Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

Portable CSP Based Design for Embedded Multi-Core Systems

Bernhard H.C. SPUTH, Oliver FAUST and Alastair R. ALLEN

Department of Engineering, University of Aberdeen,
Aberdeen, AB24 3UE, UK

{b.sputh , o.faust , a.allen}@abdn.ac.uk

Abstract. Modern lifestyle depends on embedded systems. They are everywhere: sometimes they are hidden and at other times they are handled as a fashion accessory. In order to serve us better they have to do more and more tasks at the same time. This calls for sophisticated mechanisms to handle concurrency. In this paper we present CSP (Communicating Sequential Processes) as a method which helps to solve a number of problems of embedded concurrent systems. To be specific, we describe implementations of the commstime benchmark in multithreaded, multiprocessor and architecture fusion systems. An architecture fusion system combines machine and hardware-logic architectures. Our results are twofold. First, architecture fusion systems outperform all the other systems we tested. Second, we implemented all the systems without a change in the design philosophy. The second point is the more important result, because it shows the power of CSP based design methods.

Keywords. Embedded systems, System on chip, Architecture fusion, Multithreaded, Multi-core

Introduction

Embedded systems offer more and more processing power. As the processing power rises there is a need to control or make good use of the new features incorporated into such systems. This paper details different aspects of this rise in processing power. The first aspect is multithreaded programs in embedded systems. To achieve multithreading a supportive OS (Operating System) is required. This OS must be as lightweight as possible. One example of such a lightweight OS is MANTIS, by ShahBhatti et al [1]. It offers multithreading capabilities and requires less then $500\mathrm{kB}$ of RAM. But not only is the sheer multithreading capability important the real time capabilities of the OS must be considered too. Inadequate scheduling and unfavourable resource distribution degrade real time performance [2]. Unfortunately, even with the best possible support from the OS, embedded system engineers find themselves trapped in a jungle of mutexes, semaphores and monitors. To overcome these problems we provide CSP capabilities for the XMK (Xilinx Micro Kernel) [3].

Next we considered embedded multiprocessor systems. Such systems provide the opportunity to create very powerful and energy efficient processing machines [4]. But, of course there is the danger of producing under achieving systems if the process distribution is not considered carefully enough. The desire for multi-core systems has yielded a whole range of different inter-processor communication methods [5,6]. All these methods have their advantages and disadvantages, but the main problem for all of them is that they lack a consistent mathematical basis which describes the communication.

The fusion between machine and hardware-logic architectures unleashes the real power of FPGA (Field Programmable Gate Arrays). Low power consumption and physical size are

key targets of embedded system design. The power consumption of a clocked IC (Integrated Circuit) depends quadratically on the clockspeed. Therefore, an embedded systems designer tries to reduce the clockspeed to a minimum. The use of ASICs (Application Specific Integrated Circuits) allows one to move functionality from software to hardware: this is one way to lower the clockspeed of the system. The drawback of the use of ASICs is that the system has now a larger physical size, caused by the use of extra components. This brings us to the last aspect of this paper: the fusion of machine and hardware-logic architectures. One example of such fusion between machine and hardware-logic architectures was presented by B.C. O'Neill et al. in the form of a SOPC (System on a Programmable Chip)[7]. Watt et al. have presented a programming language for such hybrid systems consisting of a standard PC and an FPGA [8]. Machine architectures, such as Harvard and Von Neumann, are very flexible, because their functionality entirely depends on the instruction addressed by the program counter. In other words, the functionality of a machine architecture is altered by jumping to new code segments. This great flexibility requires strictly sequential processing. This is the reason why machine architectures are slow compared with hardware-logic. The fusion between machine and hardware-logic architectures requires some sort of communication between them. Traditionally, multi-processor systems consist of multiple interconnected CPUs of the same type. In our system this is not the case because it combines flexible but slow machine architectures with inflexible but fast hardware-logic architectures. As for the previous cases, engineers came up with various solutions to tackle these problems [9]. But again an overall strategy is missing.

All the aspects of multithreading, multiprocessors and architecture fusion boil down to communication problems in embedded systems. In all cases we have independent entities which must communicate with each other. This paper describes how we applied methods based on CSP [10,11] to solve this communication problem in a consistent way. This consistency allows developers to concentrate on what they want to achieve, in a concurrent fashion, without worrying about how their concurrent processes communicate. This is because the designer of the CSP back-end has already worried about the communication. We demonstrate how easy it is to break up a problem into smaller communicating parts which are implemented in different architectures.

Applying CSP principles to develop FPGA implementations is by no means a new idea. In fact there has been a lot of work done in the past. First to mention here is Handel-C [12], a programming language inspired by CSP. Cook et al. developed a compiler translating occam code into a hardware-logic-core for an FPGA [13]. HCSP [14] is an extension of CSP with priority and true concurrency (multiple events happening at once) to allow one to prove the correctness of a hardware implementation.

We have chosen the commstime benchmark [15] as example, because the resulting systems are simple and yield an agreed upon benchmark figure. The next section provides a brief introduction to the commstime benchmark. Section 2 introduces the libcsp [16] port to XMK. This extension allows us to create multithreaded programs that communicate and synchronise in a well understood manner. Section 3 describes the hardware system used for the benchmark tests. The following subsections describe the individual implementations going from single core to multi-core and to a fusion of machine and hardware-logic architecture. Section 4 contains a discussion of the benchmark results. Sections 5 and 6 are conclusions and further work.

1. Commstime Benchmark

The commstime benchmark is a widely applied method [15,17,18,19,20,21] to measure the overhead of channel communication between parallel processes. This allows us to compare

different CSP environment implementations at one of their core activities: channel communication. Another reason for choosing commstime as example is that it consists of four simple processes, which communicate over channels. This allows us to distribute them over multiple processor cores and even to implement one of the processes as hardware-logic. In this section we give a brief introduction to the commstime benchmark.

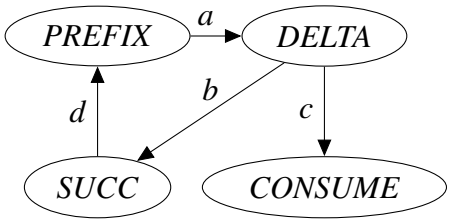

Figure 1. Process network of the commstime benchmark

The process *COMMSTIME* (Equation 1) is the top-level process of the benchmark. The diagram of Figure 1 illustrates the different processes and their connections[1]. The commstime benchmark consists of a ring of three processes: *PREFIX*, *DELTA* and *SUCC* (Equations 2 – 5). The *DELTA* process connects this ring to the process *CONSUME* (Equation 6), which performs the measurements. In our implementation *DELTA* behaves as sequential delta, thus sending out the messages sequentially, instead of concurrently. We reflect this in the model we show here.

$$COMMSTIME = PREFIX(0, d, a) \;\Big|\Big|\; DELTA(a, b, c) \;\Big|\Big|\; SUCC(b, d) \;\Big|\Big|\; CONSUME(c) \quad (1)$$

with:

$$PREFIX(value : \mathbb{Z}, in, out) = out!value \rightarrow COPY(in, out) \quad (2)$$

$$COPY(in, out) = in x : \mathbb{Z} \rightarrow out!x \rightarrow COPY(in, out) \quad (3)$$

$$DELTA(in, out1, out2) = in?x : \mathbb{Z} \rightarrow out1!x \rightarrow out2!x \rightarrow DELTA(in, out1, out2) \quad (4)$$

$$SUCC(in, out) = in x : \mathbb{Z} \rightarrow out!(x + 1) \rightarrow SUCC(in, out) \quad (5)$$

$$CONSUME(in) = c?x : \mathbb{Z} \rightarrow CONSUME(in) \quad (6)$$

The process *PREFIX* starts the communications on the ring by sending the initial integer value: *value*, to *DELTA*. This process then sends the value to *SUCC* and to *CONSUME*. *SUCC* increments the value by one and then sends it to *PREFIX*, which now behaves like *COPY* (Equation 3) and sends the value to *DELTA*. This ring operates as long as *DELTA* is able to send the value over its second channel to *CONSUME*. Thus for each round the message makes in the ring, *CONSUME* receives one message. This allows *CONSUME* to measure how much time passes between receiving two messages. Because each round requires four channel communications it is possible to determine how long an individual channel transaction takes.

[1]Please note that we omit the parameter list when referring to the processes of the commstime benchmark. Furthermore, when we omit the alphabets of the parallel operator ($||$), we imply that the processes bring their whole alphabet.

2. A CSP Back-end for XMK

To ease the development of multithreaded software for Xilinx MicroBlaze [22] based SOPCs in embedded systems we decided on extending XMK (Xilinx Microkernel) [3] with CSP capabilities. The Xilinx MicroBlaze is a configurable SoftCPU for Xilinx FPGAs. This allows one to tailor the CPU to ones needs by omitting unnecessary units: for instance, a floating point unit. Furthermore, the Xilinx MicroBlaze can be combined with hardware-logic to form architecture fusion systems like the one we present in this paper.

One of the options for this extension was an occam based approach, using either KRoC [15] or SPoC [23]. But we decided against the use of occam because this would have meant to extend the tool chain of either KRoC or SPoC to the MicroBlaze instruction set. Thus the resulting system would have been processor specific instead of OS specific.

After deciding against occam we considered C, C++, or Java based CSP environments. Unfortunately there seems to be no JVM (Java Virtual Machine) available for the MicroBlaze processor, which prevented the use of JCSP [24] or CTJ [25]. Xilinx provides tool chains only for C and C++, for use with the MicroBlaze processor. This limits our options to: C++CSP [19], CCSP [17] and libcsp [16]. Unfortunately, the C++ tool chain does not implement all features of the C++ standard. The provided C++ compiler failed to compile C++CSP, — it stumbled across some template definitions in C++CSP. Looking at CCSP and libcsp we decided to go for libcsp — it requires less memory during runtime and is easier to port than CCSP.

Libcsp is a library that provides CSP style communication and synchronisation mechanisms for C programs. It uses the pthreads [26] API (Application Programmer Interface), to create and synchronise threads. The pthreads API is part of the POSIX (Portable Operating System Interface) standard, to which many available OS conform. This practical OS independence allows developers to utilise libcsp whenever they want to use CSP in a project. This section details the problems faced during the porting of libcsp to XMK.

XMK is a small OS for the Xilinx MicroBlaze SoftCPU and FPGA internal PowerPC 405. It provides a subset of the POSIX API, which includes an incomplete implementation of pthreads.

Unfortunately, this pthreads API subset does not include an implementation of conditional variables, a monitor like synchronisation mechanism, on which the libcsp channel implementation depends. Therefore, we extended the pthreads API of XMK appropriately. This allowed us to avoid any modification of the libcsp channel implementation.

Another problem is that XMK has only mediocre support for dynamic memory management. To overcome this, we preallocate memory heaps at compile time, using arrays, and manually manage these heaps at runtime. These heaps allow us to determine at compile time whether or not a program fits into the available memory. This effectively prevents the occurrence of "out of memory errors" at runtime. However, the tradeoff is that one needs to know the required heap size during compilation.

For instance, libcsp uses dynamic memory management to allocate channels and processes. We adjusted these functions to use preallocated heaps of the corresponding data structures. Unfortunately, this required extended modifications of the libcsp API, which in fact is close to the API of the original INMOS C compiler, making the libcsp for XMK incompatible with its origin. To make this clear we may decide to change the name in the future. However, the core of the library is still the same and we made sure that the modified version of libcsp still compiles on OS other than XMK.

3. Implementations

This section describes five different implementations of the commstime benchmark. These implementations reflect different types of embedded systems designs.

All implementations were done using a Xilinx Virtex-II FPGA (Model: XC2V3000 [27]), with 3M system gates and a speed grade of -4, which is the slowest available for the Virtex-II family. For the commstime example this chip hosts three MicroBlaze processor cores (MB0, MB1 and MB2) and one *SUCC* hardware-logic core.

The individual components are connected by FSLs (Fast Simplex Links) [28]. An FSL (Fast Simplex Link) is a low latency, unidirectional point-to-point data streaming interface. The FSLs supported by the MicroBlaze are up to 32 bits wide and offer two modes of operation: blocking and non-blocking. FSLs always contain a FIFO (First In First Out) buffer. This buffer holds between one and 8192 messages, the number being determined before generating the configuration bitstream for the FPGA. Therefore, only the required amount of buffer is allocated on the FPGA. This together with their ability to block the processor from proceeding when the buffer is not ready, makes fast simplex links in fact buffered-channels. Two fast simplex links of opposite directions form one FDL (Fast Duplex Link).

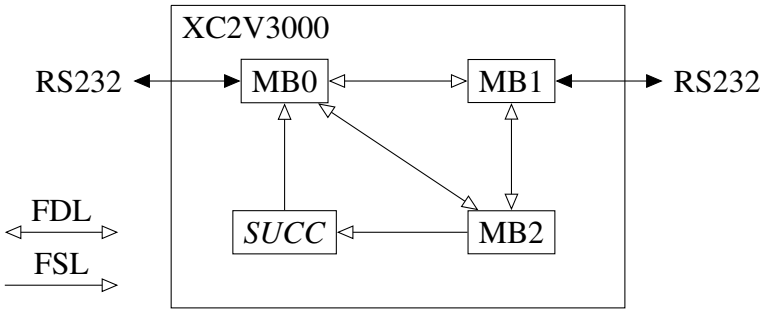

Figure 2. Top-level view of our SOPC, with the interconnects between the different processing entities

Figure 2 shows the simplified block diagram of the system. Each MicroBlaze processor core has access to 64kB of RAM and is clocked at 50MHz. Each MicroBlaze connects over the OPB (On-chip Peripheral Bus) [22] to its own timer. Once started, the timer counts the number of clock impulses (50MHz) in a 32-bit register. We use this timer in *CONSUME* to measure the time required to execute a certain number of loops. Figure 3 shows the setup of MB0 which is identical to MB1, omitting any FSL. Both connect to an RS232 interface via their OPB interface. This RS232 interface can be used to establish a serial link with an external terminal. MB2 is not connected to an RS232 interface.

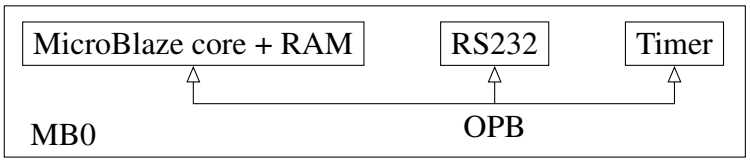

Figure 3. Individual Processor setup for MB0

3.1. Single-core Multithreaded

In the first test a single CPU, MB0, executed the commstime benchmark. Native XMK threads represented all processes of the commstime benchmark. The processes exchanged messages over libcsp channels. Figure 4 illustrates the process distribution. This implementation was straightforward, using the commstime benchmark as provided by libcsp, with small adjustments to accommodate the modified API. For $100,000$ loops the system required 36.694s.

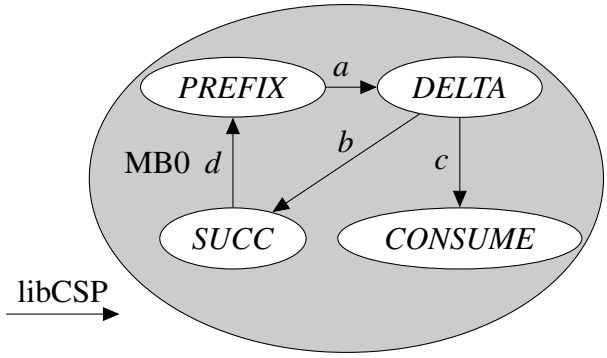

Figure 4. Process distribution over a single core system

3.2. Dual-core

As the name suggests dual-core implementation utilises two CPU cores: MB0 and MB1. Both processors run XMK and libcsp. The commstime benchmark is split, such that the processes *PREFIX* and *SUCC* run in MB0 whereas the processes *DELTA* and *CONSUME* are executed in MB1. MB0 and MB1 communicate via two FSLs. These links represent channels *a* and *b*. Figure 5 illustrates the process distribution. With this setup, the commstime benchmark performs $100,000$ loops in 10.106s, almost four times faster than the single-core implementation. In Section 4 we explain why this implemenation is almost four times faster than the single-core implementation.

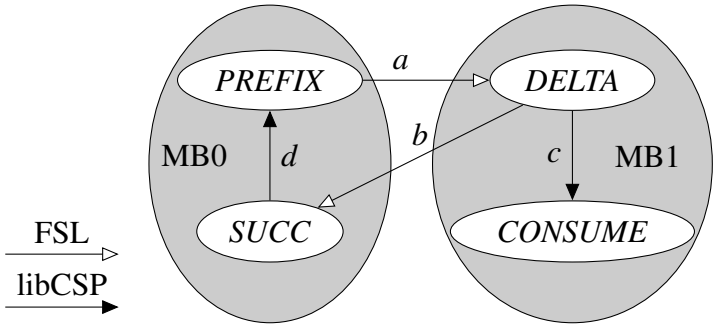

Figure 5. Process distribution over a dual-core system

3.3. Triple-core

With the triple core setup we studied the effects of process distribution on the benchmark results.

The first triple-core example utilises all three MicroBlazes. They all run XMK and libcsp. The processes are distributed as follows: *PREFIX* on MB0, *CONSUME* on MB1 and *DELTA* and *SUCC* on MB2. FSLs now provide the channels *a*, *d* and *c*. Figure 6 shows the process distribution. Commstime requires 10.114s to perform the $100,000$ loops, a slight performance degradation.

The first triple-core implementation did not show any speedup compared to the dual-core implementation. In fact it was even a little bit slower. Why is this? Because there is still a relatively slow libcsp channel between *DELTA* and *SUCC*. The slow communication of the libcsp channel is caused by the fact that it has to perform context switches. Context switches require a lot of computing resources. In this position the libcsp channel is part of a sequence of channel transactions on the channels *a*, *b* and *d*. Channel *c* connecting *DELTA* and *CONSUME* is used concurrently with the channels of the ring. Therefore, when placing

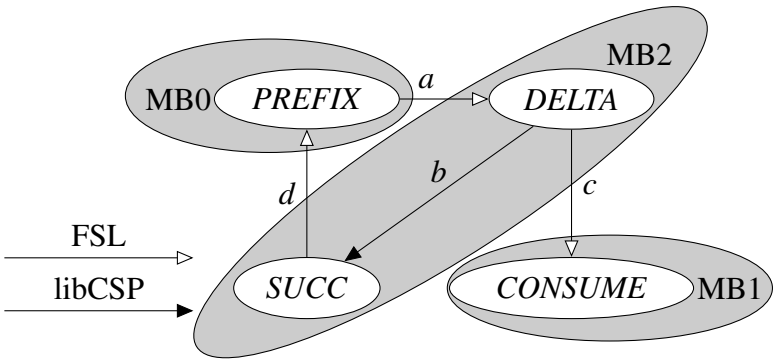

Figure 6. Process distribution I over a triple-core system

all fast channels (FSLs) in the ring and the slow libcsp channel concurrently with them, the performance should increase.

This is exactly what we have done in the second triple-core example, Figure 7 shows the process distribution. The total runtime for this setup was 10.106s, only a slight increase in performance. However, it is still not faster than the dual-core implementation.

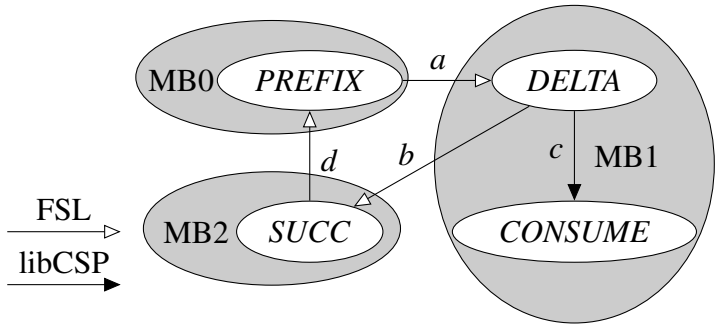

Figure 7. Process distribution II over a triple-core system

3.4. Architecture Fusion

To evaluate the performance of the FSLs and to determine whether the libcsp channel is really to blame we implemented the *SUCC* core. The *SUCC* core connects over two FSLs to MB2 and MB0. In this setup (Figure 8) each MicroBlaze executes only one process, avoiding context switches. Thus we have now a system with hardware concurrency, instead of software simulated concurrency. The runtime of the commstime benchmark, for $100,000$ loops now drops to 0.043s. At a first glance it may seem that this speedup is caused by the use of the dedicated hardware: however, this is only half the story. The *SUCC* core can only pass on messages as fast as it receives them, because it is only recycling messages, not generating new ones. That means that the other participating processors need to do their share as well for the commstime benchmark to execute. Therefore, this measurement result indicates that the context switching required by the libcsp channel is responsible for preventing the triple-core system from being faster than the dual-core system.

4. Measurement Results and Discussion

Table 1 gives the measured results for the different implementations of the commstime benchmark. To determine how much faster the multi-core implementation is compared to the single-core implementation, we calculated the speedup (S). Equation 7 gives the definition

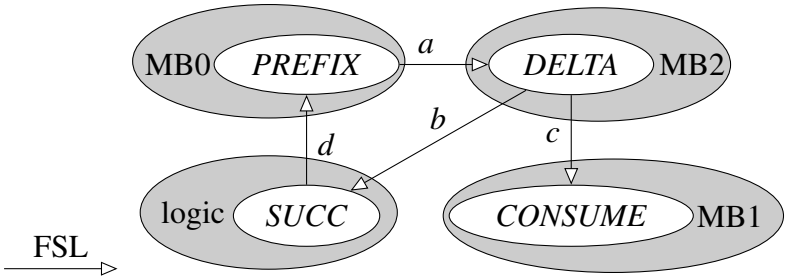

Figure 8. Communication network of the commstime benchmark

Implementation	Cores	Channel cost	Speedup (S)
Single-core	1	91.74µs	1
Dual-core	2	25.27µs	3.63
Triple-core I	3	25.28µs	3.63
Triple-core II	3	25.27µs	3.63
Architecture Fusion	4	0.11µs	859

Table 1. Measurement results

CSP Environment	Channel cost (µs)
CCSP (GNU C) [17]	0.68
CCSP (occam)[17]	0.41
JCSP [21]	23
tranx86 [18]	0.067
C++CSP (own scheduler) [19]	1.25
C++CSP (GNU `pth` scheduler) [19]	15

Table 2. Channel cost of other CSP environments

of the speedup. *SCPT* is the time the single-core implementation requires for the task and *MCPT* is the time the multi-core implementation requires.

$$S = \frac{SCPT}{MCPT} \tag{7}$$

The speedup of an algorithm when executing should not exceed the number of available processors, assuming that the processors are uniform. However, the speedup of the dual-core implementation is 3.63 which is almost double the amount of used processors. The reason for this phenomenon is that in the dual-core implementation two software channels have been substituted by two hardware channels (FSLs). These hardware channels perform the handshaking between sender and receiver not in software but in hardware, this explains this vast speedup. In the case of the two triple-core implementations we experience no speedup compared to the dual-core implementation. The reason for this is that there is still one software channel in use, which is the weakest link of the process chain and therefore determines its speed. The architecture fusion implementation removes this last software channel and thus allows the system to demonstrate its raw communication speed. This results in a speedup of 859. However, this speedup comes with a high price: the design now contains a hard to change hardware-logic component and uses the largest silicon area.

Table 2 is a collection of published channel communication cost acquired using the commstime benchmark. Please note that these results were obtained using different test systems. There are three different groups to differentiate here: implementations operating in a virtual machine (JCSP), implementations that have adjustments to a specific OS (C++CSP)

and implementations that contain optimisations to the OS and the target CPU (tranx86, CCSP). The more dependent an implementation becomes towards a specific OS or CPU the faster it becomes. This is of course not a new discovery, however it is nice to see that it holds also for these implementations. In Table 1 we experience the same effect, the single-core implementation is portable among different CPU and OS, it just needs a recompile. Adjustments made to the available hardware in the dual- and triple-core implementations lead to an increased performance. However, this implementation is still portable onto other SOPCs which offer two or three interconnected MicroBlaze processor cores. Finally, the implementation with a hardware *SUCC* process is a few magnitudes faster than the other implementations. However, this comes at high cost — being practically non-portable.

In [13] Cook et al. present a pure hardware-logic implementation of the commstime benchmark. Similarly to ours it is clocked at 50.4MHz and requires 15ns for one channel transaction, more than 7 times faster than our architecture fusion system. This solution [13] is optimised down to the gate level, by the use of a special occam-to-FPGA compiler, making the incorporation of existing hardware-logic-cores difficult.

Embedded systems, especially when battery operated, should consume as little power as possible. Our system consumes 0.660W when the FPGA is unconfigured. Configured with the architecture fusion bitstream and executing it, the system draws 1.485W. As comparison, tranx86 [18], the fastest system as of Table 2, uses an Intel Pentium III clocked at 800MHz. According to the Pentium III datasheet [29], the core of this CPU consumes maximal 27.2W[2]. We assume that the power consumption is close to the maximum rating. Because the commstime benchmark exercises parts of the CPU to the limit. Even if the CPU actually consumes less power, this figure excludes the power drawn by other vital components, such as chipset and RAM. To be fair their power consumption should be included as well. However, even without them it is obvious that our system consumes less energy.

5. Conclusions

In this paper we described the implementation of the commstime benchmark in three different embedded system setups. The different setups were realised in the same FPGA chip with the same clock frequency. This makes the benchmark results comparable. The first scenario was a single MicroBlaze soft-processor running the XMK operating system, libcsp extension and the complete commstime benchmark. This was the slowest system in the test. The multi-processor system performs better then the single-core system, however there is the additional burden of process distribution. That process distribution matters was shown with the triple-core example. Different process distributions yield different benchmark values. The fastest implementation was the architecture fusion system: there each processor had to only execute one process, thus avoiding slow context switches.

The main result presented in this paper is the sheer ease with which we moved from one implementation to the other. Basically, the CSP style communication and synchronisation allows us to overcome the traditional design borders. Effortlessly, we moved processes from one processor to another and from machine architecture to hardware-logic architecture. This is a powerful design method for embedded systems, because it helps to utilise the capabilities of different processing architectures. This ability of interlinking different architectures allows developers to utilise the architecture that is best suited to deal with their problems. This free architecture choice, within the SOPC, allows the designer to create power efficient systems, that furthermore require little physical space. The experiments show that it does not require

[2]Intel publishes a $V_{cc_{core}}$ of 1.7V and an $I_{cc_{core}}$ of 16.0A for a Pentium III with 800MHz and a CPUID of $0x686$.

much effort to run an operating system on a soft processor alongside hardware or hardware-logic components.

6. Further Work

Distributed sensor networks consist of a large number of sensor nodes. Depending on their deployment, these nodes may be battery operated systems. This places a high emphasis on minimising the energy consumption of a node. A sensor network node has to provide high processing capabilities for source and channel coding as well as signal modulation. This processing power can be achieved with architecture fusion systems. Reconfigurable hardware, such as FPGAs, allow us to move processes almost effortlessly from software to hardware-logic and back. So we can control all relevant parameters such as chip utilisation, processing time and energy consumption.

Naturally, also the inter node communications could be modelled using CSP. However, the links between sensor nodes may be unreliable. This has two causes. First of all, the environment in which the nodes are deployed may block certain links. Secondly, the sensor nodes themselves are unreliable and may cease to function. In [30] we present a simple protocol that allows one to securely broadcast a message to all sensor nodes of a cluster[3]. However, this is only the beginning of our investigation.

The current port of libcsp is still work in progress. This means it is in a state where it works, but is not pretty. There are a couple of areas where the current version of libcsp lacks support for specialities of the MicroBlaze environment. One of these areas is the integration of the FSL. At present we use these channels directly and are therefore unable to alternate over software channels and FSLs. By abstracting the FSL and using them like the transputer used its links it would be possible to create an even easier environment to develop in. The architecture fusion implementation demonstrated that an FSL provides a large enough bandwidth to multiplex a large number of channels over it, without making the FSL a bottle neck.

Another worthwhile direction of research is to develop a unified language to describe the hardware and software, similar to the work presented the Watt et al. in [8]. However, we strongly suggest to allow the use of existing hardware-logic-cores from within this language, to allow developers easy integration of already existing hardware-logic-cores.

A big problem still open is that the current implementation assumes that the developers use the provided CSP primitives correctly. Libcsp does not check whether the CSP primitives are used correctly. One possibility to overcome this is to extract the use of the CSP primitives from the C source code and translate it to CSP_M script. This CSP_M script model of the communication could then be checked using FDR [31]. However, this would also require a complete CSP_M script model of the C environment. Work in this area (Java to CSP_M script) has been presented by Hui Shi in form of Java2CSP [32].

References

[1] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *Mobile Networks and Applications*, 10(4):563–579, 2005.

[2] J. Kreuzinger, A. Schulz, M. Pfeffer, T. Ungerer, U. Brinkschulte, and C. Krakowski. Real-time scheduling on multithreaded processors. In *Seventh International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*, page 155ff, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[3] Xilinx, Inc, 2100 Logic Drive San Jose, California 95124 United States of America. *OS and Libraries Document Collection*, 24 October 2005.

[3] A cluster means here all reachable sensor nodes

[4] Marco Bekooij, Orlando Moreira, Peter Poplavko, Bart Mesman, Milan Pastrnak, and Jef van Meerbergen. Predictable embedded multiprocessor system design. *Lecture Notes in Computer Science*, 3199:77–91, January 2004.

[5] Mohamed Shalan and III Vincent J. Mooney. Hardware support for real-time embedded multiprocessor system-on-a-chip memory management. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 79–84, New York, NY, USA, 2002. ACM Press.

[6] Joann M. Paul, Donald E. Thomas, and Andrew S. Cassidy. High-level modeling and simulation of single-chip programmable heterogeneous multiprocessors. *ACM Trans. Des. Autom. Electron. Syst.*, 10(3):431–461, 2005.

[7] Brian C. O'Neill, P.W. Moore, and S. Clark. A Single Chip Solution for Distributed Processing Systems. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 83–90, September 2003.

[8] D. R. Watt and David May. A Programming Language for Hardware/Software Co-Design. In Alan G. Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, pages 167–178, 2001.

[9] David Andrews, Douglas Niehaus, Razali Jidin, Michael Finley, Wesley Peck, Michael Frisbie, Jorge Ortiz, Ed Komp, and Peter Ashenden. Programming models for hybrid FPGA-CPU computational components: a missing link. *IEEE Micro*, 24(4):42–53, 2004.

[10] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978.

[11] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, Upper Saddle River, New Jersey 07485 United States of America, first edition, 1997.

[12] Celoxica Ltd, 66 Milton Park Abingdon Oxfordshire OX14 4RX United Kingdom. *DK3 Handel-C Language Reference Manual*, 2004.

[13] Barry M. Cook and Roger M. A. Peel. Occam on Field Programmable Gate Arrays - Steps towards the Para-PC. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 211–228, 1999.

[14] Adrian E. Lawrence. HCSP: Imperative State and True Concurrency. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 39–56, 2002.

[15] David C. Wood and Peter H. Welch. The Kent Retargettable occam Compiler. In Brian C. O'Neill, editor, *Proceedings of WoTUG-19: Parallel Processing Developments*, pages 143–166, March 1996.

[16] Rick D. Beton. libcsp - a Building mechanism for CSP Communication and Synchronisation in Multithreaded C Programs. In Peter H. Welch and André W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 239–250, February 2000.

[17] James Moores. CCSP - A Portable CSP-Based Run-Time System Supporting C and occam. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 147–169, March 1999.

[18] Frederick R. M. Barnes. tranx86 – An Optimising ETC to IA32 Translator. In Alan G. Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, pages 265–282, September 2001.

[19] Neil C. Brown and Peter H. Welch. An Introduction to the Kent C++CSP Library. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, September 2003.

[20] Jan F. Broenink, Marcel A. Groothuis, and Geert K. Liet. gCSP occam Code Generation for RMoX. In *Communicating Process Architectures 2005*, pages 375–383, September 2005.

[21] Nan C. Schaller, Gerald H. Hilderink, and Peter H. Welch. Using Java for Parallel Computing - JCSP versus CTJ. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 205–226, September 2000.

[22] Xilinx, Inc., 2100 Logic Drive San Jose, California 95124 United States of America. *MicroBlaze Processor Reference Guide*, 21 February 2006.

[23] Denis A. Nicole, Sam Ellis, and Simon Hancock. occam for reliable embedded systems: lightweight runtime and model checking. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 167–172, September 2003.

[24] P. H. Welch and P. D. Austin. JCSP home page http://www.cs.kent.ac.uk/projects/ofa/jcsp/.

[25] Jan F. Broenink, André W. P. Bakkers, and Gerald H. Hilderink. Communicating Threads for Java. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–262, September 1999.

[26] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, first edition, 1998. ISBN: 1-56592-115-1.

[27] Xilinx, Inc., 2100 Logic Drive San Jose, California 95124 United States of America. *Virtex-II Platform*

FPGAs: Complete Data Sheet, v3.4 edition, 1 March 2005.

[28] Xilinx, Inc., 2100 Logic Drive San Jose, California 95124 United States of America. *Fast Simplex Link (FSL) Bus (v2.00a)*, 1 December 2005.

[29] Intel Corporation, 2200 Mission College Blvd. Santa Clara, CA 95052 USA. *Pentium III Processor for the PGA370 Socket at 500 MHz to 1.13 GHz Datasheet*, 8 edition, June 2001.

[30] Oliver Faust, Bernhard H.C. Sputh, and Alastair R. Allen. A study of percolation phenomena in process networks. In Peter Welch, Jon Kerridge, and Fred Barnes, editors, *Communicating Process Architectures 2006*, September 2006.

[31] Formal Systems (Europe) Ltd., 26 Temple Street, Oxford OX4 1JS England. *Failures-Divergence Refinement: FDR Manual*, 1997.

[32] Hui Shi. Java2CSP: A System for Verifying Concurrent Java Programs. In G. Schellhorn, W. Reif, editor, *FM-TOOLS 2000*, number 2000-07 in Ulmer Informatik-Berichte, pages 111 – 115, 2000.