Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Modelling a Multi-Core Media Processor Using JCSP

Anna KOSEK [a], Jon KERRIDGE [a] and Aly SYED [b]

[a] *School of Computing, Napier University, Edinburgh, EH10 5DT, UK*
[b] *NXP Semiconductors Research, Eindhoven, The Netherlands*

Abstract. Manufacturers are creating multi-core processors to solve specialized problems. This kind of processor can process tasks faster by running them in parallel. This paper explores the usability of the Communicating Sequential Processes model to create a simulation of a multi-core processor aimed at media processing in hand-held mobile devices. Every core in such systems can have different capabilities and can generate different amounts of heat depending on the task being performed. Heat generated reduces the performance of the core. We have used mobile processes in JCSP to implement the allocation of tasks to cores based upon the work the core has done previously.

Keywords. JCSP, multi-core processor, simulation, task allocation.

Introduction

Many manufacturers of semiconductor computer processors are designing multi-core systems these days [1]. In multi-core processing systems, allocation of work to processors can be seen as similar to the task of allocating work to people in a human society. A person responsible for controlling this process has to know the abilities of their employees and estimate the time in which a task can be finished. Tasks can often be finished faster if more workers are assigned to work on them. Generally, tasks can also be finished faster if they can be divided into smaller sub-tasks and sub-tasks can be processed concurrently. One very important condition that has to be met is that these sub-tasks have to be allocated wisely so that co-workers working on different sub-tasks can not hinder each other's progress. The manager has to allocate the task to the worker that is the best for the assignment in current circumstances. Using this idea many contemporary scientists and engineers are building multi-core processing systems.

Multi-core processor technology is one of the fastest developing hardware domains [2]. Modern personal computers already have multiple computing cores to increase a computer's performance. Multi-core systems for consumer electronics however have different challenges than those in personal computers.

Targeted media processors have been a goal of research of many scientists. In paper [3] the authors are presenting a heterogeneous multiprocessor architecture designed for media processing. The multi-core architecture presented in [4] consist of three programmable cores specialized for frequently occurring media processing operations of higher complexity. Cores are fully programmable so they can be adapted to new algorithm developments in this field [4]. More advanced research was shown in [5] presenting heterogeneous multi-core processor capable of self reconfiguring to fit new requirements.

1. Heterogeneous Multi-core Systems in Consumer Electronics

In a heterogeneous multi-core system for a consumer electronics device, it is often not possible to predict what functions the user would like to perform at what time and what data the user needs to process. A system manager has to deal with incoming requests, divide them into tasks and allocate them to one or more cores. In a heterogeneous multi-core system, every core has different capabilities and also generates different amounts of heat depending on the task being performed. The heat generated not only reduces the performance of the core and if unchecked could result in a thermal runaway leading to incorrect behaviour. Heat generation also makes a hand-held device less comfortable to use. Power consumption reduction, associated with heat generation, is very important issues when considering handheld mobile devices, only the latter is considered in this project. Heat reduction can be mitigated by an appropriate allocation of tasks to cores. In paper [6] authors show that a choice of appropriate core and switching between cores in heterogeneous multi-core architecture can optimize functions of performance and energy.

Our premise is that the amount of processing required depends on the task content, which is not always known during allocation, Therefore the system management should dynamically allocate tasks to cores based upon their previous use. If a task can not be finished in a given time, allocate it to a different core. Envisaging how such a system can work with a real application is very difficult.

A hardware simulation is usually used to shorten the hardware design and development process [7]. Hardware simulations are undertaken to prove some assumptions or new ideas about hardware architecture without actually creating the hardware itself. Computer systems used in environments where software execution should meet timing limitations are defined as real-time systems [7]. The real-time behaviour of devices is very important for their correct functioning, especially in systems designed to render various forms of media such as audio and video.

The aim of simulation presented in this paper is to show that an appropriate allocation can be done and if, for some reason, tasks can not be accomplished a different fallback plan can be adopted. The simulation uses the capabilities of the Communicating Sequential Processes (CSP) model [8] to create parallel systems. CSP is a formal model describing concurrent systems that consist of processes working simultaneously and communicating with each other [8]. The CSP model enables real-time aspects by scheduling and priority handling [9]. The concept of a system in CSP comprises a set of processes running concurrently connected through channels to exchange data and control. It has been shown that if the principles of CSP model are preserved, a correct implementation of a system can be built [9]. We have used JCSP (Java-CSP) that is an implementation of CSP and provides libraries allowing the development of Java applications.

We have built a simulation framework to investigate processing and scheduling requirements on heterogeneous multi-core processors that will be used in future devices. We can simulate the performance of a chosen architecture running a real application.

2. Function of the Simulated System

The operation of the system is captured in Figure 1. The diagram in Figure 1 was drafted as part of a project specification [10] with NXP Semiconductors. The system receives a data stream comprising audio and video compressed broadcast data. A selected channel is decoded (*Channel decode*) to extract the audio-video multiplex of streams. The audio and video components are then de-multiplexed (*Demux*) and sent on to the audio and video decoders, respectively.

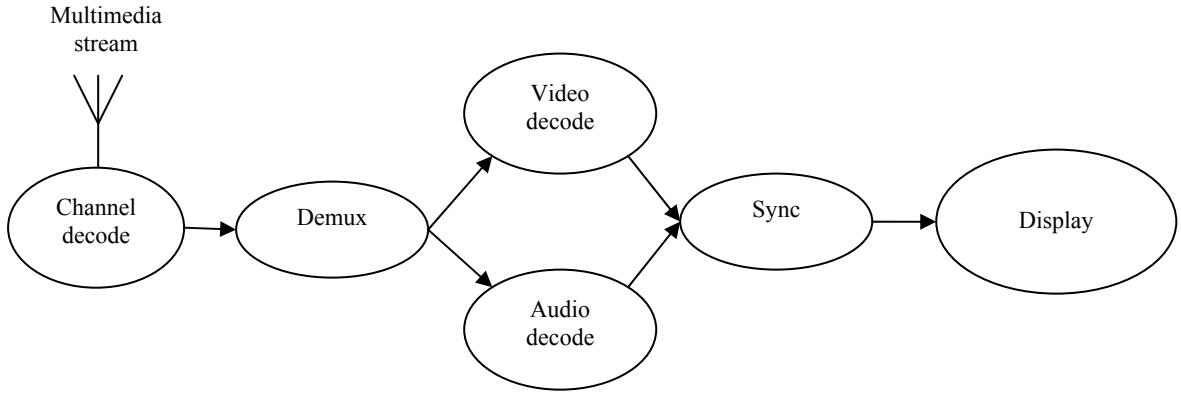

Figure1: de-multiplexing, decoding and re-synchronising broadcast multimedia data.

These decoders send their result streams to be re-synchronized for correct presentation. After this step, the display of the audio-video stream can take place, usually employing some form of rendering process. The modelled system assumes that cores of different capabilities are available.

The *Demux* process has the task of deciding which core is most appropriate to deal with the incoming task based upon: task requirements, core capability and the current state of the chosen core. The state of the core depends upon the amount of heat yet to be dissipated. The greater the residual heat, the slower the core will operate.

3. Subdividing Audio and Video Streams

As described in the section above, the system receives blocks of audio-video stream which are demultiplexed. We assign the demultiplexer also the function of dividing these streams into tasks and subtasks. These subtasks are then allocated to the different processing cores in the system. Figure 2 shows that multimedia data stream consists of blocks of data containing audio and video data. The demultiplexer then divides this block of data into different number of tasks depending on the amount of data in a block which in turn depends on the audio/video content. Blocks and tasks are numbered beginning from 0. Each task is then also divided into separate subtasks for audio and video. These subtasks can now be assigned to different cores.

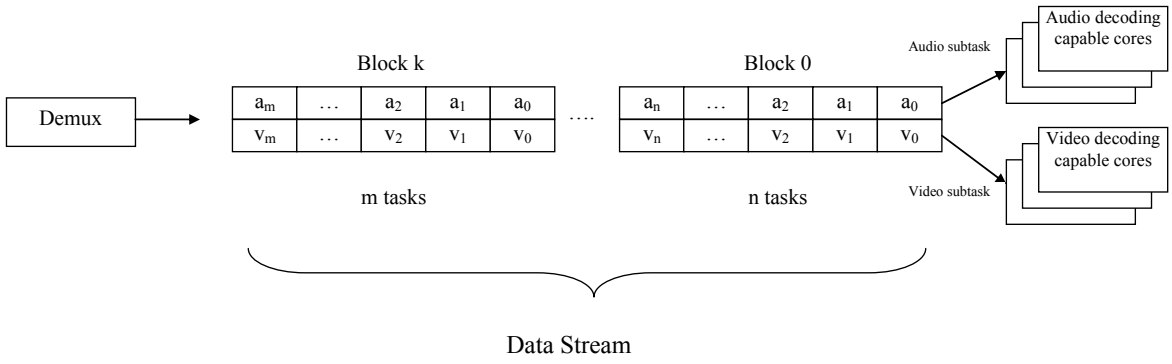

Figure 2: multimedia data stream.

In a multiplexed multimedia stream, it is unlikely that the audio-video subtasks will line up as regularly as indicated in Figure 2. An important aim of this project was to investigate the feasibility of allocating units of work to different cores. For the purpose of the designed architecture audio and video blocks always appear together.

A subtask contains the requested task type, the amount of work that has to be done and the data necessary for synchronization. Blocks are sent in sequence and the order of task synchronizing and displaying is vital. The order for synchronization is important simply because the video and audio tasks processed by the system have to match. Corresponding audio and video tasks can take a different amount of time to be processed, but have to be displayed at the same time.

4. Architecture of the Simulated System

Figure 3 shows the process architecture that has been implemented to test the feasibility of modelling a multi-core processor using JCSP and mobile processes. The system is designed to receive an audio-video data stream in a given structure (Fig. 2) consisting of blocks of data divided into audio and video subtasks. The system allocates subtasks to cores. The implemented system can run concurrently using any number of cores with different capabilities. In the present implementation, the number of cores is chosen to be nine; however, the same architecture can be applied to an arbitrary number of cores. The capabilities of these cores are defined in Table 2. The spatial arrangement of cores on Figure 3 is not relevant.

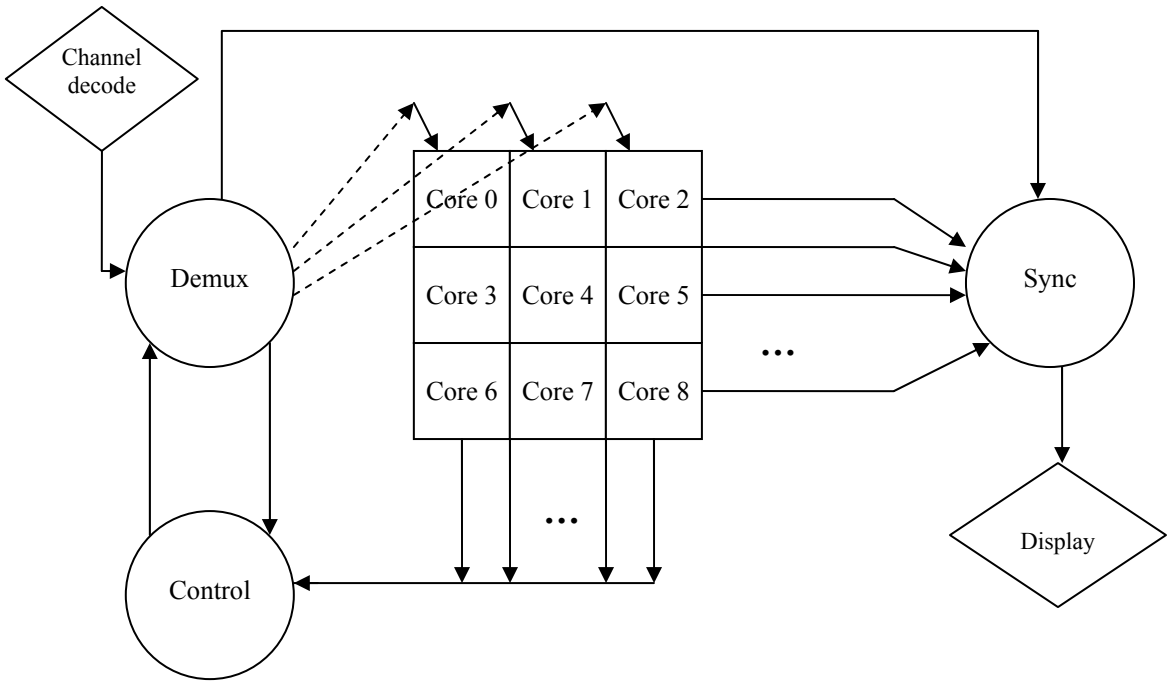

Figure 3: architecture diagram.

The system consists of processes running in parallel and connected with network channels (Fig. 3). This architecture better mimics the final design goal in that each process in the system is executed in its own JVM and hence all processes are running in parallel using a communication network implemented using network channels. This meant that we could more easily model network latency in future experiments. Every network channel is

assigned a name which is unique and recognized by a CNS Server process, from the JCSP package [11], running in the background.

In addition to the basic concepts of CSP as discussed in section 1, we base the architecture in particular on the concept of mobile processes in CSP to achieve correct scheduling of subtasks on the different cores in the system. We consider every subtask as defined in section 3 to be a mobile process that can be sent to any core in the system for execution. A scheduling of such a system is considered correct if it makes optimal use of the many cores in the system with differing capabilities.

The Channel Decode process is responsible for receiving the audio-video stream, forming it into separate blocks of audio and video data representing subtasks and sending them to the Demux process which is also responsible for scheduling them to run on different cores in the system. The Demux has a variety of cores with differing capabilities available to choose from. The decision of where to send subtask data depends on capabilities of the cores and their availability. In order to make this decision, the Demux process sends a message to the Control process with a request to identify a core where a subtask should be allocated.

The capabilities of a core are dynamic and can change when a core heats up. Therefore every time a core changes its capabilities it informs the Control process which in turn updates its knowledge about the capabilities of each core. Thus when a Control process receives a message from the Demux process that also identifies the type of subtask, the Control process makes a decision about which core to assign the data for processing based on the capabilities and availability of the cores in the system at that moment in time. The Control process then sends a message to the Demux process identifying the suitable core.

When the Demux process receives the identity of the suitable core from the Control process, it creates a mobile process containing the data structure, shown in Table 1 and sends the mobile process to the designated core. To do this, a network channel is created dynamically using the core's location and CNS Server. The connection is established only once and when the core is selected for another subtask, the channel will be used again. Only the necessary connections between Demux and core processes are created

The Demux also sends a message to the Sync process informing it of the order in which it should receive data from cores in order to re-synchronise the video and audio subtasks contained within a block.

All of the cores in the system are connected to the Sync process. This process is responsible for synchronizing data in the right order, first audio then video subtasks from blocks. The subtasks are processed on cores in parallel so the Sync process might not get them in the right order. Therefore the Sync process is informed by the Demux about a sequence of cores used for processing data. The Sync waits for audio and video parts of one task and waits for a message only from those two cores that were processing corresponding subtasks. When the connection takes place the merged task is sent to the Display process.

In the simulation, a core processor is responsible for the required processing of a subtask but the time taken, is dependent upon the core's specific capabilities and also its present heat content which can lower its processing capabilities. The core then determines the processing required for the subtask and using its own look-up table determines the factor that will be applied to a subtask to determine the actual time taken to process a subtask's data. The core then waits for that number of time units (e.g. milliseconds) before sending the updated mobile process to the Sync process.

To make the multi-core processor simulation more realistic the core that executes a subtask gets heated. Heat gained changes a core's capability, increasing the time to process a task. If a core doesn't process any data it loses heat until it reaches its initial state. Decreasing temperature also affects a core's capability and makes it work more quickly.

The system uses the properties of CSP channels and avoids including a process that will be a buffer holding subtasks that have been processed, therefore finished subtasks wait on cores for the Sync process to be ready to read them.

4.1 Data Structure and Mobility

As described in section 3, the incoming stream of data is divided into blocks, tasks and audio and video subtasks. A data structure described in this section is designed to describe these entities and to track their progress through the system.

The data structure used by a mobile process is shown in Table 1; all values are integer type. The input stream is a sequence of blocks (**b**). Each block is divided into tasks (**t**) each of which comprises a subtask that can be processed by a core. Subtasks are separated into audio and video categories (**c**). Each subtask is given a required processing type (**r**) representing the nature of the processing required. The amount of work required to process the subtask is specified as the time (**w**) taken to process on an ideal core. An ideal core is one that can undertake the requested subtask in the minimum possible time and that has not been previously used and is thus at minimum possible heat content. The output variable (**o**) is the actual time to process the subtask on the core to which it was allocated. The data structure is designed to describe the subtask and to keep information about it and update it in the system.

Table 1: structure of data carried in the system.

Name	Description	Special values
b	Block number	Starts from 0
t	Task number	Starts from 0
c	Subtask category	Audio = 0 Video = 1
r	Requested subtask type	Starts from 0
w	Amount of work of requested subtask type (units of work)	Starts from 1
o	Variable reserved for actual core performance (outcome) time needed to execute requested subtask	By default equals 0, but changes after subtask processing

The value of **o** is determined as follows:

$$o = F[r] \cdot w$$

Each core process has a table **F** such that each element of **F** contains a factor (greater than 1) which is used to determine the time it takes to process a subtask of a particular requested subtask type **r**. A core's capabilities change dynamically so entries in the **F** table will not always be the same for the same core. Therefore the output variable (**o**) holds the actual time that the core needed to process a subtask and it is updated after a subtask's execution.

Heat management is essential to avoid thermal runaways and can be done by proper dynamic scheduling of tasks so that cores do not heat up beyond a certain point. The mechanism helps overheated cores to cool down by allocating tasks to other cores. However, this has to be done in such a manner that real-time constraints are maintained. This aspect was not considered in the work represented in this paper. If a task is sent to a

core that is not ideal, because the ideal core is already hot, then the time to process the task may be longer and hence the chosen core may heat up more than would have occurred if the ideal core had been available. In a real system heat can be transferred between cores due to their adjacency. This aspect was not considered in this project.

4.2 Mobile Processes

A mobile process can visit a processing node and perform operations using processing nodes resources [12]. The mobile process interface defines two methods *connect* and *disconnect* that allow the mobile process to connect to a node's channels and disconnect when it has completed its task. Mobile processes exist in one of two states: active and passive. In the active state it can carry out some instructions and write and read from the node it is connected to. In the passive state a mobile process can be moved or activated. When a mobile process becomes passive it can be suspended or terminated. After suspension a mobile process saves its state and when it is reactivated starts from the same state. When a mobile process is terminated it may not be reactivated. When a mobile process is activated by some other process it starts working in parallel with it, channels between those two processes can be created to achieve communication.

The data processed by the simulation is carried by the mobile processes. The system uses mobility of processes to track information about every subtask. One mobile process is responsible for carrying only one subtask and storing information about it. The mobile processes are created in the Demux process and initialised with values representing a subtask (Table 1). The mobile process is disconnected from the Demux process and sent to the appropriate core through a network channel. The mobile process goes into a passive state and after it arrives at the core it has to be reactivated. The mobile process can be connected and communicates with the core process exchanging data and capturing information necessary to evaluate the core performance. The mobile process is next sent to the Sync process and connected to it to retrieve data about the task and the time in which it was processed on the core.

One objective of the project was to determine the feasibility of using the mobile process concept as a means of distributing work in such a heterogeneous multi-core environment. The idea being that a complete package comprising process and data might be easier to move around the system, especially if there is a threat that the real-time constraints may be broken and the task has to be moved to a faster core that has now become available.

Tracking information about a subtask can be performed using mobile processes as well: a process can be sent over a network that can connect to other processes, exchange data, and update its resources. This function of the system can be performed using mechanisms to run, reactivate, suspend and send processes over the network included in JCSP package.

5. Results

The simulation system as described above was built using a standard Laptop computer. In this section, we provide results.

5.1 System Performance

We have verified the simulation model by doing some experiments that confirm the correctness of its behaviour.

These experiments are:

- We have used different data streams to explore system performance to see if the stream content makes a difference to the system behaviour.
- Two different instances of the system should show the same functional behaviour if the input data streams are identical but the number and characteristics of processors in the system is changed.
- We have verified that the system output remains unchanged, although the order of communication between processes can vary.

Table 2: capabilities of the cores.

Core number:	Requested type:	Time to process the requested type:
0	0	1
	3	1
	10	1
1	3	1
	4	2
	8	3
	12	2
	16	1
	19	1
2	1	10
	7	10
	13	5
3	5	1
	6	5
	11	3
	14	1
	15	1
	17	2
4	0	10
	1	10
	…	…
	19	10
5	0	3
	9	2
	10	1
	12	1
	13	1
	14	1
6	6	1
	7	2
	8	10
7	0	2
	3	4
	5	1
	6	1
	10	7
	11	9
8	15	1

Some results are shown in tables with a short description of the system state and different data streams used to explore the system's performance. All data sets are represented as follows:

- Number – number of subtask for testing
- Block – data block number
- Task – task number

- A/V – type of subtask, audio or video
- Type – type of requested subtask
- Unit – units of work of requested subtask type

Data used to show the system's performance:

- Expected value
- Actual value

All cores have different capabilities, therefore the system will choose a core with the best suited capabilities for a given subtask type. *Expected value* is evaluated by multiplication of a core's capabilities and units of work needed for the requested subtask type. For evaluating the *Expected value* the core capabilities are listed (Table 2), both *Expected* and *Actual values* are in milliseconds. When a core starts to process the subtask it cannot be interrupted. Both expected and actual values do not take into account the total time that subtasks have been in the system. The time to send a subtask to a different core is excluded.

There are 20 different possible request types. Core 4 can perform all types but at a slow speed and was created to simulate a slow general-purpose processor. Core 8 on the other hand can perform only one task type but is very fast.

Three scenarios are presented to evaluate the system's performance using different sets of data and various numbers of cores running in parallel. The input data is randomly chosen before simulation execution, therefore the system can be run many times with the same blocks of experimental data; requested types are drawn from $\langle 0,19 \rangle$. We define that cores can process only 20 types of tasks. This number is defined for this simulation, but it can be easily modified. The scenarios show how the system works with different sets of cores and various data streams.

5.1.1 Scenario 1

The first Scenario explores the system's performance with a number of data subtasks. The data stream consists of many types of tasks. There are two blocks in the stream: the first consists of 6 subtasks and the second has 8 subtasks. All of the main system processes and all of the cores are running.

Table 3: results of scenario 1.

Number	Block	Task	A/V	Type	Unit	Expected value:	Actual value:
0	0	0	A	0	50	50	50
1	0	0	V	10	100	100	100
2	0	1	A	6	100	100	100
3	0	1	V	19	200	200	200
4	0	2	A	4	70	140	140
5	0	2	V	11	200	600	600
6	1	0	A	9	50	100	100
7	1	0	V	14	100	100	100
8	1	1	A	7	100	100	100
9	1	1	V	19	200	200	200
10	1	2	A	6	70	70	70
11	1	2	V	10	200	200	200
12	1	3	A	1	70	350	350
13	1	3	V	12	200	200	200
				Total:	**1910**	**2510**	**2510**

The results of this simulation are shown in table 3. Table 3 shows that the system reacts as expected; subtasks are allocated to cores with the best capabilities for a particular subtask. It shows that each subtask was processed in the minimum possible time because they were sent to different cores and previously used cores had sufficient time to cool down between uses.

5.1.2 Scenario 2

The second Scenario explores system's performance with the same data subtasks as those of Scenario 1. Only three cores are used in the system. Therefore all of the main system processes are running and only cores 1, 4 and 7 are running.

Table 4: results of scenario 2.

Number	Block	Task	A/V	Type	Unit	Expected value:	Actual value:
0	0	0	A	0	50	100	100
1	0	0	V	10	100	700	700
2	0	1	A	6	100	100	100
3	0	1	V	19	200	200	200
4	0	2	A	4	70	140	140
5	0	2	V	11	200	200	200
6	1	0	A	9	50	500	500
7	1	0	V	14	100	1000	1000
8	1	1	A	7	100	1000	1000
9	1	1	V	19	200	200	200
10	1	2	A	6	70	70	70
11	1	2	V	10	200	1400	1400
12	1	3	A	1	70	700	700
13	1	3	V	12	200	400	400
				Total:	**1910**	**6710**	**6710**

The results of this scenario are shown in Table 4. We observe that the expected values also equal actual values as one would expect for this scenario. The processing time in this case has increased as compared to the scenario 1 in accordance with expectation as lesser number of cores is used to perform the same task. The functional result of the system was verified to be correct.

5.1.3 Scenario 3

This scenario is designed to show how the system will perform when the heating effect of cores is taken into account. In this scenario the data stream is designed in a way that only two cores are used, although all of the available cores are running. This is because only core 2 and 4 can run task type 1 and core 4 can only handle task type 18.

The table of results (Table 5) shows that for subtasks with numbers 0-3 *Expected value* equals *Actual value*. From subtask number 4 the *Actual value* increases. This happens because only three cores are used and they heat up every time a subtask is processed. The heat influences the core's capabilities and the actual value rises. It should be noted that while calculating the *Expected values* in the table, the heating up effect is not taken into account.

This scenario demonstrates how the system reacts on allocating tasks always to the same cores. Allocating tasks only to one core can decrease the system's performance.

Those three test cases were chosen to show how the system works with different sets of cores. In Scenario 1 and 2 data blocks are the same, but the number of cores has decreased. In Scenario 1 total of actual values is 2510 milliseconds where in Scenario 2

equals 6710 milliseconds. This difference is caused by number of cores used. The third Scenario shows how overheated cores decrease the system's performance.

Table 5: results of scenario 3.

Number	Block	Task	A/V	Type	Unit	Expected value:	Actual value:
0	0	0	A	1	20	200	200
1	0	0	V	18	40	400	400
2	0	1	A	1	100	1000	1000
3	0	1	V	18	130	1300	1300
4	0	2	A	1	10	100	110
5	0	2	V	18	50	500	700
6	1	0	A	1	110	1100	2090
7	1	0	V	18	300	3000	8700
8	1	1	A	1	10	100	5100
9	1	1	V	18	20	200	1740
				Total:	**790**	**7900**	**21340**

6. Conclusions

We have presented an architecture to simulate a heterogeneous multi-core media processor. In a multi-core processor the most important aspect is to allocate tasks depending on a core's capability and to run tasks simultaneously. The software was designed to simulate work of many cores in a single processor. Task scheduling and allocation is targeted to efficient use of available heterogeneous cores. The process of allocating tasks was designed and implemented with most of the required functionalities. The system assignment is to receive a data stream, divide it into audio and video streams, process and synchronise both data streams to enable display.

The designed processor consists of cores designed to be separate processes and have different capabilities. Tasks are allocated to cores and run simultaneously achieving faster overall performance. Because of the variety of capabilities some cores are used more frequently to process some tasks. The simulation can model heat generation, its influence on cores capabilities and dissipation of heat. Task allocation is designed to reduce heat generation.

The system is built using CSP principles and consists of processes running in parallel responsible for dealing with data streams, allocating tasks, synchronising data and simulating task execution. A parallel architecture can better reflect the dynamics of the real world, and can be used to model real-time systems. The simulation captures the desired operational characteristics and requirements in terms of the utility of JCSP to model the internal dynamics of a heterogeneous multimedia processor. The JCSP mobile processes were very useful when building concurrent real-time systems. The mobile processes are used to distribute work to cores. Mobility of processes and the ability to run in parallel with other processes are the main capabilities used to build a simulation of the multi-core media processor. The CSP model can also be used to build simulations of other equipment that can help test new ideas and define problems before creating the hardware itself. In particular, the architecture did not involve the use of any buffers between system components. Modern processors often use buffers between system components to improve parallelism. In this design we used the underlying non-buffered CSP communication concept whereby data is held at the output end of a channel until the input end is ready.

7. Future Work

To meet further requirements the final version of the system needs to be extended with additional functionality. The most important system function, that was not included in prototype, is dealing with core failure. Cores can fail in two ways: either stop working or be unable to complete a subtask in time.

Cores can stop working for many reasons. When this happens a core can inform the Controller about its failure. In this case the Controller can stop allocating tasks to this core. The core can be reactivated at any time and send the Controller a signal that it is ready to process tasks. There can be unexpected core failures, where the core will not have time to inform the Controller. To avoid a complete system crash the cores could be scanned periodically. A simple signal may be sent to the core and, if it is still working, in some previously defined time it will respond. This operation can be done by the Controller. For example, every 10 seconds, the Controller can request a response from all of the cores and erase the table responsible for storing cores' capabilities. To ensure the system deals with this kind of problem functions should be added to both Controller and core processes.

In the case where a core cannot complete the task in time, the core should send appropriate information to the Controller. This would also require an additional element in the subtask data structure (Table 1) to include a maximum allowable delay.

In both core failure cases the subtask should be sent back to the Demux to be allocated again. Of course if the subtask allocation is changed the Sync process has to be informed. The Demux process has to send revised information about the sequence in which the Sync process should receive subtasks from cores. To deal with core failure all processes in the system would need to be rewritten.

The system is designed to receive subtasks of a type that can be processed in at least one core. If cores with capabilities suitable for a particular subtask stop working the Controller will not allocate the subtask. This behaviour is recognized by the current version of the system, but functions to deal with this problem are not included. The best way to solve this problem is to make the Demux repeat the request to the Controller about task allocation until some of the cores become available. Of course this loop of requests cannot last forever; there should be other conditions preventing the system from livelock.

In the current version of the system the Channel Decode process initializes the data stream directly with subtasks. This causes confusion and makes testing the system more difficult. In the final system, data should be read from an external file or a channel input. An external file can be for example an XML file with information about blocks of tasks.

One of the system functions that should be also added is to include interconnect latency into the model based upon the size of the subtask. This function would increase simulation accuracy. To make the simulation easier to use there needs to be a user interface added to the system instead of displaying results and process states in a console window.

References

[1] May, D., Processes Architecture for Multicores. Communicating Process Architectures, 2007.
[2] Ramanathan, R.M., Intel® Multi-Core Processors, Making the Move to Quad-Core and Beyond.
[3] Rutten, M.J., et al., Eclipse: Heterogeneous Multiprocessor Architecture for Flexible Media Processing. International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops, 2002.
[4] Stolberg, H.-J., et al., HiBRID-SoC: A Multi-Core System-on-Chip Architecture for Multimedia Signal Processing Applications. Design, Automation and Test in Europe Conference and Exhibition, 2003.
[5] Pericas, M., et al., A Flexible Heterogeneous Multi-Core Architecture. Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, 2007: pp. 13-24.

[6] Kumar, R., et al. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. in 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03) 2003.
[7] Shobaki, M.E., Verification of Embedded Real-Time Systems Using Hardware/Software Co-simulation. IEEE Computer Society, 1998.
[8] Hoare, C.A.R., Communicating Sequential Processes. 1985: Prentice Hall International Series in Computer Science.
[9] Bakkers, A., G. Hilderink, and J. Broenink, A Distributed Real-Time Java System Based on CSP. Architectures, Languages and Techniques, 1999.
[10] NXP, Private communication concerning the project specification. 2007.
[11] Welch, P.H., J.R. Aldous, and J. Foster. CSP Networking for Java (JCSP.net). in International Conference Computational Science - ICCS 2002. 2002. Amsterdam, The Netherlands: Springer Berlin / Heilderberg.
[12] Chalmers, K., J. Kerridge, and I. Romdhani, Mobility in JCSP: New Mobile Channel and Mobile Process Models. Communicating Process Architectures, 2005.