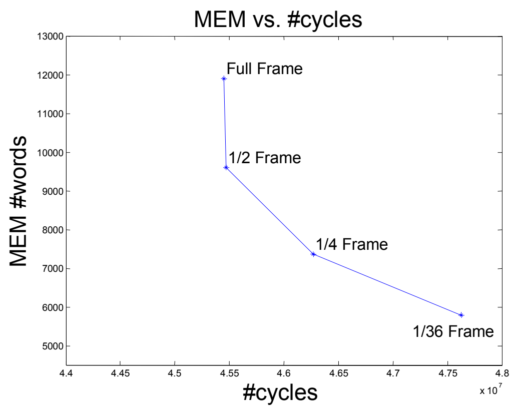

Figure 19: Simulation results for RN, when changing the communication granularity.

The implementation of CB allows channel buffers to be smaller than the vector sizes used by the tasks. One of the advantages with CB is that the channel buffer size can be reduced to achieve a memory-cycle trade-off without rewriting the tasks themselves. Results for this are shown in Figure 20.

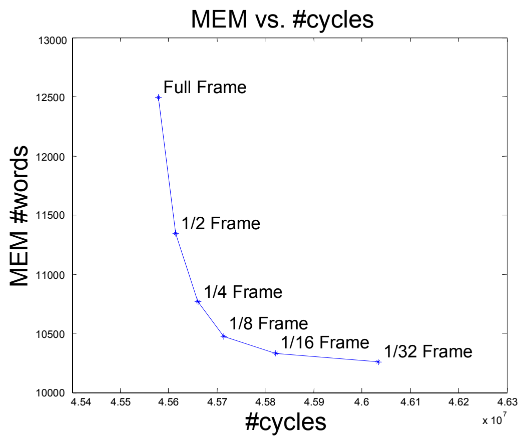

Figure 20: Simulation results for CB, when changing the channel buffer size.

6.4 Implementation Conclusions

It has been shown that TTL can be implemented efficiently on a multi-DSP architecture. It has also been shown that changing the granularity of the communication of the tasks has great impact on the memory-cycle trade-off. The direct interfaces in TTL provide benefits both regarding the memory usage and the cycle overhead. As expected, the most abstract interface type, CB, is also the most expensive to use. This proves the value of automating transformations between the various implementation alternatives.

7. TTL IN A SMART IMAGING CORE

The objective of this section is to show that the implementation of TTL in hardware, software, and mixed hardware / software is possible with reasonable costs. The implementation allows the buffer size and the buffer location to be changed and the channel administration to be relocated. This section first discusses the smart imaging core followed by a detailed description of the TTL implementation including performance results.

7.1 The Smart Imaging Core

Smart imaging applications combine image and video capturing with the processing and/or interpretation of the scene contents. An example is a camera that is able to segment a video sequence into objects, track some of them, and raise an alarm if some of these objects show an unusual behaviour. The smart imaging core described here can be embedded in a camera and is suited for automotive and mobile communication applications. Example applications are pedestrian detection [19], low-speed obstacle detection [20], and face tracking.

Each of the smart imaging applications uses low-level pixel processing, typically on image segments, for an abstraction of the scene contents (feature extraction). Furthermore, motion segmentation is used to help in tracking objects in the scene. The applications are structured such that the more control-oriented parts are combined together in a task that fits well on a CPU. All the low-level pixel processing is combined together in a pixel processing task, which is mapped onto a smart imaging coprocessor. Likewise, the main processing part of the motion segmentation is described as an independent task, which is mapped onto a motion estimator coprocessor. The architecture of the smart imaging core is depicted in Figure 21.

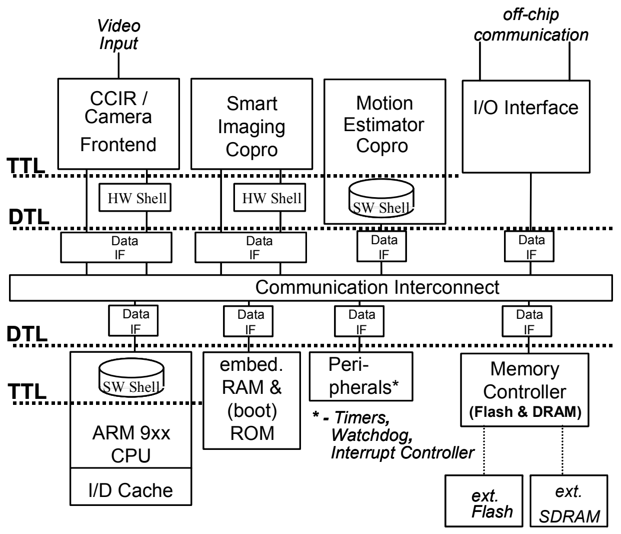

Figure 21: Architecture of the smart imaging core.

More details of the architecture can be found in [21]. The architecture globally consists of an ARM CPU, a video input unit, and two coprocessors: the motion estimator (ME) and the smart imaging (SI) coprocessor. The tasks on the coprocessors and the ARM communicate with each other using the TTL interface. By adopting the TTL interface for the coprocessors, we expect that the integration of these blocks into future systems will be significantly simplified.

7.2 TTL SHELLS

This subsection presents the TTL shells used for the smart imaging core. These are a full hardware shell for the SI coprocessor and software shells for the ARM and the motion estimator (VLIW) coprocessor.

7.2.1 *TTL Shell for the SI Coprocessor*

The TTL interface type used for the SI is the RB interface type using indirect data access. As already explained in the Multi-DSP section a TTL channel implementation consists of two parts, the channel buffer and the channel administration. In the SI core the channel buffers are always located in main (on-chip) memory. The channel administration can be placed both in the shell and in main memory. We also use two copies of the channel administration; one at the producer side and another at the consumer side. Figure 22 depicts the channel administration structure.

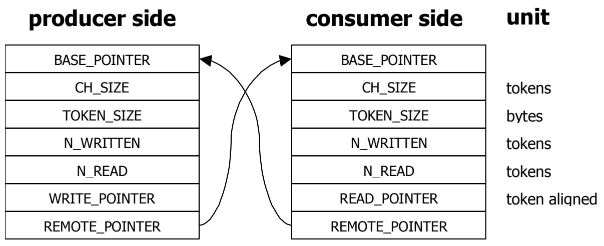

Figure 22: Channel administration.

To make sure that the channel status is handled correctly by both the producer and consumer, without the need for atomic access to the variables of the channel administration, N_WRITTEN, N_READ, READ_POINTER, and WRITE_POINTER are used. Only the producer modifies N_WRITTEN and WRITE_POINTER. Similarly only the consumer modifies N_READ and READ_POINTER. The equation "N_WRITTEN – N_READ" is used to calculate the amount of data available in the channel while "CH_SIZE – (N_WRITTEN – N_READ)" is used to derive the amount of free room. The hardware implementation includes correct handling of wraparounds. With this approach both the consumer and producer have a conservative view on the channel status. The use of two token counters N_READ and N_WRITTEN instead of two pointers as in the Multi-DSP case is due to the variable TOKEN_SIZE that can be handled with this implementation. Using the counters the implementation of the acquire functions is more efficient because no multiplication with TOKEN_SIZE is needed. The variable REMOTE_POINTER is used to reference the remote channel administration. The variable BASE_POINTER together with the OFFSET parameter provided through the TTL load and store calls is used to calculate the physical address for accessing the channel buffer. The buffer behaves as a FIFO and is implemented as a fixed size (CH_SIZE) circular buffer. This results in the equation "ADDRESS = BASE_POINTER + (READ_POINTER + OFFSET * TOKEN_SIZE) % CH_SIZE" for each read access.

The hardware implementation consists of a hardware shell with two interfaces. A TTL signal interface connects the shell to a coprocessor. See Figure 23. A DTL interface connects the shell to the communication interconnect. The *request* and *acknowledge* signals are used to handshake a TTL call from the coprocessor to the shell. The shell is able to handle both input (*port_type=0*) and output ports (*port_type=1*). Both the RB and RN interface types are supported through the *is_non_blocking* and *is_granted* signals. The signal *is_granted* indicates if access is granted in non-blocking acquire operations. The *esize* signal indicates the vector length (max. 2^{ns} tokens). The *wr_* and *rd_* signals are used to handshake the load and store data between the coprocessor and shell. In total 2^{np} logical ports (*port_id*) are handled independently by the shell. The TTL calls for these ports are however handled sequentially as each TTL call has to first finish by reasserting the acknowledge signal, hence only one set of *port_id*, *offset, etc.* signals is provided. Usually this limitation is acceptable as the number of concurrent ports that can be handled efficiently is also limited by the physical connection of the shell to the DTL infrastructure (see Figure 21). Using one DTL interface with multiple sets of TTL signals complicates the shell significantly. In that case multiple state-machines have to arbitrate for the DTL access. Obviously also multiple DTL interfaces for the shell could be used. This however is similar to just using multiple (simple) shells for one coprocessor as we provide.

The shell architecture consists of an indirect addressing unit, a control unit, two ALUs, and a DTL interface unit. Furthermore, it includes a table that maps *port_ids* to their respective channel administration. The table brings the flexibility to map the channel administration at an arbitrary location (even within the shell itself). For the smart imaging core this flexibility enables the designer to optimize the communication infrastructure per application. For a channel between tasks on the ARM (producer) and SI (consumer), the channel buffer and administration of the producer side are mapped in main memory, while the consumer side administration is mapped in the hardware shell. This mapping minimizes the time spent for the acquire functions as the administrations are distributed and mapped locally to the producer and consumer. The performance of the hardware shell with the channel administration local in the shell is as follows: the acquire functions use 5 cycles, the release functions use 7 cycles, the load function uses 5 + 2*n* cycles, and the store function uses 5 + *n* cycles. The parameter *n* is the number of tokens specified in the TTL call. In total, the implementation of a shell for the SI coprocessor with 2 logical ports, synthesized for 100MHz, takes ~0.2 mm^2 (~8.3 Kcells) in 0.18µ CMOS technology.

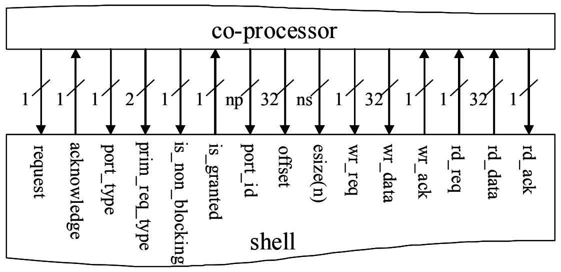

Figure 23: TTL signal interface.

7.2.2 *TTL Shell for the ME and ARM*

The motion estimator coprocessor is synthesized using a high-level architectural synthesis tool called A|RT Designer. A|RT Designer takes as input a C description of an algorithm and generates a custom VLIW processor, consisting of a data path and a controller. The controller contains an FSM, which determines the next instruction to be executed, and a micro-code ROM memory, where the motion estimation main task is stored. The data path is parameterizable in the number and the type of functional units (FUs) used. The communication to the external system is arranged via input and output FUs that implement a standard DTL interface towards the system and have a simple register file interface inside the VLIW. The output of A|RT Designer is a synthesizable RTL description of the generated VLIW processor. Instead of implementing the TTL shell for the ME completely in hardware as an FU, the implementation choice for the ME is to use the A|RT high-level synthesis tool for the implementation of both the motion estimation main process and the shell. In this case the implementation of the TTL functions is by means of C-code that is executed on the ME VLIW. This is achieved by compiling the TTL C-code together with the design description of the ME. The A|RT high-level synthesis tool adds the TTL implementation into the micro-code of the coprocessor, and executes it as part of the VLIW program. The physical communication is done via the FUs that provide standard DTL interfaces.

The ARM software implementation is a simple C-code compilation of the TTL functions. The number of ARM instructions used is: 40 for the acquire functions; 42 for the release functions; 27 for (scalar) load and store functions (6 extra for each element of a vector).

8. CONCLUSIONS

We have presented an interface-centric design method based on a task-level interface named TTL. TTL offers a framework of interoperable interface types for application development and for implementing applications on platform infrastructures. We have shown that the TTL interface provides a basis for a method and tools for mapping applications onto multiprocessors. Furthermore, we have demonstrated that TTL can be implemented efficiently on two different architectures, in hardware and in software.

Industry-wide standardization of a task-level interface like TTL can help to establish system-level design technology that supports efficient MPSoC integration with reuse of function-specific hardware and software modules across companies. Future extensions of TTL will be concerned with the modeling of timing constraints and associated verification technology.

9. ACKNOWLEDGEMENTS

We acknowledge the contributions of Jeffrey Kang, Ondrej Popp, Dennis Alders, Avneesh Maheshwari, and Ghiath Alkadi from Philips Research and of Victor Reyes from the University of Las Palmas. The work on smart imaging is partly sponsored by the European Commission in the IST-2001-34410 CAMELLIA project.

10. REFERENCES

[1] Rowson, J.A., A. Sangiovanni-Vincentelli, “Interface-Based Design”, Proc. 34th Design Automation Conference, 1997.

[2] Cesário, W.O., D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya, L. Gauthier and M. Diaz-Nava. “Multiprocessor SoC Platforms: A Component-Based Design Approach,” IEEE Design & Test, Nov.-Dec. 2002, pp. 52-63.

[3] Dziri, M-A, W. Cesário, F.R. Wagner, A.A. Jerraya, “Unified Component Integration Flow for Multi-Processor SoC Design and Validation”, DATE-04, pp. 1132-1137, 2004.

[4] Cai, L, D. Gajski, “Transaction Level Modeling: An Overview”, CODES+ISSS 2003, pp. 19-24.

[5] De Kock, E.A., G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, K.A. Vissers, "YAPI: Application Modeling for Signal Processing Systems," Proc. 37th DAC, pp. 402-405, 2000.

[6] Rutten, M.J., J. van Eijndhoven, E. Jaspers, P. van der Wolf, O.P. Gangwal, A. Timmer, E.J. Pol, “A Heterogeneous Multiprocessor Architecture for Flexible Media Processing”, IEEE Design and Test, pp. 39-50, July-August 2002.

[7] Nieuwland, A., J. Kang, O. P. Gangwal, R. Sethuraman, N. Busa, K. Goossens, R. P. Llopis, P. Lippens, “C-HEAP: A Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems,” Journal of Design Automation for Embedded Systems, vol. 7, no. 3, pp. 229-266.

[8] Catthoor, F., K. Danckaert, C. Kulkarni, E. Brockmeyer, P.G. Kjeldsberg, T. Van Achteren, T. Omnes, "Data access and storage management for embedded programmable processors," ISBN 0792376897, Kluwer, 2002.

[9] Wong, C., P. Marchal. P. Yang, “Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform,” Proc. CODES 2001, pp. 170-177, 2001.

[10] Grotker, T., S. Liao, G. Martin, S. Swan, “System Design with SystemC,” ISBN 1402070721 Kluwer, 2002.

[11] ARM, “AMBA AXI Protocol Specification,” June 2003.

[12] OCP International Partnership, “Open Core Protocol Specification, 2.0 Release Candidate,” 2003.

[13] Philips Semiconductors, “Device Transaction Level (DTL) Protocol Specification, Version 2.2,” July 2002.

[14] Brunel, J.-Y., W.M. Kruijtzer, H.J.H.N. Kenter, F. Petrot, L. Pasquier, E.A. de Kock, W.J.M. Smits, “COSY Communication IP's,” Proc. 37th DAC, pp. 406-409, 2000.

[15] De Kock, E.A., "Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study," Proc. Intl. Symposium on System Synthesis (ISSS), pp. 68-73, 2002.

[16] Cimpian, I., A. Turjan, E.F. Deprettere, E.A. de Kock, “Communication Optimization in Compaan Process Networks,” Proc. 4th International Workshop on Systems, Architectures, Modeling and Simulation (SAMOS), 2004.

[17] http://www.antlr.org/

[18] Schiffelers, R., et al., “Epics7B: A lean and mean concept,” Proceedings of International Signal Processing Conference 2003, Dallas, TX, March 31 - April 3.

[19] Abramson, Y., B. Steux, “Hardware-friendly pedestrian detection and impact prediction,” IEEE Intelligent Vehicle Symposium 2004, Parma, Italy , June 2004.

[20] Steux, B., Y. Abramson, “Robust real-time on-board vehicle tracking system using particles filter,” IFAC Symposium on Intelligent Autonomous Vehicles, July 2004.

[21] Gehrke, W., J. Jachalsky, M. Wahle, W. Kruijtzer, C. Alba, R. Sethuraman, “Flexible coprocessor architectures for ambient intelligent applications in the mobile communication and automotive domain,” Proc. SPIE VLSI Circuits and Systems, vol. 5117, pp. 310-320, April 2003.