TTL inter-task communication implementation on a shared-memory multiprocessor platform

Bei Li+, Pieter van der Wolf* and Koen Bertels+
Philips Research Eindhoven*, Computer Engineering Lab. TU Delft+, The Netherlands
E-mail: {b.li-et|k.l.m.bertels}@ewi.tudelft.nl

Abstract—**TTL is an abstract task-level interface which is used both for developing parallel application models and as a platform interface for implementing streaming applications on multi-processor architectures. Inter-task communication (ITC) is defined by TTL. The CAKE platform that we target, consists of homogeneous communicating tiles. Each tile consists of a shared memory with a heterogeneous mix of MIPS and TriMedia processors, DSPs and hardware accelerators. Due to task synchronization and data transfer in shared memory, architectural issues related with cache coherence and memory data copying are investigated. Optimizations such as padding-array insertion, prefetching and postflushing techniques are suggested. Alternative implemenations with semaphore and index/pointer as synchronization construct are explained. Prototype design and cycle true simulations with the Producer-Consumer model and the JPEG decoder application demonstrate that: compared to some old initial implementation, we can achieve almost 80% improvement on Cycles Per Token transfer (CPT), and reduce the total cycles of running the JPEG decoder application by 23%.**

Keywords—**inter-task communication; synchronization; shared-memory multiprocessor; semaphore; cache coherence**

I. Introduction

High Volume Electronics (HVE) consumer systems such as digital TVs and DVD recorders, process medias heavily. These media-processing applications are realized by hardware or software modules of the system. Layered software architecture [1] design is necessary for such system. They are classified into five groups: application, middleware, streaming, drivers and basic OS services. We focus on the streaming layer that directly provides streaming processing capabilities of the platform to middleware components. Streaming components can work together in a streaming application that can be modeled with a *task graph* (Kahn Process Network [2]). Streaming services of task communication, multi-tasking and connection management have to be offered to the streaming components. Task Transaction Level (TTL) interface was proposed [3] as an abstract interface providing streaming services. Correspondingly, it is standardized in three areas: streaming communication [4], multi-tasking and configuration and dynamic reconfiguration. TTL is not only an API to develop straming application models, but also a platform interface implementing the applications as communicating hardware/software tasks on a platform infrastructure. The implementation of TTL is platform specific. And in this paper, we focus on the implementation of inter-task communication (ITC) among software tasks only.

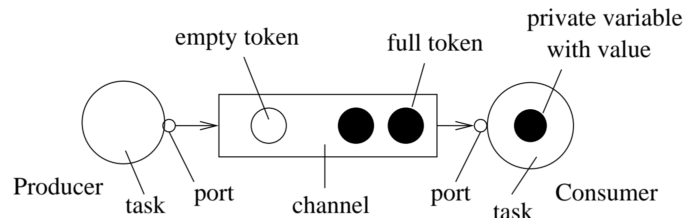

Fig. 1. Logical model of TTL inter-task communication

The definition of the TTL ITC can be logically modeled as shown in Fig. 1. *Tasks*, which execute concurrently, communicate with eath other through *channels*. *Producer* is the *task* that writes to the channel, while *consumer task* reads from the channel. Tasks connect to the channel through *ports*. A *token* is a variable holding a value that is communicated between tasks. A token is *full* or *empty* depending on if it holds value or not. We refer *full* and *empty* tokens as *data* and *room*. The *private variable* is a container for a value accessible by one task only. A communication operation basically includes task synchronization and data transfer. Tasks firstly synchronize to each other on the available *room/data* before transferring data values. Basic synchronization primitives are listed in Table I. Furthermore, *synchronization constructs* representing available room and data are necessary.

TABLE I
Basic synchronization primitives for TTL

Task	Primitive	Primitive role
Producer	acquire_room release_data	acquire empty token release full token
Consumer	acquire_data release_room	acquire full token release empty token

Our target platform, the CAKE (Computer Architecture for a Killer Experience) architecture, implements homogeneous multiprocessing with inter-connected tiles of heterogeneous computational units [5]. In this paper, we focus on the tile architecture which is of symmetric shared-

memory multi-processor architecture as illustrated in Fig. 2. It consists of general purpose CPUs (currently MIPS or TriMedia), coprocessors, a router, input/output processors, shared-memory banks and I/O device (e.g. proxy). Cache coherence protocol is write-invalidate snooping protocol [6]. Each CPU has it local cache which is regarded as L1 cache on chip. Shared-memory points to the L2 cache on chip. Running on each tile, Tile Run-Time environment (TRT) implements and provides the following services we need: threads and thread scheduling, semaphores for synchronization, mutexes and condition variables.

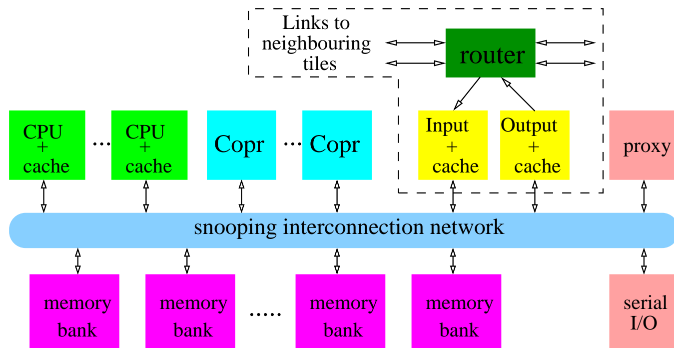

Fig. 2. CAKE tile architecture

Since communication is one of the major workloads of streaming applications and the cost of communication services is greatly determined by the implementation of the API, the aim of this paper is to seek efficient implementation of TTL inter-task communication on CAKE tile architecture in order to improve the performance of the streaming applications. We define the efficiency as execution cycles of the streaming application. For the communication operation, task synchronization and data transfer should be considered. Through our research, we found that synchronization effitiency decides the communication performance significantly in most cases. In this paper, we focus on the implementation solutions for task synchronization considering the platform obstacles. And we also look at alternative synchronization constructs. Prototyped TTL run-time environment (TTL RTE) is built based on TRT with application of these solutions. And tests prove the improvement in the application performance through our suggested methods.

In the following sections, the implementation options and optimizations are first explained in Section II. Section III introduces the benchmarks and simulation environment. Figures of performance measurements are provided and discussed in Section IV. Finally, in Section V, conclusions are drawn.

II. Implementation Details

A. Platform view

Tasks and channels are the principle elements of the ITC. Our implementation of the ITC is based on shared-memory due to the CAKE tile architecture.

- Task: To be more specific, as TRT supports threads and dynamic thread scheduler, tasks are implemented as threads running on processors. TRT looks for tasks to run on available processors concurrently. If there is only one processor configured on the platform, multiple tasks are scheduled on this processor but only one task runs at a time. Processors are sharable for all the tasks.
- Channel: As the place where the communication takes place, channels are implemented in the shared memory. It consists of *channel buffer* and *channel administration* values.
  - Channel buffer: The buffer is a part of memory area where the transferred data is stored. It is an ordered FIFO in our case as decided in [4], and uses contiguous memory.
  - Channel administration: The administration is the place where the status of the channel buffer is stored. Tasks initiate communication and coordinate with each other according to the channel administration values. The channel administration consists of static values and dynamic values. Static values, including *buffer size* and *base addresses of the buffer*, are set during the system setup. Dynamic values, including *synchronization constructs* and *reading/writing positions*, are modified by tasks during run-time. The administration values are centralized allocated in the shared memory for the sake of cheapness and fast access on the CAKE tile platform.

From above explanations, the ITC is indeed transformed into inter-processor communication and operations on the shared memory as illustrated in Fig. 3. In oval shapes, *P* represents a producer task and *C* is a consumer task. They separatly run on *CPU1* and *CPU2* shown in rectangular shapes. The channel through with they are communicating is in the shared memory. The producer writes data to the channel buffer while the consumer reads from it. Both tasks access and update the channel administration values.

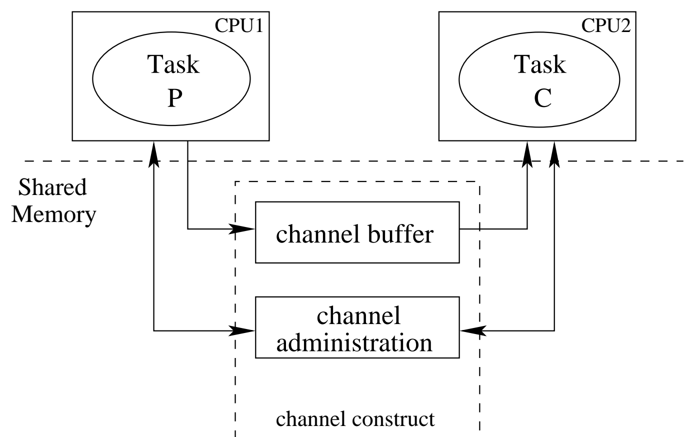

Fig. 3. The platform implementation view of the ITC

The synchronization entails accessing and updating the channel administration values, which always involves processors concurrently loading from or storing to the same area of the shared memory. Therefore, the behavior between the processors and the shared memory determines the performance of the ITC, especially when tasks run on multiple processors.

B. Architectural issues and solutions

The architectural problems rise from concurrent access and update of the shared channel administration values. Reads and writes to the same memory location are unavoidable and must be kept consistent. On the CAKE tile platform, the write-invalidate snooping protocol for cache coherence defines memory-cache behavior. A processor has exclusive access to a data item before it writes that item. All other cached cache blocks holding the item in other processors are invalidated. Each cache block is currently set to 128-bytes, which may hold multiple data items of smaller sizes. When another processor tries to read a data item from the invalidated cache block, a load miss occurs, and the value will be returned when the first processor finishes writing to its cache. Invalidation increases cache misses, which is regarded as *coherence misses* and occurs in interprocessor communication. The coherence miss is the architectural synchronization overhead of our inter-task communication. The fewer misses there are, the better performance of the task synchronization we can achieve.

There are two sources of coherence misses: *false sharing misses* and *true sharing misses*. The former misses occur when a processor attempts to read a data item not being written by another processor on the invalidated cache block; the later misses occur when a processor attempts to read the same data item being written by another processor. Software solutions in TTL implementation are investigated to reduce these misses respectively.

B.1 Reducing false sharing misses

• Padding arrays

The origin of false sharing misses is that multiple data items are allocated in the same cache block and they are read or written by different processors. The intuition is that this can be solved by allocating them in separate cache blocks. Inserting empty spaces by defining empty arrays between the channel administration values should be feasible. These arrays are regarded as *padding arrays* which intend to pad spaces. The size of the padding arrays should be equal to the size of cache block 128-bytes. The number of array elements varies depending on different data types used.

The main disadvantage of using the padding arrays is the waste of memory. However, the more padding arrays are used, the more false sharing misses can be reduced. This is a trade-off between performance and memory cost. There is one ultimate solution for the best performance that: all the values are separated by a padding array. The number of the padding array increases with the number of the channel values, which is sure to waste a lot of memory.

• Cache allocation mechanism

Not all the values need to be in a separate cache block, and whether two values can stay in one cache block is determined by their processor read/write patterns. The cache allocation mechanism we conclude explains how to group the values according to their processor access patterns.

We group the values into three catogories:

1. *Read-only values:* These values are only read by processors. They are initialized during system setup. The static channel administration values belong to this group. These values can be allocated in one cache block.

2. *One-processor read/write values:* These values are only read and written by one processor. In other words, these values are only accessed or updated either by the producer task or by the consumer task. For example, the head position of the data in the channel buffer is only needed by the consumer task; the tail position of the data in the channel buffer is only needed by the producer task. The values are only updated by the processor that the producer or the consumer task runs on. The *one-processor read/write* values can be allocated in different cache blocks according to their processors or tasks ownerships.

3. *Multi-processor read/write values:* These values are read and written by multiple processors. True sharing misses should occur on the access to these values. The synchronization constructs in the dynamic channel administration values belong to this group because they should be updated by both the producer and the consumer tasks. Allocating each of such values in a separate cache block can surely achieve the best reduction on the misses. If the number of such values is small(e.g.2, 3 or 4), we recommend to allocate each of the values in a separate cache block. When the number grows big, we recommend to combine the values with the *one-processor read/write* value groups. The update frequency of each processor should be taken into account when deciding the combination. A value should be better allocated at the side that more frequently updates the value. The values with equal update frequencies of processors are to be allocated in separate cache blocks.

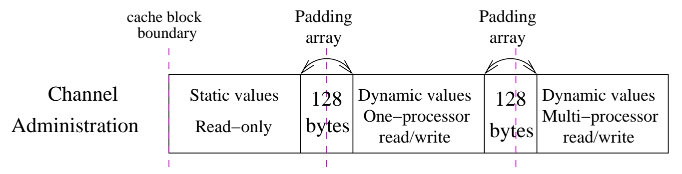

Fig. 4. Cache allocation with padding arrays

An example of applying padding arrays is conceptually given in Fig. 4 on the cache allocation of the static and dynamic channel administration values. Three major groups of values are separated by spaces of padding arrays.

B.2 Reducing true sharing misses

True sharing misses occur when concurrent processor operations applied on the same value at the same time. From platform perspective, the chances mostly depend on the speeds of the running processors and the workload of tasks on the processors. These factors are hardly controlled through the communication implementation. Our objective is to reduce the chances of such concurrent operations in the software implementation of the ITC. In the task communication primitive, the operations on the synchronization construct are unavoidable. The more often such operations are needed in each task, the more probable they occur at

the same time. Therefore, the effective way to reduce this probability is to decrease operations on the synchronization construct.

• Local storage: Instead of operating on the synchronization construct every time the communication primitives are called, it is suggested to locally store the calculated amount of available room (for producer task) or data (for consumer task) on the channel. These values are only privately read and written by the task. The amounts are probably more than what the tasks need, but no more than actual available on the channel at that time. When the local stored data show that the channel is empty or full, the task can not proceed and operations on the original synchronization construct occur to get the newest values. The advantage of this mechanism is that: local storage is to be cached locally in the processor on which the task runs and will not suffer from sharing problems; other tasks may update the shared synchronization construct, when a task runs with its local value. This method can be also called *pre-fetching*.

• Local accumulation: We also suggest to locally accumulate the amount of room (for consumer task) or data (for producer task) to be released in accumulating counters. When the counter reaches a threshold, a number of room/data is released to the channel at once. Therefore, the number of synchronization operations in releases are pending and reduced. This method can be called *post-flushing*. *Post-flushing* may cause *deadlock* when the producer task exits but its local accumulating counter has not reached the threshold. In this situation, the consumer task does not know there is data on the channel and will be blocked until deadlock. For this problem, block handling and exit handling are suggest to flush pending room/data when tasks block and exit. Due to the implementation complexity, we focus the implementation with *pre-fetching* in this paper, and will discuss the one with *post-flushing* and deadlock handling in future papers.

In fact, the methods we introduced above on reducing sharing misses are general that can be optionally applied on other appropriate implementation cases on symmetric shared-memory multiprocessor platforms.

C. Synchronization construct alternatives

For the constructs that represent room/data status of channels, we investigate two possible alternatives. Semaphore is chosen because it's directly support by CAKE TRT for thread synchronization; Index/offset is chosen because it might benefit on future hardware/software ITC implementation extension.

C.1 Semaphores based

Semaphores are non-negative counters. The P and V operations on semaphores operate counter values and block the calling tasks. Available *room* and *data* are represented in semaphores, which match the TTL communication semantics very well. The synchronization is done throung semaphore operations. With application of padding array, cache allocation mechanism and local storage, we define the channel administration values as the following: (T represents the data type of the token, which can be basic (e.g. *char*, *integer*) or complex (e.g. *struct*) data types)

```
integer padding_array1[32]  # 128-bytes

#----static values, read-only values ----#
size # size of channel buffer
*first # starting address of the buffer memory
*last  # end address of the buffer memory
integer padding_array2[32]

#---- consumer task updated only ----#
T* r  # current reading position
# local storage of prefetched amount of data
integer local_data
integer padding_array3[32];

#---- producer task updated only ----#
T* w # current writing position
# local storage of prefetched amount of room
integer local_room
integer padding_array4[32];

#---- multi-tasks read/write ----#
semaphore room # amount of available room
integer padding_array5[32];
semaphore data # amount of available data
integer padding_array6[32];
```

The padding arrays at the beginning and the end of the channel administration are applied to separate the channel values from other values in the program. How the channel administration values are related with the channel buffer is illustrated in Fig. 5. The basic principles of *pre-fetching* have

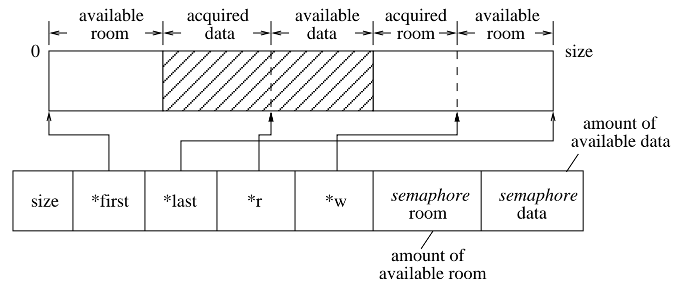

Fig. 5. The basic channel implementation: semaphore-based

been explained. Here to exemplify, the algorithms for producer and consumer tasks in semaphore-based implementation with *pre-fetching* are shown below in pseudo code, refer to Algorithm 1 and Algorithm 2. In the algorithms, `sem_Pn` decrements `room/data` to 0 and return the decremented amount.

In this paper, we aim to show proof of our suggested method for improving the application performance. Therefore, we mainly explain the semaphore-based implementation in detail and briefly describe the difference in the chan-

Algorithm 1 Semaphore: Producer acquire and release

```
1: procedure ACQUIRE_ROOM( )
2:   if local_room == 0 then
3:     local_room ← sem_Pn(room, MAX_INT)
4:   end if
5:   local_room − −
6: end procedure
1: procedure RELEASE_DATA( )
2:   sem_V(data)        ▷ semaphore V operation on data
3: end procedure
```

Algorithm 2 Semaphore: Consumer acquire and release

```
1: procedure ACQUIRE_DATA( )
2:   if local_data == 0 then
3:     local_data ← sem_Pn(data, MAX_INT)
4:   end if
5:   local_data − −
6: end procedure
1: procedure RELEASE_ROOM( )
2:   sem_V(room)        ▷ semaphore V operation on room
3: end procedure
```

nel construction of the index/offset-based implementation. The reason is that the suggested optimization methods are applied in the same way as the semaphore-based implementation. Moreover, in Section IV, we will give experiment results of semaphore-based implementation only since it is sufficient to show the improvement. Details and results of index/offset-based implementation can be received from [7].

C.2 Indexes or offsets based

The index or offset points to the values representing positions on the channel buffer. The channel status can be checked by reading these values. At least four elements are needed to distinguish the four areas of acquired room/data and available room/data, which are conceptually shown in Fig. 6. And at least one index is necessary to keep track of

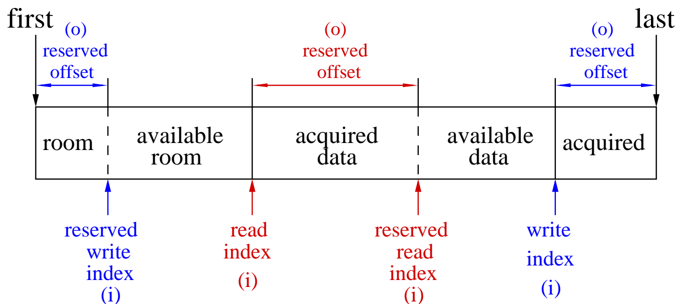

Fig. 6. Conceptual illustration of index and offset meanings, symmetric examples

reading and writing positions. The other elements can be either index or offset. By index, it means the position; by offset, it means the variable that counts the number of acquired room/data towards basic indexes. Therefore, there are four possible combinations of index and offset:

```
1i+3o      #--1 index   + 3 offsets
2i+2o      #--2 indices + 2 offsets
3i+1o      #--3 indices + 2 offsets
4i         #--4 indices
```

We choose symmetric ones which keep the producer and consumer views identical on the channel. Hence, the combination of `2i+2o` and `4i` are chosen. It is not enough with only indexed and offsets to distinguish the emptyness or fullness of the channel. There are two options for this problem: one is using wrap flags with indexes indicating the wrapping around of the index across the end of channel buffer; the other is wasting one token buffer assuming that the channel is full when the distance between the read and write indexes is 1. Therefore, options lie in: `2i+2o+2wf`, `4i+4wf`, `2i+2o` and `4i`. The content of each option is given in Table IIIn Fig. 7, it exemplifies the channel structure and administration values meanings for `2i+2o+2wf`. For synchronization in the index/offset scheme, task suspend method supported by CAKE TRT is applied. As mentioned previously, priniciple channel construction of the index/offset-based implementation is provided in this paper. Detailed explanation

TABLE II
OPTIONAL INDEX/OFFSET-BASED STRUCTURES

	Option	Contents
1	2i+2o+2wf	basic reading/writing indexes: *read and write* offset to *read/write* wrap flags for *read/write*
2	4i+4wf	basic reading/writing indexes: *read and write* indexes to where room/data can be acquired wrap flags for all the indexes
3	2i+2o	basic reading/writing indexes: *read and write* offset to *read/write*
4	4i	basic reading/writing indexes: *read and write* indexes to where room/data can be acquired

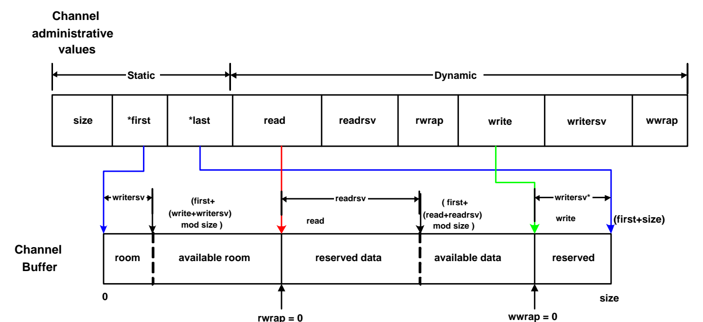

Fig. 7. Channel administration illustration with `2i+2o+2wf`

is not presented here because we intend to show the performance improvement made by our solution in this paper.

III. Experiment Setup

A. Interface choice

The TTL interface totally defines seven communication interface types, trying to consider different needs of streaming application modelinng and various potential platform implementations. All TTL communication interfaces are based on the above explained logical model. And the classification is based on whether the synchronization and data transfer are combined or separated, whether the task blocks or doesn't block when the synchronization condition is not met, etc. [4]. In this paper, we choose TTL CB (combined synchronization and data transfer, task block) interface because it is the easiest and most-commonly used interface type. Abserving the performance of CB interface is very meaningful for application designers.

B. Benchmark applications

• Producer-Consumer application: **This is the simplest application model with only one producer and one consumer (P-C) and we don't include any computation in this model. In our experiments, the token data type is fixed to *integer*. The channel size is fixed to 2048 tokens (i.e. if a token is of integer, channel size is 8-Kbytes). To show results of synchronization performance, we only use scalar operations of tranferring a small token every time. We set the number to 100000 integers in total per inter-task communication.**

• JPEG decoder application: **Besides simple P-C model, we want to test the performance with more complicated application examples. Modeled with flat task graph and utilized the CB interface, the JPEG decoder application was chozen. Totally 48 tasks are defined in its network.**

C. Benchmark

• Cycles Per Token transfer (CPT): **To evaluate the performance of an inter-task communication implementation, the time of communicating one token is wanted to show the speed of the communication. It includes the time on writing and reading the token, and can be calculated as the following:**

$$CPT = T(write) + T(read) = (T(N) - T(0))/N$$

We assume that the time of the application initialization is independent of the number of the tokens communicated. $T(N)$ represents the total number of cycles including the application initialization and communication of *N* tokens. $T(0)$ represents the total number of cycles including the application initialization and communication of zero tokens. The result of the subtraction of these two values is the total communication time for transferring *N* tokens. Therefore, *CPT* is the algorithmic average of the total communication time by *N*. The CPT of running on one processor indicates the actual cycles for the instructions of read and write involved in communicating a token. The CPT of running on more than two processors also includes influences of cache coherence. We apply CPT with the test of P-C application.

• Total number of cycles: **The total execution cycle time of an application includes: the time on initialization, the time spent on communications and the time spent on computations. We apply this benchmark with the test of the JPEG decoder application to overview the application performance.**

D. Simulation environment

The CAKE simulator, *CakeSim*, is used in the experiment. It is a cycle-true simulator based on TSS[1] model. Different number of MIPS and TriMedia processors can be requested by configuration. In this paper, we only provide test results with MIPS processors because results TriMedia repeat similar performance outcomes. And for P-C application, maximally 2 processors are necessary since only two tasks execute. Details of *CakeSim* can be found in [8]. The configuration of *CakeSime* in our experiment is as the following:

```
MIPS model:                  PR4450
Cache set associative:       12
Shared memory size:          12-Mbytes
Number of memory banks:      8
Bus burst size:              64-bytes
```

IV. Results

We first provide results explaining the performance improvement achieved by the suggested padding array and local storage mechanism. The results are from the test with P-C application. Then we compare our ITC implementation performance with a predecessor interface implementation called YAPI (one of the predecessors of TTL interface, has same communication primitives as TTL CB interface). In our results, `CB-NOOPT` stands for the implementation of CB interface without local storage optimization; `CB-PREFETCH` stands for the scheme with local storage. `Padding` or `no padding` stands for if padding arrays are applied or not.

A. Improvement achieved by padding arrays

As illustrated in Fig. 8, applying the padding arrays does not affect the performance of executions on 1 processor, but it improves the performance on 2 processors greatly. In case of 2 processors, the improvement percentages are 50.2% for CB-NOOPT and 88.9% for CB-PREFETCH. That's indeed a lot.

B. Improvement achieved by local storage

On the base of applying padding arrays, we can see that CB-PREFETCH is much better than CB-NOOPT especially in the case of running on 2 processors. The improvement percentages are respectively 7.3% on 1 processor and 74.5% on 2 processors. We here present the results in Fig. 9 again to compare more distinguishingly.

We also compare CB-PREFETCH with YAPI running P-C application. The comparison can be found in Table III.

[1]TSS is a cycle accurate C language based simulation framework used within Philips

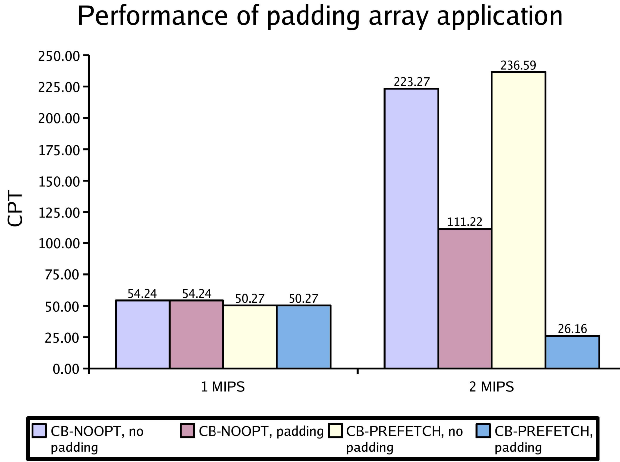

Fig. 8. Results indicate padding-array's improvement

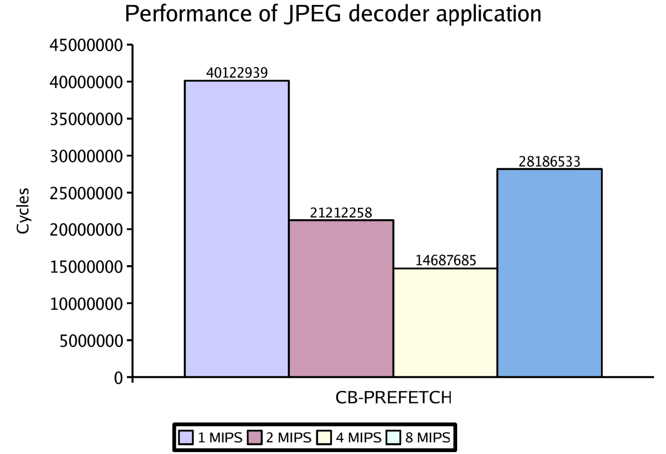

Fig. 10. Performance measurements of JPEG application

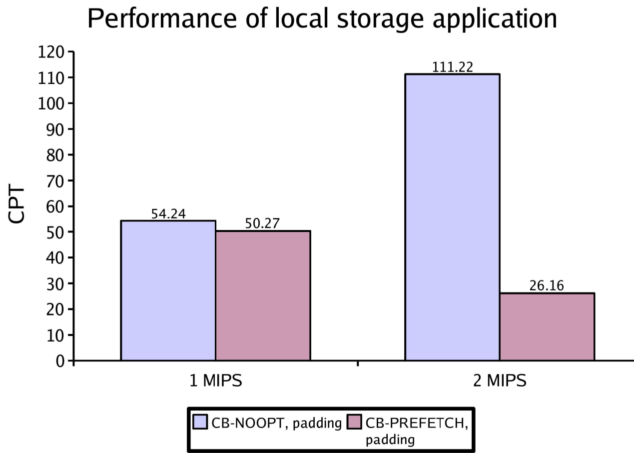

Fig. 9. Results indicate local storage improvement

We can find the improvement percentage is 26.4% on 1 processor and 32.3% on 2 processors.

TABLE III
COMPARE TO YAPI, P-C APPLICATION, CPT

Scheme	1 MIPS	2 MIPS
YAPI	68.29	36.24
TTLCB-PREFETCH	50.27	26.16

C. *Performance of JPEG application*

Previously only measurements with P-C applications are presented. With padding arrays applied, we also test our best implementation solution CB-PREFETCH with JPEG decoder application. The results in total execution cycles are given in Fig. 10.

From the results we can see that: with the increasing number of processors, the total number of cycles decreases. But it is nonlinear. When the number of processors is 8, the number of cycles goes back to increase.

We compare the JPEG application results with results of running YAPI. The comparison is given in Table IV. We find that our implementation TTLCB-PREFETCH improves the performance of the application by 18.7% on 1 MIPS, 19.3% on 2 MIPS and 22.7% on 4 MIPS.

TABLE IV
COMPARE TO YAPI, JPEG APPLICATION, TOTAL CYCLES

Scheme	1 MIPS	2 MIPS	4 MIPS
YAPI	49351167	26288530	19009813
TTLCB-PREFETCH	40122939	21212258	14687685

From the above given results, our TTL ITC implementation achieve comparable and better performance. We proved that our suggested methods of padding arrays, cache allocation mechanism and local storage are efficient and reduce communication cost.

V. CONCLUSION

In this paper, we analized the architectural factors affecting communication efficiency and discussed possible solutions to overcome the architectural obstacle. We focus on the synchronization perspective. It is found that the true and false cache coherence misses are mainly the issues that affect synchronization performance. We suggested applying padding arrays and cache mechanism to reduce false sharing misses; and advised applying local storage and local accumulation of channel administration values to reduce true sharing misses. Two kinds of synchronization constructs were discussed: semaphore and index/offset. Results which aim to prove the performance improvement achieved by our suggest solutions are presented and compared. On multiprocessors, the padding array with cache allocation mechanism can improve the performance by 89%, and the local storage can improve by 74%. And compared to YAPI a predecessor implementation, JPEG application performance can be improved by 23%. We successfully provide software solutions for reducing synchronization overhead and data trans-

fer cost to achieve total communication time improvement, especially on multiple processors.

References

[1] Jeffrey Kang, Gerben Essink, Pieter van der Wolf and Tomas Henriksson, *Position of TTL in a System Architecture*, Technical Report PR-TN-2003/00703, Nat.Lab. Philips Research, 2003

[2] Gilles Kahn, *The Semantics of a Simple Language for Parallel Programming*, Proc. of IFIP Congress 74, North-Holland Publishing Co., 1974

[3] Pieter van der Wolf, Erwin de Kock, Tomas Henriksson, Wido Kruijtzer and Gerben Essink, *Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach*, Proceedings of CODES+ISSS'04 (Stockholm, Sweden), September 8-10 2004

[4] Gerben Essink, Andrei Rădulescu, Pieter van der Wolf and Jeffrey Kang, *Task Transaction Level Interface, Inter-task Communication*, Technical Report Version 0.1, Nat.Lab. PHilips Research, 2004

[5] Paul Stravers and Jan Hoogerbrugge, *Homogeneous Multiprocessing and the Future of Silicon Design Paradigms*, Proceedings of the International Synposium on VLSI Technology, Systems and Applications (VLSI-TSA 2001), April 2001

[6] John L. Hennessy and David A. Patterson, *Computer Architecture, A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003

[7] Bei Li, Pieter van der Wolf and Koen Bertels, *TTL inter-task communication implementation on a shared-memory multiprocessor platform*, Master Thesis, August 2004.

[8] Paul Stravers and Jan Hoogerbrugge, *the spaceCAKE simulation framework*, 2003.